A Framework for modelling communication hardware in multicomputers

Pauline Catriona Haddow

Department of Computer and Information Science
The Norwegian University of Science and Technology

29 May 1998

Abstract

The main objective of the work presented herein, is to present and evaluate a new approach to modelling communication hardware in multicomputers.

There has been a lot of research within the field of performance analysis of communication systems in parallel computers, especially at the hardware level. Using traditional modelling techniques, assumptions are made to limit the complexity of the model to that which meets the model goals. With so many features that may be traded off against each other, a similar range of assumptions is found. Their use does not lessen the validity of the results obtained, but makes comparison of results a somewhat difficult task. It also prevents reuse of existing models for purposes other than those for which the model was designed. As such, a common evaluation platform is required i.e. a modelling framework.

A feature, which is not generally taken into account in performance analysis of communication hardware, is cost. That is, what effect do performance improvements have on the complexity and thus the cost of the network routers.

The goal of the framework is to enable a wide range of communication hardware in multicomputers to be modelled in sufficient detail such that performance and cost analysis is extended to the gate level. A further objective is for the modelling environment to be flexible with respect to changing design ideas and technologies. That is to support exploration of the communication hardware design space.

This thesis presents a prototype of the framework, its structure and implementation. Implementation has reached the stage where the viability of the approach may be analysed. Case studies illustrate a number of features of the framework.

Preface

This is a doctoral thesis submitted to the Norwegian University of Science and Technology. The research work presented in this thesis was carried out in part at the Computing Science Department, University of Glasgow, Scotland and the remainder at the Norwegian University of Science and Technology. The principle supervisor in this work has been Professor Lasse Natvig.

Acknowledgements

First and foremost I would like to thank my supervisor Professor Lasse Natvig for his support and advice and especially his unending determination to find time, even in the most hectic periods, to support my work. Also the members of the Computer Architecture and Design group and in particular Jan Anders Mathisen who has provided helpful technical advice in this work. Norun Brembo who is always there to sort out administrative problems and provide helpful advice.

I would also like to thank my supervisor at the University of Glasgow, Dr. Lewis Mackenzie, who introduced me to this field of research and supported me through the early stages of this work.

I would like to express my thanks to the Science and Engineering Research Council of Great Britain and the Norwegian Research Council and their cultural exchange program for supporting the earlier stages of this work. Svein Johan Knapskog, Professor Kjell Bratbergsengen and the Department of Computer and Information Science for funding support in the later stages of this work.

Part of this work has involved a newer language under development at the University of Cincinnati. I would therefore like to thank Jeffrey Walrath for his prompt assistance in answering questions raised during the implementation phase of this work.

There have been many helpers that have contributed to the final thesis. Donald Brown, who was unlucky enough to arrive in Norway in time to proof read this thesis; Professor Olav Landsverk and Dr. Gaute Myklebust whose expertise in architecture provided me with very helpful comments and advice and lastly but not least Dr. Mohammed Ould-Khoua whose expertise within parallel computers has provided invaluable support and comments both in the final draft of this thesis and in much of the earlier work.

I would like to thank my family for their unending support and encouragement in my numerous adventures, even when I moved away to the frozen north. My thanks also go to my friends here in Norway who have made me feel welcome and at home in Norway.

Finally, I would like to thank Torfinn for his support and unending optimism and cheerfulness. His repeated phrase "It's fun and challenging" has helped me through the last few months of this work.

Trondheim, 29 May 1998

Pauline Catriona Haddow

Contents

List of Figures

List of Tables

Part I

Communication Hardware Design and Analysis

Chapter 1

Introduction

The main aim of the work presented in this thesis is to develop a modelling platform for realistic cost and performance analysis of communication hardware in multicomputers. This involves two competing goals: breadth against depth.

Design of communication hardware involves a trade off between a variety of design issues, some of which are interdependent. This has led to the evolution of a large variety of communication hardware both in existing and proposed parallel computers. To be able to model this range of systems this fact places a breadth requirement on the modelling platform.

The complexity of a communication hardware model depends on the requirements of the model i.e. to offer system-level, register-transfer level or gate-level analysis. However, to offer cost analysis, gate-level modelling is required. As such, it increases the minimum complexity requirement of a model or in other words increases the depth of the model.

A further aim is to offer a modelling environment not only for todays parallel computers but those of the future. This places a requirement of flexibility on the modelling platform to cope with newer designs. As cost is implementation dependent then the implementation technology is an issue and flexibility with respect to evolving technologies is also required.

In this work a flexible modelling framework, designed to meet these goals, is presented.

The aim of this report is to present the concept and illustrate the usefulness of the framework. At present, a number of desirable features have not been fully implemented and as such analysis is currently restricted to that of latency and cost.

1.1 Performance Modelling Requirements in Communication System Analysis

There has been a lot of research within the field of performance analysis of communication systems in parallel computers especially at the hardware level. However, most of this research focuses on solving the stated problem. That is, assumptions are made such that the required problem is focused on and for which results are obtained. With so many parallel computer features that may be traded off against each other, different assumptions may be made for different purposes. This does not lessen the validity of the results obtained, but makes comparison of different results very difficult. It also prevents reuse of existing models for different purposes. As such, a more general platform is desirable for modelling parallel computers such that results may be traded off against each other on the same evaluation platform i.e. a modelling framework.

During the creation of such a framework, it is important to consider the requirements for performance modelling such that the framework may be established with these goals in mind. The remainder of this section describes some of the features of communication systems and the requirements that these features place on a realistic modelling framework for communication systems.

- In parallel computers, much research has been dedicated to improving throughput and reducing latency in interconnection networks. Recent research has led to increased throughput but more complex routers with the addition of virtual channels [DS87] or the use of various new adaptive routing schemes [Dua91b]. However, it is acknowledged that software is a major bottleneck in communication systems. Especially for small messages, as is the case in fine-grained processing, the software overhead constitutes a large percentage of message latency [HB90a].

Hardware level improvements such as the implementation of advanced routing schemes, as discussed above, give rise to increased latency at the software level [KC94] and therefore increase the software bottleneck.

Research into reducing the delay incurred at the software level has focused on three main areas. One field of thought is to move some of the software functionality into hardware e.g. compressionless routing [KLC94]. Another, is to improve the interface to the hardware i.e. the Shrimp project [Bea95], Active Messages [VEea92] and the J-Machine [Dea89]. These solutions lead to increased hardware complexity and, as such, give rise to an increase in latency at the hardware level. Another solution, as seen in the Intel Paragon [Int91], is the addition of a co-processor to handle communication operations. This enables an increase in the overlap of computation to communication but with an increase in hardware costs.

It can be seen from the above discussion that there exists software versus hardware and throughput versus latency trade-offs in communication systems. Therefore, it is desirable to be able to describe and evaluate the entire communication system i.e. at both the software and hardware level, with respect to the performance metrics of throughput and latency.

- Another factor which has a large effect on the performance of message-passing multicomputers is the communication patterns and message characteristics of parallel algorithms [HB90a]. These characteristics are represented in the interconnection network as network traffic. Assumptions such as Poisson message arrival, exponential distribution of message length and even distribution of message destination [KK79] are often used to describe network traffic for performance analysis purposes. Other assumptions are based on task granularity and inter-task dependencies. However, it has been shown by Hsu and Banerjee [HB90a], through experimentation on the hypercube topology, that many of these assumptions do not match the characteristics of real applications running on parallel computers. As such, a more accurate description of traffic is required for realistic performance analysis. Realistic traffic stimulus may be obtained using event traces from parallel programs run on real parallel computers or from parallel architecture simulators aimed at generating trace programs for a given architecture.

- Traditionally, performance analysis at the network level has been based on cycle counting. However, direct comparison of the results of various studies cannot be made as the network clock cycles differ between featured parallel computers. It has also been shown by Patel and Jayasimha [PJ95] that cycle counts can give misleading results. A more accurate method is to base results on time units – 'wall clock' time.

- It has been the custom in interconnection network performance analysis to assume a fixed delay through each of the routers independent of their complexity [Aga91]. However, it is shown by Chien [Chi93] that router complexity has a significant effect on the amount of delay through the router.

- Another feature, which is not generally taken into account in performance analysis, is that of cost. That is, what effect do performance improvements have on the cost of the system? The importance of cost in performance analysis of hardware routers was introduced by Chien [CK92].

1.2 Contributions of this Work

As stated in section 1.1 it is desirable for the parallel community to be able to model the entire communication system. That is at both the hardware and software levels. However, this is a phenomenal task considering the range of features at the software level i.e. operating system, message passing libraries and processor effects, and those at the hardware level i.e. topology choice, router design and technology considerations. As such, it was chosen to limit this work to the hardware level of the communication system such that priority could be given to a more detailed investigation and creation of a flexible modelling environment. This enables a range of designs to be modelled and analysed at an increased level of detail over that which most network modelling environments currently offer.

The remainder of this section highlights the contributions of this work.

- Issues in Communication Hardware Design

A detailed analysis of the hardware level of communication systems in multicomputers has been undertaken and the results of this survey presented. This work has focused on direct interconnection networks.

- Generalised Modelling Framework

  1. Framework

     The approach used has moved away from a focus on a given problem to providing a more general modelling platform. The framework offers the possibility of modelling a wide range of communication hardware in parallel computers. Gate level modelling is provided enabling cost analysis in addition to the more traditional performance analysis.

  2. Asynchronous and Synchronous Designs

     Modelling asynchronous designs to the gate level presents a challenge to the designer. The framework is shown not only to handle synchronous but also asynchronous designs.

  3. Design Complexity

     Detailed modelling for cost analysis is a method used to describe design complexity at the gate level. This feature not only aids design analysis but supports design space exploration.

  4. Improved Precision in Performance Details

     An additional effect of modelling design complexity is improved precision in performance data. Router delay is dependent on the complexity of the design i.e. the design instance used.

  5. Traffic

     Traffic movement is represented by the flow of packets through the gate-level designs and, as such, contention will arise due to gate-level interactions as opposed to router level interactions in less complex models.

     It should be noted that although the framework is designed to account for traffic in the form of trace files, the current implementation does not handle a loaded network.

  6. Flexibility and Expand-ability

     RTL design descriptions are modelled using generalised building blocks whose selection or interconnection may be varied to model different designs. Design parameters representing design issues are used to parameterise a given model and aid selection or creation of the appropriate design instance for each of the chosen modules. Logic synthesis of the behavioural descriptions of designs allows synthesis into various and newer technologies within the range of those technologies offered by logic synthesis tools.

- Design Reuse

  The concept of design reuse has played an important part in the development of this modelling framework. At the generalised level a set of modules has been developed to describe a range of designs. Each module represents a feature present in one or more of the routers. By identifying these features as self contained modules it aids design reuse in the development of newer router designs.

  HDL languages are used to describe gate-level designs. Their use also allows minor design changes to be implemented by small changes in the original HDL design file, re-compilation and synthesis to produce a new net-list design description. That is that each design does not have to be put together from scratch.

- Case studies

  The case studies presented include not only examples of use of the model but also present results obtained which illustrate the improved accuracy of results obtainable in models derived from the framework.

1.3 Publications

Features of the framework have been published as full conference papers. An introduction to the framework was given in [Had97]. This article represents a description of the framework at the time of writing. The key elements of the framework are, as those presented in the article but refinement of the module set and the design issues may be seen in the final version of the framework presented herein. Article [HN97] presents a case study of the Torus Routing Chip [DS86] demonstrating modelling, testing and performance analysis using the framework and comparison to a more traditional modelling environment. The latter environment was implemented by the co-author, who also participated in the comparison of these two modelling techniques. In [NH97] the work described in [HN97] is expanded to three modelling approaches and illustrates ongoing work into the ways in which these different modelling approaches may aid realism in performance analysis. The third modelling approach is the work of the co-author. Again both authors were involved in comparison of the different modelling approaches.

1.4 Limitations

There is often a trade-off between latency and throughput in performance analysis. In addition, for a more realistic throughput estimation it is important that trace files be used to stimulate the simulation. That is that traffic is an important feature in performance analysis. Traffic is not, as yet, fully implemented due to problems during the implementation phase. A number of these problems were related to the current version of the Performance Description Language (PDL) [VMM96] used in the implementation of the framework. However, improvements in PDL expected in the newer version of the language should remove the existing problems of handling feedback in designs and large trace files. As such, traffic analysis is postponed to future work.

What may be said to be the largest limitation of this work is the lack of a method to validate the gate-level results obtained. Although behavioural correctness may be checked through simulation, performance results are based on the implementation of the design. For each design made, the performance results obtained are based on HDL descriptions of the underlying designs. Standardisation, expected in the near future, will improve the problems inherent in net-list creation from different logic synthesis tools. However, a factor in the performance results achieved is still dependent on the effectiveness of the code implemented. Further investigation into this area is required.

1.5 Background

A *parallel computer* is a large collection of processing elements which can communicate to solve large problems fast [AG89]. An *interconnection network* provides the communication platform for these processing elements, consisting of nodes and their connecting channels. A *node* may include one or more processing elements and the connection structure of the interconnection network is termed the *topology* of the network.

A well recognised classification of parallel computers is *Flynn's Taxonomy* [Fly66], yielding four categories based on the multiplicity of instruction streams and data streams in a computer system. The *MIMD* category — multiple instruction multiple data may be further divided into those using a shared memory paradigm — *multiprocessors*, and those using distributed, non-shared memory — *multicomputers*. In a *message passing* multicomputer, each node contains an autonomous processor and it's local memory which is physically separate and logically private to the memories of the other nodes. Interprocess communication occurs by routing messages explicitly through a message passing interconnection network where a *message* is the logical unit of information exchange between processes [AS88]. Messages within the network are termed *network traffic*. Traffic arises from the communication inherent in a parallel algorithm and the communication arising due to the mapping of the application to the topology of the network.

The *communication system* of a parallel computer is responsible for the transfer of messages between a source and a destination. Figure 1.1 describes a node of the communication system. As shown, the general case of one computing processor per node is assumed. However, the model is not limited to this case.

At the *software level* of the communication system, communication software includes both message handling code and communication related system calls. The message passing library consists of compiled blocks of code representing message handlers, stored in memory or registers which are activated to handle outgoing and incoming messages. Message related system calls interface applications to the message handling library and the underlying communication network. The pattern of system calls for a given application represents the communication and synchronisation inherent in a parallel application [ZYH94]. However, overheads incurred at the software level of

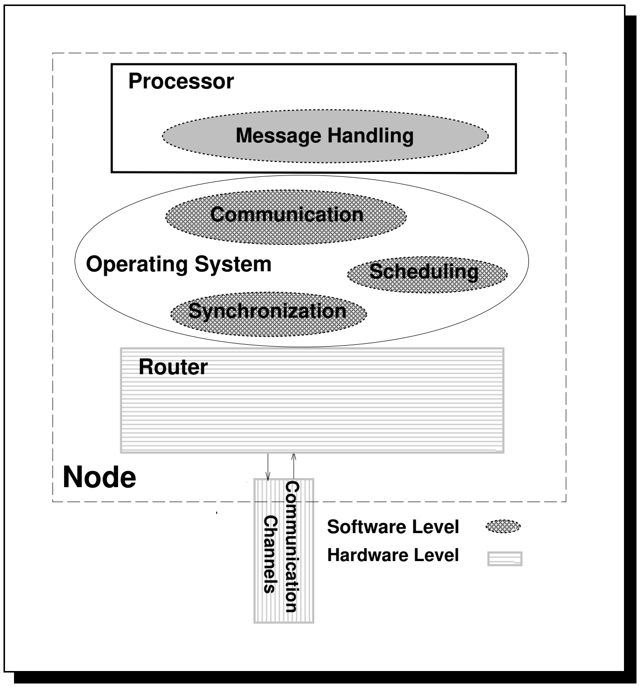

Figure 1.1: A Generic Parallel Computer Node

the communication system must also take into account processor idle time incurred as a result of the communication process.

At the *hardware level*, an interface is required to control the flow of information between the processor and the interconnection network. The complexity of this interface depends on the quality of service offered by the computer system, the topology of the interconnection network and the division of work between the interface and the compute processor itself. This interface is termed a *router*. As an instance of the router appears at each node and together they handle the routing functionality within the network, the router may also be termed a *distributed router*.

1.6 Structure of the Thesis

The thesis is split into three parts.

The first part presents the theoretical base from which the framework is constructed. Presentation of the issues in router and interconnection network design may be found in chapter 2. A description of performance metrics in communication system analysis is given in chapter 3. A detailed description of network latency and the expressions which represent latency in terms of different switching techniques are also included.

Part 2 presents the framework. In chapter 4 the generic modules of the module set and the design issues effecting instantiation of these modules are presented. Chapter 5 presents implementation of the framework and the method for design instantiation.

Part 3 includes both the case studies and ongoing work, in chapters 6 and 7 respectively. The case studies presented are used to illustrate the different features of the framework and the performance and cost results obtainable.

Finally the conclusions and suggested future work are included in chapter 8.

It should be noted that an index of terms is included at the end of the document for reference purposes.

Chapter 2

Communication

The aim of this chapter is to build up a detailed picture of the issues which effect the design of communication hardware.

In section 2.1 the key features of interconnection network design are described. The structure of a generic router is introduced in section 2.2. Section 2.3 describes the functionality of a router and section 2.4 presents the key router design issues. Section 2.5 presents a number of existing routers, both commercial and research, identifying their key design features.

2.1 Interconnection Network

An *interconnection network* (network) may be described in the form of a directed graph where the vertices represent the nodes and the arcs represent the set of communication channels. These graphs and variations of them are used to describe a specific network *topology*. These topologies are classed as *direct* (static) or *indirect* (dynamic) interconnection networks. In the former case the networks have passive links whereas in the later case there are active switching elements which allow dynamic reconfiguration of the network. In this work it is chosen to focus the attention on direct networks as these are the most common type of networks found in todays multicomputers. This may be seen from those multicomputers listed in section 2.1.1.

Interconnection networks may be characterised by their degree or diameter. The *degree* of a network is defined as the maximum number of connections at a node. This is related to the *diameter* of the network. The diameter is the maximum of the shortest paths taken over all pairs of nodes in the network. If the degree of a network is high, that is if a node is connected to many other nodes, the diameter of the network will be low. This is because there will be more direct connections and therefore traversing the interconnection network will take less steps. These factors affect the processing time required by any algorithm which requires global information movement.

2.1.1 Direct Networks

Direct networks are classified according to their dimensional layout. Examples of these include one-dimensional e.g. linear array; two-dimensional e.g. ring, star, tree, mesh and systolic array; three dimensional e.g. chordal rings, cube-connected cycles and n-dimensional e.g. n-node binary hypercube and k-ary n-cubes.

The two-dimensional mesh (mesh) and binary n-cube (hypercube) are the most common direct network topologies implemented in commercial and non-commercial multicomputers [OKMS95].

2.1.1.1 Mesh

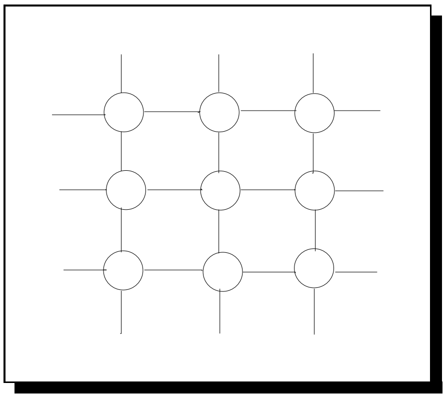

Figure 2.1: Mesh Topology

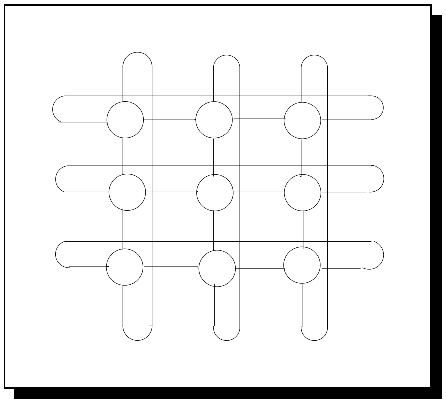

Figure 2.2: Torus Topology

The *mesh* term most commonly refers to the 2D version where there are k^2 nodes i.e. n nodes in each dimension. Each node has between two and four connections depending on it's location in the mesh — see figure 2.1. A 2D version is implemented in the Intel Paragon [Int91] and a 3D version in the J-machine [ND92].

The *torus* — end to end mesh or nearest neighbour mesh, is an extension of the basic mesh with additional wraparound connections both horizontally and vertically — see figure 2.2. The Cray T3D network is an example of a 3D torus.

Other variations of the mesh include *shifted wrap around*— wrapped edges are shifted rows/columns by the respective shift amount, and *cylindrical* — only one edge is wrapped either horizontally or vertically. Further extensions include adding extra edges e.g. Illiac IV [Bar68], MasPar MP-1 [Nic90] or using buses to connect nodes by row or column e.g. Orthogonal Multiprocessor (OMP) [HTK89].

2.1.1.2 Hypercube

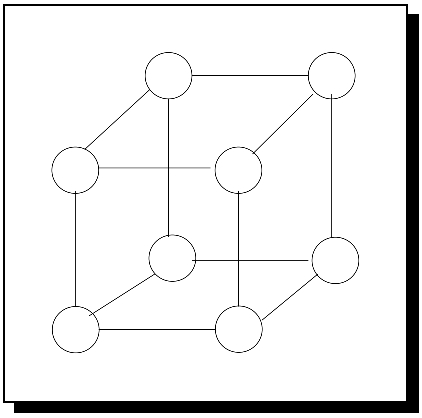

Figure 2.3: Hypercube Topology

The *hypercube* topology is also known as a Boolean n-cube or a nearest neighbour n-dimensional mesh. The number of processors in the network is 2^n arranged in an n-dimensional cube — see figure 2.3. Each node has a unique identification number such that each bit position represents a dimension in the network and each adjacent node differs by only one bit. Hypercubes support the performance requirements of 3D scientific applications. Examples of such architectures are found in the Caltech Cosmic Cube [Sei85b], the Connection Machine CM-2 [Dun90] and HC16-186 [BG88].

In a *generalised hypercube* the basic format is an n-dimensional mesh with added edges ensuring that every node in a row or column is connected to all other nodes in a row or column. An example is the B-HIVE machine of North Carolina State University [Aea86].

2.1.2 Implementation Issues

It is not sufficient to consider graph theory alone when considering the viability of topologies. An important network property is whether it is in fact implementable in current technology. Implementation issues include pin-out limitations, wire density, wire length, bandwidth and inter-chip signalling and packaging.

2.1.2.1 Pin-out Limitations

In current technology, implementation of even a small interconnection network on a single chip is not feasible. Pin limitations give rise to a trade-off between the number of channels and the width of channels due to the fixed number of pins in a given sized chip. In addition, a further pressure on this limited resource is given by the need for associated signalling for each channel. The more signals required per channel, the less channels that can be implemented. This also limits the availability of services available at the network level.

Bidirectional channels are often implemented as two unidirectional channels to avoid the additional delays incurred in using a single bidirectional channel. These additional delays are incurred due to arbitration, often implemented in the form of a token which is passed between competing routers to identify the owner of the channel. However, bidirectional channels increase the pin-out problem, requiring a set of pins for both the output and the input parts of the channels. However, a solution to this problem has been found by Dennison, Lee and Dally [Dea93]. Their solution allows simultaneous transfer in both directions over a single wire. That is two routers may communicate with each other simultaneously across the same channel. This scheme has been used in the design of the Reliable Router.

2.1.2.2 Wire Density and Bisection Width

Wire density describes the feasibility of implementing a given structure in a limited area with respect to a given implementation technology. Compactness of wires depends not only on the width of the wires and their isolation layer, but also on the necessary gap between wires. A goal in the design of the J-machine was to achieve compact wiring, as achieved in the 3D mesh structure implemented [Dea89].

The *bisection width* of a network defines the number of wires which traverse a cross section of the network topology. If the topology is symmetric, this cross section will split the network into two equal parts. Wires are of finite width and electrical problems such as crosstalk limit how close wires can be placed together. As such, there exists a finite limit to the number of wires that can be placed within a given area. This feature is, therefore, important when considering the feasibility of a given network in a given technology medium.

2.1.2.3 Wire Length

A major factor in topology analysis is the wire length factor. To ensure correct signalling between resources, the wires of the channels must exhibit transmission line properties. If the time of flight on a channel is much less than the rise time of the driving signal then transmission line factors may be ignored.

Bus-based topologies or topologies that are not easily implemented in 3D space may require long wires. The speed of transmission on these wires is limited by the speed of the communication medium. Assuming that channels are driven at the maximum speed then increasing wire length will increase the time of flight on these wires.

The nodes of a network are dependent on the implementation medium rather than the communication medium. Increasing clock speeds within the nodes leads to a reduction in rise time and thus increases the importance of transmission line effects.

2.1.2.4 Bandwidth

The *bandwidth* of a network depends on both the *channel cycle time* — the time required to send a bit over a single wire and the *channel width* i.e. the number of wires in a channel.

2.1.2.5 Inter-chip Signalling and Packaging

The performance of a parallel system at the lowest level depends on system partitioning and implementation. The effect of packaging can be the difference between on-chip, between-chip and between-board communication. In addition to the difference in propagation latency, drive signal times vary between different levels of packaging. There is greater delay incurred in off-chip driving circuitry and off-chip wiring than in on-chip. In addition, inter-chip signalling incurs a packaging penalty. Different phase signalling for on-chip and off-chip signals may be used, as in the TRC [DS86].

2.2 Structure of the Router

A *router* may be considered as consisting of three logical parts, as shown in figure 2.4. The central part implements the functionality of the router — switching, routing and flow control. The switching method describes the physical path setup i.e. end-to-end or node-to-node, the routing algorithm decides which node is the next node i.e. which outgoing link to choose, and the flow control checks the availability of the required buffers. These functions are discussed in more detail in section 2.3. The router applies this functionality to traffic entering and leaving at the two outer parts — the processor and network interface.

The structure of a router can be described as a collection of components working together to meet these functional requirements. For a specific router, the selection of the required components, the design of the individual com-

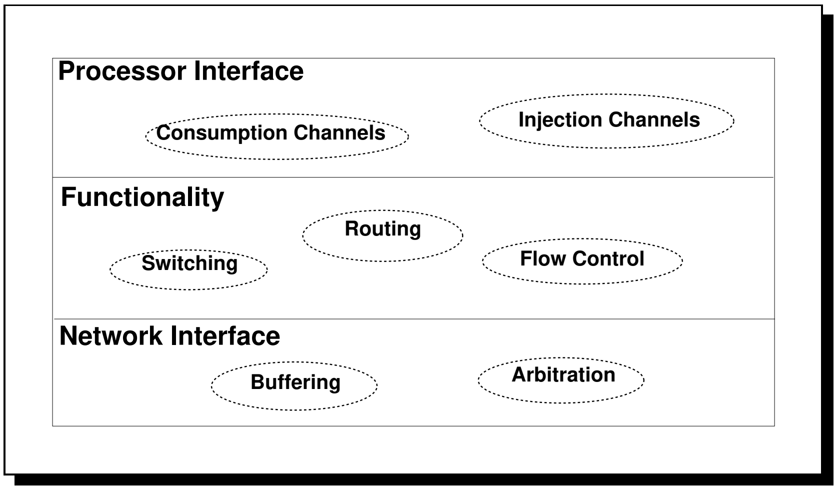

Figure 2.4: A Functional Router Description

ponents and the method of connecting them are dependent on the design choices made. The variety of design issues and their interdependencies i.e. large design space, makes optimal router design a near impossible process.

2.2.1 Processor Interface

The router should be capable of handling all requests handed to it by the processor at the speed the processor produces them — *injection rate*. This implies that there must be sufficient input channels from the processor to the router, *injection channels*, such that the traffic flows into the router at the required speed and that the router is able to handle this flow of traffic.

On the other hand, the processor must be able to handle all requests handed to it by the router at the speed the router outputs them — *consumption rate*. Similar to the above discussion, sufficient outgoing channels from the router to the processor are required to handle the flow of traffic to the processor. These are called *consumption channels*.

Injection channel and consumption channel issues are discussed further in sections 3.3.1.1 and 3.3.1.2.

2.2.2 Network Interface

The above discussion also applies to the *network interface*. The network interface is required to receive and handle requests from neighbouring nodes i.e. choose an outgoing channel and forward the incoming data, in parallel. The number of channels at this interface depends on the *degree* of the interconnection network topology i.e. the number of *neighbouring nodes*.

2.3 Main Functions

The three main functions identified in section 2.2 include switching, routing and flow control i.e. physical path setup, next node decision making and buffer availability information. As such, the three functions - switching, routing and flow control work together to send data across the network.

2.3.1 Switching

The switching method describes the process of setting up a physical path through the routers of the network. There are three established switching methods: circuit switching, message switching and packet switching. In the case of *circuit switching*, a physical path is established between the source and destination before transmission of a message begins. In *message switching*, on the other hand, no physical path is set up between the source and the destination. Instead, the message is sent one 'hop' at a time from source to destination i.e. one step on route to the destination is set up before transmission begins. *Packet switching* is similar to message switching except that a message is divided up into *packets* of fixed or variable size before being sent. A *header* is added to the front of each packet, containing the routing information needed to route packets independently through the network. The contents of the original packet is referred to as the *body*. This process is termed *packetising* and is discussed further in section 2.4.5.

Packet switching, was originally designed for MIMD parallel computers [HB87]. It may be implemented through a choice of switching techniques. The *switching technique* describes the flow and buffering of packets as they move from source to destination. The most commonly implemented switching techniques include store and forward [KK80], wormhole [Sei85a] and virtual-cut-through [KK79] routing. A discussion of these may be found in section 2.4.1.

2.3.2 Routing

A routing function maps a sending node $\mathbf{x}$ to either one or a set of adjacent node(s) $\mathbf{y}$, depending on the routing algorithm. If the destination node(s) is not an *adjacent node*, where there exists a direct connection between the two nodes, then routing consists of a sequence of routing functions. The routing decision information is usually implemented at each node in the form of a *routing table*.

A routing algorithm may be deterministic (non-adaptive or static) or adaptive (dynamic). In *deterministic* routing, a routing decision depends on both the node at which the current routing decision is made and the destination of the message. As such, the routing table for a given node is pre-determinable. In *adaptive* routing, on the other hand, routing decisions also reflect the current traffic in the network. Therefore, the routing tables of all nodes need be constantly updated to reflect traffic within the network at any point in time [Tan89]. The number of routing choices increases with the adaptivity of the network and is termed the *degree of routing freedom*. Routing algorithms are discussed in more depth in section 2.4.2.

Routing may either be implemented in software or hardware. Software routing has been implemented in the Caltech Cosmic Cube [Sei85a], and N-Cube [NCU90]. However, a major shortcoming of software routing is that it is slow, with respect to *hardware routing* and therefore limits the effective bandwidth of a channel. Most routing is now hardware controlled as in the Intel iPSC/2 [Nug88], Intel Paragon XP/S [EK93], TMC CM-5 and Cray T3D [KA95].

2.3.3 Flow Control

To move data between two adjacent nodes three elements must be present, the source buffer, the channel being allocated and the receiving buffer [Hwa93]. A method is required to monitor the availability of these resources and control the flow of data to ensure that data transmitted is received at the receiver

[Hal89]. This method is termed flow control.

The *unit of flow control* is the smallest unit of information that a queue or channel can accept or refuse [Dea87] and, as such, cannot be less than the channel width. The maximum number of bits that can be transmitted across a channel at any one time is equal to the number of single bit wires that make up a channel and is termed a *phit*. The size of the flow control unit is dependent on the choice of switching technique. Switching techniques are described in more depth in section 2.4.1. In wormhole routing and virtual-cut-through routing the *flit* is the unit of flow control. The size of the flit is normally chosen to equal the phit size to avoid the extra logic and signalling required to handle larger flit sizes [Chi93]. However, an exception to this rule can be found in the case of the Reliable Router [DD94a] where each flit must be divided into frames for sending. The flit is, however, still the unit of flow control. In store and forward routing, on the other hand, the packet is the unit of flow control.

2.4 Router Design Issues

This section outlines the fundamental issues applicable to router design in a direct network topology.

2.4.1 Switching Techniques

As described in section 2.3.1, in packet switching, a message is split into packets and each packet is routed through the interconnection network independent of other packets. However, since the packet size, in general, exceeds the width of the channel, the whole packet cannot be sent in parallel. Therefore, packets are broken into a number of flits for transmission.

In early packet switched parallel computers the switching technique implemented was *store and forward routing*, where all the flits of a packet are buffered at a node before the first flit continues.

In later generation parallel computers, *virtual-cut-through* and *wormhole routing* techniques have been implemented e.g. Cray T3D, NCUBE-2, Intel

Paragon, Caltech Mosaic, MIT J-machine. In virtual-cut-through routing, as soon as the head of the packet is ready to continue, the packet can continue towards the destination. That is, only the flits that make up the head of the packet need be buffered, the remaining flits can follow the head in a pipelined fashion. However, if the packet blocks i.e. cannot proceed, the node where the head is buffered must buffer the whole packet. Therefore, sufficient buffering capacity is required to meet this worst case situation and as such matches the buffering requirements of store and forward routing [Dea87].

Wormhole routing is similar to virtual-cut-through routing except that, when the head of a packet blocks, the remaining flits are stopped 'in place' occupying buffers and channels in the path [ND92, KLC94].

A further interesting example is the *Mad Postman* algorithm developed by Jesshope et al.[JMY89]. As soon as the first flit of a packet is received the flit is forwarded to the next node in the same direction i.e. before a routing decision is made. Once the decision is made, then a check is made as to whether the forwarded flit was sent in the appropriate direction or not. In the later case, the flit is resent either to it's local destination or to the next dimension as given by the routing decision. The incorrectly sent flit is referred to as a 'dead' flit which is recognised as such by the intermediate nodes on it's path through address comparison. It is ignored at each intermediate node until it either reaches the network boundary or blocking occurs at a node, where it is discarded.

Pipelined circuit switching (PCS) introduced by [GY93] is similar to wormhole routing except that the remaining flits of a packet do not follow the header until the header has reached it's destination and an acknowledgement has been returned. Then, the remaining flits follow in a pipelined fashion. This freedom of the header from the body allows a new backtracking routing algorithm to be implemented. As such, instead of the header blocking on contention, as in wormhole routing, the header backtracks freeing channels on it's reverse path. As soon as a profitable route is found i.e. an outgoing channel on the shortest path to the destination, the header continues.

A variant of circuit switching termed *circuit switching with blocking* is implemented in the Virtual Channel Router [HB90b]. In this version a two byte routing probe (setup packet) is used to create the path to the destination. However, as the routing probe sets up the path, the remainder of the header which includes further control information follows behind. If contention is encountered before reaching the destination then the setup packet blocks. This means that all flits of the probe and header block the channels of it's path as in wormhole routing. When the path is completed and acknowledgement returned, the message portion is sent as in traditional circuit switching. In addition, once a path has been set up and a message sent, the path is treated as 'cached'. That is that the path is reserved for the next packet of the message but may be torn down if required.

The use of virtual-cut-through or wormhole routing reduces latency by pipelining a message's transmission over a number of channels in it's path [KLC94]. This gives a significant performance advantage over store and forward routing for long messages in high diameter networks under light loading. However, under light loading with short messages there is very little performance advantage [Dal90b].

Wormhole routing reduces the buffering requirements, as described, and as such reduces the size of the router. However this increases the deadlock possibilities as described in section 2.4.3. Hot spots are also a big problem with wormhole routing as described in section 2.4.4.

The variants, mad postman, pipelined circuit switching and circuit switching with blocking attempt to reduce buffering requirements, improve fault tolerance and reduce the setup overhead respectively over the more traditional switching techniques.

2.4.2 Routing Algorithm

The goal of the routing algorithm is to efficiently route a message through the interconnection network. As stated in section 2.3.2, routing algorithms can be categorised as *adaptive* or *deterministic*. An adaptive routing function gives a set of alternative output channels [Dua91a] whereas in deterministic routing the function restricts the choice of output channels to a single channel.

There are three families of adaptive routing algorithms - global, local and a hybrid of global and local algorithms [Tan89]. The degree of adaptiveness ranges from *shortest queue* which is based on local information to *retrograde* which routes round heavily utilised links. A relatively new adaptive routing framework is termed *compressionless routing* [KLC94]. It is based on the wormhole routing technique with the addition of deadlock and recovery techniques and is described in section 2.4.3.

Another categorisation used for adaptive routing is *minimal* or *non-minimal* adaptive where minimal adaptive reduces the routing choice such that the choice of channels is restricted to those which lie in the direction of the destination — termed productive channels.

The two main classes of deterministic algorithms are random and restrictive. An example of a *random* algorithm is that developed by Rajaseraran and Raghavacheri [SR91]. Here, each node has a number of packets to send and for each packet the processor flips a 2-sided coin to decide if a packet is to be sent in the x or y direction. Whichever direction is picked, a random next node is then chosen in that dimension.

Restrictive routing, as the name suggests, places a restriction on the choice of outgoing channels from each router. Examples of restrictive routing include e-cube routing [SR77] for the hypercube topology and a variation of this termed x-y routing for the 2-dimensional mesh topology. These two routing algorithms can also be classified as *dimension-order* routing as they route messages completely in one dimension before beginning to route in the next dimension.

The routing logic increases in complexity with increased routing freedom i.e. from the simple logic of fixed routers to the complexity of fully adaptive routers. Routing complexity effects both the area requirement of routers as an implementation issue and perhaps more importantly their speed as increased complexity implies an increase in the number of gates that a signal must pass through.

Random routing has the advantage of low cost due to it's implementation simplicity. The *degree of locality* refers to the amount of nearest neighbour communication within the applications communication requirements. Applications which exhibit a low-degree of locality also benefit from an even distribution of messages. However, for applications which exhibit a high degree of locality, an even distribution of messages is not sought and random routing gives rise to a decrease in performance with respect to other routing strategies.

Restrictive routing outperforms random routing and offers some of the performance benefits of adaptive routing without the high costs. It offers hardware simplicity and ease of *logic partitioning*, leading to better scalability. In this case, logic partitioning refers to separating the routing logic into dimension dependent modules. However, restrictive routing lacks routing flexibility. The lack of fault tolerance in deterministic routing can lead to poor performance under heavy *loading* — a large amount of traffic in the network, or hot spot traffic, see section 2.4.4. Also, deadlock avoidance, as described in section 2.4.3, requires the addition of *virtual channels* which can quickly remove the cost advantage over adaptive routing.

A lot of attention, in recent literature [LH91, GN92, UP97] has focused on adaptive routing which exhibits low network latency and high bandwidth. However, the performance benefits in respect to restricted routing must be traded off against the high cost of *deadlock avoidance* [Dua91a]. This increased cost is due to the large number of virtual channels required i.e. 2^{n-1} virtual channels are required for an **n**-dimensional network. Adaptivity also leads to more complex routing decision logic. These factors together lead to an increase in the node delay resulting in an increase in the delay encountered as a packet is transmitted through the network [CF93].

In comparing routing algorithms we must also bare in mind that there is an upper bound on the speedup that can be attained. Therefore, if simpler routing algorithms can achieve near to this bound, it is shown that there is no need for more complex algorithms [Dan91, MBR93], where complexity is defined in terms of the degree of routing freedom.

2.4.3 Deadlock

Contention within the nodes of an interconnection network occurs when two or more packets compete for the same output channel. There are four methods for resolving this contention [Hwa93] — discard and retransmission, detour, blocking and buffering.

In the case of *discard and retransmission*, when contention occurs, one of the packets is discarded i.e. the information is removed from the network and the packet is resent. In general, a packet discarded must be sent again from the source. However, in the Reliable Router a Unique Token Protocol

[DD94a] is used to avoid this problem — see section 6.3.

Detouring, allows packets to be routed around contentions. That is, a packet reaching a blocked node will be automatically rerouted onto another channel attached to a different adjacent node. At the new node, if there is no contention the packet can again be routed as normal.

In wormhole routing, *blocking* is implemented. This involves stopping a packet in place i.e. the flits of a packet fill up the flit buffers and channels in their path and thus block the progress of any other packets requiring the channels that they occupy [Dea87]. On the other hand, in store and forward routing as described in section 2.4.1, if contention occurs, the packet is *buffered* at the contention node and, in effect, is removed from the network. In virtual-cut-through routing buffering is also implemented but the buffered flits are forwarded as soon as the contending packet has progressed from the node. That is that sending of the packet begins even though the complete packet may not have reached the contention node. This may be thought of as partial buffering i.e. only part of the packet must be buffered, even though a packet sized buffer must be physically available in case it is required.

Although these methods solve the problem of contention, they do not necessarily ensure forward progress and as such, deadlock may still arise. Since there cannot exist infinite buffering, we can assume that at some point the buffers may become full. *Deadlock* in an interconnection network of a parallel computer occurs when no message can be advanced to its destination because the queues of the message system are full [Kle76, Dua91a].

There are two main techniques for solving the deadlock problem. The first is to use *deadlock prevention* in the form of deadlock-free routing algorithms and the second is to use deadlock detection and recovery.

2.4.3.1 Deadlock-free Routing Algorithms

A condition for *deadlock-free routing* is the absence of cycles in a *channel dependency graph* [Dea87]. This graph differs from that described in section 2.1 in that, instead of the nodes being the vertices of the graph, the channels are the vertices. As such, the edges illustrate the dependence between the different channels, as described in the routing algorithm. Deadlock may be prevented by assigning a partial ordering on these resources within the routing algorithm.

The two main methods implemented to date are structured buffer pool [Gun81] and virtual channels [Dea87]. The *structured buffer pool* approach splits the buffer space of a physical channel into an ordered class of buffers. As a packet flows through the interconnection network it is assigned to a specific class of buffer at each intermediate node, according to the routing algorithm. Each packet is assigned a buffer independent of other packets of the same message. Many deadlock-free routing algorithms have been developed based on the structured buffer pool approach for store and forward routing [TU79, MS80, Gun81, Gel81].

The *virtual channel* approach, on the other hand, restricts the routing of packets. A group of virtual channels, each with it's own queue, are attached to each physical channel, The virtual channels are ordered and routing is restricted to visiting the channels in decreasing order. This may be achieved dynamically as in [Dua91a, LH91, DD94b] or statically as in [CK92, Dea87]. Note that virtual channels are also referred to as *virtual circuits* in the literature.

These two methods are similar in that both place a restriction on resources — structured buffer pool on the assignment of buffers to packets and virtual channels on the routing of packets. Both methods can be applied to store and forward routed networks but only the virtual channel approach can be applied to wormhole routed networks [DS86]. This is due to the fact that buffer allocation at the flow control unit is independent of other flow control units. In the case of wormhole routing the unit of flow control is the flit unlike in store and forward where the unit of flow control is the packet — see section 2.3.3. As such, for wormhole routing, this would mean that the flits of a packet are assigned buffers independent of each other. This would therefore not lead to a pipelined transmission and therefore cannot be implemented.

Another method used to avoid deadlock is the *Turn Model* for adaptive routing [GN92]. This method is based on an analysis of the mesh architecture as to the turns a packet may take on a given path and the cycles that these turns may cause. The turn model prevents sufficient turns such that there are no cycles in the channel dependency graph and as such an algorithm based on the turn model is deadlock free. However, this method is only appropriate for mesh topologies. In the case of a torus topology this method cannot be applied and therefore a virtual channel solution must be used [KLC94].

2.4.3.2 Deadlock Detection and Recovery

It has been shown [KLC94] that deadlock situations seldom occur, except when an interconnection network nears saturation. Therefore, if we assume that the interconnection network will not be loaded to saturation, it may be better to detect and recover from a deadlock situation than implement the hardware to prevent it.

In *compressionless* routing, no virtual circuits or structured buffer pool are required. Packets are transmitted as in wormhole routing. In addition, packets are padded with extra flits, if needed, such that the head of the packet reaches the destination before the last flit has left the source. As in wormhole routing, the flits occupy all channels, from that which the last flit occupies, or pad if used, to that which the header occupies. If a deadlock situation occurs, recovery is made by first discarding and then retransmitting the packet [KLC94].

A similar recovery method is used in Abort and Retry routing [RGC89] and in the BBN Butterfly [BBN87].

2.4.4 Hot Spots

A *hot spot* is a concentration of traffic at one point in the network [BWF91]. In the case of message passing architectures, it means that many nodes wish to communicate with a given node and that that node cannot consume all the messages sent to it. Hot spots arise, most typically, due to all-to-one or many-to-one message patterns which are found in many applications.

As described in section 2.4.3, in wormhole routed networks, when the head of a packet is blocked, the remaining flits of the packet occupy the channels and buffers leading to it. All packets arriving at the hot spot are effected in a similar way and in effect the hot spot problem is spread through the interconnection network in a rippling fashion. This effect is termed the *ripple effect* and is a secondary effect of hot spots. Other switching techniques can also suffer from the ripple effect although the problem is most pronounced when wormhole routing is implemented [BP93].

To reduce hot spots, one solution is to use *combining techniques*, implemented in hardware or software, where multiple messages heading for the same destination are combined into one message. These techniques lead to a reduction in traffic which, in effect, lowers the network loading and reduces latency in the interconnection network.

As discussed in section 2.2.1, the addition of extra consumption channels at each node allows traffic to be consumed from the network in parallel. This reduces hot spots and is particularly useful for networks where the router serves more than one local processor.

In wormhole routed networks, a priority system of two virtual channels may be used to solve the hot spot problem. These channels are termed, for example, priority-one and priority-zero where the priority-one network is allowed to pre-empt the priority-zero network. This, enables redistribution of the contending traffic [ND92].

2.4.5 Packetising

Packetising is a technique used to reduce the congestion caused by long messages [HB90b]. As described in section 2.3.1, messages are divided into a set of smaller units for transmission. Each unit, packet, is then routed through the network independent of the other packets.

Packetisation incurs a cost both at the sender and receiver for fragmentation and re-assembly of a packet respectively. Performing packetisation in software not only degrades processor performance but also increases communication latency [KC93]. A proposed reduction in this overhead is seen in the Virtual Channel Router where both fragmentation and re-assembly hardware exists within the router [HB90b]. Another disadvantage of packetisation is that it increases network load. This is due to the routing overhead and sequencing overhead in each packet and is more pronounced when small packets are used.

It has been shown in [KC93, Aga91] that network performance degrades in wormhole routed networks as message length increases. When long messages block, the node where the header is blocked cannot buffer the whole message.

Therefore, the message occupies and thereby holds idle, multiple channels of the network.

The suitability of packetising depends on the routing algorithm - adaptive or restrictive, and the type of network traffic - *uniform* or *bimodal*. Uniform traffic assumes that packets are of a fixed size whereas in bimodal traffic both short and long packets exist.

In restrictive routers, the effect of packetising is to reduce performance in uniform traffic [KC93]. A single output path choice in restricted routing means that if a packet is blocked this blocking effect is accumulated by other packets following the same path. If virtual channel hardware is also added, the effect is reduced.

The use of adaptive routing or virtual channels in bimodal traffic removes the effect of long messages completely and therefore packetising has no effect.

However, packetising reduces the amount of inter-channel coupling caused by long messages. This can lead to a considerable improvement in performance when there is bimodal traffic - a mixed load of short and long messages.

As such, the choice of packetisation or not is dependent on other router decisions and the type of traffic in the network. Commercial systems which incorporate packetisation include the Ametec 2010 [Sei88], Intel Paragon [Int91] and CM5 [KA95].

2.4.6 Buffering

In packet switched networks, once a packet is granted an output channel within a given router, all other packets are preventing from accessing this channel until the tail of the current packet has traversed the node. Therefore, sufficient buffering is required to handle the traffic waiting to access the channel. Buffering is required at both the interface to the router from the processor and that from neighbouring routers within the network. This is illustrated in figure 2.4.

Each physical link may have one or more associated buffers within the router. More than one buffer is required for the case of structured buffer pool deadlock prevention — as discussed in section 2.4.3.1, or in the case of *priority* *routing*. In the latter, messages are allocated a priority and are buffered according to their priority. This process ensures that higher prioritised messages, for example broadcast traffic, are not hindered by messages of lower priority.

2.4.6.1 Buffer Size

As described in section 2.4.1, both store and forward and virtual-cut-through routing require a packet-sized buffer whereas wormhole routing requires only a flit sized buffer to handle a packet in transit. However, in a loaded network traffic enters the node from both the local processor and the interconnection network. The amount of traffic from the local processor depends on the injection rate — the rate at which the messages are sent into the network, whereas the traffic from the network depends on the communication pattern and mapping of an application to the network topology. The buffer size question is therefore dependent on the switching technique, application needs and the topology itself.

2.4.6.2 Buffer Positioning

In general output queueing outperforms input queueing [Kar86].

In a synchronous system, see section 2.4.8, no packet can be transmitted from a node until the beginning of the next *network clock cycle* - time slot. Packets which are received during the prior time slot can either be buffered at the input or the output of the node.

If queueing at the input is chosen, then during a given time slot, incoming packets are buffered in the appropriate input queues. In the next time slot, the first packet in each queue traverses the node and is transmitted towards the next node, if the output is free. This implies that mean queue length will be longer on input queueing because an input that could have traversed to a free output could be queued behind an input that is queued for a busy output channel.

For a synchronous router, queueing at the output saturates only as utilisation approaches unity [Kar86]. Assuming that the crossbar can handle one input for each clock period and there are N inputs then assuming fair arbitration a queue will be handled after m time slots, where m is less than or equal to n. At saturation it may be assumed that as soon as a packet leaves a queue, a new packet replaces it and each queue must wait N time slots before being handled. As such, when queueing at the input is implemented, saturation is dependent on N.

This case not only applies to synchronous communication systems but also many asynchronous systems that are synchronous within the router. In the asynchronous case, packets do not have to queue until the beginning of a given time slot. If the router is busy, then packets queue until the router is ready to service them.

In more recent routers based on adaptive routing a form of queueing in the middle of the router may be seen. That is that queueing exists before entrance to the crossbar but after decision making at the input. This differs from input queueing where flits are queued waiting for completion of decision making – for a head flit, and forwarding of the previous flit. This feature is important in adaptive routers where there exists a large number of virtual channels and a non-partitioned crossbar. Multiplexing between the virtual channels before entrance to the crossbar leads to a significant reduction in the size of the crossbar.

2.4.7 Arbitration

The purpose of an *arbiter* is to make a decision about the assignment of a shared resource based on the execution of an algorithm [LV82].

Considering a synchronous arbiter, a hardware arbiter may be designed to select a request in one clock cycle whereas the equivalent software arbiter may take many clock cycles [NR84]. Current routers generally use hardware arbiters due to the speed-up gained over software arbiters. Therefore, consideration is restricted to the case of a hardware arbiter.

The arbitration issue is a key issue in designing high performance routers as all flits of a packet must cross the crossbar(s) and the arbiter(s) control the crossbar(s). As such, the arbiter may be said to lie on the *critical path* through a router.

To understand the complexity of the arbiter it is worth considering the requirements of the arbiter [Plu72, Tau76] as described below:

All requests to be connected in parallel.
No intercommunication between inputs(requests).
Ability to service a large number of requests.
The identity of the highest priority request to be found rapidly.
Exactly one output request to be generated from each input request.
Complete resetting of the last request before servicing the next.

From the above description it may be seen that partitioning the crossbar also reduces the complexity of the arbiter as less inputs are competing at each arbiter unit.

A particular arbiter is designed for a specified *arbitration policy* (method). These methods vary from equal acceptance to various priority acceptances. The simplest and cheapest arbitration scheme is *linear selection* where in each arbitration round, the first port is given the highest priority and the last port the least. To improve the fairness of this method, batch processing of the requests may be implemented. That is, all requests available at a given time are processed before new incoming requests are granted access to the arbiter. Another scheme termed *daisy chain*, or *round robin* seen in many router implementations adjust the priority ranking at each arbitration round. That is that the priority of a request is increased by one until reaching the maximum and then returns to the minimum priority. Another scheme seen in routers is a *mixed priority* method. That is that, fair arbitration is implemented with the addition of a priority input [Bhu87]. This may, for example, give priority to the processor injecting into the network over other competing inputs. However, the requests from internal network channels compete on a fair acceptance basis.

In addition to the arbiter method, coding of requests is an important feature in the efficiency of an arbiter. The two main coding methods used are asynchronous binary code developed by Taub, [Tau76] and the binomial code suggested by Kipnis [Kip].

2.4.8 Control

In a synchronous router, transmission of a ready message cannot begin until the beginning of the next network clock cycle. The length of the network clock cycle depends on the network diameter i.e. the time taken to traverse the interconnection network. As such, all message transfers will take the same time, independent of their destination. This means that the router can not take advantage of the locality of communication offered by the topology of the interconnection network. This disadvantages synchronous routers where an application exhibits a large amount of locality in it's communication patterns. Although, synchronous routers are advantaged in simplicity of design of the router module itself, a major problem inherent in large synchronous systems is the clock distribution problem [FK85]. In spite of this, many synchronous routers have been developed — see section 2.5.

The clock distribution problem is avoided by entirely self-timed routers [Sei80]. Asynchronous routers decrease latency because the packet propagation rate is only limited by the node delay of the routers it passes and not on the global synchronisation of the network [YN93, DS86].

2.5 Existing Hardware Routers

In this section a list of features applicable to a number of existing and proposed routers are identified and compared. These features are based on the design issues discussed in section 2.4. These routers have been chosen to represent asynchronous and synchronous, deterministic and adaptive routers so as to give a wide range of designs. It should be noted that further analysis of a number of these routers is given in section 4.3 illustrating modelling of routers within the framework at the register-transfer-level and in sections 6.1 and 6.3 describing implemented models at the gate-level.

An important criteria for the choice of routers has also been based on the amount of information available[1]. Much research into proposed routers is based on performance analysis at the system-level and therefore little architectural details or design decisions are discussed. Therefore, choice has

[1]Attempts to obtain more detailed router information for both commercial and non-commercial routers has been unsuccessful

also been limited to those routers where sufficient information is available for comparison purposes.

2.5.1 Deterministic Routers

It should be noted that in the following descriptions, the topology given for each router design is the base topology. That is that the routers may be cascaded to create routers suitable for higher dimension topologies.

2.5.1.1 Torus Routing Chip [DS86]

The Torus Routing Chip (TRC) was designed for the two-dimensional torus. The main feature of this router is it's asynchronous nature both internally and on the channels. The introduction of virtual channels for deadlock avoidance is also noted. A more detailed discussion is found in section 6.1.

2.5.1.2 Caltech Message Routing Chip [SS93]

The Caltech Message Routing Chip (MRC) has been implemented in the MIT Alewife [KCA94], Stanford Dash [Lea92] and Intel Touchstone Delta. Also variations such as the Elko router, Ginsu router, iMRC were implemented in the Caltech Mosaic C [Sea92], Amatek 2010 [Sei88], Intel Paragon XP/S [Int91] respectively. The MRC is a fully asynchronous router. A feature of the MRC is the use of ripple through first-in first-out queues(FIFOs) for signal control. These handle both external request/acknowledge signalling and synchronisation of the internal modules.

2.5.1.3 Network Design Frame [Dea87]

The Network Design Frame (NDF) builds on the work for the TRC. The main differences being the partitioning of the data paths and the separation of logic into two virtual networks. Partitioning of the data paths was necessary to reduce the size of the crossbar such that the router could be implemented within the frame of the processor chip. It also moves away from the unidirectional torus topology of the TRC to a bidirectional 2-dimensional mesh.

2.5.1.4 J-Machine Router [ND92]

The J-machine router is the followup to the the NDF. It was intended to use the NDF in the J-machine but as can be seen from the table 2.2 some variations in the design were made. An important change is the synchronous nature as opposed to the NDF's asynchronous nature. Also instead of a tail bit, a tail byte was added at the end of the packet, as in the TRC.

An interesting feature of the J-Machine router is the pad design which allows bidirectional channels to reverse direction each cycle without any additional delay. Also, arbitration for the use of the channel is pipelined with data transfer such that a channel is never idle when it has data waiting.

2.5.1.5 Virtual Channel Router [HB90b]

The Virtual Channel Router (VCR) was designed based on the Intel iPSC2 architecture. One variation in the VCR design was the use of two unidirectional channels to represent a bidirectional channel instead of a single bidirectional channel.

A unique feature of this router is the concept of cached circuits. A path is setup through the network as in circuit switching but using a wormhole routing like approach in that the setup packet blocks when contention is encountered. Unlike wormhole routing the remainder of the packet does not progress until after the path is set up. After a packet has completed transmission the circuit is cached and, as such, may be used by subsequent packets following the same route. Another feature is that the packetising function seen in many communication system takes place inside the router as opposed to within the processor as is more generally the case.

2.5.1.6 Message Handling Unit [NLE89]

The goal of the Message Handling Unit (MHU) design was to provide hardware to meet the needs of multicast and broadcast communication patterns found in many scientific applications. This hardware involves implementation of both multicast and broadcast routing algorithms and storage for the additional addresses found in a multicast packet. In addition, selection between the three routing choices — unicast, multicast and broadcast, is required.

2.5.1.7 Main Features of Deterministic Routers

In tables 2.1 and 2.2 the following abbreviations have been made:

d	– data (bits)
c	– control (bits)
t	– tail (bits)
h	– packet header
p	– parity (bits)
VC	– virtual channel
PC	– physical channel
# paths	– number of paths (centralised crossbar)
CB	– crossbar arbiter
VC	– virtual channel arbiter i.e. multiplexor controller
OP	– output arbiter i.e. channel arbiter
N/A	– information not available

Feature	**TRC**	**MRC**	**NDF**
Topology	torus	2D mesh[a]	2D Mesh
Channels	8d, 4c unidirectional	8d[b], 1t, 2c 2*unidirectional	9d, 1p,1t, 5c bidirectional
Crossbar (# paths)	centralised VC $\times$ PC	partitioned —	partitioned —
Switching technique	wormhole	wormhole	wormhole
Routing algorithm	x-y routing	x-y routing	e-cube routing
VCs	2 VCs	none	2 VCs
Buffering	output 4 flits per VC	output,input 4 flits per PC	output 2 flits per VC
Addressing	relative	relative	relative
Packet	2 flits h x flits d 1 flit t	2 flits h x flits d —	2 flits h x flits d —
Control	Asynchronous	Asynchronous	Asynchronous
Arbitration (CB) (VC) (OP)	 linear fair —	 mixed fair —	 fair priority token

Table 2.1: Main Features of Deterministic Routers (Part 1)

[a]2D mesh made up of 2 1-D MRCs
[b]9d, and 16d bit variations available

Feature	**J-Machine**	**VCR**	**MHU**
Topology	3D mesh	hypercube	hypercube
Channels	9d, 6c, 1t bidirectional	(16d,5c)[a] 2*unidirectional	N/A
Crossbar (# paths)	partitioned —	centralised VC $\times$ PC	partitioned —
Switching technique	wormhole	circuit switching with blocking	virtual-cut-through
Routing algorithm	e-cube routing	e-cube routing	greedy multicast, broadcast
VCs	2 VCs	2 VCs	none
Buffering	output 2 flits per VC	output 6 packets	output 1 message
Addressing	absolute	absolute	absolute
Packet	3 flits h x flits d	1 flit	x flits h[b] x flits d
Control	Synchronous	Asynchronous	Synchronous
Arbitration (CB) (VC) (OP)	 fair priority token	 fair fair —	 fair — —

Table 2.2: Main Features of Deterministic Routers (Part 2)

[a]Channels are serial, each bit of the flit sent in serial

[b]includes 1 flit = number of destinations, x flits = destination addresses and 1 flit = source addresses

2.5.2 Adaptive Routers

2.5.2.1 Tokkyu Router [FT95]

A newer feature in the Tokkyu router is the addition of packet expressways to improve performance. These provide low latency bypass paths within the router i.e. bypassing the crossbar and arbitration logic. If a packet is to be forwarded in the same direction and the required outgoing channel is not busy then the packet may use the packet expressway to reach the required output. It also offers misrouting of a packet as soon as a *profitable channel* cannot be found where a profitable channel is one that leads towards the destination.

2.5.2.2 Chaos Router [KS91, Bol, Bol93]

The Chaos router is a non-minimal adaptive router which introduced randomisation to simplify adaptive routing. That is to reduce the complexity inherent in adaptive routers. This form of adaptive routing is termed *chaotic routing*. In minimal adaptive, a profitable channel is sought. However, in chaotic routing, if no minimal channel is available then the head of the input queue is moved to a central queue awaiting a profitable channel. This frees the input channel and allows the next item in the queue to compete for the crossbar. As such, it reduces the problems inherent in input buffering discussed in section 2.4.6.2. If a new packet needs to be added to the middle queue and the middle queue is full then the first packet in the queue is chosen for derouting to an available output frame i.e. the non-minimal element in the routing decision.

2.5.2.3 Reliable Router [DD94a]

The newer features introduced by the Reliable Router (RR) include fault tolerance, improved router channel interface and reduced pin requirements. Link monitoring and retransmission are used to handle parity errors. A Unique Token Protocol is also included to handle packet reconstruction and identity duplication. A queue-less low-latency channel interface reduces the need for FIFO buffering at the inputs and outputs of the router. The final feature, the simultaneous bidirectional signalling method [Dea93] reduces the chip signal pin count.

Misrouting in the RR differs from that in the Tokkyu router in that misrouting does not take place immediately after a profitable route cannot be found. Misrouting follows the turn model framework in that it is used to solve a fault in the network and not used when the network is fault-free.

A more detailed discussion is found in section 6.3.

2.5.2.4 Ariadne [Aea94]

The Ariadne router is also aimed at improving fault tolerance. This is achieved by misrouting, that is using a backtracking routing algorithm. To allow backtracking of the header, a newer switching technique termed pipelined circuit switching (PCS) [GY93] is used. This is a variant of wormhole routing — see section 2.4.1.

In the Ariadne description given in [Aea94] reference to three virtual channels is made. However, within the Ariadne router no virtual channels are present. However, the internal paths are split into data and control paths. In addition, a bidirectional external channel is implemented, shared by the data path of the two opposing routers. The Ariadne router uses the term virtual channel to represent these two data paths sharing the external paths and a third virtual channel to represent the control path within the router. As this is not in keeping with the general definition of virtual channels, the Ariadne router is regarded as having no virtual channels.

2.5.2.5 Main Features of Adaptive Routers

In table 2.3 the following abbreviations are used:

d	– data (bits)
c	– control (bits)
p	– parity (bits)
VC	– virtual channel
cct-switching	– circuit switching

dim-order	– dimension order
non-minimal	– non-minimal adaptive
minimal	– minimal adaptive
fully	– fully adaptive
# paths	– number of paths (centralised CB)
CB	– crossbar arbiter
VC	– virtual channel arbiter i.e. multiplexor controller
OP	– output arbiter i.e. channel arbiter

2.5.3 Router Design Trends

Changing trends over the years are identified in tables 2.1, 2.2 and 2.3. The mesh, hypercube and their variants have become established topologies for MIMD parallel computers. Earlier routers used unidirectional channels or bidirectional channels implemented as pairs of unidirectional channels. The latter avoided noise problems apparent in bidirectional signalling and increased bandwidth but increased the pin-out problem. However, newer developments in bidirectional signalling have reduced the noise effects and thus reduced delays and in addition in the case of the reliable router, have removed arbitration delays by allowing simultaneous bidirectional signalling.

In earlier routers, a move from large crossbars to partitioned data paths may be seen. The aim being to reduce the complexity of these crossbars and their arbiters and as such reduce switching delays. However, a number of newer adaptive routers are again moving back to the centralised crossbar concept. Partitioning of crossbars eases the problem of larger crossbars and increased arbitration time caused by virtual channels. Using non-minimal adaptive routing, virtual channels are not required and therefore partitioning is not necessary. Another solution, seen in the reliable router, is multiplexing of virtual channels prior to switching at the crossbar again reducing the need to partition the crossbar.

Packet switching and in particular store and forward routing was originally designed for MIMD parallel computers. However, as shown, wormhole routing and virtual-cut-through routing have replaced the earlier store and forward networks. In addition, newer variations are emerging combining the

Feature	**Chaos**	**Reliable**	**Ariadne**	**Tokkyu**
Topology	mesh,torus	2D mesh	torus	k-ary n-cube
Channels	16d,3c bidirectional	16d,7c,1p bidirectional	8d,11c bidirectional	N/A
Crossbar (# paths)	centralised PC	centralised PC	centralised PC	partitioned expressways
Switching technique	virtual-cut-through	wormhole	pipelined cct-switching	virtual-cut -through
Routing algorithm	non-minimal, misrouting randomising	minimal, dim-order, turn model	fully, misrouting backtracking	fully, misrouting randomising
VCs	none	5VCs	none	none
Buffering	middle,input output, 1 packet	middle 16 flits	input,output	output 3 packets
Addressing	relative	absolute	N/A	relative
Packet	20 flits	64d,8p,3c[a] 1 flit head x flit data 1 flit tail 1 flit token	1 flit head x flit data 1 flit tail	1 flit head[b] x flits data
Control	synchronous	synchronous	asynchronous	synchronous
Arbitration				
(CB)	round robin	priority, random	round robin	mixed
(VCs)	—	round robin	fair	—
(OP)	arb lines	—	—	—

Table 2.3: Main Features of Adaptive Routers

[a] Each flit (75 bits) is divided into 4 frames for sending
[b] head includes packet length

benefits of packet switching with circuit switching.

Deterministic routers, in general, use a form of dimension order routing. The introduction of virtual channels, reduces the level of adaptivity required in an adaptive routing algorithm and thus reduces the complexity of it's implementation. However, in general, a large number of virtual channels are required for an adaptive routed network. To avoid incurring the cost of a large number of virtual channels, non-minimalistic adaptive algorithms are used, employing some form of misrouting to avoid deadlock. A trend of two virtual channels is emerging for those deterministic routers using virtual channels.

Buffering requirements is, as yet, an unanswered question in the research community. In deterministic routers, a single output choice is available for a destination and so a move towards output queueing has been seen. This is to avoid the case where a packet whose output is free is blocked behind a packet whose output is occupied. However, the optimum size of buffering required is still an unanswered question. In adaptive routing a choice of output channels is available. The number of output choices depends on the degree of routing freedom implemented in the algorithm i.e. the adaptivity of the algorithm. As such, the advantage of output queueing over input queueing is reduced. The adaptive routers surveyed showed a variation in buffering from input, middle to output and a mixture of these. As such, no clear trend is seen as yet.

In deterministic routers, absolute addressing is beginning to appear as it avoids the delay of the address decrementer on the critical path through the router. Various packet formats are emerging, however the trend shows a flexible packet size although it is often the case that a maximum packet size is implemented.

A trend towards synchronous routers may be seen. However, in VLSI design a newer trend is moving towards asynchronous communication between synchronous parts due to so many different parts being implemented on the same chip. Many of which may operate at different speeds. It is unclear, as yet, whether this trend will also reach the parallel community as more and more nodes are placed on the same chip.

Chapter 3

Performance of a Communication System

This chapter describes the performance metrics — throughput and latency, seen in interconnection network analysis. In addition, a newer factor — gate level cost analysis, is highlighted and the importance of trading off cost against performance is described. Network Latency is described in detail. The expressions derived, are based on recognised theory for network latency. However, the form of the expressions is that which is chosen for the framework implementation. An expression for wormhole routing is first defined and other latency expressions are derived with reference to it.

In section 3.1 the different performance metrics and their applicability to performance analysis are described. In section 3.2, the cost issue is described. Section 3.3 presents a more detailed study of network latency. Expressions for network latency in terms of wormhole routing are found in section 3.3.2. Expressions for store and forward routing, virtual-cut-through routing and circuit switching are derived with respect to wormhole routing in section 3.3.3. Section 3.3.4 summarises the relationship between the different elements of network latency derived in the earlier sections.

3.1 Performance Characteristics of Parallel Computers

The two main performance metrics for communication in parallel computers are throughput — the number of messages delivered per unit time, and latency — the time to deliver a message from a source at one end of the network to a destination at the furthest end of the network.

3.1.1 Throughput

If it is assumed that a network can handle messages at the rate messages are injected into the network then throughput can be expressed in terms of the injection rate. However, this does not represent the capacity of the network. The *capacity* of a network may be expressed as the maximum amount of traffic the network can handle i.e that may be accepted and processed by the network. That is that the injection rate is sufficient in that the network has reached saturation. However, contention within the network does not necessarily allow throughput to reach this maximum figure. Therefore, a description of throughput in terms of capacity or injection rate can be misleading.

There are two main methods for calculating throughput. The first, considers a fixed load — number of packets injected into the network. Throughput is expressed as a normalised rate — number of packets per cycle, based on the ratio of the number of packets injected into the system to the time taken for these packets to reach their destination. The second assumes a fixed time period — number of simulation cycles or number of nanoseconds. A start-up period is given to allow the network to stabilise i.e. to build up traffic within the network. Then a count is taken of the number of packets reaching their destination within the time period [Dal90b]. This may again be normalised to the number of packets per cycle. It should be noted that during this time period packets will continue to be injected into the system.

3.1.2 Latency

The total delay (latency) incurred in transmitting messages across a parallel computer includes both processor idle time due to communication and other software overheads in addition to the delay incurred in the hardware part of the communication system.

3.1.2.1 Software Overheads

Software overheads, also referred to as start-up latency [NM93, BP93], include the parallel primitive execution time for message handling by the operating system and the execution time for message handling code. Software overheads are incurred at both the sender and receiver. However, overheads incurred at the sender are much lower than those at the receiver. This can be explained with respect to the applications running on a parallel computer [Sea93]. A sending operation is synchronous with the application at the source i.e. the values required are available to send. At the destination values will tend to be stored until the receiving application is scheduled. It should be noted that implementation of Active Messages [VEea92] eliminates this scheduling overhead at the receiver by running messages as handlers instead of threads i.e. user code that is run on the stack of the currently running computation. Another proposed scheme is Optimistic Active messages [WH95] which removes some of the message contents restrictions imposed by the active message concept.

3.1.2.2 Communication Related Processor Idle Time

Processor idle time can occur if the running process either completes or is waiting and there are no processes in the ready queue awaiting scheduling.This can arise if there are sequential sections in an algorithm or due to a mismatch in mapping an algorithm to the topology of the interconnection network i.e. one processor is idle whilst others have perhaps long ready queues.

Processor idle time for waiting messages may be minimised by using a dedicated hardware router to handle routing functions as is assumed in this work.

This allows overlapped communication and computation within each node at the expense of context switching.

3.1.2.3 Network Latency

In the following discussion packet switching is assumed, that is that a message is divided up into packets before sending – see section 2.3.1.

Network latency equals the time from when the head of a packet enters the network at the source until the tail emerges at the destination [AS88]. It is usually expressed as a worst case delay for the network. Using the diameter of the network as the path length in the latency expression provides for the worst case as the diameter is the maximum number of hops a message can make crossing the network i.e. the longest path.

There are two types of delay encountered as a packet traverses a network from it's source to destination node — that encountered at each node and that crossing each channel. Two constant values are generally used to represent the delays through the router (node delay) and channels (wire delay) respectively. Previously the delay through the router was considered to be negligible. However, in more recent work based on current technology, node delay is assumed to contribute substantially to network latency [Aga91]. This difference is reflected by a much greater figure for node delay as opposed to that for the wire delay.

In some architectures latency may be very much larger than the 'average' communication time suggested by throughput i.e the inverse of throughput. Factors that increase latency include queueing and contention for output links. These factors arise from the traffic in the network which is dependent on the application's communication requirements, the topology of the network and the mapping of the application to the topology of the network. Traffic features include the number of packets sent; the size of the packets and the distribution of packets over the network. *Loading* expresses traffic in a network and may be described as the maximum number of packets that pass through a node over a specified time interval i.e. unit time. *Load-balancing*, distributing the work evenly amongst processors is important as it allows more of the communication resources of the system to be well used i.e. it enables a more even distribution of traffic across the network. If packets are prioritised then packets from the same message may not follow in sequence because of higher priority traffic and thus the latency or packet waiting time is increased.

The above description is dependent on the topology of the network and an application's requirements. That is that latency analysis takes account of the topology structure or implementation. However, what about the router itself? Complexity of the router implementation effects the delay incurred traversing each router in the network. For a given packet to reach it's destination, it will encounter a number of node delays dependent on the number of routers traversed. A more accurate router delay estimate will improve the accuracy of the estimated delay for a packet crossing the network. Therefore, instead of using a fixed delay for the delay through a router independent of the implementation, the delay should reflect a given router instance.

3.1.3 Latency Versus Throughput

The relative importance of these two metrics — latency and throughput, depends on the application. Communication intensive applications with sufficient inherent parallelism are limited by throughput. On the other hand, if a program has only a limited amount of parallelism, latency reduces performance as there is insufficient parallelism to overlap communication with computation.

Improvements in latency analysis described in section 3.1.2.3 will also aid the accuracy of throughput analysis. As such, the computed rate at which messages reach their destination will be more realistic.

3.2 The Cost Issue

Traditional performance models have not, in general, focused on the cost issue. A number of researchers have looked at the technology constraints with respect to pin-out limitations and wire density [AP92, Aga91, DS87, LOKM96, MOK94]. *Pin-out limitations* trade-off the number of network channels against the width of these channels for a given chip size whereas *wire density* describes the feasibility of implementing a given wiring structure in a limited area with respect to a given technology — see section 2.1.2. These analyses can be said to be undertaken at the system or register-transfer-level.

The work of Chien [Chi93], identified the importance of cost analysis at the gate-level. His work introduced a parameterised cost model for comparison of routing algorithms. His aim being to make the supporters of adaptive routers substantiate their claims. That is to prove that the performance improvements gained were sufficient to support the additional cost incurred.

Performance improvements, in general, incur an additional cost. Most performance improvements at the network level, such as the move from deterministic to adaptive routing, effect the complexity of the router and thus the node delay. Therefore, if cost is an issue for a given system then these performance improvements should be considered in the light of the cost incurred.

The detailed modelling platform required for cost analysis also provides a platform for design exploration. Different router designs, may range from minor design changes i.e. a small change to an individual modules implementation, to larger design changes. The choice of routing algorithm is one of many design issues for a router designer. A more detailed modelling platform enables differing design decisions to be traded-off with respect to cost and performance on a common analysis platform.

A given design implemented in a given technology may result in performance and cost figures quite different from those arising from the same design implemented in a different technology. Therefore, taking the analysis one step closer to a hardware implementation must also involve analysis with respect to a given technology. Since technology is rapidly changing, the ability to analyse a design, with respect to cost, in both current and future technology is important.

In this work, cost analysis focuses on the routers themselves. That is cost is based on a gate-level analysis of the routers of a network rather than the network level analysis seen in earlier work. Unlike the work of Chien [Chi93], this analysis is not restricted to that of routing algorithm analysis but to router design in general.

3.3 Network Latency Expressions

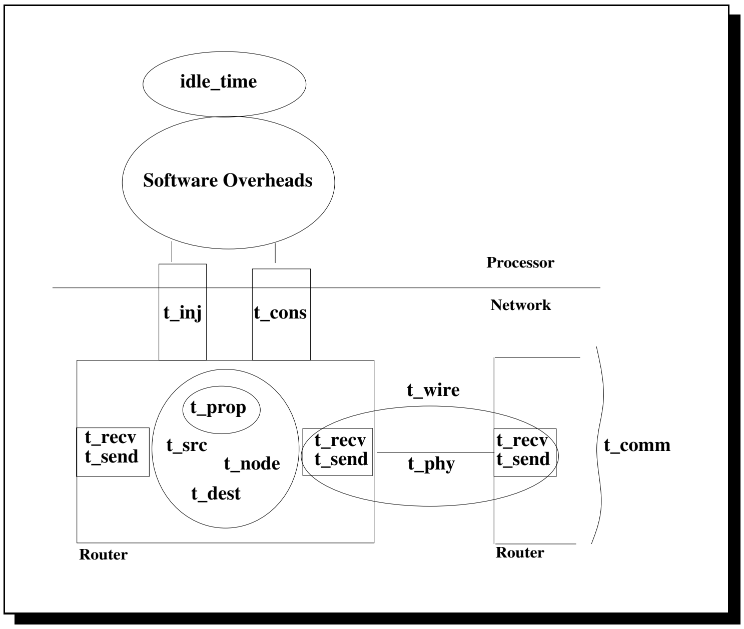

Figure 3.1: The Components of Latency in Message Transmission and Reception

Figure 3.1 illustrates the relationship between the software and hardware levels of the communication system, as described in section 3.1.2. As stated, the focus of this work is at the hardware level and as such, network latency is further refined. A number of elements of network latency are identified in the figure. These elements include those at the interface between a processor and the interconnection network : **t_inj** and **t_cons**; those at the interface between a router and a network channel : **t_recv** and **t_send**; that of the channel itself: **t_phy** and those delays within a router : **t_prop**, **t_node**, **t_dest** and **t_src**. These elements are defined and network latency expressions derived within this section. The components **t_wire** and **t_comm** combine a number of these elements.

3.3.1 Overview of Network Latency

Statement 1 *Delays encountered between routers and their local processors are non negligible.*

Earlier analysis of network latency often included only the communication delay across the interconnection network [Aga91]. However, delays are also encountered at the entrance — *injection delays*, and exit – *consumption delays*, of the network. These delays are expressed as the delays encountered on the injection and consumption channels respectively. Although the term channel is usually used to refer to connection wires between two resources, in this case it is defined as the input and output queueing to the network in addition to the wire delays themselves.

The contribution of the injection and consumption channels to end-to-end delay may be substantial. This feature is especially relevant at the destination where bottlenecks may appear [BP93]. It is, therefore, important to consider these delays as part of network latency.

Network latency is described as follows:

$$NetworkLatency = t_{inj} + t_{cons} + t_{comm} \tag{3.1}$$

where

t_{inj}	— input queueing to the network
t_{cons}	— input queueing to the processor (output queueing from the network)
t_{comm}	— communication delay

3.3.1.1 Injection Channels

The delay incurred at the injection channel, t_{inj}, is dependent on the *effective bandwidth* of the network:

$$EffectiveBandwidth = \frac{1}{t_{wire} + t_{node}} \tag{3.2}$$

where

t_{wire}	— wire delay
t_{node}	— node delay

The delays wire and node reflect the delays incurred sending data across a channel and through a router of the network respectively. It should be noted that this expression for bandwidth includes the effects of queueing and arbitration within the routers. That is that the node delay refers to the delay through a router in a loaded network. These delays are described in more detail in sections 3.3.1.2 and 3.3.2 respectively.

If the bandwidth is sufficient in that the network can handle all the messages sent to it then the injection delay can be neglected [Chi93, BP93]. This assumption is often seen in the literature. However, as can be seen from equation 3.2, node delay is inversely related to bandwidth and therefore more complex routers, which give rise to larger node delays, decrease the effective bandwidth and can lead to injection delays.

3.3.1.2 Consumption Channels

In studies of advanced routing techniques, infinite consumption channels are often assumed [LH91]. However, if the bandwidth is high and there are insufficient consumption channels to handle traffic arrival in parallel, then traffic will have to be queued and handled sequentially at the destination. An additional queueing delay may, therefore, be seen at the exit of the network. There is also an additional decision delay due to arbitration of packets when more than one packet arrives at the same time. As such, lack of sufficient consumption channels can give rise to a large bottleneck in the interconnection network. A particular case of this problem is the hot spot problem described in section 2.4.4.

A solution to these problems is multiple consumption channels. Performance is improved by allowing traffic to be queued and removed from the network in parallel. However, as implementation cost increases with an increase in the number of consumption channels, then any improvement in performance must be traded-off against these increased costs. An analysis by Balakrichnan [BP93] showed that the optimal number of consumption channels for a dimensional router was two and for adaptive routers between four and six. As such, it is important that modelling of consumption channels is not restricted to the case of a single channel.

3.3.1.3 Communication Delay

Communication delay, t_{comm}, arises from the *node delays* (switching delays) and *wire delays* across the network. Node delay is dependent on the routing algorithm and particularly on the degree of routing freedom; how arbitration of resources occurs and how deadlock prevention is achieved. Node delay is encountered not only at the source and destination node but also at the intermediate nodes as-well. Wire delay is encountered on each channel traversed on route to the destination.

Statement 2 *Wire delay (t_{wire}) is made up of not only the propagation delay (t_{prop}) at each stage but also the delays through the sending (t_{send}) and receiving circuitry (t_{recv}).*

Assuming that in current technology, a single router is implemented on a single chip, then t_{send} and t_{recv} represent the driving and receiving functionality of the output pad of the sending and input pad of the receiving routers respectively.

With present technology and for the foreseeable future, node delay is expected to dominate wire delay [Aga91]. However, as technological advances give rise to decreased feature size with a proportional increase in transistor switching speeds, this will result in an increase in wire delay in relation to node delay. As stated, part of the wire delay includes the driving and receiving circuitry of the pads. As a router incorporates both the logic which effects the node delay and the pads which effect the sending and receiving delays then these latter delays are also effected by increases in switching speeds. However, the scaling of the sending and receiving circuitry will be limited by the speed of the communication medium i.e. a signal cannot be driven onto a wire faster than the wire can transmit it. Thus improvements in the implementation technology with respect to the communication medium will increases the importance of the sending and receiving delay to those of the node delay[MT90].

3.3.2 Network Latency in Wormhole Routed Networks

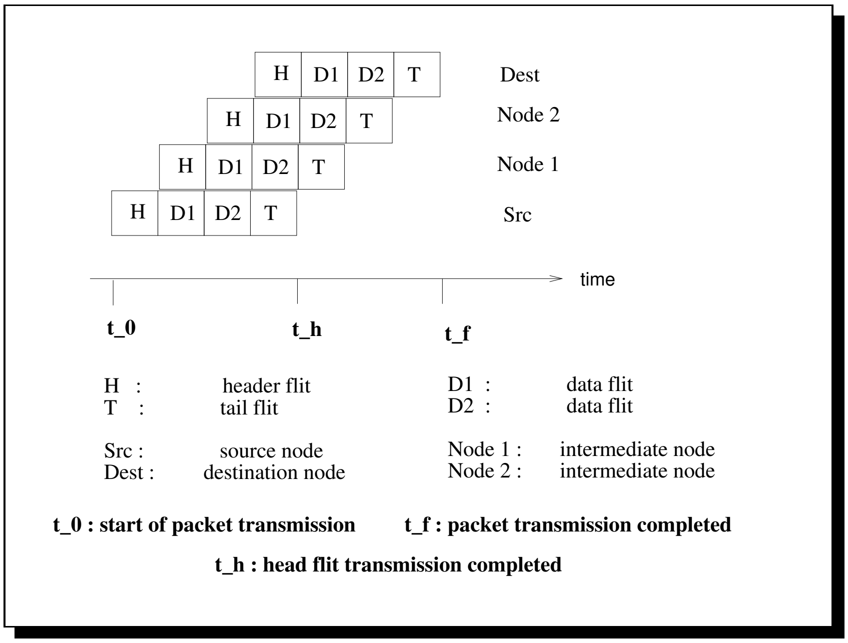

Figure 3.2: Wormhole Routing - 4 Flit Packet

Figure 3.2 presents a simplified picture of wormhole routing. The figure assumes that each flit of a packet encounters the same delay at each node and that no additional delays are encountered due to handshaking in asynchronous communication. It should also be noted that the use of a fixed delay at each node indicates either the case of an unloaded network or that of a loaded network where a packet has not encountered contention. In a network where a packet may encounter contention then the delay encountered at each node depends on the amount of queueing at the respective node. In addition, wiring delay is not illustrated. However, the figure is used to illustrate the 'pipelined effect' of wormhole routed packet transmission. As shown, when the first flit reaches the destination i.e. 'H' at 'Dest', then the flits of the body are spread across the network one at each of the previous nodes up to the length of the packet. The delay experienced by these flits is only seen at the destination as shown by the difference between **t_f** and **t_h** of three times the flit delays i.e. the delay of the body at the destination.

Statement 3 *In wormhole routing, the first flit encounters a delay crossing each router (t_{node}), and each channel (t_{wire}) whereas the delay of each of the following flits is that of the final hop.*

Assuming single flit buffers and pipelined transmission, when the head flit is being processed at the destination, the first flit of the body is at the neighbouring node. When the local processor accepts the head flit, the next flit may be forwarded from the previous node. This process continues until all flits have arrived. That is that the delays experienced by the body flits at earlier nodes, are hidden by the delay of the header flit. This is illustrated in figure 3.2.

Statement 4 *The node delay encountered by the head of a packet includes decision delays, not encountered by the flits of a packet.*

There is an important distinction between the node delay encountered by the header (t_{node}) and that encountered by each of the remaining flits (t_{prop}). Since the decision delay is only encountered by the header then the delay encountered by the header is greater than that of the other flits [BP93]. It can be shown [Dal90a] that the difference in size of the two delays, t_{node} and t_{prop}, can have a large effect on performance in high diameter networks. Since pipelined transmission is used in wormhole routing, the delay of the flits is hidden in all but the final hop. As such, the effect of the difference between these two delays is not seen at the intermediate nodes in latency analysis. However, it is worth noting that the variation in delay for each flit at each node will effect contention at each node on the path to the destination and thus throughput. The effect is, off course, dependent on the difference in size of the respective delays. However, for long messages the effect may be quite significant.

It is chosen, therefore, to split communication delay into two parts representing the total delay experienced by the header and that experienced by the body:

$$t_{comm} = t_{header} + t_{body} \tag{3.3}$$

where

t_{header}	— total delay encountered by the header flits
t_{body}	— total delay encountered by the rest of the packet

In figure 3.1 t_{comm} is shown to be made up of a number of elements. This section builds up expressions for t_{head} and t_{body} in terms of of the elements given. It should be noted that the elements of each of these expressions are subsets but not distinct subsets of the elements given.

Statement 5 *The delays incurred at the source and destination nodes are not necessarily equivalent to the delays incurred at the intermediate nodes.*

An important field of ongoing research, as stated in section 1.1, is work towards solving the *communication bottleneck* problem. A large part of which is acknowledged to come from the software level. One aim is to move some of the software functionality into hardware at the router. Currently, the processor deals with generation and receipt of data to and from the network. Movement of data within the network is, in general, controlled by the routers without interference from the intermediate processors. Thus although implementation of functionality, currently handled by the local processor, may effect intermediate nodes it is to be accepted that functions such as packetising may effect the delay encountered only at the source and destination nodes.

In arbitration for resources, one method to ease the bottleneck within the network is to prioritise traffic leaving the network, that is prioritise the path to the local processor over those to other nodes. As such, the arbitration delay will be shortest at the destination node.

These factors support separation of the delays experienced at the source and the destination nodes from those experienced at the intermediate nodes if either of the above effects are implemented in a router. If this is not the case then a simplified analysis may be found by assigning t_{src} and t_{dest} to be equivalent to t_{node}.

Statement 6 *The final hop seen by a flit includes not only the wire delay but the delay at the final router.*

In latency analysis it is often assumed that this final hop includes only the channel delay. However, an incoming flit has to be sampled from the network channel and a decision made to send it to the processor, before it may be added to the processors incoming queue. That is that the next flit waits until the previous flit completes this process before being forwarded to and processed at the destination node. This is again illustrated in figure 3.2 where each box may be assumed to represent the delay at a node.

Statement 7 *The header of a packet may consist of more than one flit. The additional flits of the header contribute* $(h-1)*(t_{wire}+t_{node})$ *to the single flit header delay.*

Due to pipelined transmission, the delay of each flit following the first flit of the header is only seen at the final hop. As header flits may encounter decision and queueing delays, then the remaining header flits also encounter t_{node} traversing a router.

From these statements the base equation for Network latency in a wormhole routed network is defined.

Definition 1 *Network Latency with respect to wormhole routed networks in an unloaded network is expressed as:*

$$Network\_latency\_WH = t_{inj} + t_{header\_WH} + t_{body\_WH} + t_{cons} \qquad (3.4)$$

$$t_{header\_WH} = \frac{F}{W} * (t_{head\_first\_WH} + t_{head\_rest\_WH}) \qquad (3.5)$$

$$t_{head\_first\_WH} = t_{src} + ((D-1) * t_{node}) + t_{dest} + (D * t_{wire}) \tag{3.6}$$

$$t_{head\_rest\_WH} = (h-1) * (t_{wire} + t_{node}) \tag{3.7}$$

$$t_{body\_WH} = (L-h) * \frac{F}{W} * (t_{wire} + t_{prop}) \tag{3.8}$$

$$t_{wire} = t_{send} + t_{phy} + t_{recv} \tag{3.9}$$

where

t_{inj}	— injection channel delay
t_{cons}	— consumption channel delay
D	— network diameter
h	— header size (flits)
F	— flit size (bits)
W	— channel width (bits)
t_{src}	— router delay at source (head flit)
t_{node}	— router delay at intermediate node (head flit)
t_{dest}	— router delay at destination (head flit)
t_{wire}	— wire delay
L	— packet length (flits)
t_{prop}	— router delay (body flit)
t_{send}	— signal driver delay
t_{phy}	— wire propagation delay
t_{recv}	— signal receiver delay

3.3.3 Expanding Network Latency Analysis

In this section, the expressions derived for network latency in terms of wormhole routing are used to derive expressions for other switching techniques — store and forward and virtual-cut-through routing, and for circuit switching. It should be noted that an expression of network latency in terms of injection, communication and consumption delays is assumed as given in equation 3.4. In addition, wire delay as expressed in equation 3.9, is not effected by changes in the switching technique or switching method and therefore is applicable to the equations of this section.

3.3.3.1 Store and Forward Routing

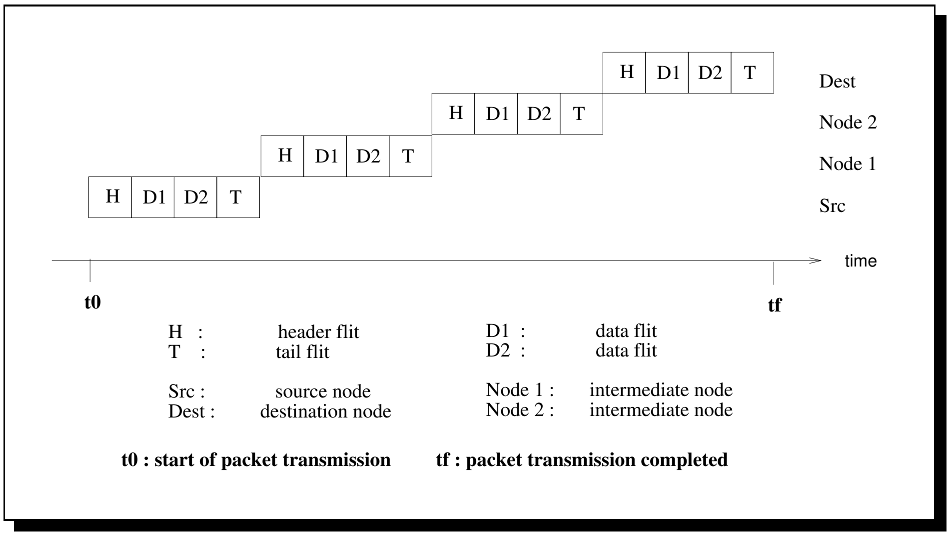

Figure 3.3: Store and Forward Routing - 4 Flit Packet

In store and forward routing the header is delayed at each node until all flits of the packet are received, as shown in figure 3.3. This means that the delay of each flit is seen at each node and on each wire. However, the delay of the flits of the body do not include decision delays and therefore it is not sufficient to multiply the delay seen by a head flit by the length of the packet. Network latency in store and forward routing may therefore be expressed as in wormhole routing, equation 3.4, replacing t_{header_WH} and t_{body_WH} with those of t_{header_SF} and t_{body_SF} shown in equations 3.10 and 3.11. It should be noted that the first term of t_{body_SF} represents the delay at the source node seen by the flits of the body and the second term represents the delay in the hops to and including the destination.

$$t_{header\_SF} = (h * \frac{F}{W}) * t_{head\_first\_WH} \tag{3.10}$$

$$t_{body\_SF} = ((L - h) * \frac{F}{W} * t_{prop}) + (D * t_{body\_WH}) \tag{3.11}$$

3.3.3.2 Virtual-Cut-Through Routing

In virtual-cut-through routing, in the absence of competing traffic, a packet flows through a network in a pipelined fashion as in wormhole routing. However, in the presence of traffic, contention may arise and virtual-cut-through uses a buffering solution, similar to the store and forward case except that the number of flits buffered depends on the length of time before contention is released. That is that the head flit may continue towards it's destination as soon as resources are freed, without waiting on the remaining flits of the packet to arrive at the node. As such, in a loaded network virtual cut through in the best case, behaves as in wormhole routing and in the worst case like store and forward routing.

As such, for an unloaded network the best case is applicable:

$$t_{header\_VCT} = t_{header\_WH} \tag{3.12}$$

$$t_{body\_VCT} = t_{body\_WH} \tag{3.13}$$

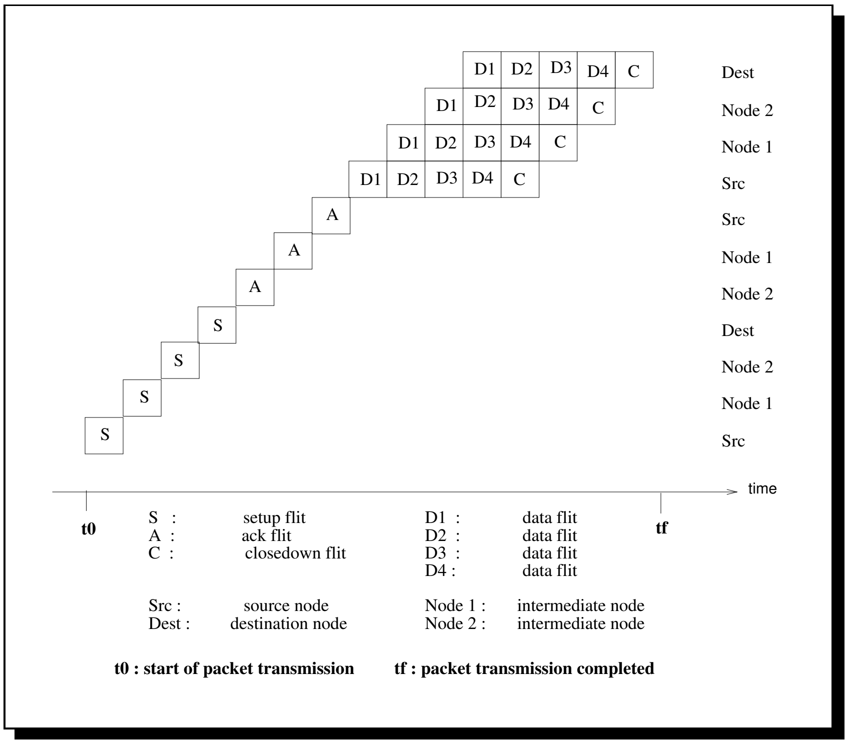

Figure 3.4: Circuit Switching - 4 Flit Message

3.3.3.3 Circuit Switching

In circuit switching a message is not packetised i.e. not split into packets with a header on each packet. However, two control packets are required for set-up and close down of the circuit. The setup packet flows through the network creating a route similar to the flow of the header in packet switching. When the route is complete, an acknowledgement is returned to the source so that it knows that the setup phase is complete, as shown in figure 3.4. Acknowledgements are included in this analysis as this form of acknowledgement is independent of channel control but are part of the switching method itself.

The rest of the message can then flow through the network in a pipelined fashion. The first flit of the message encounters a delay at each router and channel on route to the destination whereas the delay of the remaining flits is only seen at the final hop. This gives a combined expression similar to the expressions in wormhole routing for the header and the body together. However, t_{prop} is seen at each node by the leading flit of the message instead of t_{node}.

The closedown packet follows the last flit of the message as it progresses across the network.

Assuming that freeing of resources takes less time than decision for resources, it is assumed that the delay encountered at a node by the close-down packet is equivalent to that seen by a body flit (t_{prop}).

As such we have:

$$t_{setup\_CS} = t_{header\_WH} + \frac{F}{W} * ((D * t_{prop}) + ((D-1) * t_{wire})) \tag{3.14}$$

$$t_{close\_CS} = \frac{F}{W} * (t_{wire} + t_{prop}) \tag{3.15}$$

$$t_{body\_CS} = \frac{F}{W} * (((D+1) * t_{prop}) + (D * t_{wire})) + t_{body\_WH} \tag{3.16}$$

Note that the first flit of the body encounters a delay at each node, as shown in the first term of equation 3.16. As shown in the second term of the equation, the effect of the remaining flits is only seen at the final node. It should be noted that in t_{header_WH}, h is the number of flits in the setup packet. In t_{body_WH}, L in this case refers to the length of the message i.e. the body and h should be set to 1 reflecting the first flit sent in the first term.

3.3.4 Summary

Figure 3.5 summarises the interrelationship between the different elements of network latency. In section 3.3.1, network latency was expressed in terms of the injection, consumption and communication delays — as shown in the first layer of the hierarchical diagram. Communication delay is then split into the

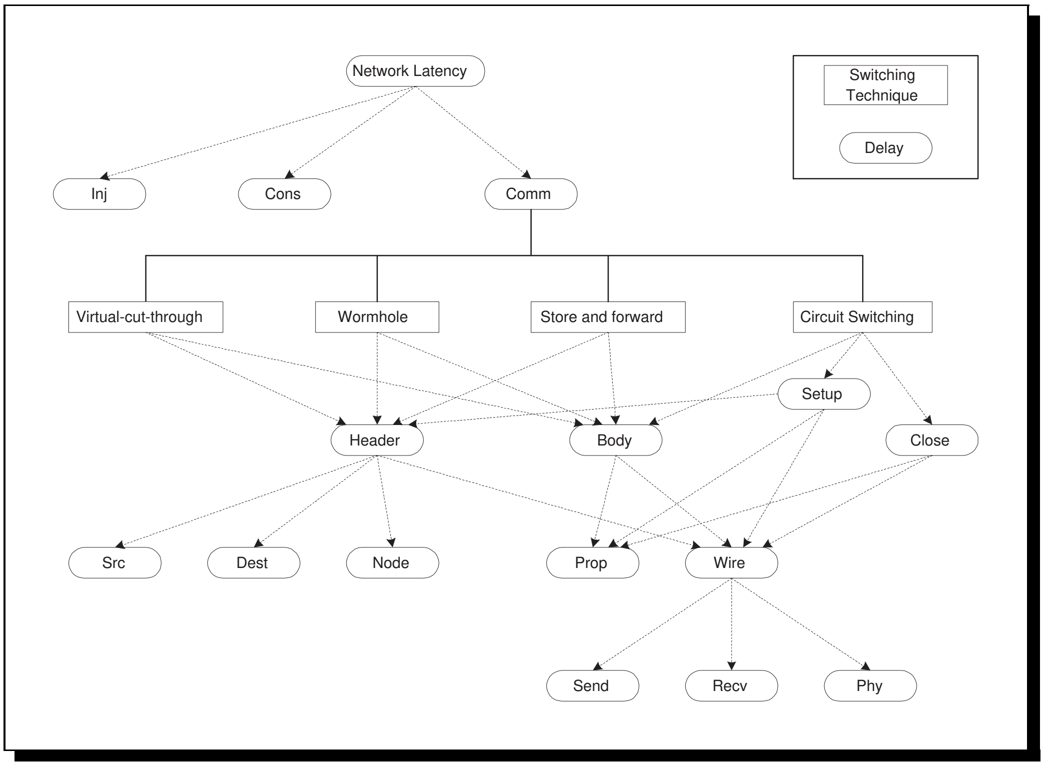

Figure 3.5: Latency in a Communication System

categories relating to the different switching techniques and methods. Each of these categories were then expressed in terms of the header and the body, as shown. In turn, the header and the body were refined to show the types of delay that the header can experience, that which the body can experience and the components of the wire delay that both the header and the flits of the body experience.

It should be noted that two additional delays are expressed under circuit switching, the setup and close delay reflecting the delay of the setup and close-down packets respectively. As shown, setup delay is expressed in terms of the header delay plus additional delays due to the packet acknowledgement. The body of the packet is expressed in terms of the propagation delay and wire delay as in the packet switching cases and the close-down packet is expressed in terms of the propagation delay and wire delay.

Considering the hierarchy expressed in figure 3.5 as analogous to a tree then the leaves of the tree may be said to represent the elements of network latency shown in figure 3.1. Further refinement of these delays is dependent on a given router instance and a given implementation technology and communication medium. Chapter 4 builds up a set of generic modules from which a design instance may be created. The design instance depends on the modules chosen, their interconnection and the designs chosen for the individual modules. From this information more detailed delay information may be obtained for the elements of Latency given in figure 3.5. The creation of designs and extraction of performance information is described in chapter 5.

Part II

The Framework

Chapter 4

Generic Router Modules

In chapter 2 the issues in communication hardware design, as well as the features of a range of routers, were identified. Derived from these facts, the set of router modules was identified and are presented in this chapter. This module set is a feature of the framework. A given subset of this set of modules may be interconnected to form a router register-transfer level description. Furthermore, the work of chapter 2 was used to identify design issues which effect the design and placement of the individual modules within the given router. These issues are also presented in this chapter.

Section 4.1 identifies the set of modules and explains the functions that each module performs within the router. In section 4.2, an analysis of each module is presented which considers the effect of different design issues on the complexity of the module. Design issue effects are further considered in section 4.2.2 considering both placement of a module on a path and the number of instances of a module required in a design. Section 4.3 provides examples of modular router descriptions of existing routers using the modules of the module set.

4.1 Identification of the Modules in a Hardware Router

The two switching methods considered in this work are packet switching and circuit switching. In packet switching, as described in section 2.3.1, each packet creates it's own route across a network independent of the other packets of the message.

The main sections of a packet are the header — one or more flits, and the body — zero or more flits. To signal the end of a packet, either a tail flit or a tail bit within each flit is generally used. Another method is to incorporate the *packet length* (number of flits) within the header. The header holds the addressing and control information for the packet. In circuit switching on the other hand, a *set-up* packet creates a route for a message and then the complete message is sent across the reserved route. A *close-down* packet is then used, similar to the tail, to remove the path. As such, both the header, in packet switching, and the setup packet in circuit switching contain control and addressing information required by the routers on their path.

The three address modules listed in table 4.2— comparator (**AC**), decrementer (**AD**) and converter (**AX**) are all interested in the address information included in the incoming packet. All routers require some form of comparator to make a decision based on the destination address of the incoming packet and the *local address* — the nodes unique address within the network.

If a router uses *relative addressing* i.e. the address expresses the number of hops remaining in each dimension, then the comparator is in fact a zero-checker. If the destination has been reached then each component of the address i.e. the address in each dimension, is zero. As such, this tells the routing decision module to forward the packet to the local processor. If the destination has not been reached then the zero-checker will give a negative result indicating to the routing decision module that an output channel selection must be made. When forwarding, the address must be updated so that the address shows the correct remaining number of hops to the destination. This is the job of the address decrementer.

In the case of *absolute addressing* where the address represents the actual address of the destination node, the comparator compares the actual address of the destination against the actual address of the local node. If a match is found then the destination is reached and the packet may be forwarded to the local processor. Otherwise a decision regarding the output choice is made. No address decrementer is required in this case. However, an address converter may be used to produce a relative address for routing decision purposes, if required.

In section 2.4.6, buffering within a router is discussed. As stated, size, placement and type of buffering varies between different routers and is dependent on various routing decisions. Different types of buffers may also be required if more than one type of communication protocol is allowed within the same router i.e. different buffering for long and short packets. However, some form of buffering is required in all routers to handle congestion and decision making delays. In routers using virtual channels, buffering for these channels is included in the virtual channel module (**VC**) whereas other types of buffering are included in the buffer (**B**) module.

One of the main functions of the router, identified in section 2.2 is switching. This is the responsibility of the crossbar module (**CB**). Crossbars switch data from a chosen input to a chosen output. The choice of input and output is controlled by the arbiter (**AR**). Arbiter selection is based on the requests received, their respective output selection and on the arbitration method itself — see section 4.2.1.3.

Routing, another of the main functions is primarily handled by the routing decision module (**RD**). This module implements the routing algorithm. Different algorithms may be run in parallel as in the case of the Reliable Router so that various selections are available at the same time — see section 6.3.2.7. The results from either the comparator or address converter provide information regarding the packet destination. Other control inputs received in the incoming flit may also be used in the decision process. Information such as packet priority, buffer availability at neighbouring nodes or traffic details (in the case of adaptive routing) may be required.

Buffer availability, is either handled within the RD module — in the case of a full signal, or handled by the flow control unit (**FC**) which passes the required information to the RD module. Once the routing decision is made, the result is stored so that it may be used for the remainder of the packet. It is also forwarded to the arbiter(s) and any other modules awaiting the result. One of these waiting modules is the address decrementer, in the case of relative addressing. In the case of adaptive routing using relative addressing, the header will need to be updated to reflect the direction taken. As this is not a simple decrement operation, this function is handled separately by a module named the flit update (**FU**) module.

In the case of circuit switching, the chosen channel is reserved until the close down packet is received. The channel update module controls channel status and is updated by the routing decision module.

If virtual channel deadlock prevention is employed in the router then control of data movement between the physical channels and the virtual channels is controlled by the virtual channel controller (**VC**). The virtual channel controller includes both buffering for each of the virtual channels and a multiplexer and controller (arbiter) for switching these virtual channels onto a single physical channel. It should be noted that the functions required to handle virtual channels are collected together as a single unit i.e. module, in the module set.

In the case of bidirectional channels, an output module is required to arbitrate between *neighbouring nodes* i.e. nodes which are directly connected by the network, for the use of the channel. The output module (**OP**) may also generate a request signal as a data valid signal to validate the output data reaching the neighbouring node.

Control of both external channels and internal modules depends on the choice of synchronous or asynchronous control. In the synchronous case, the synchroniser (**SY**) synchronises incoming data to the local clock. This may be required in the case of a network where routers are operated synchronously and channels asynchronous. In addition, it is required in a synchronous system where there is no global clock. That is that each router operates at the speed of it's own local clock.

In the case of an asynchronous router, a signal manager (**SM**) is required to control both external and internal signals. The two main methods for asynchronous control in networks are *request/acknowledge signalling* and *control code signalling*. There are different methods of request/acknowledge signalling. If it is the case that a different method is implemented internally, within the router, from that used externally, then a signal converter is re-

Function	**Components**
Processor Interface	IJ, CC
Switching	AR, CB
Routing	AC, AD, AX, CU, FU, RD
Flow Control	MB, MS, VC, SM, FC, B
Network Interface	SC, SY, OP

Table 4.1: Functionality of the Router Modules

quired (**SC**).

The two modules, message split (**MS**) and message build (**MB**) are, if required, usually seen at the input and output of a router respectively. Message split splits an incoming packet into flits destined for the control path (header/tail) and flits destined for the data path. There are two methods of splitting a packet between these paths. The first sends the header/tail flits on the control path along with a copy of the respective flits on the data path. The second is similar, but no copy of the control information is sent on the data path. This means that the packet has to be constructed before being sent. This is the job of the message build module.

The two final modules, required in all routers are the input and output queueing between the routers and their local processors. These are termed the injection (**IJ**) and consumption (**CC**) modules respectively.

Table 4.1 lists the identified modules in terms of the router structure identified in section 2.2 and a summary of the module set is provided in table 4.2.

Symbol	Component	Description
AC	Address Comparator	compares address or zero checker
AD	Address Decrementer	decrements relative address
AR	Arbiter	arbitrates between crossbar requests
AX	Address Converter	absolute to relative address
B	Buffer	buffering
CB	Crossbar	switching
CC	Consumption	queueing out of the network, to processor
CU	Channel Update	updates channel status (circuit switching)
FC	Flow Control	records buffer space at neighbouring nodes
FU	Flit Update	creates or alters flit
IJ	Injection	queueing into the network,from processor
MB	Message Build	rebuilds packet or flit
MS	Message Split	splits packets flits into smaller units
OP	Output	arbitration for bidirectional channels
RD	Routing Decision	makes routing decision
SC	Signal Converter	conversion between different phase signalling
SM	Signal Manager	asynchronous controller
SY	Synchroniser	synchronises incoming data to local clock
VC	Virtual Channel Controller	virtual channel buffering, multiplexing, arbitration

Table 4.2: The Set of Router Modules

4.2 Analysis of the Modules

The selection of the required modules, the complexity of the individual modules and the method of connecting them is dependent on the design choices made for the router. Router design involves a large number of design issue decisions, some of which are interdependent. A number of these issues were identified and described in section 2.4.

There are two important issues with respect to further analysis of the modules. The complexity of a module effects the cost and delay of the particular module and thus the overall cost and delay of the router. It is often the case that only one unit of a given module lies on the longest path through the router and thus only one unit of the module effects the longest path delay and thus latency within the network. However, the number of units of the module required within the router design effects the cost of the router and is therefore an important consideration. Placement of a module may also effect router performance and cost.

4.2.1 Design Issue Effects on Module Complexity

In this section, the discussion given in section 4.1 is expanded to a more detailed discussion of the design decisions which effect the individual module's complexity. This discussion thus identifies the set of design parameters, representing design issues, which are required within the framework. A subset of these design parameters are identified to effect an individual module. At the end of this section table 4.3 presents a summary of the design issues effecting complexity and identifies those modules effected by a particular design decision.

4.2.1.1 Address Comparator (AC)

The purpose of the address comparator is to make a decision based on incoming data. This decision is based on addressing information in the header. In the case of a one flit header, the final decision is made after the single header flit has been processed. In a multiple flit header, the number of flits is dependent on the degree of the network i.e. an address flit for each dimension. A decision is given after each flit is processed. It is up to the module receiving this information from the AC to determine whether a final decision has been made or not. In general, this is the responsibility of the routing decision module. As such, the minimal number of flits that must be processed by the comparator is equal to the number of flits in the header.

Implementation of the AC is dependent on the addressing method chosen, the flit size and the packet format.

4.2.1.2 Address Decrementer (AD)

An address decrementer is required for deterministic routers that use relative addressing. It's function is not to make a decision, but based on an incoming decision — whether to decrement or not, the incoming data is either decremented by one or forwarded unchanged. That is that it awaits a decision, usually from the routing decision module before processing the input data.

Similar to the comparator, if multiple flits are used to represent the header then for each flit received, the data is either forwarded unchanged or decremented depending on the decision received. That is that the decrement process is applied to the flit. However, if a single flit header is used represent more than one dimension, then the decrementer must decrement that part of the flit which represents the address in the dimension to be decremented. One possible implementation is to create a number of sub-modules each handling the decrementer functionality for a given dimension i.e. ADx and ADy, and each driven by a separate decision signal from the routing decision module. Note that each submodule's data path is equivalent to the address width for each dimension. This implementation retains the functionality of the decrementer and moves any additional complexity to the routing decision module.

The above discussion focuses on flit size and address size in a given dimension. As such, the decrementer is dependent on the flit size and the packet format. Although implementation of the decrementer may include fast circuitry to avoid delay proportional to the flit size e.g. manchester carry chain using carry look-ahead circuitry, the complexity of the router is still dependent on the flit size although it should not be assumed to be a direct-dependence.

4.2.1.3 Arbiter (AR)

There may be more than one *arbitration level* within the router i.e. more than one stage where a resource decision is made. The first arbitration level — termed AR in table 4.2, arbitrates between external requests for output channels on a packet basis i.e. it controls the switching of the crossbar. As such, this type of arbitration is required in all routers. A second level arbitrates between the virtual channels of a physical channel. This arbiter is found within the virtual channel controller module and controls the virtual channel multiplexer — termed a multiplexer controller. The third level occurs in the case where bidirectional channels are implemented as a single physical channel. Arbitration between the neighbouring nodes is required to control access to the channel. This arbitration level is implemented in the output module.

Each of the arbitration levels may be implemented using a different arbitration method. That is that the method that best suits a particular resource may be used. In addition, the switching technique effects the arbiter. When an arbiter makes a decision, it makes the decision based on the arriving requests. If the unit of flow control within the router is the flit i.e. wormhole or virtual-cut-through routing, then it makes a new decision for each request on a flit basis. However, since each flit of a packet must continue on the same route then that route must be reserved such that if another request wishes to send on this reserved route that the request is not granted. This is the case for the crossbar arbiter. For the VC arbiter, the resource under arbitration is the physical channel. The structure of a packet is preserved by separating the flits transferred over a physical channel into queues representing the different virtual channels. As such, reservation of a physical channel for the length of a packet is not required allowing packets from different virtual channels to be interleaved.

In the case of store and forward routing, the flow control unit is the packet and as such, the arbitration decision is made on a packet basis. That is that no other requests are served until the whole packet has been handled. This removes the extra logic in the arbiter needed to both reserve channels and to check for reservation when a new requester requests a given output.

Another feature effecting the arbiter is the choice of asynchronous or synchronous control. Although asynchronous control increases the complexity of most modules, the effect on the arbiter is more pronounced. In a synchronous arbiter, an arbitration decision is based on those requests waiting at the beginning of a new clock cycle. It may be assumed that the crossbar completes the transfer and the requester clears the request within a clock cycle and therefore a new transfer can begin at the next clock cycle.

However, in the asynchronous case, the requests waiting may be handled as soon as the previous decision has been handled and acknowledged. Since the arrival time of different requests is unknown and the time at which a given decision will be made is also unknown this makes the asynchronous design more complicated. That is that all requests arriving must be registered and a request already handled will most likely not be cleared and as such will be available in the request list at the beginning of a new arbitration round. It is therefore important to differentiate between handled and not handled requests. Also, an acknowledgement from the crossbar is required to confirm that the previous selection has been transfered before a new decision is given to the crossbar. If it is assumed that transfer across the crossbar takes less time than the arbiter decision then this feature may be ignored. However, a problem in asynchronous design is to ensure that signals behave in the manner expected, making assumptions may lead to problems in timing and thus it is a safer measure to await an acknowledgement before clearing the previous selection.

The number of competing inputs to the arbiter also effects it's complexity. The number of inputs depends on the number of virtual channels and the degree of the network. Partitioning of the crossbar also gives rise to reduced sized arbiters i.e. less inputs. Another feature effecting the arbiter complexity through it's inputs is the coding of the inputs requests i.e. the width of the input request channels and the effect that this coding has on the realisation of the arbitration method.

4.2.1.4 Address Converter (AX)

The address converter converts an absolute address to a relative address. This may be used in adaptive routers which use absolute addressing to avoid the need to update the header both at the start of transmission and at each node. Conversion to relative addressing may help simplify the routing decision module.

All flits in the header encounter the converter. As in the decrementer, for the case of a single flit header, sub-modules of the address converter may be used to generate relative addresses in each dimension.

A typical implementation is to XOR the destination address with the local address and this is off course dependent on the size of the address. Therefore the converter is dependent on the flit size, and communication protocol.

4.2.1.5 Buffering (B)

There are two main types of buffering within a router. The first represents queueing for a shared resource and the second is a simple information store.

A form for queueing must be available on the data path. Queueing is, in general, implemented as a FIFO. However, it may be the case, in some forms of adaptive routing, that access to flits, other than the first flit in a buffer is necessary. The width of a FIFO is equivalent to the flit size and the minimum depth of the FIFO is dependent on the switching technique. Store and forward or virtual-cut-through routing require a minimum of a packet sized buffer on any path. Thus the buffer size is dependent on the packet size i.e. communication protocol. This also indicates that for a router design using these switching techniques either a fixed packet size or a maximum packet size should be chosen such that the minimum buffer size is equivalent to the maximum packet size. In the case of wormhole routing the minimum buffer size is a single flit. In both cases the buffer size chosen is dependent on the minimum buffer size and expected traffic flow and contention within the network. Traffic flow is application dependent and variations in applications requirements means that there is no single correct buffer size.

Information storage may be accomplished by a latch. When a routing decision is made, based on the information in the header flits, this information must be stored so as to be available to the flits of the body of the packet. This is particularly important in the case of wormhole routing where the head of a packet may not be available at a given node when the flits of a body enter the node. In circuit switching, after a decision is made based on the setup packet, the route must be stored for the length of the message, in other words until a close-down packet is received. This information is again stored in a buffer.

Different forms of buffering for different packet types may also be found. An example of this is splitting buffering for short and long messages for networks designed to handle bimodal traffic more efficiently.

4.2.1.6 Crossbar (CB)

The switching functionality of a router is usually achieved by a crossbar and it's associated arbiter. In general, for an $n \times n$ crossbar where there are n input and n output ports, 2n buses are required for the input and output data and n^2 crossing points for switching between these buses. Control of these crossing points depends on the arbitration method and is therefore left to the arbiter module. A one-to-one mapping is the only mapping available across a crossbar switch [Hwa93]. For an $n \times n$ crossbar switch we have **n!** possible permutations of the mapping.

In a central crossbar, the number of ports is dependent on the degree of the network, number of virtual channels and virtual channel multiplexing. Three different configurations for virtual channels were proposed by Dally [Dal92] for a centralised crossbar. The different configurations effect the complexity of both the CB and AR modules. In the first case, virtual channels are multiplexed after the crossbar, thus giving rise to a larger crossbar. In the second case, the virtual channels are multiplexed before the crossbar thus reducing the number of inputs to the crossbar and hence it's complexity. In the third case, the number of inputs to the crossbar is equivalent to the number of virtual channels whereas the number of outputs is equivalent to the number of physical channels. This case gives rise to an improvement in the complexity of the crossbar over the first case and gives rise to simpler arbitration than in the second.

It should be noted that a crossbar may also be partitioned, effecting the size and thus the complexity of the individual crossbars. CB complexity is dependent on both the number of ports and *request coding* — coding of the control signals from the arbiter module.

4.2.1.7 Consumption Channel (CC)

The consumption channel is a queue and as such it's size is dependent on the flit size and chosen buffer size. The buffer size chosen depends on the buffering requirements which are dependent on the difference in the rate that the network delivers packets to that that the processor accepts them. Thus the size issue is traffic dependent.

4.2.1.8 Channel Update (CU)

The channel update module holds the channel status in the form of a finite state machine. Channels may be active, free or in some cases cached. The cached state is used in switching techniques which reserve the channel after a packet has been sent so that it may be reused by the other packets of a message or torn down if another packet from another message requires the channel. If the channel status is restricted to active or free then a simple buffer is usually used to hold the status of the channel in use and therefore a channel update module is not required.

4.2.1.9 Flow Control (FC)

In this case flow control involves a counter to count the remaining free spaces in the queue at the receiver, that is incrementing each time a flit is buffered and decrementing each time the receiver forwards a flit to another node. As such, this process involves communication with the relevant neighbouring node. On the other hand, flow control may consist of a control bit which indicates if the receiving buffer is full or not. If full, flits may not be sent and therefore the routing module waits on the full signal to clear before sending the flit to the buffer. As this latter process involves waiting on a signal then it's functionality may be implemented within the routing decision module.

A counter is required for each possible receiving buffer, the size of which is equivalent to the buffer size. The number of counters required is dependent on the routing algorithm i.e. the number of possible outgoing paths. Other signals may also add to the complexity of the flow control unit.

4.2.1.10 Flit Update (FU)

In the case of adaptive routing, the FU module is used if a new or altered header is required. It may also be used to alter other flits within a packet if changes to control signals within a flit are to be made. In deterministic routing, FU is not required as the only change applied to the header flit is decrementing which is achieved by the AD module.

The flit update module is similar to the decrementer in that the header flits may be changed with respect to decisions entering the module usually from the routing decision module. However, unlike the address decrementer, the header update module need not necessarily receive a header for updating but may create a header based on the incoming signals and the flit size.

As such, flit update implementation is dependent on the control signals received, the addressing method used, the flit size and the packet format.

4.2.1.11 Injection channel (IJ)

The injection channel is a queue and again it's complexity is dependent on the flit and buffer size. The size of the buffer required is dependent on the difference between the processors injection rate and the networks consumption rate. It is often the case that injection from the processor is prioritised over network traffic within the router such that consumption rate may in fact be equivalent to the injection rate. In this case no injection channel module is required within the router. Thus the arbitration method effects the buffer size required. It should be noted, however, that this does not prevent queueing out with the network i.e. at the software level of the communication system, if the processor generates packets faster than the injection rate. In the case where injection rate exceeds consumption rate then buffering is required at the entrance to the network i.e. an injection channel.

4.2.1.12 Message Build (MB)

Message Build may be thought of as the inverse of message split, that is it reassembles the unit of data split by the message split module. A packet is not necessarily reassembled as a single packet within the node but the MB makes sure that the updated flits of a packet follow the order given in the packet format.

If adaptive routing with absolute addressing is implemented with split data and control paths then the updated header will need to be attached to the data of the packet. Therefore MB is required. On the other hand if relative addressing is used along with path splitting and a copy of the header is placed on the data path then MB is not required. As such, addressing and the routing algorithm effect the requirement for MB but not it's complexity.

The complexity of Message Build depends on the packet format and the flit size.

4.2.1.13 Message Split (MS)

This module is used to divide the flits of a packet into those required on the control path and those that may follow the data path. It should be noted that those flits sent on the control path may also be sent on the data path as described in section 4.1.

The method of message splitting depends on the packet format. One method is to add a control bit(s) to the flit format to indicate which direction a flit should be sent. This may involve a check of these bits in each flit or the relevant bits of the header may indicate how many flits should be directed onto the control path. The former method involves a decision for each flit and the later involves a count of the number of flits sent so far on the control path. Thus the complexity of message split is dependent on the packet format and the flit size. Message split may either split between data and control paths and/or between virtual channel paths. In the case of virtual channels a priority code is required within the control bits to indicate which virtual channel is to receive the flits.

4.2.1.14 Output (OP)

The output module was originally named the output pad. However, basically the function of a pad at the output is driving the data. As such, the output module provides any additional functions not provided by a driver pad. These include generation of any additional signals on sending or arbitration for bidirectional channels. Complexity is therefore dependent on the arbitration method used, the control method and the complexity of any additional functions to be formed. In addition, it is of course dependent on the flit size as it lies on the data path through the router.

One method of arbitration seen in the case of bidirectional channels is token-based arbitration implemented as a single arbitration line. When a router has something to send and holds the token then the data may be forwarded. Otherwise a request is sent to the neighbouring node by driving the arbitration line high. Release of control by the opposite side is given by driving the line low.

Another method is to pipeline arbitration with data transfer. As such, when both routers have data to send, each side will send up to a maximum number of flits and then release control of the channel.

If request/acknowledge signalling is implemented, then it is the job of the output module to generate a request signal on data transfer. Receipt of acknowledgements from the receiving node is handled by the signal manager and forwarded to the output module.

4.2.1.15 Routing Decision (RD)

The routing decision module implements the routing algorithm or algorithms, if necessary, within the router. Once a routing decision is made for a packet, this information is sent to the arbiter. Since all the flits of the packet must follow the same path, this decision is held for each flit of the packet until a tail is registered. At this stage the decision is cancelled and the decision module awaits a new packet.

In deterministic routers, the routing decision gives rise to a single output choice and this decision is predetermined. Based on this decision and the buffering method chosen, the incoming data will either be buffered within the node awaiting sending or sent to the receiving node and buffered there. The former case assumes middle or output buffering whereas the latter case assumes input buffering. In all cases, the routing decision makes not only a choice of route but also implies a choice of buffer since buffers are available for each route. Flow control within the router must make sure that the receiving buffer has room for the incoming flit.

In the former case a simple check of the full signal is required and as such may easily be incorporated in the routing module. In the latter case a more complex flow control mechanism is required and is as such moved to the flow control module.

In adaptive routers, many status signals are required to give updated information from the rest of the network to the router. As such, is not only the algorithm more complex but also more inputs are required, leading to a more complex module.

4.2.1.16 Signal Converter (SC)

A signal conversion module is required in asynchronous routers to convert between two different signalling methods, external and internal i.e. 4-phase to 2-phase or vice versa. This is therefore similar to the synchroniser in the synchronous case which converts between two different clocks, external and internal.

4.2.1.17 Signal Manager (SM)

The signal manager is used in asynchronous routers to control the external signals — communication on the channels, and the internal signals — control of the internal modules. The complexity of the signal manager depends on the type of signalling implemented. The two main methods for asynchronous signalling on the channels are termed request/acknowledge signalling and control code signalling.

Request/acknowledge signalling may involve *2-phase* or *4-phase* signalling. The former requires request and acknowledge lines for each physical channel. If the sender has something to send it toggles the request line. When the receiver is ready to receive further data i.e it acknowledges the previous receipt, it toggles the acknowledge line. The sender may therefore continue to send when the request and acknowledge lines are equivalent. In 4-phase, active high status is used to indicate request or acknowledgement. That is that both lines must clear after a request/acknowledgement cycle.

Another method of controlling request/acknowledge signalling without the use of a signal manager is to incorporate the control functionality within the buffers of the router. That is that the functionality of the buffering module is expanded to include control. This method requires extra buffering in the form of ripple through FIFOs where data and requests ripple forward and acknowledgements ripple backwards. Extra buffering is required to ensure correct timing of the control signals and data as both data and control flow through the router.

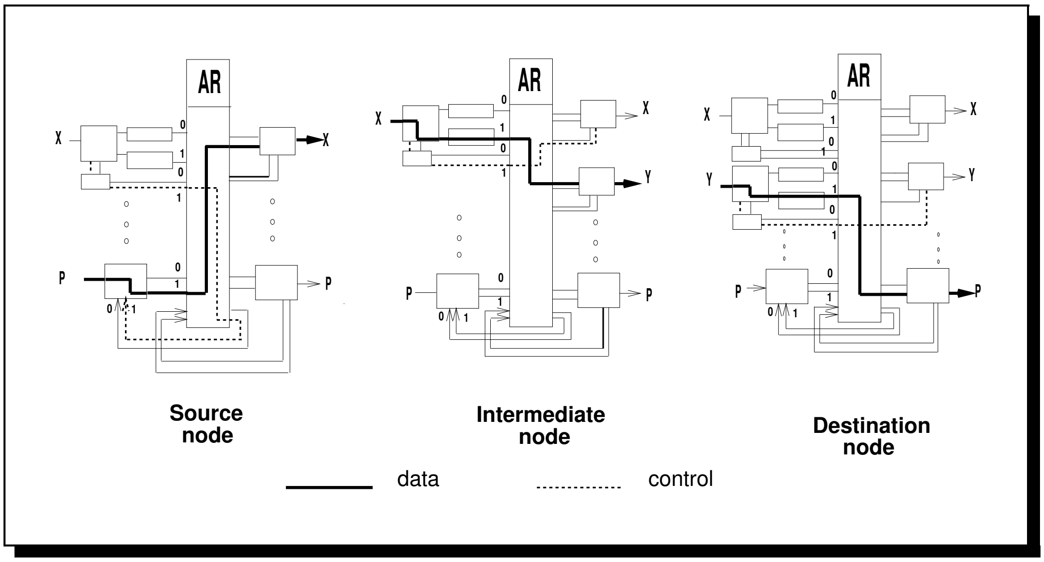

Figure 4.1: The Flow of Data and Control Codes Through a Network

In control code signalling two unidirectional channels between adjacent nodes is assumed. Thus data may be sent from either node to the respective neighbour. Control codes flow in the opposite direction from data and are transmitted in the data packets of the opposite channel, if active i.e. the neighbour has data to send. Otherwise, special control packets are sent.

At the input to the router, control codes are separated from data and sent to the control signal manager. These are used by the control signal manager to signal to the output buffer that the outgoing channel is available i.e. the neighbour is ready to receive more data. On the other hand, if there is free space in the router's outgoing buffer, it will insert a ready control code into the data channel leading to the previous router. Otherwise, where the buffer is full, a not-ready control code is sent.

Figure 4.1, illustrates the case of a packet flowing from source to destination. The bold lines illustrate the flow of data at each stage through the crossbar and out on the channel towards the destination or to the local processor as the case may be. The dashed lines illustrate the control code flow. It should be noticed that the control codes flow through the crossbar but do not follow the path of the data. Instead, they proceed to the outgoing channel feeding into the previous router on the routed path. Also, it may be noted that control codes heading for the processor cause a feedback on the exit of the crossbar. This is due to the fact that input to the processor is at the output of the crossbar and vice versa. Note that the control flow illustrated at the source node occurs after the intermediate node has received the incoming flit. When space is available at the intermediate node to receive further data, a ready control code is switched across the crossbar to the direction of incoming data.

4.2.1.18 Synchroniser (SY)

A synchroniser synchronises data sent on the transmission clock to the local clock before being sent to the other modules of the router. As such, it lies on the data path resulting in a complexity depending on the flit size and synchronisation implementation.

4.2.1.19 Virtual Channel Controller (VC)

The Virtual Channel Controller module includes both queueing for and multiplexing between the virtual channels of a physical channel. It's complexity depends on the number of virtual channels sharing a physical channel, the size of the buffers required and the method of arbitration used in the multiplexer controller. It is often the case that fair arbitration is used to arbitrate between virtual channels.

Figure 4.2 shows a possible configuration of a VC module taken from the Torus Routing Chip. The submodules B, MX and CT represent the virtual channel buffers, multiplexer and multiplexer controller respectively.

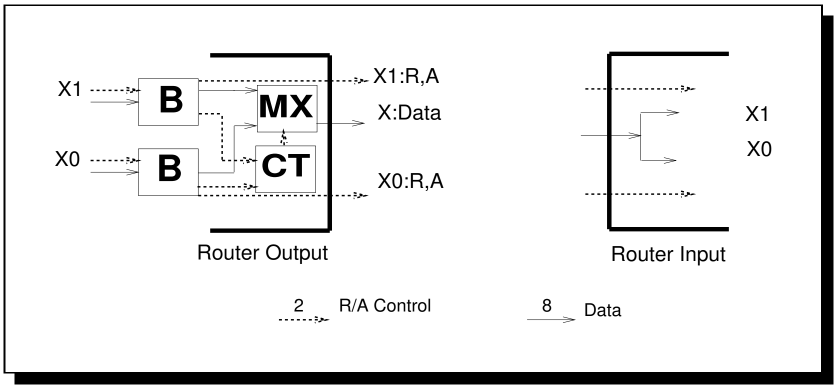

Figure 4.2: A Virtual Channel Control

4.2.1.20 Summary

Table 4.3 summarises the discussion given above in the light of different design decisions. Note that the complexity of all modules are effected by changed in control.

Issues	**Components**																		
	AC	AD	AR	AX	B	CB	CC	CU	FC	FU	IJ	MB	MS	OP	RD	SC	SM	SY	VC
Flit size(bits)	x	x		x	x	x	x			x	x	x	x	x				x	x
Packet Format	x	x		x						x		x	x		x				
Addressing	x									x									
Switching Technique			x		x		x	x	x		x								x
Buffer Size					x		x		x		x								x
Routing Algorithm									x						x				
Arbitration Method			x								x			x					x
Request Coding			x			x													
Control	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Number of VCs			x			x							x		x				x
Degree			x			x									x				
Partitioning			x			x													
Traffic					x		x				x								x

Table 4.3: Design Issues Effecting the Complexity of Modules

4.2.2 Design Issue Effects on the Number and Placement of Modules

The following discussion refers to the paths shown in figure 4.3. The following abbreviations are used in the discussion:

PP:	physical path
VP:	virtual path
CP:	control path
DP:	data path
PCP:	physical control path
PDP:	physical data path
VCP:	virtual control path
VDP:	virtual data path

To be able to consider the number of units of a given module required for a router design, it is important to consider the number of paths through a router. In the simplest case, one path exists for each external input and a crossbar switches between these inputs and an equivalent number of outputs. That is that the number of paths is equivalent to the number of physical input channels. A single physical path is shown in (a).

Since decision making takes longer than data progression within the node, a speed up may be obtained by splitting decision making flits from non decision making flits i.e. to create a control and a data path for each physical path as shown in (b). This allows data flits to avoid decision making modules when traversing a router and as such reduces the delay,t_{prop}, for each body flit.

Path analysis is further complicated by the introduction of virtual channels. Virtual channels increase the number of paths through a router by a factor equivalent to the number of virtual channels. The complete path through the router is not necessarily split into the number of virtual paths. Instead parts will consist of a single physical path and where chosen, split into the number of virtual channels. That is that some modules are placed on the physical path and others on the virtual paths. In (c) a single physical channel is split into 2 virtual channels and the number of crossbar ports is increased by a factor of 2. In (d), the case where multiplexing of virtual channels occurs

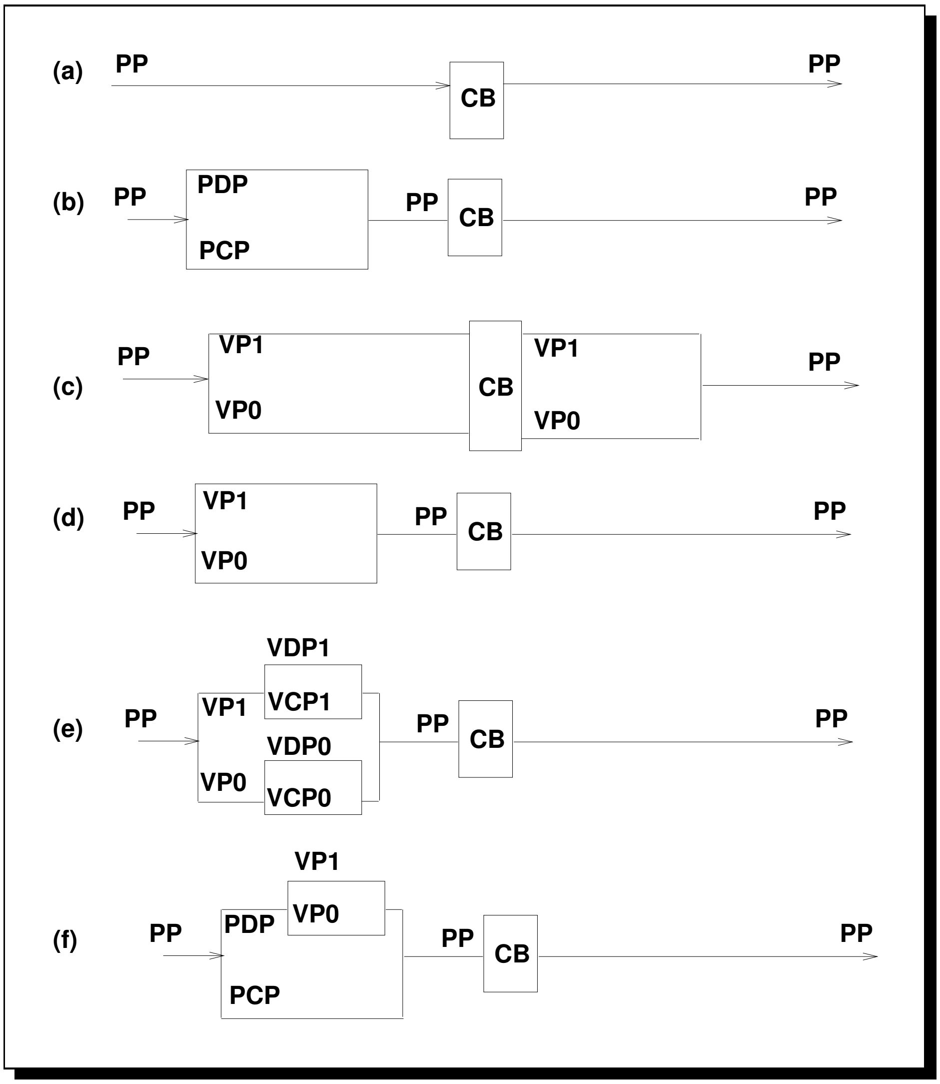

Figure 4.3: Path Variations within a Router

Design Issues	**PP**	**VP**	**PDP**	**PCP**	**VDP**	**VCP**
Degree	x		x	x		
Number of VCs		x			x	x
Path Splitting			x	x	x	x

Table 4.4: Design Issues Effecting Path Choice

before the crossbar is shown, resulting in a decrease in the number of ports required in the crossbar.

Another variation is to allow both virtual channels and path splitting in the same design. An example of this is given in (e) where each virtual channel is split into data and control paths. In the final example given in figure 4.3, path splitting occurs before the data path is split into virtual channels.

Other path variations are possible but these examples are used to highlight the different types of paths within a router. In the following discussion, path analysis is used to illustrate the number of instances of a module required in a given design. In addition, module placement can effect the performance of a router. Therefore, the effect of various module placements is included in the following discussion where relevant.

Table 4.4 illustrates those design issues expressed in section 4.2.1.20 which effect the number of paths in the router. A new design issue is introduced termed path splitting. Path splitting does not effect the complexity of individual modules as it refers to a choice of paths for placement of a given module. This differs from the analysis of section 4.2.1 where a single module may be distributed over a number of virtual channels thus effecting each unit's complexity.

4.2.2.1 Address Comparator

An address comparator is required on each PP. If virtual channels exist then the comparator may either be included on the VP or PP. If path splitting has occurred then the comparator placement depends on the packet format chosen. In the case where a tail flit (a unique data pattern) is added to indicate 'end of packet', the comparator must process each flit of the packet looking for the tail flit. That is that the comparator must be placed on the data path — PDP or VDP. On the other hand, if a control bit(s) is/are used to indicate end of packet then the data and control path may be separated, if required, and the comparator may be placed on either the control path — PCP or VCP, or data path — PDP or VDP. The tail bit(s) are extracted from the incoming flit and sent to the routing decision module instead of through the comparator.

4.2.2.2 Address Decrementer

If absolute addressing is implemented then an address decrementer is not required. However, in the case of relative addressing, a decrementer module is required for every address comparator module.

As the AD receives it's decision signals from the routing decision module, it may be thought that it must be placed after the routing decision module on the path. However, it may also be placed in parallel such that as soon as the routing decision module has completed it's decrement decision the decrementer can begin processing whilst the routing module completes it's processing of the current flit.

If virtual channels are implemented then it is more often the case that the AD lies on the virtual channel path — VP. Assuming path splitting, since the decrementer changes the header then if placed on the control path — PCP or VCP, a message build module will be required to reassemble the packet. Another option is to place the decrementer on the data path — PDP or VDP, such that the updated header will reside at the front of the packet. However, this means that each flit of the packet must pass through the decrementer module adding an additional delay to the flits of the body.

4.2.2.3 Arbiter

The number of crossbar arbiters, AR, required in a router is dependent on the partitioning of the crossbar. If the crossbar is not partitioned then only one crossbar arbiter is required. Otherwise the number of arbiters is equivalent to the number of crossbars. Each crossbar arbiter is placed in parallel to it's crossbar and receives inputs from the routing decision modules. As such is placed after the routing decision module on the path. The arbiter only effects control signals and as such is placed on the control path — PCP or VCP.

4.2.2.4 Address Converter

The address converter, where used, supports the routing modules and therefore the number of units required is equivalent to the number of routing decision modules. The address converter is required to read the incoming address but the result is only used by the routing module and not forwarded on the data path. Therefore, the AX is usually seen on the control path – PCP, or VCP. After conversion to relative addressing, an AC is required to make a decision based on the address. Again this decision is passed to the routing decision module. As such, the AX is placed before the AC and the RD on the path. The number of address converters required is equivalent to the number of address comparators.

4.2.2.5 Buffer

Buffer storage may be found on both the data — PDP, and control paths — PCP, representing queueing and information storage respectively. Queueing within the router may be found at the input, output, or middle of the router as described in section 2.4.6.2. In addition, a latch type buffer may be used on the data path at either the entrance to or exit of the router for temporary storage of a flit. This storage is usually available on each path and therefore the number of units required is dependent on the number of virtual channels in addition to the degree of the network. If virtual channels are implemented then buffering for each virtual channel is found within each virtual channel controller i.e. one queue per virtual channel.

4.2.2.6 Consumption

The consumption module represents the number of outgoing queues from the network. As it represents queues, then it is placed on the data path — PDP or VDP. The number of units required depends on the number of paths out of the network to the local processor. This may be chosen to be equivalent to the number of virtual channels but not necessarily.

4.2.2.7 Crossbar

The number of crossbar modules required depends on the method of crossbar partitioning used. If the crossbar is non-partitioned then a single crossbar module is required. The crossbar modules switch between paths and therefore do not lie on any single path.

4.2.2.8 Channel Update

The channel update module contains status information both used and updated by the decision module. As such it is a control module, implemented on the control path — VCP or PCP, and the number of units required depends on the number of routing decision modules.

Since each routing decision module requires access to the channel update module, distributing the module over any virtual channels on a a physical channel reduces the access bottleneck incurred by a single module implementation. As such, if virtual channels are implemented then a channel update module should be placed on each VCP, otherwise on the PCP.

4.2.2.9 Flow Control

The flow control module is used in combination with the routing decision module to create a routing decision. As such, the number of modules required follows the number of routing decision modules. The module is purely for control and does not alter the incoming data and therefore must lie on the control path — PCP or VCP.

4.2.2.10 Flit Update

Flit update depends on a decision made by the routing decision modules and changes the data of the flit. As such, the number of modules required, in general follows the number of routing decision modules and as in channel update, the module should be available for any path. As it is dependent on the results obtained from the routing decision module it should be placed after the routing decision module but on the data path— PDP or VDP.

4.2.2.11 Injection

The number of injection modules required depends on the number of queues available for injection into the network. As these represent queues they must lie on the data path from the processor — PDP or VDP. As in the consumption channels, the number of queues may be chosen to be dependent on the number of virtual channels.

4.2.2.12 Message Build

Message build is not always required in a router where message split is used. However, if required, the number of units is equivalent to the number of MS modules. It effects the data and thus must lie on the data path — PDP or VDP, at the join of the control and data paths.

4.2.2.13 Message Split

This module is only required when path splitting has been implemented. Message split is the module required to divide incoming flits between the data and control paths and as such must lie on the incoming data path — PDP. In the case of virtual channel control, splitting may also occur on the virtual channel data path — VDP.

4.2.2.14 Output

The output module adds the final alterations to outgoing data and as such must lie on the data path. Virtual channels, if implemented, are multiplexed onto the physical channels before the output module and as such do not effect the number of units required. As such the output module must lie on the PDP.

4.2.2.15 Routing Decision

Since the routing decision module does not need to see the incoming data on the data bus then it need not lie on the data path. If virtual channels are implemented then the routing decision module may either be partitioned over these paths or held as a single module on the control path. Thus the RD module lies on either the PCP or the VCP.

The routing decision must be made before switching of the crossbar and as such the RD module always lies on the input side of the crossbar on the relevant paths.

4.2.2.16 Signal Converter

The signal converter, converts external signals and therefore a minimum of one unit per physical channel is required at both the input to and output from the router. This module converts signals and not data and, as such, must lie on the control path — PCP.

Where virtual channels are implemented, the signal converters at the inputs may be distributed across the virtual channels — VCP or on the physical paths – PCP. At the output, the signal converter lies on the physical channels path i.e. on the PCP. This is due to the fact that multiplexing of virtual channels occurs before sending and as such, a single physical path is seen at the output of the router.

4.2.2.17 Signal Manager

The signal manager may control both external and internal signals. Control of external signals requires a signal manager for each physical path. However, where virtual channels are implemented, the signal manager may be distributed across the virtual channels or a single signal manager may control the signals for all the virtual channels. Both cases are used in the routers studied.

This is a signal control module and therefore lies on the control path, not the data path.

4.2.2.18 Synchroniser

The synchroniser synchronises incoming data and, as such, is required near the input of each physical path such that it is the first module to receive incoming data. If virtual channels are implemented it's function may be distributed across the virtual channels or retained on the physical path — PDP, before the path divides into virtual paths — VDP.

4.2.2.19 Virtual Channel Controller

The virtual channel controller, controls the multiplexing, buffering and arbitration of virtual channels onto the physical channel. As such, it must lie on the physical path and as it includes queueing must lie on the data path i.e. PDP.

4.2.2.20 Summary

Tables 4.5 and 4.6 summarise the design issues effecting the number of instances of a given module required in a design. The symbol '(x)' represents a design choice as to whether a module is distributed across the virtual channels and the letters 'c' and 'd' represent inclusion on the control or data paths respectively. Again the use of brackets represents a design choice and this time the choice is between whether to place the given module on the control or data path.

The factors given in tables 4.5 and 4.6 refer to the issues that effect the paths given in table 4.4 as well as the crossbar partitioning issue. This issue was not included in table 4.4 as the crossbar partitioning does not effect the number of paths in a design. It does, however, effect the number of crossbar and arbiter modules required. Issues which result in zero units of a module required are not included in the tables.

Issues	**Components**									
	AC	AD	AR	AX	B	CB	CC	CU	FC	FU
Degree	x	x		x	x			x	x	x
Number of VCs	(x)	x		(x)	x		(x)	x	(x)	x
Path Splitting	(c,d)	(c,d)	c	c	(c,d)		d	c	c	d
Partitioning			x			x				

Table 4.5: Design Issues Effecting the Number of Units of a Module Required (part a)

Issues	**Components**								
	IJ	MB	MS	OP	RD	SC	SM	SY	VC
Degree		x	x	x	x	x	x	x	x
Number of VCs	(x)	(x)	(x)		(x)	(x)	(x)	(x)	
Path Splitting	d	d	d	d	c	c	c	d	d
Partitioning									

Table 4.6: Design Issues Effecting the Number of Units Required of each Module (part b)

4.3 Modular Description of Existing Routers

In this section, a more detailed description of a number of the routers described in section 2.5 is given. The routers analysed include the Network Design Frame, the Caltech Message Routing Chip ,the J-Machine Router and the Virtual Channel Router. It is shown how each router's architecture may be described using the building blocks provided by the module set. Thus the figures describing the architecture of each router is a register transfer level description. Chapter 5 describes how these descriptions are used along with the design issued described in section 4.2 to create modelling instances within the framework. In addition, in sections 6.1 and 6.3, two examples of existing routers are modelled to the gate level using the framework.

4.3.1 Network Design Frame (NDF)

The name design frame refers to the fact that the router was designed to be integrated into the pad frame of the processor chip. The design is based on the Torus Routing Chip but to reduce the router size such that it fitted into the frame of the chip, partitioning of the data paths was implemented.

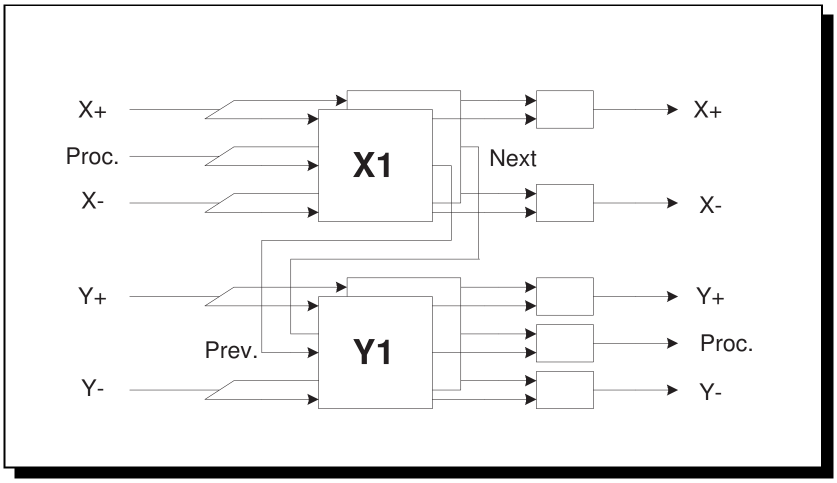

Figure 4.4: Network Design Frame Architecture

The NDF was designed for a bidirectional mesh. The logic of the router is divided into two priority networks and each priority network is further divided into two logic blocks representing the two dimensions — as shown in figure 4.4. Each block receives flits from the negative e.g. **X-** , and the positive direction e.g. **X+** in addition to flits from the previous level e.g. **Proc**. As illustrated, the previous level may either be the input from the processor or previous block. Likewise the results from a block will either be sent out in the same dimension or forwarded to the next dimension which may be another block, or the processor itself.

As each block implements the logic for a single dimension and a single priority, the two priorities must compete for use of the physical channels. That is that a virtual channel module is required at each of the outputs from a block. The VC module provides buffering and multiplexing for a given output for both priority levels i.e. both virtual channels.

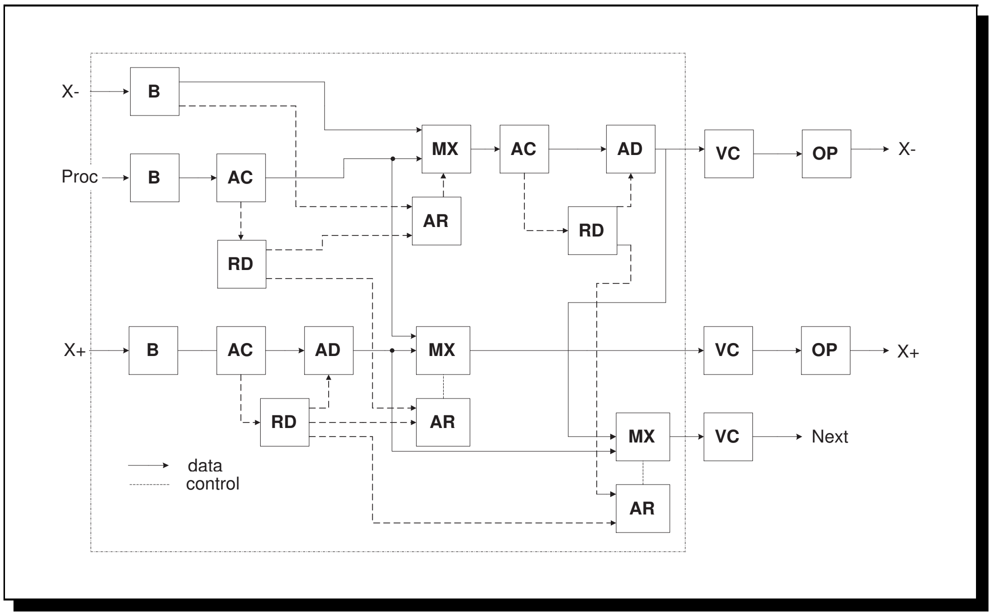

Figure 4.5: Network Design Frame Logic Block and Exit Paths

After selection of the appropriate virtual channel to be sent on the physical channel, arbitration for the physical channel is required. The NDF implements bidirectional channels using a single channel and an arbitration line. After arbitration is won, since request/acknowledge signalling is used, a request signal must be generated when valid data is driven on the bus. These jobs are part of the output module and as such are placed only on the external paths.

Within each of the logic blocks, symmetric paths have not been implemented. However, since relative addressing is used, each flit in the current dimension requires a zero checker, AC, a routing decision module, RD, and an address decrementer, AD. For the path from the higher level, a sign checker, AC, is required instead of a zero checker to find out if the flit is to be forwarded in the positive, negative or to the next dimension. A sign check is not required for incoming flits in the current direction unless they change dimension in which case they will encounter a sign checker in the next level block. Switching between the different data paths occurs by way of 2x1 multiplexers, that is the crossbar has been partitioned into a number of 2x1 multiplexers. An overview of a single block is given in figure 4.5. As shown, most of the modules lie on the data path through the router apart from the RD and AR modules which lie on control paths.

Due to deterministic routing and a partitioned data path the routing decision is very simple. If the zero-checker — AC component, detects a zero head flit then the flit is passed to the next dimension and the header flit stripped. Otherwise the address is decremented and the flit progresses in the same direction. As such it is only a two way decision based on the signal from the zero-checker. The decrementer operates in parallel with the RD component and as such when the RD has made it's decision based on the result from the zero-checker it controls the multiplexer to either select the decremented address or the original address.

4.3.2 The J-Machine Router

The J-Machine router, illustrated in figure 4.6, is similar to the Network Design Frame in that router logic is duplicated for the two priority networks. However, a three dimension mesh router is designed instead of the two dimensions in the NDF case. The J-Machine router is also a fully synchronous router.

Instead of partitioning the crossbar into a number of 2x1 multiplexers, two 2x1 multiplexers and one 3x1 multiplexer are used within each logic block.

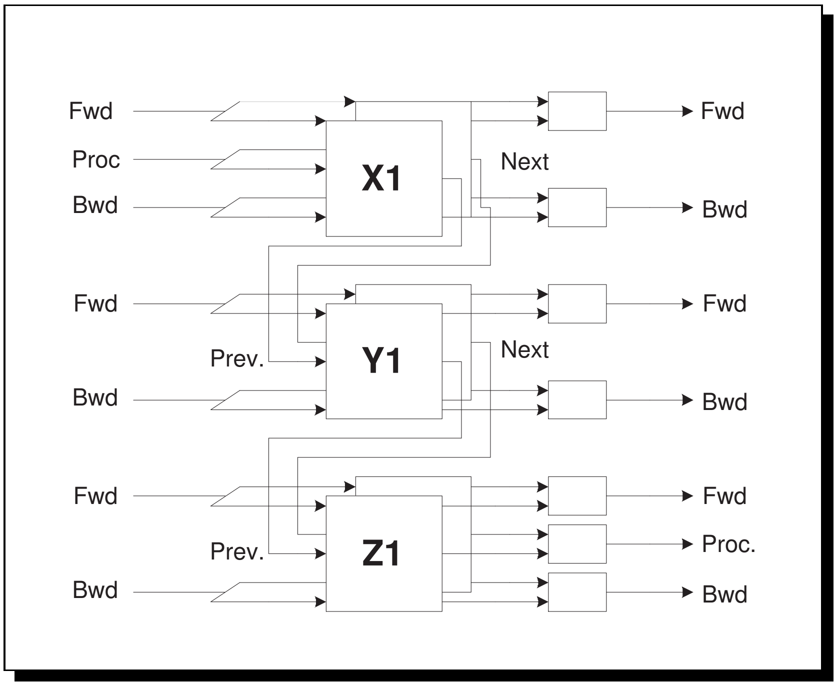

Figure 4.6: J-Machine Router Architecture

This provides a more symmetric structure within the block, as shown in figure 4.7.

As in the NDF, an AC module is used to perform sign checking at the entrance to a logic block from the previous dimension. However, in the current dimension, the AC module used in the NDF is replaced with an address converter, AX, to convert the absolute address to the relative address for routing decision purposes. Since absolute addressing is used, no address decrementer is required. Buffering is again placed at the output and consists of two flit buffers per virtual channel.

4.3.3 Caltech Message Routing Chip (MRC)

The MRC is designed for a 1-D topology requiring a 1 flit header. However, as a single dimension topology is no longer of interest in parallel computers,

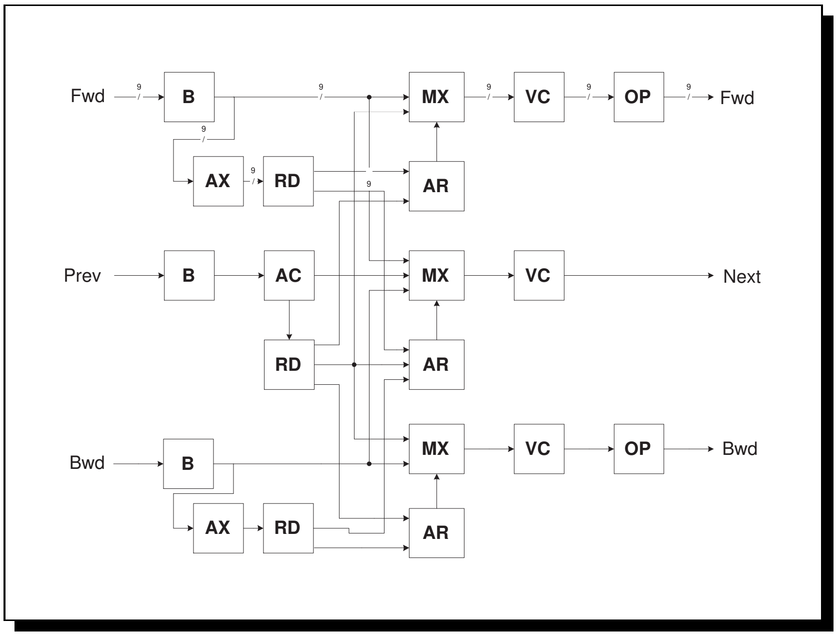

Figure 4.7: J-Machine Router - Routing Path

the base topology of the MRC design is assumed i.e. the two dimensional mesh where a logic block is equivalent to a one dimension of the mesh. The basic structure of the router architecture is similar to that of the previous two architectures except that virtual channels are not implemented. Therefore, as shown in figure 4.8, there is no duplicate logic representing a second logic level.

As stated in section 2.5.1.7, three variations of the chip exist — 8, 9 and 16-bit versions. However, only the first 8 bits of the header flit are considered in the routing decisions which allows the same decision logic to be used in all three variants.

As in the case of the Torus Routing Chip, the router and the channels are fully asynchronous. However, instead of using request/acknowledge signalling within the router to achieve synchronisation of operations, ripple through FIFOs are used.

Relative addressing is implemented requiring both an address comparator and an address decrementer to aid the routing decision module. However,

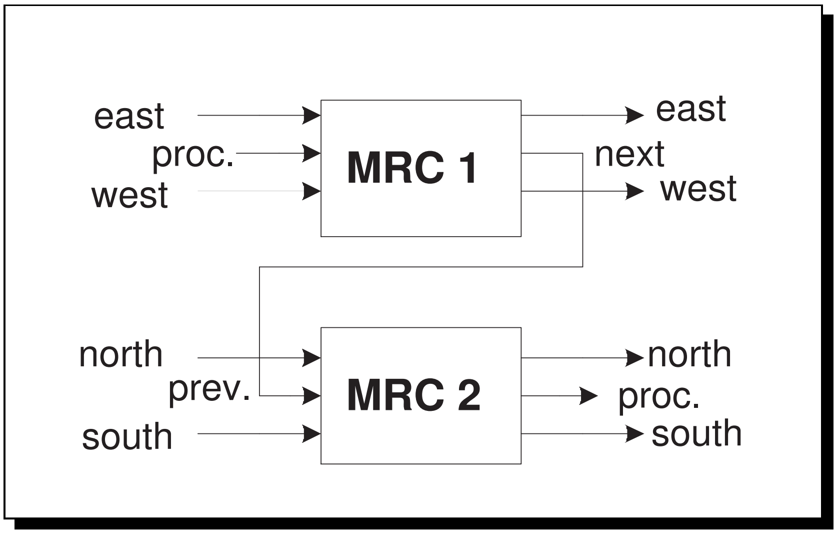

Figure 4.8: 2-D Message Routing Chip Architecture

in this case the address decrementer precedes the address comparator on the path. This means that an incoming packet destined for the current node will enter the node with a non-zero header flit. However, decrementing occurs before the AC module receives the flit and as such a zero address will be detected.

The crossbar is partitioned similar to the J-machine, in that each logic block uses two 2x1 multiplexers and one 3x1 multiplexer. Fair arbitration is implemented in the crossbar. However the three-to-one multiplexer required is implemented as two two-to-one multiplexers giving priority to injected traffic i.e. from the processor or the previous dimension.

An illustration of part of a logic block is given in figure 4.9. It is interesting to note that not only does the address decrementer precede the address comparator but that the routing decision module is also placed on the main data path. Placing the RD module on the main path, means that all flits through the router encounter a delay through this module. However, the MRC was optimised for throughput and not latency and as such a single path controlled by ripple through FIFOs is implemented.

These ripple through FIFOs allow requests to progress with data. Acknowl-

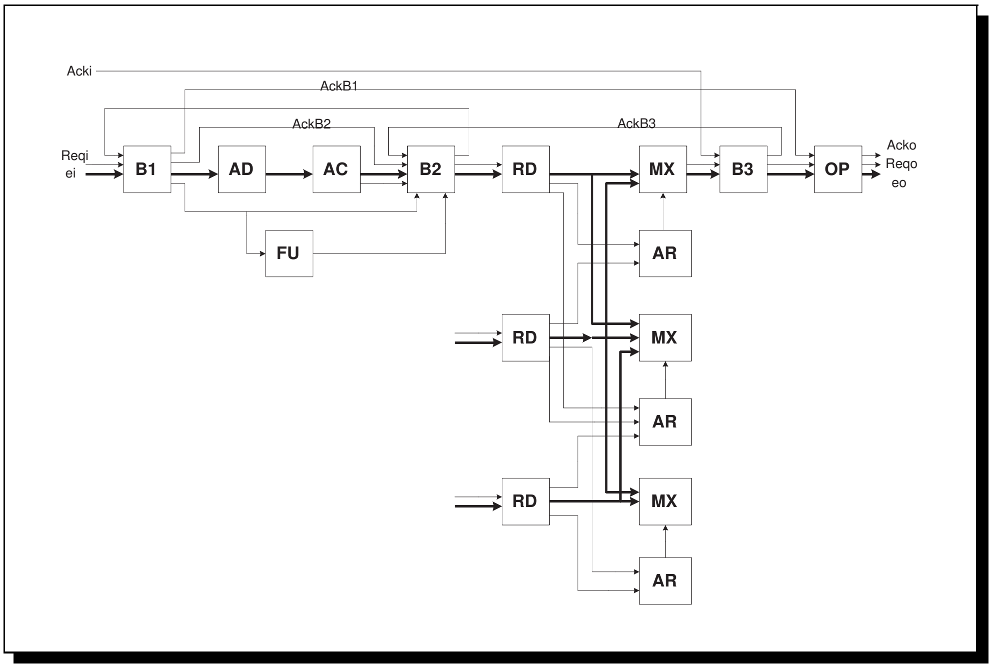

Figure 4.9: Message Routing Chip - Routing Path

edgements from each FIFO flow backwards. That is that an acknowledgement from a given FIFO flows back to the previous FIFO i.e. the first FIFO reached in the direction of the input to the router, as shown in figure 4.9. The first FIFO thus returns it's acknowledgement to the previous router and must therefore forward the acknowledgement to the output module for sending. It should be noted that for clarity only the first path of the router block is shown in full along with the building blocks which illustrate the distribution of the crossbar and paths within the logic block.

4.3.4 Virtual Channel Router

The virtual channel router (VCR) is designed for an n-dimensional hypercube network. A new routing technique is used which can be said to be a combination of wormhole routing and circuit switching — see section 2.5.1.5.

The architecture of the VCR is shown in figure 4.10. An unusual feature of

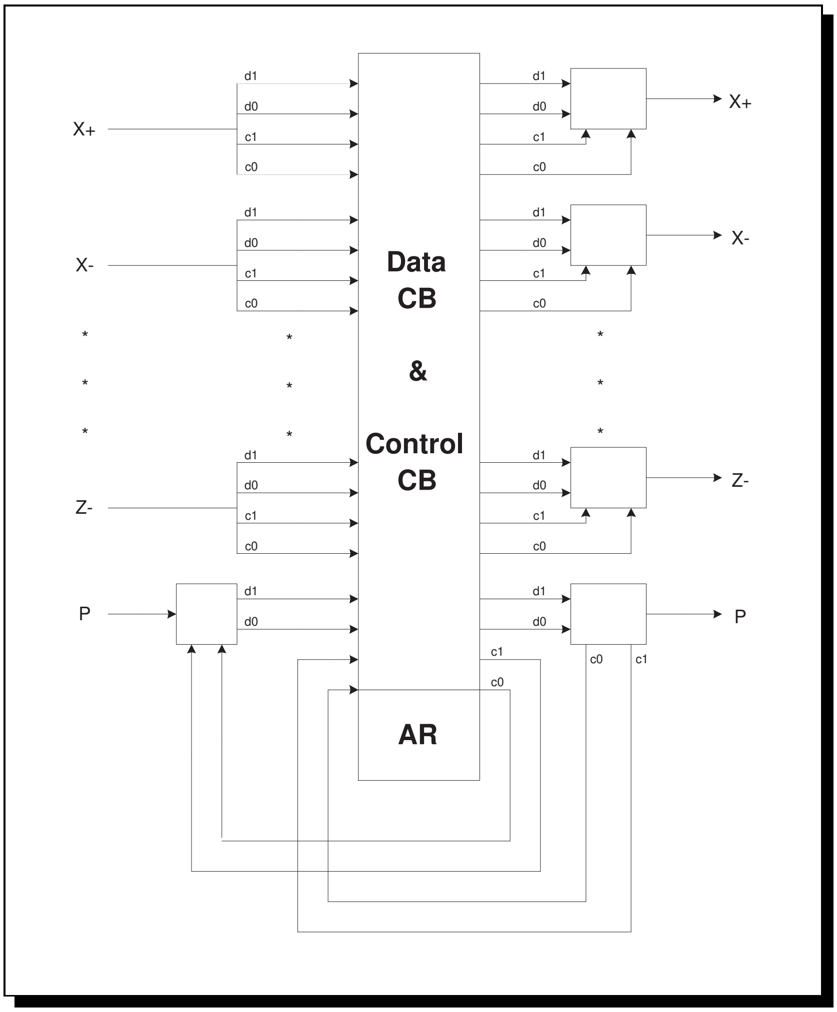

Figure 4.10: Virtual Channel Router Architecture

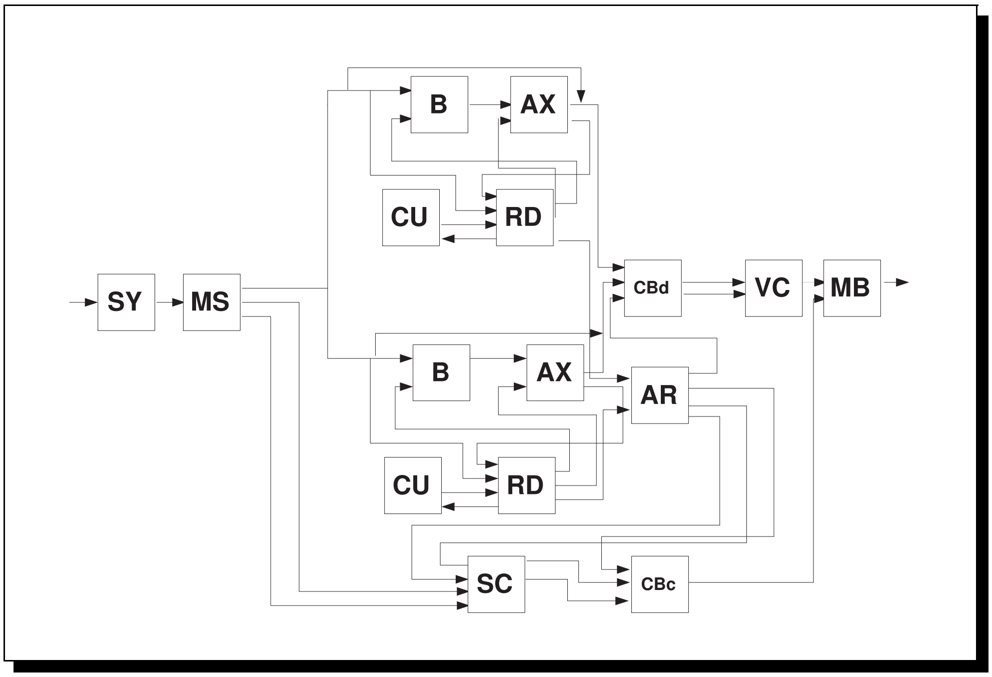

Figure 4.11: Virtual Channel Router - Router Path

this router is that data is sent in serial on the physical channels. Each one-bit data channel has an associated strobe line which strobes the data both on the rise and falling edge.

Flow control and signalling in the VCR is by way of control codes. These are transmitted in the data packets of the opposite channel, if active. Otherwise i.e when the channel is cached or free, special control packets are sent. A signal control module, is used to handle both receipt and creation of control codes. These codes are created by the SM and switched to the required output using a control crossbar. That is that two central crossbars are implemented, one to handle data transfer and one to handle control transfer. A single arbiter controls both crossbars sending grants in parallel to the selected input of both crossbars. It is worth noting that the chosen connections across each of these crossbars are not equivalent. The data crossbar connects to the output route which leads to the following node, whereas the control crossbar connects to the output route which leads back to the previous router.

Two virtual channels are implemented and channel selection is based on a tag bit which is the first bit of an incoming packet. As such, a message split module is required not only to split the control information i.e. control codes, from the data information but to decide which virtual channel path should receive the incoming data. Note that in the case of request/acknowledge signalling, data is forwarded to both virtual channels and request lines are used to select the appropriate channel to accept the incoming data. The extracted control codes from the incoming packet are forwarded to the signal manager on the control path.

Multiplexing between virtual channels occurs after the crossbar, that is that all virtual channel paths traverse the crossbar. Following the virtual channel module, a message build module is required to add any required control information, supplied by the signal manager, to the outgoing packet. A single path through the router is illustrated in figure 4.11.

Chapter 5

Implementation of the Framework

This chapter discusses the modelling tools chosen, the implementation of the framework and the process of model creation using the framework.

The modelling tools are discussed in section 5.1. Technology considerations for performance evaluation are presented in section 5.2. Section 5.3 presents an overview of the framework whereas section 5.4 provides a more detailed insight into the framework. Finally in section 5.5, the method of model creation using the framework is presented.

5.1 Modelling Tools

This work attacks the problem of parallel computer communication architecture modelling on two main fronts. The first is to provide a framework to analyse a wide range of both existing or proposed networks and their respective routers. As such, generalisation is the key. The Performance Description Language PDL [VMM96] is designed specifically for generic performance modelling enabling design instances to be compiled and analysed and is, therefore, chosen for the framework.

The second is to provide cost and performance analysis where performance data is achievable at the gate-level. With this in mind, the performance model needs to be nearer to the hardware implementation and a chosen technology. As such specialisation or in other words instantiation is the key. HDL languages e.g. VHDL and Verilog, offer a flexible medium in which to describe hardware enabling gate-level designs to be realised through the process of logic synthesis.

The combination of PDL's ability to express generality in design and HDLs' ability to realise gate-level, technology specific designs allow the project goals to be met. That is offering a wide range of design choices in sufficient detail such as to enable cost and performance analysis to be undertaken.

5.1.1 Performance Description Language(PDL)

In PDL, a system description is built up of modules, ports and carriers. Ports are primitive objects in that they cannot contain other objects. Carriers provide port to port connections thus providing a path for transmission of data between ports. Ports provide gateways for data flow in and out of a module. A module may be built up of submodules. The modules, carriers and ports of these submodules provide the next layer in a hierarchical design.

Attributes and their evaluation rules may be attached to carriers, ports and modules. Hierarchical attribute propagation is provided through containment of modules within other modules and lateral attribute propagation is provided by the connectivity provided by the carriers of the design.

A generic module description provides design generality in that each module is not restricted to a given number of inputs or outputs or specific submodules. The generic description provides for any number of inputs or outputs and any number of submodules of the submodule types declared in the module. Each submodule is also given a general description including a list of possible submodules and inputs and outputs. This process continues through a hierarchical design until there are no more modules to be defined.

A generic description may also include attributes. These may be in the form of primitive or non-primitive attributes. Primitive attributes provide data entrance points for relevant data to be fed into a module. This data provides values for attributes used in performance expressions within the given module. Non primitive attributes are assigned values derived from these performance expressions. That is that their values are derived from calculations within the module at evaluation time. These attributes may be used to pass the performance evaluation results to the next higher level in the module hierarchy.

Both primitive and non-primitive attributes may also be static or dynamic. Primitive static attributes may be thought of as constants in that they are allocated a single value. However, non-primitive static attributes obtain their values from an expression within the module and as such are similar to the variable concept. Dynamic attributes on the other hand, can take a number of different values during the evaluation of a design.

Primitive dynamic attribute values are obtained from data files but, unlike the static case, a number of values may be supplied for a given attribute. The number of values supplied gives the number of evaluation cycles required to obtain the simulation results for each of these attributes. That is that within a design evaluation, a number of sub evaluations each realising a set of results, will be required. Any attribute that is dependent on a dynamic attribute must also be dynamic such that it's result will be evaluated for each new value of the source attribute. As such, the number of sub-evaluations required for a design depends on the largest number of values given to any primitive dynamic value. If a primitive dynamic attribute is allocated less values than this maximum figure then extra values are added during evaluation. These values are equivalent to the last value allocated to the attribute in the data file.

At the top level, performance expressions are evaluated using the intermediate results either calculated or provided at the lower levels and also from primitive attributes added at this level. As such, a generic module in PDL provides both a frame for possible design instances and the evaluation rules and data gathering mechanisms to enable performance analysis of a given design to be undertaken. Using the dynamic attributes described a number of results may be obtained for a given design within a single run of a model.

A design instance involves the creation of a performance model for a given design using the framework provided. It is described in the form of a PDL net-list. A number of different designs may be described within the bounds of the frame provided by the generic module descriptions in PDL. A PDL net-

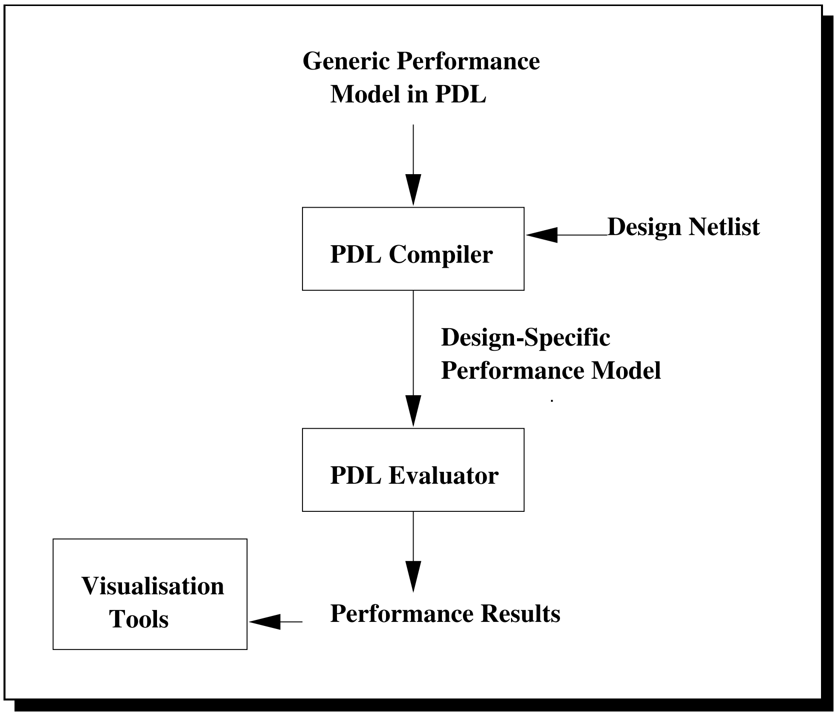

Figure 5.1: PDL Evaluation Environment [VMM96]

list, representing the design instance to be evaluated, is compiled together with the PDL performance model (generic descriptions) to generate a design-specific performance model. The evaluator then evaluates this model with the help of primitive attribute values provided by the data file. The performance results obtained may then be fed into the visualisation tools available, where applicable. This process is described in figure 5.1.

5.1.2 VHDL for Logic Synthesis

HDL languages provide a valuable description tool for register-transfer-level design (RTL). Designs may be specified with respect to their desired behaviour — behavioural designs or with respect to the interconnection of a given set of components — structural designs. Simulation tools provide a graphical means by which signal paths through these designs can be traced and the desired behaviour verified.

To obtain a gate-level design, the HDL description may be synthesised with respect to a given technology. This gate-level description can again be simulated and the results compared with the results obtained from the RTL simulation. The net-list resulting from logic synthesis describes the design with respect to a set of logic gates and their interconnections.

As described in section 5.1, there are two main HDL languages currently in use — Verilog and VHDL. There is much disagreement in the HDL community as to which language is better suited to a particular application. Suitability in this case refers more to the ease of using the language for the purpose chosen than whether the language can meet the requirements. In the framework, the choice of language is left to the designer as both languages enable the requirements to be met i.e. register-transfer level behavioural descriptions and logic synthesis. In the case studies described in section 6, VHDL is the implementation language.

5.2 Technology Considerations

Although it is a modelling framework being described, it is important to consider the steps required from modelling to an actual design implementation such that realism is inherent in the models created. Implementation is technology dependent and the two technologies considered in this work are ASIC and FPGA technology.

In ASIC design the basic components are logic gates and, as such, the net-list design description describes the interconnection of the basic components. However, in the case of FPGA technology, components are termed configurable logic blocks (CLBs). Unlike the logic gates of ASIC design where each gate represents a single simple function, each CLB includes a number of function generators (LUTs). Any logic function may be implemented in a function generator within the limits of the number of input ports available on the block.

The effects of routing delay in ASIC design performance may be said to be negligible with respect to logic delay. However, in FPGA technology a substantial proportion of delay arises from the interconnection of the CLBs.

For a design to be implemented on a given FPGA device, sufficient CLBs are required to meet the logic demands of the design and sufficient routing available to interconnect the chosen CLBs. The former requirement is easy to solve and includes a check of whether the number of CLBs required is within the limit available for the device chosen. The latter requirement is more complex. This is dependent on the placement of the logic to be routed and the number of routing channels available between these CLBs. In earlier devices, a rule of thumb used is that, if less than 60% of the available CLBs in a device are required for a particular design, then the design can be routed. However, in newer devices such as the XC4036 device used in the case studies of chapter 6, the increased number of routing channels may increase this assumed factor to around 90% of the CLBs of the FPGA.

Although these newer devices have greater routing capacity, two questions need be raised regarding the reliability of results obtained. Computer Aided Design tools may provide the possibility to optimise a design for area, speed or both. However, if the routing step of the design process is undertaken can it be guaranteed that the same placement and routing will be achieved when several modules are placed together on an FPGA? That is, if delay is a measure of the longest path through a module will a different placement and routing give rise to the same delay results assuming that the same optimisation choice made is applied to all modules? Also how can an estimate for routing between modules be found?

After the modules are routed, it is difficult to compare the resulting delay due to the placement and routing effects discussed. However, a useful solution is to normalise the routing. A normalised estimate for routing may be obtained from the FPGA CAD tool for the device chosen. This is based on 'best case' routing . Best case routing refers to the assumption that all nets connect neighbouring CLBs — that is that there exists no nets longer than the space dividing two neighbouring CLBs. Therefore, for a given module the longest path delay is based on the number of CLBs on the path — logic levels, and the number of interconnections — logic levels minus one.

Best case routing values are very optimistic and in practice are not realisable. However, as normalised values they give a base for fair comparison between different design instances, a goal of this work.

To interconnect routers it is assumed that each router is implemented on a single FPGA. As such, communication between routers requires communication between two FPGAs. It may be assumed that the delay on a wire connecting these two FPGAs is so small that it may be regarded as zero. However, what may not be neglected is the time taken to drive a signal from an FPGA onto the wire and receive a signal from the wire. That is, the delay on the output and input pads of the FPGAs are non-negligible. In Xilinx technology with TTL inputs these delays are referred to as top (propagation delay from output to pad) and tpid (propagation delays from pad to Inputs). It should be noted that communication between a router and its local processor also requires off-chip driving or receiving for sending to or receiving from the local processor.

5.3 Structure of the Framework

Figure 5.2 illustrates the key parts of the framework. This section provides a clockwise description of the different parts starting at the generic router.

The main component of the interconnection network of a communication system is a router. Assuming the case of a distributed router, a router exists at each node of the interconnection network. Therefore, the interconnection network is expressed as an interconnection of routers. Different direct network topologies may be implemented by varying the interconnection of the routers.

The router itself may be regarded as an interconnection of components, where the term component refers to standard components found in the digital libraries of computer aided design tools. It is chosen to model a generic router as an interconnection of a set of modules where a module is not necessarily a single router component but may be build up from a number of components. However, not all routers require the same modules. As such, a set of modules is provided, a subset of which may be used to describe a given router. Varying the modules or their interconnection allows different router models to be developed. In section 4.1 the set of identified router modules included in the framework is presented and discussed.

To take the design process one step further. Each module is expressed as an interconnection of logic gates. The framework includes a set of generic

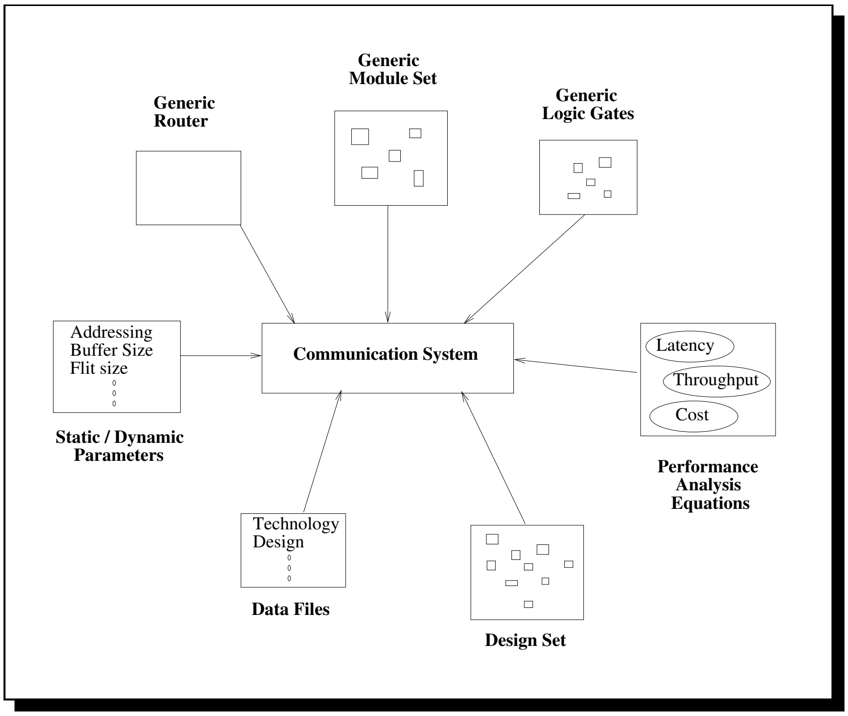

Figure 5.2: Overview of the Framework

logic gates, a subset of which may be used to describe a module design. This description takes the form of a net-list i.e. a listing of the logic gates required and their interconnections.

Performance analysis equations currently consist of expressions for two of the three key performance features: latency, and cost. The values given to variables in these expressions are either obtained direct from the static parameters provided as input to a model or are derived within a model. Other parameters effect the choice of performance equations for the evaluation. In the case of latency analysis, different performance equations exist for different switching techniques, as described in section 3.3.3. A more detailed description of performance evaluation is given in section 5.4.2.

The framework is designed to handle the third performance metric, throughput and as such, this feature is included in figure 5.2. However, this part of the framework is not fully implemented, as yet.

To move away from generality and towards specialisation, detailed design descriptions are required to create a sufficiently detailed model. These are provided in the set of designs. For each module in the generic module set, one or more design instances are available in the design set. Each design is a net-list description of a design implementation for the respective module. These designs are technology specific, generated from logic synthesis, as described in section 5.1.2.

In the case of FPGA technology, in the process of net-list generation, these designs are analysed for delay and cost as described in section 5.2. This design data is stored in a design data file. This data is absorbed by the generic modules through their primitive attributes. In the case of ASIC technology, two choices are given in the framework. Similar to FPGA technology a design may be analysed within the HDL synthesis tool to generate timing information for the module. However, as the lowest level of the framework is the logic gates, technology data files may hold information regarding gate delays which are absorbed by the generic logic gates to be used directly in performance analysis. It should be noted, however, that this latter process avoids the loading effects on individual gate delays. The choice of which method to use is left to the user although by default module delay parameters are assumed.

The final and important feature of the framework structure is the existence of design parameters. As stated, the goal is to provide a flexible framework such that a wide range of communication systems at the hardware level can be modelled. This wide range of designs is provided, in part, by the generality of design given in the generic modules and also in the addition of parameters to the framework.

Static parameters provide information regarding design choices made and thus allow these choices to be fed into a model and their effects evaluated. Dynamic parameters may replace static parameters such that a range of results are obtained for a corresponding range of inputs within one run of the evaluator.

Dynamic parameters also allow traffic patterns to be fed into a model enabling loaded performance analyses to be undertaken. However this feature is not completed within the framework as yet — see section 8.

5.4 Implementation of the Framework

5.4.1 PDL General Files

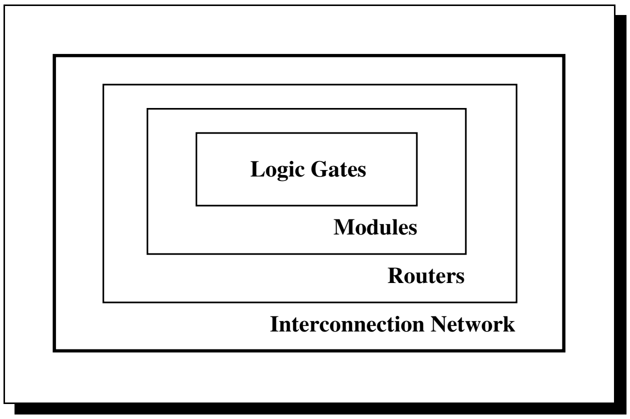

Figure 5.3: Containment in the Modules

The framework is built up of a hierarchy of submodules. At each level of the hierarchy, the PDL description defines the type of submodules allowed i.e. the modules of the next lower level in the hierarchy, the attributes of the module and their evaluation rules. This hierarchical modelling can be described as a containment of submodules at each level. Figure 5.3 illustrates the four level hierarchy provided in the framework for network models and introduces the modules of each level as we descend the hierarchy i.e. routers, modules from the module set and logic gates respectively. It should be noted that the lowest level, logic gates, only applies to ASIC technology and not FPGA technology. This is due to the fact that the elements of FPGA technology are CLBs and not logic gates. However, both technologies are expressed in terms of logic gates in the net-lists.

```
#include "test.pdl"                          -- generic router module
#include "switch_tech.pdl"                   -- functions for latency calculations
#include "techdata.pdl"                      -- technical data required
type
  switching_techniques : enum{WH, VCT, SF, CS}; -- new type for this level
end type;
module topology                              -- MODULE DECLARATION
   modules                                   -- SUB-MODULES
      RTs{}: router;                         -- any number of generic routers
   carriers                                  -- NETS
      wires{} : wire;                        -- any number of connections of type wire
                                             -- wire declared in test.pdl
   ports                                     -- INPUT/OUTPUT PORTS
      inputs{}: inout;                       -- any number of inputs of type inout
      outputs{}: inout;                      -- any number of outputs of type inout
                                             -- inout declared in test.pdl
   attributes -- ATTRIBUTES
      cost : int;                            -- total gate count from lower levels
                                             -- evaluated attribute

      primitive dynamic sw_tech : switching_techniques := WH;
                                             -- primitive, supplied in the data file
                                             -- dynamic, more than one value
      dynamic net_latency : real := 0.0;     -- evaluated from dynamic values
                                             -- results for each dynamic value

      primitive flit_size : real;            -- flit size (data bits)
      primitive pack_size : real;            -- packet size (flits)
      primitive net_diam : real;             -- network diameter
      ..........
      primitive prop_delay : real;           -- delay of the body flits(actual)
   rules                                     -- ATTRIBUTE EVALUATION RULES
      cost =  list_int_sum(foreach x in RTs{x'cost});
                                             -- sum of the cost from each router
                                             -- function described in function file
      inputs{}'delay = 0.0;                  -- initialises network input delays
      let delay_val[]: real = (foreach d in RTs{d'delay});
                                             -- list of router delays
      let node_delay : real = delay_val[1];  -- delay seen by a header flit
      -- let prop_delay : real = delay_val[1]   -- sets delay of body flits equal to head
      net_latency =                          -- evaluation of network latency,
                                             -- switching technique dependant
      case sw_tech of                        -- delared functions in test.pdl
          WH : get_lat_WH(....);
          VCT : get_lat_VCT(.......);
          SF : get_lat_SF( ......);
          CS : get_lat_CS(.....);
          others : 0.0;
      end case;
end module;
```

Figure 5.4: PDL Network Description

To provide an insight into the PDL coding techniques used and the interrelationship between the different levels, an example PDL hierarchical description is described starting at the top-level, the interconnection network. Figure 5.4 illustrates the generic network description where "...." represents missing code and is used for readability. A PDL model is compiled starting with the lowest level of the hierarchy. As such, a file description must first include the file holding the PDL description of the next level in the hierarchy, if any. In this case test.pdl holds the router generic description. Any library files or type declarations not previously declared i.e. in the lower level files, are also included. In addition, data files holding values for primitive attributes declared at this level may either be included or added at evaluation time.

The main element of the file is the description of the interconnection topology module itself — topology. This declaration includes the declaration of the submodules, of which in this case, there is only one type — routers. This is due to the fact, as described in section 2.2, that a network may only be made up of an interconnection of routers. Note that the type router refers to the generic router description and therefore a network may be made up of different routers of type router. The carriers, ports and attributes of the module are declared along with a set of evaluation rules for the non-primitive attributes. During evaluation of this file the PDL evaluator will expect to find a value or values, for the primitive attributes in the data file. Note that a number of the primitive attributes e.g. switching technique, flit size, pack_size (packet format) and net_diam (diameter); refer to the design issues discussed in section 4.2. That is that a number of design issues not only effect the complexity and placement of modules but also effect the network performance itself. These design issues have been declared as real in the PDL code. However, these could have been declared as integer as the PDL compiler handles conversion of integer to real when the attribute is used in an expression to generate a real result i.e. 'net_latency'.

To calculate propagation delay i.e. the actual delay seen by the flits of the body of a packet, two compilations are required. The first compiles the PDL file letting 'prop_delay' be equivalent to 'node_delay' and the data in the PDL data file is updated to represent those delays seen by the flits of the body. After compilation and evaluation, 'rout_delay' represents the delay through the router seen by the flits of the body. This value is then added to the data file as 'prop_delay' and the rest of the data file is updated to reflect the delays seen by the head flits. The network level PDL file, shown in figure 5.4 is also updated i.e. prop_delay is now a primitive attribute and not part of the evaluated attributes. That is that 'prop_delay' is no longer equivalent to the node delay. Compilation and evaluation produces 'rout_delay' which this time represents that seen by the header and 'net_latency' represents network latency reflecting the difference difference in router delays seen by the header and body flits.

The 'net_latency' expressions are PDL functions representing the equations for network latency derived in chapter 2. 'Net_latency' is declared as 'dynamic' such that several latency results may be obtained in a single evaluation of the PDL evaluator. For example, to consider the effect of changing network size on latency, a range of network diameters may be given in the data file and evaluation will provide a set of results. To achieve this, the 'net_diam' attribute declaration is changed to 'dynamic'. An example data file is included in appendix F.

The next level, the generic router description is given in figure 5.5. The structure of the file follows the structure of the topology description given in figure 5.4. Both the types wire and inout, declared at this level, may be seen to be used at the higher level. The input and output ports of wire are described as being of type inout. This type has attributes delay and values where the attribute values is used to transmit the bits of a packet through the network. The wire delay expression depends on whether analysis is with respect to ASIC or FPGA technology. As discussed in section 5.2, in ASIC technology wire delay may be assumed to be negligible i.e. nominal delay = 0.0, whereas in FPGA technology wire delay may be assumed to be the device dependent nominal delay. In the instance shown wire delay is set to 0.984 representing that for device XC4036 used in the case studies. The values expression states that the value at the input is equivalent to the value at the output i.e. no loss of value on a wire (transmission line quality).

```
#include "rout_func.pdl"                          -- functions file
#include "logicgates.pdl"                         -- generic logic gates
#include "addcomp.pdl"                            -- generic address comparator
#include "adddecr.pdl"                            -- generic address decrementer
.......
type
  logic : int;                                    -- needs restricted to 0,1
  logiclist[] : logic;                            -- flit value
end type;

port inout                                        -- PORT TYPE
  attributes
     delay : real;                                -- propogation delay between modules
     dynamic values : logic:= 0;                  -- propogates data between modules
end port;
                                                  -- i.e. traffic
carrier wire
   ports
     output : inout;
     input : inout;
   attributes
     nom_net : real;
   rules
     --nom_delay = 0.0;                            -- ASIC
     nom_net = 0.984;                              -- FPGA (XC4036)
     input'delay = output'delay + nom_net;
     input'values = output'values;
end carrier;

module router -- MODULE DECLARATION
   modules                                        -- includes all modules in the module set
      ACs{}: add_comp;                            -- delared as sets:
      ADs{}: add_decr;                            -- allows any number of each module
      MXs{}: multiplexer;
      ............
   carriers                                       -- NETS
      wires{} : wire;
   ports                                          -- INPUT/OUTPUT PORTS
      inputs{}: inout;
      outputs{}: inout;
   attributes                                   -- ATTRIBUTES
      cost : int;                                 -- number of gates/CLBs
      delay : real; -- longest path delay
      -- FPGA technology
      primitive sci_cost : int;
      primitive sm_cost : int;
      primitive b_cost : int;
      primitive sci_delay : real;
      primitive sm_delay : real;
      primitive b_delay : real;
      ......
   rules                                          -- EVALUATION RULES
      -- ASIC technology
      --delay = list_max(foreach x in outputs{x'delay});
      --cost =  list_int_sum(foreach x in ADs{x'num_gates}) +
      --        list_int_sum(foreach x in ACs{x'num_gates}) +
      --        list_int_sum(foreach x in MXs{x'num_gates}) +
              ..........;
      -- FPGA technology
      SCIs{}'sci_delay = sci_delay;
      SCOs{}'sco_delay = sco_delay;
      SMs{}'sm_delay = sm_delay;
      .......
      delay = list_max(foreach x in outputs{x'delay});
      cost = list_int_sum(foreach x in SCIs{sci_cost})+
             list_int_sum(foreach x in SCOs{sco_cost})+
             list_int_sum(foreach x in SMs{sm_cost})+
             ......
end module;
```

Figure 5.5: PDL Router Description

The possible submodules of the router are the set of all router modules, each of which is declared as a set. This allows null, one or more of each module in a given router instance. Evaluation of cost and delay is technology dependent. In FPGA technology, the basic components of the design are CLBs — see section 5.2, and module delays are added as primitive attributes to the model as shown. They are then passed from the router level to the respective modules i.e. SCIs'sci_delay = sci_delay'. ASIC module delays may be calculated within the PDL model based on the logic gate delays of the underlying design. They may also, similar to the FPGA delays, be evaluated within the CAD tool and the module delays entered as primitive attributes. The cost of FPGA designs is expressed in terms of CLBs and these values are entered as primitive attributes. In ASIC technology, similar to delay, can these values either be entered as primitives or calculated with respect to the underlying design. Note that evaluated values for cost and delay from this level are those used in the evaluation rules at the network level : RTs(x'cost) and RTs(d'delay) respectively.

```
module multiplexer
   modules                                            -- MODULE DECLARATION
      ANDs{}: and_comp;                               -- includes all possible logic gates
      ORs{}: or_comp;
      NOTs{}: not_comp;
      ............
   carriers                                           -- NETS
      wires{}: wire;
   ports                                              -- INPUT/OUTPUT PORTS
      inputs{}: inout;
      outputs{}: inout;
   attributes                                         -- ATTRIBUTES
      --num_gates : int;                                 -- number of gates  (ASIC)
      primitive mx_delay : real;                      -- standby fixed delay (FPGA)
      time : real                                     -- temp for delay calculation
   rules
     -- ASIC technology
     --num_gates = #(foreach x in ANDs{1})             -- module cost
     --          + #(foreach x in ORs{1})
     --               ..... ;
     -- FPGA technology
     time = list_max(foreach x in inputs{x'delay}) + mx_delay;
     outputs{}'delay = time;
end module;
```

Figure 5.6: PDL Multiplexer Description

Figure 5.6 gives an example of a generic module description, in this case the multiplexer module. The module declaration follows the format described above but this time the submodules included are the set of generic logic gates.

Again a difference between evaluation for FPGA and ASIC technology may be seen. As stated, cost for FPGA technology evaluation is entered at the higher level as a primitive attribute and, as such, no cost evaluation is required at this level. In ASIC technology cost depends on the number of gates in the design. The symbol '#' returns the number of elements in a set e.g. '#(foreach x in ANDs1)' returns the number of AND gates. That is, it returns the number of instances of the given sub-module in the design, in this case the multiplexer.

In FPGA technology, the module delays entered at the higher level are used in the calculation of the output delay of the module. Lateral propagation within the module propagates the delays from previous modules on the path to the inputs of the multiplexor. The output delay is dependent on the maximum delay at the inputs and the delay traversing the module (mx_delay). In ASIC technology, the basic components effecting delay are the logic gates and, as such, a similar calculation of delay may be seen at that level. Lateral propagation of delays at the logic gate level will automatically provide values at the output of the this level. As such, no expression for delay is required at this level.

Figure 5.7 describes part of the generic logic gates file. It should be noted that no submodules or carriers are included at this level as this is the lowest level of the hierarchy.

The main function of generic logic gates is a logic function dependent on the gate type. This function is applied to the values at the inputs, which represent traffic or control flowing through a router. As such the inputs are dynamic values resulting in dynamic output values for the logic gate. In ASIC technology, where it is chosen to calculate delay at the gate-level instead of supplying module delays to the performance model, the gates absorb input delays through their ports. The relevant gate delay is added to the largest of these values and their output delays are forwarded using the lateral propagation feature of PDL.

```
module and_comp                                  -- MODULE DECLARATION
  ports                                          -- INPUT/OUTPUT PORTS
     inputs{}: inout;
     output: inout;
  attributes                                     -- ATTRIBUTES
     primitive and_delay : real;
  rules                                          -- EVALUATION RULES
     output'values = find_and_val(foreach x in inputs{x'values});
     output'delay = list_max(foreach x in inputs{x'delay}) + and_delay;
end module;

module or_comp
....
module nor_comp
....

.....
```

Figure 5.7: PDL Logic Gates Description

5.4.2 PDL Performance Evaluation

The delay traversing a given router module is based on the longest path delay of the underlying design. By default, these values are supplied as primitive parameters to the framework. However in the case of ASIC design, logic gate values may be supplied instead and the longest path delay calculated at the router module level, as described in section 5.3.

Delay is calculated from the longest path across a router. In the case of asynchronous control, each module calculates it's own longest path delay and these values are propagated across the carriers using hierarchical attribute property, the largest path delay of the router is the largest delay figure at it's outputs. The delay of the nets interconnecting modules and or routers will depend on whether a nominal delay is added to the analysis or not. That is whether this parameter is given a zero or non-zero value. It has been chosen to differentiate between those nets internal in the router and those between routers.

In the case of synchronous control, delay is not only dependent on the underlying logic but also the clock speed and the number of clock cycles required to obtain valid results at the outputs. The simplest method, although not necessarily the most efficient, is to clock each element at the speed of the slowest element. Estimates for maximum clocking for a given module are therefore supplied to a model instead of the longest path delay, such that the clock speed for a given router design may be chosen. The minimum of the speed of the slowest module and the maximum speed for the given technology is chosen as the clock speed in the analysis. Delay in the router is therefore based on the number of clock cycles required and the chosen clock speed.

At present, for ASIC design, cost analysis is based on the number of logic gates and is derived from the number of instances of the different gate types in the design. For more detailed cost analyses, the required details may be added to the technology data files and the cost expressions expanded in the PDL files. For FPGA technology, the number of CLBs has been chosen as the cost measure and these values are supplied in the technology data file.

Throughput in a network is traffic dependent. Traffic is represented in the network as value attributes. Traffic enters the network from the data file as primitive dynamic attributes of the input ports. These values are passed through the gate level design using the lateral propagation feature of PDL. At each gate encountered the resulting output value is dependent on the input values and the logic function of the gate.

5.4.3 PDL Design Files

The design file describes a specific communication system design at all levels of the hierarchy. As shown in figure 5.8, each module instance at the higher level is described in terms of the chosen submodule instances and a given interconnection pattern. Any number of a given module type may be included in the chosen set.

In terms of expressing this in PDL, a single design file of the form displayed in figure 5.9 is required. This design serves no purpose other than to demonstrate the coding style. The format of the code is that of the PDL net-list format. The implementation describes a four level hierarchy beginning at the logic gate level for each module of each router and concluding at the network

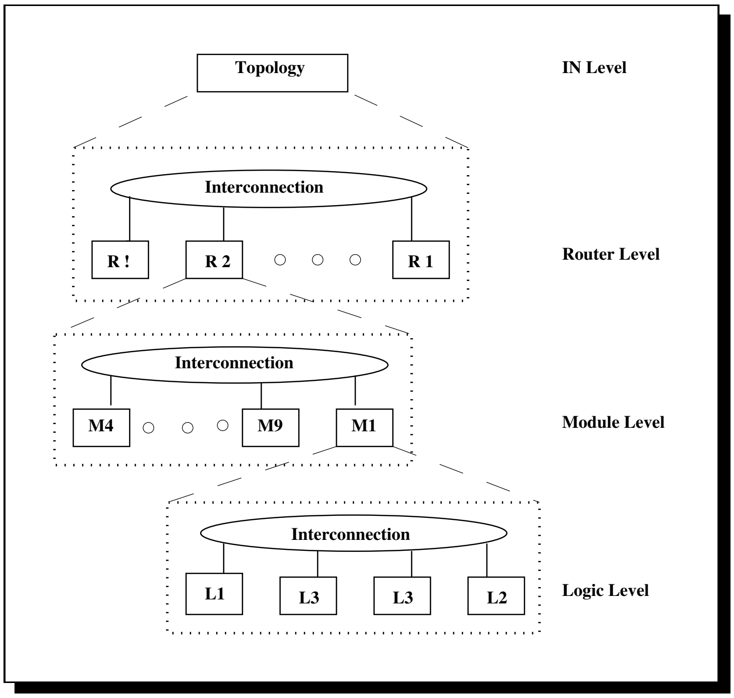

Figure 5.8: The Hierarchical Modelling Structure

level.

It should be noted that for any design instance, all sub-levels required must be written within the file itself that is that the '#include' concept of the PDL descriptions does not exist within the design file in PDL. That is that a bottom-up description must be given within a single file. Evaluation of a design begins at the lowest level and results are propagated up the hierarchy. If the design is not presented using a bottom-up methodology then evaluation will occur in the incorrect order giving rise to undesired incomplete results. As such, it is important to ensure that correct order is preserved. The logic gates of a module are declared before the declaration of the module. Likewise the modules of a router are declared before a router is declared. All router declarations are declared before the topology is declared.

This order restriction in PDL is due to the partial-evaluation property of PDL. That is, an incomplete set of data may be added to the design for evaluation and partial results obtained. However, if this is not the aim and it is the order of the design file which is incorrect,, then it is assumed those parts of the design file that are not available in the order of evaluation contribute to a partial evaluation instead. That is, their effects are ignored. The partial evaluation implementation feature of PDL also prevents feedback in designs. That is, only sequential designs have been provided for. However, this latter problem is under investigation. As such, the partial evaluation feature of PDL can be an asset if only part of the data is available but in this instance is a disadvantage.

In the bottom section of code, the network module itself is instantiated. This module describes a 4 router network where input and output ports for each router are provided at the interface of the topology. As such, it models 4 nodes, where 4 processors communicate through their respective routers with the network.

Every instance of a module at any level — logic gate, module set, router or network level, must have a unique name. That is the name is unique within the design file. This makes the development of large designs rather cumbersome but this problem has been reduced by using a simple Perl program which handles renaming of the modules.

There are two methods for expressing connections between modules at a given level. The first, as shown in figure 5.9, assumes adjacent ports. That is that identical attribute values appear at these ports. This is expressed in the form portname = modulename.portname. Identical attribute values may also be expressed in the form of the code in figure 5.10 by assigning the attributes of the ports to be equivalent in the carrier type definition. As such, either version may be used for the case where there are no delays on the carriers, whereas figure 5.10's style must be used if delays are to be added to carriers.

```
                module  and_comp = AND2Aa            -- logic gate submodules
                 ports
                 inputs = {I1 = AC1.in1, I2 = ..};   -- connecting module ports
                 outputs = out = outa;
                end module;
                module  and_comp = AND2Ab            -- logic gate submodule
                ......
        module addcomp = AC1                         -- ADDRESS COMPARATOR
         modules
         ANDS = {AND2Aa, AND2Ab.......};
         ORs = .....
         ...........
         ports
         inputs = {in1 = R1.bix1, in2 = R1.bix2,..};  -- connected module ports
         outputs = {out1, out2, out3};
        end module;
                module and_comp = AND2a               -- logic gate submodule
                .......
                module and_comp = AND2b               -- logic gate submodule
                 ports
                ..........
        module adddec = AD1                           -- ADDRESS DECREMENTER
         modules
         ANDs = {AND2a, AND2b....};
         ..........
         ports
         inputs = {in1 = AC1.outa, in2 = AC1.outb ...}; -- connected module ports
         outputs = {out1, out2, out3};
        end module;
        ...................                            -- OTHER MODULES
   module router = R1                                  -- ROUTER R1 design
    modules
    ACs = {AC1,...};
    ADs = {AD1....};
    ..........
    ports
    inputs = {bix11, bix12, ....};
    outputs = {box, ....., bop};
   end module;
   -----------------------------------------------------------------------------
                module  and_comp = AND2e               -- logic gate submodule
                ......
        module addcomp = AC2                           --ADDRESS COMPARATOR module
         modules
         ANDS = {AND2e.......};
         ...............
         .............                                 -- OTHER MODULES
   module router = R2                                  -- ROUTER R2 design
    modules
    ACs = {AC2,...};
    ..........
    ports
    inputs = {bix21 = ....};
    .........
   end module;
   ----------------------------------------------------------------------------
   .............                                       -- ROUTER R3 design
   ----------------------------------------------------------------------------
   ..................                                  -- ROUTER R4 design
   ----------------------------------------------------------------------------
module trc_topology = TRC1                             -- NETWORK design
 modules
 RTs = {R1, R2, R3, R4};
 ports
 inputs = {inp1= R1.bop, inp2...., inp3...., inp4....};
 outputs = {outp1, outp2, outp3, outp4};
end module;
```

Figure 5.9: A PDL Design File without Carriers

```
           module and_comp = AND2a
            ports
            inputs = {I1, I2}
            output = outa;
           end module;
           ...........
           carrier wire = M1                        -- equiv {I1 = AC1.in1}
            ports
            output = out = AC1.in1;
            input = in = AND2a.I1;
           end carrier;
           carrier wire  = M2
           .................
    module addcomp = AC1                            -- address comparator module
     modules
     ANDS = {AND2Aa, AND2Ab.......};
     ORs = .....
     ..........
     carriers
     wires = {M1, M2.....};
     ports
     inputs = {in1, in2,  ...};
     outputs = {out1, out2, out3};
    end module;
    ---- end AC1--------------------
    .........                                       -- other router modules
carrier wire = W1                                   -- equiv {in1 = R1.bix1}
 ports
 output = out = R.bix1;
 input = in = AC1.in1;
end carrier;
carrier wire = W2
........
module router = R
 --two component router
 modules
 MXs = MUX2;
 carriers    wires = {W1, ......};
 ports
 inputs = {bix1, bix2....};
 outputs = {box};
end module;
```

Figure 5.10: A PDL Design File using Carriers

5.4.4 VHDL Implementation

Each design instance of a module in the module set is described in VHDL or Verilog, compiled, simulated and synthesised. The resulting net-list is then converted into PDL net-list format using a Perl program — see appendix E, and added to the design set. The design set may hold one or more designs for each module.

Design Issues	**Parameter Change**	**Code Change**
Flit Size	x	
Packet Format		x
Addressing		x
Switching Technique		x
Buffer Size	x	
Routing Algorithm		x
Arbitration Method		x
Request Coding		x
Control		x
Number of VCs	x	x
Degree	x	x
Partitioning		x

Table 5.1: Design Issue Effects on Module Implementations

Each module is characterised with key parameters that effect the design of the module. Module refinement occurs through changes to variables or constants within the VHDL code, re-compilation and logic synthesis to produce a new design instance. It is often the case that a design change effects more than one module. As such, it was decided to put the design parameters into a package available from a library included in all the module descriptions. Where possible, all references to design decisions within the modules refer to the constants and variables of the package.

In section 4.2.1, the design issues effecting the complexity of each module was identified. As stated, those issues which may be used to parameterise the coding of a module are included in the design package of the framework. Table 5.1 highlights those design issues which are used as parameters and those which effect a more major design change in the modules of the module set.

A standardisation effort is in progress to standardise logic synthesis methods that is that logic synthesis tools following the standard will generate a similar net-list for a given design. However, the lack of completion of this standard means that each logic synthesis tool may give rise to different net-lists. As such, it is advisable that all modules in a design instance originate from the same logic synthesis tool or at least the effect of this feature be taken into account when results are analysed.

5.5 Using the Framework to Create a Design Instance

Figure 5.11 gives an overview of the steps required to create a design instance for a given router i.e. put together the design file and the associated data files.

To start with, the key design features should be identified i.e. the design parameter list updated.

From the design information provided — in the case of an existing design, or from a specification — in the case of a new design, the type of generic modules required can be identified. This information may take the form of circuit diagrams or written descriptions of the design. In the case of existing designs limited availability of information may make this task somewhat complicated. Different naming schemes also exist within routers and different features are lumped together adding to the complexity of this task. However, it is important to consider the functionality of each generic module and match these to the functionality offered within the router to be modelled. That is to look away from the names used in the design specification. This makes it easier to match areas of the design to the building blocks provided. Some blocks will be easier to identify than others. However, after these are identified it is easier to begin to picture the functionality or blocks required to complete the design. That is that the building block nature of the framework enables pieces of the design to be categorised and put together and the missing pieces identified i.e. similar to a jigsaw puzzle.

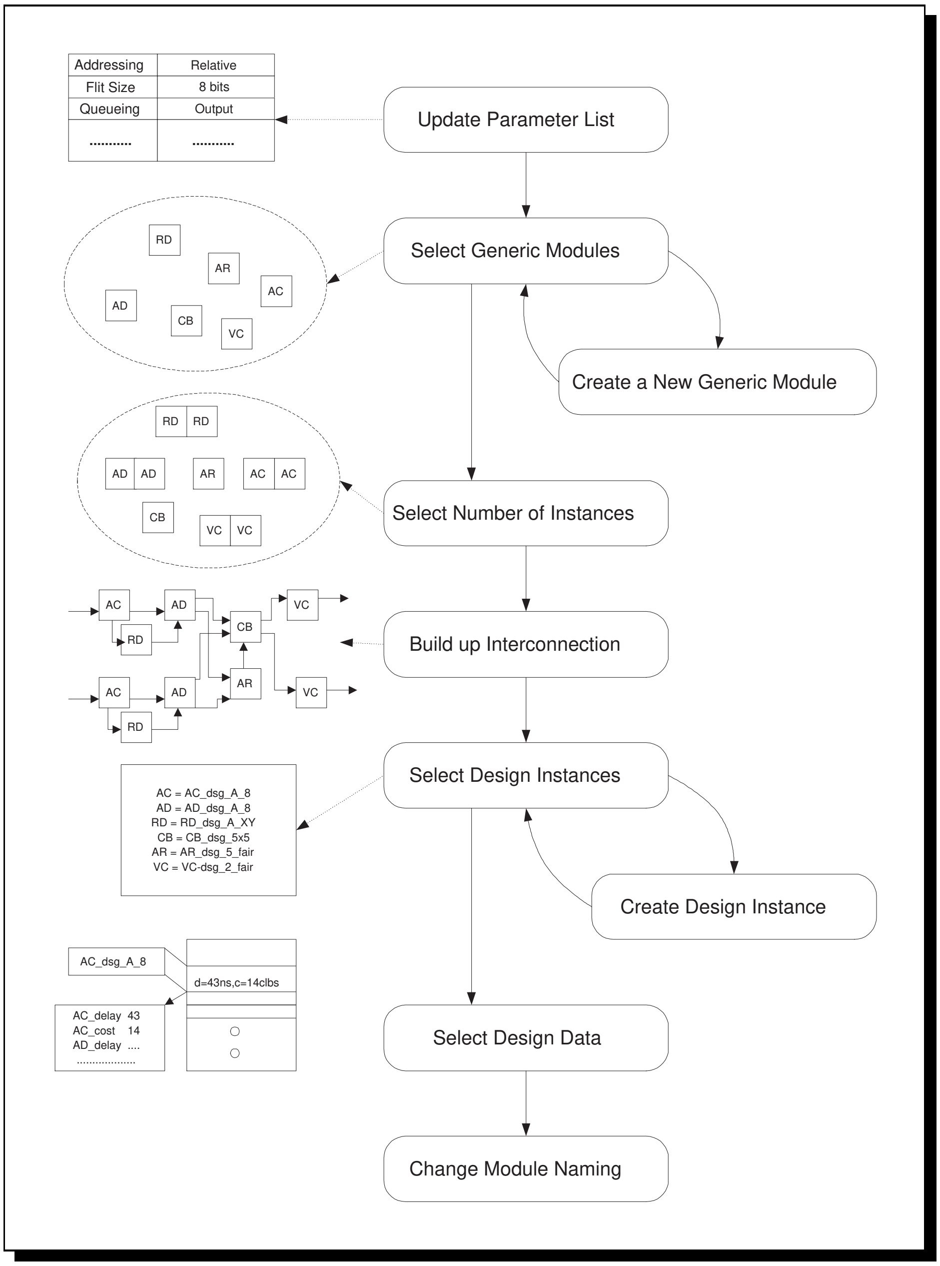

Figure 5.11: Choosing a Router Configuration

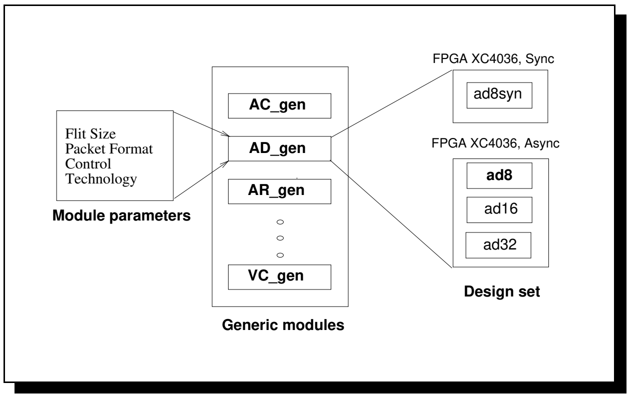

Figure 5.12: Relationship Between Parameter Choice and Module Design

If a function cannot be matched to an existing module then a new generic module is required. However, is should be borne in mind that a slight change in functionality will consist of a slightly expanded design and not necessarily a new module.

The next stage is to select the number of instances of each module required. Now the focus moves away from functionality to the number of paths through the router and those modules required on each path. At present this is not automated in the framework but the work of section 4.2.2, highlights the features to be taken into account when placing a module.

At this stage it is possible to piece together a system description using the generic modules and interconnecting them as required. That is that a PDL design file may be written for this design in the style described in section 5.4.3. Each instance of a module should be given a unique name.

The next stage is to pick a design for each of the generic modules chosen. Each module in the module set has an associated parameter list which effects the design of the given module. The module parameters are used to select a design from the set of designs available for the current module — as illustrated in figure 5.12. In the case illustrated, four designs are available for the address decrementer module (AD_gen) and 4 module parameters — flit size, packet format, control and technology, are used to realise the design choice. It should be noted that technology is not listed in table 4.3, with reference to the complexity of the address decrementer. This is due to the fact that all modules are effected by the technology choice. That is that only the latter modules are listed as being effected by the control choice. Another factor that effects all modules and as such is not listed in table 4.3 is the implementation technology. In the choice given, all designs are implemented in FPGA technology using XC4036 device. However designs using other devices or other technologies may be added to the design set such that the technology choice is used to select the correct design. It should be noted that automation of this selection feature has not been implemented as yet.

The next stage is to select the relevant design information from the design file. That is to select the appropriate delay and cost figures for each module type. These are then fed into a data file for inclusion in performance evaluation. Again, this feature is currently conducted manually. As shown, from the design name, the performance data may be found. This data is included in the design data file — see table A.1 in appendix A. The relevant data may then be extracted and added to the PDL data file – see data file example in appendix F.

If traffic effects are not being studied and performance data for cost and delay are available then it is sufficient that only the generic module declarations exist within the design file. However, if traffic is required the design file is updated with the details of each module instance i.e. design file includes gate level descriptions of each module. A Perl conversion program is used to aid module renaming such that all modules within the hierarchy have unique names.

At this point the design file is complete and the router configured. If the performance expressions provided within the framework are suitable for the performance analysis required then the pdl model is ready for compilation and evaluation. Otherwise the performance expressions may be adapted by altering the evaluation rules within the PDL descriptions. The generic file (.pdl) is then compiled along with the design file (.dsg) and the result is evaluated using the data files.

5.5.1 Creating New Generic Modules

In section 1.1, the possibility of moving part of the routing functionality from the software level to the hardware level was introduced. This may involve expansion of existing modules to handle extra features or it may involve a new function not currently handled in the module set. That is that a new generic module is required.

The process of generic module creation is simple as generic modules basically differ in their attribute declarations and associated evaluation rules. As such, a copy is made of an existing generic module (.pdl file) and those router features which effect the design of the new module are added as attributes to the new module. The module is then added to the module set and the PDL router description updated to include submodules of the new type. In addition, at least one new design for the new module must be added to the design set – see section 5.5.2.

5.5.2 Creating New Module Design Instances

If a design already exists for a given module then two possible methods are used for creating a new design. The simplest is the case where the design feature to be changed is held in the design package. That is that any update to the package will effect the variables of the given design and therefore a new design is created by updating the package, recompiling the given module(s) and re-synthesising to create the required net-lists. These net-lists are then converted to pdl net-list format using the Perl program created for this purpose.

The second method requires recoding of the VHDL description. In this case a design change alters a feature not included in the package that is it effects the behaviour of the module. An example of this may be changing the arbitration method in the arbiter. Here, an existing arbiter design may be copied and those parts of the code effected by this design change rewritten. Although this method involves more work than the first method, the starting point of a correct working module aids faster construction of a new design — basically the benefits of reuse.

If a new generic module is created then a new design must be created. This involves creating the code from start. However, especially for the asynchronous case, the presence of module examples aids coding of a new module.

Part III

Case Studies, Ongoing Work, Conclusions and Future work

Chapter 6

Case Studies

The following case studies have been chosen to highlight the key features of the framework.

The first case study models an existing router design, the Torus Router Chip (TRC). The TRC is well referenced in the literature and, as such, may be regarded as one of the better known designs. This example focuses on the gate level modelling aspect of the framework. An additional element to this case study is the asynchronous nature of the TRC. To implement a synchronous router at this level of detail is a much simpler task. Therefore, for the framework to be said to be generic it was important to illustrate that both synchronous and asynchronous designs could be modelled.

The aim of Case 2 is to illustrate the design space exploration feature of the framework. Varying one or more of the design issue decisions may lead to alteration of one or more modules or their interconnection i.e. give rise to new design instance. The TRC design of Case 1 is used as the base design for this study and the effect of design changes on the performance and cost of the router are presented.

Case 3 takes a closer look at the problems of modelling an existing router based on the limited information available for this task. It takes, as an example, a more recent router developed at the Massachusetts Institute of Technology (MIT), termed the Reliable Router. This router is synchronous and includes many newer features in router design.

These first three case studies have been concerned with the case of a single router chip. However, a router chip exists as part of an interconnection network. As such, it is important to consider the design changes above in light of the network environment. Therefore, in Case 4, network analysis is presented using the TRC of Case 1 as the base design. This case illustrates calibration of network level performance using gate-level modelling giving rise to increased accuracy in performance predictions.

A typical performance analysis of interconnection networks seen in the literature is comparison between different switching techniques. Case 5 presents a similar analysis except that the gate-level modelling again provides performance data that increases the accuracy of the results presented.

FPGA technology is chosen as the implementation technology for the following case studies. Each module design described, has been implemented using VHDL'93 and compiled, simulated and synthesised using Foundation. Timing information for the given designs is based on the best case routing assumption described in section 5.2.

In this work certain assumptions have been made as to the actual hardware implementation of the designs due to the limited availability of detailed documentation[1]. In addition, FPGA technology has been used as the implementation technology which differs from the original technology of the designs modelled. As such, comparison between the original implementations and the modelled results is not considered to be meaningful and, as such. are not included in the analysis presented.

6.1 Case 1: Torus Routing Chip

In section 2.5, a number of features applicable to existing and proposed routers are identified and compared. In section 4.3 a more detailed description of a number of these routers is given with respect to the modules required and their interconnection. In this case study, router analysis is taken one step further to the detailed design instances of the individual modules. An instance of the model based on an existing router — the Torus Routing Chip (TRC), is described.

[1] Repeated attempts to obtain more detailed design information have been unsuccessful.

In the implementation method chosen, where HDL code is written to describe the behaviour of the individual modules, it was found that both the creation of code that was synthesisable and secondly that behaved correctly was a challenge in asynchronous design.

It was decided, therefore, to include detailed descriptions of each of the modules of the TRC together with the performance data achievable. This additional information is aimed to give the reader a clearer picture of the workings of the various modules and an insight into a number of the problems incurred in asynchronous design.

As the focus of this section is on the gate level designs then a bottom-up approach to presentation of the model is used. That is in the reverse of the model development process illustrated in figure 5.11 of section 5.5.

6.1.1 The TRC Modules

In this section the modules of the TRC implementation are described in detail. It is chosen to present the modules in order of appearance within the router from input to output. As described in section 5.3 each module is implemented in VHDL and synthesised to a gate-level description.

Each of the modules is described with reference to the design issue decisions made. An overview of these design issues and their effects on the router modules can be found in section 4.2.1.20. It should be borne in mind that all the modules and their interconnections operate in an asynchronous manner.

The numbering system used in the figures of this section refer to the order that signals appear. For those modules whose behaviour is described by a finite state machines, an enable signal, EN replaces the clock of a clocked state machine and is set high. Therefore this signal has number '0' representing that it is always set. It is assumed that reset is the first signal to appear and thus is given the number '1' and therefore numbering of other signals begins at the number two. Note that the order of signal clearing is not given in the figures for simplicity of understanding.

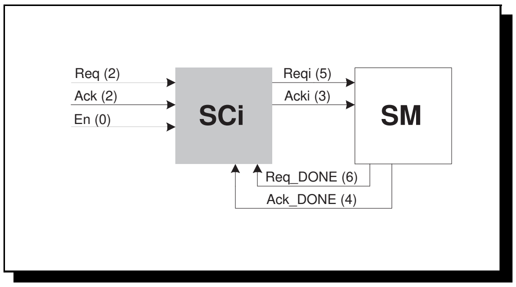

Figure 6.1: TRC Signal Converter Input Module

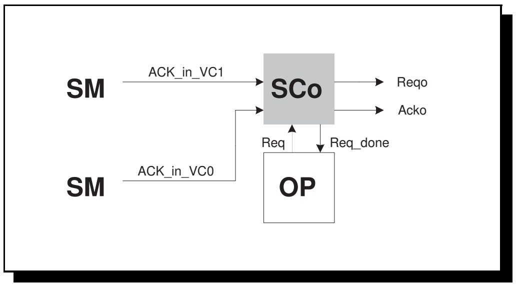

Figure 6.2: TRC Signal Converter Output Module

6.1.1.1 Signal Converter

The design issue 'control' effects the design of the signal converter. The TRC is fully asynchronous and uses two-phase signalling for external connection and four-phase signalling for internal connections. As such, two implementations of the signal converter are required, one at the input i.e. two-phase to four-phase, and one at the output i.e. four-phase to two-phase. These are illustrated in figures 6.1 and 6.2 respectively.

In both cases, if a request and an acknowledgement are received simultaneously the acknowledgement is processed first. This prioritises freeing space in the output buffers before accepting new flits which require buffer space. As such, it supports forward progress. Figure 6.3, illustrates this process for the case of the input module. As shown, if an acknowledgement is received then the state machine processes the acknowledgement, states Ack_send and UpdateA_state, and then checks for outstanding requests to process before returning to the idle state, the Wait_state.

6.1.1.2 Signal Manager

The functionality of the signal manager may be split into two parts: firstly controlling receipt and secondly controlling transmission of flits. Due to the signalling convention implemented in the TRC, as described above, the signal manager does not receive or send signals directly from or to the external channels. Instead it communicates signals through the signal converters. Control of receipt also involves control of internal modules at the input to

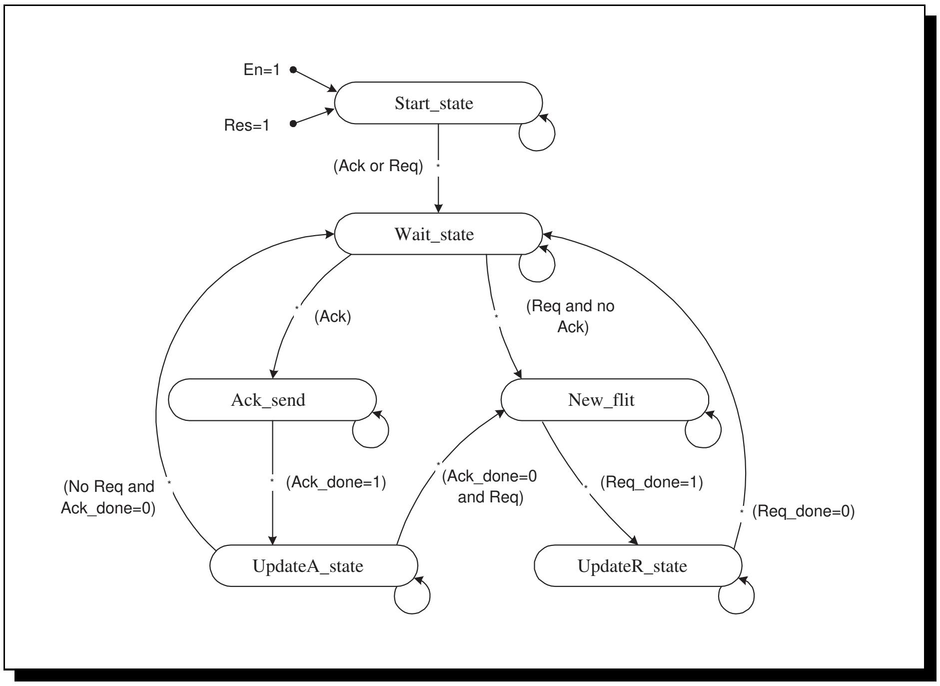

Figure 6.3: An FSM Description of the TRC Signal Converter Input Module

the router i.e. at the input side of the crossbar. Figure 6.4 illustrates the interaction of the signal manager with it's neighbouring modules.

The operation of the Signal Manager is illustrated in the finite state machine of figure 6.5. If both a request and acknowledgement are received at the same time, the acknowledgement is processed first, as in the signal converter modules.

On receipt of a flit i.e. a request from the signal converter, it then enables the buffer and address comparator. The signal manager then gives over control to the RD module and awaits acknowledgement from the RD module, confirming completion of the receipt of the flit. This allows the SM to send an acknowledgement to the output signal controller for forwarding to the sending node. In addition, it sends a request done to the SCi to signal completion at the input.

Processing of an external acknowledgement involves the forwarding of the

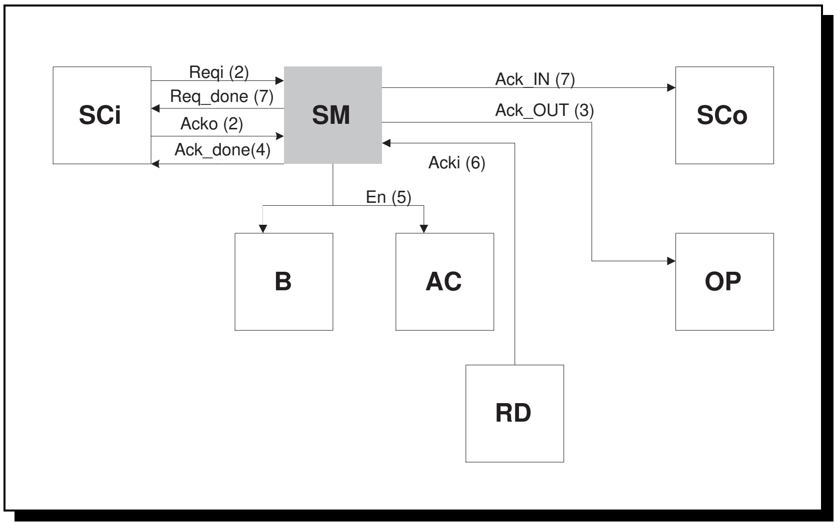

Figure 6.4: TRC Signal Manager

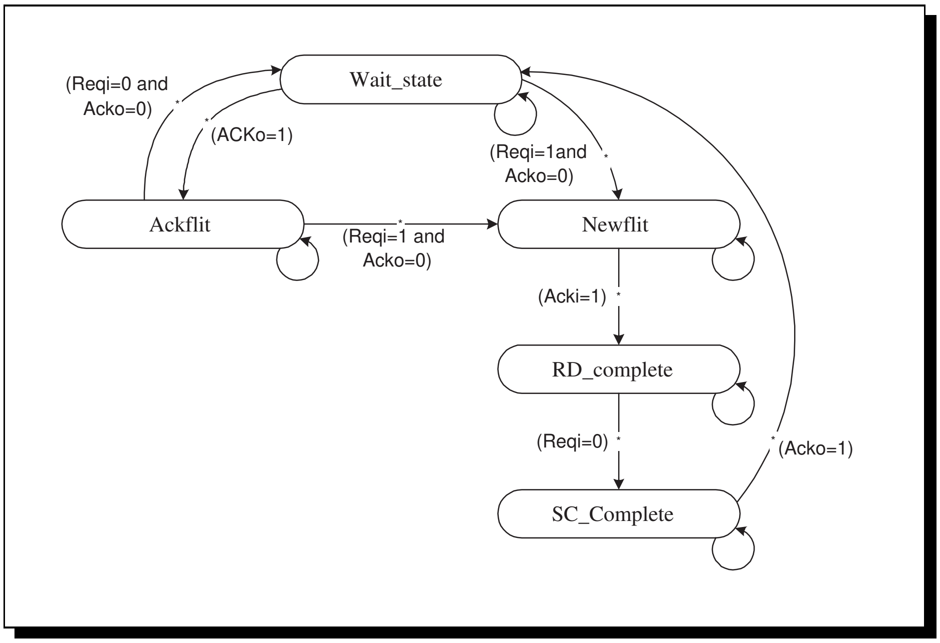

Figure 6.5: A FSM Description of the TRC Signal Manager Module

acknowledgement to the output module to confirm that a flit sent from the node has been received by it's neighbour. Again a signal, Ack-done, is sent to the SCi module to confirm completion of the operation.

Under testing, the importance of not only setting but also clearing the signals in the correct order was identified as an important consideration to obtaining correct functionality. This factor was particularly noticeable when the signal manager was connected to other modules during testing.

6.1.1.3 Buffer

The buffer module is a simple latch used to receive and hold each flit during input processing. With a flit size of 8, an 8-bit latch is required. The buffer module interconnections are shown in figure 6.6. Enabling of the buffer module is under the control of the signal manager.

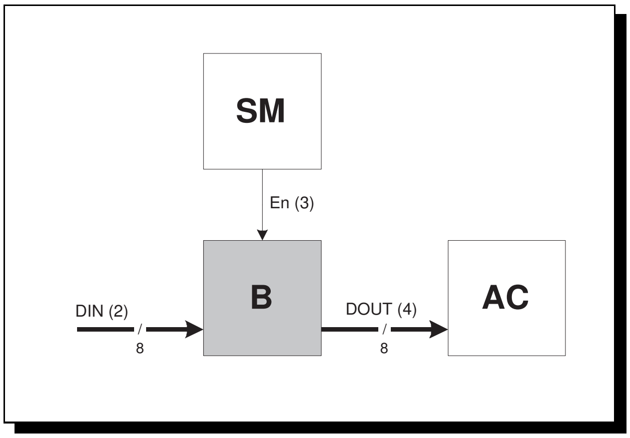

Figure 6.6: TRC Input buffer

6.1.1.4 Address Comparator

Interconnections to the AC module are shown in figure 6.7. Relative addressing requires a zero-checker implementation of the address comparator.

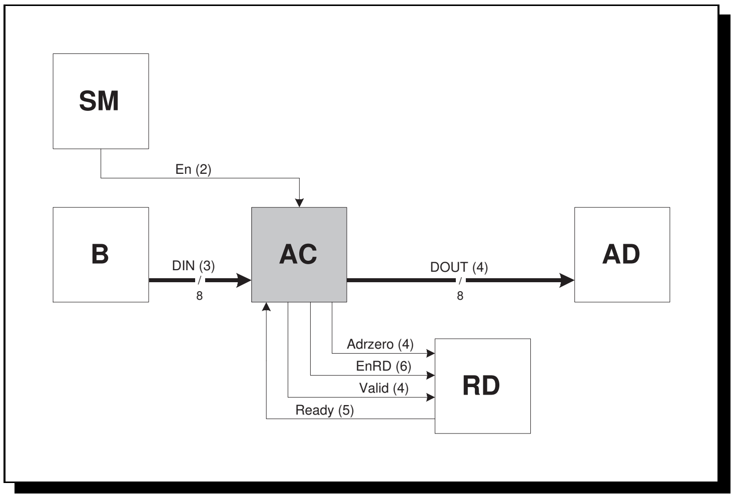

Figure 6.7: TRC Address Comparator

In this case, the packet format provides a single flit address for each dimension. As such, the HDL description is parameterised by the flitsize and therefore describes an 8 bit zero-checker.

Again this module is under the control of the signal manager and the decision made within the module is communicated to the RD module. However, the communication of the result presented a problem. At first, an internal solution was attempted. The decision was sent to the RD module from one process within the AC. A control signal from the same process was used to drive another process, again within the AC, which issued an enable to the RD module indicating a valid result. Testing of the module indicated correct functionality. However, when the AC module was tested in combination with other input modules of the TRC a problem was identified using this approach. The internal control of the enable signal, EnRD, could not be seen from outside and therefore under simulation an inconsistence was seen in the order of receipt of these signals.

To solve this problem a handshake was introduced between these two modules: the signals Valid — telling the RD module that the decision had been made and the signal Ready telling the AC module that the RD module has received the decision signal. After completion of the handshake, the enable signal(EnRD) is sent.

6.1.1.5 Routing Decision

The TRC uses a deterministic x-y routing algorithm involving two virtual channels (VC1 and VC0) in each dimension for deadlock avoidance. The algorithm states that a packet is transmitted first in x until the destination x address is reached and then in y to the destination. Deadlock avoidance requires that packets which are continuing in the same dimension or moving to a new dimension are allocated VC1 unless an edge node is reached. At an edge node the packet moves to VC0 and continues in VC0 until it reaches the destination or moves to a new dimension.

A distributed routing decision is implemented in the TRC. To implement the above algorithm, each RD module must be given a path identifier and an edge ID for the given router. The edgeID is parameterised by the 'degree' i.e. whether it is an edge in the x, y or both dimensions. All path references are parameterised with 'Datapaths'.

Unlike the modules described above, the RD module must be enabled for the length of a packet. The module is implemented as an asynchronous finite state machine where enable is always set. At reset, the RD module moves into the Wait_state where no flits have been received. Processing within the state machine continues until after the tail packet is received. After the tail is processed then the state machine returns to the Wait_state. It should be noted that since the algorithm is deterministic then only one output choice is available for a given destination. This means that if a full signal is received, indicating that there is no space in the output buffer then the state machine waits on the current state until the full signal is disabled.

During implementation of the state machine many problems arose due to the definition of the next state update process. At first, the clocked enable of the next state was removed. Although this did not prevent the state machine from functioning, the behaviour was inconsistent with that expected. Eventually a new solution was introduced. Instead of removing the clocked next state, it was replaced by the enable signal which unlike the clock has a steady value. As such, it does not drive the state-machine but allows the states to drive themselves i.e. asynchronous.

The state machine description is given in figure 6.9. As shown, a large number of states are required to handle the operation of the state machine.

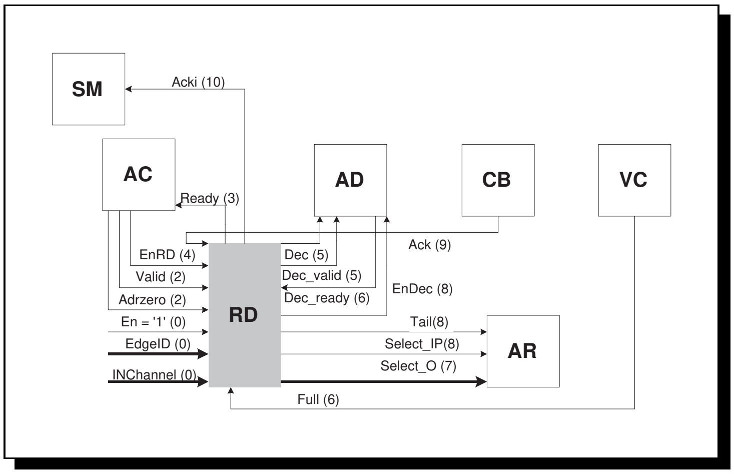

Figure 6.8: TRC Routing Decision Module

The original implementation involved fewer states and was shown to operate correctly under module testing. However, when tested together with the other modules, many additional states were added to handle the handshaking problems which arose.

Figure 6.8 illustrates the RD module's communication with neighbouring modules. Acknowledgement from the CB module indicates completion of a flit transfer enabling the RD module to send an acknowledgement to the SM module. On receipt of the tail flit the RD module sends a tail signal to the AR module indicating that the reserved output path may be released, see section 6.1.1.7.

A VHDL implementation of the routing decision module may be found in appendix B.

6.1.1.6 Address Decrementer

An additional feature to the expected decrement function, of the AD module, is the generation of a request signal. That is, a data valid signal for the data to be forwarded across the crossbar. The AD module is shown in figure 6.10.

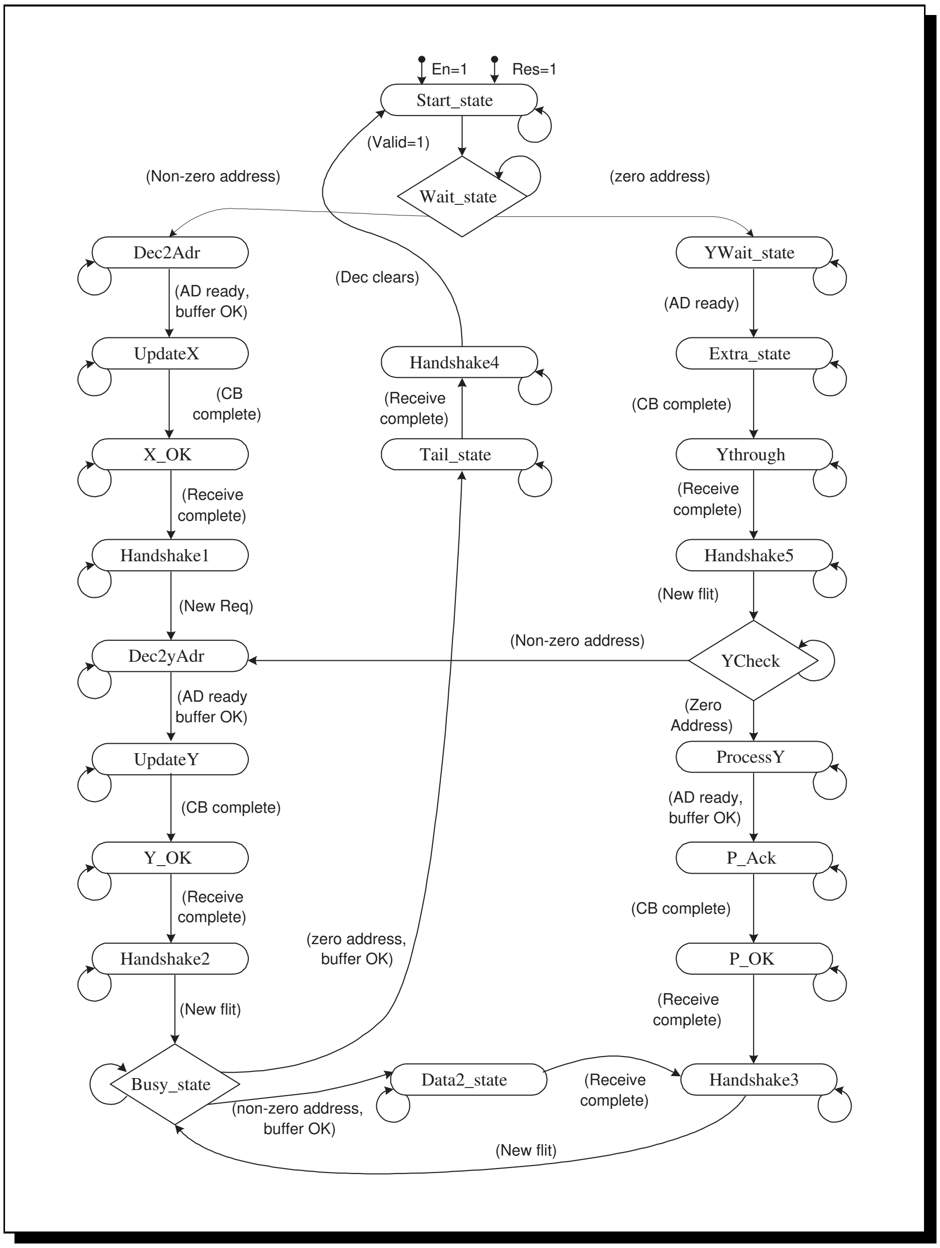

Figure 6.9: A FSM Description of the TRC Routing Decision Module

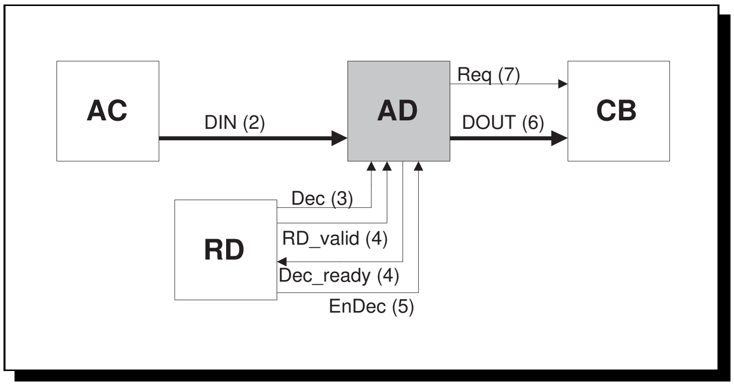

Figure 6.10: TRC Address Decrementer

Again, as in the AC module, a number of signals are used for handshaking between the RD and, in this case, the AD module. The handshaking is similar to that of the AC handshake but involves the signals Dec (RD decision) and RD_valid, Dec_ready and EnDec to complete the handshake. It should be noted that the AD module is not enabled by the SM module and is therefore under the control of the RD module. Also, since the AD module is placed on the data path then it is enabled for each flit received whether the decrement operation is required or not. The AD module is parameterised by the flitsize and single flit addressing per dimension requires an 8 bit address decrementer.

6.1.1.7 Arbiter

In the TRC, the crossbar arbiter is implemented in the form of small arbitration units within the crossbar implementation itself. However, for simplicity and to reserve a clear definition between the different modules, the arbiter is implemented as a single arbiter module. Since the TRC uses a single crossbar module connecting all the input paths to output paths, a single arbiter module is required.

Output channel selection and buffer availability is under the control of the RD module. As such, receipt of a request implies that the output channel requested is either available or at least there is buffer space at the output. The arbiter is, therefore, left with the jobs of selecting between the inputs with pending requests and advising the crossbar of the selected input and

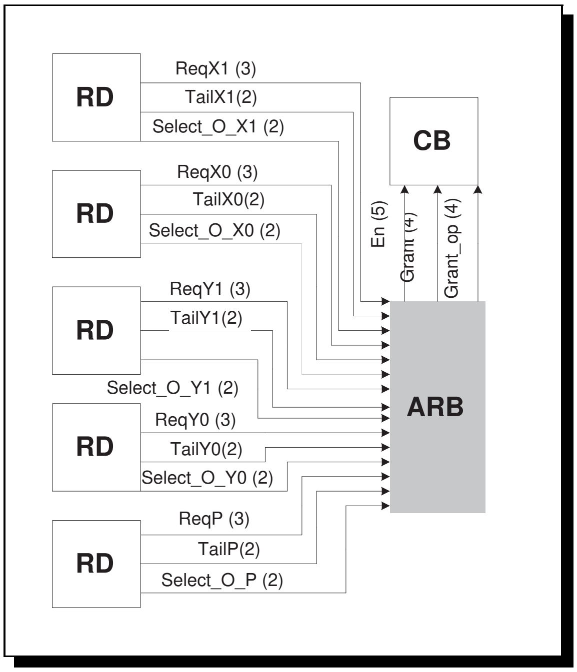

Figure 6.11: TRC Arbiter

it's requested output.

It should be noted that the TRC uses the wormhole routing switching technique. This means that the arbiter, arbitrates on a flit basis but must reserve the selected output channel for the entire packet. Linear priority arbitration is used to arbitrate between the pending requests. As virtual channel multiplexing occurs at the output, requests are received for each of the virtual channels competing for the use of the crossbar as well as that of the processor. The crossbar is not partitioned and therefore a single arbiter arbitrates between all the requests and the AR module is parameterised by the number of paths crossing the CB, DataPaths. As shown in figure 6.11, an enable signal is sent to the crossbar providing a means to disable the crossbar during arbitration.

In the documentation available for the TRC, a 6 wire select channel is described for each routing decision model. Due to the lack of information to the contrary, these lines are implemented as a single request and 5 select lines.

6.1.1.8 Crossbar

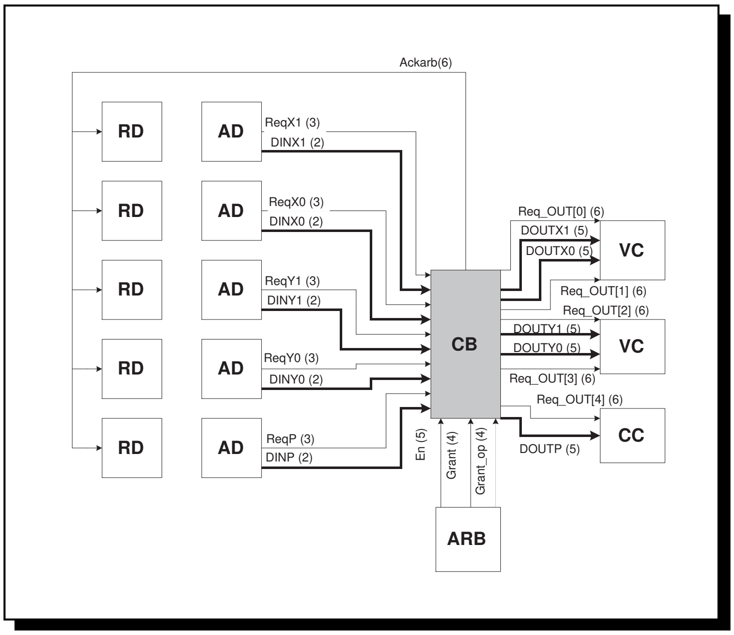

Figure 6.12: TRC Crossbar Module

The crossbar forwards data from the granted input to the selected output under control of the AR module. As shown in figure 6.12, a request signal is generated after the output data is driven. However, as buffer space is assured by the RD module then this request is a valid data signal for the neighbouring module and not a request for space.

The HDL description is parameterised by 'datapaths' representing the number of paths crossing the CB module and the 'flitsize' representing the path widths. A non-partitioned 5×5 crossbar with a path width of 8 is implemented.

6.1.1.9 Virtual Channel Controller

The interaction of the virtual channel controller with the other modules is illustrated in figure 6.13. As described in section 4.2.1.19 and illustrated in

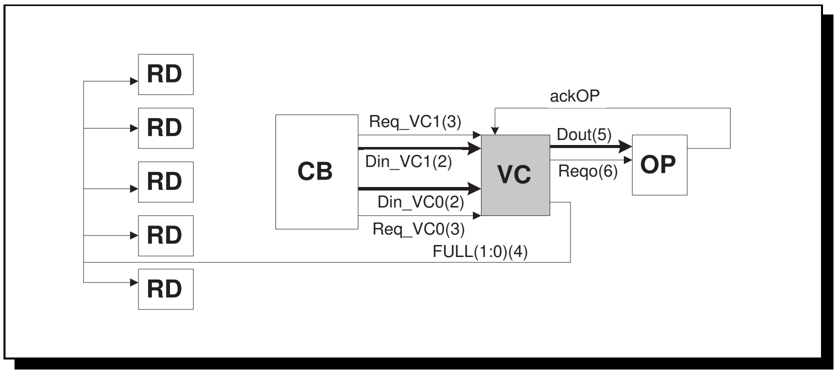

Figure 6.13: TRC VC Module

figure 4.2 the VC module includes buffering, multiplexing and control of the virtual channels for a physical channel.

The buffers of the VC module are eight bits wide and four bits deep and are implemented as FIFO queues, parameterised in the HDL descriptions using 'flitsize' and 'fifoblocks' respectively.

In the implementation it is assumed that no incoming requests are received after a full signal is sent from a given buffer as this is under the control of the sending RD module. All full signals are transmitted to all RD modules, as shown. Incoming requests are handled after processing of outgoing data to support forward progress. That is that acknowledgements to the FIFO are handled before incoming data.

As soon as the first block in the FIFO is ready i.e. has accepted data, the data is driven and a request is sent to the multiplexer. Requests from the FIFOs are sent to the multiplexer controller when data is ready to be sent i.e. the first flit of a buffer is full. The controller operates on a fair arbitration policy for wormhole routing. This means that i.e. the channel alternates on a flit basis when both FIFOs have data ready to send. The number of requests competing for the arbiter is equivalent to the number of FIFOs i.e. the number of virtual channels.

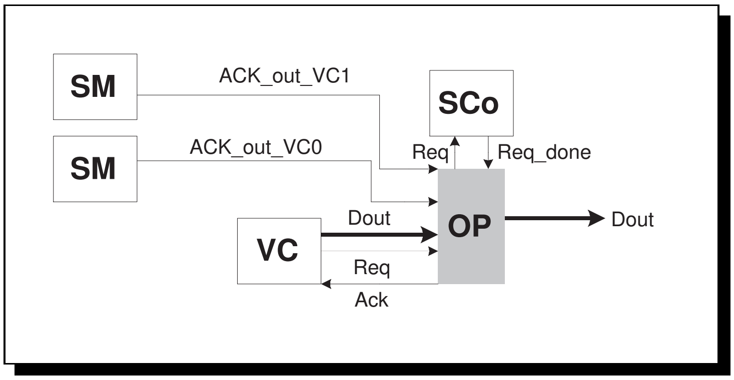

Figure 6.14: TRC Output Module

6.1.1.10 Output Modules

The output module controls the sending of data to the output pad for driving onto the external channel. If previously sent data has not been acknowledged then it awaits the acknowledgement from the signal manager before sending. After data is forwarded, a request (data valid) signal is sent to the signal converter for conversion and transmission. An acknowledgement is also returned to the VC module after completion. The output module is illustrated in figure 6.14.

In this case, no arbitration for the external channels is required as the external channels are unidirectional.

6.1.1.11 Consumption Channel

The consumption channel module buffers and controls the transmission of data from the TRC to the local processor. Due to lack of information to the contrary, control is chosen to be in the form of request/acknowledge signalling. However, unlike the case of the OP modules, no signal conversion is required.

The consumption channel, as shown in figure 6.15, is a FIFO with the same controls and size as those included in the FIFO modules of the VC module i.e. 8 bits wide and 4 bits deep. On receipt of data and a valid request from

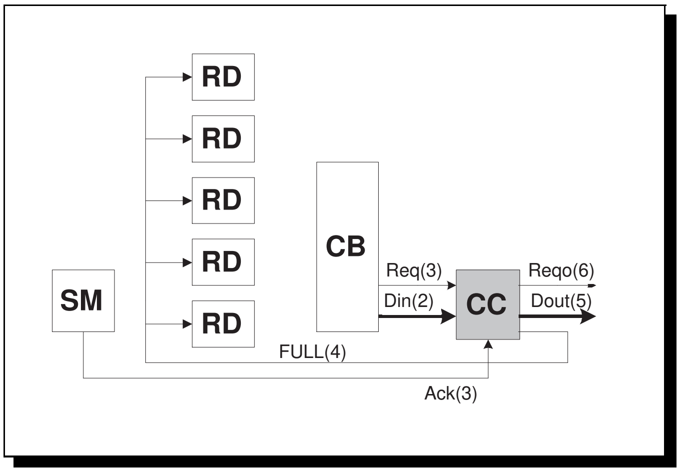

Figure 6.15: TRC Consumption Channel

the crossbar, data is stored in the buffer. As soon as the first block is filled, a request is forwarded, this time to the processor. Acknowledgement from the processor, updates the FIFO and drives any new data onto the channel. To prevent buffer overflow, a full signal is sent to the RD modules, stopping further requests addressed to the buffer.

6.1.2 Router Level Design Description of the TRC

A router level description involves an interconnection of modules. As stated in section 4.2.2, the number of instances of each module depends on a number of design issues. Table 6.1 summarises the choices made in the TRC and the resulting module instances required. It should be noted that although there is a single injection and consumption channel, no injection module is required. This is due to the fact that there is no queueing at the entrance to the network and therefore the injection is represented by a channel. However, queueing at the output exists and therefore a CC module is required. It is also assumed that signal conversion is not required at the interface to the processor and as such no SC modules are required on the processor path.

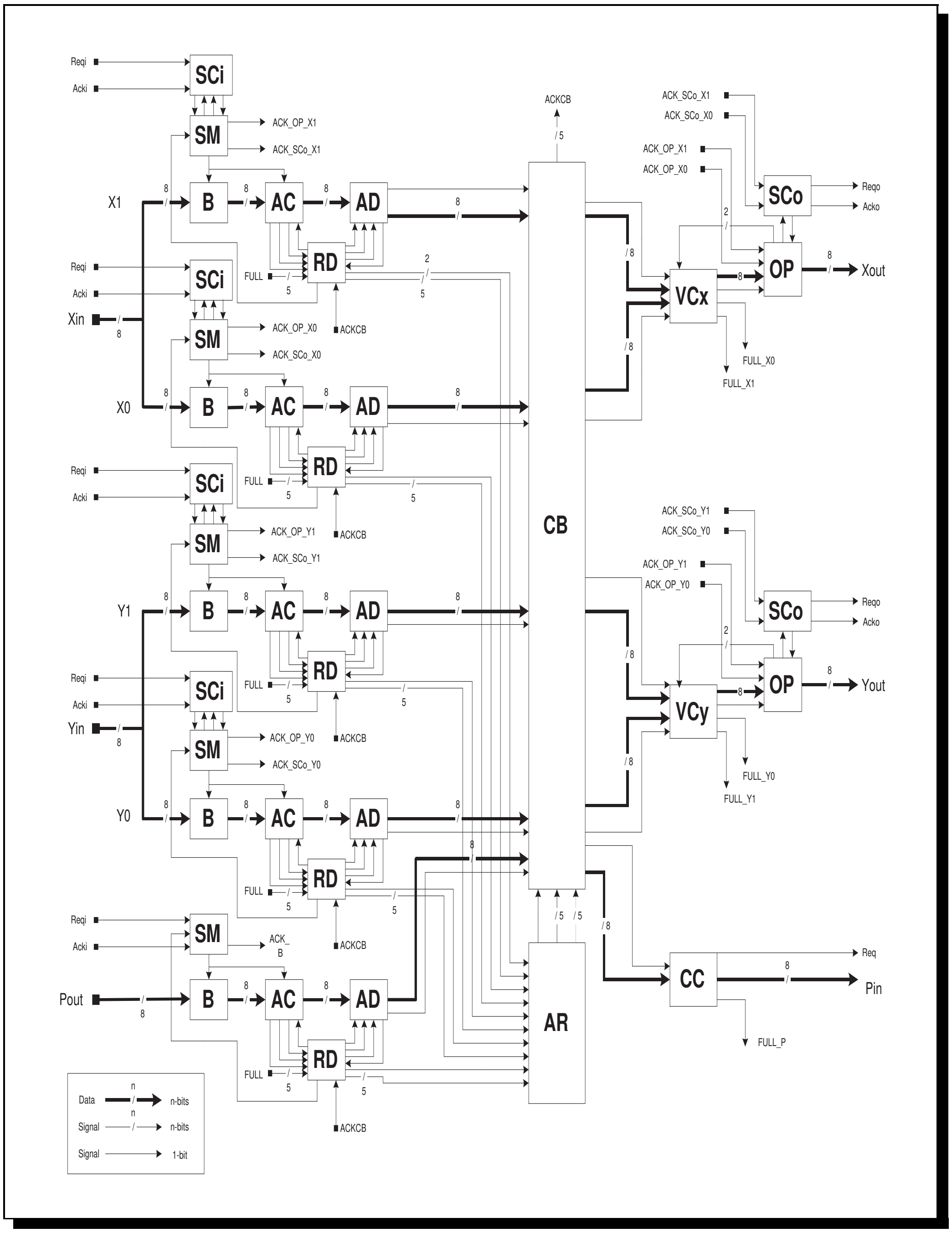

Figure 6.16: The Torus Routing Chip Design

Issues	**Components**											
	AC	AD	AR	B	CB	CC	RD	SCi	SCo	SM	OP	VC
Degree	2	2		2			2	2	2	2	2	2
Number of VCs	2	2		2			2	2		2		
Partitioning			1		1							
No. of Inj/Cons	1	1		1		1	1			1		
Total	5	5	1	5	1	1	5	4	2	5	2	2

Table 6.1: Number of Units Required of the TRC Modules

An overview of the TRC design expressed as an interconnection of modules is given in figure 6.16. As such, it is a pictorial representation of a PDL design file where the design is expressed as a single router module made up of an interconnection of submodules.

It should be noted that in figure 6.16 that a number of wires have been split and identical names used to ease the readability of the figure. These names may differ slightly from those used in the descriptions of the individual modules given in section 6.1.1 but are used to make the figure more understandable. It should also be noted that constant input signals such as En, EdgeID and Inchannel are omitted from figure 6.16 for readability.

6.1.3 Testing

The TRC implementation is a two level hierarchical design description and as such testing began with testing at the lowest level of the design — the modules.

6.1.3.1 Module Level Testing

For each module, correct logic synthesis of the VHDL register-transfer level description is assumed as this operation is undertaken by the logic synthesis tool — Foundation. Behavioural correctness of the synthesised modules is, however, not assumed. Each module has been simulated after synthesis to test for correct signal behaviour. This identified many errors within the asynchronous behaviour of the modules.

Incoming Channel	**Available Outgoing Channels**
P_OUT	P_IN, Xo_VC1, Yo_VC1
Xi_VC1	P_IN, Xo_VC1, Xo_VC0, Yo_VC1
Xi_VC0	P_IN, Xo_VC0, Yo_VC1
Yi_VC1	P_IN, Yo_VC1, Yo_VC0
Yi_VC0	P_IN, Yo_VC0

Table 6.2: Test Set 1 : Output Choices under Deterministic Routing

Due to the difficulties involved in asynchronous design and the ease of fault correction and re-testing individual modules in the same environment, it was chosen to test a complete path of the design within Foundation. That is to join the modules together with the use of the schematic capture tool and test the asynchronous behaviour with the aid of the graphical simulation environment.

Testing of the complete path followed a stepwise order. That is that one module at a time was added to the simulated circuit, after individual testing, and testing of the expanding design was undertaken. This stepwise process aided both understanding of the asynchronous behaviour and easier identification of problems within the asynchronous design. It was soon discovered that although a module may appear to operate correctly alone, the asynchronous external signals may not arrive from the neighbouring nodes as and when expected and thus effect the asynchronous behaviour of the module. Many errors were identified during this process.

6.1.3.2 Router Testing

To test the complete router design it is important to ensure that the router responds to incoming messages in the manner described in the specification. Response in this situation refers to the order that incoming messages are handled and the choice of outgoing channel.

As stated earlier, the routing algorithm implemented in the TRC is a form of restricted routing. This means that not all outgoing channels are available for a given incoming packet. The possible outgoing channels for a given incoming

Incoming Channels	**Outgoing Channel**
P_OUT, Xi_VC1, Xi_VC0, Yi_VC1, Yi_VC0	P_IN
P_OUT, Xi_VC1, Yi_VC1	P_IN
Xi_VC1, Xi_VC0	Xo_VC0
P_OUT, Xi_VC1, Xi_VC0, Yi_VC1	Yo_VC1
P_OUT, Yi_VC1, Yi_VC0	Yo_VC0

Table 6.3: Test Set 2 : Arbitration Priority for Competing Input Channels

packet were therefore identified based on the routing algorithm, as shown in table 6.2. It should be noted that P_OUT is out from the processor and in to the router whereas P_IN is into the processor and out of the router. The entries in the table, except for the processor channel are of the form (external channel)(input/output)_(virtual channel) e.g. Xi_VC1. Taking this channel as an example, a packet may be at it's destination (P_IN), continuing forward (Xo_VC1), reached an edge and forwarding (Xo_VC0) or changing direction (Yo_VC1). For the purpose of these tests an unloaded network is assumed. A loaded network does not change the correctness of the routing decision for a deterministic routing algorithm.

For the purposes of this test, the modules of a single virtual channel path at the input to the crossbar were interconnected. The input path was attached to different inputs of the crossbar and it's arbiter to create the part of the design to be tested. Note that arbitration from several sources could also be tested by adding the appropriate stimulus to the unattached arbiter inputs. An interactive simulator enabled correctness checking one cycle at a time. It was felt that this was beneficial both to testing but also in understanding the asynchronous nature of the design.

The second set of tests, listed in table 6.3, involved a loaded network with more than one packet competing for the same output channel. Again those incoming channels which may compete for a given output channel were identified and requests sent from each of the required input channels to the given output channel. The arrival sequence in the results highlighted the correctness of the arbitration implementation.

These tests involve the central crossbar and arbiter and as such it is enough to be able to enter data into different crossbar channels in parallel and test the reaction of the arbiter both for selection of the correct output and also to give correct priority to the input.

A second arbitration level exists within the router at the multiplexor controller. This module controls the selection between the two virtual channels which share a physical channel. If both VCs have packets waiting, then multiplexing between the packets occurs on a flit basis. That is that each may send one flit before control is passed to the other waiting FIFO. Testing of this arbitration level was conducted at the module level as the control, buffering and multiplexing of virtual channels are collected together within the VC module.

6.1.3.3 Limitation with the Current Testing Method

No independent testing of the design has been conducted. That is that the designer has both created the behavioural descriptions and created and followed through the testing procedure. Independent testing is an important feature in architecture development and was desirable for this work. However, lack of assistance within the scope of this work did not enable this feature to be undertaken.

6.1.4 Performance and Cost Analyses of the TRC Modules

Estimates for the cost and delay for each of the modules in the TRC design is given in table 6.4. The XC4036XL device using package PG11 which includes 1296 CLBs and 288 IOBs was used as the implementation device.

Delay and cost results for the individual modules were obtained after mapping to the chosen device from Foundation using level 1 (highest) implementation effort and balanced optimisation. Optimisation for speed or area would have given rise to results biased towards the respective optimisation goal.

Symbol	**8 bit** (# clbs)	**LPD** (ns)
AC	5	7.086
AD	4	6.536
AR	43	12.954
B	4	5.448
CB	71	10.12
CC	88	19.55
OP	5	4.847
RD	43	33.932
SCi	9	7.756
SCo	13	9.415
SM	5	13.695
VC		
— Fifo	88	19.550
— Mx	5	4.898
— Ctrl	1	4.919

Table 6.4: Torus Routing Chip Module Performance and Cost Data

A feature of the optimisation process is the removal of redundant logic. This may cause problems if the logic removed is desirable for the module's correct functionality. A check of the number of IOBs in the optimised design against those specified in the VHDL code ensured that no valuable logic was removed.

Results obtained from longest path delay analysis of a given module may include pad delays. This is due to the fact that each module was analysed alone on an FPGA device. Subtraction of the relevant pad delay(s) from the longest path delay figures obtained was therefore made where necessary. The data for Tpid (at the inputs) and Tops (at the outputs) is available both in the XILINX data book and from the delay report in Foundation itself. These figures are 1.18ns and 5.06ns respectively for the device chosen. It should be noted that the figures given in table 6.4 are after adjustment. More detailed information regarding the path measurement for the individual modules is given in the appendix.

It should also be borne in mind that the delay figures include the normalised routing factor discussed in section 5.2 which for the case of the device chosen is 0.984ns.

Total Cost # clbs	**LPD** (ns)
943	149.40

Table 6.5: TRC Total Cost and Longest Path Delay

Total cost is dependent on the number of units of each module required in a design. From the cost of each module given in table 6.4 and the number of instances of each module required, given in table 6.1 of section 6.1.2, the total cost of the TRC design was obtained as shown in table 6.5.

The longest path delay for a single router depends on the interconnection of the modules. As stated, the PDL design file describes the TRC interconnection illustrated in figure 6.16. Normalised wire delays are used to describe the delay in interconnecting the modules. The resulting longest path delay through the router is 149.40ns as shown.

6.1.5 Implementation of the TRC

The results shown assume that a TRC may be implemented on a single FPGA as no account of off-chip routing is given within the TRC design itself. An important consideration is, therefore, to show that the TRC design described can be implemented on the device chosen. As described in section 5.2, the total number of CLBs along with the total number of IOBs required gives an indication of the viability of implementing a given design within the chosen device.

The total number of CLBs in the overall design must be within those available which for the case of the XC4036 is 1296 CLBs. As shown in table 6.5, the TRC design is well within this boundary. Another factor discussed in

Modules	**Input**	**Output**	**#Paths**	**Total**
SCi	2	—	4	8
SMp	2	—	1	2
B	8	—	5	40
SCo	—	2	2	4
OP	—	8	2	16
CC	—	10	1	10
Total IOBs	—	—	—	80

Table 6.6: IOB Requirements for the TRC

section 5.2, is whether the design can be routed. Since the TRC design uses only 73% of the total CLBs available then it is assumed that the design may be routed.

The number of IOBs required depends on those modules lying at the input and output of the router on each of the paths. Port requirements for the respective modules are given in table 6.6. Note that this does not include internal communications with other modules. For the given FPGA device chosen with 288 IOBs, the TRC design with 80 IOBs is not IOB bound. Note that the number of paths (#paths) represents the number of instances of the given module required, as described in section 6.1.2.

6.2 Case 2: Exploring the Design Space

For the framework to provide a platform for design space exploration it was important that simple design changes may be realised with minimal rewriting within a model. This section illustrates the process of design changes and the performance and cost results obtainable. The effects of these changes are considered both in the light of the individual modules effected and the latency and cost results of the router itself.

Two cases are considered. The first considers the case where a design change effects a constant which parameterises a number of the design descriptions. The design issue chosen is the flitsize. The second involves a design change which requires a rewrite of part of the code for the effected modules. The design issue chosen is control.

6.2.1 Case 2a: Changing a Design Parameter

In the Torus Routing Chip, the flit size was chosen to be 8 bits. If pin-out is sufficient to permit an increase in flit size then the resulting increased data path may be expected to increase performance. However, what is the cost of this increase?

In this section, the effect of a flitsize of 16 and of 32 bits on a TRC structured design are considered. It is assumed that channel width is altered accordingly such that the flit size is assumed to equal the channel width. Without this assumption, extra hardware would be required to handle fragmentation and re-assembly of each flit.

The flit size is not a factor that changes the number of units of a module required in a design, see table 4.5 in section 4.2.2. As such, it does not effect the interconnection structure of the modules of the router. Instead, the flitsize changes the width of the data paths through the router and, as such, it effects the number of inputs and outputs of each of the modules on the data paths. As described in section 4.2.1.20, this change effects the complexity of the modules.

For the case of the Torus Routing Chip, the modules that lie on the data path include the buffer, address comparator, address decrementer, crossbar, virtual channel controller, output module and consumption channel.

As in section 6.1 the designs are instantiated in XILINX FPGA technology using the XC4000XL series, device 4036 and part PG411. Balanced optimisation and level 1 effort are again chosen.

Table 6.7 describes module costs for the three flit sizes chosen. Note that for those modules that lie on the data path, this increase in complexity has resulted in an increase in cost. The total cost for a 16 bit and 32 bit TRC router are given in table 6.8 along with the results from the original 8 bit version. From the 32 bit case, it can be seen that although the bus width

Symbol	**8 bit** (# CLBs)	**16 bit** (# CLBs)	**32 bit** (# CLBs)
AC	5	9	18
AD	4	8	16
AR	43	43	43
B	4	8	16
CB	71	123	226
CC	88	137	231
OP	5	4	5
RD	43	43	43
SCo	13	13	13
SCi	9	9	9
SM	5	5	5
VC			
— Fifo	88	137	231
— Mx	5	9	17
— Ctrl	1	1	1

Table 6.7: Flit Size Variation on TRC Module Costs

has increased by a factor of four, the total cost has only increased by a factor of around two. Although this result sounds promising for the performance improvement, it still indicates that the performance improvement does not come for free.

Again a check of CLBs and IOBs requirements can ascertain if the designs are implementable. Since the number of CLBs available is 1296 then in both cases the designs exceeds the device size chosen. The 16 bit case requires an XC4044 device with 1600 CLBs available and the 32 bit case requires an XC4062 device with 2304 CLBs available. It should be noted that another solution would be to partition the design. However, both solutions incur an increase in cost.

Symbol	**8 bit** (# CLBs)	**16 bit** (# CLBs)	**32 bit** (# CLBs)
Total Cost	943	1306	2022

Table 6.8: TRC Cost for Varying Flitsize

Modules	**# Ports** 16 bit	**# Ports** 32 bit	**# paths**	**Total** 16 bit	**Total** 32 bit
SCi	2	2	4	8	8
SMp	2	2	1	2	2
B	16	32	5	80	160
SCo	2	2	2	4	4
OP	16	32	2	32	64
CC	18	34	1	18	34
Total IOBs	—	—	—	144	272

Table 6.9: IOB Requirements for 16 and 32 bit flit sized TRCs

As shown in table 6.9, the total number of IOBs required for the 16 bit and 32 bit versions are 148 and 276 respectively. Since the XC4036 device has 288 IOBs then the designs are not IOB bound.

Delay analysis was also undertaken. Not surprisingly no increase in delay was seen for the cases considered. This is due to the fact that although complexity is increased, in this case the change in complexity does not effect the longest path through a given module. This example illustrates a positive performance improvement in that any resulting increase in throughput through the increased bus width does not increase latency. Thus the throughput verses latency, see section 3.1.3, trade off does not apply in this case.

From table 5.1 of section 5.4.4, it may be seen that this design change requires changes to the flitsize parameter of the HDL descriptions i.e. a change to the constant in the VHDL library. Each module effected by this variation must be re-synthesised and timing analysis undertaken. The results are then used to update the data file for the current model. However, data changes do not require re-compilation of the model. The TRC design at the RTL level is unchanged and, as such, the design specific model generated from the generic PDL model and the TRC router design does not have to be recompiled. Instead the design specific model is reevaluated with the updated data file.

6.2.2 Case 2b : Altering a Design Description

As stated, the TRC is an asynchronous router both internally and externally. In this section the results from a synchronous variant of the TRC are presented. Figures 6.17 and 6.18 illustrate paths through the TRC using asynchronous and synchronous control respectively.

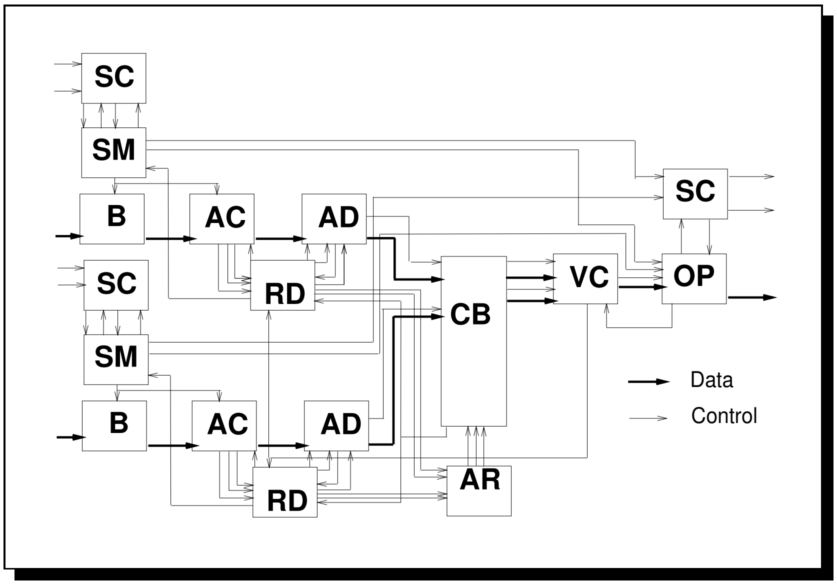

Figure 6.17: TRC Asynchronous Path

To describe the process of adaption to synchronous control, the process shown in figure 5.11 of section 5.5 is used as a guide.

The design issue control does not effect the constants parameterising the

VHDL code descriptions or the parameters for performance evaluation such that no parameters within the TRC model need be changed.

Selection of generic modules, in this case, involves consideration of those modules already selected for the asynchronous TRC and removal of those modules whose function is solely asynchronous control. Any additional modules needed to achieve synchronous control may be added to the design.

Figure 6.17 illustrates the path from a single dimension of the network through the asynchronous router. The path is split into two virtual channel paths at the input and a single path at the output. From the module set summary in table 4.2 of section 4.1 it may be seen that the modules SC and SM — signal converter and signal manager respectively, are asynchronous control modules. As such, these modules may be removed for the synchronous case. The RD module is also involved in asynchronous control, as described in section 6.1.1.2, but as this is not the sole function of the module then this module should not be removed.

The only module within the module set whose sole function is synchronous control is the synchroniser. This module handles synchronisation of data driven on different clocks. However, assuming that this version of the synchronous TRC will operate within a globally clocked network then the SY module is not required.

From tables 4.5 and 4.6 it may be observed that control does not effect the number of units of a module within a design and therefore for those modules remaining, no change is made to the number of instances selected for the design.

To build up the interconnection, the structure of figure 6.17 is adjusted for the removed modules and the handshaking signals which are no longer required. The resulting design is shown in figure 6.18. The PDL design file for the asynchronous case is then modified to reflect the changes in the register-transfer level design.

Assuming that no design instances exist for the required synchronous modules then design instances must be created. Referring to table 4.3 of section 4.2.1.20, control is seen to effect the complexity of all modules. Also, from table 5.1 of section 5.4.4, it may be seen that this design change requires changes to the HDL descriptions i.e. coding, rather than to the HDL

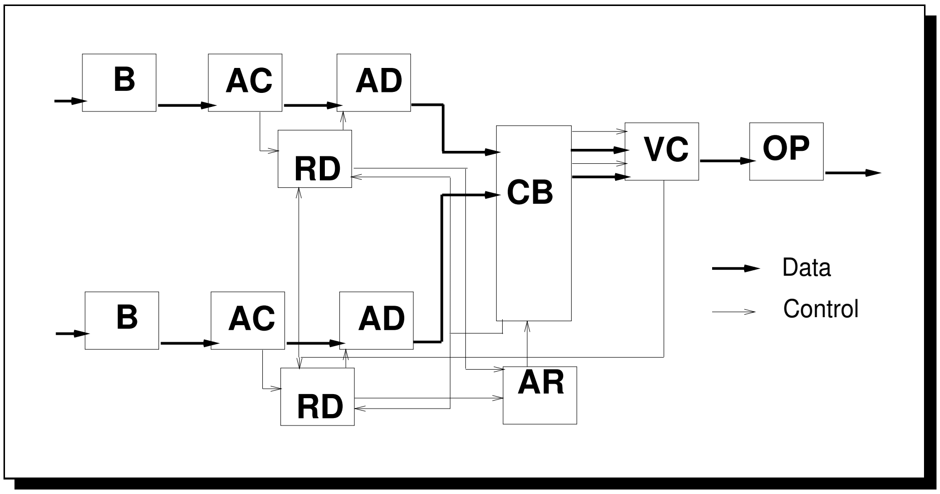

Figure 6.18: TRC Synchronous Path

parameters.

Conversion from an asynchronous to a synchronous design description requires relatively small changes to the code. In the asynchronous case, when reset is not high then the process may continue. In the synchronous case, a clock and clock event condition are added to the non reset case. As such, the functionality of the module is unchanged, just its control. The main coding changes required are in those modules involved in handshaking.

The model is completed by synthesis and timing analysis of the revised designs and updating the model data file. This time re-compilation of the PDL design is required to compile changes to the register-transfer level design description. The model is then evaluated using the data from the revised designs.

The cost and delay results for the modules of the synchronous TRC are given in table 6.10 along with a repeat of the asynchronous 8-bit version of the TRC for comparison purposes.

It should be noted that the RD module is implemented as a clocked finite state machine. That is that a number of clock cycles are required for the complete state machine. However, after each state a decision is made thus allowing the RD module to be modelled using a single clock cycle. As can

Symbol	Asynchronous Cost (# CLBs)	Asynchronous Delay (ns)	Synchronous Cost (# CLBs)	Synchronous Clk Period (ns)
AC	5	7.086	5	3.914
AD	4	6.536	5	3.624
AR	43	12.954	45	15.238
B	4	5.448	4	3.624
CB	71	10.12	45	11.605
CC	88	19.55	60	12.608
OP	5	4.847	4	3.624
RD	43	33.932	23	16.174
SCi	9	7.756	—	—
SCo	13	9.415	—	—
SM	5	13.695	—	—
VC				
— Fifo	88	19.550	60	12.608
— Mx	5	4.898	4	3.073
— Ctrl	1	4.919	1	3.073

Table 6.10: Performance of Asynchronous and Synchronous TRC Modules

be seen from the results of table 6.10, the RD module is the slowest module. As described in section 5.4.2, this clock period is therefore used as the clock speed for the design i.e 62 MHz.

The results for both the original and synchronous versions are shown in table 6.11. It is interesting to note the large reduction in cost in the synchronous case. This is caused in part by the removal of the asynchronous control modules i.e. eleven module instances, but also the decrease in complexity of individual modules.

The removal of the asynchronous modules has not, however, led to a reduction in router delay. This is due to the fact that all modules are clocked at the speed of the slowest module. It should be noted, however that this is a conservative estimate for a clocked system.

Asyn. Cost (# clbs)	**Asyn. Delay** (ns)	**Syn. Cost** (# clbs)	**Syn. Delay** (ns)
943	149.40	593	157.374

Table 6.11: TRC Latency and Cost for a 16 Node Torus

6.3 Case 3: The Reliable Router

As stated, the framework is designed to support modelling of both existing and proposed router designs. However, a problem still exists and that is the lack of information that is available for existing routers. During the survey of existing routers, the choice of routers to analyse in more depth was limited by the information available, not only at the gate level but also at the register-transfer level. This identified an extra function that the framework needed to fulfil. That was to provide an infrastructure which would aid the modeller identify and piece together the missing pieces of the 'jigsaw puzzle'.

In this section, a router instance based on the Reliable Router, is described. The Reliable Router offers a synchronous example with many newer features not seen in earlier router designs. In comparison with other documented routers, there is a reasonable amount of detailed information available. However, it still provides the modeller with a difficult task to find the missing pieces and work out the functionality of the existing pieces.

The description given follows the process illustrated in figure 5.11 of section 5.5. It also relates the model development to that of the TRC model described in section 6.1.

6.3.1 Building up an Interconnection of Modules

The first stage is to identify the design parameters. These are given in table 2.3 of section 2.5.

The second stage, identifying the generic modules is somewhat more complicated. Figure 6.19 gives an overview of the router. Again as in the TRC, a centralised crossbar is used but in this case a 6x6 crossbar (**CB**) is required.

Again an arbiter module (**AR**) is required to control access and switching of the crossbar.

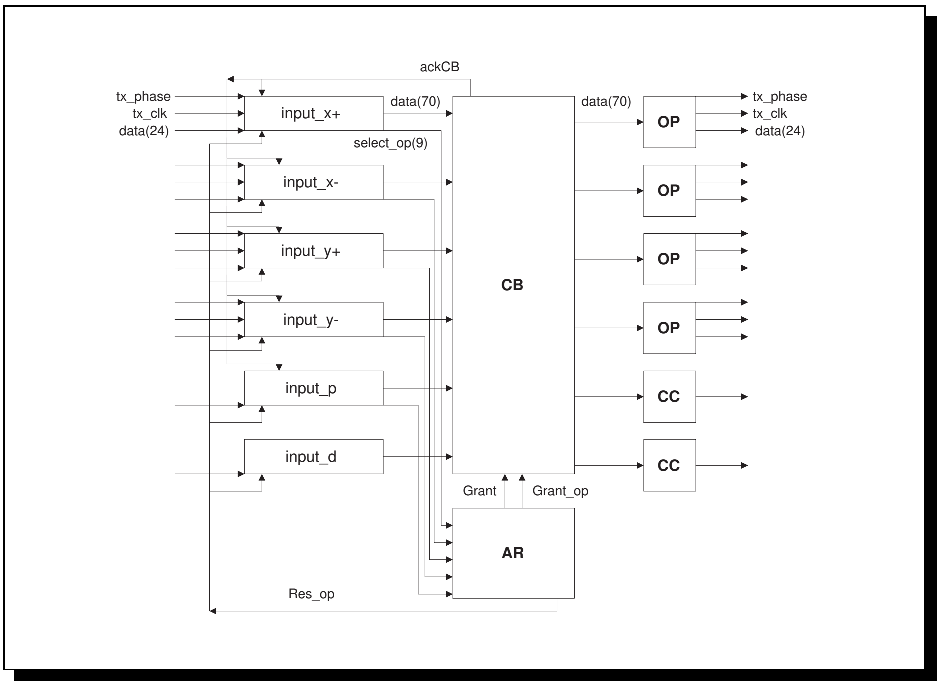

Figure 6.19: Reliable Router Overview

The next stage is to identify the modules required on a single path through the router. Most of the complexity of the Reliable Router is placed at the input and as such the output stage i.e. between the crossbar and the output pads, consist solely of an output module(**OP**) to handle fragmentation of a flit into 4 frames for sending across the external channel, adding control information and generating the clocking signals required.

Regarding the TRC design and the identified features of the reliable router, the lack of a VC module (**VC)** at the output may at first be surprising. However, as large data paths and a large number of virtual channels have been implemented then the choice to place the VC module at the entrance to the crossbar may be seen to be an important decision with respect to reducing the size of the crossbar.

To decide which modules are required at the input side of the router reference

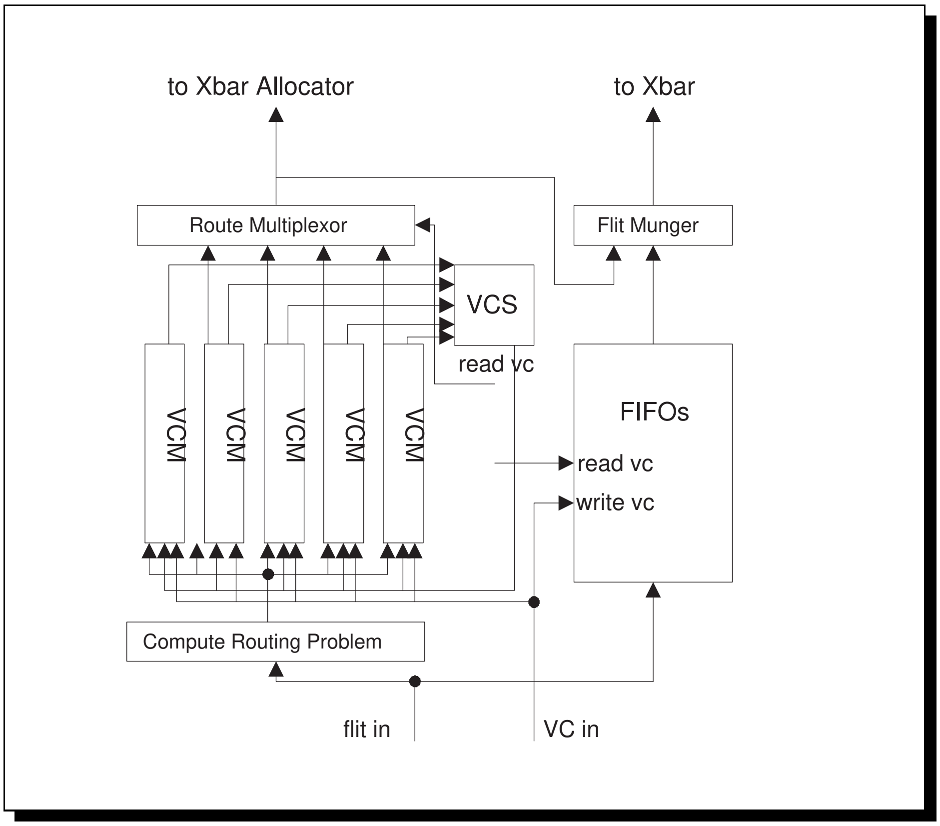

Figure 6.20: The Reliable Router Input Controller

is made to more detailed diagrams available in the literature. Figure 6.20 illustrates what is termed as the input controller i.e. the functionality at the input to the crossbar. By reference to the module set it may be seen that the functionality depicted describes a number of modules instead of a single module. As such, the required modules must be identified.

The simplest method is to first pick out the virtual channel controller which, as described above, is within the input controller. It is important at this point to look away from the naming given and consider the functionality of the virtual channel controller. From the definition given in section 4.2.1.19, queueing for the virtual channels, a multiplexer and a multiplexer controller are required. These are termed FIFO, Route Multiplexer and Virtual Channel Controller respectively. It should be noted that the data from the FIFOs does not flow through the multiplexer module, as is the case in the TRC. Instead the multiplexer multiplexes the routing decision from the chosen virtual channel. The multiplexer controller which chooses the virtual channel also sends a read signal to the FIFOs to drive the required data onto a data bus shared by all FIFOs.

At this point we are left with the unknowns "Compute Routing Problem", "Virtual Channel Module" and "Flit Munger". It should be noted that the title "Virtual channel Module" refers to the description given in the Reliable Router literature and bears no reference to the virtual channel controller **VC** described in the module set. From the description of the "Compute Routing Problem" and "Flit Munger" and the module set descriptions these may be seen to be versions of the address comparator **AC** and flit update modules **FU**. This has now reduced the problem to finding module equivalents for the "virtual channel module" which is illustrated in figure 6.21.

From figure 6.21, a flow control unit exists and it's functionality given in the literature matches that of the flow control module **FC** of the module set. Also, that the blocks Routing Problem and Route are buffers **B** for storage of control information. The other three modules, Optimistic Router, Route FSM and Control are all involved in the routing decision and operation of the routing module and as such are included in the routing decision module **RD**.

Note the lack of Address Decrementer indicating the use of absolute instead of relative addressing in agreement with the design issues of section 2.5.2.5.

Two inputs flit_in and VCin — data and virtual channel identifier respectively, are shown in figure 6.20. This suggests that a message split module is required to extract the control and data information from the incoming flit. The message split module **MS** therefore precedes the address comparator on the data path.

This is not the complete set of modules required. The Reliable Router is required to reassemble the flit from the frames sent on a channel to retain the flit flow control unit requirement of wormhole routing. A message build module **MB** is therefore required. In addition, as the reliable router does not use a global clock but a then the incoming data must be synchronised to the local clock i.e a synchroniser module is required **SY**. This is placed after the message build module on the path as the message build module is driven

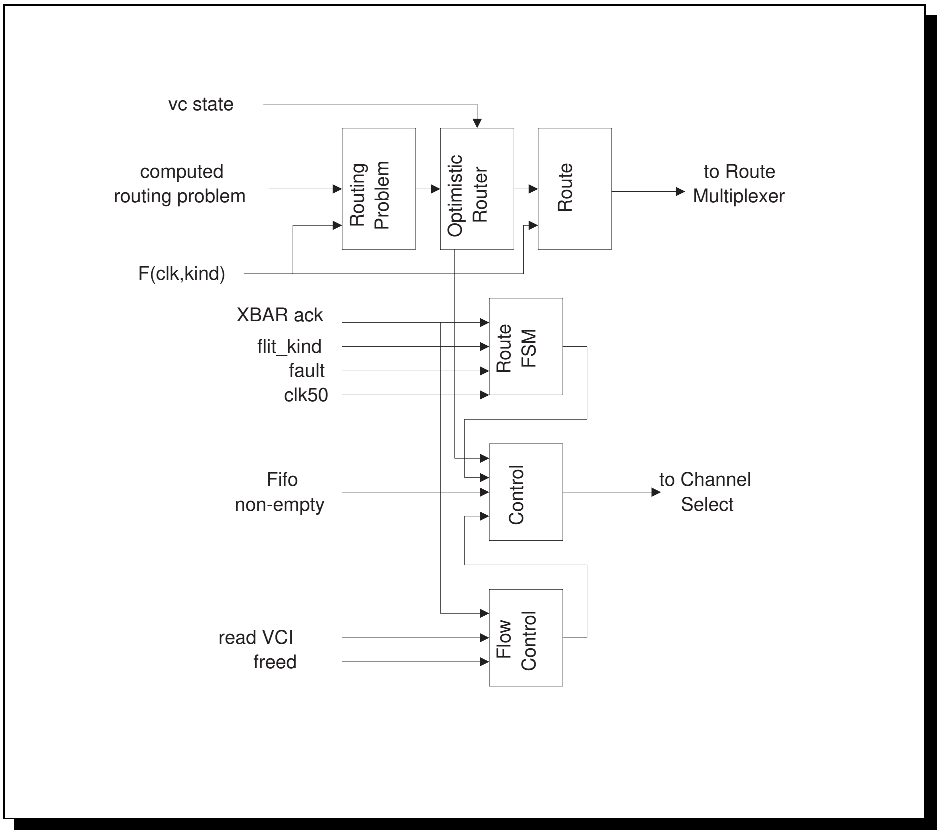

Figure 6.21: Reliable Router "Virtual Channel Module"

by the incoming clock of the sender. Finally a sampler is required to sample the synchronised data on the local clock i.e. a simple latch **B**.

To summarise, the input includes a front end which handles the receipt and synchronisation of incoming data. The incoming path is then split into control and data and the data forwarded to the queues of the virtual channel controller. On the control path, the address comparator provides routing information to the appropriate routing decision module. The appropriate routing decision module then, with the help of the flow control unit makes a routing decision which is forwarded to virtual channel controller. This in turn selects a virtual channel and forwards the routing decision to the crossbar arbiter and the data to the crossbar after updating the data with the chosen output virtual channel identifier. It should be noted that both the address comparator decision and the routing decision are stored in the appropriate buffers for later use, if required.

At this stage the generic modules required for the RR have been identified and a feeling for their intra-behaviour established. These modules, in the order of appearance on a path, are: MB, SY, B, MS, AC, B, RD, B, VC, FU, AR, CB and OP (CC).

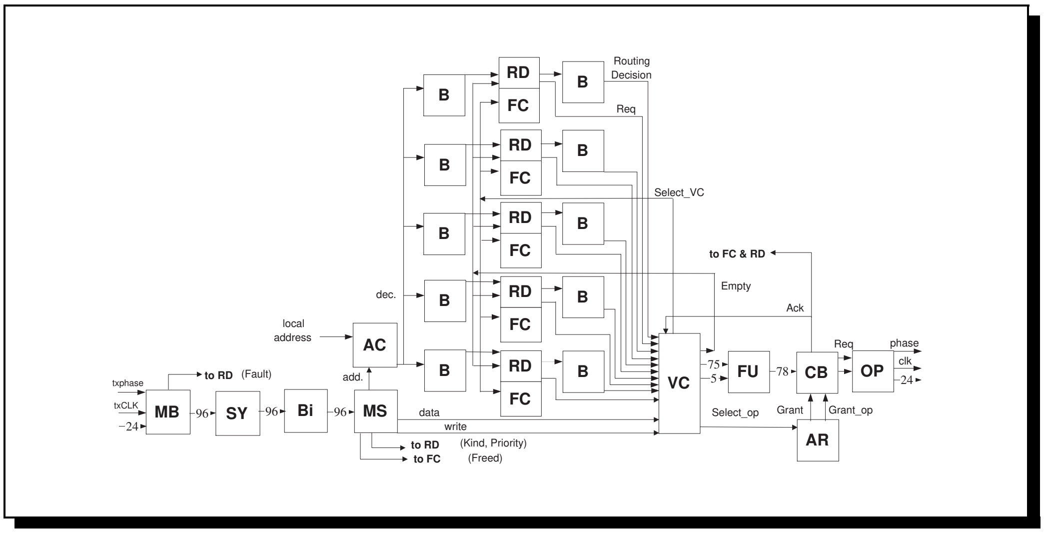

Figure 6.22: Reliable Router Data Path

The next stage is to select the number of instances of each of the modules required. Consideration should be given to a single path through the router. From the discussion above and analysis of the design information available, a path may be described as in figure 6.22. This illustrates the path splitting inherent in the RR design and the choice of module placement. Similar to the TRC of section 6.1, the total number of modules required may be obtained from the design issues of the TRC as shown in table 6.12.

It is assumed that both the diagnostic and the processor path are similar to the other paths except for the front end. If it is assumed that communication between the router and its local processor operate at the same clock speed then synchronisation and sampling is not required. It is further assumed that a wide enough channel exists between the processor and it's router such that

	Components												
Issues	AC	AR	B	CC	CB	FC	FU	MB	MS	RD	SY	OP	VC
Degree	4		12			4	4	4	4	4	4	4	4
Number of VCs			5			5				5			
Partitioning		1			1								
No. of Inj/Cons	2		4	2		2	2		2				2
Total	6	1	66	2	1	30	6	4	6	30	4	5	6

Table 6.12: Number of Units Required of each of the RR Modules

a complete flit may be transmitted over the channel. This removes the need for the MB module and simplifies the consumption channel. The processor path, therefore, begins with the MS module.

To put the modules together as a PDL design, each module instance is given a unique name and the modules interconnected following the pattern illustrated in figure 6.22 and the router structure given in figure 6.19.

The next stages involve selection of the appropriate design instances and the relevant data for inclusion in the PDL model. This process is not described in detail in this case study. Instead it is chosen to give a brief summary of the modules themselves to aid understanding of the performance data obtained.

6.3.2 The RR Modules

This remainder of this section describes the design of the individual modules implemented and how these differ from the case of the TRC. In addition, it highlights the places where newer features are added to an existing generic module.

Although the lack of detailed information made the detailed design somewhat difficult, the building block nature of the framework aided simplification of this task into a set of sub tasks i.e. design of each module. Development of each module consisted of filling in the design information available and adding features that were shown to be a part of the module through the requirements of other modules. That is, several attempts at the design of most of the modules were required before reaching a satisfactory design.

A features of the router not currently modelled is the Unique Token Protocol(UTP). Although the UTP is a newer function handling fault tolerance in networks, implementation does not require the creation of any new modules but changes to existing modules.

The synchronisation module has not been modelled due to the complexity of this newer feature. It is chosen instead to override the gate level description for this module and instead provide a worst case estimate of one clock cycle as rapported in the literature. A conservative assessment for the cost of the synchroniser is given by using a buffer cost in its place.

6.3.2.1 Message Build

In the RR a series of level-sensitive latches are used to assemble the 4 frame flit, clocked on the transmitters clock. That is that the sender sends both it's own clock along with a signal to help the receiver identify the individual frames of the incoming data. A parity checker is also included in the module. From this, a decision is made whether to accept or reject the incoming flit.

6.3.2.2 Synchroniser

Each router operates on it's own local clock, thus avoiding the problems involved in clock distribution over large parallel systems. However, this means that synchronisation between these two clocks is required. The reliable router implements an effective synchronising method to synchronise the data received on the transmitters clock to that of it's local clock.

6.3.2.3 Buffer

There are two buffer types included in the input side of the crossbar. The first is a sampler. The sampler's purpose is to sample the entire flit in the receiver's clock before forwarding it to the rest of the router. A 96 bit buffer is required for this purpose. 4 bit buffers, latches, are also required to store the address comparator decision and the routing decision respectively.

6.3.2.4 Message Split

The purpose of this module is to separate the data bits from control bits within the incoming data and forward these to the respective modules.

6.3.2.5 Address Comparator

The address comparator compares both the absolute x and y address against the local address in a single clock cycle. The result expresses either a match in both x and y indicating that the destination is reached or an indication as to where the destination lies in the network with respect to the current node i.e. in the positive, negative or at the destination for the given dimension. The decision format is a two bit decision for each dimension.

6.3.2.6 Flow Controller

The flow controller is a counter which holds control over free space in the neighbouring node. It is incremented when a flit is sent to a neighbouring node and decremented when the neighbouring node sends a flit to it's next node. This latter control signal is received from the MS module after extraction from the incoming packet. Based on the status of the counter, and confirmation of acknowledgement from the virtual channels controller that the virtual channel has been chosen, it sends an 'ok-to-send' to the routing decision module.

6.3.2.7 Routing Decision

The main part of the routing decision module is made up of not one but three different routing algorithms. The first two are involved in a normal routing decision i.e. adaptive and deterministic routing algorithms. The third, fault tolerance provides for routing to go around faulty channels. Processes for all three algorithms are driven in parallel such that all possible routing decisions are available within a single clock cycle.

The adaptive algorithm is minimal adaptive in that the network may forward a flit on any productive channel, that is a channel in the direction of the destination. This decision is a random choice based on information from the arbiter as to which output virtual channels are available and the channel priority of the packet i.e. VC1 or VC0. The deterministic algorithm is a dimension-order algorithm for mesh networks similar to that described in the TRC. If a free channel cannot be found using the adaptive routing algorithm then the deterministic choice is selected. The choice is forwarded to the VC module and competes with requests from other virtual channels wishing to use the crossbar. If a request is not accepted within a clock cycle then at the next clock cycle the decision is recomputed and a new request submitted. If a request is not handled within seven attempts then the request is given top arbitration priority.

If the node receives a faulty channel report and this channel is required for the current packet under the algorithms described above, then the packet is, instead, routed using the the turn model routing algorithm. A single virtual channel path is available for routing using the turn model.

An example implementation of the routing decision module is found in appendix C.

6.3.2.8 Virtual Channel Controller

The virtual channel controller selects between five virtual channels based on a round robin arbitration method. The virtual channel controller selects the appropriate virtual channel and splits the eleven bit wide selection into a six bit physical channel selection to be forwarded to the crossbar arbiter and a five bit virtual channel selection. This virtual channel selection is passed to the flit update module to be added to the flit being forwarded across the crossbar. In addition, the arbitration priority of the packet is added to the arbitration request.

The VC selection is used to send a read signal to the appropriate FIFO such that it's data is driven onto the shared data bus. This differs from the FIFO implementation of the TRC where data is driven on the output bus as soon as there is data available in the FIFO. In the RR it is, therefore, important that the FIFO implementation uses a tri-state bus at the output so that only one FIFO is driving the bus at any one time.

It should be noted that the multiplexer controller uses a round robin arbitration method ensuring fairness but increasing the complexity of the arbiter over that which would results from a simpler arbitration method.

6.3.2.9 Flit Update

The FU module receives data from the VC module along with the virtual channel identifier (VCI) of the selected output virtual channel. The VCI is received as a 5 bit one-hot coded selection. The VCI coding is converted to binary and added to the outgoing data.

6.3.2.10 Crossbar

The crossbar is implemented as in the TRC except that control is synchronous, each channel is 70 bits wide and there are six input and output channels. The width of the data bus is assumed from the fact that the flit size is 75, including data, kind and priority. Since the priority check is completed at the MS module, then it is assumed that these 8 bits are stripped from the flit before further transmission within the router. In addition, the new three bit virtual channel identifier is added in the FU module.

6.3.2.11 Output Controller

The purpose of this module is to split the flit up into 4 frames and add any extra control information to the frames as required. However, in addition to driving data it also sends the clock and clock-phase signal to the neighbouring node for the process described in section 6.3.2.1. The output controller receives and sends a flit every 2 clock cycles.

6.3.3 Modelled Performance

Again it should be borne in mind when considering the figures obtained for the Reliable Router that these are based on the information available and the chosen implementation technology for the study — FPGA technology. This assessment is off course limited to the features of the Reliable Router implemented in the model.The VHDL library used in the implementation of the Reliable Router Modules may be found in appendix D.

In table 6.13, the cost and delay results for the individual modules of the Reliable Router are presented. The results for the original implementation of the TRC are also presented for comparison. Absolute addressing removes the requirement for an address decrementer and synchronous control removes the requirement for the signal converters and signal manager. However, a number of new modules are added to the design. The large buffer module for sampling and the synchroniser module to handle the new synchronous features are required. Increased complexity in the output module and the addition of the message build module are required to handle the 4-frame flit sending on the channel. The message split module handles path splitting such that the large data path does not flow through the control modules. Other new modules include flit update and flow control.

As described, the Reliable Router offers a number of newer router features in addition to having very wide data paths. These changes, with respect to those seen in the Torus Routing Chip, give rise to a large increase in cost. The cost of the TRC was given in section 6.1 as 943 CLBs of which 5×88 i.e. 440 CLBs, represents the cost of the FIFOs, leaving 503 CLBs for the remaining logic. In the Reliable Router, on the other hand, the non-FIFO logic adds up to 4541 CLBs and each FIFO bank (5 FIFOs) costs 8910 CLBs. of which there are six. That is, a substantial increase in cost is seen in the Reliable Router. Note that the number of CLBs is far in excess of any FPGA device such that partitioning over a number of devices would be required. However, the aim of this analysis is to highlight the difference between the TRC and the Reliable Router and, as such, the results are given in CLBs.

It should be noted that the synchroniser module, although not implemented in this work, is given in the literature as having a worst case delay of one clock cycle. As such, for simplicity, a cost equivalent to the sampler is assumed and delay is fixed at one clock cycle. Referring to the literature, it may be seen that the entire path though the Reliable Router takes a minimum of seven and a maximum of eight clock cycles. That is, eight clock cycles includes the worst case synchroniser delay. As described in section 5.4.2, the maximum clock speed in the framework is derived from the slowest module. In this case, the slowest module is the routing decision module with a minimum clock period of 22.494ns (44.5Mhz). This gives a worst case longest path

Symbol	**Cost** (# clbs)	**Delay** ns	**Cost** (# clbs)	**Delay** ns
AC	5	7.086	16	9.925
AD	4	6.536	—	—
AR	43	12.954	15	10.88
B	4	5.448	2	3.624
	—	—	48	3.624
CB	71	10.12	1306	12.98
CC	88	19.55	38	3.624
FC	—	—	10	15.733
FU	—	—	37	7.038
MB	—	—	91	4.023
MS	—	—	42	3.624
OP	5	4.847	64	13.257
RD	43	33.932	34	22.494
SCi	9	7.756	—	—
SCo	13	9.415	—	—
SM	5	13.695	—	—
SY	—	—	38	1 CC
VC — Fifo	88	19.55	1782	17.866
— Mx	5	4.898	25	10.457
— Ctrl	1	4.919	15	8.04

Table 6.13: The Torus Routing Chip and Reliable Router Design Data

delay for the Reliable Router of 179.95ns. This may be compared against 149.4ns in the TRC model.

From these results it may be seen that although delay increases due to the increased complexity that this is not so significant compared with the very large increase in cost. This is not so surprising when the large increase in cost is due to very wide data paths.

6.4 Case 4: Network Level Analysis

In this section an analysis of network latency with respect to network size and packet size is presented. The study uses the TRC design of section 6.1 as the base design. It should be noted that reference to specific delays refer to the definitions giving in chapter 3.

The experiments are divided into two parts. The analysis of section 6.4.1 assumes that t_{head} is equivalent to t_{prop}. That is that all flits experience the same delay traversing a router. Section 6.4.2 expands this analysis by separating and calibrating the delays experienced by the body flits from those of the header flits. The delay adjustment is derived from an analysis of the given router design.

6.4.1 Case 4a: Network Performance Assuming that $\mathbf{t}_{node}$ is Equivalent to $\mathbf{t}_{prop}$

The first experiment considers the effect of varying the network size on the resulting network latency whilst holding the packet size constant. To compare the effect of smaller and larger packets i.e. 8 and 128 flit packets respectively , two sets of results are shown in figure 6.23.

In wormhole routing, the aim is to pipeline a packet's transmission across the network so as to reduce the latency dependence on diameter. It is, therefore, not surprising that the results indicate a sub-linear relationship between network size and latency. The results also show that increasing the packet size increases latency but that linearity is unchanged.

The results shown in figure 6.24, considers the packet size issue in more detail. Results for two network sizes are presented — a 256 and a 4096 node network, this time varying the packet size. A linear relationship between the packet size and network latency may be seen from the graph. Increasing the network size increases latency but retains the same linearity in the results.

Unlike most performance analysis seen in the literature, the network level results are calibrated by the router design instance chosen. Calibration of network latency does not change the expected trend of a linear relationship between latency and increasing packet size but increases the precision of the

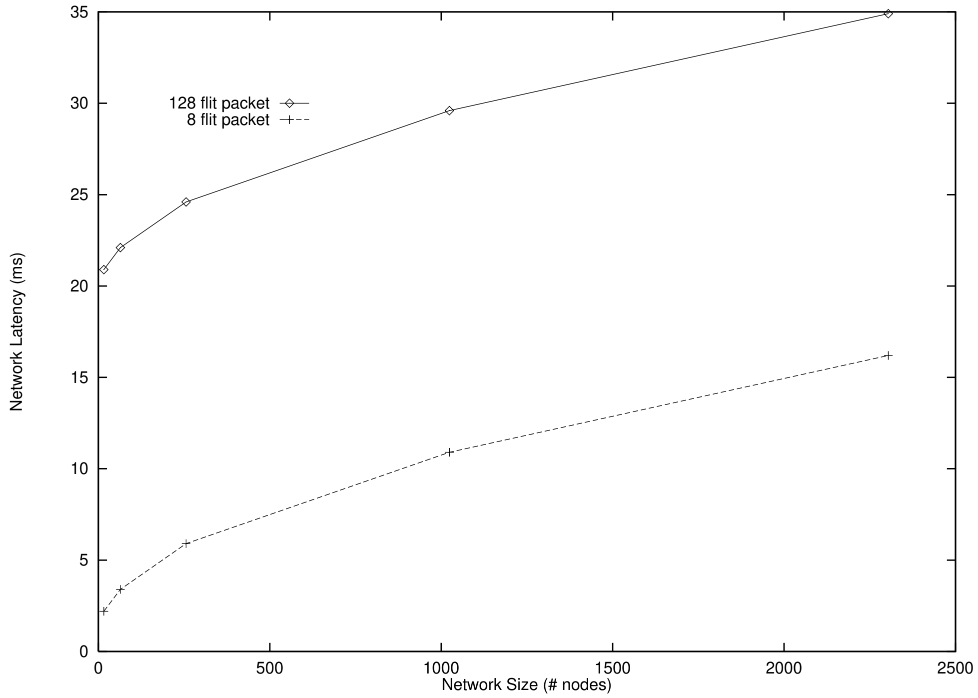

Figure 6.23: Wormhole Routed Network Latency Using a Fixed Packet Size and t_{head} Equivalent to t_{prop}

results shown.

6.4.2 Case 4b: Network Performance Assuming that $\mathbf{t}_{node}$ is not Equivalent to $\mathbf{t}_{prop}$

In the following experiments t_{prop} is not assumed to be equivalent to t_{head}. The value of t_{prop} is obtained through analysis of the underlying design.

In the TRC, the flits of the body follow the same path as the header through the router chip itself. That is that they encounter delays through each of the modules that the header traverses. However, for the body flits, no routing decision is required. The routing decision, based on the header, is stored in the RD module. However, the RD module is also involved in control and,

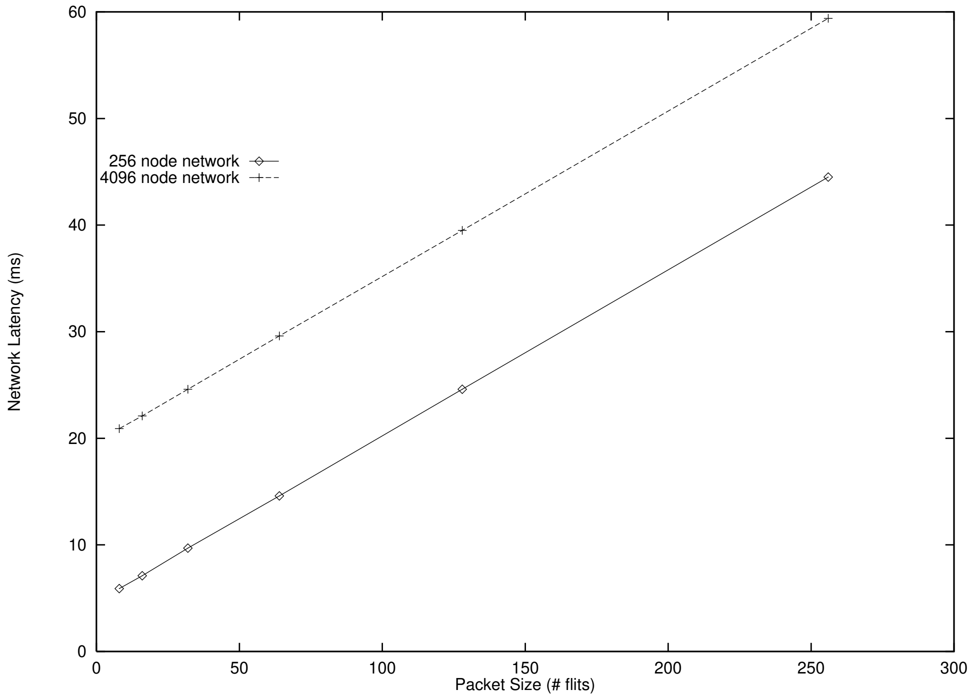

Figure 6.24: Wormhole Routed Network Latency Using a Fixed Network Size and t_{head} Equivalent to t_{prop}

as such, the delay encountered by the body flit at the RD module is that of control alone. For the case of the TRC, the delay that is seen by the body flits, is 13.672ns whereas that seen by the header flits is 33.932 ns.

Using this revised estimate for t_{prop}, the experiments of section 6.4.1 are refined. Figure 6.25 compares the effect of the equivalence assumption against the calibrated propagation delay for a 128 flit packet. As shown, linearity is not effected and both results show similar sub-linear curves. However, a decrease in network latency is seen in the calibrated case due to the more precise performance data used.

In the second case, depicted in figure 6.26, it may be seen that as packet size increases, the effect of the calibrated propagation delay on the resulting latency increases. That is the decrease in latency, due to the improved

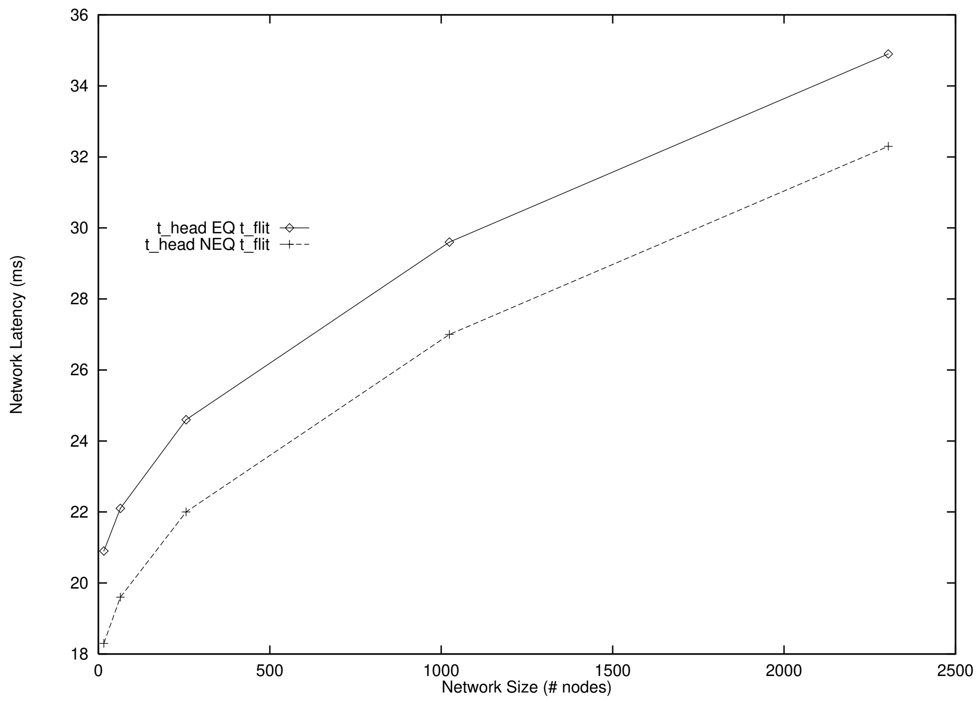

Figure 6.25: Wormhole Routed Network Latency Using a Fixed Packet Size of 128 flits

performance data, is more noticeable as packet size increases. This may be explained by the fact that as packet size increases, the proportion of flits encountering the lower delay increases thus reducing latency.

It should be noted that in many newer routers path partitioning exists, as described in section 4.2.2. That is, modules involved in decision making are separated from non decision making modules such that body flits do not flow through the control path. As such, the difference between the non calibrated t_{prop} and the calibrated t_{prop} increases. Likewise, the difference between latency based on the calibrated and non-calibrated t_{prop} is more significant identifying the importance of more precise performance data for analysing todays routers.

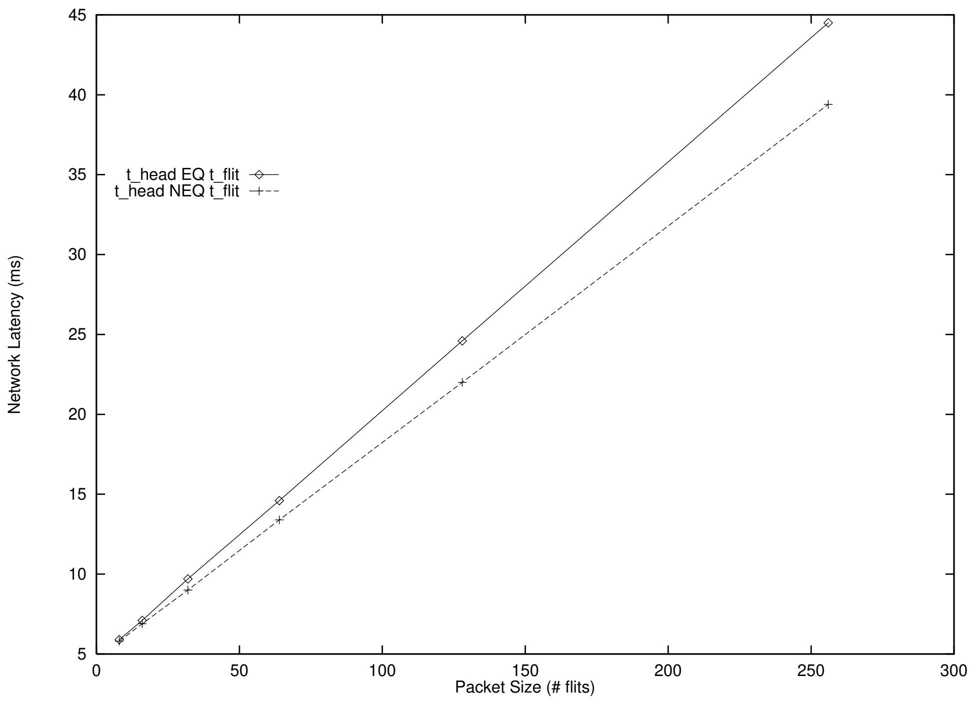

Figure 6.26: Wormhole Routed Network Latency Using a Fixed Network Size of 256 nodes

6.5 Case 5: Comparison of Switching Techniques

In this section an analysis based on the TRC design is made with respect to the different switching techniques. This analysis requires refinement of the gate level design for the cases of virtual-cut-through and store and forward routing.

As described in section 2.4.1, switching techniques differ in their buffering and flow control requirements. In more typical performance analyses of interconnection networks, it is not unusual for buffering requirements to be included in throughput analysis. That is the inclusion of contention resulting from network traffic with respect to the buffer size chosen. However, what is the effect of buffering on latency and cost? Also what effect does the unit of flow control have on these factors? To provide answers to these questions, a more detailed analysis of switching technique choice with respect to router design is required.

Both store and forward and virtual cut through routing require buffering for a complete packet whereas in the case of wormhole routing, a minimum buffer size equivalent to a single flit buffer is required.

Assuming a 16 flit packet, then the store and forward and virtual cut through router will require 16 flit deep FIFOs. It is assumed in the wormhole routed case that a 4 flit deep FIFOs is implemented, as in the original implementation.

In wormhole routed networks, the routing decision made must be retained at a node to be used by the remaining flits of a packet after the header has left the node. That is that the RD module must retain the routing decision. However, in wormhole routing, flow control within the router is conducted at the flit level placing a requirement on the arbiter to arbitrate between requests for output channels on a flit basis. To ensure that no two inputs are granted the same output, which would interleave the flits of a packet with those from other packets, the arbiter must reserve granted output channels until after the tail flit is received. This increased functionality increases the complexity of the arbiter module in wormhole routing. A similar situation is seen in the case of virtual cut through routing.

In store and forward routing, on the other hand, arbitration is on a packet basis. This means that once an arbitration decision is made, the arbiter awaits the tail flit before considering other requests. As such, the extra logic required to store reserved channels is not required. A less complex arbiter module is, therefore, required in this case. However, it is assumed that arbitration at the outputs remains on a flit basis as this is not effected by a change in switching technique.

The modules of the TRC design effected by a change in switching technique are the FIFOs of the VC, the CC and the AR module. The cost and delay of these modules, after adjustments in their design for the appropriate switching technique, are given in tables 6.14 and 6.15 respectively.

Table 6.16 represents the respective router design costs with respect to these

Symbol	**WH** (# clbs)	**VCT** (# clbs)	**SF** (# clbs)
VC-fifo	88	539	539
CC	88	539	539
AR	43	43	19

Table 6.14: Switching Technique Effect on Module Cost

Symbol	**WH** (ns)	**VCT** (ns)	**SF** (ns)
VC-fifo	19.550	20.375	20.375
CC	19.550	20.375	20.375
AR	12.954	12.954	6.235

Table 6.15: Switching Technique Variation effects on Module Delay

changes. Following the discussion above, it is not surprising that virtual-cut-through routing has the highest cost since it requires both reservation of output channels and increased buffering.

To analyse the effect of these changes with respect to router delay and network latency then the PDL data file is updated for the revised design data and the appropriate performance expression for latency is selected.

In this case only the experiment with respect to network size is conducted as the design under testing can hold a 16 flit packet. Due to the buffering restrictions of store and forward and virtual cut through routing, larger packet sizes will require the appropriate buffer design implementation to obtain their respective performance data. As these designs are not currently in the design library, this test is not possible.

Symbol	**WH** (# clbs)	**VCT** (# clbs)	**SF** (# clbs)
Total Cost	943	3198	3174

Table 6.16: Switching Technique Effect on Total Cost

A packet size of 16 flits is chosen to match the buffer design implemented and propagation delay is calibrated to reflect the particular design.

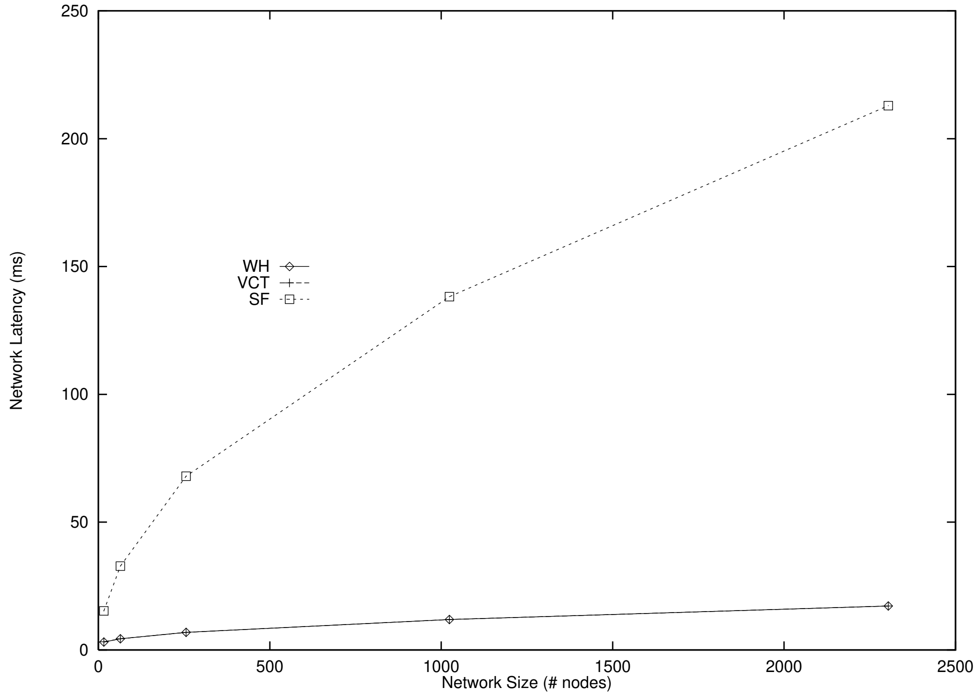

Figure 6.27: Latency Comparison Using a Fixed Packet Size of 16 flits and t_{head} not Equivalent to t_{prop}

As shown in figure 6.27, the switching techniques results are sub-linear. However, as the network size increases, the increasing latency due to the non-pipelining effect of store and forward routing may be observed. The graph indicates identical latency results for the cases of wormhole and virtual cut through routing. As shown in table 6.15, the router delay of the respective router designs for these switching techniques are not identical. However, the cumulative effect of this difference on network latency is comparatively small with respect to store and forward routing. However, it should be borne in mind that the results shown in figure 6.27 are based on unloaded latency, that is in the absence of network traffic. The addition of traffic would add contention to the analysis. In the worst case contention will cause network latency in virtual cut through routing case to be equivalent to that of store and forward routing. That is that the results shown represent a best case analysis for virtual cut through routing.

Chapter 7

Ongoing Work

This chapter considers ongoing work into performance analysis of communication hardware in multicomputers. Section 7.1 considers loaded network latency analysis as opposed to the unloaded analysis conducted in the case studies of chapter 6. In section 7.2 consideration is given to control signalling and the effect of this feature on theoretical expressions for latency. Section 7.3 presents a summary of work undertaken jointly with Professor Lasse Natvig investigating architecture modelling at various levels and the performance results achievable.

7.1 Latency Analysis in a Loaded Network

Statement 8 *The delay encountered traversing a router may be divided into a fixed and a variable part.*

The fixed delay is dependent on the design of the router and the node in question i.e. source, intermediate or destination (see section 3.3.2), and is independent of network loading. Variable delays, on the other hand, are made up of queueing and arbitration delays which are dependent on both network loading as well as some of the design issues. The size of a variable delay depends on the amount of queueing or arbitration required which, in turn, depends on the amount of traffic competing within the node at a particular point in time.

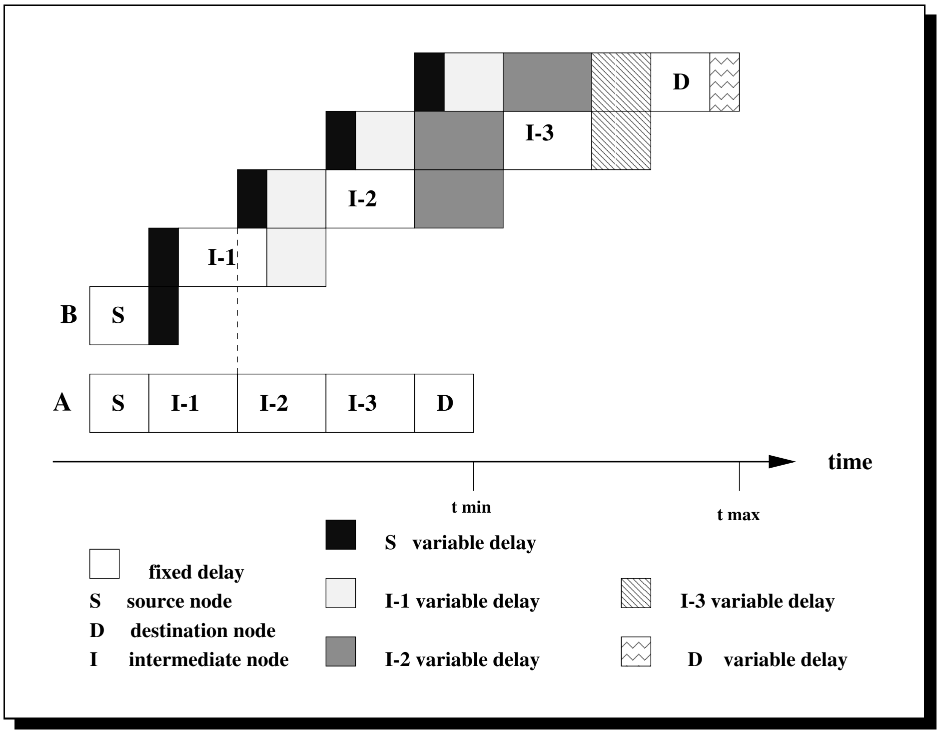

Figure 7.1: Wormhole Routing - Movement of the Head Flit

Figure 7.1 illustrates the delays encountered while traversing the routers on route to the destination. **A** represents an unloaded network and **B** a loaded network. It should be noted, however, that this figure represents a simplified picture of delay due to the absence of channels on the path.

In the case of the packet header, a routing decision and arbitration for the output channel is required at each node. As such, a variable delay is incurred at each node on route to the destination. As described in section 3.3.2, the fixed delay of the source and destination nodes may differ from that of the intermediate nodes. The variable delays are compounded as the header flows through the network. Taking node **I-2** as an example, the dashed line indicates the start of the fixed delay in A. As shown, the fixed delay of **I-2** is delayed by the variable delays of the source **S** and **I-1**. In addition, **I-2** adds it's own variable delay to the compounded delay of the path. The effect of the variable delays is shown to increase the path delay from **t_min** to **t_max** where **t_max** represents the network latency in a loaded network and **t_min** that of an unloaded network.

Statement 9 *For pipelined transmission, assuming that a one-flit buffer is available at each output channel, the following condition must be satisfied for the flow of the flits of the packet.*

$$(t_{Fix} + t_{Flex})_{src} \geq (t_{Fix} + t_{Flex})_{int1} \geq (t_{Fix} + t_{Flex})_{int2} \ldots \geq (t_{Fix} + t_{Flex})_{dest} \tag{7.1}$$

where t_{Fix} is the fixed delay and t_{Flex} the variable delay encountered at a node and $_{intX}$ represents an intermediate node in the network.

Expression 7.1 restricts movement of the flits of a packet within the network to a pipelined fashion, that is that a flit cannot overtake a preceding flit of the same packet. This highlights the situation that although the body flits are not involved in routing decisions, they may not progress faster than the header and, as such, may require to wait at a node until the header completes at the next node. As such, the body encounters both a fixed and variable delay where the variable delay reflects the waiting time for the header in addition to arbitration for the output channel.

The analysis presented in the case studies of chapter 6, represents unloaded latency analysis. Arbitration delay is included in this analysis as a fixed delay, representing the delay due to the implementation i.e. a single request being handled by the arbiter ignoring contention. The variable delay at the arbiter, described above, represents the queueing delay i.e. the additional delay due to competing traffic. A fixed delay derived from the FIFO implementation is also included in the analysis of chapter 6. Again the variable delay, queueing, is absent from the analysis. To model a loaded network for latency analysis, variable delays need to be measured.

In theoretical network analysis, queueing may be expressed using probability analysis [Kle76, Pad]. However, as stated in section 1.1, it has been shown by Hsu and Banerjee [HB90a], that many of these assumptions do not match the characteristics of real applications running on parallel computers. Therefore for more realistic analysis, variable delays should be derived from actual application traffic patterns.

The framework is designed to handle traffic down to the gate level. That is that the current implementation allows a flit to flow through the gate level description of a network, encountering other flits from other packets and queueing for shared resources where and when required. As such, the gate-level modelling provides a natural queueing model for analysis of latency with respect to contention.

At present, timing analysis is restricted to unloaded latency based on the longest path delay of the underlying design. That is, the fixed portion of latency. A form for tagging of data and updating of the simulation clock is required such that the flexible delay may be taken into the analysis. This feature is now brought under consideration.

7.2 Network Control Effects on Latency Expressions

An important feature, not generally included in latency analysis is control. Without some form of control in the network, the sending node has no way of knowing if the receiving node is ready to receive or not. In asynchronous networks, as described in section 4.2.1.17, control is usually implemented as request/acknowledge lines or control codes. In this section, consideration is given to the case of request/acknowledge signalling assuming a wormhole routed network and one-flit output buffers.

Statement 10 *The delay incurred by request signalling is overlapped with data transfer whereas acknowledge signals are pipelined back along the data path, compounding the delay at each hop.*

Assuming that a request line is driven high after data is placed on the outgoing channel, that is after data is valid then data is available at the receiver when the request is received. Data is either buffered at the input or traverses the router, buffering in the middle or at the output. In this work, output buffering is assumed.

As request signals are sent in parallel to the data, no additional delay is encountered by the request. On acceptance of the data by node(i+1), an acknowledge signal is returned to the previous node(i). This frees node(i)'s output buffer and allows it to accept data from node(i-1).

At node(i+1) processing of the input data continues while the acknowledgement signal is sent back to node(i). As such, the signal is again overlapped by data movement and, as such, no additional delay is incurred at node(i+1). However, the time taken for the acknowledgement to arrive at node(i) delays acceptance of data from node(i-1) at node(i).

The additional delay whilst awaiting an acknowledgement may be estimated as t_{wire}. Therefore, the data at node(i-1) is delayed by t_{wire}. However, although data is pipelined through the network i.e. forward towards the destination, acknowledgements are pipelined backwards along the path of transmission. The delay is accumulated along this path and is therefore dependent on the path length.

Statement 11 *In wormhole routing, in the presence of one-flit output buffers and request/acknowledge signalling, each of the flits of the body contribute more than the delay of the final hop to the network delay.*

Figure 7.2 illustrates delays incurred and the buffer contents as a 3-flit packet flows through the network. Considering first the right hand side of the diagram, each flit is processed at a node until a request is sent to the next node at which time, the flit is buffered in the output buffer. The flit remains in the output buffer until acknowledgement is received. That is the first flit, **F1** is buffered at node 1 until after the acknowledgement is received, that is that the flit is safely received at the next node. At this point the next flit may be received at the source. Processing at the second node begins as soon as the request is received and continues until the process repeats and a request is sent to the following node and an acknowledgement is awaited. It can be seen from the diagram that when the second and third flits flow through the network, gaps appear in the flow. That is, for example at Node 1, on receipt of the acknowledgement the buffer is cleared before a new request can be processed.

From the left side of figure 7.2 it can be seen that as **F1** is being processed at the destination, Node 2 receives an acknowledgement from the destination **Ack d-2**, thus freeing Node 2's buffer and allowing Node 1 to forward a flit **F2** to Node 2 by generating a request, **Req 1-2**. Flit **F3** on the other hand,

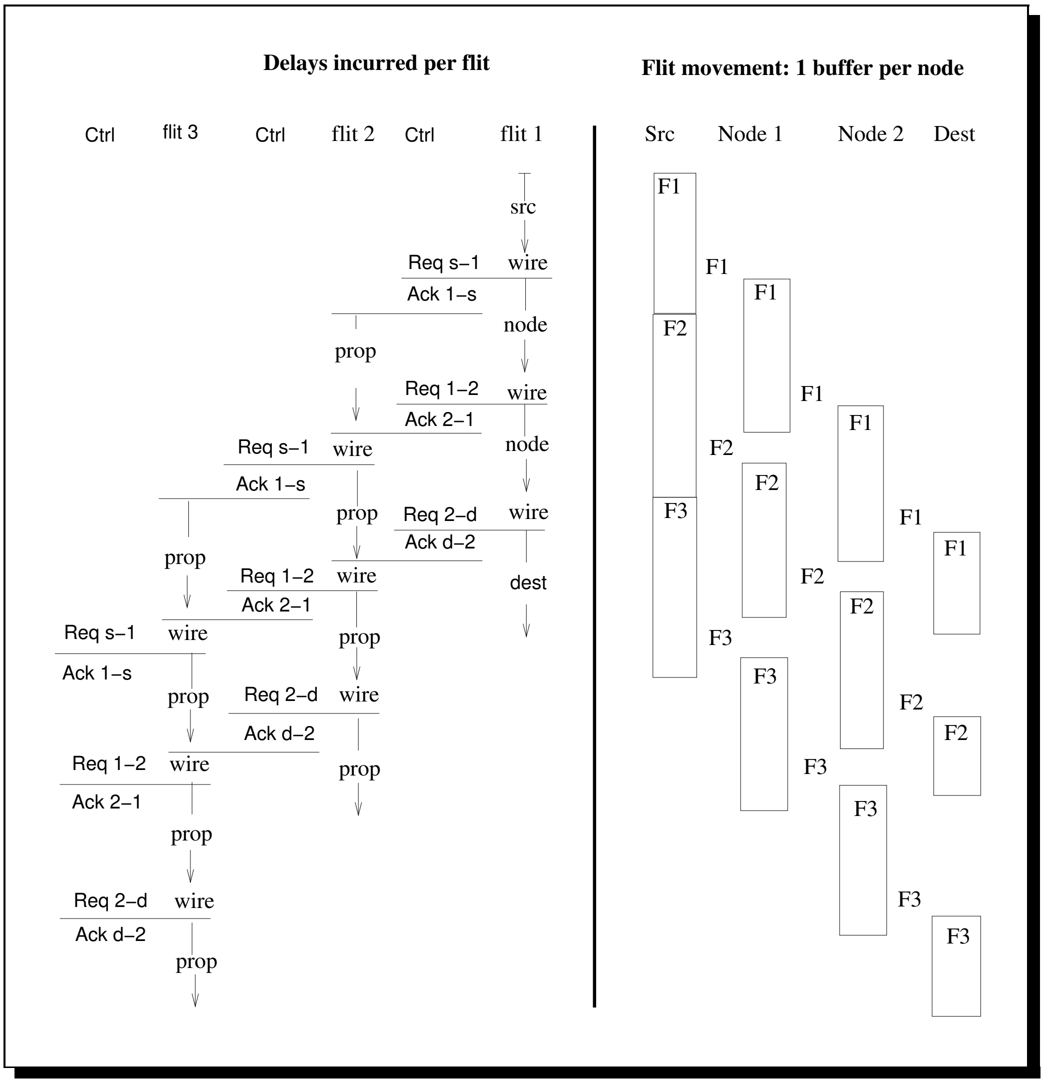

Figure 7.2: Wormhole Routing - One flit output buffer, 3 flit packet

must wait on the ack, **Ack 2-1** freeing buffer space at node 1, before it can send a request to Node 1.

This process of acknowledgements may be seen, both from the description above and figure 7.2, to delay both the start of each flit and progress of each flit. Considering flit 2, the request to Node 2 is delayed by the acknowledgement and, assuming for simplicity that t_{prop} is equivalent to t_{dest} then the time that F2 completes with respect to F1 is $(2 * t_{wire}) + t_{ack} + t_{prop}$. A similar scenario may be seen as F2 completes and F3 follows. Assuming that t_{ack} is equivalent to t_{wire} then the additional delay with respect to control signalling in this case may be seen to be $6 * t_{wire} + 2 * t_{prop}$.

This section has presented a theoretical discussion of the effect of control on latency analysis. The theoretical result presented which is based on a 3-flit packet indicates that investigation into buffer size with respect to packet size can give rise to improved theoretical latency expressions. Performance analyses of loaded networks are required to validate the above discussion.

7.3 Multilevel Architectural Modelling

This work set out to compare a more complex model developed using the framework (VHDLsyn) to that of a more typical network performance model (VERsim). The architecture used for this purpose is the Torus Routing Chip, a description of which is found along with the complex model description in section 6.1. VERsim may be said to be an algorithmic or behavioural approach at the system level. A detailed description of which may be found in [Nat97].

A multicomputer router was seen to offer a sufficiently complex architecture suitable for both modelling techniques. The additional element of asynchronous design in the TRC added an extra element to this work.

A comparison of the weaknesses and strengths of the two approaches is presented in table 7.1. It should be noted that comparison is based on a loaded network. Traffic analysis in VHDLsyn requires the inclusion of gate-level design descriptions within the design files. For smaller designs this does not cause a problem. However, for larger designs the unique naming requirements and the absence of the 'include' functionality in the design files makes

	VERsim	**VHDLsyn**
Language & size (no of lines)	Verilog (2000)	VHDL/PDL (3000)
Development Time	3 man months	5 man months
Complexity	medium	medium / high
HW Realisability	medium / poor	very good
Scalability	good	poor
Flexibility	poor	good
Execution Time	fast	fast

Table 7.1: Comparison of Model Characteristics

this process cumbersome and therefore does not allow for modelling larger systems within a reasonable development time. In unloaded analysis, the gate-level design descriptions within the design file are not required. Instead only the performance data is included which, as in the loaded case, is used to parameterise the model for analysis purposes.

A strong trade-off between scalability and flexibility is indicated for loaded networks. The term scalability is used to refer to the possibility of expanding the model for analysis of larger systems i.e. with an increased number of nodes.

An important feature of VHDLsyn is its flexibility, as discussed in section 5.1.

Asynchronous control particularly within the router has given rise to increased development and testing time in VHDLsyn. It is believed that a synchronous example would have reduced the development time. In VERsim the choice of synchronous or asynchronous control does not adversely effect

	VERsim	**VHDLsyn**
Cost	medium / poor	very good
Throughput	good	currently restricted
Latency (unloaded)	medium	very good
Latency (loaded)	medium	currently restricted

Table 7.2: Achievable Performance Metrics within the Models

the development time.

The initial aim for VERsim was to examine the usability of Verilog as a tool for system-level modelling. That is, to provide a test-bed to for experimenting with different network sizes and different traffic patterns. As such, HW realisation was not the primary focus. However, if the focus is on HW realisation it is possible to write a high-level simulator more attuned to this goal.

Neither of the two models showed excessive execution times in the tests undertaken illustrating the suitability of HDL languages for both these modelling approaches.

Goals within performance modelling may be categorised by the type of performance results being sought. In table 7.2 the models are compared for their abilities to handle different performance metrics.

If cost is an issue then a more complex model is required so that more realistic estimates of cost may be obtained. This need not necessarily be in the form of VHDLsyn but may also be in the form of a parameterised cost model as in [Chi93].

To measure throughput it is important that the model offers the ability to accept a variety of data inputs so that both the effect of different loads on the network and different traffic patterns may be tested. VERsim offers these possibilities. VHDLsyn, on the other hand, suffers from the data interface and statistical gathering limitations in the current version of PDL. However this problem is currently under investigation.

This project is continuing with the inclusion of BSPlab [DU97] which is an environment for experimenting with BSP programs on different computer architectures. Parallel applications are written in Visual C++ using the BSPlib standard [Hil97] for communication and synchronisation. One of the architectures that are available in BSPlab for executing BSP programs is a multicomputer where the nodes are interconnected in a 2-dimensional torus. This enables different parallel applications to be run on the Torus Routing Chip network such as to provide realistic traffic patterns for use in the former models.

It is believed that the abstract performance models of BSPlab may be calibrated by the more detailed performance models of VHDLsyn and VERsim. On the other hand, it is believed that both of these models will benefit from the traffic patterns generated from BSPlab. The aim of integrating these three models is to increase realism in the models by providing realistic load (top-down) and performance measurements closer to the actual hardware (bottom up).

Chapter 8

Conclusions and Future Directions

This chapter presents the conclusions derived from this work, some of the experiences obtained and recommendations for future work.

8.1 Concluding Remarks

The goal of this thesis has been to introduce and test out a new approach to modelling and performance analysis of communication hardware in parallel computers. That is to move away from the single problem, single solution concept to a more general problem but not necessarily general solution.

This work has attempted to define the communication hardware 'problem' by identifying the range of issues which may effect the design and performance of communication hardware. To solve the 'problem' a more complete modelling environment was sought and to this end a prototype modelling framework is presented.

To fully understand the effect of design changes, it is not sufficient to consider performance alone. Improvements in the performance of communication hardware have also led to an undesirable increase in cost. However, how should cost be treated? In performance analysis, the goal is to obtain better performance. However, in cost analysis, the question is not so much what is the cheapest cost but more is the performance benefit worth the cost? This work aims to draw attention to the cost factor in performance analysis and also to the added benefit which detailed modelling for cost analysis gives i.e. increased accuracy in performance results through more realistic performance data.

To summarise, a new approach to performance analysis of multicomputer communication hardware has been introduced combining the following three features:

- A common platform for the design and analysis of current, proposed and future design ideas and technologies within communication hardware.
- The expansion of design dependent cost analysis of routers within communication architecture modelling to a range of design issues.
- Improved accuracy in performance analysis due to the inclusion of gate-level performance data.

To conclude, the main achievement of this work may be summarised as answering the question can this approach be achieved? The answer off course being yes. The framework may be regarded as a prototype i.e. a possible solution to the new general approach to communication hardware modelling. As such, it does not constitute a complete solution but a significant step in this direction. Much work is still needed both to improve and expand the existing implementation. This is discussed in section 8.3.

8.2 Experiences

The work of case study 1 of section 6.1 provides an example in the use of logic synthesis for asynchronous circuits. Although asynchronous design is not new to the HDL community, logic synthesis is developed with synchronous circuit design in mind. In the creation of asynchronous modules, synthesisable asynchronous coding was found, not surprisingly, to be a challenge. In addition, both in asynchronous and synchronous coding, the importance of effective coding was highlighted when minor code changes created much larger designs i.e. a large increase in the number of CLBs.

Another unusual feature with respect to asynchronous design is the use of FPGA technology which is essentially a synchronous technology. This highlighted a new problem. As a synchronous technology, the timing analysator for FPGA designs is more attuned to synchronous analysis than to asynchronous analysis. This made longest path selection somewhat more complicated. As such, a more detailed description of the paths used for the results produced for the asynchronous modules are included in appendix A for completeness.

PDL is a newer language under development at the University of Cinncinati. The use of this language whilst in the development/expansion stage has led to a number of incomplete features within the implementation side of this work but has given rise to a test bed for testing out this newer language. It is believed that with the improvements which are scheduled to be included in the newer version, this language should be very suitable for this type of work.

8.3 Recommendations for Future Work

There are a number of issues and open problems that require further investigation. Chapter 7 has illustrated a number of issues currently under investigation. Areas of future work are summarised as follows:

- In the current implementation of the framework there is room for improvement. An important stage missing in the development of the design set is independent testing — as described in section 6.1.3.3. In addition, the gate level results are dependent on the synthesised HDL descriptions, a factor of which is the effectiveness of coding. Improvements in coding is desirable to provide more realistic performance data. Improvements in automisation and user friendliness of the framework is also desirable.

- Traffic handling is an essential feature for a realistic modelling environment. Implementation of this feature requires improvements in the

PDL interface to handle large trace files and a solution to the feedback problem currently existing in PDL. Expressions for throughput and loaded latency will also need to be added to the PDL model.

- Expansion of the design set to include examples synthesised into ASIC technology is desirable. In addition, further expansion of the design set is required such that a larger choice of module designs are available to meet different router design instances.

- The current converter from net-list to PDL format is based on the XNF format. Conversion is also required for EDIF format which is a more widely recognised standard.

- It would be interesting to consider which parts, if any, of the software level of the communication system could be integrated into the framework, thus providing a more complete framework for communication system modelling. One such interesting area is message passing standards such as the message passing interface, MPI. The first stage being to obtain knowledge as to how these are implemented on contemporary multicomputers and how communication performance is effected by the interaction between the hardware of the communication system and the operating system and message passing library.

- The investigations conducted have, as given in the title, focused on multicomputers. It is believed, however, that this work is also applicable to multiprocessors. Another interesting area is distributed systems. Distributed systems are also often direct connect systems and as such have many similarities to parallel computer networks. It would be interesting to investigate how much of the framework may be reused and otherwise adapted for this purpose.

Appendix A

Design Library

This appendix presents a summary of the designs in the current design set. It is chosen to present this summary as a table including the relevant performance data for each of the designs.

Examples of design implementations may be found in appendices B and C representing the asynchronous coded TRC routing module (dec_xy) and the synchronous coded Reliable Router routing module respectively.

The format of table A.1 is as follows:

Design	— design instance i.e. design name
HDL-file	— HDL file used
Device	— technology choice
Cost	— number of CLBs required to implement the design
Period	— (synchronous designs only)
	— minimum clock period
LPD	— (asynchronous designs only)
	— each path required to obtain part results shown
	— sum of path delays provides final result
	— values adjusted for pad delays where required
Notes	— brief description of design
	— identifies paths used for asynchronous analysis
	— identifies pad delays removed

Table A.1: The Set of Module Designs

	Design	**HDL-file**	**Device**	**Cost** clbs	**Period** (ns)	**LPD** (ns)	**Notes**
AC	addcomp8	AcAsyn	XC4036	5	—	7.086	8 bits Asyn zerochecker,TRC
						6.102	Din(pad) → Adrzero(pad)(6.24)
						0.984	Ready(pad) → EnRD(pad)(6.24)
	addcom16			9	—	7.476	16 bits Asyn zerochecker
						6.492	Din(pad) → Adrzero(pad)(6.24)
						0.984	Ready(pad) → EnRD(pad)(6.24)
	addcom32			18	—	9.370	32 bits Asyn zerochecker
						8.386	Din(pad) → Adrzero(pad)(6.24)
						0.984	Ready(pad) → EnRD(pad)(6.24)
	ac8syn	acsyn	XC4036	5	3.914	—	8 bits Syn zerochecker
	ac5xysyn	acxysyn	XC4036	16	9.925	—	(2 x 4 bit), Syn AC, RR
AD	ad8	adasyn	XC4036	4	—	6.536	8 bits, Asyn, TRC
						3.268	RDvalid(pad) → Decready(pad)(6.24)
						3.268	En(pad) → Req(pad) (6.24)
	ad16			8	—	6.536	16 bits, Asyn
						3.268	RDvalid(pad) → Decready(pad)(6.24)
						3.268	En(pad) → Req(pad) (6.24)
	ad32			16	—	6.536	32 bits, Asyn
						3.268	RDvalid(pad) → Decready(pad)(6.24)
						3.268	En(pad) → Req(pad) (6.24)

continued on next page

continued from previous page

	Design	HDL-file	Device	Cost	Period	LPD	Notes
	ad8syn	adsyn	XC4036	5	3.624	—	8bits Syn
AR	arb5whA	arb_whA	XC4036	43	—	12.954	5 req, wh, linear, TRC SelectO(pad) → Grant_op(net)(5.06)
	arb5sfA	arb_sfA	XC4036	19	—	6.235	5 req, sf, linear Req(pad) → Grant_op(pad)(6.24)
	arb5whSL	arb-whsL	XC4036	45	15.238	—	5 req wh sync linear
	arb6sfS	arb_sfS	XC4036	15	8.591	—	6 req, sf, Round robin
	ar6Prwhs	ar_prwhs	XC4036	21	10.875	—	6 req, wh, priority, sync
	ar5sfsrr	ar_sfsr	XC4036	15	8.04	—	5 req, sf, round robin, sync
AX	convert8	convert	XC4036	13	10.766	—	Abs to Rel, Sync
B	latch8as	latchasy	XC4036	4	—	5.448	8 bits latch, Asyn, TRC Din(pad) → Dout(pad)(6.24)
	latch16a			8	—	5.448	16 bits, Asyn, latch Din(pad) → Dout(pad)(6.24)
	latch32a			16	—	5.448	32 bits, Asyn, latch Din(pad) → Dout(pad)(6.24)
	latch4s	latch4sy	XC4036	2	3.624	—	4 bit Syn latch, RR
	latch8s	latsyn	XC4036	4	3.624	—	8 bits Syn latch
	latch16s			8	3.624	—	16 bits Syn latch
	latch75s			38	3.624	—	75 bits Syn latch
	latch96s	lat28sy	XC4036	48	3.624	—	96 bits Syn latch, RR sampler
CB	CB5-8a	crossas	XC4036	71	—	10.12	5x5,8 bits,Asyn, TRC

continued on next page

continued from previous page

	Design	HDL-file	Device	Cost	Period	LPD	Notes
							Grant(pad) $\rightarrow$ AckArb(pad)(6.24)
	CB5_16a			123	—	10.12	5x5,16 bits, Asyn Grant(pad) $\rightarrow$ AckArb(pad)(6.24)
	CB5_32a			226	—	10.12	5x5,32 bits, Asyn
	CB5_8s	crossyn	XC4036	45	11.605	—	5x5, 8 bits, Syn
	CB6_78s	cbsyn2	XC4044	1448	12.980	—	6x6,78 bits, Syn
	CB6_70s			1306	12.980	—	6x6,70 bits, Syn, RR
CC	CC8-4a	fifoasyn	XC4036	88	—	19.55	8 wide, 4 deep, TRC
	cc-rr	latsyn	XC4036	38	3.624	—	75 bit latch, RR
FC	flowctrl	flowctrl	XC4036	10	15.733	—	RR, control of 16 bit buffer
Fifo	fo8-4a	fifoasyn	XC4036	88	—	19.550 14.703 4.847	8 wide, 4 deep, TRC Req(pad) $\rightarrow$ Fifo(net)(1.18) Fifo(clb) $\rightarrow$ Dout(pad)(5.06)
	fo16-4a			137	—	19.550	16 wide, 4 deep Req(pad) $\rightarrow$ Fifo(net)(1.18) Fifo(clb) $\rightarrow$ Dout(pad)(5.06)
	fo32-4a			231	—	19.550	32 wide, 4 deep Req(pad) $\rightarrow$ Fifo(net)(1.18) Fifo(clb) $\rightarrow$ Dout(pad)(5.06)
	fo6-16a			539	—	20.375 15.528	8 wide, 16 deep, async Req(pad) $\rightarrow$ Fifo(net)(1.18)

continued on next page

continued from previous page

	Design	HDL-file	Device	Cost	Period	LPD	Notes
						4.847	Fifo(clb) → Dout(pad)(5.06)
	fo8-4s	fifosyn	XC4036	60	12.608	—	8 wide, 4 deep, Sync
	fo78-16		XC4062	1998	16.759	—	78 wide, bus, 16 deep, sync
	fo67-16	fifosynE	XC4052	1782	17.866	—	67 wide, bus, 16 deep, sync, RR
FU	fuRR	flitupd	XC4036	37	7.038	—	add VCI to flit, RR
IJ	ij8	fifoasyn	XC4036	88	—	19.55	8 wide, 4 deep fifo, asyn
MB	messbld	messbld	XC4036	91	4.023	—	4 frames → flit, parity check, RR
MS	messplt	mesSplit	XC4036	42	3.624	—	flit → 4 frames, RR
Mx	mux2-8a	muxasyn	XC4036	5	—	4.898	2-1, asyn, 8 bits data, TRC DataA(pad) → Mux_out(pad)(6.24)
	mux2-16a			9	—	4.898	2-1, 16 bits data, asyn DataA(pad) → Mux_out(pad)(6.24)
	mux2-32a			17	—	4.898	2-1, 32 bits data, asyn DataA(pad) → Mux_out(pad)(6.24)
	mx2-11s	multsyn	XC4036	7	7.641	—	2-1, sync, 11 bits control
	mx5-11s			21	10.475	—	5x1, 11 bits, control
	mx5-14			25	10.457	—	5x1,14 bits,control,round robin,RR
	mx2-8sD	multsynD	XC4036	4	3.073	—	2-1, sync, 8 bits data
Mxctrl	Mxctrl2A	ctrlAsyn	XC4036	1	—	4.919	fair, 2 reqs, TRC Res(pad) → Grant(pad)(6.24)
	multct2s	MultCtsy	XC4036	1	3.073	—	fair, sync, 2 reqs
	Multct5s			73	40.362	—	5 reqs, RR

continued on next page

continued from previous page

	Design	HDL-file	Device	Cost	Period	LPD	Notes
	Mxcsyac2	MxCtsyAK	XC4036	3	3.472	—	sync, 2reqs, fair,ack
OP	opas8	outctasy	XC4036	5	—	4.847	8 bit, 2 VCs, TRC State0(clb) → Req_out(pad)(5.06)
	opas16			4	—	4.847	16 bit, 2 VCs State0(clb) → Req_out(pad)(5.06)
	opas32			5	—	4.847	32 bit, 2 VCs State0(clb) → Req_out(pad)(5.06)
	opsy8	latsyn	XC4036	4	3.624	—	8 bit latch, no signalling reqd
	op24fr4s	op24_4fs	XC4036	64	13.257	—	RR, 4 frame, parity add
RD	XYrouta (head) (body)	dec_xy	XC4036	43	—	 33.932 13.672	deter, unitorus, TRC State_0(clb) → State_25(net) State_11(clb) → State_15(net)
	XYroutS	dec_xyS	XC4036	23	16.174		sync TRC
	syncrd	rd-rr	XC4036	34	22.494	—	sync RR
SC	Sc_inp	sci	XC4036	9	—	7.756	input,TRC State0(clb) → Reqi(pad)(5.06)
	Sc_op	sco	XC4036	13	—	9.415	output,TRC State0(clb) → Req_out(pad)(5.06)
SM	sig_mgr	sigmgr	XC4036	5	—	13.695	TRC State0(clb) → Ack_in(pad)(5.06)
SY	—	—	—	—	1cc	—	RR

Appendix B

TRC: Routing Decision Module (VHDL)

"Dec_xy.vhd"

```
library IEEE;
use IEEE.std_logic_1164.all;
library Routerpar;
use Routerpar.Design_Issues.all;

entity Dec_xy is
  port(EN       : in STD_LOGIC;
    Res        : in STD_LOGIC;
    Valid      : in STD_LOGIC;          -- AC/RD handshake
    EnRD       : in STD_lOGIC;          -- AC/RD handshake
    Dec_ready  : in STD_LOGIC;          -- RD/AD handshake
    Adrzero    : in STD_LOGIC;          -- result
    EdgeID     : in STD_LOGIC_VECTOR(Degree-1 downto 0);   --edge=1
    Inchannel  : in STD_LOGIC_VECTOR(DataPaths-1 downto 0);--channel
    Full       : in STD_LOGIC_VECTOR(DataPaths-1 downto 0);--full=1
    Ack        : in STD_LOGIC;          -- CB complete
    Dec        : out STD_LOGIC;         -- result
    Select_IP  : out STD_LOGIC;         -- arb request
    Select_O   : out STD_LOGIC_VECTOR(DataPaths-1 downto 0);--CB-op
    Acki       : out STD_LOGIC;         -- receipt complete
    Ready      : out STD_LOGIC;         -- AC/RD handshake
    Dec_valid  : out STD_LOGIC;         -- RD/AD handshake
```

```
    EnDec       : out STD_LOGIC;          -- RD/AD handshake
    Tail        : out STD_LOGIC           -- end of packet
        );
end Dec_xy;

architecture Dec_xy_fsm of Dec_xy is

  type ONEHOT_STATE_TYPE is(Start_state,Wait_state,Dec2Adr,Updatex,
       X_OK,Handshake1,Dec2yAdr,Updatey,Y_OK,Handshake2,Busy2_state,
       Free_buffer,Data_sent,Data2_state,Handshake3,Tail_state,
       Tail_buff,Tail_sent,Handshake4,YWait_state,Ythrough,Handshake5,
       Checky, Processy, P_ack, P_OK);

  attribute ENUM_encoding : string;
  attribute ENUM_encoding of ONEHOT_STATE_TYPE: type is "one hot";
  signal Current_State, Next_State: ONEHOT_STATE_TYPE;
  signal temp,temp1,temp2 : std_logic_vector(Datapaths-1 downto 0);

  function newtemp
    (dim:string;Inch:std_logic_vector(Datapaths-1 downto 0);
                    Edge:std_logic) return std_logic_vector is
  -- implements deadlock prevention in routing algorithm
  -- Changes vc. at an edge or change of direction, if necessary
    variable temp_sel : std_logic_vector(Datapaths-1 downto 0);
  begin
    if dim = "x" then
      temp_sel := xVC1;
      if Inch = xVC0 or Edge = '1'  then
        temp_sel := xVC0;
      end if;
    else
      temp_sel := yVC1;
      if Inch = yVC0 or Edge = '1'  then
        temp_sel := yVC0;
      end if;
    end if;
    return temp_sel;
  end newtemp;

function waitonbuffer
  (opchoice:std_logic_vector;full:std_logic_vector)return boolean is
  -- stops further processing until space at output
  -- is sending to a full buffer, waits
```

```
  type i_type is range 0 to opchoice'length;
  variable i : i_type;
  variable waitonfull : boolean;
begin
    assert opchoice'length = full'length
      report "unequal vector lengths"
      severity warning;
    test : for i in opchoice'range loop
      waitonfull := false;
      if opchoice(i) = '1' and full(i) = '1' then
        waitonfull := true;
      end if;
    end loop test;
  return waitonfull;
end waitonbuffer;

begin

adjusttemp : process (Inchannel, EdgeID)
  -- adjusts channel choice for deadlock prevention
begin
  temp1 <= newtemp("x", Inchannel, EdgeID(0));
  temp2 <= newtemp("y", Inchannel, EdgeID(1));
end process;

updates : process(En, Res, Next_state)
  -- controls operation of state machine
  -- asynchronous, En held high
begin
  if Res = '1' then
    Current_State <= Start_state;
  elsif EN = '1' then
    Current_state <= Next_state;
  else
    Current_state <= current_state;
  end if;
end process;

main :  process (Current_State, ack, Dec_ready, temp, temp1, temp2,
                    valid, EnRD, Adrzero, Inchannel, EdgeID,  Full)
begin
  Case Current_State is
```

```
    when Start_state =>
      Acki <= '0';
      EnDec <= '0';
      Dec <= '0';
      Select_IP <= '0';
      Select_O <= clear;
      Ready <= '0';
      Dec_valid <= '0';
      temp <= temp1;
      tail <= '0';
      if Valid = '1' then        -- AC decision ready, H/sh
        Next_state <= Wait_state;
      else
        Next_state <= Start_state;
      end if;

      when Wait_state =>
        Acki <= '0';
        EnDec <= '0';
        Dec <= '0';
        Select_IP <= '0';
        Select_O <= clear;
        Ready <= '1';            -- h/sh with AC
        temp <= temp1;
        Dec_valid <= '1';        -- h/sh with AD
        tail <= '0';
        if EnRD = '1' and Adrzero = '0'  then   --send in x
          Next_state <= Dec2Adr;
        elsif EnRD = '1' and Adrzero = '1' then --check y add.
          Next_state <= YWait_state;
        else
          Next_state <= Wait_state;
        end if;

-- if AC result = 0, x address is not zero
-- packet has arrived from x or p

      when Dec2Adr =>
        Acki <= '0';
        EnDec <= '0';           -- await h/sh from AD
        Dec <= '1'; -- decrement decision
        Ready <= '1';
        Select_IP <= '0';
```

```
    Select_O <= clear;
    Dec_valid <= '1';
    temp <= temp1;
    tail <= '0';
    if Dec_ready = '1' and waitonbuffer(temp, full) = false then
      Next_state <= Updatex;    -- AD ready, buffer available
    else
      Next_state <= Dec2Adr;
    end if;

  when Updatex =>
    Acki <= '0';
    EnDec <= '1';            -- h/sh AD
    Dec <= '1';
    Ready <= '1';
    Select_IP <= '1';        -- request AR
    Select_O <=  temp;       -- op decision
    Dec_valid <= '1';
    tail <= '0';
    if ack = '1'  then       -- CB complete
      Next_state <= X_OK;
    else
      Next_state <= Updatex;
    end if;

  when  X_OK =>
    Acki<= '1';        -- ack packet receipt (SM)
    EnDec <= '1';
    Dec <= '1';
    Ready <= '1';
    Select_IP <= '1';
    Select_O <= temp;
    Dec_valid <= '1';
    tail <= '0';
    if valid = '0'  then  -- original signal cleared
      Next_state <= Handshake1;
    else
      Next_state <= X_OK;
    end if;

  when Handshake1 =>   -- await new flit
    Acki <= '0';
    Endec <= '0';
```

```
        Dec <= '0';
        Ready <= '0';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '0';
        if valid = '1'  then -- new flit, new AC decision
          Next_state <= Busy2_state;  -- data arrived
        else
          Next_state <= Handshake1;
        end if;

-- packet is to be sent on y

      when Dec2yAdr =>
        Acki <= '0';
        EnDec <= '0';
        Dec <= '1';
        Ready <= '1';
        Select_IP <= '0';
        Select_O <= clear;
        Dec_valid <= '1';
        tail <= '0';
        if Dec_ready = '1' and waitonbuffer(temp,full) = false then
          Next_state <= Updatey;  -- AD ready, free buffer
        else
          Next_state <= Dec2yAdr;
        end if;

      when Updatey =>
        Acki <= '0';
        EnDec <= '1';
        Dec <= '1';
        ready <= '1';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '0';
        if ack = '1'  then  -- CB complete
          Next_state <= Y_OK;
        else
          Next_state <= Updatey;
        end if;
```

```
      when Y_OK =>
        Acki <= '1';
        EnDec <= '1';
        Dec <= '1';
        Ready <= '1';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '0';
        if valid = '0'  then -- clearing signals
          Next_state <= Handshake2;
        else
          Next_state <= Y_OK;
        end if;

      when Handshake2 =>
Acki <= '0';
        EnDec <= '0';
        Dec <= '0';
        Ready <= '0';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '0';
        if  valid = '1'  then  -- new flit, data
          Next_state <= Busy2_state;
        else
          Next_state <= Handshake2;
        end if;

-- data/tail  flits arrive

      when Busy2_State =>
        Acki <= '0';
        EnDec <= '0';
        Dec <= '0';
        Ready <= '1';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '0';
        if EnRD = '1' and Adrzero = '0'  then
```

```
        Next_state <= Free_buffer;  -- data flit received
      elsif EnRD = '1' and Adrzero = '1'  then
        Next_state <= Tail_state;   -- tail flit received
      else
        Next_state <= Busy2_state;
      end if;

    when Free_buffer =>
      Acki <= '0';
      EnDec <= '0';
      Dec <= '0';
      Ready <= '1';
      Select_IP <= '1';
      Select_O <= temp;
      Dec_valid <= '1';
      tail <= '0';
      if Dec_ready = '1' and  waitonbuffer(temp, full) = false then
        Next_state <= Data_sent;    -- AD ready, available buffer
      else
        Next_state <= Free_buffer;
      end if;

    when Data_sent =>
      Acki <= '0';
      EnDec <= '1';   -- AD enabled
      Dec <= '0';     -- Dec = no decrement
      Ready <= '1';
      Select_IP <= '1';
      Select_O <= temp;
      Dec_valid <= '1';
      tail <= '0';
      if  ack = '1' then -- cb complete
        next_state <= Data2_state;
      else
        next_state <= Data_sent;
      end if;

    when Data2_state =>
      Acki <= '1';
      EnDec <= '1';
      Dec <= '0' ;
      Ready <= '1';
      Select_O <= temp;
```

```
        Select_IP <= '1';
        Dec_valid <= '1';
        tail <= '0';
        if valid = '0'  then  -- signals clearing
          Next_State <= Handshake3;
        else
          Next_state <= Data2_state;
        end if;

      when Handshake3 =>
        Acki <= '0';
        EnDec <= '0';
        Dec <= '0';
        Ready <= '0';
        Select_O <= temp;
        Select_IP <= '1';
        Dec_valid <= '0';
        tail <= '0';
        if valid = '1'  then -- new flit
          Next_State <= Busy2_state;
        else
          Next_State <= Handshake3;
        end if;

-- sending tail flit

      when Tail_state=>
        Acki <= '0';
        EnDec <= '0';
        Dec <= '0';
        Ready <= '1';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '1';
        if Dec_ready = '1' and  waitonbuffer(temp, full) = false then
          Next_state <= Tail_buff; --AD ready, free buffer space
        else
          Next_state <= Tail_state;
        end if;

      when Tail_buff =>
        Acki <= '0';
```

```
        EnDec <= '1';  -- enable AD
        Dec <= '0';    -- no decrement
        Ready <= '1';
        Select_IP <= '1';
        Select_O <= temp;
        Dec_valid <= '1';
        tail <= '1';
        if  ack = '1' then -- cb complete
          Next_state <= Tail_sent;
        else
          Next_state <= Tail_buff;
        end if;

      when Tail_sent =>
         Acki <= '1';
         EnDec <= '1';
         Dec <= '0';
         Ready <= '1';
         Select_O <= temp;
         Select_IP <= '1';
         Dec_valid <= '1';
         tail <= '1';
         if valid = '0'  then -- clearing signals
           Next_State <= Handshake4;
         else
           Next_state <= Tail_sent;
         end if;

      when Handshake4 =>
         Acki <= '0';
         EnDec <= '0';
         Dec <= '0';
         Ready <= '0';
         Select_O <= clear;
         Select_IP <='0';
         Dec_valid <= '1';
         tail <= '0';
         if Dec_ready = '0'   then -- ensure all signals are cleared
           Next_state <= Start_state;
         else
           Next_state <= Handshake4;
          end if;
```

```
-- check y address

       when YWait_state =>
         Acki <= '0';
         EnDec <= '0';
         Dec <= '0';
         Ready <= '1';
         temp <= temp2;
         Select_O <= clear;
         Select_IP <= '0';
         Dec_valid <= '1';
         tail <= '0';
         if Dec_ready = '1'  then -- AD h/sh
           Next_state <= Ythrough;
         else
           Next_state <= YWait_state;
         end if;

       when Ythrough =>
         Acki <= '1';   -- ack x address and await y address,
         EnDec <= '0';
         Dec <= '0';
         Ready <= '1';
         Select_O <= clear;
         Select_IP <= '1';
         Dec_valid <= '1';
         temp <= temp2;
         tail <= '0';
         if valid = '0'  then  -- signals clearing
           Next_state <= Handshake5;
         else
           Next_state <= Ythrough;
         end if;

       when Handshake5 =>
         Acki <= '0';
         EnDec <= '0';
         Dec <= '0';
         Ready <= '0';
         Select_O <= clear;
         Select_IP <= '0';
         Dec_valid <= '0';
         temp <= temp2;
```

```
          tail <= '0';
          if  valid = '1' then  -- new flit, check y address
            Next_state <= Checky;
          else
            Next_state <= Handshake5;
          end if;

        when Checky =>
          Acki <= '0';
          EnDec <= '0';
          Dec <= '0';
          Ready <= '1';
          Select_O <= clear;
          Select_IP <= '0';
          Dec_valid <= '1';
          temp <= temp2;
          tail <= '0';
          if EnRD = '1' and Adrzero = '1'   then
            Next_state <= Processy;  -- at destination
          elsif EnRD = '1' and Adrzero = '0' then
            Next_state <= Dec2yAdr;  -- send in y
          else
            Next_state <= Checky;
          end if;

-- at destination

        when Processy =>
          Acki <= '0';
          EnDec <= '0';
          Dec <= '0';
          Ready <= '1';
          Select_IP <= '0';
          Select_O <= clear;
          Dec_valid <= '1';
          tail <= '0';
          if  Dec_ready = '1' and waitonbuffer(p,full) = false then
            Next_state <= P_ack;  -- AD ready, buffer space
          else
            Next_state <= Processy;
          end if;

        when P_ack =>
```

```
        Acki <= '0';
        EnDec <= '1';  -- AD enabled
        Dec <= '0';    -- no decrement
        Ready <= '1';
        Select_IP <= '1';
        Select_O <= p;
        Dec_valid <= '1';
        tail <= '0';
        if ack = '1' then  -- CB complete
          Next_state <= P_OK;
        else
          Next_state <= P_ack;
        end if;

      when  P_OK =>
        Acki <= '1';
        EnDec <= '1';
        Dec <= '0';
        Ready <= '1';
        Select_O <= p;
        temp <= p;
        Select_IP <= '1';
        Dec_valid <= '1';
        tail <= '0';
        if valid = '0'  then -- signals clearing
          Next_state <= Handshake3; -- await data
        else
          Next_state <= P_OK;
        end if;

     end Case;
end process;

end dec_xy_fsm;
```

Appendix C

Reliable Router : Routing Decision Module (VHDL)

"Rd_rr.vhd"

```
library IEEE;
use IEEE.std_logic_1164.all;
library Routerpar;
use Routerpar.Design_Issues.all;

entity rd_rr is
  generic (seed : std_logic_vector(1 downto 0) := "01") ;
    port (
      Reset      : in STD_LOGIC;
      CLK        : in STD_LOGIC;
      En         : in STD_LOGIC;      -- RD module selection
      AckCB      : in STD_LOGIC;      -- CB acknowledgement
      Kind       : in STD_LOGIC_VECTOR(2 downto 0);--flit type
      Fault      : in STD_LOGIC_VECTOR(Datapaths-1 downto 0);
      AC_dec     : in STD_LOGIC_VECTOR(3 downto 0);--x,y decision
      Prior      : in STD_LOGIC;      -- VC priority
      Res_op     : in STD_LOGIC_VECTOR(DataPaths-1 downto 0);
                                      -- non available outputs
      Empty      : in STD_LOGIC;      -- awaiting data
      Ok_to_send : in STD_LOGIC;      -- flow control
      RD_ID      : in STD_LOGIC_VECTOR((NoVC+Datapaths-1) downto 0);
                                      -- RD module address
      RD_pr      : out STD_LOGIC;     -- forces highest AR priority
```

```
      VC_Req     : out STD_LOGIC;     -- request to VC
      Dec        : inout STD_LOGIC_VECTOR(10 downto 0)
                                      -- routing decision
    );
end rd_rr;

architecture rd_rr_behav of rd_rr is

  type states is ( Idle_state, Needs_rout, Routed);
  signal nxt_state  : states;
  signal Req_dec    : Std_logic;
  signal Channelflt : std_logic;
  signal Rand : std_logic_vector(1 downto 0);
  signal Rand_upd : std_logic_vector(1 downto 0);
  signal Go : std_logic;

  function choice ( rand, V1, V2: STD_LOGIC_VECTOR)
                                   return STD_LOGIC_VECTOR is
    variable temp: std_logic_vector(V1'length-1 downto 0);
    -- makes random channel selection
    begin
      if Rand = "01" then
        temp := V1;
      elsif Rand = "00" then
        temp := V2;
      else
        temp := V1;
      end if;
    return temp;
  end;

  function adaptive (Rand,ACdec, Res_op: STD_LOGIC_VECTOR;
               Priority : STD_LOGIC) return STD_LOGIC_VECTOR is
    -- implementation of the adaptive routing algorithm
    variable Phy_channel : std_logic_vector(Datapaths-1 downto 0);
    variable VC_channel  : std_logic_vector(NoVC-1 downto 0);
    variable zerovector  : std_logic_vector(Datapaths-1 downto 0);
  begin
     zerovector := (others => '0');
    if ACdec(1 downto 0)   = match then -- at x address
      if ACdec(3 downto 2) = match then -- at y address
        if Res_op(4) = '1' then                 -- P_RR reserved
          Phy_channel := (others => '0');
```

```
      else
        Phy_channel := p; -- send to processor
      end if;
    elsif ACdec(3 downto 2) = plus  then
      if Res_op(2) = '1' then                   -- Yplus reserved
        Phy_channel := (others => '0');
      else
        Phy_channel := yVC1;          -- send to y+
      end if;
    else
      if Res_op(3) = '1' then                   -- Yminus reserved
        Phy_channel := (others => '0');
      else
        Phy_channel := yVC0; -- send to y-
      end if;
    end if;
  elsif ACdec(1 downto 0) = plus then
    if ACdec(3 downto 2) = match then           -- at y address
      if Res_op(0) = '1' then                   -- P_RR reserved
        Phy_channel := (others => '0');
      else
        Phy_channel := XVC1; -- send to x+
      end if;
    elsif ACdec(3 downto 2) = plus then
      if Res_op(2) = '0' then         -- y+ free
        if Res_op(0) = '0' then -- x+ free
          Phy_channel := choice(Rand, yVC1, xVC1);--choice y+,x+
        else
          Phy_channel := yVC1;
        end if;
      else -- y+ reserved
        if Res_op(0) ='0' then -- x+ free
          Phy_channel := xVC1;
        else -- y+, x+ reserved
          Phy_channel := (others =>'0');
        end if;
      end if;
    else
      if Res_op(3) = '0' then         -- y- free
        if Res_op(0) = '0' then -- x+ free
          Phy_channel := choice(Rand, yVC0, xVC1);--choice y-,x+
        else
          Phy_channel := yVC0; -- send to y-
```

```
          end if;
        else -- y- reserved
          if Res_op(0) ='0' then -- x+ free
            Phy_channel := xVC1;
           else -- y-, x+ reserved
             Phy_channel := (others =>'0');
           end if;
         end if;
      end if;
    elsif ACdec(1 downto 0) = minus then
      if ACdec(3 downto 2) = match then -- at y address
        if Res_op(1) = '1' then                -- P_RR reserved
          Phy_channel := (others => '0');
        else
          Phy_channel := xVC0;         -- send to x-
        end if;
      elsif ACdec(3 downto 2) = plus then
        if Res_op(2) = '0' then -- y+ free
          if Res_op(1) = '0' then -- x- free
            Phy_channel := choice(Rand, yVC1, xVC0);--choice y+,x-
          else
            Phy_channel := yVC1; -- send to y+
          end if;
        else -- y+ reserved
          if Res_op(1) ='0' then -- x- free
            Phy_channel := xVC0; -- send to x-
          else -- y+, x- reserved
            Phy_channel := (others =>'0');
          end if;
        end if;
      else
        if Res_op(3) = '0' then -- y- free
          if Res_op(1) = '0' then -- x- free
            Phy_channel := choice(Rand, yVC0, xVC0);--choice y-,x-
          else
            Phy_channel := yVC0; -- send to y-
          end if;
        else -- y- reserved
          if Res_op(1) ='0' then -- x- free
            Phy_channel := xVC0;               -- send to x-
          else -- y-, x- reserved
            Phy_channel := (others =>'0');
          end if;
```

```
        end if;
      end if;
    else
      Phy_channel := (others => '0');
    end if;
    if Phy_channel /= zerovector then
      if Priority = '1'  then -- VC selection
        VC_channel := VC0_5;         -- adapt1
      else
        VC_channel := VC1_5;
      end if;                 -- adapt2
    else
      VC_channel:= (others => '0');
    end if;
    return VC_channel & Phy_channel;
  end;

  function deterministic(ACdec:STD_LOGIC_VECTOR; Priority:STD_LOGIC)
                                          return STD_LOGIC_VECTOR is
    -- implements deterministic routing algorithm
    variable Phy_channel : std_logic_vector(5 downto 0);
    variable VC_channel : std_logic_vector(4 downto 0);
  begin
    if ACdec(1 downto 0)   = match then -- at x address
      if ACdec(3 downto 2) = match then -- at y address
        Phy_channel := p; -- send to processor
      elsif ACdec(3 downto 2) = plus  then
        Phy_channel := yVC1;  -- send to y+
      else Phy_channel := yVC0; -- send to y-
     end if;
    elsif ACdec(1 downto 0)   = plus then
      Phy_channel := xVC1; -- send to x+
    else
      Phy_channel := xVC0; -- send to x-
    end if;
    if Priority = '1' then -- VC selection
      VC_channel := VC2_5; -- deter1
    else
      VC_channel := VC3_5; -- deter0
    end if;
    return VC_channel & Phy_channel;
  end;
```

```
function turnmodel( Rand, ch_fault : Std_logic_vector)
                                   return STD_LOGIC_VECTOR is
  -- implements fault tolerance algorithm, turn model
  variable Phy_channel : std_logic_vector(5 downto 0);
begin
  if ch_fault = xVC1 then   -- x+ fault
    if Rand = "01" then
      Phy_channel := xVC1;
    elsif Rand = "00" then
      Phy_channel := yVC1;
    else
     Phy_channel := yVC0;
    end if;
  elsif ch_fault = xVC0 then -- x- fault
    if Rand = "01" then
      Phy_channel := xVC0;
    elsif Rand = "00" then
      Phy_channel := yVC1;
    else
      Phy_channel := yVC0;
    end if;
  elsif ch_fault = yVC1 then -- y+ fault
    if Rand = "01" then
      Phy_channel := yVC1;
    elsif Rand = "00" then
      Phy_channel := xVC1;
    else
     Phy_channel := xVC0;
    end if;
  elsif ch_fault = yVC0 then         -- y- fault
    if Rand = "01" then
      Phy_channel := yVC0;
    elsif Rand = "00" then
      Phy_channel := xVC1;
    else
     Phy_channel := xVC0;
    end if;
  else
    Phy_channel := (others => '0');
  end if;
 return Phy_channel;
end;
```

```
begin

  random : process(Reset, clk, rand_upd)
    -- generates random value
    variable temp : std_logic;
  begin
     if Reset = '1' then
       Rand_upd <= seed;
       Rand <= "00";
     elsif clk'event and clk = '1' then
       Rand <= Rand_upd;
     else
       temp := Rand_upd(1) xor Rand_upd(0);
       Rand_upd <= not temp & Rand_upd(1);
    end if;
  end process;

 routing: process (Reset, CLK, Rand, kind, AC_dec,
                                 Prior, Fault,  Res_op, En)
   -- makes routing decision from choice of routing algorithms
   variable Adapt, Deter,  zeroadd : STD_LOGIC_VECTOR(10 downto 0);
   variable zerovector : std_logic_vector(Datapaths-1 downto 0);
 begin
   zerovector := (others =>'0');
   if Reset = '1'  then
     Dec <= (others => '0');
     Req_dec <= '0';
   elsif CLK'event and CLK = '1' and En = '1' then
     zeroadd := (others => '0');
     Adapt := adaptive(Rand, AC_dec, Res_op, Prior);
     Deter := deterministic(AC_dec, Prior);
     if kind = token then
       Req_dec <= '0';
       Dec <= Dec;
     elsif kind = head then
       if (RD_ID = VC4_5 & xVC1) or (RD_ID = VC4_5 & xVC0) then
         -- previous turn in the x direction
         -- continue on fault channels
         Dec(5 downto 0) <= turnmodel(Rand, Deter(5 downto 0));
         Dec(10 downto 6) <= VC4_5;      -- faulty VC
       else
         --adaptive first and then deterministic
         --     if not sending on a faulty channel
```

```
        if  Adapt(Datapaths-1 downto 0) /= Fault then
           --choose adaptive choice
          Dec <= Adapt;
        elsif  Fault = Deter(Datapaths-1 downto 0) then
          -- choose fault choice
          Dec(5 downto 0) <= turnmodel(Rand, Deter(5 downto 0));
          Dec(10 downto 6) <= VC4_5;  -- faulty VC
        else                   -- choose deterministic
          Dec <= Deter;
        end if;
      end if;
      Req_dec <='1';
    else
      Dec <= Dec;
    end if;
  end if;
end process;

state : process (Reset, CLK, nxt_state, Kind, AckCB, En)
  -- state machine control of routing process
  variable refusal : integer range 0 to 15;
  variable zerovector : std_logic_vector(datapaths-1 downto 0);
  begin
    zerovector := (others => '0');
    if Reset = '1' then
      nxt_state <= idle_state;
      refusal := 0;
      RD_pr <= '0';
    elsif CLK'event and CLK = '1' and En = '1' then
        case nxt_state is
          when idle_state =>
            if Kind = head and AckCB = '0' then
              nxt_state <= Needs_rout;
            elsif Kind = head and AckCB = '1' then
              nxt_state <= Routed;
            end if;
            refusal:= 0;
          when Needs_rout =>
            if AckCB = '1' then
              nxt_state <= Routed;
              refusal := 0;
            elsif AckCB = '0' then
              nxt_state <= Needs_rout;
```

```
              refusal := refusal +1;
            end if;
          when Routed =>
            if AckCB = '1' and Kind = token then
              nxt_state <= Idle_state;
              refusal := 0; -- eop, CB acceptance
            else
              nxt_state <= Routed;
              if AckCB = '0' then -- not eop, CB refusal
                refusal := refusal + 1;
              else
                refusal := 0; -- not eop, Cb acceptance
              end if;
            end if;
        end case;
        if refusal >= 7 then           -- changes AR priority
          RD_pr <= '1';
        else
          RD_pr <= '0';
        end if;
      end if;
   end process;

control : process (CLK, Reset, OK_to_send, Empty, Req_dec,
                                          nxt_state, Kind, En)
   -- decides if VC is eligible to send, if true VC_req = 1
 begin
   if Reset = '1' then
       VC_req <= '0';
   elsif CLK'event and CLK = '1' and En = '1' then
       if empty = '0' and OK_to_send = '1' then
         -- if fifo is not empty, free buffer space, vc selected
         if nxt_state = routed and (Kind = data or Kind = tail)  then
           -- routing decision previously made
           VC_Req <= '1';
         elsif nxt_state = needs_rout and Req_dec = '1' then
           -- awaits completion of routing decision, Req_dec
           VC_Req <= '1';
         else
           VC_req <= '0';
         end if;
       else
         VC_req <= '0';
```

```
       end if;
    end if;
  end process;
end rd_rr_behav;
```

Appendix D

Design Set : VHDL Library

“Routerpar.vhd”

```
-- Holds all design issues which parameterise the HDL code
-- of the design library
-- Other constants and functions used by several modules
-- Parameters given match those of the Reliable Router

library IEEE;
use IEEE.std_logic_1164.ALL;

package Design_issues is

  constant FlitSize  : natural := 75;-- flit size
  constant Framesize : natural := 24;-- frame size
  constant Noframes  : natural := 4; -- flit = 4 frames
  constant NoVC      : natural := 5; -- virtual circuits
  constant CBVC      : natural := 1; -- VCs entering CB
  constant Degree    : natural := 2; -- topology degree
  constant NoCC      : natural := 1; -- consumption channels
  constant Diag      : natural := 0; -- diagnostic channels
  constant Direct    : natural := 1; -- 1 = unidir.,2 = bidir.
  constant DataPaths : natural := (Direct*(CBVC*Degree))+NOCC+Diag;
  constant mxwidth   : natural := 14;-- mx path(bits)
  constant FifoBlocks: natural := 16; -- Fifo depth

  constant clear : std_logic_vector := "000000";
  constant xVC1  : std_logic_vector := "000001";
```

```
constant xVC0  : std_logic_vector := "000010";
constant yVC1  : std_logic_vector := "000100";
constant yVC0  : std_logic_vector := "001000";
constant p     : std_logic_vector := "010000";
constant d     : std_logic_vector := "100000";

constant VC0  : std_logic_vector := "001";
constant VC1  : std_logic_vector := "010";
constant VC2  : std_logic_vector := "011";
constant VC3  : std_logic_vector := "100";
constant VC4  : std_logic_vector := "101";

constant VC0_5  : std_logic_vector := "00001";
constant VC1_5  : std_logic_vector := "00010";
constant VC2_5  : std_logic_vector := "00100";
constant VC3_5  : std_logic_vector := "01000";
constant VC4_5  : std_logic_vector := "10000";

constant head : std_logic_vector := "001";
constant data : std_logic_vector := "010";
constant tail : std_logic_vector := "100";
constant token : std_logic_vector := "111";

constant match : std_logic_vector := "11";
constant plus : std_logic_vector := "01";
constant minus : std_logic_vector:= "10";

type arb_sel is record        -- AR requests (OP)
  Sel0 : std_logic_vector(DataPaths-1 downto 0);
  Sel1 : std_logic_vector(DataPaths-1 downto 0);
  Sel2 : std_logic_vector(DataPaths-1 downto 0);
  Sel3 : std_logic_vector(DataPaths-1 downto 0);
  Sel4 : std_logic_vector(DataPaths-1 downto 0);
  Sel5 : std_logic_vector(DataPaths-1 downto 0);
end record;

type arb_pr is record        -- AR request priority
  Sel0 : std_logic_vector(2 downto 0);
  Sel1 : std_logic_vector(2 downto 0);
  Sel2 : std_logic_vector(2 downto 0);
  Sel3 : std_logic_vector(2 downto 0);
  Sel4 : std_logic_vector(2 downto 0);
  Sel5 : std_logic_vector(2 downto 0);
```

```
  end record;

  type CBpaths is record  -- CB data (includes new VCI(3))
    x1 : std_logic_vector(Flitsize+2 downto 0);
    x0 : std_logic_vector(Flitsize+2 downto 0);
    y1 : std_logic_vector(Flitsize+2 downto 0);
    y0 : std_logic_vector(Flitsize+2 downto 0);
    p  : std_logic_vector(Flitsize+2 downto 0);
    d  : std_logic_vector(Flitsize+2 downto 0);
  end record;

  type MXpaths is record  -- MX paths
    p0 : std_logic_vector(mxwidth-1 downto 0);
    p1 : std_logic_vector(mxwidth-1 downto 0);
    p2 : std_logic_vector(mxwidth-1 downto 0);
    p3 : std_logic_vector(mxwidth-1 downto 0);
    p4 : std_logic_vector(mxwidth-1 downto 0);
  end record;

  type fifo_type is record -- Fifo structure( 16 deep)
    Data0  : std_logic_vector(FlitSize-1 downto 0);
    Data1  : std_logic_vector(FlitSize-1 downto 0);
    Data2  : std_logic_vector(FlitSize-1 downto 0);
    Data3  : std_logic_vector(FlitSize-1 downto 0);
    Data4  : std_logic_vector(FlitSize-1 downto 0);
    Data5  : std_logic_vector(FlitSize-1 downto 0);
    Data6  : std_logic_vector(FlitSize-1 downto 0);
    Data7  : std_logic_vector(FlitSize-1 downto 0);
    Data8  : std_logic_vector(FlitSize-1 downto 0);
    Data9  : std_logic_vector(FlitSize-1 downto 0);
    Data10 : std_logic_vector(FlitSize-1 downto 0);
    Data11 : std_logic_vector(FlitSize-1 downto 0);
    Data12 : std_logic_vector(FlitSize-1 downto 0);
    Data13 : std_logic_vector(FlitSize-1 downto 0);
    Data14 : std_logic_vector(FlitSize-1 downto 0);
    Data15 : std_logic_vector(FlitSize-1 downto 0);
  end record;

function check_par
   -- parity checker
   (parity_bit: std_logic; byte:std_logic_vector(7 downto 0))
     return std_logic;
```

```
function make_par
   -- parity creater
   (byte:std_logic_vector(7 downto 0))
     return std_logic;

function sub
  -- subtractor
  (x :  std_logic_vector;
   y :  std_logic_vector) return std_logic_vector;

procedure compare -- from logiblock source
   -- comparator
   (a: IN std_logic_vector;b: IN std_logic_vector;
   aeqb: OUT std_logic;agtb: OUT std_logic;altb: OUT std_logic);

function outputok
  -- checks reserved op list,
  -- OK if same requestor or OP not reserved
  (in_ch, reserved_ip, reqd_op,reserved_op :
       std_logic_vector(DataPaths-1 downto 0))return boolean;

function update_res_OP
  -- add new output to reserved OP list on Grant
  -- remove output from reserved list on Tail
  (reserved, out_channel, Tail, Grant:
       std_logic_vector(DataPaths-1 downto 0))return std_logic_vector;

function update_res_IP
  -- on grant add new input to IP list holding reserved channels
  -- remove input from IP list on Tail
   (reserved, Tail, channel: std_logic_vector(DataPaths-1 downto 0))
        return std_logic_vector;

end Design_issues;


package body Design_issues is

function check_par (parity_bit: std_logic; byte
             : std_logic_vector(7 downto 0)) return std_logic is
        --parity checker, returns 0 of OK
  variable temp,temp2 : std_logic;
begin
```

```
  temp := byte(7) xor byte(6) xor byte(5) xor byte(4) xor byte(3)
                             xor byte (2) xor byte(1) xor byte(0);
  if temp = parity_bit then
    temp2 := '0';
  else
    temp2 := '1';
  end if;
  return temp2;
end;

function make_par(byte:std_logic_vector(7 downto 0))
                                     return std_logic is
  --returns parity
  variable temp: std_logic;
  begin
    temp := byte(7) xor byte(6) xor byte(5) xor byte(4) xor byte(3)
                               xor byte (2) xor byte(1) xor byte(0);
    return temp;
 end;

PROCEDURE compare
   (a: IN std_logic_vector;
    b: IN std_logic_vector;
    aeqb: OUT std_logic;
    agtb: OUT std_logic;
    altb: OUT std_logic) IS
  VARIABLE eq, gt, lt: std_logic;
  BEGIN
    eq := '1';
    gt := '0';
    lt := '0';
    FOR i in a'LEFT DOWNTO a'RIGHT LOOP
      IF (a(i) /= b(i) AND eq = '1') THEN
eq := '0';
IF (a(i) > b(i)) THEN
  gt := '1';
        ELSE
  lt := '1';
        END IF;
     END IF;
    END LOOP;
  aeqb := eq;
  altb := lt;
```

```
  agtb := gt;
  RETURN;
  END compare;

  function sub
  (x :  std_logic_vector;
   y :  std_logic_vector) return std_logic_vector is
  type i_type is range 0 to y'length;
  variable i : i_type;
  variable vx : std_logic_vector(x'length-1 downto 0) := x;
  variable vy : std_logic_vector(y'length-1 downto 0) := y;
  variable carry : std_logic_vector(x'length downto 0);
  variable temp : std_logic_vector(x'length-1 downto 0);
  begin
    assert x'length = y'length
      report "unequal vector lengths"
      severity warning;
    if vx'length = vy'length then
      carry(carry'low) := '0';
      build : for i in vx'range loop
        temp(i) := vx(i) xor (vy(i) xor carry(i));
        carry(i+1) := ((not vx(i)) and vy(i)) or
            ((not vx(i)) and carry(i)) or (vy(i) and carry(i));
      end loop build;
    end if;
    return temp;
end sub;

function outputok (in_ch, reserved_ip, reqd_op, reserved_op :
       std_logic_vector(DataPaths-1 downto 0)) return boolean is
   -- ok to allocate output if either output is free
   --or requestor has already reserved output
  variable tempOK, zerovector : std_logic_vector(DataPaths-1 downto 0);
  variable temp2 : boolean;
begin
   --tempOK := (others => '0');
  zerovector := (others => '0');
  for i in 0 to Reqd_op'length-1 loop
    if Reqd_op(i) = '1' and reserved_op(i) = '0'then   -- output free
      tempOK(i) :='1';
    elsif Reqd_op(i) = '1' and reserved_op(i) = '1' then
      for j in 0 to in_ch'length-1 loop
        if in_ch(j) = '1' and reserved_ip(j) = '1' then
```

```
        --output reserved of requesting channel
        tempOK(i) := '1';
      else
        tempOK(i) := '0';
      end if;
      exit when tempOK(i) = '1';
    end loop;
  else
    tempOK(i) := '0';
  end if;
end loop;
if tempOK = zerovector then
  temp2 := false;
else
  temp2 := true;
end if;
return temp2;
end;

function update_res_OP(reserved, out_channel, Tail, Grant:
     std_logic_vector(DataPaths-1 downto 0)) return std_logic_vector is
  variable temp: std_logic_vector(DataPaths-1 downto 0);
  variable clear : std_logic;
  begin
    temp := reserved;
    clear := '0';
    for j in 0 to Grant'length-1 loop
    if Grant(j) = '1' and Tail(j) = '1' then
      clear := '1';
    end if;
    end loop;
    for i in 0 to out_channel'length-1 loop
      -- add reserved output on Req
      if out_channel(i) = '1' and clear = '0' then
        temp(i) := '1';
      elsif out_channel(i) = '1' and clear = '1' then
        temp(i) := '0';
      else
        temp(i) := temp(i);
      end if;
    end loop;
    return temp;
  end;
```

```
function update_res_IP(reserved, Tail, channel:
      std_logic_vector(DataPaths-1 downto 0)) return std_logic_vector is
  variable temp : std_logic_vector(DataPaths-1 downto 0);
begin
  temp := reserved;
  Case channel is
    when xVC1 =>
      if Tail(0) = '1' then
        temp(0) := '0';
      else
        temp(0) := '1';
      end if;
    when xVC0 =>
      if Tail(1) = '1' then
  temp(1) := '0';
      else
        temp(1) := '1';
      end if;
    when yVC1 =>
      if Tail(2) = '1' then
        temp(2) := '0';
      else
        temp(2) := '1';
      end if;
    when yVC0 =>
      if Tail(3) = '1' then
  temp(3) := '0';
      else
        temp(3) := '1';
      end if;
    when p =>
      if Tail(4) = '1' then
  temp(4) := '0';
      else
        temp(4) := '1';
      end if;
    when others =>
      temp := temp;
    end case;
   return temp;
 end;
```

```
end Design_issues;
```

Appendix E

Netlist Conversion Program (PERL)

Convert.pl

```
# This perl conversion program converts XNF format netlists to PDL
# format netlist for inclusion in a PDL design file. Conversion is
# required for gate-level traffic analysis of a model.

open(OUTPUT, "> out") || die "can't open output file : $!\n";

# converts symbols to modules
#----------------------------

open(FILE, "$ARGV[0]") || die "can't open file: $!\n";  #input file

while (<FILE>){
  # strip control character from names
  s/</_/;
  s/>/_/;
  if (/\bSYM\b/){
    #converts first line of code
    ($name, $type, $dummy) = /^SYM, (\w+), (\w+), (\w+)(.*)$/;
    print OUTPUT  "module $type = M$name \n";
    print OUTPUT "   ports\n";
    print OUTPUT "     inputs = {"
    }
  if (/\bPIN\b/){
```

```
    #converts input-pin code
    #tests for inv signal
    $test1 = /^PIN, (\w+), (\w), (\w+), ,(\w+)(.*)$/;
    ($dummy, $type, $name ) = /^PIN, (\w+), (\w), (\w+)(.*)$/;
    # tests for first I pin
    if (($dummy eq "IO") || ($dummy eq "I"))  {
      if ($test1){
$invvar = $name;
        print OUTPUT "INV$name = I$name.INV$name";
        }
      else{                        # non-inv signal
          print OUTPUT "$name = M$name.$name";
          }
      }
      if (($dummy ne "IO") && ($type eq "I"))  {  # other I pins
            print OUTPUT ", $name = M$name.$name";
}
      #convert output-pin code
      if ($type eq "O")  {
print OUTPUT "};\n";
print OUTPUT "     output = $name;\n";
print OUTPUT "end module;\n\n\n";
      }
    }
}
close(FILE);

open(OUTPUT, ">> out") || die "can't open output file : $!\n";

open(FILE2, "$ARGV[1]") || die "can't open file: $!\n";  #input file

while (<FILE2>) {
  # strip control character from names
  s/</_/;
  s/>/_/;
  if (/\bPIN\b/){
    # set up variables
    ($dummy, $type, $name ) = /^PIN, (\w+), (\w), (\w+)(.*)$/;
    # INV module required
    if (/INV/) {
      print OUTPUT  "module INV = I$name$rest \n";
      print OUTPUT "   ports\n";
      print OUTPUT "     input = $name = M$name.$name; \n";
```

```
      print OUTPUT "     output = INV; \n";
      print OUTPUT "end module; \n\n\n";
    }
  }
}
close(FILE2);

# collect port names (signals) from input file
#---------------------------------------------

open(FILE, "$ARGV[0]") || die "can't open file: $!\n";
  while (<FILE>){
    s/</_/;
    s/>/_/;
    if (/\bSIG\b/){
      ($sig) = /^SIG, (\w+),(.*)$/;
      push(@signals, "$sig");
     }
  }
close(FILE);

#collect all the submodule names from output file
#------------------------------------------------
open(FILE2, "$ARGV[1]") || die "can't open file: $!\n";
  while (<FILE2>) {
    if (/^module/) {
      ($type, $dummy, $name) = /^module (\w+) (\S) (\w+)(.*)$/;
      push(@records, "$type:$name");
    }
  }
  print OUTPUT "module  INSERT TYPE???   =  \n\n";
  print OUTPUT "  modules \n\n";
  @records = sort (@records);
  foreach $record (@records) {
    ($t, $n) = split(/:/, $record);
    if (!$newtype) {
      # first type allocation
      $newtype = $t;
      print OUTPUT "    \U$newtype\Es = {";  #capitalise#
      print OUTPUT "$n";
    }
    elsif ($t eq $newtype) {
      print OUTPUT ", $n";
```

```
    }
    else {
      # change of type
      $newtype = $t;
      print OUTPUT "};\n\n    \U$newtype\Es = {";
      print OUTPUT "$n";
    }
  }
  print OUTPUT "};\n\n";

# outputs input(output) ports
#---------------------------------
  $newsignal = pop(@signals);
  while ($newsignal) {
    open(FILE2, "$ARGV[1]") || die "can't open file: $!\n";
    @foo = grep(/\b$newsignal\b/, <FILE2>) ;
    $find = pop(@foo);
    if (grep(/input/, $find)) {
      push(@inputs, "$newsignal");
    }
    elsif (grep(/output/, $find)){
     push(@outputs, "$newsignal");}
    $newsignal = pop(@signals);
 }
close(FILE2);
  print OUTPUT "\n   ports\n\n";
  print OUTPUT "     inputs = {";
  $ip = pop(@inputs);
  print OUTPUT "$ip";
  while ($ip) {
    $ip = pop(@inputs);
    if ($ip) {                   #avoids extra , print
      print OUTPUT ", $ip";
     }
  }
  print OUTPUT "};\n\n    outputs = {";
  $op = pop(@outputs);
  print OUTPUT "$op";
  while ($op) {
    $op = pop(@outputs);
    if ($op) {                   #avoids extra , print
      print OUTPUT ", $op";
      }
```

```
}
print OUTPUT "}; \n end module; \n\n\n";
```

Appendix F

Example PDL Data File

Net_change.data

```
-- example of TRC data in Torus network
-- fixed packet size, varying network size

TRC, sci_cost, 9;
TRC, sm_cost, 5;
TRC, b_cost, 4;
TRC, ac_cost, 5;
TRC, rd_cost, 43;
TRC, ad_cost, 4;
TRC, cb_cost, 71;
TRC, ar_cost, 43;
TRC, op_cost, 5;
TRC, sco_cost, 13;
TRC, cc_cost, 88;

TRC, sci_delay, 7.76;
TRC, sm_delay, 13.70;
TRC, b_delay, 5.45;
TRC, ac_delay, 7.09;
TRC, rd_delay, 33.93;
TRC, ad_delay, 6.54;
TRC, cb_delay, 10.12;
TRC, ar_delay, 12.95;
TRC, op_delay, 4.85;
```

```
TRC, sco_delay, 9.42;
TRC, cc_delay, 19.55;

VCx, fo_delay, 19.55;
VCy, fo_delay, 19.55;
VCx, mx_delay, 4.90;
VCy, mx_delay, 4.90;
VCx, ar2_delay, 4.92;
VCy, ar2_delay, 4.92;

VCx, fo_cost, 88;
VCx, mx_cost, 5;
VCx, ar2_cost, 1;
VCy, fo_cost, 88;
VCy, mx_cost, 5;
VCy, ar2_cost, 1;

top1, chan_wid, 8.0;
top1, flit_size, 8.0;
top1, pack_size, 128.0;
top1, net_diam, 6.0, 14.0, 30.0, 62.0, 94.0, 126.0;
top1, head_size, 2.0;
top1, nodes, 16, 64, 256, 1024, 2304, 4096;

carry : top1'chan_wid, top1'flit_size, top1'pack_size, top1'head_size;

top1, trecv,   1.18;
top1, ext_phy, 0.0;
top1, tsend,   5.06;

top1, sw_tech, WH;

carry : top1'trecv, top1'ext_phy, top1'tsend, top1'sw_tech;
```

Bibliography

[Aea86] D.P. Agarwal et al. B-HIVE: A Heterogenous, Interconnected, Versatile and Expandable Multicomputer System. *Computer Architecture News*, Jan 1986.

[Aea94] J.D. Allen et al. Adriane* - An Adaptive Router for Fault-tolerant Multicomputers. In *Proceedings of 21st Annual Intl. Symposium on Computer Architecture*, pp 278–288, 1994.

[AG89] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Co., 1989.

[Aga91] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Trans. on Parallel and Distributed Systems*, 2(4), Oct. 1991.

[AP92] S. Abraham and K. Padmanabhan. Performance of Multicomputer Networks under Pin-Out Constraints. *Journal of Parallel and Distrbuted Systems*, pp 237–248, Aug 1992.

[AS88] W.C. Athas and C.L. Seitz. Multicomputers:Message Passing Concurrent Computers. *IEEE Computer*, 21(8) pp 9–24, Aug. 1988.

[Bar68] G.H. Barnes et al. The ILLIAC IV Computer. *IEEE Transactions on Computers*, pp 746–757, Aug 1968.

[BBN87] BBN Advanced Computers, Inc. *Butterfly Products Overview*, October 1987.

[Bea95] M.A. Blumrich et al. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, pp 21–28, Feb. 1995.

[BG88] K. Bratbergsengen and T. Gjelsvik. The Development of the Cross8 and HC16-186 Parallel Database Computers. In *IWDM'89*, pp 359–372, Nov 1988.

[Bhu87] L.N. Bhuyan. Analysis of Interconnection Networks with Different Arbiter Designs. *Journal of Parrallel and Distributed Computing*, (4) pp 384–403, 1987.

[Bol] K. Bolding et al. The Chaos Routing Chip: Design and Implementation of an Adaptive Router. In *Proceedings of VLSI'93, IFIP, 1993*.

[Bol93] K. Bolding. *Chaotic Routing - Design and Implementation of an Adaptive Multicomputer Network Router*. PhD thesis, University of Washington, 1993.

[BP93] S. Balakrishnan and D.K. Panda. Impact of Multiple Consumption Channels on Wormhole Routed k-ary n-cube Networks. In *Proc. of the Intern. Parallel Processing Symposium '93*, pp 163–167, 1993.

[BWF91] D. Badouel, C. Wuthrich, and E.L. Fiume. Routing Strategies and Message Contention on Low-Dimensional Interconnection Networks. 1991.

[CF93] A.A. Chien and W. Feng. Techniques for Efficient Execution of Fine-Grained Concurrent Programs. In *Proc. of the Yale Workshop on Languages and Compilers for Parallel Computing*, 1993.

[Chi93] A.A. Chien. A Cost and Speed Model for K-Ary N-Cube Wormhole Routers. In *Proc. of Hot Interconnects '93*, Aug. 1993.

[CK92] A.A. Chien and J.H. Kim. Planar-Adaptive Routing: Low Cost Adaptive Networks for Multiprocessors. In *19th Intern. Sympos. On Computer Architecture*, 1992.

[Dal90a] W.J. Dally. *Network and Processor Architecture for Message-Driven Computers*. Morgan Kaufmann Publishers, Inc., 1990.

[Dal90b] W.J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[Dal92] W.J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2) pp 194–205, March 1992.

[Dan91] S.P. Dandamudi. *Hierarchical Hypercube Multicomputer interconnection Networks*. Ellis Horwood, 1991.

[DD94a] W.J. Dally and L.R. Dennison. The Reliable Router: A Reliable and High Performance Communication Substrate for Parallel Computers. In *Proceedings of the 1st International Parallel Computer Routing and Communication Workshop*, 1994.

[DD94b] W.J. Dally and L.R. Dennison et al. Architecture and Implementation of the Reliable Router. In *Proceedings of the Hot Interconnects 11*, Aug. 1994.

[Dea87] W.J. Dally et al. Deadlock-Free Message Routing in Multicomputer Interconnection Networks. *IEEE Trans. Comp.*, C36(5), 1987.

[Dea89] W.J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pp 1147–1153, August 1989.

[Dea93] L.R. Dennison and et al. High Performance Bidirectional Signalling in VLSI Systems. In *Research on Integrated Systems:Proceedings of the 1993 Symposium*, 1993.

[DS86] W.J. Dally and C.L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1 pp 187–196, Jan 1986.

[DS87] W.J. Dally and P.Song. Design of a Self-Timed VLSI Multicomputer Communications Controller. *Proceedings of the Intern. Conf. on Computer Design*, (IEEE CS Press, 2473) pp 230–234, 1987.

[DU97] H. Dybdahl and I. Uthus. Simulation of the BSP Model on Different Computer Architectures. Technical report, Diploma Thesis, Department of Computing and Information Systems, Norwegian University of Science and Technology, 1997.

[Dua91a] J.Duato. Deadlock-Free Adaptive Routing Algorithms for Multicomputers: Evaluation of a New Algorithm. *Proc. 3rd IEEE Symp. Parallel and Distrib. Processing*, pp 840–847, 1991.

[Dua91b] J. Duato. On the Design of Deadlock-Free Adaptive Routing Algorithms for Multicomputers: Design Methodologies. In *Parallel Architectures and Languages European Proceedings*, pp 390–405, June 1991.

[Dun90] R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, pp 5–15, Feb. 1990.

[EK93] R. Esser and R. Knecht. Intel Paragon XP/S - Architecture and Software Environment. In *Proceedings of Supercomputer '93*, 1993.

[FK85] A.L. Fisher and H.T. Kung. Synchronizing Large VLSI Processor Arrays. *IEEE Trans. Comp.*, C-34(8) pp 734–740, 1985.

[Fly66] M.J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54 pp 1901–1909, 1966.

[FT95] A. Flavell and Y. Takahashi. Tokkyu: A High Performance, Randomizing, Adaptive Message Router with Packet Expressway. *IEEE Trans. Inf. & Syst.*, pp 1248–1260, Oct 1995.

[Gel81] D. Gelernter. A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks. *IEEE Transactions on Computers*, c-30(10) pp 709–715, Oct. 1981.

[GN92] C.G. Glass and L.M. Ni. The Turn Model for Adaptive Routing. *Proceed. of the Intern. Sympos. on Computer Architecture*, pp 278–287, 1992.

[Gun81] K.D. Gunther. Prevention of Deadlocks in Packet-Switched Data Transport Systwms. *IEEE Transactions on computers*, Com-29 pp 512–524, April 1981.

[GY93] P.T. Gaughan and S. Yalamanchili. A Family of Fault-tolerant Routing Protocols for Direct Multiprocessor Networks. Technical report, EE Dept.,Georgia Institute of Technology, 1993.

[Had97] P.C. Haddow. A Generalisaton of Router Chip Design. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pp 307–313, Jan. 1997.

[Hal89] F. Halsall. *Data Communications, Computer Networks and OSI.* Addison-Wesley, 2nd edition, 1989.

[HB87] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill International, 3rd edition, 1987.

[HB90a] J. Hsu and P. Banarjee. Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp 260–269, May 1990.

[HB90b] J.M. Hsu and P. Banerjee. Hardware Support for Message Routing in a Distributed Memory Multicomputer. In *1990 International Conference on Parallel Processing.* University of Illinois at Urbana-Champaign, 1990.

[Hil97] M.D. Hill et al. BSPlib: The BSP Programming Library. Technical report, PRG-TR-29-97, Oxford University Computing Laboratory, 1997.

[HN97] P.C. Haddow and L. Natvig. A Case Study in Multilevel Architecture Modelling using HDLs. In *4th Asia-Pacific Conference on Hardware Description Languages (APCHDL'97)*, pp 24–31, 1997.

[HTK89] K. Hwang, P. Tseng, and D. Kim. An Orthogonal Multiprocessor for Parallel Scientific Computations. *IEEE Transactions on Computers*, pp 47–60, Jan 1989.

[Hwa93] K. Hwang. *Advanced Computer Archiecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[Int91] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[JMY89] C.R. Jesshope, P.R. Miller, and J.T. Yantchev. High Performance Communications in Processor Networks. *Proc. 16th Annu. Int. Symp. Computer Acrhitecture*, pp 150–157, May. 1989.

[KA95] V. Karamcheti and A.A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of ISCA '95*, June 1995.

[Kar86] M.J. Karol et al. Input Vs. Output Queueing on a Space-Division Packet Switch. *IEEE*, pp 659–665, 86.

[KC93] J.H. Kim and A.A. Chien. The Impact of Packetization in Wormhole Routed Networks. In *Europe (PARLE) 1993*, 1993.

[KC94] V. Karamcheti and A.A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? *ASPLOS VI*, pp 51–60, Oct. 1994.

[KCA94] J. Kubiatowicz, D. Chaiken, and A. Agarwal. The Alewife CMMU: Addressing the Communications Gap. In *HOTCHIPS 94*. lab. for CS, MIT, 1994.

[Kip] S Kipnis. Priority Arbitration with Buses. Technical report, MIT.

[KK79] P. Kermani and L. Kleinrock. Virtual-Cut-Through: A New Computer Communication Technique. *Computer Networks*, 3 pp 267–286, 1979.

[KK80] S.P. Kermani and L. Kleinrock. A Trade-Off Study of Switching Systems in Computer Communication Networks. *IEEE Transactions on Computers*, 29(12) pp 1052–1060, Dec. 1980.

[KLC94] J.H. Kim, Z. Liu, and A.A. Chien. Compressionless Routing. In *Proceedings of ISCA'94*, April 1994.

[Kle76] L. Kleinrock. *Queueing Systems*, volume 2. New York: Wiley, 1976.

[KS91] S. Konstantinidou and L. Snyder. Chaos Router: Architecture and Performance. In *Proceedings of the 18th International Symposium on Computer Architecture*, pp 212–221, May 1991.

[Lea92] D. Lenoski and et al. The Stanford Dash Multiprocessor. *IEEE Computer*, pp 63–79, Mar 1992.

[LH91] D.H. Linder and J.C. Harden. An Adaptive and Fault-Tolerant Wormhole Routing Strategy for K-Ary N-Cubes. *IEEE Trans. Computers*, 40(1) pp 2–12, Jan. 1991.

[LOKM96] S. Loucif, M. Ould-Khaoua, and L.M. Mackenzie. The Express Channel Concept in Wormhole-Routed Hypermsh and k-ary n-Cubes. *Proc. 8th IEEE Symposium on Parallel and Distributd Prcessing, SPDP'96*, pp 595–602, Oct 1996.

[LV82] T. Lang and M. Valero. M-Users B-Servers Arbiter for Multiple-Buses Multiprocessors. *Microprocessors and Microprogramming*, 10 pp 11–18, 1982.

[MBR93] Q.M. Malluhi, M.A. Bayoumi, and T.R.N. Rao. On the Hierarchical Hypercube Interconnection Network. In *International Parallel Processing Symposium*, 1993.

[MOK94] R. Sotudeh M. Ould-Khoua, L.M. Mackenzie. Comparitive Evaluation of Wormhole-Routed Hypergraph and Multi-Stage Interconnection Networks. *The Computer Journal*, 39(3) pp 232–240, 1994.

[MS80] P.M. Merlin and P.J. Schweitzer. Deadlock Avoidance in Store and Forward Networks. *IEEE Transactions on communications*, COM-28 pp 345–354, Mar 1980.

[MT90] J.E. Midwinter and M.G. Taylor. Optoelectronic Interconnects in VLSI : The Reality of Optical Computing. *IEEE LCS*, pp 40–46, May 1990.

[Nat97] L. Natvig. High-Level Architectural Simulation of the Torus Routing Chip. In *Proceedings of the 6'th International Verilog HDL Conference (IVC'97)*, pp 48–55, April 1997.

[NCU90] *nCUBE, nCUBE 6400 Processor Manual.* nCUBE company, Beaverton, OR 97006, 1990.

[ND92] P.R. Nuth and W.J. Dally. The J-Machine Network. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1992.

[NH97] L. Natvig and P.C. Haddow. A Multilevel Simulation Study Linking Parallel Applications to Executable and Realisable Hardware Models. In *Norsk Informatikk Konferense, NIK'97*, pp 303–314, 1997.

[Nic90] J.R. Nickolls. The Design of the MasPar MP1: A Cost Effective Massively Parallel Computer. *IEEE Digest of Papers-Compcon*, pp 25–28, 1990.

[NLE89] L.M Ni, Y. Lan, and A.H. Esfahanian. A Vlsi Router Design for Hypercube Multiprocessors. *the VLSI journal*, 7 pp 103–125, 1989.

[NM93] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pp 62–77, Feb 1993.

[NR84] J.C.C. Nelson and M.K. Refai. Design of a Hardware Arbiter for Multi-Microprocessor Systems. *Microprocessors and Microsystems*, 8(1) pp 21–24, Jan./Feb. 1984.

[Nug88] S.F. Nugent. The IPSC-2 Direct Connect Communications Technology. In *Proc. 3rd Conf. On Hypercube Concurrent Computers and Applications*, pp 51–60, Jan. 1988.

[OKMS95] M.Ould-Khaoua, L.M. Mackenzie, and R.Sotudeh. Fully Adaptive Routing Algorithm, for Wormhole-Routed N-Dimensional Meshes. In *Euro-PAR*, 95.

[Pad] K.Padmanabhan. On the Trade-off Between Node Degree and Communication Channel Width in Shuffle-exchange Networks. In 3rd IEEE Symp. on Parallel and Distributed Processing,Dec. 1991.

[PJ95] K.C. Patel and D.N. Jayasimha. Measures of Message Latency for Wormhole Routed Networks. Technical report, Dept. of C and IS, Ohio State University, 1995.

[Plu72] W.W. Plummer. Asynchronous Arbiters. *IEEE Transactions on Computers*, 21(1) pp 37–43, Jan. 1972.

[RGC89] D. Reeves, E. Gehringer, and A. Chandiramani. Adaptive Routing and Deadlock Recovery: A Simulation Study. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, 1989.

[Sea92] C. Seitz et al. Mosaic C: An Experimental Fine-grain Multicomputer. Technical report, CA 91125, California Institute of Technology, 1992.

[Sea93] E. Spertus et al. Evaluation of Mechanisms for Fine Grain Parallel Programs in the J-Machine and the CM-5. Technical report, AI, Lab MIT and EECS, University of California, 1993.

[Sei80] C.L. Seitz. *System Timing*, chapter ??, p ?? Addison Wesley, 1980.

[Sei85a] C.L. Seitz. The Hypercube Communication Chip. Technical Report 5182:DF:85, Dept. CS, California Institute of Technology, 1985.

[Sei85b] C.L. Seitz. The Cosmic Cube. *Commun. of the ACM*, 28 pp 22–33, 1985.

[Sei88] C.L. Seitz et al. The Architecture and Programming of the Ametek Series 2010 Multicomputer. In *Proceedings of the Conference on Hypercube Computers and Concurrent Applications*, pp 33–36, 88.

[SR77] H. Sullivan and T.R. Bashkow. A Large Scale Homogeneous, Fully Distributed Parallel Machine. In *Proc. 4th Symp. Computer Arch.*, pp 105–124, Mar. 1977.

[SR91] M. Raghavacheri S. Rajaseraran. Optimised Randomized Algorithms for Multipacket and Cut-Through Routing on the Mesh. In *IEEE*, 1991.

[SS93] C.L. Seitz and Wen-King Su. A Family of Routing and Communication Chips Based on the Mosaic. In *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pp 320–337, 1993.

[Tan89] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall International, 2nd edition, 1989.

[Tau76] D.M. Taub. Contention Resolving Circuits for Computer Interrupt Systems. *Proceedings of the IEE*, 123(9) pp 845–850, Sep. 1976.

[TU79] S. Toueg and J.D. Ullman. Deadlock-Free Packet Switching Networks. In *Proc. 11th ACM Symposium on Theory of Computing*, pp 89–98, 1979.

[UP97] V. Upadhyay, J. Varavithya and M. Prasant. A Traffic-Balanced Adaptive Routing Scheme for Two-Dimensional Meshes. *IEEE Transactions on Computer*, pp 190–197, Feb 1997.

[VEea92] T. Von Eicken et al. Active Messages: a Mechanism for Integrated Communiction and Computation. In *Proceedings of the 19Th International Symposium on Computer Architecture*, May 1992.

[VMM96] B. Vemuri, R. Mandayam, and V. Meduri. Performance Modelling Using PDL. *IEEE Computer*, pp 44–53, April 1996.

[WH95] D. A. Wallach and W.C Hseith et al. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. Technical report, Lab. for CS, MIT, 1995.

[YN93] J. Yantchev and I. Nedelchev. Implementation of a Packet Switching Device as a Delay-Insensitive Circuit. Technical report, Oxford University, University of Surrey, 1993.

[ZYH94] X. Zhang, Y. Yan, and K. He. Evaluation and Measurment of Multiprocessor Latency Patterns. In *Proc. Of the 8h Int. Parallel Processing Symposium*, April 1994.

Index