Network-on-Chip

Final Report

Danny Nicholls

Supervisor: Philip Watts

Second Assessor: Benn Thomsen

March 2012

Abstract

As multi-core computing becomes mainstream and application specific consumer electronic Systems-on-Chip (SoC) technology becomes more complicated, Network-on-Chip (NoC) technology has become an important and growing field of research with analysts predicting mainstream adoption within two to five years. As design perspectives tend towards many-core, increasing core counts correspond with an increase in bandwidth demand in order to facilitate high core utilization and a critical need for a low power, scalable interconnection architecture that provides ultra-low latency communication with a high traffic throughput whilst maintaining end-to-end quality of service and accommodating a variety of transaction protocols.

A scalable interconnection network utilising 5-port NoC routers has been designed in SystemVerilog and tested by simulation using Mentor Graphics Modelsim. The router uses X,Y dimension-ordered routing and a flit based flow control technique. Correct routing of packets has been demonstrated in simulation using the router design in a 1024 core network. In this report, I will describe the router design and present results showing the effect of buffer size on performance on a 64 core 2D-Mesh network under uniformly distributed Bernoulli traffic.

DECLARATION

I have read and understood the College and Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is all my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures and computer programs.

Name: ……

Signature: ……

Date: ……

Contents

1. Introduction

1.1 TRENDS IN SYSTEM AND COMMUNICATION ARCHITECTURE EVOLUTION

Arteris identified four trends that have driven evolutions in system architecture and consequently in the required interconnection architecture in both general purpose computing chips and application specific SoCs [4]. These trends are:

1.1.1 Application convergence

Greater system integration for multiprocessor systems-on-chip (MPSoC) requires the mixing of heterogeneous traffic types in the same SoC design (Video, Communication, Computing etc.) and the sharing of resources [7].

1.1.2 Moore's Law

Increasing transistor counts may lead to many-core architectures including hundreds and even thousands of cores integrated on a single chip [8] due to the increasing power consumption and the diminishing performance returns of uniprocessor architecture. In addition to the integration of many general-purpose cores on a single chip, MPSoCs are increasing integration by incorporating many heterogeneous IP blocks in one chip [9]. These advances are key to enabling application convergence but also new approaches to computing; allowing parallel processing on a single chip with many processing cores impacts multiple areas of computing from faster web applications to the utilization of clusters of multi core computers by CERN to run high energy physics applications [1].

1.1.3 Consequences of silicon process evolutions between generations

As the amount of transistors available increases with each new technology generation, metal layers do not scale down proportionally to transistor scaling therefore the cost of gates with respect to wires from both a die area and performance perspective decreases [5].

1.1.4 Time to market pressures

Modular design of multicore chips alongside the use of synthesizable Register Transfer Level (RTL) rather than manual layout is reducing design complexity and time to market. However the choice of available implementation solutions to fit a bus architecture into a design flow that can adapt easily to changing SoC feature and process technology requirements is diminishing. With developing SoC designs, using a bus architecture can increase time to market of derivative SoC designs when changing single IP blocks in the SoC as the whole bus may need to be redesigned to accommodate the change [10].

1.2 MULTICORE GOES MAINSTREAM

The evolutionary trends in system and communication architecture have lead to an increasing commercialization of multi-core computing with Intel recently announcing that the multi-core architecture has become mainstream [1]. These chips may contain several homogenous cores for general computing or several heterogeneous cores in an application specific MPSoC.

The increase in core count to move from multi- to many-core computing from a design perspective will require a continuing evolution of the supporting infrastructure (interconnect, memory, hierarchy etc) with the interconnect tending towards a scalable, high bandwidth communication fabric. In an MPSoC containing many heterogeneous IP blocks, a standard interface between IP cores and the

interconnect using Standard Network Interface Units (NIUs) will allow cores using a variety of protocols to be connected in a plug and play fashion, increasing design isolation and reducing time to market. This evolution has prompted analysts to suggest that the NoC architecture will become mainstream within two to five years [2,3].

1.3 EVOLUTION OF INTERCONNECTION NETWORKS

Dedicated wiring can be used to connect small numbers of cores however as core count increases the amount of wiring needed to connect each core becomes prohibitive. Traditionally busses and crossbars designed for either a specific set of features relevant to a narrow target market or to support a specific processor have been implemented in virtually all SoCs with a low core count [4].

With a bus interconnect an increase in complexity due to application convergence combined with increasing core counts and variety of IP protocols corresponds with a prohibitive increase in the required power, arbitration latency due to a centralized arbiter and the required bus bandwidth with bus traffic quickly reaching saturation. As a result bus based architectures scale poorly to only a small number of cores with a complicated design procedure increasing time to market. Crossbars have been used to provide increased bandwidth with respect to busses however these also scale poorly as they require a large die area and are heavy power consumers [9].

Many bus architectures such as AMBA have been continually redesigned to address these problems at the expense of IP reuse strategies, as in many cases each successive evolution has required changes both in the bus implementation and in the bus interface [4].

1.4 NETWORK-ON-CHIP

NoC architecture is an emerging paradigm providing a scalable alternative to bus and crossbar interconnects. In a NoC scalable bandwidth with low power and die area overheads correlate sub-linearly with an increase in core count due to an efficient use of wiring and the multiplexing of heterogeneous communication traffic on the same interconnects. Packet-switched NoCs have routers at every component that connects to the network (core, cache, memory controller etc) connected to neighbours using short, local on-chip wiring thus reducing routing congestion. When using a regular topology, these links can be fixed length and built modularly reducing design complexity and consequently, easing time to market pressures.

1.5 PROJECT OBJECTIVE

Investigate the NoC architecture through the design and simulation of a scalable NoC. The performance of the network is to be measured in terms of latency and throughput.

1.6 REPORT STRUCTURE

This report is split into 7 further sections; Theoretical Design, Implementation, Performance Analysis, Results, Conclusions, References and Appendices. Section 2: ‘Theoretical Design’, gives a brief overview of the research work done and resulting design choices. Section 3: ’Implementation’, gives an overview of the network implementation. Section 4: ‘Performance Analysis’, introduces the design of the network simulator. Section 5: ‘Results’, contains example result data created using the network simulator on a 64 core, 64 bit network. Section 6: ‘Conclusions’, concludes the report and proposes further work. Section 7: ‘Appendix’, contains the source code for the SystemVerilog modules discussed in the report.

2. Theoretical Design

The project development process was broken down into several building blocks; topology, routing, flow control, router microarchitecture, implementation and testing. The work in each section was planned and a timescale given. The following gives a breakdown of the research work done and the resulting design choices [13].

2.1 TOPOLOGY

Research into different network topologies, their impact on performance and cost, and an analysis of the current state of the art technologies was carried out using research papers, journals, books and a case study of the Tilera TILE64 iMesh.

2.1.1 Research into the available network topologies

The physical layout and the connections between the cores and channels in the NoC are determined by the topology. Jerger and Peh (2009) provided an overview of the various direct, indirect and irregular topologies proposed for Network on Chip architecture. Many different topologies were proposed including rings, meshes, tori, butterflies, clos networks and fat trees [9]. A more in-depth explanation of these topologies and their variants is provided by Dally and Towles (2003) [14].

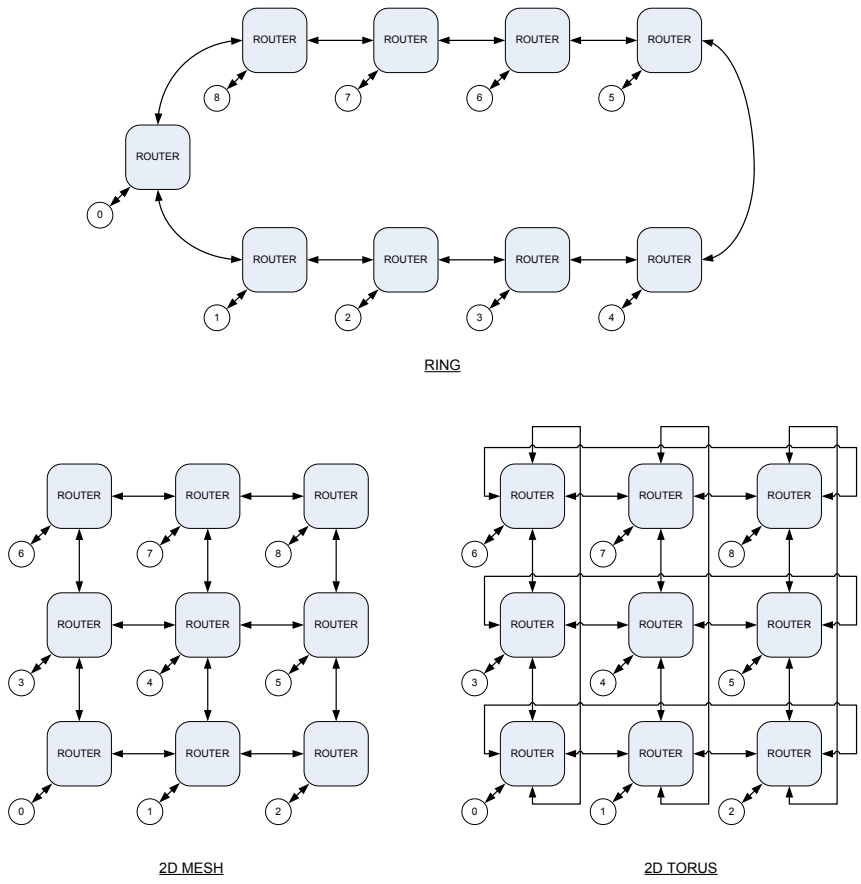

FIGURE 1: Direct Topologies. It can be seen that in the 2D Torus, the nodes at the edge of the network have the same port number (degree) as those in the centre. It is said to be edge symmetric and balances load more efficiently than the 2D Mesh, which may require higher demand for the central channels. Indirect topologies are much more complex and not as easily scalable. For Multi-Processor System-on-Chips containing heterogeneous cores an irregular, custom designed topology may be used.

2.1.2 Research into the Impact on performance and cost

The topology determines the minimum number of hops and the length of interconnect a message must traverse in order to reach its destination as well as the maximum path diversity. Implementation complexity is determined by the node degree and the ease of physically laying out the required topology on a chip [9,14]. An overview of the abstract metrics used for the comparison of topologies such as degree, hop count, maximum channel load and path diversity and how they act as proxies for network delay, area and power is provided by Jerger and Peh (2009). Although useful for estimation, these metrics do not take all implementation factors into account and do not always correlate with performance and cost in practice [9]. Pande, et al. (2005) encompassed these implementation factors developing an evaluation methodology for characterizing and contrasting different network topologies with respect to their performance and design tradeoffs, providing comparative results for a number of common topologies in terms of latency, throughput, power consumption, die area and ease of implementation [15]. Further study by Balfour and Dally (2006) developed and compared detailed area and energy models for tiled chip multiprocessors (CMP) demonstrating that the introduction of a second parallel mesh network improves both network efficiency and performance with respect to a single mesh, presenting a concentrated mesh topology that proved 24% more area efficient and 48% more energy efficient than the other networks evaluated [16].

In a survey of NoC architectures Agarwal, Iskander and Shankar (2009) summarized over 60 research papers and contributions in the NoC area and indicated the 2D-Mesh architecture as the choice of several researchers due to its efficiency in terms of latency, power consumption and ease of implementation with respect to other proposed topologies [17].

2.1.3 Tilera TILE64 iMesh Case Study

Researchers and chip manufacturers have used the various topologies for their NoC implementations [17].The Tile processors first implementation, the TILE64, and its 'iMesh' NoC developed by Tilera is described by Wentzlaff et al. (2007). The iMesh connects a 2D 8x8 grid of homogenous general-purpose cores using five physical 2D-Mesh networks. The simple mesh topology was chosen due to its ability to be mapped effectively on a 2D planar silicon substrate. The more complex toroid topology would have increased the costs in wiring congestion and length by an approximate factor of 2. The choice to implement five physical networks over virtual channels allowed leveraging of abundant on chip wiring resources to increase the available bandwidth and allow for the separation of heterogeneous traffic types thus avoiding interference. Multiple physical networks also reduce the need for buffering, reducing the die-area overhead of the network and increasing the simplification of design via replication [18].

2.1.4 State of the art

Camacho et al. (2011) propose the NearestneighboR Mesh (NR-Mesh) topology as an alternative to the 2D-Mesh. In the proposed topology, each end-node is connected to four neighboring routers in a 2D-mesh configuration resulting in comparative data showing a reduction in execution time of 23% and a 47% reduction in power consumption [19].

2.1.5 Design Choice

Although various other topologies have been proposed with increased efficiency and reduced execution time [16,18,19], a simple 2D-Mesh shall be used as it provides an easily implemented tile based, scalable topology matching the planar chip surface that performs efficiently in terms of latency and power consumption with respect to other topologies. This design will also provide the evolutionary and hierarchical design principle basis for the more complex yet more efficient topologies [17]. Each node and router combination will be numbered with an integer value and addressed with a binary according to its location in a 2D Cartesian plane as shown in figure 2.

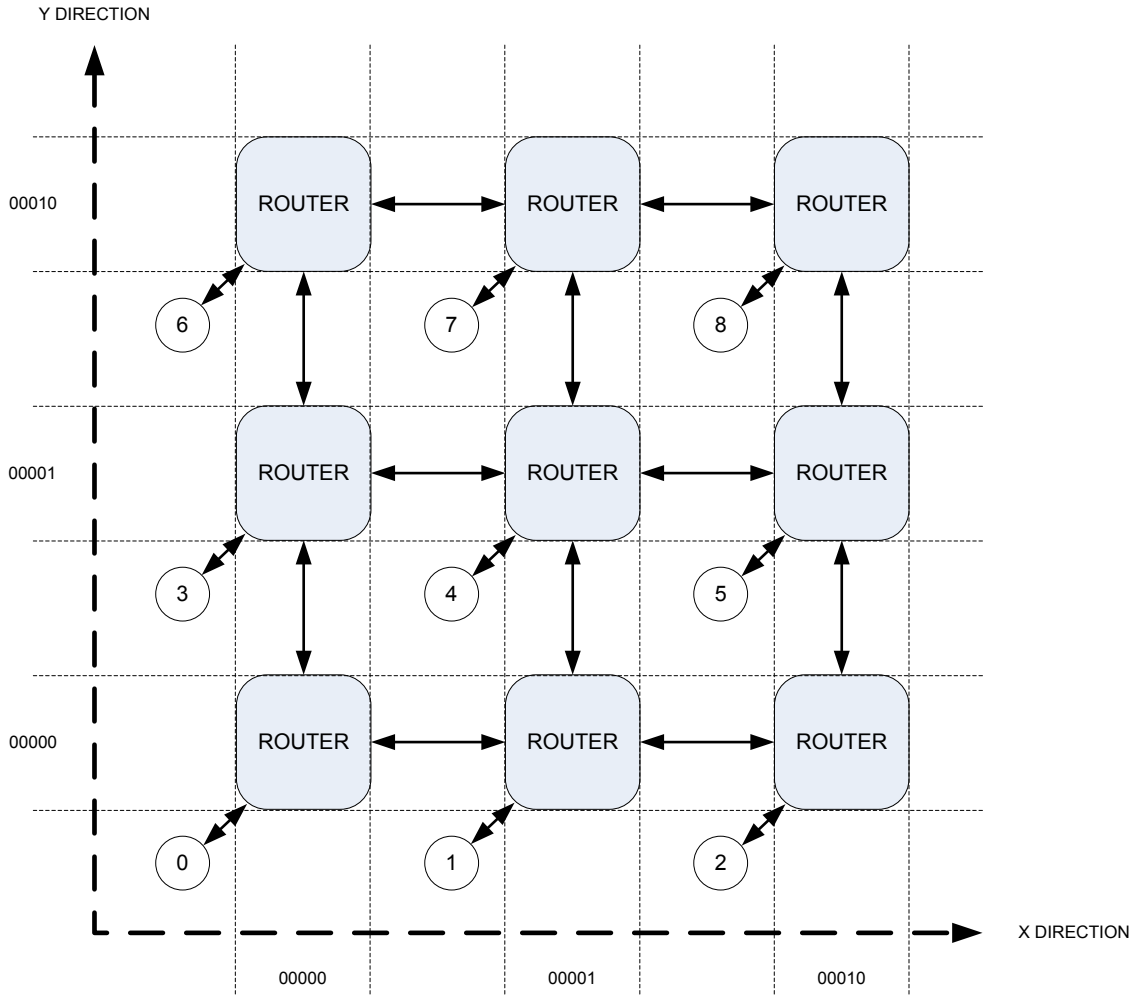

FIGURE 2: 2D Mesh numbering and addressing. Using a 5 bit binary to address each node and router combination will allow for a maximum of 32 nodes in the X direction and 32 nodes in the Y direction. This enables a total network size of 1024 nodes before the address size has to be increased.

2.2 ROUTING

Research into different routing algorithms, their impact on performance and cost and an analysis of the current state of the art technologies was carried out using research papers, journals, books and a case study of the Tilera TILE64 iMesh.

2.2.1 Research into available routing algorithms

The route traffic will take between the nodes in the 2D-Mesh is determined by the routing algorithm. Jerger and Peh (2009) provided an overview of the various deterministic, oblivious and adaptive routing algorithms proposed for the Network on Chip architecture and their possible implementation methods. The three main implementations proposed were source routing, node table based routing and combinational circuits [9].A more in-depth explanation of these routing methods, their variants and mechanics is provided by Dally and Towles (2003) [14].

2.2.2 Research into the Impact on Performance and Cost

The algorithm must be carefully designed in order to distribute traffic evenly ensuring minimal contention, reducing hotspots and preventing deadlock; thus ensuring minimal latency, high throughput and performance [9,14]. In a study including a comparison of deterministic dimension ordered routing and adaptive routing algorithms applied to mesh, torus and cube networks Neeb, Thul and Wehn (2005) showed that when the 2D mesh size exceeds 4x4, deterministic dimension ordered routing is superior to adaptive routing algorithms in terms of throughput and has a lower implementation complexity [20].However, dimension ordered routing only provides one path from source node to destination and any source of irregularity in the mesh will result in routing difficulty. Flich, Rodrigo and Duato (2008) identified four main sources of irregularity; the high integration scale reducing communication reliability, fabrication faults, chip virtualization. Logic Based Distributed Routing is proposed as a flexible routing algorithm that does not require routing tables at each node [21]. This study was extended by Rodrigo, Flich, Duato and Hummel (2008) to incorporate support for multicast and broadcast [22].

2.2.3 Tilera TILE64 iMesh Case Study

The iMesh contains five physically separate networks. These are the User Dynamic Network (UDN), I/O Dynamic Network (IDN), Static Network (STN), Memory Dynamic Network (MDN), and Tile Dynamic Network (TDN). The dynamic networks are routed using a dimension ordered algorithm and the static network utilizes circuit switching in order to preset the route [18].

2.2.4 State of the art

In their proposal for the NR-Mesh topology Camacho et al. (2011) proposed an adaptive routing algorithm that routed round network components that have been turned off. In collaboration with the topology, this algorithm maximized power efficiency by maximizing the time network components were turned off resulting in a reduction in execution time of 23% and a 47% reduction in power consumption [19].

2.2.5 Design Choice

As the topology chosen is a 2D-Mesh, a dimension ordered routing algorithm will be implemented due to the algorithm being deadlock free, power efficient and of low implementation complexity. Dimension ordered routing on a 2D-Mesh also provides the evolutionary and hierarchical design principle basis for the more complex adaptive routing algorithms such as the one proposed for the NR-Mesh [19].

Each flit shall be structured so that it contains a five bit binary destination address corresponding to the Cartesian plane location of the destination node described in section 2.1.5.

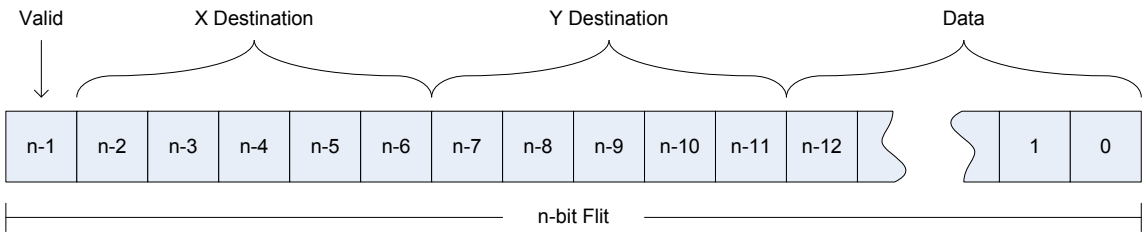

FIGURE 3: n-bit Flit structure. A single bit qualifies the data when '1'. Using a 5 bit binary to address each node and router combination will allow for a maximum of 32 nodes in the X direction and 32 nodes in the Y direction. This enables a total network size of 1024 nodes before the address size has to be increased. The rest of the flit is available to store network data.

When the flit is stored in an input buffer, a small amount of logic at each router will compare this destination address with the current location of the flit. This will create an output request detailing the required output port as shown in figure 4.

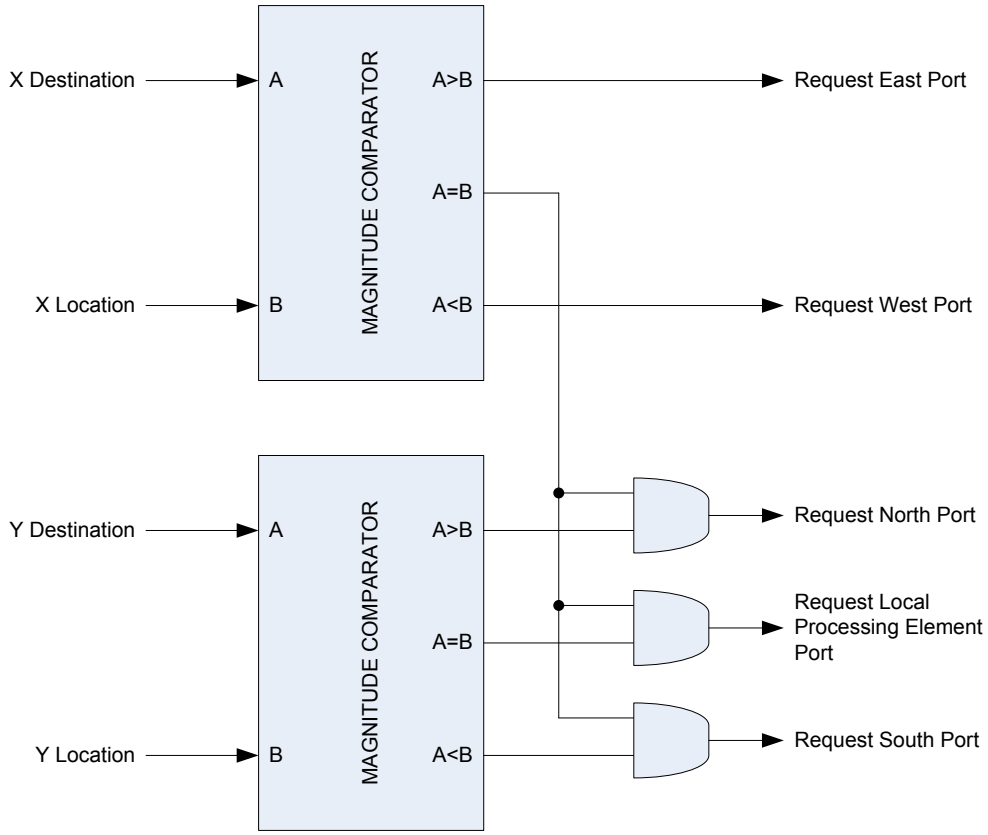

FIGURE 4: 2D Mesh X,Y Dimension Ordered routing calculation . Using two magnitude comparators, the routing logic will first request ports corresponding to movement in the X direction. Once the X Destination is equal to the X location the logic will request ports in the Y direction. Upon both the X and Y location matching the destination, the logic will request the Local Processing Element port.

2.3 FLOW CONTROL

Research into different flow control techniques, their impact on performance and cost was carried out using research papers, journals, books and a case study of the Tilera TILE64 iMesh.

2.3.1 Research into available flow control techniques

The flow control protocol determines how shared resources are allocated to messages as they traverse the network using network buffers and links according to the granularity of the data. Jerger and Peh (2009) provided an overview of the various message-based, packet-based switching techniques proposed for Network on Chip architecture and their possible implementation methods, buffering and virtual channel requirements etc. The main switching techniques proposed were circuit switching, wormhole flit-based packet switching, store and forward message-based packet switching and cut through message-based packet switching [9]. These techniques can be broadly split into either buffered or bufferless flow control, bufferless flow control being reserved for circuit switched networks. In a survey of NoC architectures Agarwal, Iskander and Shankar (2009) summarized over 60 research papers and contributions in the NoC area identifying five types of buffered flow control techniques; credit based, hand shaking, ACK/NACK, STALL/GO and T-Error [17].

2.3.2 Research into the Impact on Performance and Cost

The protocol will be instrumental in determining network energy and power consumption. Currently wormhole flow control is the dominant flow control technique used in NoC proposals and chips due to the smaller buffer requirements with respect to other techniques, buffers consuming power and die area [26].Dally and Towles (2003) explain the head-of-line blocking problems associated with wormhole flow control and propose the use of virtual channels in order to alleviate this. As the NoC flow control mechanisms increase, the die area and power consumption will also increase [9,14].

2.3.3 Tilera TILE64 iMesh Case Study

Of the iMesh's five physically separate networks, the four dynamic networks (UDN, IDN, MDN, TDN) utilize wormhole flow control without virtual channels and the static network (STN) uses circuit switching configuring the routing decisions statically at each switch point. Wentzlaff et al. (2007) explain how the choice to use wormhole routing minimizes buffering requirements to simple three entry credit controlled 'first in first out' (FIFO) buffers. The choice to implement five physical networks over virtual channels allowed leveraging of abundant on chip wiring resources to increase the available bandwidth and allow for the separation of heterogeneous traffic types removing the need for virtual channels, reducing the die-area overhead of the network and increasing the simplification of design via replication [18].

2.3.4 Design Choice

Wormhole flow control without virtual channels shall be implemented using fixed length STALL/GO controlled FIFO buffers. This will provide the evolutionary and hierarchical design principle basis for more complex routers and networks.

2.4 ROUTER MICROARCHITECTURE

Research into different microarchitectures, their impact on performance and cost and an analysis of the current state of the art technologies was carried out using research papers, books, journals and a case study of the Tilera TILE64 iMesh.

2.3.1 Research into available microarchitectures

The router microarchitecture determines the components to be used and their respective layout on the chip. An overview of router microarchitectures covering router pipeline, buffer organization, switch design, allocators and arbiters is presented by Jerger and Peh (2009) [9].A more in-depth explanation of these architectures, their variants and mechanics is provided by Dally and Towles (2003) [14].

2.3.2 Research into the Impact on Performance and Cost

Simple input-queued switch architectures may suffer from head of line blocking, a significant contributor to congestion in NoCs. Karol, Hluchyj and Morgan (1987) compared input versus output queuing and showed that under uniformly distributed, fixed packet length, on-off Bernoulli traffic, throughput was limited to 58.6% by the saturation of an input queued switch [23]. Dally and Towles (2003) propose the use of virtual channels to enable higher throughput [14]. As latency and throughput demands of the NoC are raised, die area and power consumption will increase [9,14]. The router microarchitecture has a significant impact on the power consumption of routers, a major challenge facing researchers. Over 28% of the total power requirement for the Intel TeraFLOPS processor is dissipated by the interconnection network despite a design target energy budget of 10% [24]. Similarly, approximately 40% of the total power requirement for the MIT RAW chip is dissipated by the interconnection network [25].

2.3.3 Tilera TILE64 iMesh Case Study

The iMesh uses wormhole routing without virtual channels and as such requires only one small input buffer queue at each of the five ports. These are provided using 3 entry FIFO buffers. The router uses a single stage pipeline when the message is to travel straight, with an additional route calculation stage when a turn is required [18].

2.3.4 State of the art

The input-queued switch architecture forms the basis for most inexpensive low-end routers. As designs push towards higher clock speeds and lower power budgets, different architectures are being proposed in order to meet these requirements. Wang and Jao (2011) propose a partially buffered crossbar router. The crossbar of the router contains partial buffers at each output port of the crossbar [26]. Zhang, Morris and Kodi (2011) propose a dual crossbar design called DXbar providing a 20% performance increase in terms of throughput, latency and power efficiency [27].

2.3.5 Design Choice

Figure 5 shows the chosen microarchitecture of the router. The five ports correspond to the north, south, east and west neighbouring routers and the local processing element (PE). The input buffering is organized as single fixed length FIFO queues with no virtual channels similar to that used in the Tilera TILE64 iMesh. There is no output buffering.

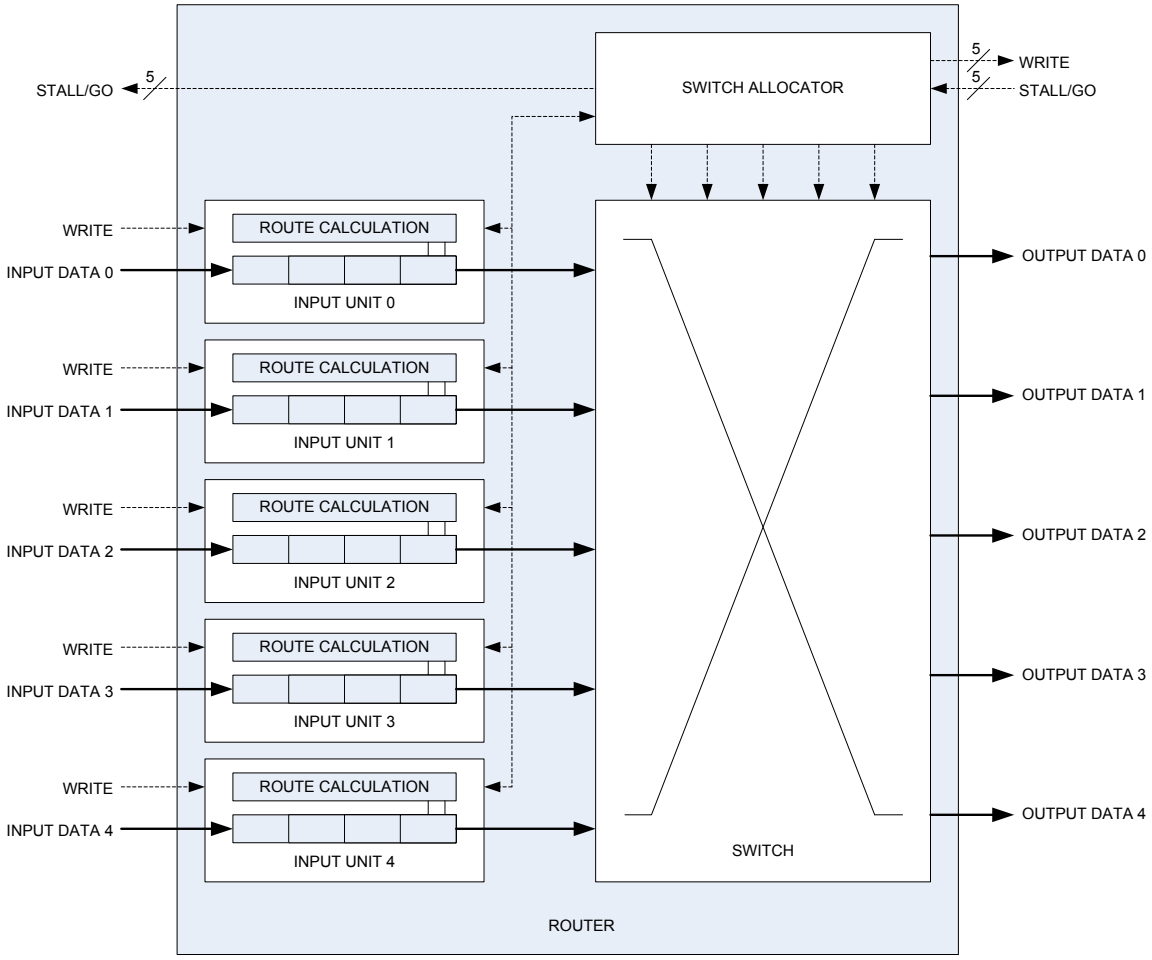

FIGURE 5: STALL/GO Base, Wormhole Routed, FIFO Input Buffered, Five Port Router for a 2D-Mesh or Torus network. The data path is shown in solid lines whereas the control path is shown using a broken line. The routing calculation in a Torus network will differ to that of the routing calculation in a Mesh network in order to take advantage of the edge symmetry.

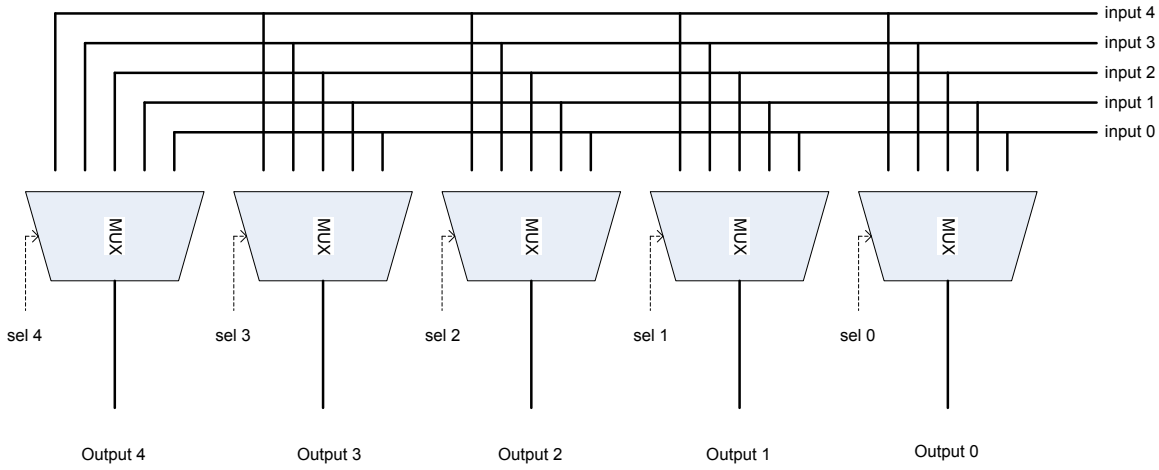

FIGURE 6: Simple 5 input, 5 output cross bar switch.

Incoming flits are sampled at each input on each clock cycle provided the corresponding 'write' signal is asserted. When flits are written to an input queue, route calculation combinational logic compares the bits in the X and Y destination field in order to generate an output port request. The logic ensures that the packet will first traverse in the X direction and then in the Y direction. The switch is a simple five input, five output crossbar as shown in figure 6.

Only one flit per cycle can traverse the switch to each output port. In order to ensure packets are not dropped, output port requests are sent to the switch allocator which allocates access to the multiple output ports to the multiple input units one at a time. This is done through the use of multiple variable priority iterative arbiters utilising a round-robin priority input.

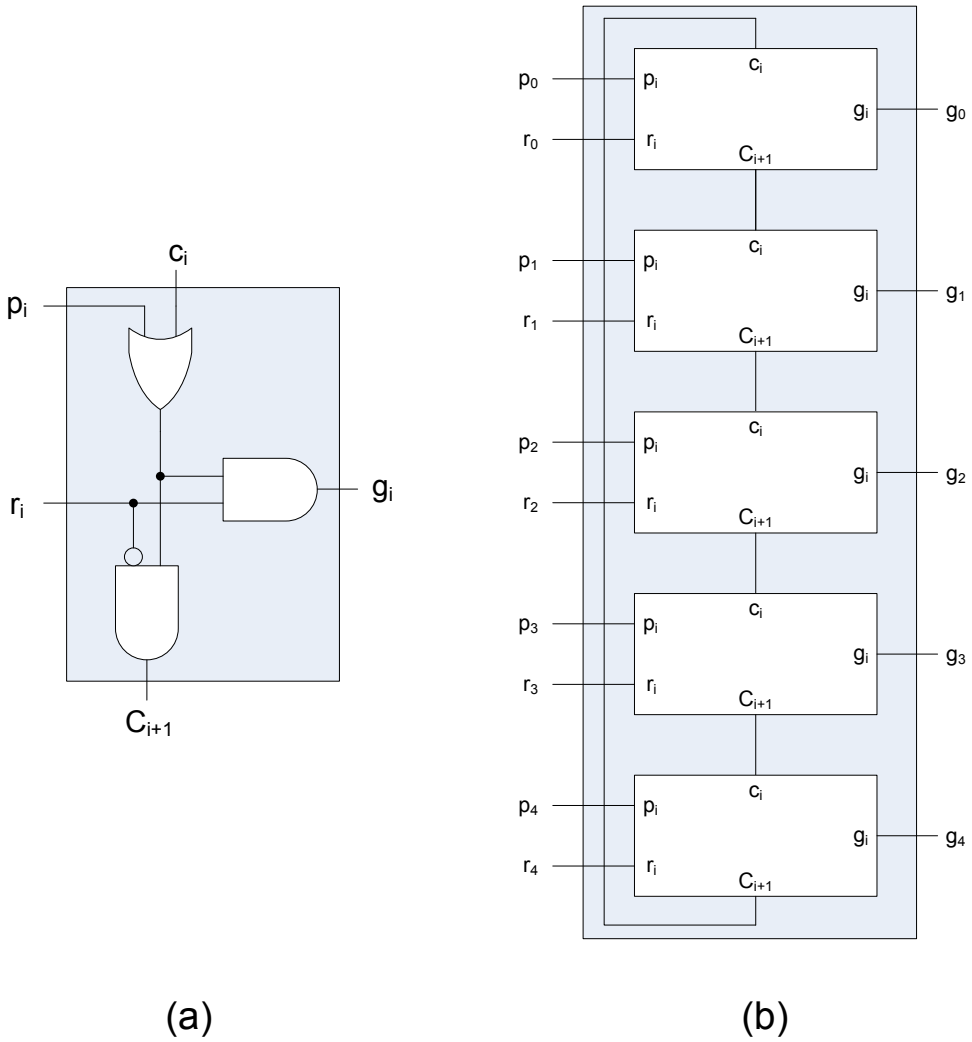

FIGURE 7: Variable Priority Iterative Arbiter. A one-hot signal 'p' is used to select the high-priority request input. From this point the priority decreases cyclically around the carry chain. When using round-robin priority generation, the request last served will have lowest priority at the next round of arbitration. (a) A one-bit slice of a variable priority arbiter. (b) A five-bit variable priority arbiter composed of five such bit slices [REF]

Each output has a corresponding 5 bit arbiter. The arbiter assigns the output to one of the requesting inputs. The five arbiters together constitute the switch allocator as shown in figure 8.

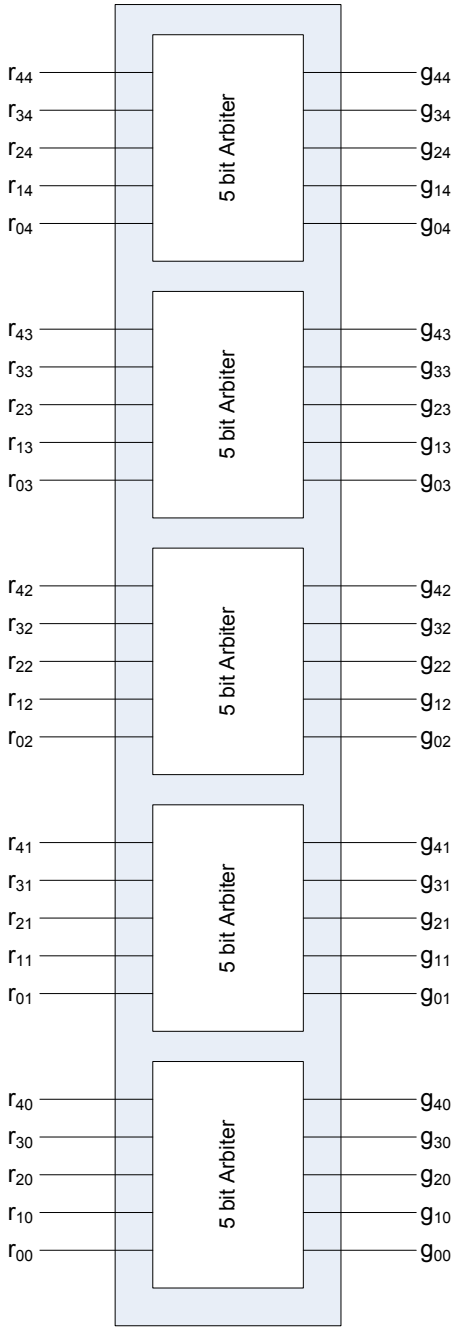

FIGURE 8: Allocator composed of five, Variable Priority Iterative Arbiters. When request input r_{ij} is asserted, input unit *i* wants access to output *j*. When grant output g_{ij} is asserted, input unit *i* has been granted access to output *j*.

Upon winning arbitration, the switch allocator sets the switch (switch allocation), asserts the corresponding write signal and indicates to the input unit that the message has been sent. The packet then traverses the switch and link.

The router keeps track of buffer availability in downstream routers via a STALL/GO system, kept simple by the use of fixed length buffers without virtual channels. When an input buffer is full, a hold signal is asserted. The upstream router monitors these signals and ensures that when asserted, no data is sent on that port.

The following figure illustrates the router pipeline for a packet composed of four flits assuming no contention.

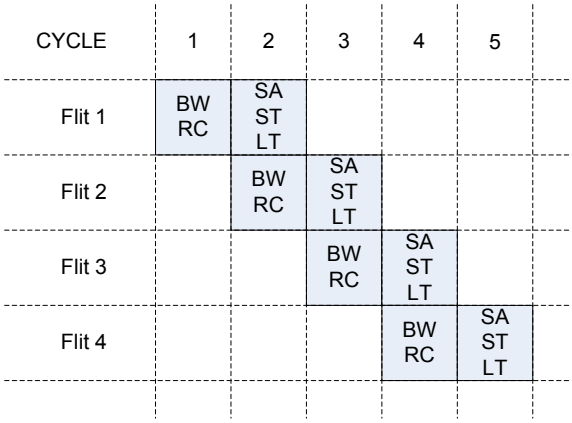

FIGURE 9: Pipelined routing of a packet. Each flit of the packet proceeds through the stages of Buffer Write (BW), Route Calculation (RC), Switch Allocation (SA), Switch Traversal (ST) and Link Traversal (LT). In the absence of contention, each flit of the packet enters the pipeline one cycle behind the preceding flit. The network does not in fact distinguish between head, tail or body flits as each flit contains the destination address.

3. Implementation

The router has been implemented using SystemVerilog. SystemVerilog provides an IEEE standard, unified hardware design, specification, and verification language [28]. The following provides an overview of the individual module input and outputs and the module hierarchy. The annotated SystemVerilog code for each module can be found in appendix A.

3.1 MODULE HIERARCHY

The following diagram details the module hierarchy.

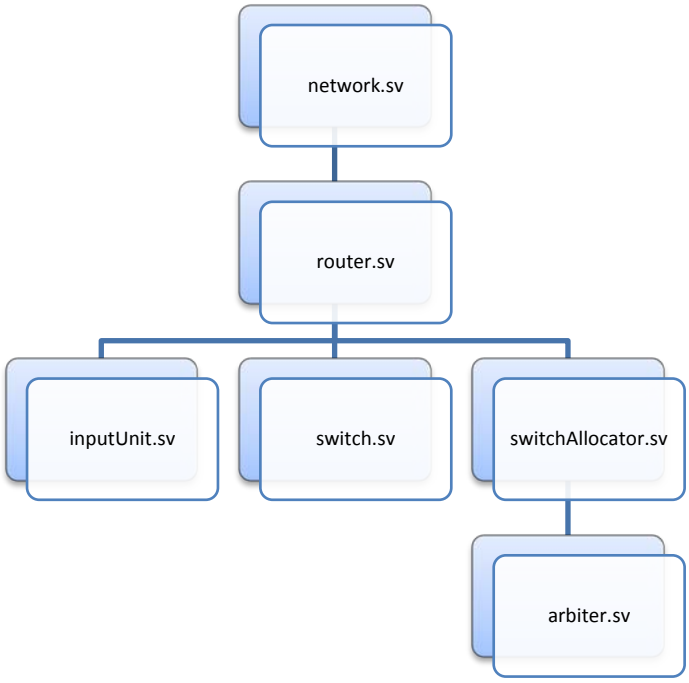

FIGURE 10: Block diagram of the SystemVerilog module heirarchy.

3.2 NETWORK

The network module was implemented using synthesizable SystemVerilog code. The module instantiates a number of router.sv modules, addresses and interconnects them according to given parameters for the number of nodes in the X and Y direction, the width of the network interconnects (flit size), the input buffer depth and the type of network, either mesh or torus.

3.2.1 Module

The following block diagram shows the network.sv module, its parameters, inputs and outputs.

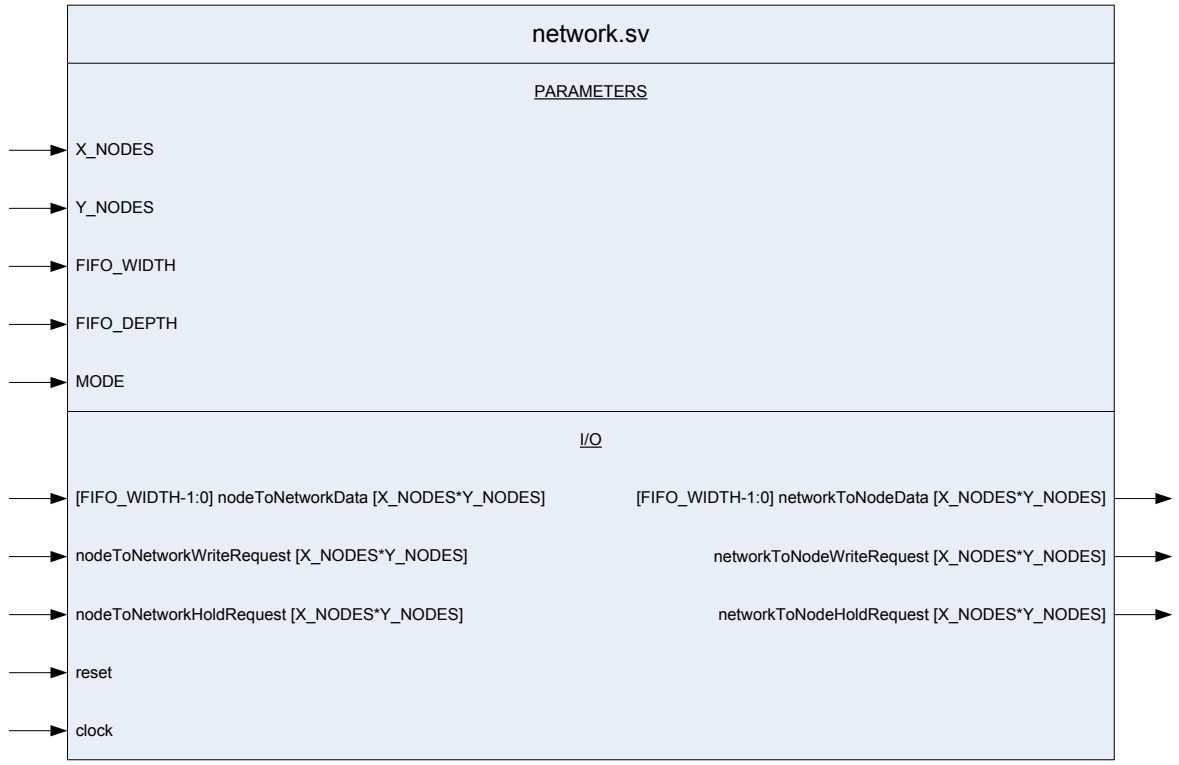

FIGURE 11: Block diagram of the SystemVerilog module 'network.sv' detailing parameters and I/0 ports.

3.2.1 Parameters

The table below details the parameters used in the 'network.sv' module.

Name	Description
X_NODES	Number of nodes in the network in the X direction
Y_NODES	Number of nodes in the network in the Y direction
FIFO_WIDTH	Flit size
FIFO_DEPTH	Input buffer size
MODE	Type of network '0' for mesh, '1' for torus

3.2.3 Inputs

The following table details the inputs used in the 'network.sv' module.

Name	Size	Description
nodeToNetworkData	[FIFO_WIDTH-1: 0] * [X_NODES*Y_NODES]	Data connections from each node, of a flit size and node number determined by the parameters.
nodeToNetworkWriteRequest	[X_NODES*Y_NODES]	A single connection from each node that qualifies the corresponding 'nodeToNetworkData'. Logic high indicates that the data on 'nodeToNetworkData' is valid and to be sampled at the next clock cycle.
nodeToNetworkHoldRequest	[X_NODES*Y_NODES]	A single connection from each node. Logic high indicates to the network that the nodes own internal input buffer is full. This stops the network outputting data.
reset	1	Logic high clears all network resources.
clock		Clock.

3.2.4 Outputs

The following table details the outputs generated by the 'network.sv' module.

Name	Size	Description
networkToNodeData	[FIFO_WIDTH-1: 0] * [X_NODES*Y_NODES]	Data connections to each node from the network, of a flit size and node number determined by the parameters.
networkToNodeWriteRequest	[X_NODES*Y_NODES]	A single connection to each node from the network that qualifies the 'networkToNodeData'. Logic high indicates that the data on 'networkToNodeData' is valid and to be sampled at the next clock cycle.
networkToNodeHoldRequest	[X_NODES*Y_NODES]	A single connection from each node. Logic high indicates to the network that the nodes own internal input buffer is full. This stops the network outputting data.

3.3 ROUTER

The five port, input buffered router described in section 2.3.5 was implemented using synthesizable SystemVerilog code. This module instantiates five instances of the 'inputUnit.sv' module (one for each port), one instance of the 'switch.sv' module and one instance of the 'switchAllocator.sv' module according to given parameters for the X and Y location of the router, the number of nodes in the X and Y direction, the width of the network interconnects (flit size), the input buffer depth and the type of network, either mesh or torus.

3.3.1 Module

The following block diagram shows the 'router.sv' module, its parameters, inputs and outputs.

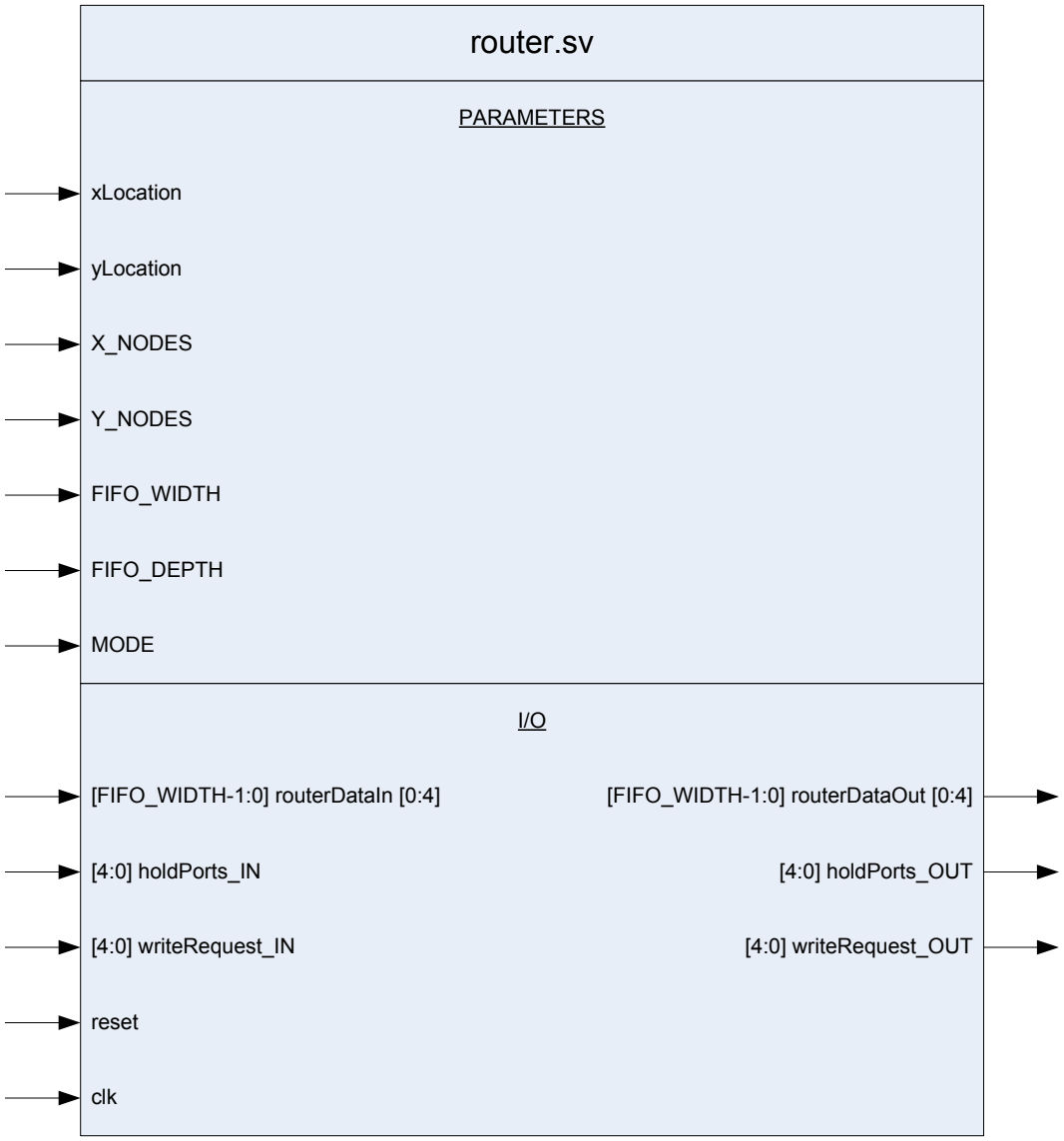

FIGURE 12: Block diagram of the SystemVerilog module 'router.sv' detailing parameters and I/0 ports.

3.3.3 Parameters

The following table details the parameters used in the 'router.sv' module.

Name	Description
xLocation	This routers location in the X direction given as a five bit unsigned binary number
yLocation	This routers location in the Y direction given as a five bit unsigned binary number
X_NODES	Number of nodes in the network in the X direction
Y_NODES	Number of nodes in the network in the Y direction
FIFO_WIDTH	Flit size
FIFO_DEPTH	Input buffer size
MODE	Type of network '0' for mesh, '1' for torus

3.3.2 Inputs

The following table details the inputs used in the 'router.sv' module.

Name	Size	Description
routerDataIn	[FIFO_WIDTH-1: 0] * [0:4]	Five input ports corresponding to the neighbouring routers and one from the local processing element, of a flit size determined by the parameters.
holdPorts_IN	[4:0]	Five connections, one from each neighbouring routers and one from the local processing element. Logic high indicates that the downstream node or routers input buffer is full and that no data should be sent on that port.
writeRequest_IN	[4:0]	Five connections, one from each of the neighbouring routers and one from the local processing element that qualifies the corresponding 'routerDataIn'. Logic high indicates that the corresponding data on 'routerDataIn' is valid and to be sampled at the next clock cycle.
reset	1	Logic high clears all network resources.
clock	1	Clock.

3.3.3 Outputs

The following table details the outputs generated by the 'router.sv' module.

Name	Size	Description
routerDataOut	[FIFO_WIDTH-1: 0] * [0:4]	Five output ports, one from each of the neighbouring routers and one from the local processing element, of a flit size determined by the parameters.
holdPorts_OUT	[4:0]	Five connections, one to each of the neighbouring routers and one to the local processing element. Logic high indicates to the upstream node or routers that this input buffer is full and that no data should be sent to this port.
writeRequest_OUT	[4:0]	Five connections, one to each of the neighbouring routers and one to the local processing element that qualifies the corresponding 'routerDataOut'. Logic high indicates to the downstream router or node that the corresponding data on 'routerDataOut' is valid and to be sampled at the next clock cycle.

3.4 SWITCH

A five input, five output crossbar switch as described in section 2.3.5 was implemented using synthesizable SystemVerilog code. A single 'switch.sv' module is instantiated by the 'router.sv' module according to given parameters for the number of input, outputs, and the flit size.

3.4.1 Module

The following block diagram shows the 'switch.sv' module, its parameters, inputs and outputs.

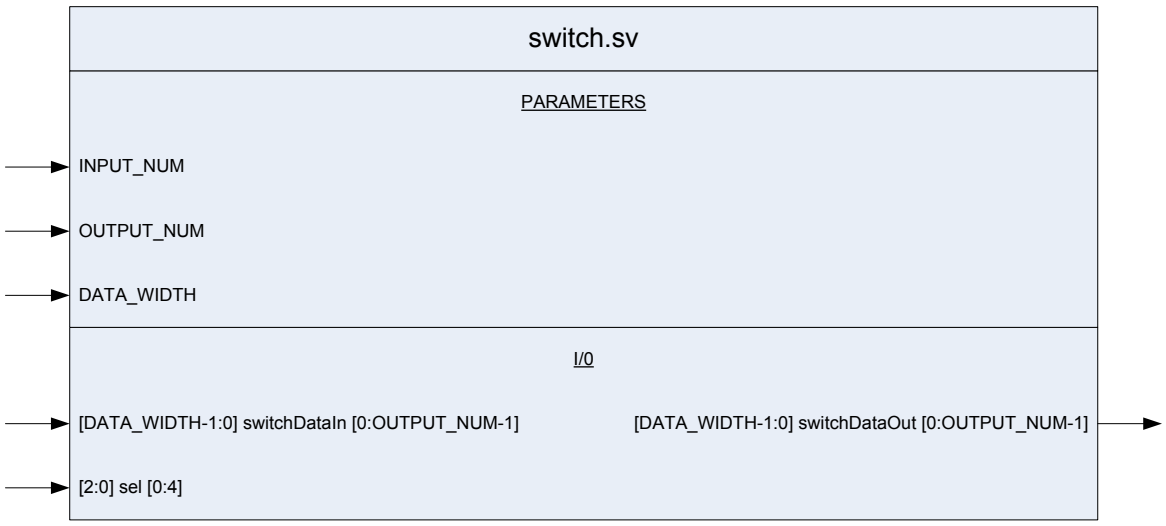

FIGURE 13: Block diagram of the SystemVerilog module 'switch.sv' detailing parameters and I/0 ports.

3.4.2 Parameters

The table below details the parameters used in the 'switch.sv' module.

Name	Description
INPUT_NUM	The number of input ports required (5)
OUTPUT_NUM	The number of output ports required (5)
DATA_WIDTH	Flit size

3.3.2 Inputs

The following table details the inputs used in the 'switch.sv' module.

Name	Size	Description
switchDataIn	[DATA_WIDTH-1: 0] * [0:OUTPUT_NUM-1]	Five input ports corresponding to the five input buffers, of a flit size determined by the parameters.
sel	[2:0] * [0:4]	Five input ports corresponding to the selection switch of a multiplexer for each output port. A three bit binary selects the corresponding requested input to the output.

3.3.3 Outputs

The following table details the outputs generated by the 'switch.sv' module.

Name	Size	Description
switchDataOut	[DATA_WIDTH-1: 0] * [0:OUTPUT_NUM-1]	Five output ports corresponding to the north, south, east and west neighbouring routers and the local processing element, of a flit size determined by the parameters.

3.5 SWITCH ALLOCATOR

A five input, five output switch allocator as described in section 2.3.5 was implemented using synthesizable SystemVerilog code. A single 'switchAllocator.sv' module is instantiated by the 'router.sv' module.

3.5.1 Module

The following block diagram shows the 'switchAllocator.sv' module, its inputs and its outputs.

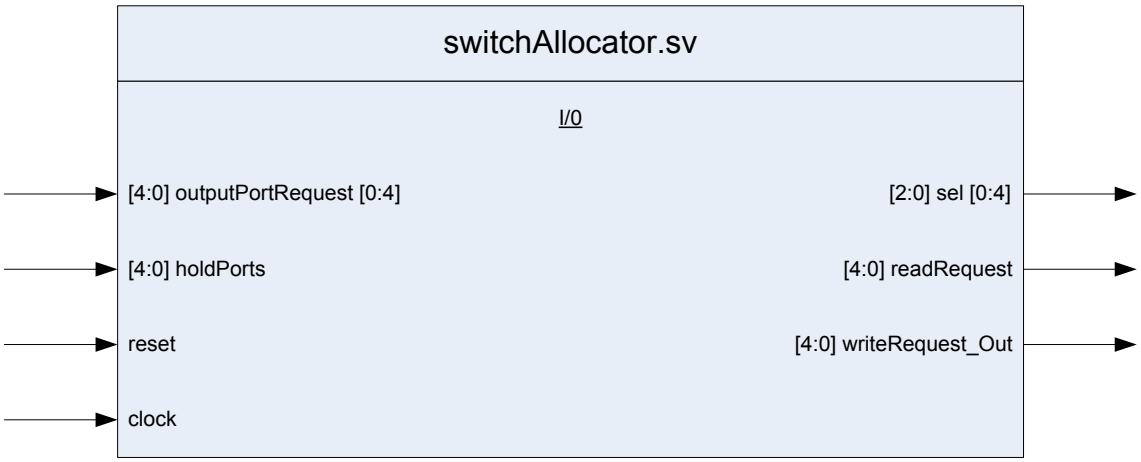

FIGURE 14: Block diagram of the SystemVerilog module 'switchAllocator.sv' detailing I/0 ports.

3.5.2 Inputs

The following table details the inputs used in the 'switchAllocator.sv' module.

Name	Size	Description
outputPortRequest	[4:0] * [0:4]	Five, one-hot five bit words corresponding to the five input buffers and the five output ports. Each input buffer creates a one-hot output port request which is passed to the switch allocator for arbitration.
holdPorts	[4:0]	Five connections, one to each of the neighbouring routers and one to the local processing element. Logic high indicates that the downstream node or routers input buffer is full and that no data should be sent on that port, eliminating it from arbitration.
reset	1	Logic high clears all network resources.
clock	1	Clock.

3.5.3 Outputs

The following table details the outputs generated by the 'switchAllocator.sv' module.

Name	Size	Description
sel	[2:0] * [0:4]	Five ports corresponding to the selection switch of a multiplexer for the north, south, east and west neighbouring routers and the local processing element output. A three bit binary selects the corresponding requested input to the output.
readRequest	[4:0]	A single connection to each input unit. Logic high indicates the unit has won arbitration and the data is to be sampled at the next clock cycle.
writeRequest_Out		Five connections, one to each of the neighbouring routers and one to the local processing element that qualifies the corresponding 'routerDataOut'. Logic high indicates to the downstream router or node that the corresponding data on 'routerDataOut' is valid and to be sampled at the next clock cycle.

3.6 ARBITER

A variable priority iterative arbiter using a round robin priority input was implemented using synthesizable SystemVerilog code. Five 'arbiter.sv' modules are instantiated by the 'switchAllocator.sv' module.

3.6.1 Module

The following block diagram shows the 'arbiter.sv' module, its inputs and its outputs.

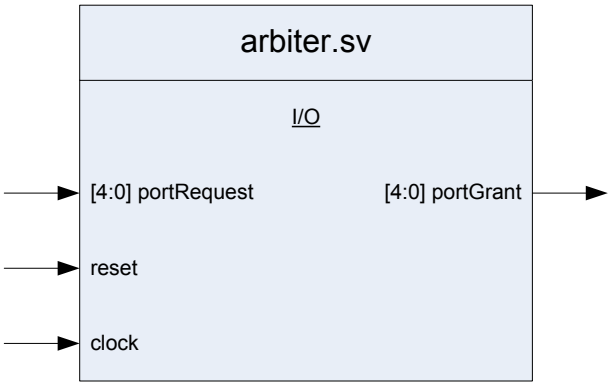

FIGURE 15: Block diagram of the SystemVerilog module 'arbiter.sv' showing its I/0 ports.

3.6.2 Inputs

The following table details the inputs used in the 'arbiter.sv' module.

Name	Size	Description
portRequest	[4:0]	One-hot, five bit word corresponding to the five input ports. Logic high indicates an output port request from the corresponding input router.
reset	1	Logic high clears all network resources.
clock	1	Clock.

3.6.3 Output

The following table details the outputs generated by the 'arbiter.sv' module.

Name	Size	Description
portGrant	[4:0]	One-hot, five bit word corresponding to the five input ports. Logic high indicates an output port request from the corresponding input router has been granted.

3.7 INPUT UNIT

An input unit containing a fixed length, first in first out queue and routing calculation logic was implemented using synthesizable SystemVerilog code. The 'router.sv' module instantiates five instances of the 'inputUnit.sv' module (one for each port) according to given parameters for the X and Y location of the router, the number of nodes in the X and Y direction, the width of the network interconnects (flit size), the input buffer depth and the type of network, either mesh or torus.

3.7.1 Module

The following block diagram shows the 'inputUnit.sv' module, its parameters, inputs and outputs.

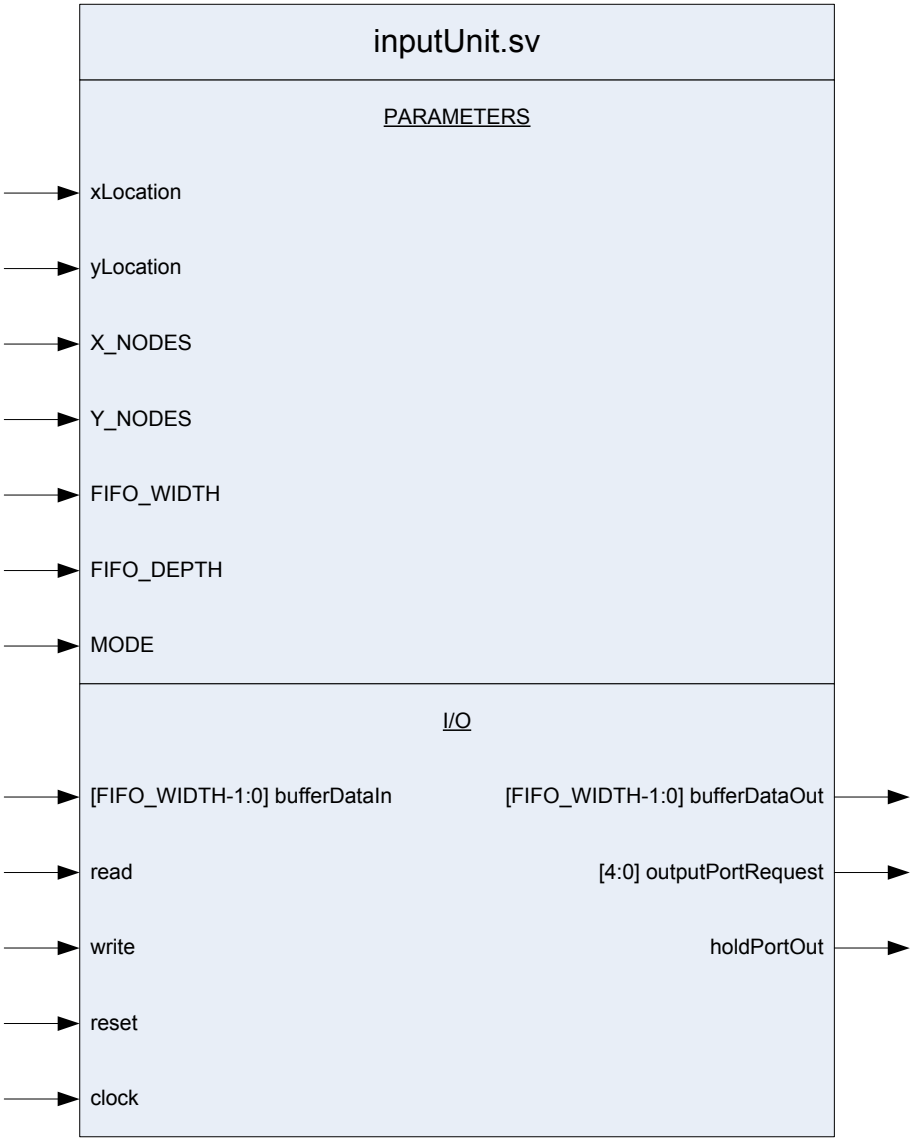

FIGURE 16: Block diagram of the SystemVerilog module 'inputUnit.sv' detailing parameters and I/0 ports.

3.7.2 Parameters

The following table details the parameters used in the 'inputUnit.sv' module.

Name	Description
xLocation	This routers location in the X direction given as a five bit unsigned binary number
yLocation	This routers location in the Y direction given as a five bit unsigned binary number
X_NODES	Number of nodes in the network in the X direction
Y_NODES	Number of nodes in the network in the Y direction
FIFO_WIDTH	Flit size
FIFO_DEPTH	Input buffer size
MODE	Type of network '0' for mesh, '1' for torus

3.7.3 Inputs

The following table details the inputs used in the 'inputUnit.sv' module.

Name	Size	Description
bufferDataIn	[FIFO_WIDTH-1: 0]	Input for data of a flit size determined by the parameters.
read	1	Logic high indicates the unit has won arbitration and the data at the front of the queue is to be sampled at the next clock cycle.
write	1	Single bit that qualifies 'bufferDataIn'. Logic high indicates the data on 'bufferDataIn' is valid and to be sampled at the next clock cycle.
reset	1	Logic high clears all network resources.
clock	1	Clock.

3.7.4 Outputs

The following table details the outputs generated by the 'inputUnit.sv' module.

Name	Size	Description
bufferDataOut	[FIFO_WIDTH-1: 0]	Output for data of a flit size determined by the parameters
outputPortRequest	[4:0]	One-hot, five bit word where each individual bit corresponds to one of the five output ports. Logic high indicates to the switch allocator that the data at the front of the buffer queue wants to leave on the corresponding output port.

4. Performance Analysis

In order to test the network a module was designed that simulated the presence of a node at each port on the interconnection network. These simulated nodes are controlled so as to perform a specified sequence of events that obtain the required data to interpret results. The purpose of this module was to measure the performance of the network in terms of throughput and latency.

4.1 THEORETICAL DESIGN OF NETWORK PERFORMANCE ANALYSIS

Each network port has connected to it a node simulator. The node simulator is comprises of a packet source separated from the network by a variable length source queue. Packets are generated according to a pattern specified by the simulation control and added to the end of the queue. When a packet is generated, it is time stamped and counted. A complimentary process counts the packets being written from the network to the node and calculates the time spent in the network by deducting the time in the timestamp from the current time.

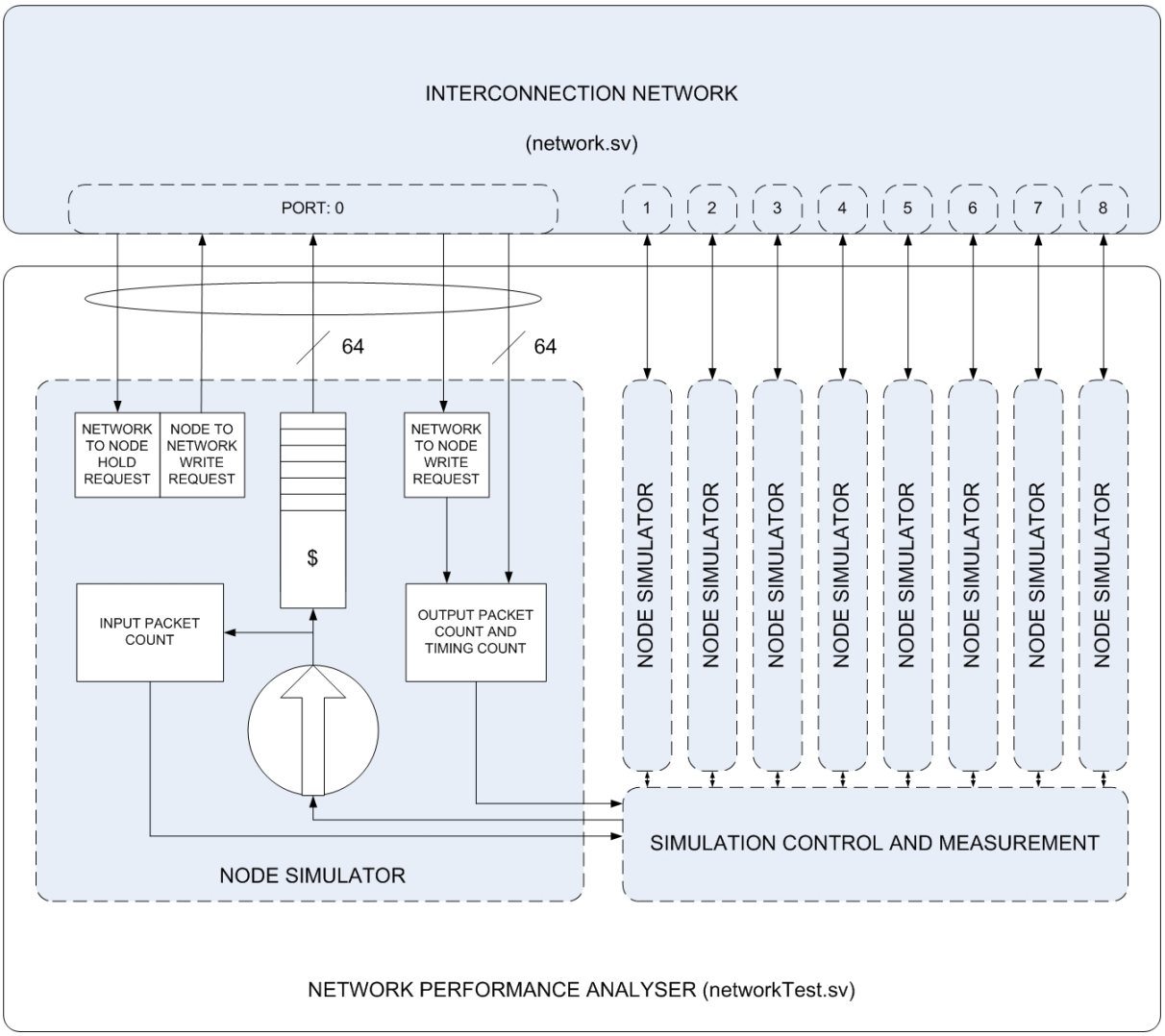

FIGURE 17: Theoretical design of a network performance analyser. The interconnection network under test that is shown has nine ports requiring node simulation.

4.1 PACKET GENERATION

A synthetic workload was created according to the Bernoulli injection process. With an injection rate r, the injection process P is a finite sequence of independent random variables with the probability of packet injection equal to the process rate:

$$P(A = 1) = r \quad [14]$$

Each time a packet was generated, it was given a random X and Y destination using the uniformly distributed random function in SystemVerilog. In order to identify how long packets spent in the network, the cycle at time of generation is stored in the data field. Once generated the packet is counted and added to the end of the source queue. Storing the cycle at time of generation in the data field of the packet ensured that the time the packet spent in the source queue waiting to enter the network was counted in the latency calculation.

4.2 STEADY STATE

For simplicity, the network simulator initialized the network with empty buffers and free resources. Packets entering the network initially see little contention and traverse the network without delay, later, if packets begin to fill up buffers the time taken to traverse the network will increase. After a certain amount of time the network will reach an equilibrium known as the steady state. At its steady state the influence of the initialization on the network performance is minimized. The time taken to reach steady state is known as the warm up period. This effect is mirrored at the end of the simulation as the last packets drain from the network, the packets leaving the network during this period will see less contention as no more packets are being injected.

In order to avoid a systemic error in measurement it is necessary to ensure the measurements for throughput and latency are taken using packets that enter the network only during its steady state; after a certain warm up period. It is also important to ensure that the simulation stays in its steady state by continuing to inject packets until all measurement packets have left the network. To do this, the simulation is split into three stages; warm up, steady state, and drain. The simulator performs a parameter defined number of cycles for each period. During the warm up and drain stages injected packets have no data stored in the data field ensuring that the measurement packets are easily identified by the simulator.

4.3 THROUGHPUT

Throughput is the rate at which the network delivers packets for a given traffic pattern [14]. During the steady state simulation stage packets arriving at the output of all ports on the network are counted. This value, accepted traffic, is contrasted with the offered traffic, the total count of packets generated by each node simulator attached to the network.

4.4 LATENCY

Each packet to be measured has written to its data field the cycle that it was generated. When measurement packets arrive from the network, indicated by the non-zero data field, the simulator subtracts the number stored in the data field from the current cycle thus providing a measure of latency. All other packets are ignored. Overall latency is calculated by taking a time average over all packets. As with throughput, average latency is contrasted with the offered traffic.

The measurements are taken using the batch means method. Measurements are taken from a long, single simulation run providing a set of n samples $\{X_0, X_1, X_2, \dots, X_n\}$. These samples are split into k individual batches, B, where k is between 20 and 30. For simplicity, n is chosen as sk. The batch means are thus computed as

$$\bar{B}_i = \frac{1}{s}\sum_{j=0}^{s-1} X_{si+j}, 0 \leq i < k \quad [14]$$

The sample mean is the mean of the batch means.

$$\bar{B} = \frac{1}{k}\sum_{i=0}^{k-1} \bar{B}_i \quad [14]$$

The standard deviation is

$$\sigma^2 = \frac{1}{k-1}\sum_{i=0}^{k-1} (\bar{B} - \bar{B}_i)^2 \quad [14]$$

The samples in the batch means method are an average of many different original samples, the variance between batch means is greatly reduced with comparison to the variance of the original sample set, which in turn reduces the standard deviation and increases confidence in the results [14].

4.5 IMPLEMENTATION

The network simulator was implemented using SystemVerilog. The following provides an overview of the module, the annotated SystemVerilog code can be found in Appendix A.

4.5.1 Module

The following block diagram shows the 'networkTest.sv' module and its parameters. The module outputs its results to three text files 'throughputFile.txt','latencyFile.txt' and 'steadyStateEstimationFile.txt'.

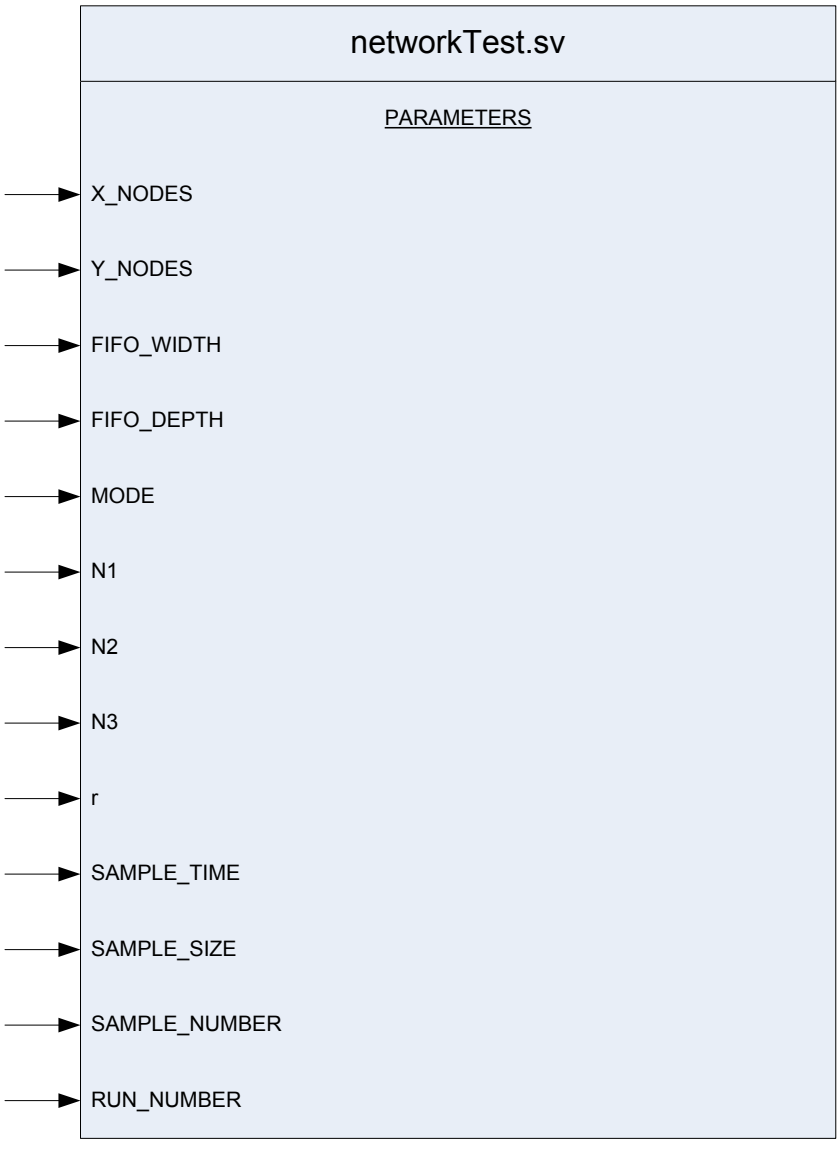

FIGURE 18: The 'networkTest.sv' module. The module instantiates an instance of the 'network.sv' module according to given parameters for the number of nodes in the X and Y direction, flit size, input buffer depth and network type (mesh or torus) and performs a simulation according to given parameters .

4.5.2 Parameters

The following table details the parameters used in the 'inputUnit.sv' module.

Name	Description
X_NODES	Number of nodes in the network in the X direction
Y_NODES	Number of nodes in the network in the Y direction
FIFO_WIDTH	Flit size
FIFO_DEPTH	Input buffer size
MODE	Type of network '0' for mesh, '1' for torus
N1	Warm Up Stage number of cycles
N2	Steady State Stage number of cycles
N3	Drain Stage number of cycles
r	Injection rate
SAMPLE_TIME	Number of cycles per batch average
SAMPLE_SIZE	Number of samples per batch average
SAMPLE_NUMBER	Number of samples
RUN_NUMBER	A number to indicate which simulation run. The random seed is a function of this parameter ensuring independent yet reproducible simulation runs.

5. Results

A scalable, synthesizable NoC has been designed and simulated using SystemVerilog. A complimentary scalable network simulator has been designed using SystemVerilog enabling performance analysis of the designed NoC under differing configurations. The following sections provide example simulation results from a 64 core 2D Mesh NoC from which the impact of input buffer size on network performance was investigated and a . Simulations were run using Mentor Graphics ModelSim.

5.2 Simulation Warm Up Time Estimation of a 64 core 2D Mesh NoC

In order to estimate the simulation warm up time, a simulation was run that counted packet arrivals and performed batch averages of 100 individual packet latencies. A linear fit of the data was produced and the results are shown in figure NUMBER. It can be seen that from a single simulation run the network doesn't appear to reach its steady state even after 30,000 packet arrivals. This is because the process being sampled is random, introducing sampling error. By creating an ensemble average composed of 30 independent simulation runs the sampling error is minimized and the network can be seen to enter the steady state after approximately 2,000 packet arrivals. The warm up time for the throughput and latency calculation was thus chosen as 2000 cycles as injection rates greater than approximately 0.015 would inject more than 2000 packets over this amount of cycles.

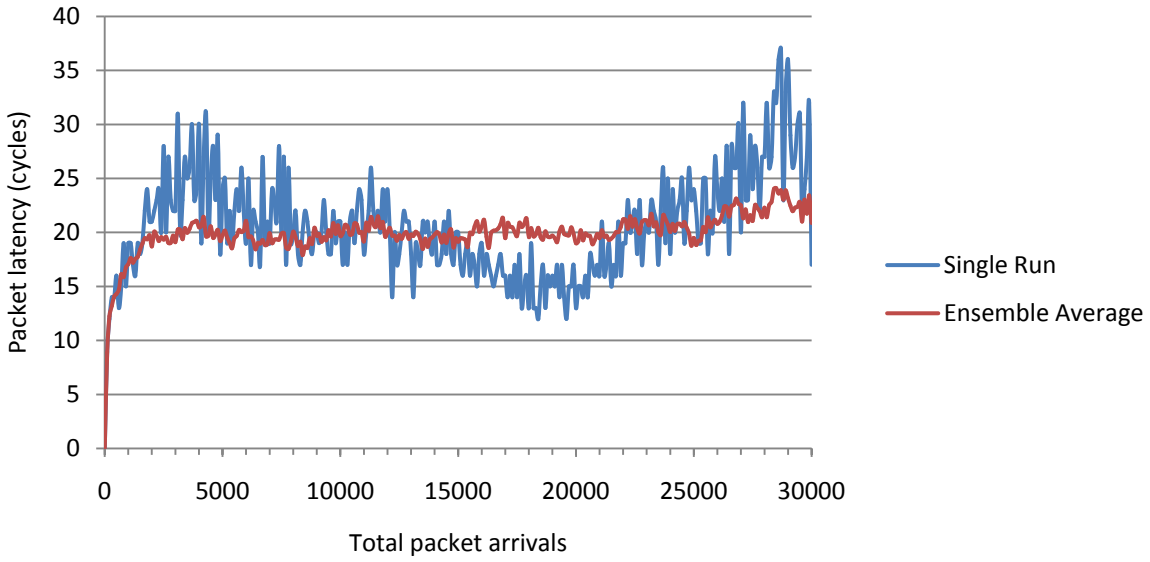

FIGURE 19: Average packet latency in a 64 core 2D mesh under uniform Bernoulli traffic at 30% capacity. Each sample is a time average of 100 individual packet latencies and the ensemble average is composed of 30 independent simulation runs.

5.3 Throughput of a 64 core 2D Mesh NoC

In order to calculate the throughput, multiple simulations were run with differing input buffer depths over a range of injection rates in order to measure accepted traffic as a function of demand. To automate this process '.do' files were created to be interpreted by Mentor Graphics ModelSim.

The results are shown in figure 20 and 21. Although multiple simulations were run, these were sampled using the batch means method. This means that for each injection rate, only one traffic pattern was examined. Batches from a single simulation run are not independent as the packets remaining in the network at the end of one batch are what constitute the beginning of the next batch. This effect was minimized by using a minimum batch size of 122 packets.

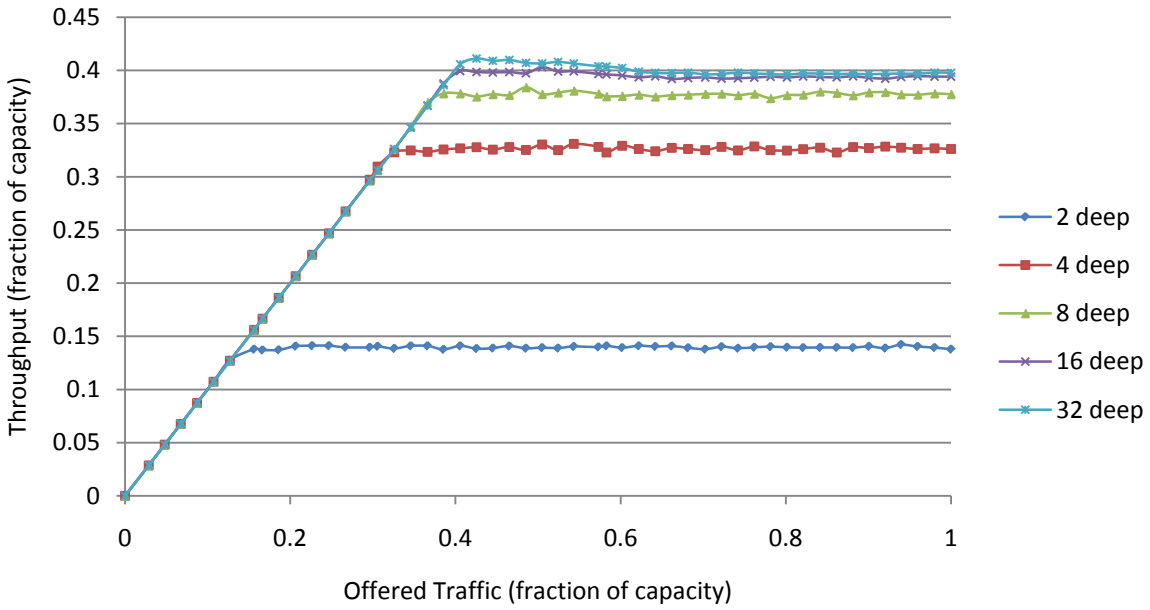

FIGURE 20: Throughput as a function of Offered Traffic in a 64 core 2D mesh under uniform Bernoulli traffic measured over 2000 cycles after a warm up period of 2000 cycles. Each series represents a different router input buffer depth. As the performance of the network does not degrade as offered traffic increases past saturation the network is said to be stable.

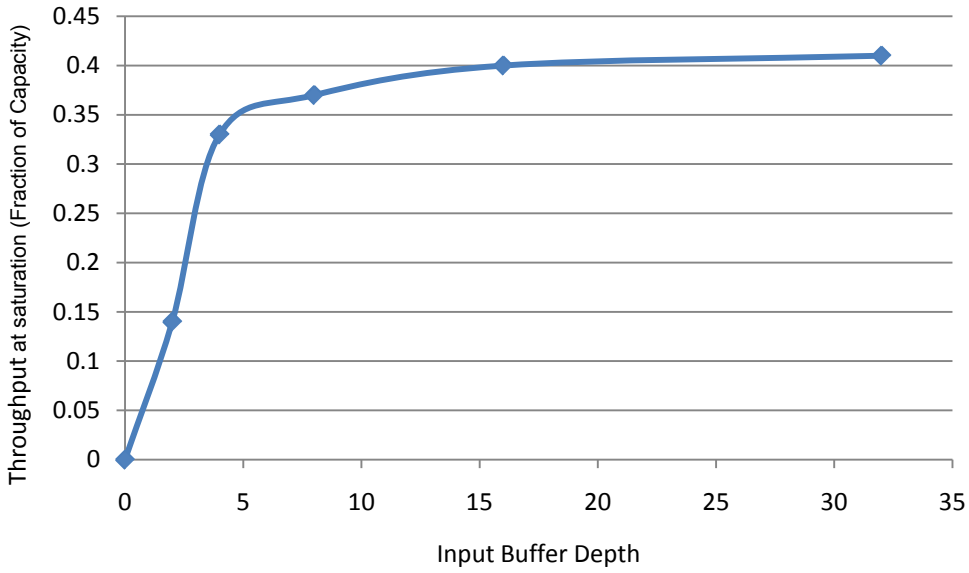

FIGURE 21: Throughput at Saturation as a function of Input Buffer Depth in a 64 core 2D mesh under uniform Bernoulli traffic measured over 2000 cycles after a warm up period of 2000 cycles.

It can be seen that the effect of increasing buffer sizes decreases rapidly. This is an important result; increasing buffer sizes increases the physical size of the routers on the chip.

5.4 Latency of a 64 core 2D Mesh NoC

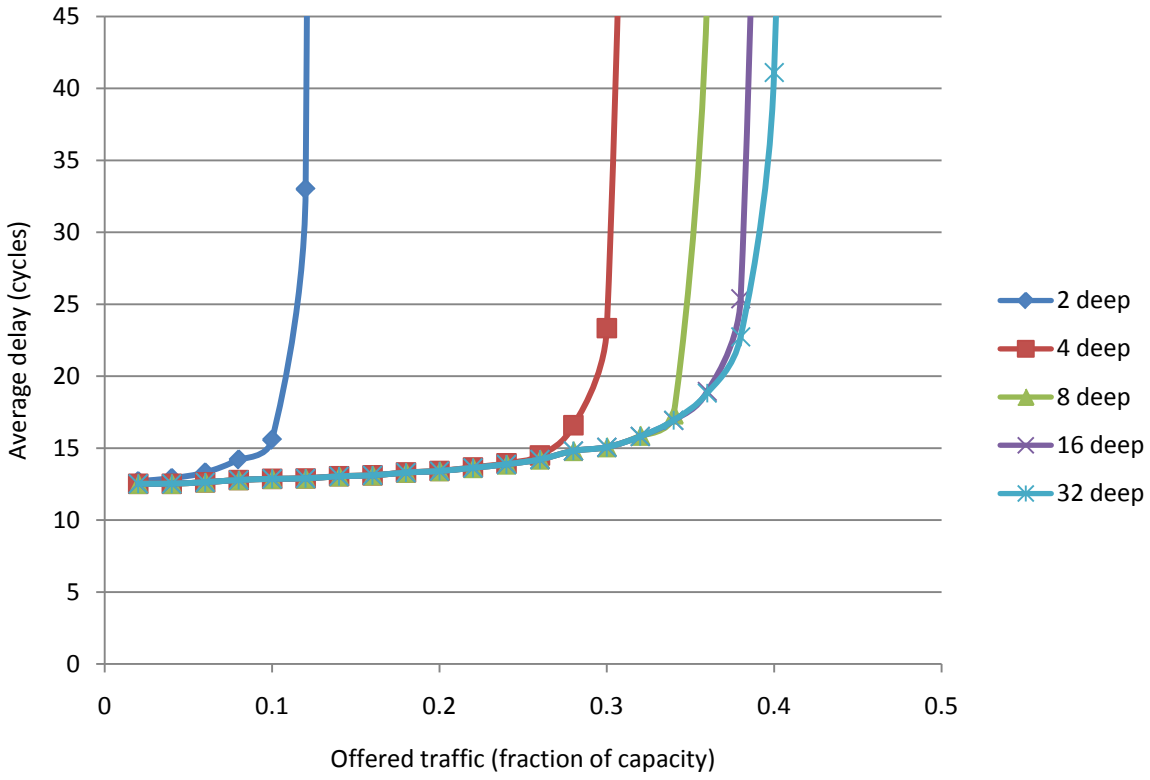

FIGURE 22: Average Latency against Offered Traffic in a 64 core 2D mesh under uniform Bernoulli traffic measured over 2000 cycles after a warm up period of 2000 cycles.

It can be seen that the increase in buffer size, which translates directly to router die size makes little difference to the steady state latency average however, confirming previous results, does increase the point at which the network enters saturation.

6. Conclusion

A scalable network on chip and a network simulation tool was successfully designed and simulated using SystemVerilog. The network took user defined parameters making it scalable to any number of cores, connected either using a Mesh or Torus topology, allowing for any flit size and for any input buffer depth.

This allowed me to investigate many aspects of the Network-on-Chip Architecture. From simulations of a 64 Core 2D Mesh investigated the effect of increasing input buffer sizes on both average throughput and average latency. Increasing buffer sizes allowed for an increase in the network saturation point however steady state latency performance increases were negligible. This result is important as buffers take up valuable die area. In terms of fault tolerance, as the network uses deterministic dimension-ordered routing any single fault will disconnect all source destination pairs that communicate over that channel. The performance of the network therefore degrades gracefully with each fault as the available node communication pairs are reduced. The importance of this fault behaviour is application specific.

The router, network and simulation tool provide a model for future work. With further testing mathematical models could be approximated for throughput at saturation with respect to input buffer size that may allow for novel buffer management techniques. The saturation throughput could be increased by improving the efficiency of arbitration and allocation of resources at the cost of added implementation complexity. Power and die savings could be made by changing binary counting to gray code. A different routing algorithm could be implemented at the cost of added complexity in order to increase the fault tolerance. Similar simulations to the ones shown could be run with the network in the torus configuration to provide comparisons between topologies. The network simulator could be updated to include differing traffic types such as an application driven load. The network could be synthesised using an FPGA and timing analysis performed.

7. Appendix

7.1 Design Module SystemVerilog Code

The following section contains copies of the SystemVerilog code used to implement the various modules described in section 3 and 4. Each section is titled with the file name of the code contained within it.

7.1.1 network.sv

```
module network
  #(parameter X_NODES = 3,
    parameter Y_NODES = 3,
    parameter FIFO_WIDTH = 64,
    parameter FIFO_DEPTH = 4,
    parameter MODE = 1)

    // Inputs and outputs for connected nodes

    (output logic [FIFO_WIDTH-1:0] networkToNodeData [X_NODES*Y_NODES],
     output logic networkToNodeWriteRequest          [X_NODES*Y_NODES],
     output logic networkToNodeHoldRequest           [X_NODES*Y_NODES],

     // ------------------------------------------------------------------------
     // These outputs are used only for debugging with testNetwork module.  Comment
     // when not in use.
     output logic [FIFO_WIDTH-1:0] networkData       [0: (X_NODES * Y_NODES)-1][0:4],
     output logic [FIFO_WIDTH-1:0] routerInputData   [0: (X_NODES * Y_NODES)-1][0:4],
     // ------------------------------------------------------------------------

      input logic [FIFO_WIDTH-1:0] nodeToNetworkData [X_NODES*Y_NODES],
      input logic nodeToNetworkWriteRequest          [X_NODES*Y_NODES],
      input logic nodeToNetworkHoldRequest           [X_NODES*Y_NODES],
      input logic reset, clk);


  // Internal logic for generating node numbers and x and y location

  logic [2:0]  xLocation;
  logic [2:0]  yLocation;
  logic nodeNumber;
```

```
  // Internal logic for calculating and switching network data

  // logic [FIFO_WIDTH-1:0] networkData       [0: (X_NODES * Y_NODES)-1][0:4];   Currently being used for debugging.
  // logic [FIFO_WIDTH-1:0] routerInputData   [0: (X_NODES * Y_NODES)-1][0:4];
  logic [4:0]  routerToNetworkHoldPorts    [0: (X_NODES * Y_NODES)-1];
  logic [4:0]  networkToRouterHoldPorts    [0: (X_NODES * Y_NODES)-1];
  logic [4:0]  routerToNetworkWriteRequest [0: (X_NODES * Y_NODES)-1];
  logic [4:0]  networkToRouterWriteRequest [0: (X_NODES * Y_NODES)-1];


  // Generate router inputs

  always_comb
    begin
      for (int nodeNumber = 0; nodeNumber < (X_NODES*Y_NODES); nodeNumber++)
        begin
          if (MODE == 0)
            begin


              // Generate mesh router input Data using network data

              routerInputData[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? networkData [nodeNumber + X_NODES][2] :
'b0;
              routerInputData[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? 'b0 : networkData [nodeNumber + 1][3];
              routerInputData[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? networkData [nodeNumber - X_NODES][0] :
'b0;
              routerInputData[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? 'b0 : networkData [nodeNumber -1][1];
              routerInputData[nodeNumber][4] = nodeToNetworkData[nodeNumber];


              // Generate mesh router hold request inputs

              networkToRouterHoldPorts[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? routerToNetworkHoldPorts
[nodeNumber + X_NODES][2] : 1'b0;
              networkToRouterHoldPorts[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? 1'b0 : routerToNetworkHoldPorts
[nodeNumber + 1][3];
              networkToRouterHoldPorts[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? routerToNetworkHoldPorts
[nodeNumber - X_NODES][0] : 1'b0;
              networkToRouterHoldPorts[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? 1'b0 : routerToNetworkHoldPorts
[nodeNumber -1][1];
              networkToRouterHoldPorts[nodeNumber][4] = nodeToNetworkHoldRequest[nodeNumber];
```

```
              // Generate mesh router write request inputs

              networkToRouterWriteRequest[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? routerToNetworkWriteRequest
[nodeNumber + X_NODES][2] : 1'b0;
              networkToRouterWriteRequest[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? 1'b0 :
routerToNetworkWriteRequest [nodeNumber + 1][3];
              networkToRouterWriteRequest[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? routerToNetworkWriteRequest
[nodeNumber - X_NODES][0] : 1'b0;
              networkToRouterWriteRequest[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? 1'b0 :
routerToNetworkWriteRequest [nodeNumber -1][1];
              networkToRouterWriteRequest[nodeNumber][4] = nodeToNetworkWriteRequest[nodeNumber];


              // Generate mesh network to Node data

              networkToNodeData [nodeNumber] = networkData[nodeNumber][4];
              networkToNodeWriteRequest [nodeNumber] = routerToNetworkWriteRequest [nodeNumber][4];
              networkToNodeHoldRequest [nodeNumber]  = routerToNetworkHoldPorts [nodeNumber][4];

            end

          if (MODE == 1)
            begin

              // Generate tori router input Data using network data

              routerInputData[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? networkData [nodeNumber + X_NODES][2]
: networkData [nodeNumber - (X_NODES*(Y_NODES-1))][2];
              routerInputData[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? networkData [nodeNumber - (X_NODES-1)][3]
: networkData [nodeNumber + 1][3];
              routerInputData[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? networkData [nodeNumber - X_NODES][0]
: networkData [nodeNumber + (X_NODES*(Y_NODES-1))][0];
              routerInputData[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? networkData [nodeNumber + (X_NODES-1)][1]
: networkData [nodeNumber -1][1];
              routerInputData[nodeNumber][4] = nodeToNetworkData[nodeNumber];


              // Generate tori router hold request inputs

              networkToRouterHoldPorts[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? routerToNetworkHoldPorts
[nodeNumber + X_NODES][2]     : routerToNetworkHoldPorts [nodeNumber - (X_NODES*(Y_NODES-1))][2];
              networkToRouterHoldPorts[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? routerToNetworkHoldPorts
[nodeNumber - (X_NODES-1)][3] : routerToNetworkHoldPorts [nodeNumber + 1][3];
```

```
            networkToRouterHoldPorts[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? routerToNetworkHoldPorts
[nodeNumber - X_NODES][0]     : routerToNetworkHoldPorts [nodeNumber + (X_NODES*(Y_NODES-1))][0];
            networkToRouterHoldPorts[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? routerToNetworkHoldPorts
[nodeNumber + (X_NODES-1)][1] : routerToNetworkHoldPorts [nodeNumber -1][1];
            networkToRouterHoldPorts[nodeNumber][4] = nodeToNetworkHoldRequest[nodeNumber];


            // Generate tori router write request inputs

            networkToRouterWriteRequest[nodeNumber][0] = (nodeNumber < (X_NODES*(Y_NODES-1))) ? routerToNetworkWriteRequest
[nodeNumber + X_NODES][2]     : routerToNetworkWriteRequest [nodeNumber - (X_NODES*(Y_NODES-1))][2];
            networkToRouterWriteRequest[nodeNumber][1] = (((nodeNumber + 1) % X_NODES) == 0)  ? routerToNetworkWriteRequest
[nodeNumber - (X_NODES-1)][3] : routerToNetworkWriteRequest [nodeNumber + 1][3];
            networkToRouterWriteRequest[nodeNumber][2] = (nodeNumber > (X_NODES-1))           ? routerToNetworkWriteRequest
[nodeNumber - X_NODES][0]     : routerToNetworkWriteRequest [nodeNumber + (X_NODES*(Y_NODES-1))][0];
            networkToRouterWriteRequest[nodeNumber][3] = ((nodeNumber % X_NODES) == 0)        ? routerToNetworkWriteRequest
[nodeNumber + (X_NODES-1)][1] : routerToNetworkWriteRequest [nodeNumber -1][1];
            networkToRouterWriteRequest[nodeNumber][4] = nodeToNetworkWriteRequest[nodeNumber];


            // Generate tori network to Node data

            networkToNodeData [nodeNumber]         = networkData[nodeNumber][4];
            networkToNodeWriteRequest [nodeNumber] = routerToNetworkWriteRequest [nodeNumber][4];
            networkToNodeHoldRequest [nodeNumber]  = routerToNetworkHoldPorts [nodeNumber][4];

          end
        end
    end


  // Generate 2D Network

  genvar x,y;
    generate
      for ( y = 0; y < Y_NODES; y++)

        begin
          for ( x = 0; x < X_NODES; x++)
            begin
              router #(x, y, X_NODES, Y_NODES, FIFO_WIDTH, FIFO_DEPTH, MODE) router (networkData[x+(X_NODES*y)][0:4],
routerToNetworkHoldPorts[x+(X_NODES*y)], routerToNetworkWriteRequest[x+(X_NODES*y)], routerInputData[x+(X_NODES*y)][0:4],
networkToRouterHoldPorts[x+(X_NODES*y)], networkToRouterWriteRequest[x+(X_NODES*y)], reset, clk);
            end
```

```
        end
      endgenerate

endmodule
```

7.1.2 router.sv

```
module router

  // The router location and the width and depth of the FIFO are paramatised

  #(parameter xLocation = 0,
    parameter yLocation = 0,
    parameter X_NODES = 3,
    parameter Y_NODES = 3,
    parameter FIFO_WIDTH = 32,
    parameter FIFO_DEPTH = 4,
    parameter MODE = 0)


    // Outputs and inputs

    (output logic [FIFO_WIDTH-1:0] routerDataOut [0:4],
     output logic [4:0] holdPorts_OUT, writeRequest_OUT,
      input logic [FIFO_WIDTH-1:0] routerDataIn [0:4],
      input logic [4:0] holdPorts_IN, writeRequest_IN,
      input logic reset, clk);


    // Internal Logic, essentially this is just connections.

    logic [FIFO_WIDTH-1:0] bufferDataOut [0:4];
    logic [4:0] outputPortRequest [0:4];
    logic [4:0] readRequest;
    logic [2:0] sel [0:4];


    // Generate 5 input units

    genvar j;
    generate
      for ( j = 0; j< 5; j++ )
        inputUnit #(xLocation, yLocation, X_NODES, Y_NODES, FIFO_WIDTH, FIFO_DEPTH, MODE) inputUnit (bufferDataOut[j],
outputPortRequest[j], holdPorts_OUT[j], routerDataIn[j], readRequest[j], writeRequest_IN[j], holdPorts_IN[j], reset, clk);
    endgenerate
```

```
// Instantiate switch allocator and switch

    switchAllocator switchAllocator (sel, readRequest, writeRequest_OUT, outputPortRequest, holdPorts_IN, reset, clk);
    switch #(5,5,FIFO_WIDTH) switch (routerDataOut, bufferDataOut, sel);


endmodule
```

7.1.3 switch.sv

```
module switch

 // The number of inputs, outputs and datawidth for the crossbar is paramatised.  Standard is a 5x5 64 bit crossbar.

  #(parameter INPUT_NUM = 5,
    parameter OUTPUT_NUM = 5,
    parameter DATA_WIDTH = 64)


  // Input data, output data and the port selects are described using an array

  (output logic [DATA_WIDTH-1:0] switchDataOut [0:OUTPUT_NUM-1],

    input logic [DATA_WIDTH-1:0] switchDataIn [0:OUTPUT_NUM-1],
    input logic [2:0] sel [0:4]);


    // Multiplexers are used to create crossbar.  This needs to be paramatised.

    always_comb
      begin
      case(sel[0])
        3'b000 : switchDataOut[0] = switchDataIn[0];
        3'b001 : switchDataOut[0] = switchDataIn[1];
        3'b010 : switchDataOut[0] = switchDataIn[2];
        3'b011 : switchDataOut[0] = switchDataIn[3];
        3'b100 : switchDataOut[0] = switchDataIn[4];
        3'b101 : switchDataOut[0] = 1'bz;
      endcase

      case(sel[1])
        3'b000 : switchDataOut[1] = switchDataIn[0];
        3'b001 : switchDataOut[1] = switchDataIn[1];
        3'b010 : switchDataOut[1] = switchDataIn[2];
        3'b011 : switchDataOut[1] = switchDataIn[3];
        3'b100 : switchDataOut[1] = switchDataIn[4];
        3'b101 : switchDataOut[1] = 1'bz;
```

```
      endcase

      case(sel[2])
        3'b000 : switchDataOut[2] = switchDataIn[0];
        3'b001 : switchDataOut[2] = switchDataIn[1];
        3'b010 : switchDataOut[2] = switchDataIn[2];
        3'b011 : switchDataOut[2] = switchDataIn[3];
        3'b100 : switchDataOut[2] = switchDataIn[4];
        3'b101 : switchDataOut[2] = 1'bz;
      endcase

      case(sel[3])
        3'b000 : switchDataOut[3] = switchDataIn[0];
        3'b001 : switchDataOut[3] = switchDataIn[1];
        3'b010 : switchDataOut[3] = switchDataIn[2];
        3'b011 : switchDataOut[3] = switchDataIn[3];
        3'b100 : switchDataOut[3] = switchDataIn[4];
        3'b101 : switchDataOut[3] = 1'bz;
      endcase

      case(sel[4])
        3'b000 : switchDataOut[4] = switchDataIn[0];
        3'b001 : switchDataOut[4] = switchDataIn[1];
        3'b010 : switchDataOut[4] = switchDataIn[2];
        3'b011 : switchDataOut[4] = switchDataIn[3];
        3'b100 : switchDataOut[4] = switchDataIn[4];
        3'b101 : switchDataOut[4] = 1'bz;
      endcase
      end
endmodule
```

7.1.4 switchAllocator.sv

```
module switchAllocator

  // Output to the crossbar a matrix containg the mux selection information and individual read requests to each
  // input unit contained in the router

  (output logic [2:0] sel [0:4],
   output logic [4:0] readRequest, writeRequest_Out,


    // Output port requests from all 5 input units contained within a single router are inputted to be arbitrated.
    // The one-hot request refers to the outgoing ports {Local, West, South, East, North}

    input logic [4:0] outputPortRequest [0:4],
    input logic [4:0] holdPorts,
    input logic reset, clk);


    // Internal logic for the computed request matrix and grant matrix.

    logic [4:0] requestMatrix [0:4];
    logic [4:0] grantMatrix [0:4];


    // Create an output port request matrix which is essentially just the transpose of the inputs after a logical 'and' with
    // the negation of the holdPorts data.  This will hold any data from being sent to full buffers.

    always_comb
      begin
        for (int j = 0; j < 5; j++)
          requestMatrix[j] = {outputPortRequest[4][j] && ~holdPorts[j], outputPortRequest[3][j] && ~holdPorts[j],
outputPortRequest[2][j] && ~holdPorts[j], outputPortRequest[1][j] && ~holdPorts[j], outputPortRequest[0][j] && ~holdPorts[j]};
        end


    // Input the request matrix into 5 arbiters in order to generate a grant matrix.

    genvar i;
    generate
    for ( i = 0; i < 5; i++ )
      begin
        arbiter #(5) ai (grantMatrix[i],requestMatrix[i], reset, clk);
```

```
    end
  endgenerate


  // Using the grantMatrix, determine which internal buffers need to read, and which output ports need to write
  // As the matrix is one-hot in both dimensions, an or of the rows and columns indicates whether to write/read

  always_comb
    begin
      readRequest[0] = |{grantMatrix[0][0], grantMatrix[1][0], grantMatrix[2][0], grantMatrix[3][0], grantMatrix[4][0]};
      readRequest[1] = |{grantMatrix[0][1], grantMatrix[1][1], grantMatrix[2][1], grantMatrix[3][1], grantMatrix[4][1]};
      readRequest[2] = |{grantMatrix[0][2], grantMatrix[1][2], grantMatrix[2][2], grantMatrix[3][2], grantMatrix[4][2]};
      readRequest[3] = |{grantMatrix[0][3], grantMatrix[1][3], grantMatrix[2][3], grantMatrix[3][3], grantMatrix[4][3]};
      readRequest[4] = |{grantMatrix[0][4], grantMatrix[1][4], grantMatrix[2][4], grantMatrix[3][4], grantMatrix[4][4]};
    end

  always_ff @(posedge clk)
    begin
      writeRequest_Out[0] = |grantMatrix[0];
      writeRequest_Out[1] = |grantMatrix[1];
      writeRequest_Out[2] = |grantMatrix[2];
      writeRequest_Out[3] = |grantMatrix[3];
      writeRequest_Out[4] = |grantMatrix[4];
    end


  // Encode the information from each word in the grant matrix to create a mux selection matrix

  always_ff @(posedge clk)
    begin
      for (int i = 0; i < 5; i++)
        begin
          unique casez (grantMatrix[i]) // Notice the invert.  It is a result of the transposition of the request matrix.
            5'b00001 : sel[i] = 000;      // Select input 0 on output port i
            5'b0001? : sel[i] = 001;      // Select input 1 on output port i
            5'b001?? : sel[i] = 010;      // Select input 2 on output port i
            5'b01??? : sel[i] = 011;      // Select input 3 on output port i
            5'b1???? : sel[i] = 100;      // Select input 4 on output port i
             default : sel[i] = 101;      // Select z on crossbar
          endcase
        end
    end

endmodule
```

7.1.5 arbiter.sv

```
module arbiter

// ------------------------------------------------------------------------------------------
// The arbiter outputs a one-hot 5 bit word denoting the port request that has been granted
// from the port requests inputted in a one-hot fashion.
// ------------------------------------------------------------------------------------------

  (output logic [4:0] portGrant,
    input logic [4:0] portRequest,
    input logic         reset, clk);

          logic [4:0] portPriority;
          logic [5:0]   carryBit;

// ------------------------------------------------------------------------------------------
// Netlist describing combinational logic of a 5 bit variable priority iterative arbiter with
// the wrap around combinational logic.  Some simulations have problems with wrap around
// logic.This logic can be used with different schemes to generate the one-hot priority input.
// ------------------------------------------------------------------------------------------

logic a, b, c, d, e;

or  gate1 (a, carryBit[0], portPriority[0]);
and gate2 (portGrant[0], portRequest[0], a), gate3 (carryBit[1], a, ~portRequest[0]);
or  gate4 (b, carryBit[1], portPriority[1]);
and gate5 (portGrant[1], portRequest[1], b), gate6 (carryBit[2], b, ~portRequest[1]);
or  gate7 (c, carryBit[2], portPriority[2]);
and gate8 (portGrant[2], portRequest[2], c), gate9 (carryBit[3], c, ~portRequest[2]);
or  gate10 (d, carryBit[3], portPriority[3]);
and gate11 (portGrant[3], portRequest[3], d), gate12 (carryBit[4], d, ~portRequest[3]);
or  gate13 (e, carryBit[4], portPriority[4]);
and gate14 (portGrant[4], portRequest[4], e), gate15 (carryBit[0], e, ~portRequest[4]);

// ------------------------------------------------------------------------------------------
// Priority input generation.  Fixed input, rotating and round robin methods are included.
// ------------------------------------------------------------------------------------------

always_ff @ (posedge clk)
  begin
    if (reset)
```

```
      portPriority <= 'b00001;
    else
      begin
        // Fixed (could simply be assigned)
        // portPriority <= 5'b10000;

        // Rotating
        // portPriority <= {portPriority[n-2:0], portPriority[n-1]};

        // Round Robin
        portPriority <= |portGrant ? {portGrant[3:0], portGrant[4]} : portPriority;
      end
  end

// ------------------------------------------------------------------------------------------
// Netlist describing an n-bit variable priority iterative arbiter without the wrap around
// combinational logic.  Some simulations have problems with wrap around logic.
// ------------------------------------------------------------------------------------------

/*
logic a,b,c,d,e,f,g,h,i,j;
logic [n-1:0] portGrantA;
logic [n-1:0] portGrantB;

or  gate1 (a, 0, portPriority[0]);
and gate2 (portGrantA[0], portRequest[0], a), gate3 (carryBit[1], a, ~portRequest[0]);
or  gate4 (b, carryBit[1], portPriority[1]);
and gate5 (portGrantA[1], portRequest[1], b), gate6 (carryBit[2], b, ~portRequest[1]);
or  gate7 (c, carryBit[2], portPriority[2]);
and gate8 (portGrantA[2], portRequest[2], c), gate9 (carryBit[3], c, ~portRequest[2]);
or  gate10 (d, carryBit[3], portPriority[3]);
and gate11 (portGrantA[3], portRequest[3], d), gate12 (carryBit[4], d, ~portRequest[3]);
or  gate13 (e, carryBit[4], portPriority[4]);
and gate14 (portGrantA[4], portRequest[4], e), gate15 (carryBit[5], e, ~portRequest[4]);

or  gate16 (f, carryBit[5], portPriority[0]);
and gate17 (portGrantB[0], portRequest[0], f), gate18 (carryBit[6], f, ~portRequest[0]);
or  gate19 (g, carryBit[6], portPriority[1]);
and gate20 (portGrantB[1], portRequest[1], g), gate21 (carryBit[7], g, ~portRequest[1]);
or  gate22 (h, carryBit[7], portPriority[2]);
and gate23 (portGrantB[2], portRequest[2], h), gate24 (carryBit[8], h, ~portRequest[2]);
or  gate25 (i, carryBit[8], portPriority[3]);
and gate26 (portGrantB[3], portRequest[3], i), gate27 (carryBit[9], i, ~portRequest[3]);
or  gate28 (j, carryBit[9], portPriority[4]);
```

```
and gate29 (portGrantB[4], portRequest[4], j), gate30 (carryBit[10], j, ~portRequest[4]);

or gate31 (portGrant[0], portGrantA[0], portGrantB[0]);
or gate32 (portGrant[1], portGrantA[1], portGrantB[1]);
or gate33 (portGrant[2], portGrantA[2], portGrantB[2]);
or gate34 (portGrant[3], portGrantA[3], portGrantB[3]);
or gate35 (portGrant[4], portGrantA[4], portGrantB[4]);
*/

endmodule
```

7.1.6 inputUnit.sv

```
module inputUnit

// ------------------------------------------------------------------------------------------------------------------
// The X,Y location of the input unit, the number of X and Y nodes and the width and the depth of the FIFO are paramatised.
// The input unit has two modes, mode 0 when used for a 2D Mesh and mode 1 when used for a 2D Tori and is used to
// ensure the correct routing method is used.
// ------------------------------------------------------------------------------------------------------------------

  #(parameter xLocation = 0,
    parameter yLocation = 0,
    parameter X_NODES = 3,
    parameter Y_NODES = 3,
    parameter FIFO_WIDTH = 64,
    parameter FIFO_DEPTH = 4,
    parameter MODE = 0)


// ------------------------------------------------------------------------------------------------------------------
//   I/O   Type     Packed Size        Name             Unpacked Size                          Description
// ------------------------------------------------------------------------------------------------------------------

   (output logic  [FIFO_WIDTH-1:0] bufferDataOut,                        // - Data that has been read from the buffer
    output logic             [4:0] outputPortRequest,                    // - Requests for the first flit in memory
    output logic                   holdPortOut,                          // - Signals when the buffer is full

     input logic  [FIFO_WIDTH-1:0] bufferDataIn,                         // - Data at the input port to be written to mem
     input logic                   read, write,
     input logic                   reset, clk);

           logic  [FIFO_WIDTH-1:0] FIFO                [0:FIFO_DEPTH-1];  // - Memory array of 'MEM_DEPTH', 'MEM_WIDTH'
                                                                          // bit words.
           logic             [4:0] xDestination;                         // - X and Y destination of the flit stored at
           logic             [4:0] yDestination;                         // the front of the buffer.
```

```
// ------------------------------------------------------------------------------------------------------------------------
// Function to calculate an approximation of log2(n).  The result provides the amount of bits necessary to reproduce an
// integer as an unsigned binary.
// ------------------------------------------------------------------------------------------------------------------------

function int log2
  (input int n);
    begin
      log2 = 0;     // log2 is zero to start
      n--;          // decrement 'n'
      while (n > 0) // While n is greater than 0
        begin
          log2++;   // Increment 'log2'
          n >>= 1;  // Bitwise shift 'n' to the right by one position
        end
    end
endfunction


// ------------------------------------------------------------------------------------------------------------------------
// Control Flags (read_ptr, write_ptr, nearly, full, empty).  The read and write pointers are calculated using a binary
// representation for each cell.  Each pointer has one more significant bit than required.  This enables comparison to
// check if the FIFO is full or empty. If the write and read pointers including their MSB are equal, the FIFO is empty.
// This same calculation is used to calculate if nearly full by the nearly pointer being one ahead of the write pointer
// ------------------------------------------------------------------------------------------------------------------------

logic [log2(FIFO_DEPTH):0] read_ptr, write_ptr, nearly_ptr;          // Note the inclusion of an extra significant bit.

assign nearly_ptr = write_ptr+1;
assign nearly  = (((nearly_ptr[log2(FIFO_DEPTH)-1:0]) == read_ptr[log2(FIFO_DEPTH)-1:0])
                    && (nearly_ptr[log2(FIFO_DEPTH)]) ^ read_ptr[log2(FIFO_DEPTH)]);
assign full  = (((write_ptr[log2(FIFO_DEPTH)-1:0]) == read_ptr[log2(FIFO_DEPTH)-1:0])
                    && (write_ptr[log2(FIFO_DEPTH)]) ^ read_ptr[log2(FIFO_DEPTH)]);
assign empty = (write_ptr == read_ptr) || (FIFO[read_ptr[log2(FIFO_DEPTH)-1:0]][FIFO_WIDTH-1] == 0);
assign holdPortOut = (nearly || full);
```

```
// ------------------------------------------------------------------------------------------------------------------------
// Memory Read and Write on the rising edge of the clock, with write acknowledgement commented out as not used.
// ------------------------------------------------------------------------------------------------------------------------

  always_ff @ (posedge clk)
    begin

      if (read && ~empty)                                          // If the read signal is asserted and the FIFO is not empty
        begin
        bufferDataOut <= FIFO[read_ptr[log2(FIFO_DEPTH)-1:0]];  // dataOut will be read from the memory
        read_ptr++;                                              // The read_ptr is incremented, truncating if to big.
        end

   if (write && ~full)                                           // If the write signal is asserted and the FIFO is not full
        begin
        FIFO[write_ptr[log2(FIFO_DEPTH)-1:0]] <= bufferDataIn;  // dataIn will be written to memory
        write_ptr <= write_ptr+1;                                // The write pointer is incremented, truncating if to big.
        end

    end

// ------------------------------------------------------------------------------------------------------------------------
// Route calculation of first bit in memory queue.  This Assuming each flit has its destination encoded into the 6 most
// significant bits in the form of the binary x location and y location of a 2D mesh.  The bottom left node denoted
// as 000000.  This could be improved by using gray code for the node reference and including flit types.  Also, it could
// be possible to use a generate case so that input units in the corners and edges of the mesh do not have all the
// calculations as they have less possible directions.
// The form of routing is dimension ordered, the calculation takes place by first checking if there is data in the fifo.
// If there is then the calculation reads the x and y destination contained in the flit header and compares this with the
// current x and y location issuing port requests in the x direction until matched, then the y direction.  The one-hot
// request is represented using 5 bits {Local, West, South, East, North}.  When no data is in the FIFO, no request is made.
// ------------------------------------------------------------------------------------------------------------------------

assign xDestination = FIFO[read_ptr[log2(FIFO_DEPTH)-1:0]][FIFO_WIDTH-2:FIFO_WIDTH-6];
assign yDestination = FIFO[read_ptr[log2(FIFO_DEPTH)-1:0]][FIFO_WIDTH-7:FIFO_WIDTH-11];

always_comb
  begin

  // Mode 0 XY routing for 2D Mesh

    if (~empty && (MODE==0))
      begin
        if (xDestination != xLocation)
```

```
        outputPortRequest <= (xDestination > xLocation) ? 5'b00010 : 5'b01000;
      else if (yDestination != yLocation)
        outputPortRequest <= (yDestination > yLocation) ? 5'b00001 : 5'b00100;
      else
        outputPortRequest <= 5'b10000;
    end


// Mode 1 XY routing for 2D Tori

  else if (~empty && (MODE == 1))
    begin
      if (xDestination != xLocation)
        begin
          if (xDestination < xLocation)
            outputPortRequest <= ((xLocation - xDestination) < ((X_NODES-xLocation)+(xDestination))) ? 5'b01000 : 5'b00010;
          if (xDestination > xLocation)
            outputPortRequest <= ((xDestination - xLocation) < ((X_NODES-xDestination)+(xLocation))) ? 5'b00010 : 5'b01000;
        end
      else if (yDestination != yLocation)
        begin
          if (yDestination < yLocation)
            outputPortRequest <= ((yLocation - yDestination) < ((Y_NODES-yLocation)+(yDestination))) ? 5'b00100 : 5'b00001;
          if (yDestination > yLocation)
            outputPortRequest <= ((yDestination - yLocation) < ((Y_NODES-yDestination)+(yLocation))) ? 5'b00001 : 5'b00100;
        end
      else outputPortRequest <= 5'b10000;
    end


// When empty no port is requested

  else outputPortRequest <= 5'b00000;
end
```

```
// ------------------------------------------------------------------------------------------------------------------------
// Reset everything.
// ------------------------------------------------------------------------------------------------------------------------

always_ff@(posedge clk)
  begin
    if (reset)
      begin
        write_ptr <= 'b0;
        read_ptr <= 'b0;
        bufferDataOut <= 'b0;
        bufferDataIn <= 'b0;
        read <= 'b0;
        write <= 'b0;
      end
    end

endmodule
```

7.1.7 networkTest.sv

```
module networkTest

  #(parameter X_NODES = 8,
    parameter Y_NODES = 8,
    parameter FIFO_WIDTH = 64,
    parameter FIFO_DEPTH = 4,
    parameter MODE = 1,
    parameter N1 = 0,
    parameter N2 = 1,
    parameter N3 = 0,
    parameter r  = 1,
    parameter SAMPLE_TIME = 10,
    parameter int SAMPLE_SIZE = ((N2*X_NODES*Y_NODES*r)/25),
    parameter SAMPLE_NUMBER = 30,
    parameter RUN_NUMBER = 0);

// ------------------------------------------------------------------------------
// Function to calculate an approximation of log2(n)
// ------------------------------------------------------------------------------

function int log2(input int n);
  begin
    log2 = 0;     // log2 is zero to start
    n--;          // decrement 'n'
    while (n > 0) // While n is greater than 0
      begin
        log2++;   // Increment 'log2'
        n >>= 1;  // Bitwise shift 'n' to the right by one position
      end
  end
endfunction

// ------------------------------------------------------------------------------
// Declare the structure of the flits that the network takes.
// ------------------------------------------------------------------------------

typedef struct{

       logic        Valid;        // First bit 1 for valid
  rand logic  [4:0] xDestination; // X destination in next five bits
  rand logic  [4:0] yDestination; // Y destination in next five bits
```

```
        logic  [9:0] nodeOrigin;   // stores which node the flit was sent from
        logic [42:0] data;         // Data in rest

} flit_test;


// ------------------------------------------------------------------------------
// Declare the class NOC_Message to contain methods to be performed on
// the flits such as writing data and creating random destinations.
// ------------------------------------------------------------------------------

class NOC_Message;

  // Class Members

  rand flit_test message;
  rand int    xseed;
  rand int    yseed;


  // Class construct

  function new (int i);
    begin
      xseed = $random(i+(RUN_NUMBER*2));
      yseed = $random(i+10+(RUN_NUMBER*3));
    end
  endfunction


  // class method to set data

  task set_data (logic newData);
    message.data = newData;
  endtask


  // class method to randomise x and y destination within constraints.
  // this is needed as the randomize() function in SystemVerilog is not
  // available in the student version.

  task randomise();
    begin
    message.xDestination = $dist_uniform(xseed,0,(X_NODES-1));
```

```
    message.yDestination = $dist_uniform(yseed,0,(Y_NODES-1));
    end
  endtask

endclass: NOC_Message


// ------------------------------------------------------------------------------
//                              Module logic
// ------------------------------------------------------------------------------
//  TYPE  packed size    Name                                  Unpacked Size
// ------------------------------------------------------------------------------

  logic                  reset;
  logic                  clk;
  logic                  valid;
  logic            [4:0] xDestination                          [(X_NODES*Y_NODES)];
  logic            [4:0] yDestination                          [(X_NODES*Y_NODES)];
  logic            [9:0] nodeOrigin                            [(X_NODES*Y_NODES)];
  logic           [42:0] data;
  logic [FIFO_WIDTH-1:0] randomData                            [(X_NODES*Y_NODES)];
  logic [FIFO_WIDTH-1:0] nodeToNetworkDataQueue                [(X_NODES*Y_NODES)][$];
  logic [FIFO_WIDTH-1:0] nodeToNetworkData                     [(X_NODES*Y_NODES)];
  logic [FIFO_WIDTH-1:0] networkToNodeData                     [(X_NODES*Y_NODES)];
  logic                  nodeToNetworkWriteRequest             [(X_NODES*Y_NODES)];
  logic                  nodeToNetworkHoldRequest              [(X_NODES*Y_NODES)];
  logic                  networkToNodeWriteRequest             [(X_NODES*Y_NODES)];
  logic                  networkToNodeHoldRequest              [(X_NODES*Y_NODES)];
  logic [FIFO_WIDTH-1:0] networkData                           [(X_NODES*Y_NODES)][0:4];
  logic [FIFO_WIDTH-1:0] routerInputData                       [(X_NODES*Y_NODES)][0:4];
  logic           [42:0] cycle;
  int                    countTotal;
  int                    countRecievedFromN2;
  int                    countRecievedFromNode                 [(X_NODES*Y_NODES)];
  int                    countThroughput;
  int                    xDestinationFlag                      [(X_NODES*Y_NODES)];
  int                    yDestinationFlag                      [(X_NODES*Y_NODES)];
  int                    dataFlag                               [(X_NODES*Y_NODES)];
  int                    networkDataXDestinationFlag           [(X_NODES*Y_NODES)][0:4];
  int                    networkDataYDestinationFlag           [(X_NODES*Y_NODES)][0:4];
  int                    networkDataDataFlag                   [(X_NODES*Y_NODES)][0:4];
  int                    networkDataOriginFlag                 [(X_NODES*Y_NODES)][0:4];
  int                    routerInputDataXDestinationFlag       [(X_NODES*Y_NODES)][0:4];
  int                    routerInputDataYDestinationFlag       [(X_NODES*Y_NODES)][0:4];
```

```
  int                     routerInputDataDataFlag            [(X_NODES*Y_NODES)][0:4];
  int                     routerInputDataOriginFlag          [(X_NODES*Y_NODES)][0:4];
  int                     queueSize                          [(X_NODES*Y_NODES)];
  logic           [N2:1]  recieved                           [X_NODES*Y_NODES];
  int                     injection;
  int                     injectionSeed;
  int                     packetInN1, packetInN2, packetInN3;
  int                     nodeTotalLatency                   [X_NODES*Y_NODES];
  int                     totalLatency, averageLatency;
  int                     batchAverageLatencyA               [SAMPLE_NUMBER];
  int                     batchAverageLatencyB               [SAMPLE_NUMBER];
  int                     batchNumberA, batchNumberB;
  int                     latencyCountedA, latencyCountedB;
  int                     recievedCountedA, recievedCountedB;
  int                     ssef, latencyFile, throughputFile;


// ------------------------------------------------------------------------------
// Cycle and Clock pulse creation.  Clock pulse rising edge every 100ps.
// ------------------------------------------------------------------------------

initial
  begin
    clk = 1;
    forever #0.5ns clk = ~clk;
  end

initial
  begin
    cycle = 0;
    forever #1ns cycle = cycle+1;
  end


// ------------------------------------------------------------------------------
// Creation of a network and reseting.
// ------------------------------------------------------------------------------

network #(X_NODES, Y_NODES, FIFO_WIDTH, FIFO_DEPTH, MODE) networktest (networkToNodeData, networkToNodeWriteRequest,
networkToNodeHoldRequest, networkData, routerInputData, nodeToNetworkData, nodeToNetworkWriteRequest, nodeToNetworkHoldRequest,
reset, clk);

initial
  begin
```

```
    reset = 1'b1;
    for (int i = 0; i<X_NODES*Y_NODES; i++)
      begin
        nodeToNetworkHoldRequest[i] = 0;
        nodeToNetworkWriteRequest[i] = 0;
      end
    #50ps
    reset= 1'b0;
  end


// ------------------------------------------------------------------------------
// Creation of randomish input data.  Depending on the injection rate 'r', on each
// clock pulse, new data might be created of the class type NOC_Message,
// using the flit_test structure.  The x and y destination is randomised and and the
// node of origin and cycle stored then inserted into a variable length queue to hold
// the data until the network is ready.  This block utilises the SystemVerilog queue
// function push_back().  There are three distinct stages, warm up, steady state and
// drain.
// ------------------------------------------------------------------------------

initial
  begin
    NOC_Message test_message[0:(X_NODES*Y_NODES)-1];
    injectionSeed = $random;
    for (int i = 0; i < (X_NODES*Y_NODES); i++)
      begin
        test_message[i] = new(i);
      end


    // Warm up

    repeat (N1)
      begin
        #1ns
        for (int i = 0; i < X_NODES*Y_NODES; i++)
          begin
            injection = $dist_uniform(injectionSeed, 0, 100);
            if (injection < (r*100)+1)
              begin
              test_message[i].randomise();
              valid           = 1'b1;
              xDestination[i] = test_message[i].message.xDestination;
```

```
            yDestination[i] = test_message[i].message.yDestination;
            data            = '0;
            nodeOrigin[i]   = i;
            randomData[i]  = {valid, xDestination[i], yDestination[i], nodeOrigin[i], data};
            nodeToNetworkDataQueue[i].push_back(randomData[i]);
            packetInN1++;
            end
        end
    end


  // Steady State (during which throughput is calculated)

  repeat (N2)
    begin
      #1ns
      for (int i = 0; i < X_NODES*Y_NODES; i++)
        begin
          countThroughput = networkToNodeWriteRequest[i] ? countThroughput+1 : countThroughput;
          injection = $dist_uniform(injectionSeed, 0, 100);
          if (injection < (r*100)+1)
            begin
            test_message[i].randomise();
            valid           = 1'b1;
            xDestination[i] = test_message[i].message.xDestination;
            yDestination[i] = test_message[i].message.yDestination;
            data            = cycle;
            nodeOrigin[i]   = i;
            randomData[i]  = {valid, xDestination[i], yDestination[i], nodeOrigin[i], data};
            nodeToNetworkDataQueue[i].push_back(randomData[i]);
            packetInN2++;
            end
        end
    end

  throughputFile = $fopen("throughput.txt","a");
  $fdisplay(throughputFile, "%d, %d, %d, %d, %d, %d, %d, %d, %f, %d, %d",X_NODES, Y_NODES, FIFO_WIDTH, FIFO_DEPTH, MODE, N1,
N2, N3, r, packetInN2, countThroughput);
  $fclose(throughputFile);

  // Drain

  repeat (N3)
    begin
```

```
        #1ns
        for (int i = 0; i < X_NODES*Y_NODES; i++)
          begin
            injection = $dist_uniform(injectionSeed, 0, 100);
            if (injection < (r*100)+1)
              begin
              test_message[i].randomise();
              valid           = 1'b1;
              xDestination[i] = test_message[i].message.xDestination;
              yDestination[i] = test_message[i].message.yDestination;
              data            = 0;
              nodeOrigin[i]   = i;
              randomData[i]  = {valid, xDestination[i], yDestination[i], nodeOrigin[i], data};
              nodeToNetworkDataQueue[i].push_back(randomData[i]);
              packetInN3++;
              end
          end
      end

    latencyFile = $fopen("latencyFile.txt","a");
    $fwrite(latencyFile, "%d, %d, %d, %d", packetInN2, countRecievedFromN2, totalLatency, averageLatency);
    $fclose(latencyFile);

  end

// ------------------------------------------------------------------------------
// On each clock pulse, write Data to network provided there is data in the queue and
// that there is space in the input unit.  This block utilises the SystemVerilog
// queue functions .size() and .pop_front().
// ------------------------------------------------------------------------------

always_ff@(posedge clk)
  begin
    for (int i = 0; i < X_NODES*Y_NODES; i++)
      begin
        if ((nodeToNetworkDataQueue[i].size() != 0) && (networkToNodeHoldRequest[i] != 1))
          begin
            nodeToNetworkWriteRequest[i] <= 1'b1;
            nodeToNetworkData[i] <= nodeToNetworkDataQueue[i].pop_front();
          end
        else nodeToNetworkWriteRequest[i] <= 1'b0;
      end
  end
```

```
// ------------------------------------------------------------------------------
// Read incoming data and make the various calculations neccessary to gain results.
// This is independent of number of cycles designated for each phase of test.
// ------------------------------------------------------------------------------

// Count total flits recieved from the network

always@(posedge clk)
  begin
    for (int i=0; i < X_NODES*Y_NODES; i++)
      begin
        countTotal = networkToNodeWriteRequest[i] ? countTotal+1 : countTotal;
      end
  end

// Count total flits recieved from each individual node origin

always@(posedge clk)
  for (int j = 0; j < X_NODES*Y_NODES; j++)
    begin
      for (int k = 0; k < X_NODES*Y_NODES; k++)
        countRecievedFromNode[j] = ((networkToNodeData[k][FIFO_WIDTH-12:FIFO_WIDTH-21] == j) && (networkToNodeData[k][FIFO_WIDTH-
1] == 1)) ? countRecievedFromNode[j] + 1 : countRecievedFromNode[j];
    end


// Indicate wether a flit has been recieved from each cycle for each node
// (currently only useful when r=1)

always@(posedge clk)
  for (int i = 0; i < X_NODES*Y_NODES; i++)
    begin
      for (int k = 0; k < X_NODES*Y_NODES; k++)
        begin
          if (networkToNodeData[i][FIFO_WIDTH-12:FIFO_WIDTH-21] == k)
            recieved[k][networkToNodeData[i][FIFO_WIDTH-22:0]] = 1 ;
        end
    end


// Calculate average latency of all packets entering the network during N2

always@(posedge clk)
  begin
```

```
    for (int i=0; i < X_NODES*Y_NODES; i++)
      begin
        if (networkToNodeWriteRequest[i] == 1)
          nodeTotalLatency[i] = (networkToNodeData[i][FIFO_WIDTH-22:0] != 0) ? ((cycle-1) - networkToNodeData[i][FIFO_WIDTH-
22:0]) + nodeTotalLatency[i] : nodeTotalLatency[i];
          totalLatency = nodeTotalLatency.sum();
          averageLatency = totalLatency / packetInN2;
      end
  end


// Calculate average latency of packets entering the network during N2 in batches
// according to amount of arrivals

always@(posedge clk)
  begin
    ssef = $fopen("steadyStateEstimateFile.txt", "a");
    latencyFile = $fopen("latencyFile.txt","a");
    if (cycle == 1)
      begin
        $fdisplay(ssef, " ");
        $fwrite(ssef, "run %d,", RUN_NUMBER);
        $fdisplay(latencyFile, " ");
        $fwrite(latencyFile, "%d, %d, %d, %d, %d, %d, %d, %d, %f,",X_NODES, Y_NODES, FIFO_WIDTH, FIFO_DEPTH, MODE, N1, N2, N3,
r);
      end
    for (int i=0; i < X_NODES*Y_NODES; i++)
      begin
        countRecievedFromN2 = ((networkToNodeWriteRequest[i] == 1) && (networkToNodeData[i][FIFO_WIDTH-22:0] != 0)) ?
countRecievedFromN2+1 : countRecievedFromN2;
        if ((countRecievedFromN2 % SAMPLE_SIZE == 0) && (countRecievedFromN2 != 0) && (totalLatency != latencyCountedA) &&
(batchNumberA < SAMPLE_NUMBER))
          begin
            batchAverageLatencyA[batchNumberA] = ((totalLatency - latencyCountedA) / (countRecievedFromN2 - recievedCountedA));
            if (batchAverageLatencyA[batchNumberA] != 0)
              begin
                $fwrite(ssef, "%d,", batchAverageLatencyA[batchNumberA]);
                $fwrite(latencyFile, "%d,", batchAverageLatencyA[batchNumberA]);
              end
            latencyCountedA = totalLatency;
            recievedCountedA = countRecievedFromN2;
            batchNumberA = (recievedCountedA / SAMPLE_SIZE);
          end
        if (countRecievedFromN2 == packetInN2)
```

```
          begin
            batchAverageLatencyA[batchNumberA] = ((totalLatency - latencyCountedA) / (countRecievedFromN2 - recievedCountedA));
            if (batchAverageLatencyA[batchNumberA] != 0)
              begin
                $fwrite(latencyFile, "%d,", batchAverageLatencyA[batchNumberA]);
              end
            latencyCountedA = totalLatency;
            recievedCountedA = countRecievedFromN2;
            batchNumberA = (recievedCountedA / SAMPLE_SIZE);
          end
      end
    $fclose(latencyFile);
    $fclose(ssef);
  end


// Calculate average latency of packets entering the network during N2 in batches
// according to cycles

always@(posedge clk)
  begin
    if ((cycle % SAMPLE_TIME == 0) && (totalLatency != 0) && (countRecievedFromN2 != packetInN2))
      begin
        batchAverageLatencyB[batchNumberB] = ((totalLatency - latencyCountedB) / (countRecievedFromN2 - recievedCountedB));
        latencyCountedB = totalLatency;
        recievedCountedB = countRecievedFromN2;
        batchNumberB = batchNumberB +1;
      end
  end


// ------------------------------------------------------------------------------------
// Creation of flags for deciphering complex simulation wave windows during debug
// ------------------------------------------------------------------------------------

initial
  begin
    #0.5ns // stalls the calculations so they happen mid clock cycle.
    forever #1ns
      begin
        for (int j = 0; j < X_NODES*Y_NODES; j++)
          begin
            xDestinationFlag[j] = (nodeToNetworkData[j][FIFO_WIDTH-1] == 1) ? nodeToNetworkData[j][FIFO_WIDTH-2:FIFO_WIDTH-6] :9;
```

```
            yDestinationFlag[j] = (nodeToNetworkData[j][FIFO_WIDTH-1] == 1) ? nodeToNetworkData[j][FIFO_WIDTH-7:FIFO_WIDTH-11]
:9;
            dataFlag[j] = (nodeToNetworkData[j][FIFO_WIDTH-1] == 1) ? nodeToNetworkData[j][FIFO_WIDTH-22:0] : 0;
            queueSize[j] = nodeToNetworkDataQueue[j].size();
            for (int k = 0; k < 5; k++)
              begin
                networkDataXDestinationFlag[j][k] = (networkData[j][k][FIFO_WIDTH-1] == 1) ? networkData[j][k][FIFO_WIDTH-
2:FIFO_WIDTH-6] :9;
                networkDataYDestinationFlag[j][k] = (networkData[j][k][FIFO_WIDTH-1] == 1) ? networkData[j][k][FIFO_WIDTH-
7:FIFO_WIDTH-11] :9;
                networkDataOriginFlag[j][k] = (networkData[j][k][FIFO_WIDTH-1] == 1) ? networkData[j][k][FIFO_WIDTH-
12:FIFO_WIDTH-21] : 9;
                networkDataDataFlag[j][k] = (networkData[j][k][FIFO_WIDTH-1] == 1) ? networkData[j][k][FIFO_WIDTH-22:0] : 0;

                routerInputDataXDestinationFlag[j][k] = (routerInputData[j][k][FIFO_WIDTH-1] == 1) ?
routerInputData[j][k][FIFO_WIDTH-2:FIFO_WIDTH-6] :9;
                routerInputDataYDestinationFlag[j][k] = (routerInputData[j][k][FIFO_WIDTH-1] == 1) ?
routerInputData[j][k][FIFO_WIDTH-7:FIFO_WIDTH-11] :9;
                routerInputDataOriginFlag[j][k] = (routerInputData[j][k][FIFO_WIDTH-1] == 1) ? routerInputData[j][k][FIFO_WIDTH-
12:FIFO_WIDTH-21] : 9;
                routerInputDataDataFlag[j][k] = (routerInputData[j][k][FIFO_WIDTH-1] == 1) ? routerInputData[j][k][FIFO_WIDTH-
22:0] : 0;
              end
          end
      end
  end

endmodule
```

8. References

[1] Intel Newsroom, 2011.*The Future Accelerated: Multi-Core Goes Mainstream, Computing Pushed to Extremes*[ONLINE] Available at: <http://newsroom.intel.com/community/intel_newsroom/blog/2011/09/15/the-future-accelerated-multi-core-goes-mainstream-computing-pushed-to-extremes> [Accessed 02 October 2011].

[2] EETimes, 2011.*Gartner expands technology 'hype' curve in 2011*[ONLINE] Available at: <http://www.eetimes.com/electronics-news/4218627/Gartner-expands-technology--hype--curve-in-2011> [Accessed 02 October 2011].

[3] Arteris, 2011.*NoC Interconnect Technology Becoming Mainstream* [ONLINE] Available at: <http://info.arteris.com/blog/bid/71294/NoC-Interconnect-Technology-Becoming-Mainstream> [Accessed 02 October 2011].

[4] Arteris, 2005.*A comparison of Network-on-Chip and Busses.*[ONLINE] Available at: <http://www.design-reuse.com/articles/10496/a-comparison-of-network-on-chip-and-busses.html> [Accessed 02 October 2011].

[5]Saxena, P. Shelar, R.S. Sapatnekar, S., 2010. *Routing Congestion in VLSI Circuits: Estimation and Optimization (Integrated Circuits and Systems).*Softcover reprint of hardcover 1st ed. 2007 Edition. Springer.

[6]2011.*Third Year Projects - Guidelines for Students.*[ONLINE] Available at: <http://www.ee.ucl.ac.uk/~afernand/projects/3rd-Year/ProjGuideStudents.html> [Accessed 02 October 2011].

[7] EETimes, 2011.*Application convergence.* [ONLINE] Available at: <http://www.eetimes.com/electronics-news/4048718/Application-convergence> [Accessed 02 October 2011].

[8] Borkar, S. 2007. *Thousand Core Chips – A technology perspective*. [ONLINE] Available at: <http://dl.acm.org/citation.cfm?id=1278667>[Accessed 02 October 2011].

[9] Jerger, N.E. Peh, L., 2009. *On-Chip Networks (Synthesis Lectures on Computer Architecture).*1st Edition. Morgan and Claypool Publishers.

[10] Objective Analysis, 2011.*NoC Interconnect Improves SoC Economics.* [ONLINE] Available at: <http://info.arteris.com/download-soc-economics-research-paper/>[Accessed 02 October 2011].

[11] Microsoft Research, 2011. *BEE3* [ONLINE] Available at: <http://research.microsoft.com/en-us/projects/bee3/>[Accessed 02 October 2011].

[12] Princton University, 2009. *What is PARSEC?* [ONLINE] Available at: <http://parsec.cs.princeton.edu/overview.htm> {Accessed 02 October 2011]

[13] Nicholls, D., 2011.*Network-on-Chip for a Multicore Computer Project Proposal.*

[14] Dally, W.J. and Towles, B., 2004. *Principles and Practices of Interconnection Networks.*Morgan Kaufmann.

[15] Pande, P. P., et al., 2005. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8) pp.1025-1040.

[16] Balfour, J. and Dally, W. J., 2006. Design tradeoffs for Tiled CMP On-Chip Networks. *Proceedings of the International Conference on Supercomputing*, pp.187-198.

[17] Agarwal, A., Iskander, C. and Shankar, R., 2009. Survey of Network on Chip (NoC) Architectures & Contributions.*Journal of Engineering, Computing and Architecture*, 3(1).

[18]Wentzlaff, D., et al., 2007. On-chip Interconnection Architecture of the Tile Processor.*IEEE Micro*, 27(5), pp.15-31.

[19] Camacho, J., et al., 2011. A Power-Efficient Network On-Chip Topology.*Proceedings of the Fifth International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip*, pp.23-26.

[20] Neeb, C., Thul, M. and When N., 2005. Network on-chip-centric approach to interleaving in high throughput channel decoders. *Proceedings of the IEEE International Symposium on Circuits and Systems,* pp.1766-1769.

[21] Flich, J., Rodrigo, S., and Duato, J., 2008. An Efficient Implementation of Distributed Routing Algorithms for NoCs.*Second ACM/IEEE International Symposium on Networks-on-Chip,* pp.87-96.

[22] Rodrigo, S., Flich, J., Duato, J., Hummel, M., 2008. Efficient Unicast and Multicast Support for CMP. *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture,* pp.364-375.

[23] Karol, M., Hluchyj, M. and Morgan, S., 1987. Input Versus Output Queuing on a Space-Division Packet Switch.*IEEE Transactions on Communications,* 35(12), pp.1347-1356.

[24] Hoskote, Y et al., 2007. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro* 27(5), pp.51-61.

[25] Batten, C. et al., 2008. Building Manycore processor-to-dram networks with monolithic silicon photonics. *Proceedings of the 35th international Symposium on High Performance Interconnects.*

[26] Wang, W. and Jao, Z., 2011.Design of a Partially buffered crossbar router for mesh-based Network-on-Chips.*2011 IEEE International conference on High Performance Computing and Communications,* pp.722-777.

[27] Zhang, Y., Morris, R. and Kodi, A.K., 2011. Design of a performance enhanced and power reduced dual-crossbar Network-on-Chip (NoC) architecture. *Microprocessors and Microsystems 35,* 35(1), pp.110-118.

[28] IEEE Standards, 2011. 1800-2009 – *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language* [ONLINE] Available at: <http://standards.ieee.org/findstds/standard/1800-2009.html> {Accessed 27 November 2012}

9. Contact Information

Email : danny.nicholls.09@ucl.ac.uk

Phone : 07834826112