

University of Windsor

Scholarship at UWindor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

2010

NoC Prototyping on FPGAs: Component Design, Architecture Implementation and Comparison

Matt Murawski

University of Windsor

Follow this and additional works at: <https://scholar.uwindsor.ca/etd>

Recommended Citation

Murawski, Matt, "NoC Prototyping on FPGAs: Component Design, Architecture Implementation and Comparison" (2010). *Electronic Theses and Dissertations*. 134.

<https://scholar.uwindsor.ca/etd/134>

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

NoC Prototyping on FPGAs: Component Design, Architecture Implementation and
Comparison

By

Matt Murawski

A Thesis

Submitted to the Faculty of Graduate Studies
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2010

© 2010 Matt Murawski

All Rights Reserved. No Part of this document may be reproduced, stored or otherwise retained in a retrieval system or transmitted in any form, on any medium by any means without prior written permission of the author

NoC Prototyping on FPGAs: Component Design, Architecture Implementation and
Comparison

By
Matt Murawski

APPROVED BY:

A.C. Sodan
Computer Science

J. Wu
Electrical and Computer Engineering

M. A. S Khalid, Advisor
Electrical and Computer Engineering

Dr. Rashidzadeh, Chair of Defense
Electrical and Computer Engineering

May 18, 2010

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Continuing improvements in integrated circuit technology over the past few decades enables increasingly large and complex Systems-on-Chip. Due to the large number of components used, the traditional bus-based interconnect scheme becomes cumbersome and restrictive. Hence, the Network-on-Chip interconnect paradigm becomes appealing due to its many advantages such as scalability and superior performance. Much research remains to be done exploring NoC architectures using real world benchmarks. In this thesis we describe the design space exploration of two major NoC components; a flexible adapter based on the Altera Avalon standard and a parameterizable wormhole router. Two well known NoC architectures, torus and ring, were synthesized for Altera FPGAs using these NoC components. The architectures were compared on the basis of packet latency, area and throughput, using a benchmark application. Simulation results show that the ring architecture gives superior area versus performance tradeoffs for the benchmark used.

Acknowledgements

It is an honor for me to have worked with Dr. Mohammed A. S. Khalid throughout my Masters degree here at the University of Windsor. His guidance, encouragement, wisdom and support carried me through the course of this thesis. My deepest gratitude goes out to him.

I would like to extend my gratitude to Dr. Jonathan Wu and Dr. Angela C. Sodan for serving on my committee and providing constructive criticism of this work. Dr. Rashid Rashidzadeh has been of tremendous help regarding CAD tool issues and I'd like to thank him for his patience and support.

My thanks goes out to my office colleagues, Mike, Mohan, Thuan, Krunal, Abdelrazag and Omar, for their friendly attitude and support whenever needed.

My gratitude goes to the ECE Dept. staff members Don Tersigni, Frank Cicchello and Andria Ballo for their help and assistance.

Gratitude goes to my parents and siblings for providing support and encouragement throughout my work. Their understanding has been greatly appreciated. My gratitude also goes to my brother, Peter, for all the cooking and cleaning I neglected to do!

My friends Dan and Nathan deserve acknowledgement. Thanks for the great stress relief of weekly Baldur's Gate and your understanding of Masters student hardships. We had great times in World of Warcraft every Friday with Matt, Mike, Trinh, and Chris. Scott, your friendship, aid and understanding will not go unnoticed.

Lastly, 7-11, thanks for all the pop.

Table of Contents

Author's Declaration of Originality	iv
Abstract	v
Acknowledgements	vi
List of Figures	x
List of Tables	xii
List of Abbreviations	xiii
Chapter 1 Introduction.....	16
1.1 Thesis Objectives	19
1.2 Thesis Organization.....	20
Chapter 2 Background and Previous Work.....	21
2.1 Network-on-Chip Overview.....	21
2.1.1 Application Layer - System.....	23
2.1.2 Transaction Layer – Adapter	24
2.1.3 Data Link Layer – Network.....	25
2.1.4 Physical Layer – Link.....	28
2.2 Standard Sockets	28
2.2.1 Wishbone	28
2.2.2 Avalon Interface	34
2.3 FPGA Technology.....	37

2.4 CAD Tools for NoC Implementation on FPGAs	37
2.4.1 Altera Quartus II	37
2.4.2 Altera SOPC Builder	38
2.4.3 Nios II Embedded Design Suite (EDS)	38
2.4.4 Mentor Graphics ModelSim	39
2.5 Related Work	39
2.6 Summary	41
Chapter 3 NoC Adapter and Router Design	42
3.1 Adapter Overview	42
3.2 Router Overview	49
3.3 Summary	53
Chapter 4 NoC Implementation and Evaluation Framework	54
4.1 Multi-CPU Benchmark System	54
4.2 Implementation of NoC in SOPC Builder	55
4.3 Nios II Programming	58
4.4 Modelsim Simulation Environment	59
4.5 Summary	60
Chapter 5 FPGA Implementation of Torus and Ring NoC Architectures	61
5.1 Topology	61
5.2 Placement and Routing	63
5.3 NoC Generator	66
5.4 Summary	68
Chapter 6 Component Evaluation and Architecture Comparison	69
6.1 Design Space Exploration of Adapter and Router	69
6.2 Experimental Framework and Evaluation Metrics	73

6.3 Comparison of Torus and Ring	74
6.4 Summary	80
Chapter 7 Conclusions and Future Work	81
References	84
VITA AUCTORIS	89

List of Figures

Figure 1 - NoC Component Overview	17
Figure 2 - NoC Example System	22
Figure 3 – NoC OSI Layers	23
Figure 4 - Components Used In NoC System.....	23
Figure 5 - Network Adapter[4]	25
Figure 6 - Example of Deadlock	27
Figure 7 - Single Transfer Handshaking Protocol for Wishbone	30
Figure 8 - Single Read Transfer for Wishbone	30
Figure 9 - Single Write Request for Wishbone.....	31
Figure 10 - Wishbone RMW.....	32
Figure 11 - Block Read Request for Wishbone	33
Figure 12 - Incrementing Bursts for Wishbone	34
Figure 13 - Example Avalon-MM Transfer.....	36
Figure 14 - Quartus II Design Flow	38
Figure 15 - Adapter Overview	43
Figure 16 - Adapter Design Overview.....	45
Figure 17 - Address to Destination Parameter Design.....	46
Figure 18 - Avalon-Wishbone Glue Logic for Master.....	46
Figure 19 - Master Sampler 3-Concurrent Process Flowchart in Burst Mode	48
Figure 20 - Slave Sampler 3-Concurrent Process Flowchart in Burst Mode.....	49
Figure 21 - Router Overview Diagram	50
Figure 22 - Internal Router Design	51
Figure 23 - Priority Table Design	52
Figure 24 - NoC Parameters in SOPC Builder	56

Figure 25 - Torus NoC Implemented in SOPC Builder.....	56
Figure 26 - Torus NoC Connected to Slave Components.....	57
Figure 27 - Bridging Example	57
Figure 28 - CPU 1 Benchmark Flowchart	58
Figure 29 - CPU 2 and 3 Benchmark Flowchart.....	59
Figure 30 - Router Regional Handshaking	60
Figure 31 - Mesh vs. Torus	62
Figure 32 - 10-Node Ring Example.....	62
Figure 33 - Torus Core Placement	63
Figure 34 - Placement for Routing.....	64
Figure 35 - Torus CPU Routing.....	65
Figure 36 - Torus Source Routing Paths.....	65
Figure 37 - Ring Placement	66
Figure 38 - Ring Routing	66
Figure 39 - Individual Router Area vs. Flit size	70
Figure 40 - Master Adapter Area	70
Figure 41 - Slave Adapter Area	71
Figure 42 - FIFO Area	71
Figure 43 - Router Area vs. Number of Ports.....	72
Figure 44 - Router Area vs Flit Size	72
Figure 45 - Power Usage of Discrete NoC Components	73
Figure 46 - Clock Frequency of Discrete NoC Components.....	73
Figure 47 - Average Latency of Two NoC Topologies	75
Figure 48 - Total Time to Complete Nios II Program	75
Figure 49 – Throughput Comparison of Two NoC Topologies	76
Figure 50 – Bandwidth Comparison of Two NoC Topologies.....	76
Figure 51 - NoC Area – ALUTs	77
Figure 52 - NoC Area - Registers	77
Figure 53 - Latency Change vs. Area Change, with Respect To 16 Bit Flit Size.....	79
Figure 54 - Throughput Change vs Area Change, With Respect To 16 Bit Flit Size.....	79

List of Tables

Table 1 - Cycle Type Identifier.....	33
Table 2 - Burst Type Extension	34
Table 3 - Request Type Design.....	44
Table 4 - Master-Slave Dependencies	65

List of Abbreviations

Abbreviation	Definition
ALUT	Adaptive Look Up Table
ASIC	Application Specific Integrated Circuit
AVM	Avalon Master
AVS	Avalon Slave
AWB	Avalon-Wishbone
BE	Best Effort
BTE	Burst Type Extension
CAD	Computer Aided Design
CLK	Clock
CPU	Central Processing Unit
CS	Chip Select
CTI	Cycle Tag Identifier
DDR	Double Data Rate
EDS	Embedded Design Suite
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GS	Guaranteed Service
HDL	Hardware Description Language
I/O	Input / Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IP	Intellectual Property
IRQ	Interrupt Request

JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
KB	Kilo Byte
LAN	Local Area Network
LE	Logic Element
LED	Light Emitting Diode
MB	Mega Byte
MPSoC	Multi-Processor System on Chip
NA	Network Adapter
NoC	Network on Chip
OCF	Open Core Protocol
OE	Output Enable
OSI	Open Systems Interconnection
PIO	Peripheral Input Output
PWR	Parameterizable Wormhole Router
QoS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
S&F	Store and Forward
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on Chip
SOPC	System on Programmable Chip
TDM	Time Division Multiplexing
UART	Universal Asynchronous Receiver/Transmitter
VC	Virtual Channel
VCI	Virtual Component Interface
VCT	Virtual Cut Through
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
WB	Wishbone

WBM	Wishbone Master
WBS	Wishbone Slave
WH	Wormhole

Chapter 1

Introduction

As humankind pushes forward through the Information Age, digital technology becomes smaller, and more powerful. It follows without surprise that devices embedded with digital technology are becoming more widespread and common, such as cell phones, digital cameras, and global positioning systems. Such embedded computing systems, or simply “embedded systems”, are structures of electronic hardware designed to perform singular functions repeatedly within tightly constrained design metrics [1].

As Moore’s Law - Integrated Circuit (IC) designs double in capacity nearly every two years - continues to hold true for the past half decade, the design of embedded systems becomes increasingly difficult and complex. To combat this, designers have shifted their focus from micro-level design to macro-level system design through the employment of hardware reuse. This shift in focus is known as System-on-Chip (SoC) and involves interfacing pre-made hardware modules together to form a coherent system. Those hardware blocks are known as Intellectual Property (IP) cores and they vary from Central Processing Units (CPU), to Random Access Memory (RAM) modules, to counters and to other specific logic designs. Flexibility in IP cores is embraced by means of *parameters*, which offer increased compatibility and proficiency to otherwise black-box modules.

In SoCs, interconnect structures between IP cores traditionally use a shared bus design, a point-to-point design or a hybrid thereof. Systems level designers have been facing constriction with bus-based, computation-centric interconnection systems [2].

While bus-based interconnections proved to be sufficient for small embedded systems involving few IP cores, the routing of the bus wires, the increasingly difficult custom interconnect design process (such as [3]) and the decreasing performance due to the increasing amount of cores attached to the bus/buses have shown that a paradigm shift is imminent. Borrowing from macro-network communication architectures, Network-on-Chip (NoC) provides a communication-centric design that shows promise in providing the scalability and performance needed for large SoCs [4].

An NoC consists of four major components – IP cores, the network adapters, routing nodes and links. These are similar to the components in a macro computer network, where an IP core is similar to a desktop computer, the network adapters are the wireless network and LAN cards, the routing nodes are the switches and routers and the links are the physical cables connecting the system together. Figure 1 demonstrates an overview of NoC components in a simple 4x4 mesh topology with one core per routing node. As with macro networks, there are many different architectures, mechanisms, parameters and techniques involved in NoCs and hence, many areas remain open for research.

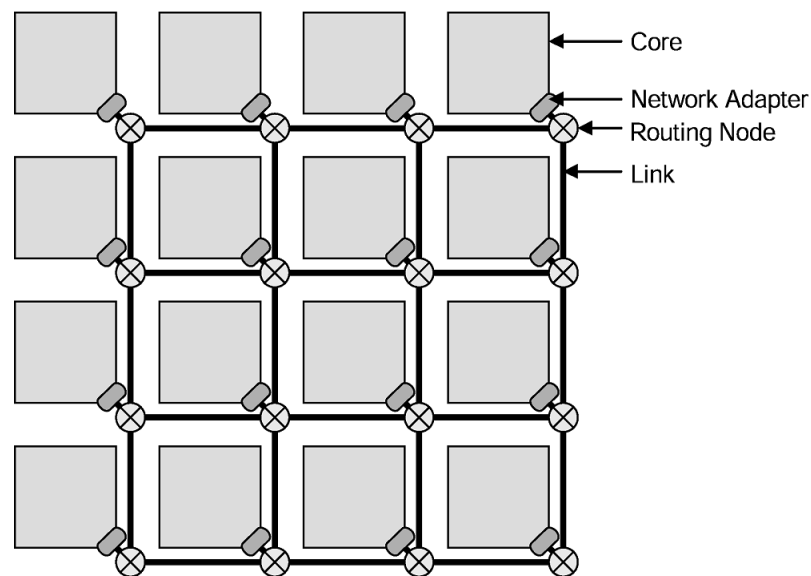


Figure 1 - NoC Component Overview

Numerous NoCs [5] [6] [7] [8] have been proposed to address the growing complexity of SoCs and their communication infrastructure needs. Due to the nature of

NoCs and the systems they are integrated in, designing and simulating test applications is troublesome and time consuming, hence the more favourable *traffic generator* evaluation tool is used. A traffic generator injects artificial distributions of network traffic in order to simulate a realistic system [9]. While traffic generators provide an elegant approach for NoC comparison, it provides a ham-fisted investigation which does not address specific applications and hence misses out on important optimizations due to unique or “bursty” traffic patterns (such as in [10]).

Furthermore, an assortment of NoC parameters, such as topology and channel width, requires additional exploration. The massive design space of NoCs means that even a simple design of an NoC component is a valuable research contribution since it expands on existing understandings and provides reinforcement to existing theories.

Existing IP cores utilize standard socket interfaces such as Wishbone [11], Avalon [12], AMBA [13] and CoreConnect [13], so conforming NoC protocols to these pre-existing bus protocols presents a practical challenge. Thus, the feasibility of instantiating NoCs in existing SoCs and their supporting Integrated Development Environments (IDE) holds as a valuable research topic since it ties theory with practicality.

The requirement for reducing time-to-market for digital designs has quickly led to the creation of Field Programmable Gate Arrays (FPGA). An FPGA is an integrated circuit containing pre-designed resources which can be programmed and configured to act like the desired digital hardware description. This provides a means of a quick prototyping medium for IC designs due to the flexibility of the programmable components, the liberty of manufacturing problems and the fast design cycles. These traits make FPGAs an irreplaceable platform for research purposes.

The task of this thesis is to explore the implementation of a practical NoC. It follows that discrete components need to be created and investigated in order to design an NoC, which includes a network adapter, router and supporting modules. This leads to the study of designing an NoC architecture, the automation of realizing NoC modules with C++ programs, the effects of different NoC parameters on high level NoC and FPGA evaluation metrics and the challenges and issues of using real world SoC software.

1.1 Thesis Objectives

The end goal of this research is to explore the feasibility, implementation, design space exploration and evaluation of a practical NoC, targeting FPGAs. In order to achieve this goal, discrete NoC components need to be designed and made flexible for use in an even further parameterizable NoC system. Thus, the effects on higher level evaluation metrics can be studied against an assortment of NoC parameters. There are several major objectives:

1. Design and evaluate a network adapter to interface a standard socket with the NoC protocol, while providing easy and powerful flexibility through the use of VHDL generics.
2. Design and evaluate a worm-hole router that provides the parameters needed in order to be instantiated within a wide range of NoC architectures.
3. Design the required support modules needed for a functional NoC system.
4. Automate the construction of an NoC architecture using the above modules with a C++ program based on user input.
5. Create a series of NoCs with different topologies and channel widths for evaluation.
6. Instantiate the NoC architectures within a realistic benchmark SoC using existing design and simulation software.
7. Utilize high level evaluation metrics to obtain results in an automatic fashion and without adding additional resources or performance degradation.
8. Synthesize the NoCs targeting an FPGA for area results.
9. Synthesize individual NoC components with varying parameters targeting an FPGA for area, power and latency results.

After the NoC protocol was established in the NoC adapter, it was designed to address the initial goal of interfacing with a standard socket. Wishbone [11] was chosen due to its open source nature and was later adapted for Avalon [12] interface for use with Altera's SoPC Builder [14] design software. Using VHDL generics, the wormhole router was designed in order to be flexible enough to fit a designer's needs merely by specifying generic maps. FIFOs and address-to-destination modules were created in order to fulfill

goal number three to create a functional NoC component library. Goal four and five were addressed by using C++ and VHDL component maps to create full NoCs using user input. The sixth goal was validated by using Altera's Quartus II [15], SOPC Builder [14], Nios II IDE [16] and Mentor Graphic's Modelsim [17] to implement and simulate the NoC architectures. Using VHDL file I/O functions and Modelsim variable watching, throughput and average packet latency were measured with the aid of a C++ parser program in order to meet goal seven's requirements. For goal eight, Altera Quartus II [15] was used to synthesize the eight NoC variants for resource usage measurements. Finally, goal nine was achieved by using Altera Quartus II's synthesis, fitting, timing and power analyzer tools with the variations of individual NoC components.

1.2 Thesis Organization

This thesis aims to explore NoC prototyping on FPGAs through component design, implementation and parameter evaluation. It begins with background and related work regarding FPGAs and NoCs, granting the reader the needed technological understandings for the topics contained in this thesis. Chapter 3 continues on with in-depth descriptions of the adapter and router discrete NoC components. Chapter 4 covers the system benchmark and supporting topics required for high level NoC evaluation. In Chapter 5, the process of the design of the NoC is detailed, as well as the NoC Generator program. Chapter 6 explains the evaluations and results of the NoCs and discrete components. Chapter 7 concludes the thesis and explains possible future work.

Chapter 2

Background and Previous Work

This chapter covers a detailed overview of Network-on-Chip research beginning with its relation to the computer network OSI model. It then flows through each of the four related layers, beginning at the top-most layer, describing the aspects of the NoC paradigm. This is followed by the standard socket section, which goes in depth about the Avalon [12] and Wishbone [11] interfaces. Following this, FPGA technology is covered, detailing the benefits and some explanations about the technology. The CAD tools used in this thesis are briefed and the chapter concludes with a subsection describing related research.

2.1 Network-on-Chip Overview

The Network-on-Chip (NoC) paradigm is an architecture inspired by macro computer networks, where data communication is enabled through the use of communication-centric hardware and protocols. [18] shows an excellent example of an NoC overview in Figure 2, where the switch nodes are responsible for routing data between the IP cores, the links are responsible for connecting the nodes, the network interfaces decouple the cores from the NoC and the IP cores carry out higher level functions.

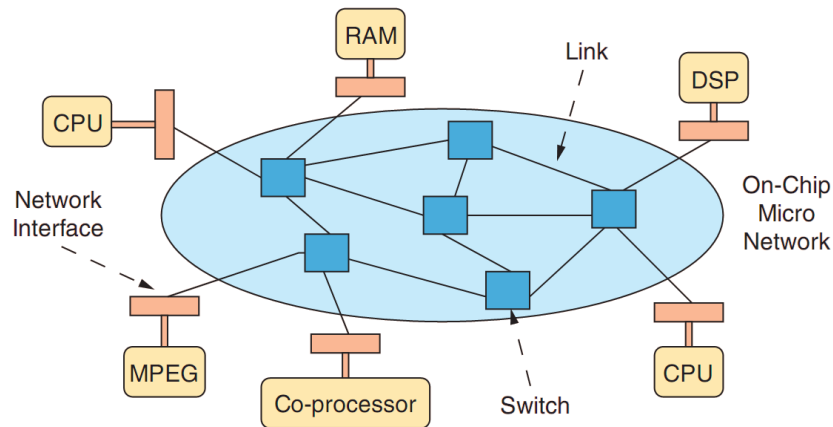


Figure 2 - NoC Example System

It follows that an NoC is similarly structured after the layered Open Systems Interconnect (OSI) model of macro networks, but due to the limited scope in which on-chip communication encompasses, some layers can be compressed [19]. A 4-layer stack can be utilized, based on a compressed OSI model.

The Application Layer is formed from the Application, Presentation and Session layers of the OSI model, which is the top-most layer. It consists of Intellectual Property (IP) cores and the communication between them. The Transaction Layer consists of the Transport and Network layers of the OSI models. This layer consists of the network adapters, which link the IP cores and the NoC through (de)packetization, error handling and end-to-end connection. The Data Link Layer consists of the inner workings of the routers themselves and is responsible for the flow of traffic between two routers. The Physical Layer is the actual link between the switches - the size of the links, as well as the handshaking protocol between them, lie within this layer's responsibilities. [20] illustrates these layers and how they are interconnected. A more practical structure for NoC architecture explanation uses four categories: System, adapter, network and link. [4]

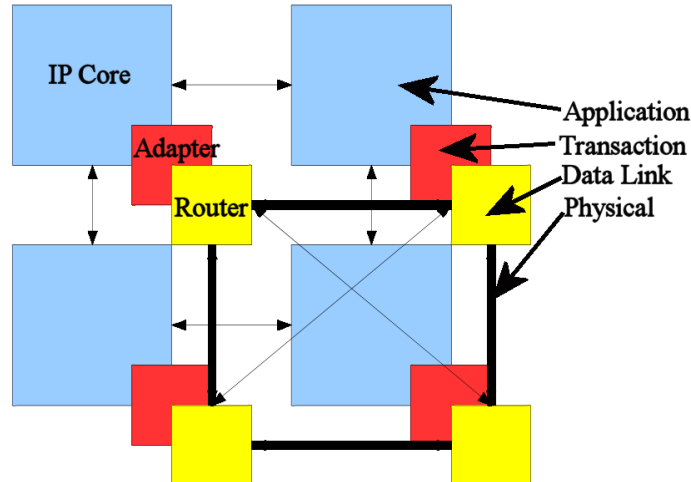


Figure 3 – NoC OSI Layers

2.1.1 Application Layer - System

The Application layer covers the IP cores themselves, including communication between them. This thesis uses many IP cores, most of which are common and straightforward to a computer engineer, but the CPU itself is worthy of attention. Figure 4 contains the components used in this thesis.

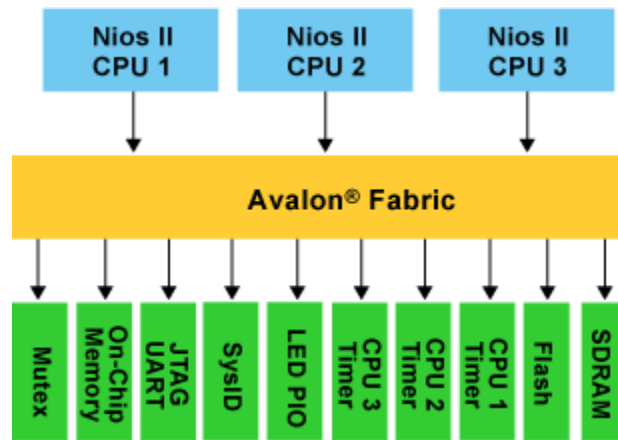


Figure 4 - Components Used In NoC System

The Nios II is a general purpose RISC soft-core configurable CPU, provided by Altera Corporation in Quartus II [15] and SOPC [14] Builder software[21]. It can have features added and removed in order to optimize resource usage and meet performance requirements. The Nios II [21] CPU interfaces with other cores via the Avalon [12] system interconnect fabric. There are three basic versions of the Nios II: Nios II/e, Nios

II/s and Nios II/f, standing for Economy, Standard and Fast, respectively. The Economy is designed for the least area usage, while Fast provides the most features and performance. Standard provides a reasonable compromise between the two.

While the exact topology of the NoC does not lie within the Application layer, the mapping of the IP cores within the NoC does and is known as clustering. Clustering, or *mapping*, is when the NoC designer decides where different IP cores are embedded in the network in order to optimize certain metrics [22]. Methodologies for regular topologies to optimize energy usage and performance of IP mapping [23][24] have been proposed while irregular topologies is still an open area for research.

The IP cores are further divided by their homogeneity and granularity. Traditional parallel computers have course-grained and homogeneous cores, while NoCs are more flexible [4]. An MPSoC, using the Nios II CPU, was used to form homogeneous processing clusters to perform a JPEG encoding benchmark using a packet-switched NoC in [25].

2.1.2 Transaction Layer – Adapter

At this layer, the cores are interfaced with the NoC through a Network Adapter (NA). The NA encapsulates the messages from the cores into packets or streams usable by the NoC, effectively decoupling the cores from the network [4]. Through the use of standard socket protocols, such as OCP [26], Wishbone [27], Avalon [28] and VCI [29], the reusability of an adapter increases. This comes with a price, where conforming to a socket adds additional resources and latency. In a packet-switched NoC, the packets are delimited into three sections: The packet, the flit and the phit. A message is the data generated by the core, which is encapsulated by the packet, which contains additional information such as source and destination addresses, tail information and so on. A packet is subdivided into flits, which is a basic datagram. The phit is the physical unit that can be transmitted, which is commonly the same as the flit.

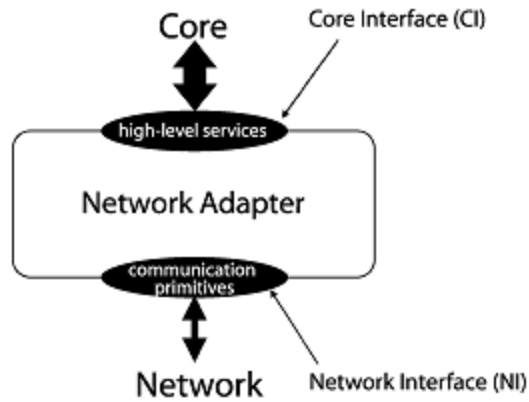


Figure 5 - Network Adapter [4]

2.1.3 Data Link Layer – Network

The Data Link layer, or *network*, is the heart of the NoC since it is responsible for delivering the messages to their destination. Since there are so many different designs for this layer, it is further subdivided into sections – Topology, Protocol, Flow Control and Quality of Service (QoS).

2.1.3.1 Topology

Topology concerns the logical and physical layout of the network and is divided into regular and irregular topologies. While irregular topologies are generally superior, they are difficult to design and lack flexibility [30]. Regular topologies offer simpler physical mapping onto ASICs as well as simpler routing schemes and predictable power and area scaling. Typical regular topologies include mesh, torus, ring, star and binary tree. Mesh and torus are the most common topologies used in NoC research [31]. The main components which make up an NoC is the topology and protocol.

2.1.3.2 Protocol

The protocol of an NoC deals with the strategy of how data moves through the network. It is further broken down into three categories – switching, routing and connection mechanism.

Switching involves the methodology of data transportation while routing is the intelligence behind it. Circuit switching is akin to bus-based systems, where a path between cores is set up and data flows between them asynchronously. Packet switching

involves encapsulating messages within a datagram, which then pushes through the NoC by means of buffers. The key trade-off between the two is that circuit-switched techniques have increased bandwidth at the expense of channel set-up time, while packet-switched techniques allow reactive performance. It has been shown in [32] that packet-switched NoCs perform better when the amount of active links is below 40%.

Routing involves the specific path that the data takes through the NoC and is a key component in reducing congestion in the network as well as affecting average latency and power. Routing strategies are either deterministic or adaptive. A deterministic strategy's routing path is set according to the source and destination alone, where an adaptive strategy adjusts the path mid-traversal according to other factors, such as congestion and priority. A minimal routing scheme always utilizes the shortest path possible between cores, while a non-minimal scheme does not. The control mechanism for routing is either centralized or not; a bus-based system has a centralized arbiter, while a router can have routing decisions made locally inside each node.

The connection mechanism concerns the coordination of connection paths between cores. A connection-oriented mechanism creates the path between cores before transmission, while a connection-less mechanism performs pathing on a per-hop basis. A circuit-switching technique is always connection-oriented, while a packet-switched technique can be either.

2.1.3.3 Flow Control

The flow control defines the mechanisms of how packets flow through the NoC routes, which in turn encompasses local and global issues [33].

One concept of flow control is the Virtual Channel (VC). The VC involves sharing a physical link between routers by means of Time Division Multiplexing (TDM), which, at the expense of additional logic, reduces congestion, improves wire utilization, improves performance and reduces deadlocking. Deadlocking is a situation when network resources become indefinitely frozen waiting for successive interdependent resources to free. Figure 6 illustrates an example of deadlock. Router 1 needs to send to router 4's local port, router 2 needs to send to router 3's local port and so on, but router 1's east port is waiting for router 2's south port, whom is waiting for router 4's west port, whom is waiting for router 3's north port, whom is waiting for router 1's east port.

Virtual channels can break the loop, as can proper routing and placement to avoid these situations [4].

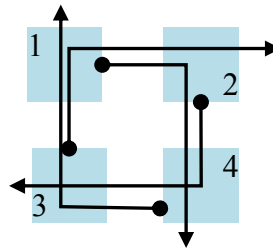


Figure 6 - Example of Deadlock

The forwarding strategy involves the methodology behind packet storage and flow within and between nodes for packet-switching techniques. The Store and Forward (S&F) strategy buffers the entire packet within a node and routes the whole packet through the network. While this offers a simple design, the routers have large buffers which consume a lot of area. Wormhole routing involves routing flits instead of packets. Each router contains one flit so the packet spans multiple nodes. This allows for reduced buffer space but it causes congestion issues as the worm spans multiple routers. The Virtual-Cut Through (VCT) strategy is a mix between worm-hole and store-and-forward strategies. Before the first flit is sent, look-ahead mechanisms guarantee an open path before creating a worm; if the path is blocked, then the packet is buffered, similar to store-and-forward.

2.1.3.4 Quality of Service

Quality of Service (QoS) is the set of priorities and guarantees regarding specific performance metrics provided to the cores by the network. The services could be latency, power, throughput, jitter and so on. There are two identities of QoS – Best Effort (BE) and Guaranteed Service (GS). Best Effort attempts to improve performance and resource use at the cost of reduced predictability of traffic. Guaranteed Service QoS is inherently connection-oriented and provides the maximum predictability for traffic.

2.1.4 Physical Layer – Link

Links are the physical or virtual channels between nodes in an NoC. While research regarding link design is more relevant to ASIC implemented NoCs, there are still some metrics regarding links on FPGAs. Handshaking protocol and bit width can reduce FPGA resource usage as well as power consumption. Most of the issues regarding physical wire problems, such as crosstalk, swing, noise and so on, are generally not issues on FPGAs.

2.2 Standard Sockets

SoC core reusability is increased by the use of standard sockets. The interfaces used in this thesis include Silicore/Opencore.org's Wishbone and Altera's Avalon.

2.2.1 Wishbone

Wishbone is an open-source synchronous SoC interconnection architecture, intended to be a general purpose interface between IP core modules. A handshaking protocol for transfers allows variable transfer speeds.

2.2.1.1 Signals

Wishbone has a variety of signals, used to provide flexibility and compatibility for attached IP cores. The signals common to both master and slave devices are:

CLK_I – Clock input. All Wishbone output signals are registered on the rising clock edge.

DAT_I – Input data array, with a maximum size of 64 bits.

DAT_O – Output data array, with a maximum size of 64 bits.

RST_I – Synchronous reset signal

TGD_I – Input data tag array, containing information regarding the *DAT_I* signal. The data tag contains user defined information.

TGD_O – Output data tag array, associated with the *DAT_O* signal.

Master signals include:

ACK_I – Acknowledge signal used for the handshaking protocol, which indicates the termination of a bus cycle.

ADR_O – Address output array

CYC_O – Cycle output signal, indicating a valid bus cycle when asserted. For burst and block cycles, the *CYC_O* signal is held high for multiple transfers until the final cycle.

ERR_I – Error input signal, used as an alternative to *ACK_I* to indicate a failed transfer. The exact functionality of this signal depends on the IP core.

LOCK_O – Lock output signal, used to ensure a transfer is uninterruptable. The exact functionality of this signal depends on the IP core.

RTY_I – Retry input signal, used as an alternative to *ACK_I*. The exact functionality of *RTY_I* depends on the IP core.

SEL_O – Select output array, used for fine control over data granularity. The size of *SEL_O* depends on the data width and granularity. For example, 8 bits are used for a 64 bit data bus with byte granularity.

STB_O – Strobe output signal, used to indicate valid data transfer cycles. Unlike *CYC_O*, *STB_O* is deasserted after a transfer.

TGA_O – Address tag output signal, used to contain tag information associated with the *ADR_O* signal. For burst transfers, the *TGA_O* tag contains Cycle Tag Identifier (CTI), and Burst Type Extension (BTE) tags regarding burst specifics.

TGC_O – Cycle tag output signal, used to contain tag information regarding a bus cycle. It can be used to distinguish between a single, block or RMW cycle.

WE_O – Write enable output signal, used to indicate a write transfer.

Slave signals receive the exact same master signals, but in an opposite direction. For example, *CYC_I* receives the cycle output signal, whereas *ACK_I* sends an acknowledge response from the slave to the master's *ACK_O* signal. The types of Wishbone bus cycles are divided into three sections – Single, block and burst.

Single transfers use a handshaking protocol shown in Figure 7. The master core initiates a transfer with the strobe signal, where the slave responds with *ACK*, *ERR* or *RTY*. Strobe is held high until a response is received, where the strobe signal is then de-asserted. A cycle termination signal (*ACK*, *RTY* or *ERR*) must be asserted according to the logical AND of *STB* and *CYC*.

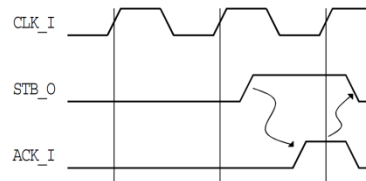


Figure 7 - Single Transfer Handshaking Protocol for Wishbone

A more detailed waveform is shown in Figure 8, where a sample single read transfer is shown. *CYC* and *STB* are asserted to indicate a read request, where the address, selection and associated tags are also applied. The slave responds with an acknowledge signal at clock edge (1), as well as the data and associated tags.

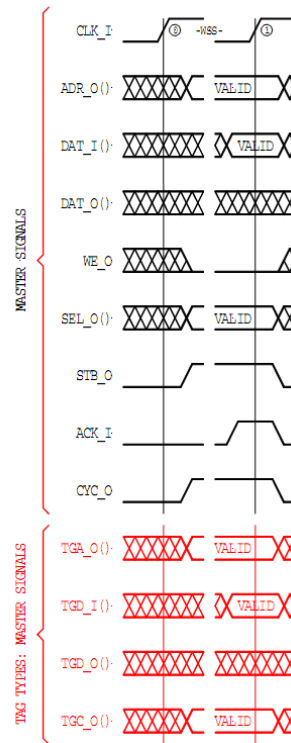


Figure 8 - Single Read Transfer for Wishbone

A single write request is very similar, shown in Figure 9, where *WE_O* is asserted, data is provided by the master on *DAT_O* and the slave terminates the transfer with an acknowledge at edge (1).

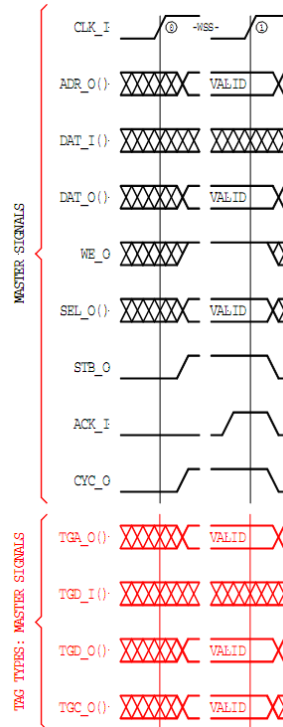


Figure 9 - Single Write Request for Wishbone

These two requests can be performed in a Read-Modify-Write (RMW) request, shown in Figure 10. The *CYC* signal is held high for the duration of the transfer, while the separate strobe signals perform the actual individual transfers.

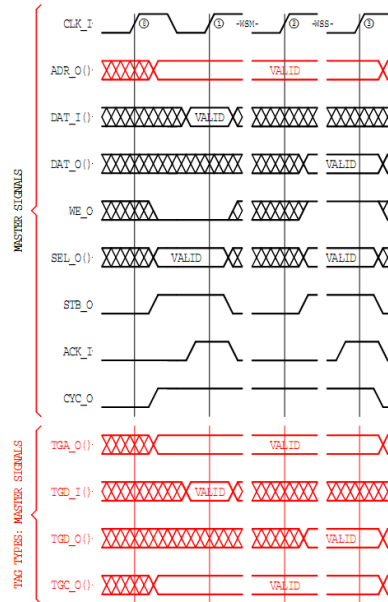


Figure 10 - Wishbone RMW

The block transfers operate slightly differently, where the acknowledge signal may be held high for a number of cycles for multiple transfers for increased bandwidth and reduced delay. A block read request is shown in

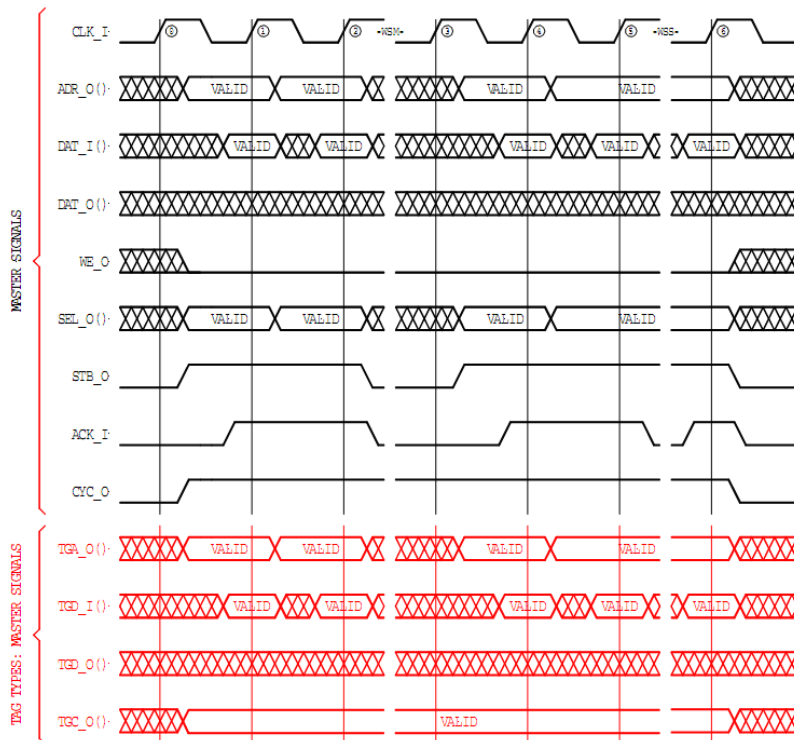


Figure 11. Note that *CYC* is asserted for the entire duration of the transfer.

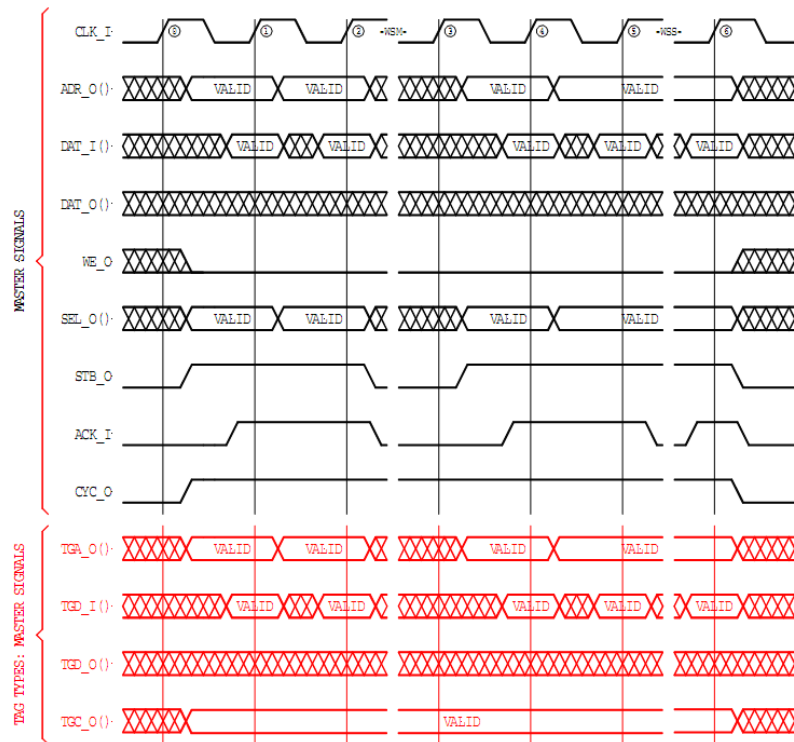


Figure 11 - Block Read Request for Wishbone

Burst transfers address the issue of the additional delays involved when cycle termination signals, in order to reduce wire routing delay, become synchronous. Additional tag signals are used in order to let the slave know of predictable transfers in advance. The Address Tag contains two additional identifiers, used to specify burst characteristics: Cycle Tag Identifier (CTI) and Burst Type Extension (BTE). CTI is 3 bits, and BTE is 2 bits. They are shown in Table 1 and Table 2.

CTI(2:0)	Description
000	Classic cycle
001	Constant address burst cycle
010	Incrementing address burst cycle
011-110	Unused
111	End-of-Burst

Table 1 - Cycle Type Identifier

BTE(1:0)	Description
00	Linear burst

01	4-beat wrap burst
10	8-beat wrap burst
11	16-beat wrap burst

Table 2 - Burst Type Extension

The Classic Cycle is not a burst transfer, where no information about future master cycles is given. End-of-Burst is used to indicate that the current cycle is the last cycle in the burst. Constant address cycle causes a continual access to the same address, until End-of-Burst is given. Lastly, Incrementing address burst uses the Burst Type Extension tag to further define the address behavior. Consecutive addresses, based on BTE are applied. Linear burst simply adds one to the address per cycle, while the beat wrap bursts are modulo the wrap size. Figure 12 is an example of a incrementing address burst transfer.

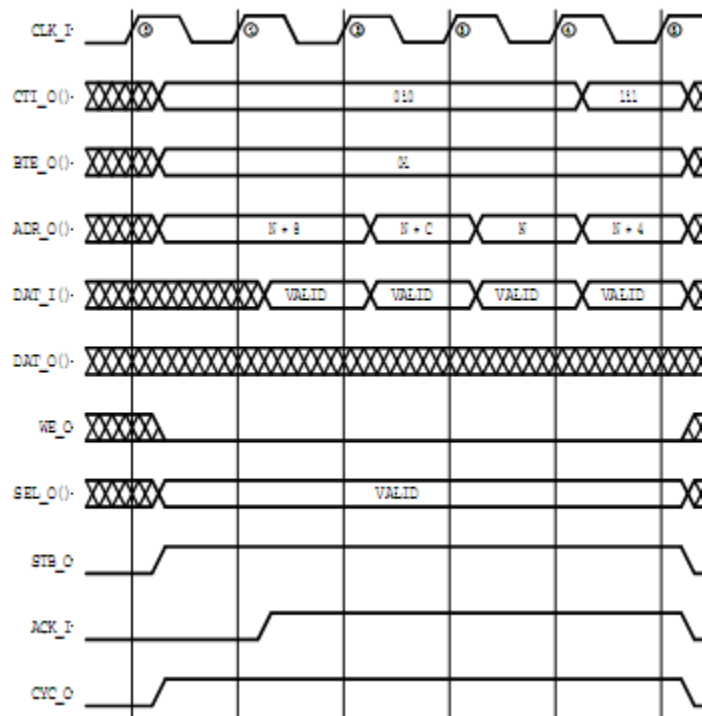


Figure 12 - Incrementing Bursts for Wishbone

2.2.2 Avalon Interface

Altera's Avalon interface is a flexible interconnection architecture aimed at SoCs on FPGAs. While Avalon has six different types of interface – Memory Mapped,

Streaming, Tristate, Clock, Interrupt and Conduit – the Memory Mapped interface will be the main focus due to the nature of the research. The other types will be briefly explained.

2.2.2.1 Avalon-MM

The slave interface uses the following signals. Note that not all of them are required.

Read – *Read* is asserted to indicate a read transfer, where *readdata* is required.

Write – *Write* is asserted to indicate a write transfer, where *writedata* is required.

Address – Contains the address used for read and write requests, and can be up to 32 bits.

Readdata – Contains the data for a read response.

Writedata – Contains the data for a write request.

Byteenable – Used for fine control over data granularity. Selects a specific byte lane for transfer, and has the available bit widths of 1, 2, 4, 8, 16, 32, 64 and 128.

Begintransfer – Asserted for the first cycle of each transfer, regardless of *waitrequest*.

Waitrequest – Asserted by the slave to indicate that it is unable to respond to a request.

Readdatavalid – Asserted when data is supplied in response to a read request.

Burstcount – Indicates the number of transfers that a burst contains, with a maximum size of 32 bits.

Beginbursttransfer – Asserted on the first burst cycle to indicate the start of a burst transfer.

Figure 13 demonstrates examples of slave read and write transfers using Avalon-MM.

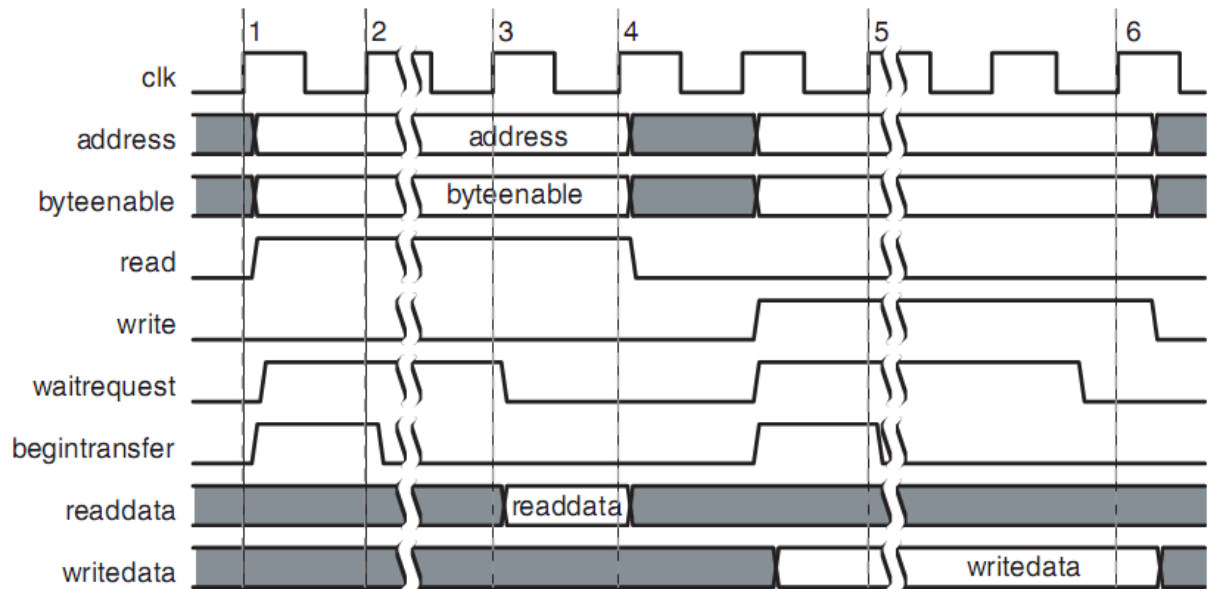


Figure 13 - Example Avalon-MM Transfer

2.2.2.2 Avalon-ST

Avalon Streaming (Avalon-ST) interfaces are used for driving unidirectional and high bandwidth data, where applications include DSP, packets and multiplexed streams. Connected components act as either a source or a sink, with data flowing from the source into the sink.

2.2.2.3 Avalon-MM Tristate

Avalon Memory-Mapped tristate interfaces allow off-chip components to be used. It is relatively similar to Avalon-MM, but with the inclusion of Chip Select (CS) and Output Enable (OE) signals, as well as a bidirectional data line. When chip select is present, all signals are ignored unless CS is asserted. When OE is deasserted, the slave will not drive its data lines.

2.2.2.4 Clock

Clock provides synchronization for the Avalon interface and includes a synchronous reset signal. All internal logic returns to initial states when reset is asserted.

2.2.2.5 Interrupt

Each applicable slave device has an interrupt output signal (IRQ), which is asserted when service is needed. The master device receives up to 32 interrupt signals and, depending on the IRQ scheme, services each interrupt according to a priority table.

2.2.2.6 Conduit

The Conduit interface is used with Altera's SOPC Builder software and is used for exporting signals for connection with external FPGA pins.

2.3 FPGA Technology

Field Programmable Gate Arrays (FPGA) are special integrated circuits, which provide pre-fabricated components and switches. They can be used to instantiate user-defined logic with Hardware Description Languages (HDL), such as VHDL or Verilog. Since FPGAs have grown alongside traditional Application Specific Integrated Circuit (ASIC) manufacturing processes, they contain millions of gate elements and provide an excellent prototype medium for integrated circuit designs. A typical FPGA contains Logic Elements (LE), in the form of look-up tables, which are used to implement custom logic. Newer FPGAs contain more advanced components, such as DSPs, block memories, multipliers, registers and even CPUs. In this thesis, an Altera Stratix II EP2S60F672C3 FPGA is targeted for synthesis analysis in order to provide component area usage as well as power and clock frequency.

2.4 CAD Tools for NoC Implementation on FPGAs

2.4.1 Altera Quartus II

Altera Corporation's Quartus II software is a design environment targeting Altera FPGAs. It provides solutions for all phases of FPGA design flow, shown in Figure 14. The Design Entry consists of writing the HDL files and setting their compilation hierarchy. Synthesis involves compiling and analyzing the design files in order to find the required FPGA resources and their connectivity to realize the design, and Place and Route fits the design onto the FPGA hardware using the available resources. Timing Analysis analyzes the performance of the logic and attempts to meet timing requirements.

Simulation is a verification tool, and the FPGA hardware implementation is the final Programming and Configuration stage. Version 9.0 running in CentOS 4.7 was used in this research, where Synthesis was only used for the NoC, and up to Timing Analysis is used for individual components.

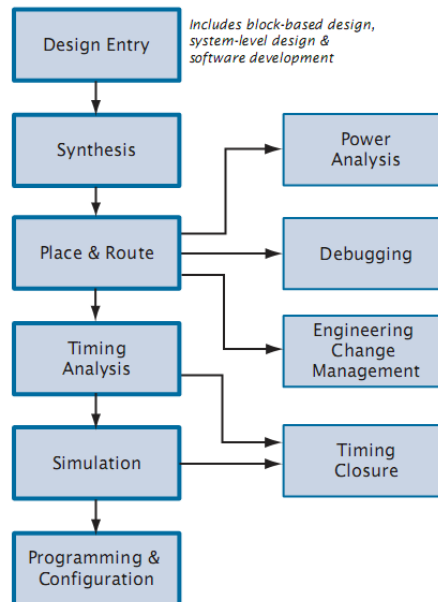


Figure 14 - Quartus II Design Flow

2.4.2 Altera SOPC Builder

System-On-a-Programmable-Chip (SOPC) Builder is included with the Quartus II software and allows for design of embedded systems using the softcore Nios II CPU. Devices use the Avalon interconnection fabric and the NoC component is imported and customized with VHDL generics.

2.4.3 Nios II Embedded Design Suite (EDS)

The Nios II EDS provides a design environment for configuring, programming, debugging and simulating the Nios II CPUs. The Eclipse IDE in the Nios II EDS provides a C/C++ compiler, linker and assembler for Nios II programs. Other such features include in-circuit debugging and Flash programming.

2.4.4 Mentor Graphics ModelSim

Mentor Graphic's ModelSim [17] is a simulation engine for VHDL and Verilog designs. It includes code coverage, assertion tests, breakpoints and in-depth signal and variable simulation that are otherwise not provided by Quartus II's [14] simulation engine. This tool was used to simulate and evaluate the NoC architectures.

2.5 Related Work

In this work, many different areas are touched on. We start with the Avalon-Wishbone glue logic and look at related work in that area. Next, we look at related work in the area of NoC adapters, followed by related work that builds an NoC with the Nios II CPU and supporting software. Other areas of related work include similar routers synthesized for FPGAs and their evaluation methodologies. Nevertheless, this paper demonstrates the similarity between the two standard sockets.

Regarding the glue logic between the Wishbone and Avalon interface sockets, a Wishbone compatible I2C controller was ported to the Avalon bus [34]. The glue logic was verified with simulation results. While the logic is correct for single transfers, there is much missing in the way of variable latency support and high speed Avalon block transfers. The *readdatavalid* signal is not supported in this paper and block transfers will not be queued and hence, forced into a wait state. Lastly, there is no mention of burst transfer glue logic.

A packet-switched wormhole router was implemented [27], utilizing Virtex-4 SRL16 components for FIFO implementation, which increases efficiency but decreases portability and design reuse. A Wishbone adapter was included, which supports burst transfers. Since the routers are input queued, deadlock becomes an issue and was solved by adding a separate read request buffer into the Wishbone adapters, which halts any incoming request when the buffer fills. They tested the design with 16 switches, memories and transaction generators. The individual router was synthesized for Xilinx FPGAs with four and five ports and was compared to related work.

In [35], a 4x4 packet-switched mesh NoC was implemented with SOPC Builder using Nios II CPUs. Multiple Stratix II FPGA boards running at 50MHz were used in order to fit the entire design, which results in an on-board throughput of 650Mbps. Inter-

board communication operates at 50Mbps. A software driver is used to access NoC functions within the Nios II CPUs. The system was verified by probing certain NoC components as a message traverses the network and returns to the sender, and thus was found that the maximum communication rate was 43.4 kPackets/s. This large difference between the theoretical bandwidth of 640Mbps is due to the large amount of time required for the packet to traverse the software routines.

In [36], a packet-switched wormhole router with input queuing was designed and analyzed. The router has four regional ports and one local port, and uses X-Y routing. A 3x3 mesh NoC architecture was implemented with traffic generators attached. The buffer size and traffic patterns were analyzed and explored, resulting in overall increased performance as buffer size increased. A 2x2 NoC was synthesized targeting a Xilinx XC2V1000 FPGA.

In [37], evaluation schemes for NoCs are developed in order to compare performance and characteristics of NoCs. Throughput is defined as the total number flits traversed per time per number of IP cores. Transport latency is defined as the average number of cycles required for a packet to traverse the network. These evaluation methodologies were analyzed using a wormhole router simulator contrasted with various network topologies. The topologies used were SPIN, OCTO, CLICHÉ, Folded torus and BFT. They also compared traffic generator injection loads with throughput and average transport latency. They conclude by stating that this is an important basis for NoC evaluation methodology.

Aetheral [7] is a wormhole-routing NoC developed at Philips Research Laboratories which provides two types of services – guaranteed and best effort, as a result of combining a GS and a BE router. A six port router was implemented for ASIC technology using 0.175 mm² and a four port network interface was implemented for the same technology, using 0.172 mm². They were both implemented on 0.13 μm technology running at 500 MHz.

In [38], a store-and-forward packet-switched router is designed targeting FPGAs. X-Y routing is used and the router is designed in order to reduce FPGA resource usage. A single five port router was found to use 352 Xilinx Virtex-II Pro FPGA slices (2.57% of a XC2VP30). A 3x3 mesh network is implemented, using 28% of the XC2VP30

FPGA. To conclude, timing results required to transmit a packet were shown for various flit sizes and mesh sizes.

2.6 Summary

This chapter has educated the reader on NoCs, socket standards, FPGAs and supporting software. It began with an overview of NoCs, which was further broken down into four sections based on the OSI model. Each section covered the details and design techniques involved in NoCs. The Wishbone and Avalon socket standards were discussed next, educating the reader on the operation of such standards. A brief description of FPGAs is covered, and is concluded with discussions on the software tools used in this work. Related work is discussed. Chapter 3 begins to detail the design and structure of the discrete NoC components developed in this thesis.

Chapter 3

NoC Adapter and Router Design

This chapter discusses the design and structure of the discrete NoC components developed in this thesis. It begins with the NoC adapter, where the NoC protocol is established as well as the supporting modules including *awb* and *adr2dest*. The chapter concludes with a discussion on the wormhole router's design and details.

3.1 Adapter Overview

The PWR adapter is responsible for sending and receiving packets from the NoC and converting them into Wishbone [11] or Avalon [12] signals. It essentially makes the IP cores compatible with the NoC.

The PWR project began being Wishbone-compliant but switched to Avalon to make use of Altera's CAD tools. Thus, the Wishbone aspects of the adapter remain largely untested and are an open area for research.

The adapter is divided into two types of adapters – Master and Slave. As illustrated in Figure 15, the Master adapter is responsible for receiving requests from a master component (such as a CPU) and applying the response signals. The Slave adapter is responsible for applying the master requests and receiving the slave responses.

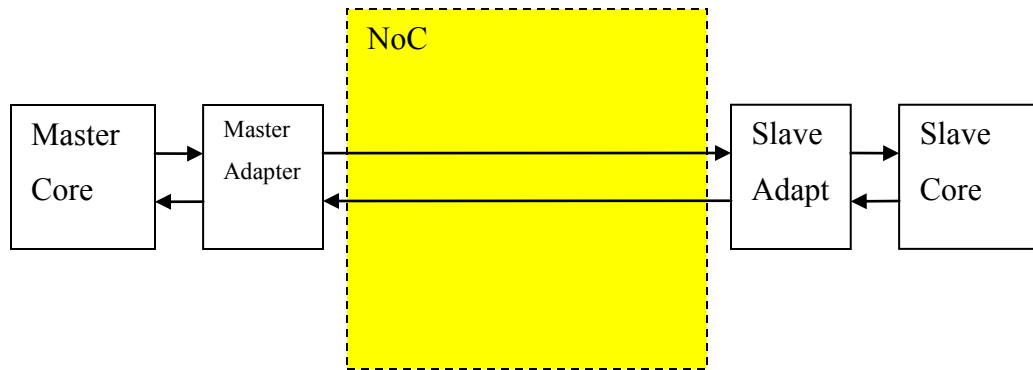


Figure 15 - Adapter Overview

The adapter contains a variety of VHDL generics, offering a degree of design flexibility. The adapter is designed to be compatible with a wide range of signal widths and to conform to Avalon and Wishbone standards. Avalon interface compatibility is obtained through the use of a glue logic module. The logic utilization of the glue logic is very small, and hence negligible. These parameters are divided up into three sections: Interface, NoC and internal. Interface parameters provide flexibility with the Wishbone/Avalon interfacing. NoC parameters allow the adapter to operate in a variety of different NoC architectures. Internal parameters concern the internal operation of the adapter.

Common for both adapters, data width (*WB_width*), address width (*adr_width*), address tag width (*tga_width*), cycle tag width (*tgc_width*), data tag width (*tgd_width*) and selection width (*sel_width*) are VHDL generics used to specify Wishbone interface parameters. Specific to the slave adapter, *cti_lsb* and *bte_lsb* both indicate cycle type identifier and burst type extension least significant bit locations, respectively.

NoC parameters are *flit_size*, *fifo_depth*, *src_width* and *dest_width*. *Flit_size* is the size of a flit, in bits. *Fifo_depth* is the number of registers in the adapter's FIFOs, which allows the adapter to queue up flits if the NoC is congested. *Sr_c* and *dest_width* are the bit widths of the source and destination NoC addresses, respectively. They should both be equal, where the separate parameters are present for future optimization allowing lower bits for source addresses.

The internal parameters are *fast_burst*, *burst_depth*, *burst_tag_en*, *no_ack*, *sdram_delay* and *Avalon_bursts*. *Fast_burst* indicates that burst and block transfers are to be queued up using a burst buffer, thus opening request types 4 and 5. *Burst_depth* is the size of the burst buffer – this parameter is useful if there is a small flit size but large packet size (due to large data width, for example) since more requests can be queued and hence the CPU does not get stalled. *Burst_tag_en* is used to enable burst tags for Wishbone transfers – 1 to enable, 0 to disable. *No_ack* is used when there is no acknowledge signal for reads and writes – For this thesis, it is set to 1. *Sdram_delay* is used with single transfers and delays forming a packet by one cycle – this was needed for interfacing with SDRAM in single transfer mode. *Avalon_bursts* is used if Avalon block transfers are used – this distinction is required since Wishbone block transfers are different from Avalon's as explained in chapter 2.

Packets are made up of flits and the minimum packet size is three bits. The first three bits in a packet is always the request type, while the rest of the packet depends on the request type. The adapter analyses the request (or response) of the IP core and chooses the appropriate request type. Table 3 indicates all the request types and their size. In the case of this research, only request types 3, 4 and 7 are used due to the exclusive use of Avalon block transfers.

Single/burst read request from master	<table><tr><td>Address</td><td>tg</td><td>d</td><td>tg</td><td>c</td><td>tg</td><td>a</td><td>C</td><td>L</td><td>S</td><td>W</td><td>Source</td><td>Dest</td><td>001/011</td></tr></table>															Address	tg	d	tg	c	tg	a	C	L	S	W	Source	Dest	001/011		
Address	tg	d	tg	c	tg	a	C	L	S	W	Source	Dest	001/011																		
Single/burst read response from slave	<table><tr><td>Data</td><td>A</td><td>E</td><td>R</td><td>Dest</td><td>101/11</td></tr></table>															Data	A	E	R	Dest	101/11										
Data	A	E	R	Dest	101/11																										
Single/burst write request from master	<table><tr><td>Data</td><td>Address</td><td>tg</td><td>d</td><td>tg</td><td>c</td><td>tg</td><td>a</td><td>sel</td><td>C</td><td>L</td><td>S</td><td>W</td><td>Source</td><td>Dest</td><td>010/100</td></tr></table>															Data	Address	tg	d	tg	c	tg	a	sel	C	L	S	W	Source	Dest	010/100
Data	Address	tg	d	tg	c	tg	a	sel	C	L	S	W	Source	Dest	010/100																
Single/burst write response from slave	<table><tr><td>A</td><td>E</td><td>R</td><td>Source</td><td>Dest</td><td>110/000</td></tr></table>															A	E	R	Source	Dest	110/000										
A	E	R	Source	Dest	110/000																										

Table 3 - Request Type Design

The complete adapter is formed of five modules – *adr2dest*, *awb*, *fifo*, *master/slave sampler* and *master/slave top*. *Adr2dest* is responsible for converting the address signals into NoC destinations. *Awb* is the Avalon-Wishbone glue logic. *FIFO* is the first-in, first-out register bank used to queue incoming and outgoing flits for the adapter. The sampler is the main logic of the adapter, responsible for packetizing and de-

packetizing the interface requests and responses. Finally, the ‘top’ module is responsible for the hand-shaking protocol between the sampler and FIFOs, and the sampler and NoC. Each component is described below.

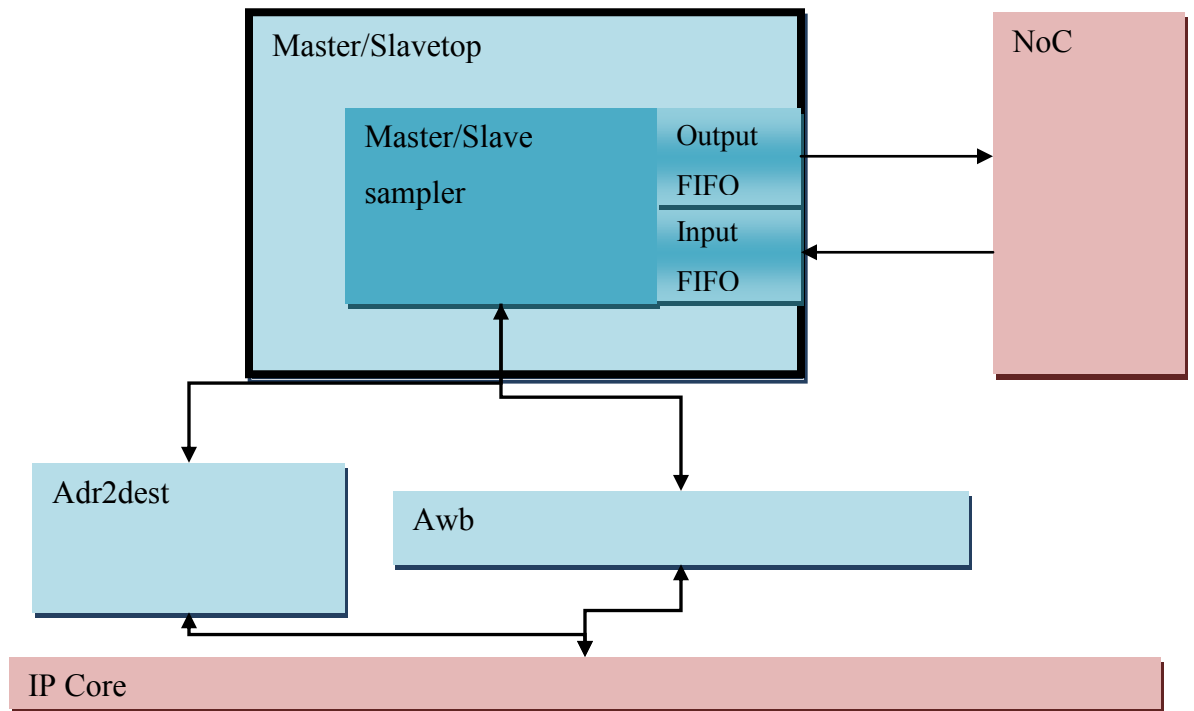
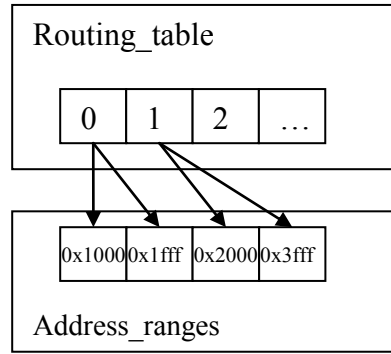


Figure 16 - Adapter Design Overview

Adr2dest (address to destination) is a simple look-up table for destinations. For a specific range of addresses, a destination is output. There are two unique VHDL generics in this module: *routing_table* and *address_ranges*. Routing table is an array of integers and contains the destinations that are to be outputted. There are 33 elements in this array and can be expanded by editing the *quick_convert* package. The *quick_convert* package contains functions and VHDL types used in the source code. *Address_ranges* contains the lower and upper addresses in order to output the destination. For *address_ranges* 0 and 1, *routing_table* 0 is outputted. For 2 and 3, *routing_table* 1 is outputted and so on. Figure 17 demonstrates this functionality.



**Figure 17 - Address to Destination
Parameter Design**

Awb is the Avalon-Wishbone glue logic. It contains both the slave and master adapter interfaces, indicated with a prefix *wbs/wbm* or *avs/avm* for Wishbone and Avalon, respectively. Most of the logic is simple name changes for the signals to make building in SOPC Builder [14] easier and the component interface conversion is bidirectional. There is additional clocked logic used to delay the de-assertion of Avalon read/write signals by one cycle since de-asserting these signals is not allowed immediately when the *wait_request* signal is de-asserted as well. The connections are illustrated in Figure 18.

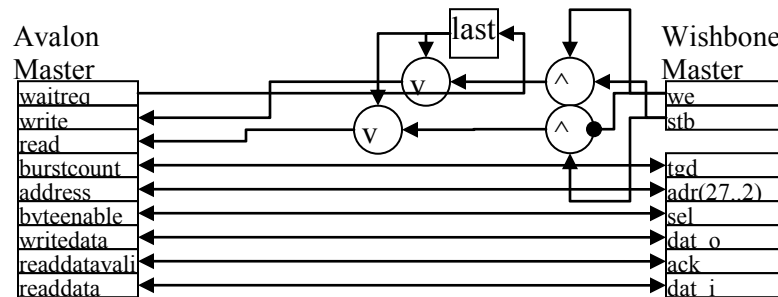


Figure 18 - Avalon-Wishbone Glue Logic for Master

FIFO is an array of registers, responsible for queuing flits in to and out of the adapter. An extra ‘overflow’ register is provided to help stop issues with control signal latency. The FIFO’s depth is specified with VHDL generics. ‘Empty’ and ‘Full’ are used to indicate when the FIFO can be read from or written to.

The samplers have two unique versions – master and slave. The operation of the samplers is based around the idea of ‘sampling’ and saving bus signals, yet the operation

of the adapters is more complicated than this. Simply sampling the bus at specific intervals, placing in a packet and sending over the network would cause a lot of wasted packets being sent since some transaction signals are predictable. The Wishbone operation handles three types of transactions: Single, block and burst. Single transactions in the adapter perform one complete transaction at a time. Block transactions is essentially the same as single transactions for Wishbone, but with a key difference in that the acknowledge signal is predicted to be asserted for write requests and is done so artificially, thus increasing the speed of the adapter. Read transactions for Wishbone block transfers operate the same as single transfers, since a response is required and cannot be predicted. Burst transfers include the cycle and address tags (CTI and BTE, respectively) so read requests can be sped up, similar to how block writes work. The Avalon block transfers operate differently in that requests can be ‘queued’ without an acknowledgement for the previous request. Thus, a specific VHDL generic (*Avalon_bursts*) is used to switch the adapter into Avalon’s block request queuing mode. The operation of the master adapter in this mode is illustrated in Figure 19, and the slave adapter is shown in Figure 20.

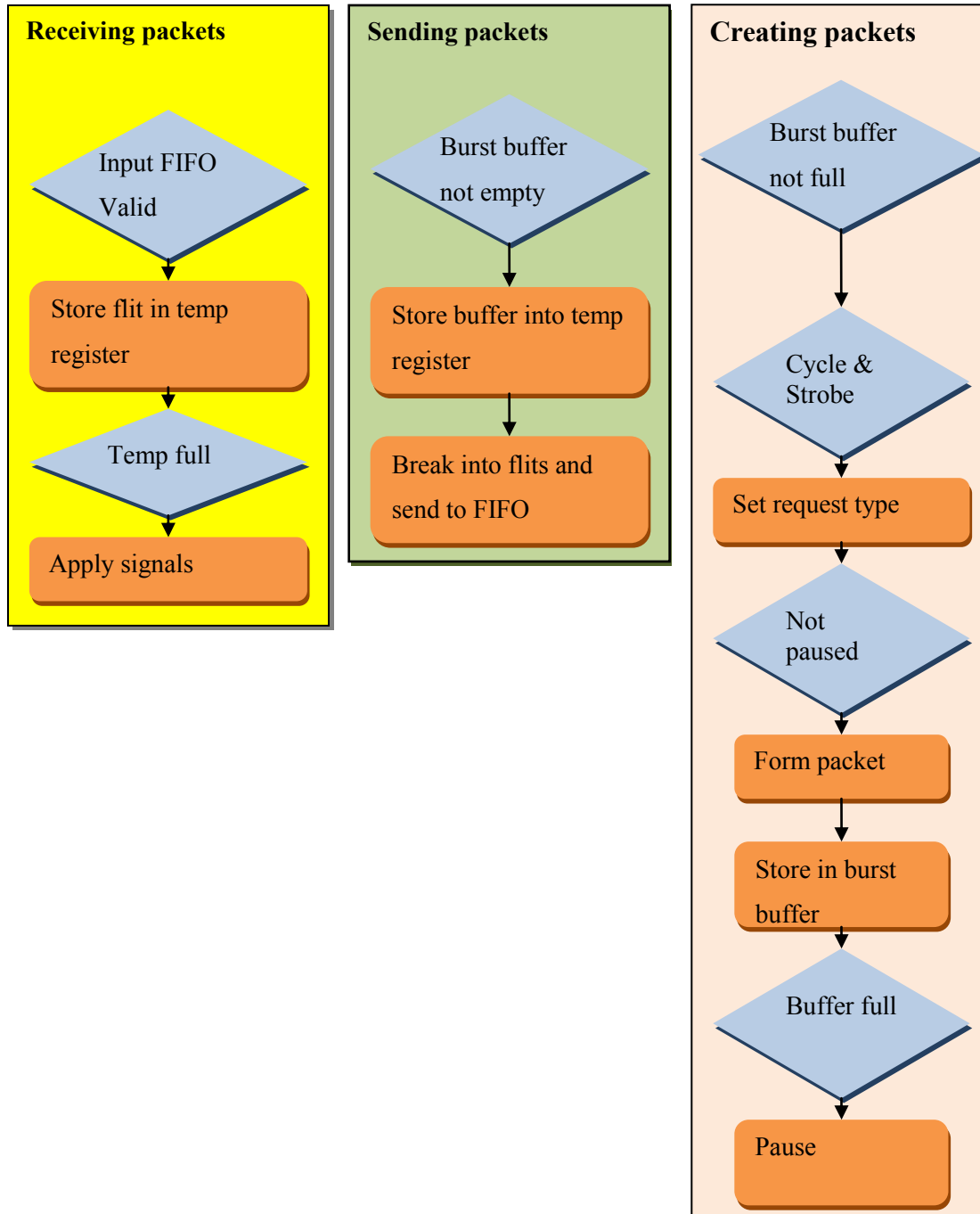


Figure 19 - Master Sampler 3-Concurrent Process Flowchart in Burst Mode

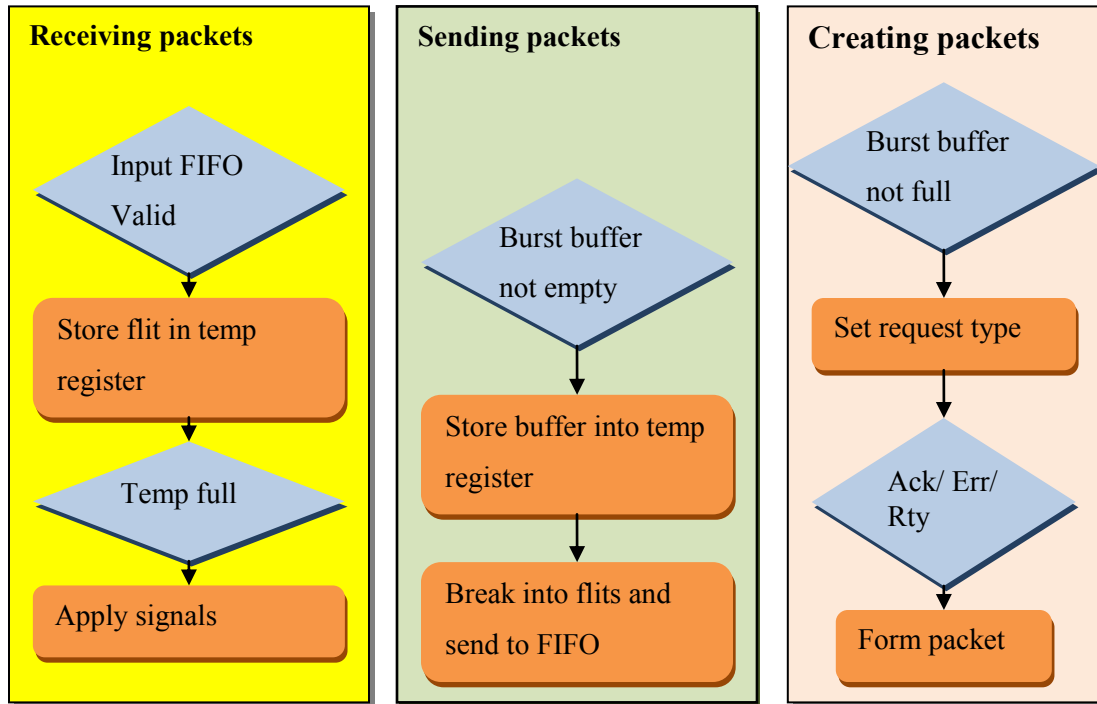


Figure 20 - Slave Sampler 3-Concurrent Process Flowchart in Burst Mode

The Top modules (*mastertop* and *slavetop*) connect the samplers with input and output FIFOs. The top modules are also responsible for providing the handshaking protocol between the FIFOs and the NoC. This is done with two flip-flops – *wait_for_noc_ack* and *noc_sent*. *Wait_for_noc_ack* is set when an output FIFO sends a flit and is cleared when the NoC acknowledges (via deasserting the *receive_ready* signal). *Noc_sent* is similar, where it only writes the first flit to the input FIFO until *noc_send* is deasserted.

3.2 Router Overview

The PWR router is a packet-switched wormhole router with two deterministic routing schemes – X-Y routing and source routing, and a general top view of the module is shown in Figure 21. Note that the number of ports is defined by the VHDL generics.

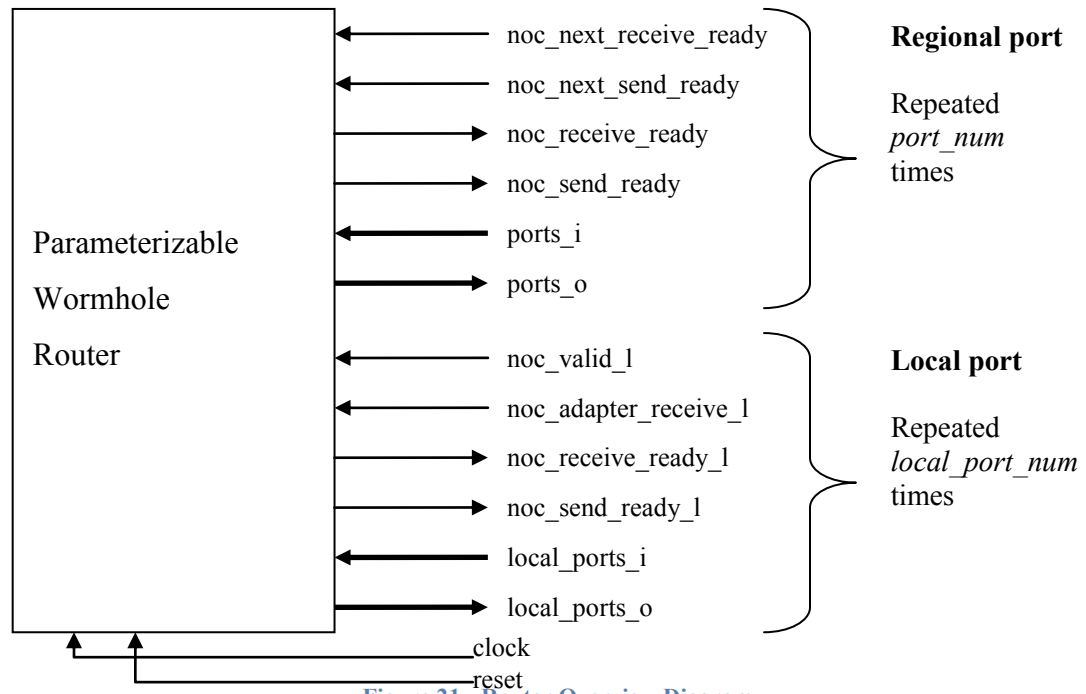


Figure 21 - Router Overview Diagram

For source routing, the protocol is routed in that a unique routing lookup table is present in each router. The arbitration is round-robin and flow control uses a send/acknowledge protocol. The switching mechanism uses VHDL FOR loops to implement a full crossbar switch. A simplified data view of the router is shown in Figure 22.

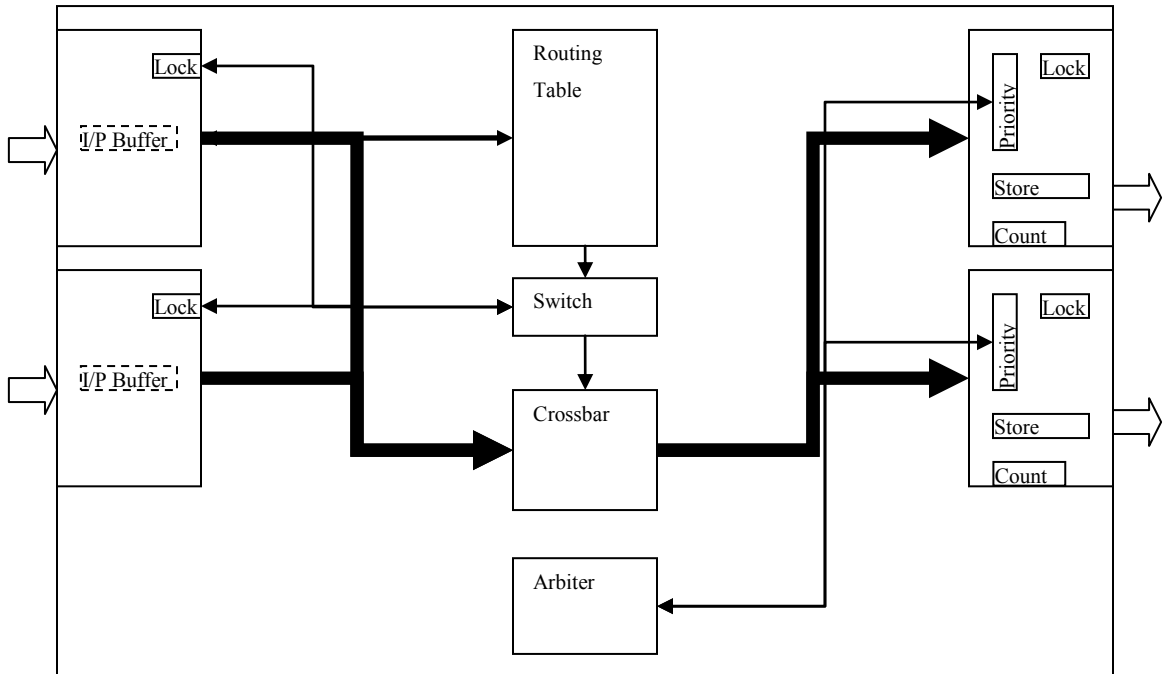


Figure 22 - Internal Router Design

Each output port has a storage register, large enough to hold a single flit. Incoming flits are stored after internal routing and arbitration by means of obtaining the destination address. In the case of the destination address not being obtainable in the first incoming flit, a special “input buffer” register is used to store incoming flits. Once the flit is stored, the input and output ports are locked and a counter is started. Once the counter reaches zero, the worm is complete, meaning the entire packet has been successfully sent through the node. The input and output ports are unlocked and the priority table of the output port is adjusted so that the input port has the lowest priority.

There are a number of VHDL generics which allows flexibility in the router design, and these are broken up into three sections: Interface, NoC and Internal. The Interface parameters include the bus interconnect bit-widths: *WB_width*, *adr_width*, *tga_width*, *tgc_width*, *tgd_width*, and *sel_width*. These parameters are used to create the constants required to create the flit worm. The parameters *fast_burst*, *burst_depth*, *cti_lsb*, *bte_lsb* and *burst_tag_en* are unused. The NoC parameters are *flit_size*, *routing_table*, *src_width* and *dest_width*. The Internal parameters include *num_ports*, *num_local_ports*, *routing_type*, *xy_col* and *xy_row*. *Num_slave* and *num_master* are

unused. *Num_ports* is used to indicate the amount of ports in the router, where *num_local_ports* is for local ports. *Routing_type* is set to 1 to indicate source routing and 2 for xy routing. *Xy_col* and *xy_row* are used to indicate the location of the router for xy routing.

The basic port of the router operates with a handshaking protocol involving the signals *noc_receive_ready*, *noc_next_receive_ready*, *noc_send_ready* and *noc_next_send_ready*. These signals are named differently for local ports, but operate the same. *Noc_receive_ready* is de-asserted for one cycle to indicate to the previous port that the send was successful. *Noc_next_receive_ready* is simply an input signal from the next router's *noc_receive_ready* signal. *Noc_send_ready* is asserted when the port wishes to send its flit and remains high until *noc_next_receive_ready* is 0. *Noc_next_send_ready* is the send signal from the next router.

The input and output ports are locked via a flip-flop for the count of flits inside the packet. Once the last flit is received and successfully sent to the next router, the input and output ports are unlocked and the priority of that input port is set to the lowest priority.

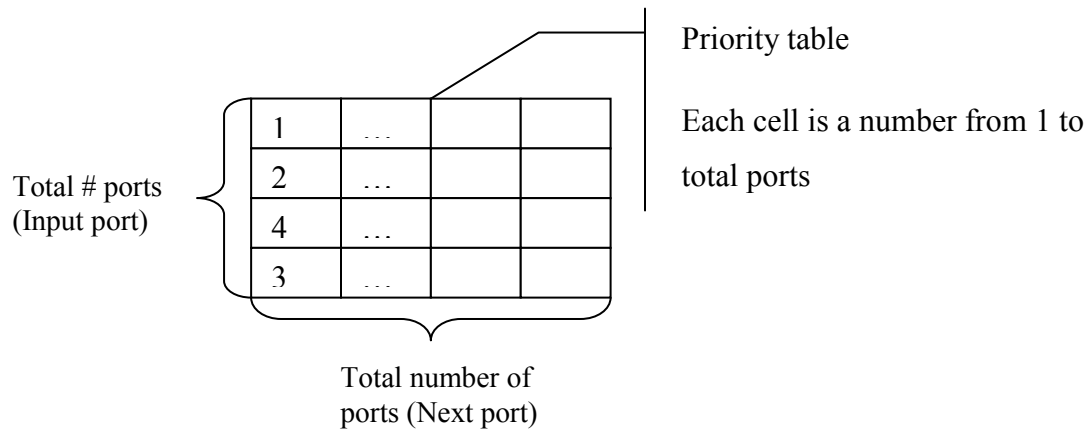


Figure 23 - Priority Table Design

More detail on the arbitration handling in the router is as follows. The priority table consists of a table of priorities per input port, per output port. Each output port contains an array of priority values for each input port, where a high value indicates a high priority. When multiple input ports try to route a flit to the same output (destination) port, the highest priority input port gets precedence. That input port's

priority gets reduced to 1 and all other port priorities are incremented.

The PWR router's QoS mechanism is Guaranteed Service (GS). The flits that form a packet cannot get "mixed up" or corrupted. If an issue ever arises, it would be due to a malformed packet from the adapter and may result in an assertion error when the router cannot understand the request type (first three bits) in a packet. Deadlocking can occur if the adapters freeze up due to FIFOs becoming full and not emptying, and interdependent resources become locked in a permanent wait state. Livelocking occurs when resources get frozen in perpetually changing states, such as through re-sending or redirection of packets [4]. Since PWR uses deterministic routing schemes, it becomes up to the NoC designer to guarantee that livelocking cannot occur, as well as to minimize deadlocking with careful placement and routing. Through the use of directed graphs, a deadlock-free system can be realized [22].

3.3 Summary

This chapter discussed the design and structure of the discrete NoC components designed in this work. It began with a description of the NoC adapter, followed by design architectures of the supporting modules including *awb* and *adr2dest*. The details of the adapter are discussed, which is followed by the wormhole router's design. The chapter concludes with discussion on the router. The upcoming Chapter 4 discusses the implementation of the NoC and the framework to evaluate the design.

Chapter 4

NoC Implementation and Evaluation

Framework

Chapter 4 begins with a discussion of the test system to be enabled by the NoC, including the components used and the operation of the system. The NoC is applied to this system, detailing certain difficulties involved, which follows with in-depth details of the operation of the benchmark system. Chapter 4 concludes with a discussion of the evaluation environment.

4.1 Multi-CPU Benchmark System

The goal is to have real traffic from a practical system, as well as to have this traffic change and flow according to the performance of the system. Work with NoCs using real systems has been done [22], but in general, there needs to be more research on the topic. A multi-processor design example was chosen from Altera's website, with the intention to replace the Avalon [12] bus fabric with an NoC. It was modified to suit a simulation environment and to reduce the amount of components. The modified multiprocessor example, shown in Figure 4, contains three Nios [21] II/f soft core CPUs, three 1 ms timers, 16 MB of flash memory (AMD29LV128M123R_BYTE), a mutex, 64 KB of on-chip RAM, 1 KB of message buffer RAM (on-chip), 256 Mbit (16 bit) SDRAM (Nios Development Board, Stratix II), a JTAG UART interface module, a sysid module and an LED PIO. The system operates by means of initially booting off the Flash memory, followed by reading the data and instruction code from the DDR

SDRAM. Each timer is responsible for sending interrupt requests to the CPUs, who then take turns attempting to acquire a mutex lock. Once a lock is established, the CPU writes a message to the message buffer; CPU two and three also send the signal to the LED PIO. CPU one is responsible for reading this buffer, clearing it and sending the message to the JTAG UART interface module. Each CPU sends a total of five messages and then idles indefinitely. The program code is contained within the SDRAM for all three CPUs, within separate locations. All the CPUs have their reset vector in FLASH memory. CPU one's interrupt vector is contained within the on-chip memory module, while CPU two and three's interrupt vectors are in the SDRAM. The functionality of this design is as follows – Each CPU attempts to acquire the mutex, which results in them writing a message to the message buffer. CPU one is responsible for reading the message, sending it to the UART module and clearing the message. Of course, CPU one must have a mutex lock. The timers interrupt their respective CPUs, which cause them to attempt to acquire a mutex lock.

4.2 Implementation of NoC in SOPC Builder

The NoC is added into SOPC's [14] component editor. To clear confusion, master and slave adapters are connected to master and slave IP cores, respectively. A master adapter's Avalon interface is called "slave" for reasons that the Nios II master interface needs to be connected to a slave interface. The Avalon parameter *maximum pending reads* is set to 8 due to the block transfers of the Nios II being in groups of 8 and to increase performance. The parameters in SOPC Builder were set according to Figure 24.

num_slave	11
num_master	12
flit_size	8
routing_type	1
VB_width	32
adr_width	27
tga_width	0
tgc_width	0
tgd_width	0
sel_width	4
dest_width	5
src_width	5
burst_count_width	1

Figure 24 - NoC Parameters in SOPC Builder

The component is then added to the system and the interfaces are connected. Figure 25 shows the torus NoC implemented in SOPC Builder. Each address is assigned manually, where the address bus from the Nios II is 32 bits and the address bus from the NoC is 27 bits. This means the upper 5 bits are ignored by the NoC but are used by the Avalon fabric's arbitration. Since masters are connected to their own buses, then master adapters can have the same addresses.

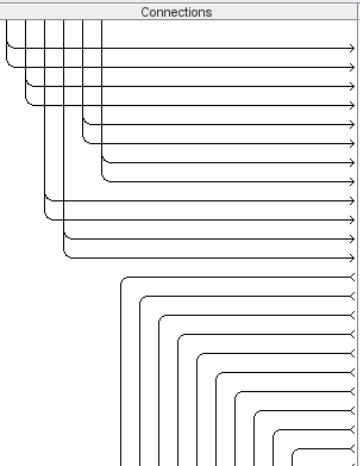
Use	Connections	Module Name	Description	Clock	Base	End
<input checked="" type="checkbox"/>		torus_noc_0	torus_noc			
		avalon_slave_0	Avalon Memory Mapped Slave	clk	0x00000000	0x1fffffff
		avalon_slave_1	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_slave_2	Avalon Memory Mapped Slave		0x00000000	0x1fffffff
		avalon_slave_3	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_slave_4	Avalon Memory Mapped Slave		0x00000000	0x1fffffff
		avalon_slave_5	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_slave_6	Avalon Memory Mapped Slave		0x00000000	0x1fffffff
		avalon_slave_7	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_slave_8	Avalon Memory Mapped Slave		0x00000000	0x1fffffff
		avalon_slave_9	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_slave_10	Avalon Memory Mapped Slave		0x00000000	0x1fffffff
		avalon_slave_11	Avalon Memory Mapped Slave		0x20000000	0x3fffffff
		avalon_master_0	Avalon Memory Mapped Master			
		avalon_master_1	Avalon Memory Mapped Master			
		avalon_master_2	Avalon Memory Mapped Master			
		avalon_master_3	Avalon Memory Mapped Master			
		avalon_master_4	Avalon Memory Mapped Master			
		avalon_master_5	Avalon Memory Mapped Master			
		avalon_master_6	Avalon Memory Mapped Master			
		avalon_master_7	Avalon Memory Mapped Master			
		avalon_master_8	Avalon Memory Mapped Master			
		avalon_master_9	Avalon Memory Mapped Master			
		avalon_master_10	Avalon Memory Mapped Master			

Figure 25 - Torus NoC Implemented in SOPC Builder

The slave components are assigned addresses manually, but each component must have a unique address range in the range of 27 bits. Figure 26 illustrates the Torus NoC connected to the slave components and their respective assigned addresses.

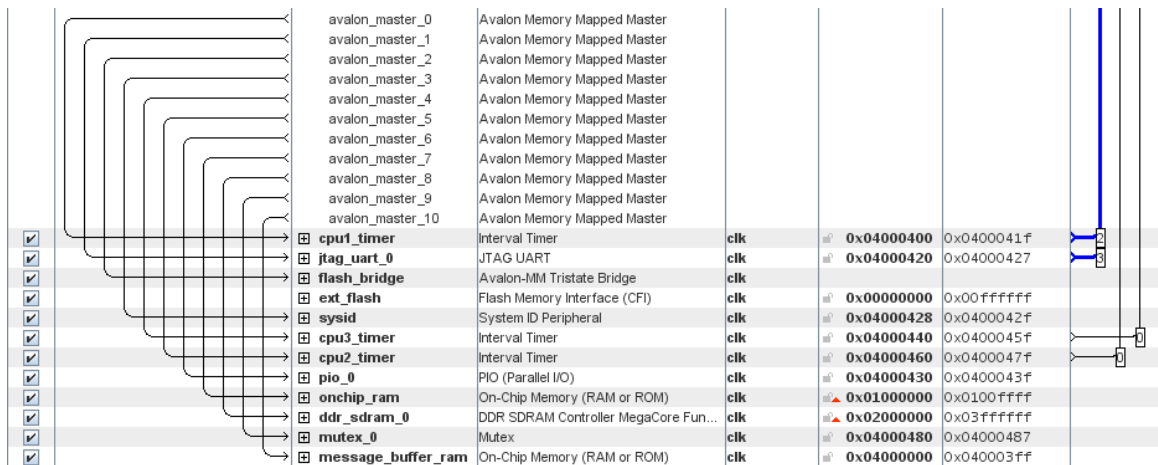


Figure 26 - Torus NoC Connected to Slave Components

Once the component is created, a Tcl script is automatically created by SOPC Builder. This script must be manually modified in order to make the master adapters act as bridges so the reset and exception vectors can be set in the Nios II CPUs. Figure 27 illustrates the concept of bridges, where the Nios II “sees” a master adapter as being directly connected to a memory module. Since each Nios II has their reset and exception vectors pointing to different memory components, and that an interface can only bridge to one other component, it follows that there must be two adapters – one for each vector. If the reset and exception vectors pointed to the same memory module, then only one adapter would be needed. For each master adapter, the *set_interface_property bridgesToMaster* parameters must be modified so they contain the slave adapter’s name.

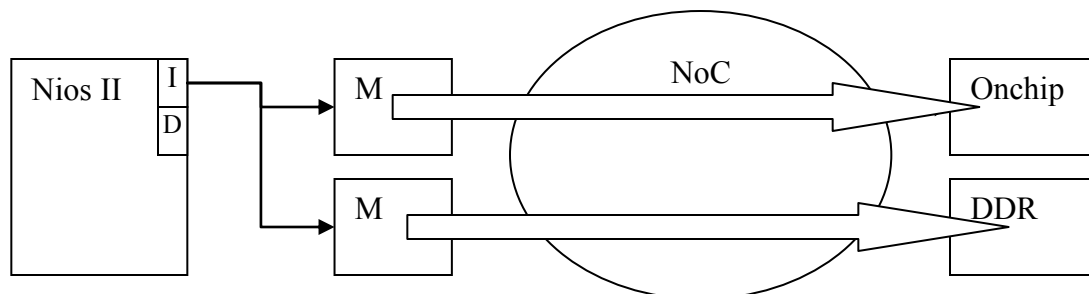


Figure 27 - Bridging Example

The last issue involves the number of adapters for each Nios II CPU. Four were used for each Nios II adapter so that each Nios II bus (data and instruction) gets its own adapter,

which is then divided again due to the bridging issue. Due to the adapters queuing requests and responding when ready, this caused problems when the bus arbiter was not granting access to the correct bus when the adapters were responding. Having separate adapters overcomes this issue.

4.3 Nios II Programming

Before getting into details about the benchmark program, an issue with Nios 2 EDS [16] must be addressed. Since each program resides in different portions of the same memory block and that Nios 2 EDS overwrites the data block when compiling the code for each CPU, a script was set up to copy and concatenate the program files after each compile.

Each CPU attempts to acquire a mutex lock, which results in them writing an incrementing counter to the message buffer. The counters stop at five, after which no more messages are sent from that CPU. The three CPUs are numbered one to three, where CPU one is responsible for clearing the message buffer and writing to the message to UART. CPUs two and three do not clear the message buffer or write to UART, but they write to the PIO. They have the exact same code, but different program locations in the SDRAM. The timers interrupt their respective CPUs, which cause them to attempt to acquire a mutex lock. Figure 28 and Figure 28 illustrate the flowcharts of the programs.

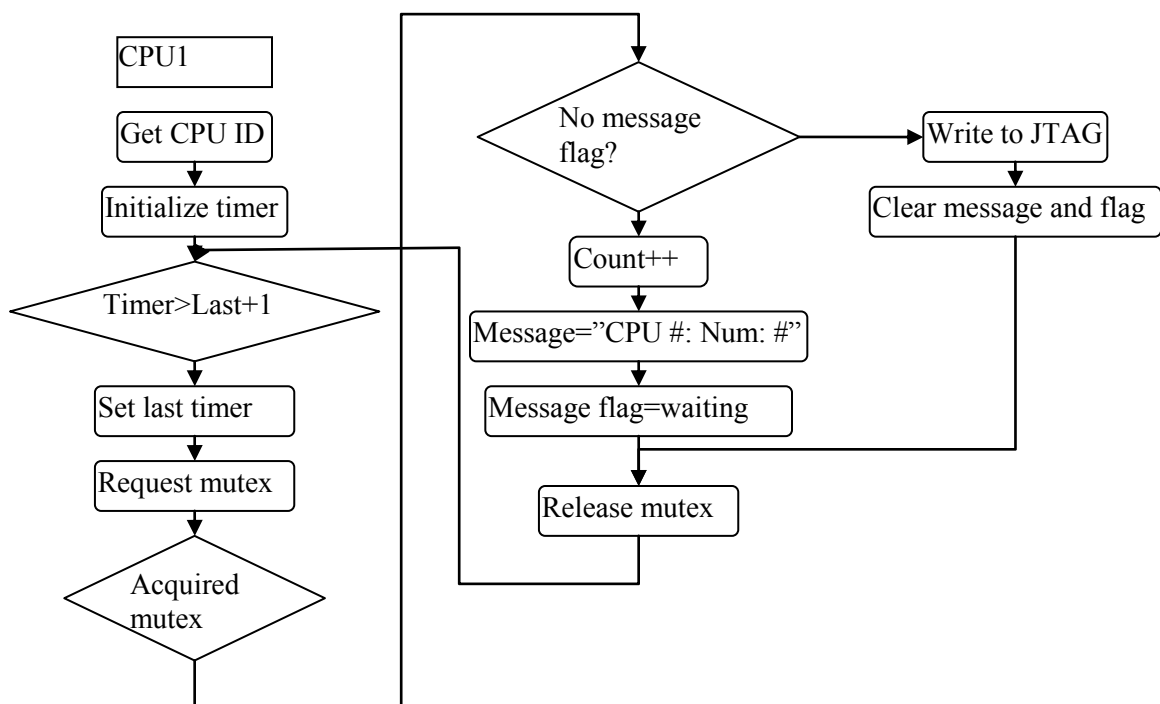


Figure 28 - CPU 1 Benchmark Flowchart

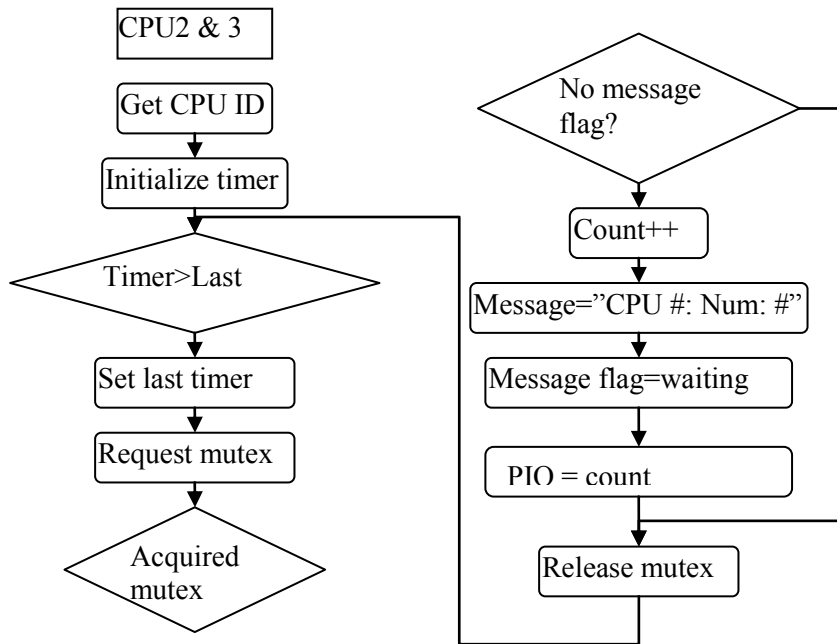


Figure 29 - CPU 2 and 3 Benchmark Flowchart

4.4 Modelsim Simulation Environment

Modelsim [17] allows for very fine-grained simulation, and hence is perfect for the purposes of this research. Using Altera's built-in scripts, the program code is loaded into Modelsim and the automatically generated project files are used. The JTAG UART module outputs its messages to Modelsim's console, which is then used as a basis for simulation end-time. Once each CPU outputs its 5 messages (CPU #: Num: #), the runtime is recorded at the final write operation to SDRAM. Figure 30 shows a sample of the router regional handshaking protocol as seen in Modelsim. At period 1, the router sets the send bit to high on port number two. Four cycles later, at period 2, there is a response on the receive signal from port number two, indicating that it has received the flit. The router deasserts the send signal on port two and, one cycle later, reasserts it to send another flit. This is just an example of what is seen in Modelsim in order to verify the operation of the system.

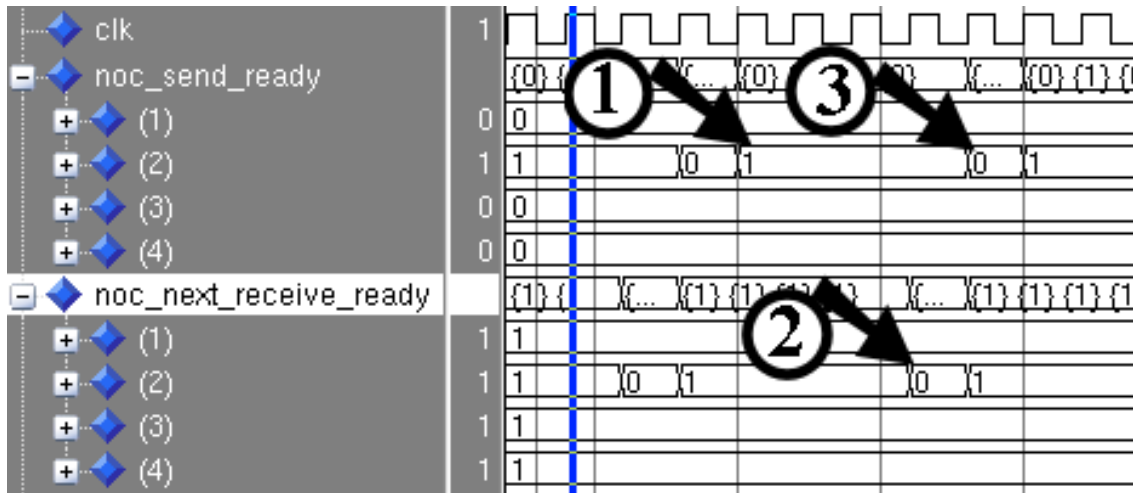


Figure 30 - Router Regional Handshaking

The performance data of the NoC is recorded through Modelsim by viewing the *Messages* VHDL variable inside each master adapter. When a packet is formed or received, the *Messages* variable is incremented by 1. When the program is complete, each variable is accumulated in order to measure throughput. The traversal time is measured by means of VHDL file IO functions, executed when a packet is formed and absorbed. A continuous timer's value, the packet value and whether or not the packet is absorbed or formed is written to file. A C++ program matches the formed and absorbed tags and subtracts the timer values, thus calculating the traversal time in cycles. Modelsim also provides simulation time, which represents the time taken to run the program. The simulated clock period is 20 ns, or 50 MHz.

4.5 Summary

This chapter covered the test system enabled by the NoC and the details of its components and operation. Issues involved in implementing the NoC within this system as well as the details of the system operation were discussed. It concluded with a discussion of the ModelSim evaluation environment used in this thesis. Chapter 5 will discuss the details of the NoCs implemented in this thesis.

Chapter 5

FPGA Implementation of Torus and Ring NoC Architectures

This chapter discusses the details of the two NoC architectures designed in this thesis. It begins with a network topology discussion, followed by details of the IP core placement in each NoC. The routing of such cores is discussed in detail, which concludes with a briefing on the NoC Generator program.

5.1 Topology

Two regular topologies were implemented and simulated in Modelsim – Torus and ring. The torus topology is similar to the mesh topology, in that all nodes have the same number of neighbours by means of ‘wrapping’ node links to opposite sides [39]. Figure 31 illustrates the differences between the two topologies.

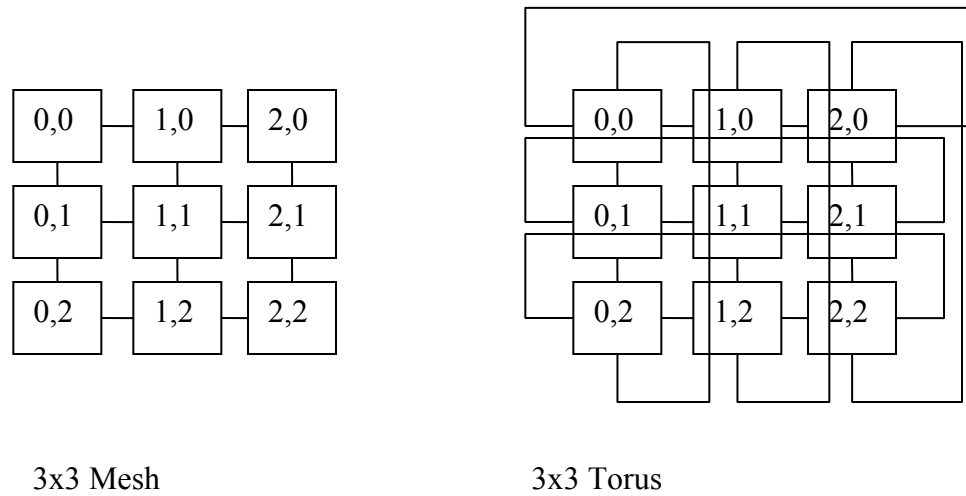


Figure 31 - Mesh vs. Torus

As mentioned in [31], the ring topology is one of the least studied NoC topologies and was chosen for that reason. Conversely, mesh and torus topologies are the most studied cases. The ring topology's nodes have two ports, which daisy-chain the connections until a loop is formed. Figure 32 illustrates an example ring topology.

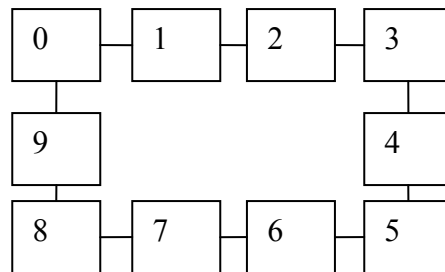


Figure 32 - 10-Node Ring Example

For the ring topology, one router is included per IP core. The 4x4 torus was designed to contain n^2 nodes and since the root of fourteen is irrational, then there will be two pure routing nodes.

5.2 Placement and Routing

The goal of placement and routing in both topologies was to have the smallest path between cores, while avoiding congestion. This was done somewhat arbitrarily as seen in Figure 33. Routers 3 and 10 are pure routing nodes, with no cores attached.

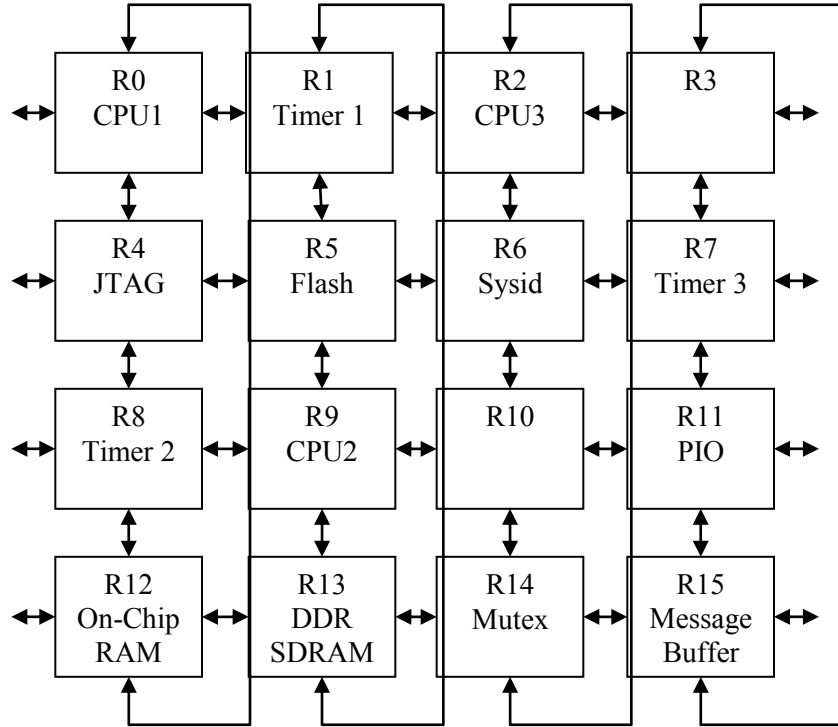


Figure 33 - Torus Core Placement

To aid the process of routing, the core placement is summarized in Figure 34, which better visualizes the wrapped connectivity property of the torus topology that is available for routing.

F	S	T	J	F	S	T	J	F	S
C		P	T	C		P	T	C	
D	X	M	O	D	X	M	O	D	X
T	C		C	T	C		C	T	C
F	S	T	J	F	S	T	J	F	S
C		P	T	C		P	T	C	
D	X	M	O	D	X	M	O	D	X
T	C		C	T	C		C	T	C
F	S	T	J	F	S	T	J	F	S
C		P	T	C		P	T	C	

C = CPU
 T = Timer
 J = JTAG UART
 F = Flash
 S = Sysid
 P = PIO
 O = On-Chip RAM
 D = DDR SDRAM
 X = Mutex
 M = Message Buffer

Figure 34 - Placement for Routing

In general, each CPU has dependencies for on other cores and is summarized in Table 4. For example, CPU 1 depends on Timer 1, but CPU 2 does not depend on Timer 2. While the routing allows for a path between non-dependent cores, in general this should *not* happen and is merely included for fullness. Table 4 summarizes these dependencies, which apply to both topologies.

	CPU 1	CPU 2	CPU 3
Flash	Yes	Yes	Yes
DDR	Yes	Yes	Yes
Timer 1	Yes	No	No
Timer 2	No	Yes	No
Timer 3	No	No	Yes
JTAG UART	Yes	No	No
Mutex	Yes	Yes	Yes
Message Buffer	Yes	Yes	Yes
PIO	No	Yes	Yes
On-Chip	Yes	No	No

RAM			
Sysid	Yes	Yes	Yes

Table 4 - Master-Slave Dependencies

An overview of the routing goals is shown in Figure 35, which demonstrates the routing to each dependent core from the three CPUs. Figure 36 shows all the routing paths for all 14 destinations. The extra adapters used for bridging and separate bus and data bus paths are the same as destinations 0, 9 and 2 for CPUs 1, 2 and 3, respectively.

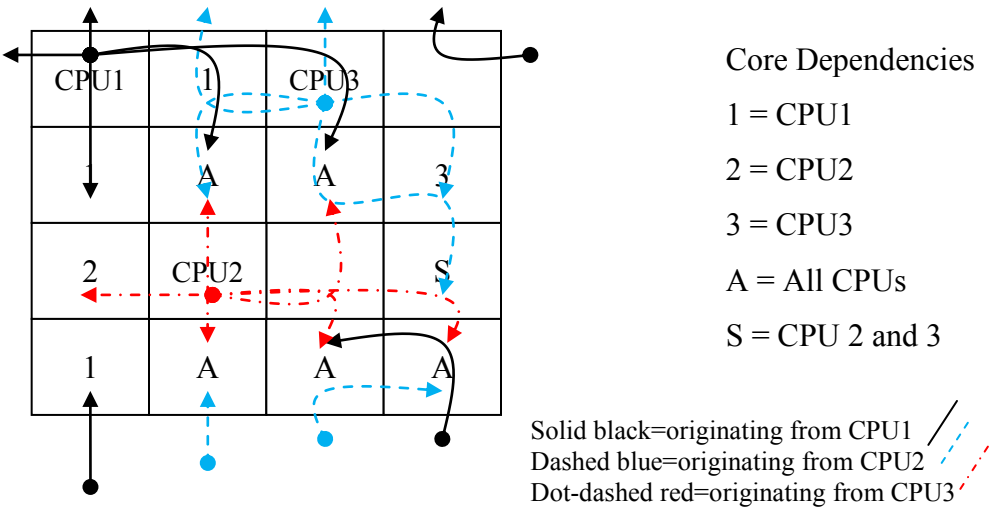


Figure 35 - Torus CPU Routing

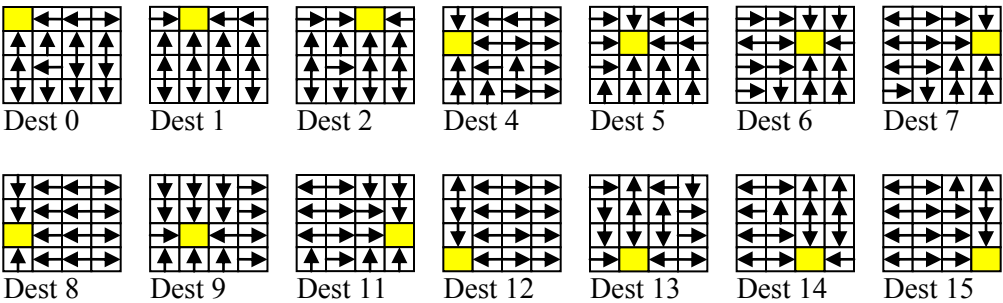


Figure 36 - Torus Source Routing Paths

The placement of the cores in the ring network was relatively straight forward, where cores were placed in order to minimize routes and keep high throughput cores close together. The placement is summarized in Figure 37 and the routing is summarized in Figure 38.

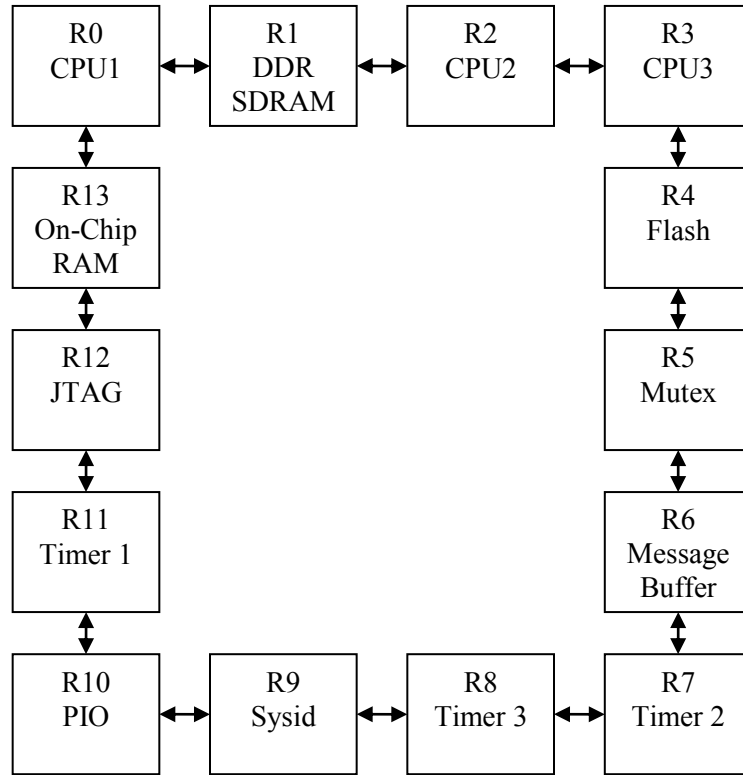


Figure 37 - Ring Placement

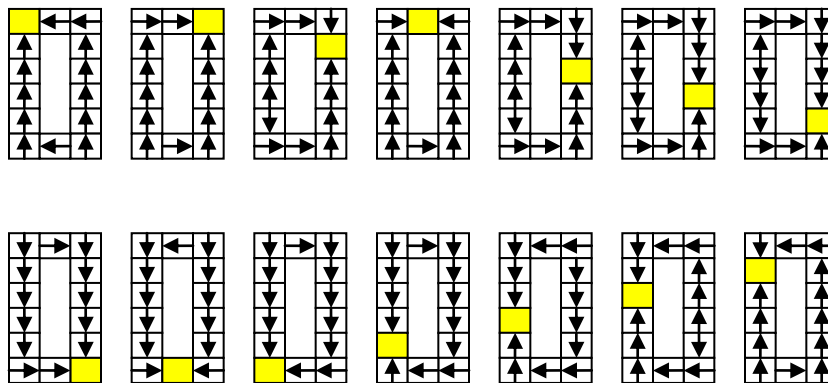


Figure 38 - Ring Routing

5.3 NoC Generator

In order to ease the production of the topologies, a C++ program was written, called *noc_gen*. *Noc_gen* accepts input parameters from the user either by a keyboard

peripheral or a file input. The generator then uses VHDL components, signals and mapping to repeatedly instantiate and connects the routers and adapters. Any user input is saved to a file for easy reproduction. The input parameters and ordering of noc_gen is listed below.

- a) File name
- b) Number of slaves (adapters)
- c) Number of masters (adapters)
- d) Flit size
- e) Routing type (1=source, 2=xy)
- f) Data bus width
- g) Address width
- h) Cycle tag width
- i) Data tag width
- j) Select line width
- k) Destination width
- l) Source width
- m) Use bursts? (true or false)
 - a. If true – Burst count width
- n) Number of routers
 - a. If burst true - Fast burst?
 - b. Burst depth
 - c. CTI least-significant bit
 - d. BTE least significant bit
 - e. Burst tag enable
- o) Number of ports
- p) Number of local ports
 - a. Repeat O and P for each router
- q) For each router port – Connect to router #
- r) Connect to port #
 - a. Repeat Q and R for all router ports, until they are all connected
- s) For each local port – Connect to adapter

- a. Repeat S for each local port
- t) Routing table – Input for each router
- u) FIFO depth
- v) Source Number
- w) No acknowledge (no_ack)
 - a. Repeat u-w for all master adapters
- x) FIFO depth
- y) No_ack
 - a. Repeat x and y for all slave adapters
- z) Routing table
- aa) Routing table ranges
 - a. Repeat Z and AA for all Address To Destination modules (adr2dest)
- bb) Number of beats
- cc) Line wrap burst (true or false)
 - a. Repeat BB and CC for all Avalon-Wishbone modules (awb)

Only two NoC architectures (torus and ring) were implemented with *noc_gen*. The flit size was varied using SoPC Builder's VHDL generic assignment functionality.

5.4 Summary

This chapter covered the details of the NoC architectures implemented in this thesis. The details of the IP core placement within the NoC designs and routing algorithms between them were described and discussed. Chapter 5 concludes with a discussion of the NoC Generator C++ program which was used to fabricate the NoC architectures. Chapter 6 evaluates the two NoC architectures with different metrics and synthesizes the NoCs and discrete components.

Chapter 6

Component Evaluation and Architecture Comparison

Chapter 6 evaluates the NoC designs and components in this thesis by first synthesizing the discrete NoC components for a Stratix II FPGA. This follows with a discussion of the evaluation metrics utilized. The chapter concludes with in-depth details and discussion of the NoC evaluations.

6.1 Design Space Exploration of Adapter and Router

The area, clock frequency and power usage of the router and adapter is measured from Quartus II's [15] synthesis tools to yield ALUT and register usage. For the router, the port size is varied from 2 to 6 with a constant flit size of 64 bits, and conversely flit size is varied from 4 to 64 in binary incremental, with a constant port number of 5. The two adapters have their flit sizes varied from 4 to 64. Power is measured with a constant signal change rate of 12%. The FIFO buffer module is also synthesized in order to observe area effects as flit size increases. Figure 39, Figure 40, Figure 41 and Figure 42 present the FPGA resource usages of the router, master adapter, slave adapter and FIFO with a varying flit size. Figure 43 presents the router FPGA resource usages as the number of ports changes.

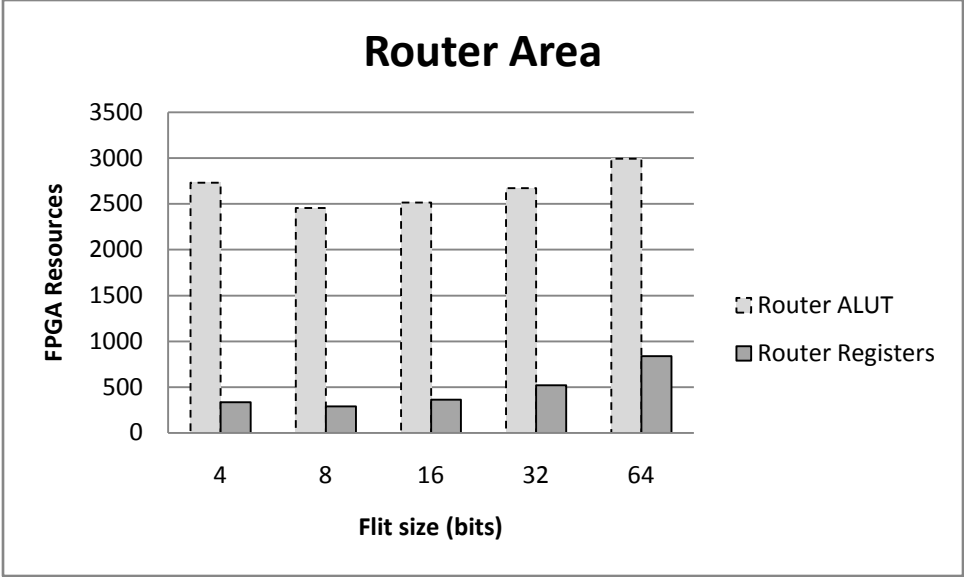


Figure 39 - Individual Router Area vs. Flit size

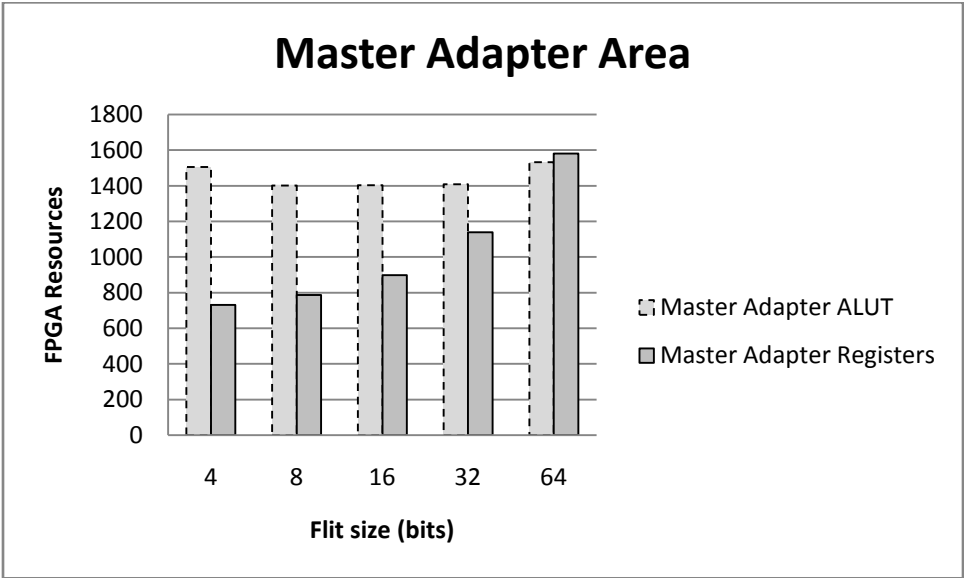


Figure 40 - Master Adapter Area

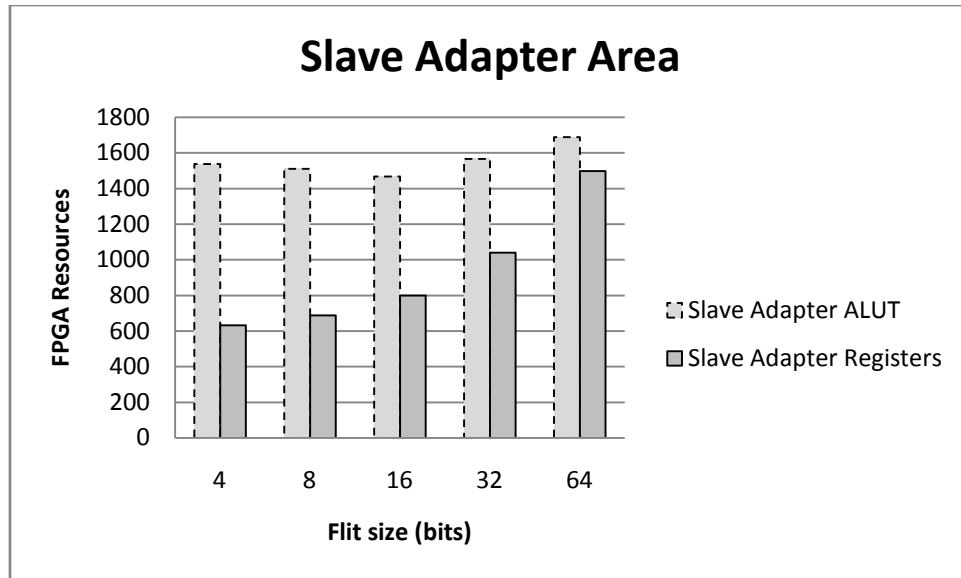


Figure 41 - Slave Adapter Area

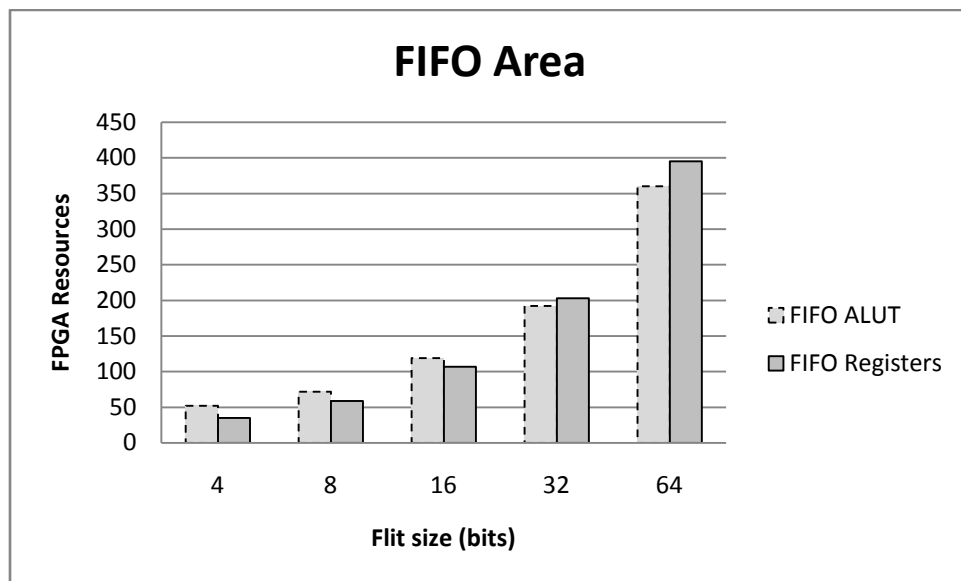


Figure 42 - FIFO Area

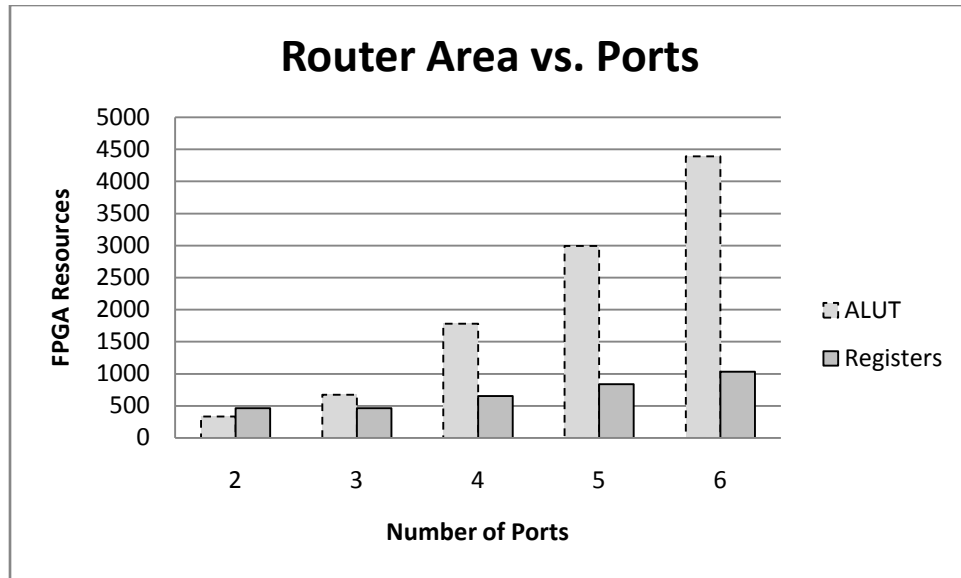


Figure 43 - Router Area vs. Number of Ports

In order to decrease compilation time, the router is set to 3 ports for power measurements. The PowerPlay Power Analyzer Tool in Quartus is set to use a 12.5% I/O signal toggle rate. These final results are fitted and timing is analyzed in Figure 44, Figure 45 and Figure 46.

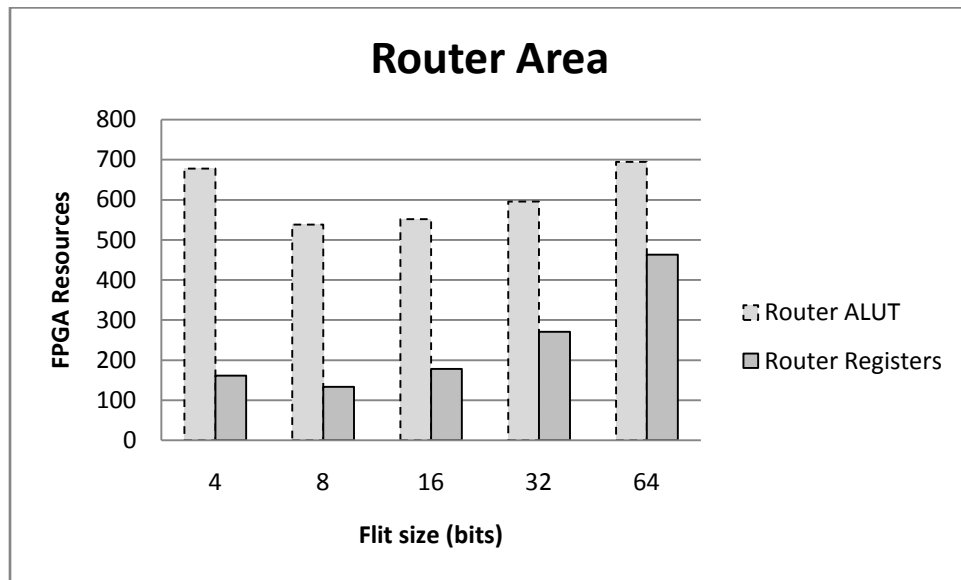


Figure 44 - Router Area vs Flit Size

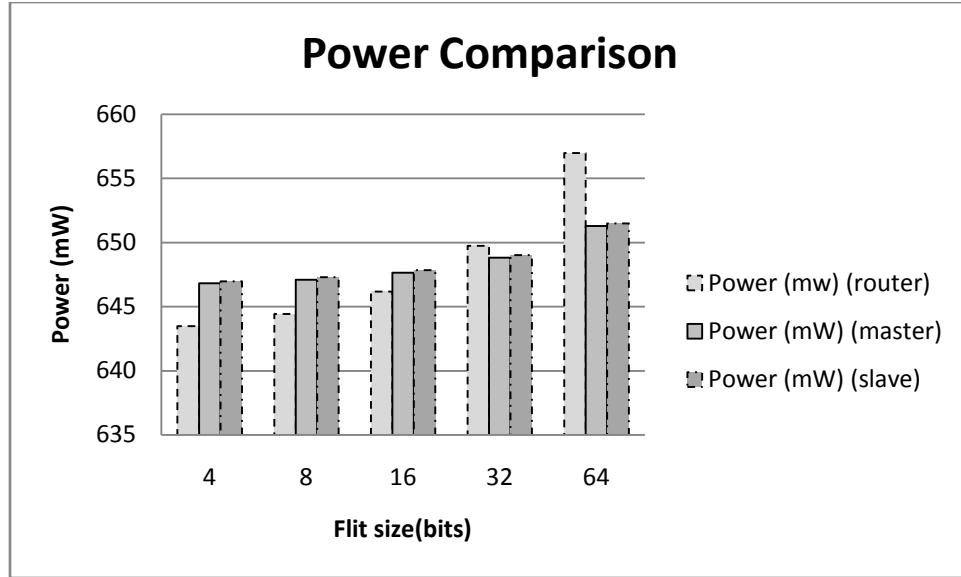


Figure 45 - Power Usage of Discrete NoC Components

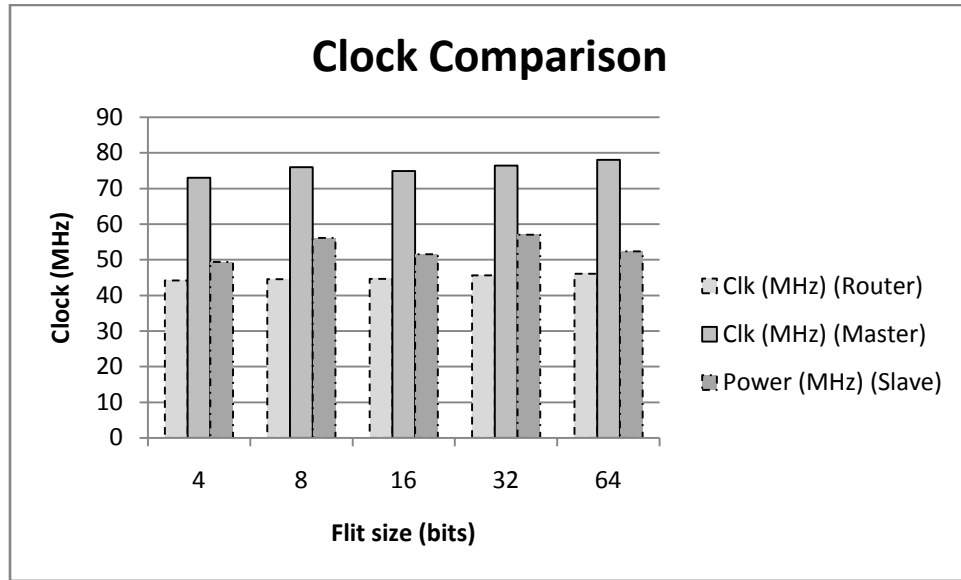


Figure 46 - Clock Frequency of Discrete NoC Components

6.2 Experimental Framework and Evaluation Metrics

The performance of the NoCs is measured with two metrics – Packet latency and throughput. Packet latency is measured as the number of cycles between an adapter forming a packet and the receiving adapter absorbing a packet [37]. The packet, a counter's value and a sender/receiver bit is outputted with VHDL's file IO system. The sender and receiver bits are matched in a C++ program and the counter values are subtracted, resulting in the number of cycles a packet requires in order to

traverse the NoC. The C++ code also outputs the total amount of each packet type. Throughput is calculated with the following formula:

$$Throughput = \frac{(Number\ of\ messages * Message\ Length)}{Number\ of\ IPs * Total\ time\ taken}$$

Number of messages is the total amount of packets sent in the simulation – an accumulator in each adapter counts the total messages sent at the point of (de)packetization, which is shown in Modelsim [17]. Message length is the average number of flits contained in a message. Number of IPs is the total count of functional cores attached to the network (it is a constant of 14 in this case). Total time taken is the number of cycles required to run the simulation. Since the program loops indefinitely, the total time taken is marked at the final write to DDR SDRAM.

6.3 Comparison of Torus and Ring

The interrupt timers were originally set to 0.5ms and there were issues with the 8 bit flit size system running incorrectly due to the NoC delay being too long and hence Nios II [16] data becomes corrupted. 1 ms interrupt times were used for all measurements. This decreases throughput since the benchmark takes longer to run, hence *Total time taken* increases. Regardless of interrupt time, packet latency measurements were unaffected. The average latency of both NoCs is shown in Figure 47, with the torus topology clearly having the longest latency, regardless of flit size.

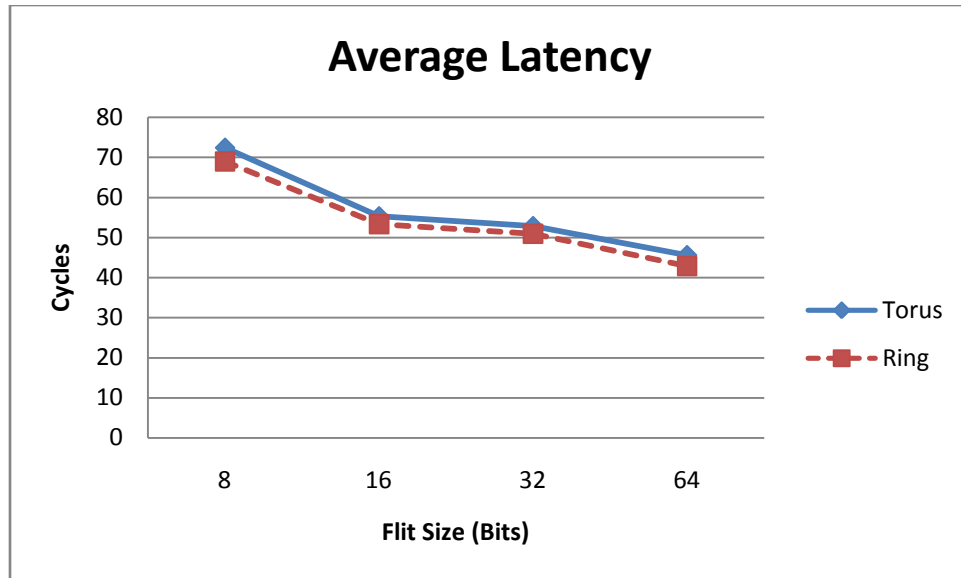


Figure 47 - Average Latency of Two NoC Topologies

The total time for the software benchmark to complete is shown in Figure 48. While this data is not too useful on its own, it is useful for future researchers interested in comparing the raw data. The ring NoC with 64 bit flit sizes takes 1154153 cycles, which is 197919 cycles less than the torus architecture. The goal of this research is not to prove whether or not bus systems are superior or not and hence those systems are not compared. Research has already been done on the topic, such as in [40], which demonstrates increased performance and scalability as the system increases.

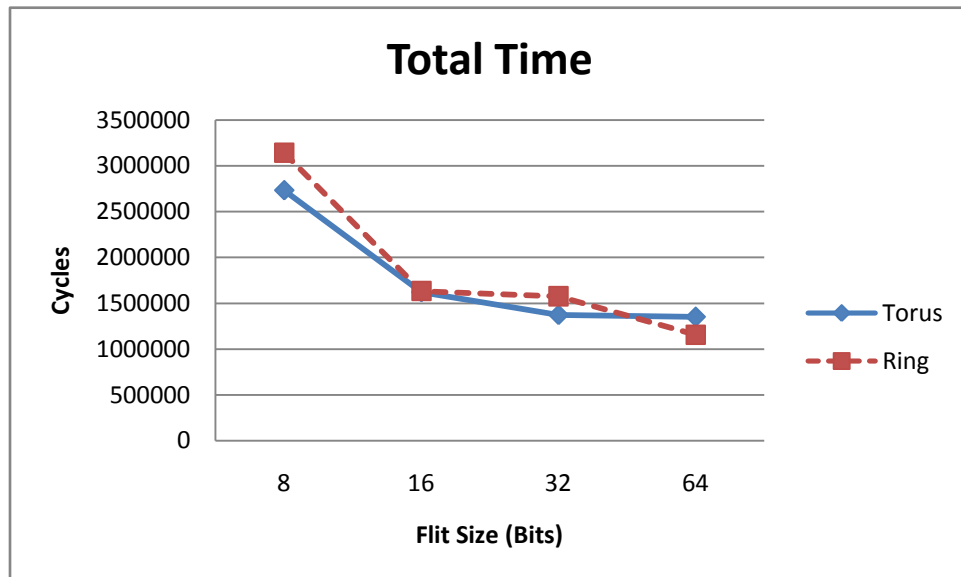


Figure 48 - Total Time to Complete Nios II Program

The throughput of the two NoCs, previously discussed as flits per cycle per core, is shown in Figure 49, demonstrating somewhat competitive performance for both NoCs and its impact on flit size. The bandwidth is shown in Figure 50, which is shown for comparative purposes.

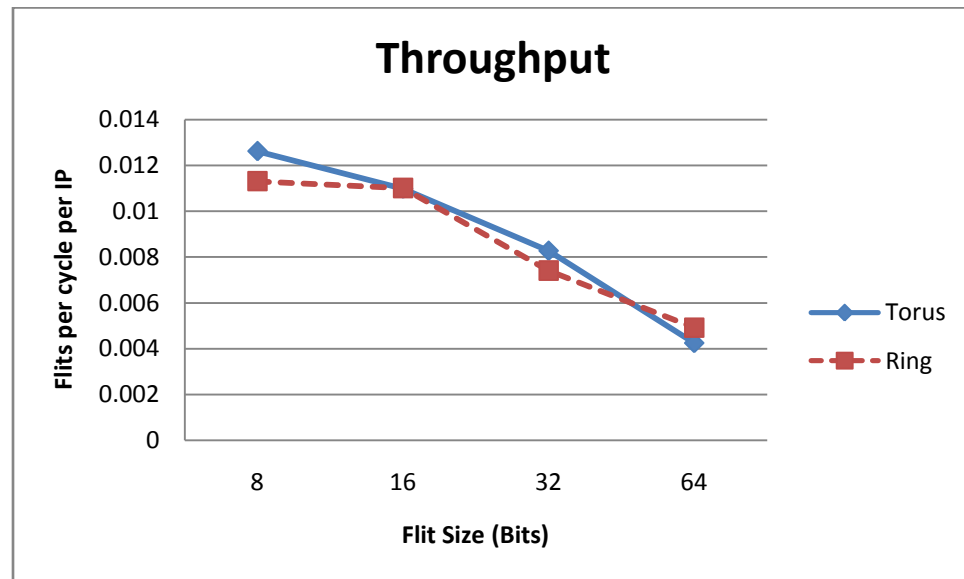


Figure 49 – Throughput Comparison of Two NoC Topologies

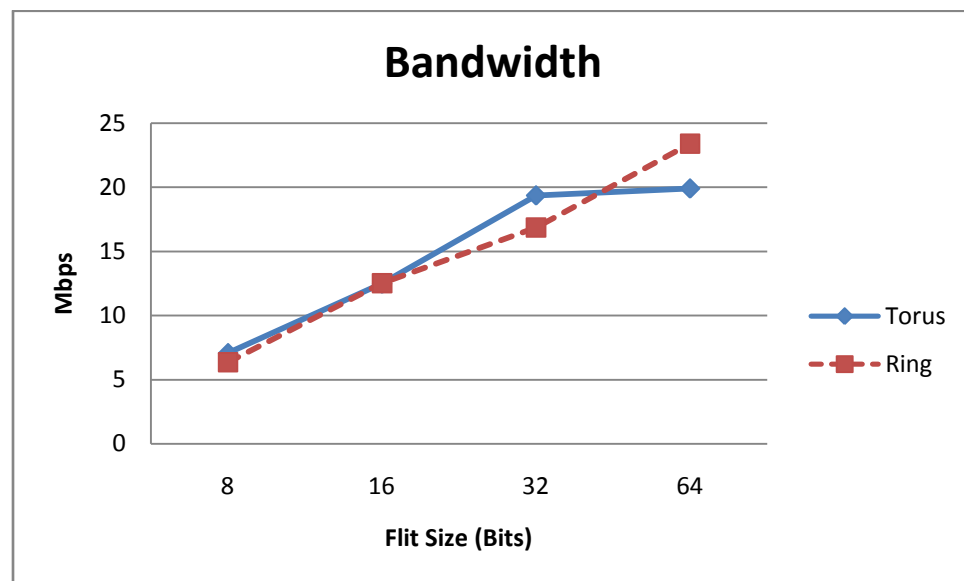


Figure 50 – Bandwidth Comparison of Two NoC Topologies

The FPGA resource usage of the two NoCs is shown in Figure 51 and Figure 52, showing ALUT and register usage, respectively. Clearly, the ring topology uses less area throughout.

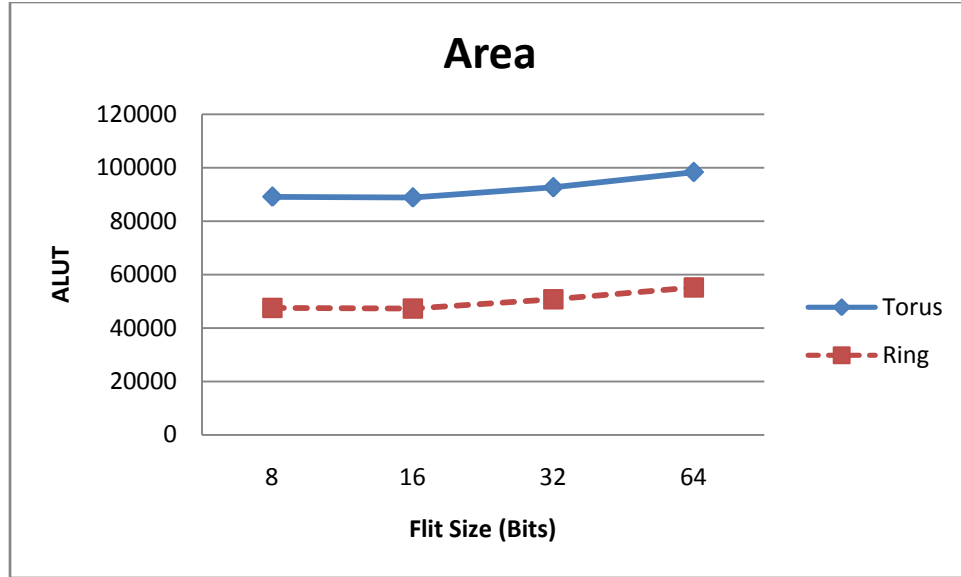


Figure 51 - NoC Area – ALUTs

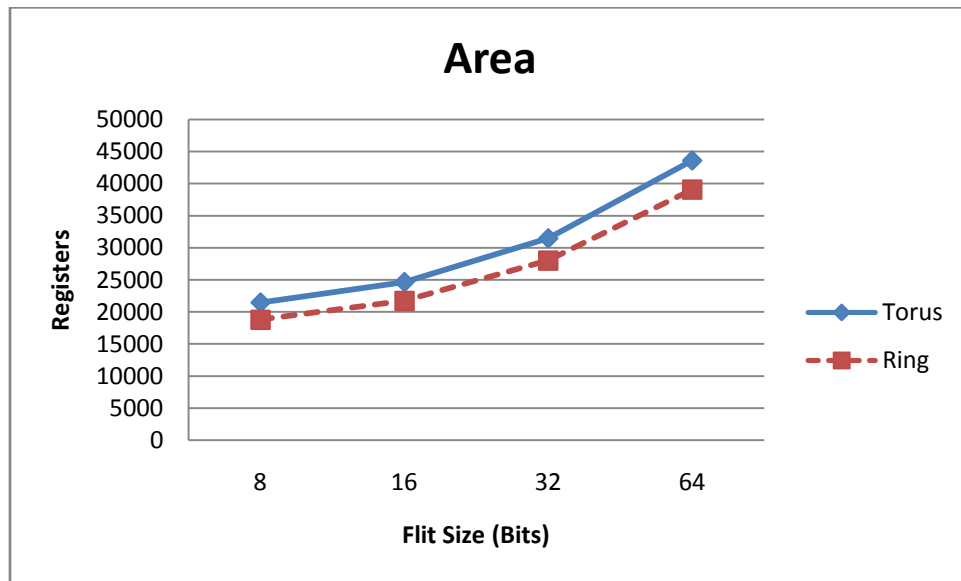


Figure 52 - NoC Area - Registers

These results are compared with [37], where five topologies are used (BFT, Cliché, Spin, folded torus and octagon), with traffic generators. The traffic generators were designed to emulate a multi-processor system. While throughput measurements average about 0.5 to 0.8 in [37], our throughput measurements range 0.005 to 0.02. Packet latency in [37] averages around 30 cycles, while this research ranges from 45 to 70.

These higher packet latencies can be attributed to the placement of the high-priority items such as DDR SDRAM and the mutex. The DDR SDRAM makes up the

majority of the traffic and the placement in the torus topology has room for optimization. The ring topology has a distinct disadvantage in that there are only two neighbours allowed due to the number of ports being set to 2 for all routers. This means one of the CPUs is required to have a 2-hop delay for SDRAM transactions. Regardless of the ring's downside, the average packet latencies were overall lower due to the somewhat improved placement. This means if the placement of the DDR SDRAM were fully optimized, then overall packet latency would improve. PWR also includes real routers and adapter delay into packet latency, while [37] does not.

One must consider that a real system is used, which is based on a 1 ms timer. This means there will be moments when the CPUs are simply waiting for an interrupt and no requests are made (roughly 900us delays), which makes overall *Total time taken* increase. In [37], the traffic generator creates “dead traffic” moments but is not accurate according to real traffic [22]. Concerning bandwidth, results from [37] were in the Gbps range whereas PWR's were in the lower Mbps. Again, this is due to bandwidth relying on the *Total time taken*. Concerning the comparison of ring versus torus topologies as well as results versus flit size - It is interesting to note that for bandwidth, the torus topology begins to level off after 32 bits, whereas the ring topology still increases – this is most likely due to the placement of cores resulting in higher average latency. For average message latency, the plots for both topologies are relatively the same, with the torus having an overall higher latency. It is difficult to compare these two plots since message latency is affected by placement and routing in both topologies. Finally, throughput, which is a measure of efficiency, remains on the torus' side except for 64 bit flit sizes, where ring takes over. Again this is most likely due to placement of the DDR SDRAM creating a higher average latency in the torus topology.

Rather expectedly, the torus architecture takes up significantly more FPGA resources, compared to ring. This is due to each router having twice as many ports, as well as the additional two pure routing nodes. This shows a classic case of performance versus resources, where the low performance of the single-digit flit sizes results in the lowest resource count and vice versa for 64 bit flit sizes.

In order to recommend an optimal channel width based off the results of the torus and ring evaluations, one must compare the effect of channel width with respect to the

area resource usage difference. Since a flit size of 16 bits uses the least ALUT resources, it was thus chosen as the basis of comparison between the other flit sizes. The difference in performance between the flit size of attention and 16 bit flit sizes is divided by the difference in area between the two. The following two figures illustrate these plots.

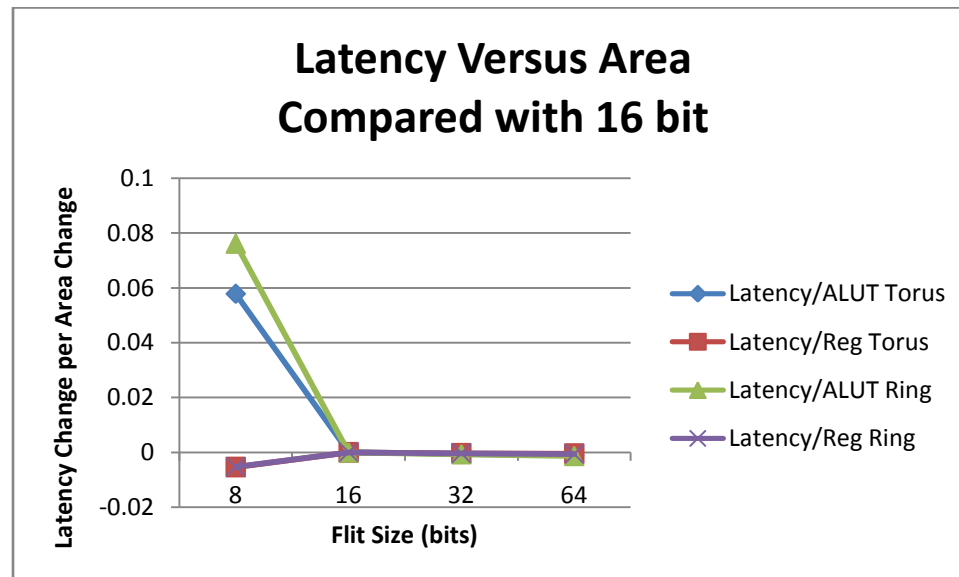


Figure 53 - Latency Change vs. Area Change, with Respect To 16 Bit Flit Size

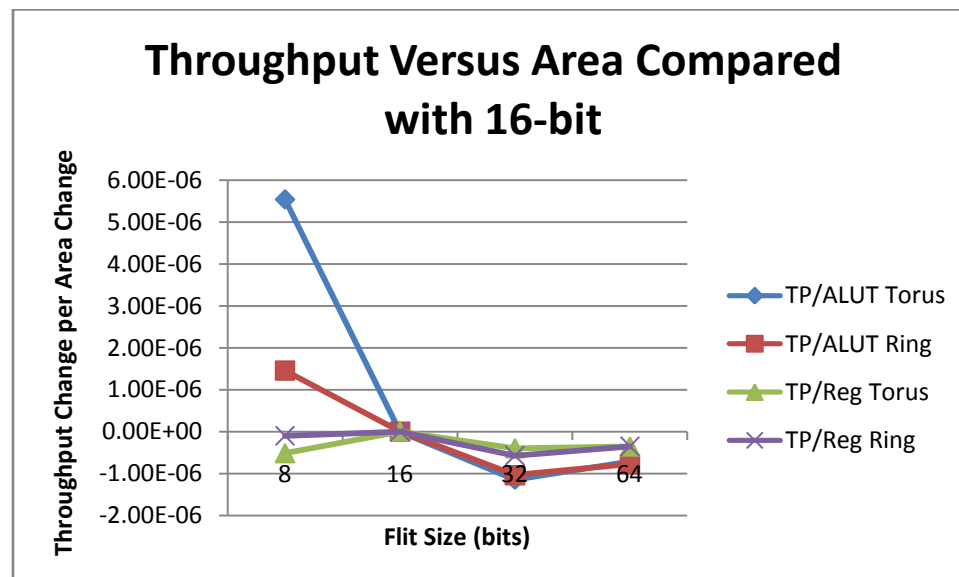


Figure 54 - Throughput Change vs Area Change, With Respect To 16 Bit Flit Size

Ideally, message latency should be reduced per added resource. Clearly, increasing the flit size does not improve the latency per area. It can be seen that with a flit size of 8 bits, the latency difference per area difference is significant for ALUT resources and decreases

with register resources. This means that if a designer wishes to have the most efficient use of registers for message latency, an 8 bit flit size is recommended, while a 16 bit flit size has the best efficiency for general area resources for message latency.

For the efficiency of area for throughput, a high throughput difference per added resource is ideal. It can be seen that an 8 bit flit size offers the highest ALUT efficiency for throughput, and the 32 and 64 bit flit sizes offer lower area efficiency for throughput compared with 8 and 16 bit flit sizes. From these results, it can be deduced that a 16 bit flit size is an optimal channel width to use for general designs, due to the efficiency of an FPGA's resources for message latency and throughput. For area-constrained environments, an 8 bit flit size is recommended due to its higher area resource efficiency at the expense of larger message latencies.

6.4 Summary

This chapter evaluated the design of the NoC architectures and its discrete components. The evaluation metrics involved in these evaluations were discussed, and the results of these evaluations were discussed. Chapter 7 discusses the conclusions that were drawn from this thesis.

Chapter 7

Conclusions and Future Work

As integrated circuit technology expands, allowing for larger and increasingly complicated systems-on-chip, the traditional bus-based communication system becomes cumbersome and restricting. As the communication architecture shifts towards the network-on-chip paradigm, it becomes apparent that there is a large design space available to designers. Combine an NoC's massive parameter space with various applications and it becomes clear that there is no single solution.

While traffic generators provide a reasonable means of evaluating NoC designs, they are not accurate [22]. Since real traffic and real systems are rarely tested in NoC research, this became an important task to pursue in this thesis. Very little work has been done implementing NoCs in traditional SoC design software, such as Altera's SOPC Builder. Lastly, certain topologies remain largely popular, such as mesh and torus, while others require additional research, such as hierarchy and ring, which is the reason why the torus and ring topologies are compared.

Key research contributions include developing a realistic system benchmark for evaluating an NoC. Altera's SOPC Builder provided a satisfactory means of instantiating the NoCs, but there were problems to overcome. Bridging of a single adapter to multiple memory modules was not possible and hence caused additional resources to be wasted. The lack of control over the bus arbitrator also caused problems, resulting in adapters not knowing whether or not the control outputs will be received; again, this resulted in additional adapters, wasting resources.

Another unique contribution includes the glue logic between Wishbone and Avalon communication fabrics. While single-transfer transactions are extremely similar

between the two interfaces, the block transfers include a large difference. While Wishbone requires the acknowledge signal and its assertion in order to start another transaction, the Avalon block transfer rule allows for multiple transfers to be ‘queued’ without a preceding acknowledgement signal.

While worm-hole switching router designs are relatively common, the effect of channel width on high level evaluation metrics is an open research area [22]. This was studied by analyzing the flit width versus throughput, average packet latency, area, power and clock frequency. The flexibility and parameter selection of the router is also a unique trait, which provides a frame for future research.

Interfacing a core using a standard socket with the NoC is not to be overlooked. The adapter, originally designed for Wishbone, was modified for use with the Avalon interface. By providing a huge amount of flexibility through signal vector widths, FIFO depths and more, the adapter functions in most SoC designs and offers a large design space that should not be overlooked.

Concerning the topology comparison, overall the ring topology utilizes the least FPGA resources, provides the lowest packet latency and competitive throughput versus the torus for the system benchmark. From a designer perspective, the ring topology was also much easier to route, map and place. Conversely, the torus topology provides greater flexibility by means of the larger node neighbour count and flexible routing paths.

It can be seen from the results that flit size has a large effect on various evaluation metrics for NoCs and related components. A low average latency results from high flit size, due to the smaller worm being transmitted. Throughput has an interesting relationship with flit size, as a low flit size results in the torus having the largest throughput. A low flit size results in larger worms and hence the NoC becomes more active. Interestingly, flit size seems to have little effect on the clock frequency of the individual NoC components, while having a predictable effect on area and power. From the area efficiency results, it can be deduced that a 16 bit flit size is an optimal channel width to use for general designs, due to the efficiency of an FPGA’s resources for message latency and throughput. For area-constrained environments, an 8 bit flit size is recommended due to its higher area resource efficiency at the expense of larger message latencies. A 64 bit flit size offers the lowest message latency but uses the most FPGA

resources.

Future work includes implementing additional system benchmarks in order to further evaluate the two topologies. Additional topologies can also be implemented, including hierarchy, star and customized architectures. While the effect of FIFO depth was compared with resource usage, it would be interesting to compare it with high level performance metrics, such as throughput and packet latency. There are more NoC parameters, such as switching, arbitration, routing and placement that will provide a suitable area for research.

References

1. **Vahid, Frank and Givargis, Tony.** *Embedded System Design - A unified hardware/software introduction*. John Wiley & Sons, Inc., 2002. 81-265-0837-X.
2. **Guerrier, Pierre and Greiner, Alain.** *A generic architecture for on-chip packet-switched interconnections*. Paris, France, 2000. Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000. pp. 250-256.
3. **Sodan, A, et al.** *Parallelism via multithreaded and multicore CPUs*. Computer. November 30, 2009, Vol. PP, 99.
4. **Bjerregaard, Tobias and Mahadevan, Shankar.** *A survey of research and practices of Network-on-Chip*. Issue 1, 2006, ACM Computing Surveys (CSUR), Vol. 38, pp. 1-51.
5. **Hilton, C. and Nelson, B.** *PNoC: a flexible circuit-switched NoC for FPGA-based systems*. Issue 3, 2006, IEE Proceedings - Computers and Digital Techniques, Vol. 153.
6. **Bjerregaard, T. and Sparso, J.** *A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip*. 2005. Design, Automation and Test in Europe, 2005. Proceedings. Vol. 2, pp. 1226 - 1231.
7. **Goossens, K., Dielissen, J. and Radulescu, A.** *AEthereal network on chip: concepts, architectures, and implementations*. Issue 5, 2005, Design & Test of Computers, IEEE, Vol. 22, pp. 414 - 421.
8. **Dall'Osso, M., et al.** *Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs*. 2003. Computer Design, 2003. Proceedings. 21st International Conference on. pp. 536 - 539.

9. **Carara, Everton, Moraes, Fernando and Calazans, Ney.** *Router architecture for high-performance NoCs*. Copacabana, Rio de Janeiro, 2007. SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design. pp. 111 - 116.
10. **Varatkar, G.V. and Marculescu, R.** *On-chip traffic modeling and synthesis for MPEG-2 video applications*. Issue 1, 2004, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Vol. 12, pp. 108 - 119.
11. **Silicore corporation; opencores.org.** WISHBONE, Rev.B3 Specs. *Wishbone::OpenCores*. [Online] July 9, 2002. [Cited: 05 03, 2010.] http://opencores.org/downloads/wbspec_b3.pdf.
12. **Altera Corporation.** Altera Avalon Interface Specifications. *Literature: SOPC Builder*. [Online] April 2009. [Cited: May 3, 2010.] http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.
13. **Usselmann, Rudolf.** OpenCores SoC bus review. *Wishbone:: OpenCores*. [Online] January 9, 2001. [Cited: May 3, 2010.] http://opencores.org/downloads/soc_bus_comparison.pdf.
14. **Altera Corporation.** Quartus II Handbook Version 9.1 - Volume 4 SOPC Builder. *Quartus II Development Software Literature*. [Online] November 2009. [Cited: May 5, 2010.] http://www.altera.com/literature/hb/qts/qts_qii5v4.pdf.
15. **Altera Corporation.** Quartus II Handbook Version 9.1. *Design Software*. [Online] November 2009. [Cited: May 5, 2010.] http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf.
16. **Altera Corporation.** Nios II Software Developer's Handbook. *Literature: Nios II Processor*. [Online] November 2009. [Cited: May 5, 2010.] http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf.
17. **Mentor Graphics Corporation.** ModelSim SE User's Manual. *ModelSim SE | Verilog, VHDL, SystemVerilog Design & Simulation | ModelSim - Advanced Simulation*

and Debugging:. [Online] 2010. [Cited: May 5, 2010.]

http://portal.model.com/modelsim/resources/references/modelsim_se_user.pdf.

18. **Bertozi, D. and Benini, L.** *Xpipes: a network-on-chip architecture for gigascale systems-on-chip*. Issue 2, 2004, Circuits and Systems Magazine, IEEE, Vol. 4, pp. 18 - 31.
19. **Dehyadgari, Masood, et al.** *A new protocol stack model for Network on Chip*. Issue 2-3, Karlsruhe, 2006, IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architecture, Vol. 00, pp. 1-2.
20. **Sgroi, M., et al.** *Addressing the System-on-a-Chip interconnect woes through communication-based design*. 2001. Design Automation Conference, 2001. Proceedings. pp. 667 - 672.
21. **Altera Corporation.** Nios II Processor Reference Handbook. *Literature: Nios II Processor*. [Online] November 2009. [Cited: April 20, 2010.]
<http://www.altera.com/literature/lit-nio2.jsp>.
22. **Ogras, Umit Y., Hu, Jingcao and Marculescu, Radu.** *Key research problems in NoC design: A holistic perspective*. Jersey City, NJ, USA, 2005. Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. CODES+ISSS '05. pp. 69-74.
23. **Hu, Jingcao and Marculescu, R.** *Energy- and performance-aware mapping for regular NoC architectures*. 2005, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 551-562.
24. **Murali, S. and De Micheli, G.** *Bandwidth-constrained mapping of cores onto NoC architectures*. 2004. Proceedings Design, Automation and Test in Europe Conference and Exhibition. Vol. 2, pp. 896 - 901.
25. **Geng, Luo-Feng, et al.** *Prototype design of cluster-based homogeneous Multiprocessor System-on-Chip*. Hong Kong, 2009. Conference on Anti-counterfeiting,

Security, and Identification in Communication, 2009. ASID 2009. 3rd International. pp. 311-315.

26. **Dall'Osso, M., et al.** *Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs*. 2003. Computer Design, 2003. Proceedings. 21st International Conference on . pp. 536-539.

27. **Ehliar, A. and Liu, Dake.** *An FPGA based open source Network-on-Chip architecture*. Amsterdam, 2007. International Conference on Field Programmable Logic and Applications, 2007. pp. 800-803.

28. **Joven, J., et al.** *xENoC - An eXperimental Network-On-Chip environment for parallel distributed computing on NoC-based MPSoC architectures*. Toulouse, 2008. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing. pp. 141-148.

29. **Saastamoinen, I., Siguenza-Tortosa, D. and Nurmi, J.** *Interconnect IP node for future system-on-chip designs*. Christchurch, 2002. The First IEEE International Workshop on Electronic Design, Test and Applications, 2002. Proceedings. pp. 116 - 120.

30. **Srinivasan, K., Chatha, K.S. and Konjevod, G.** *Application specific Network-on-Chip design with guaranteed quality approximation algorithms*. Yokohama, 2007. Asia and South Pacific Design Automation Conference. pp. 184-190.

31. **Salminen, Erno, Kulmala, Ari and Hamalainen, Timo D.** Survey of Network-on-Chip proposals. *OCP-IP: White Papers Page*. [Online] April 2008. [Cited: April 22, 2010.] http://ocpip.org/white_papers.php.

32. **Kapre, N., et al.** *Packet switched vs. time multiplexed FPGA overlay networks*. Napa, CA, 2006. Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on. pp. 205 - 216.

33. **Peh, Li-Shiuan and Dally, W.J.** *A delay model for router microarchitectures*. Issue 1, 2001, IEEE Micro, Vol. 21, pp. 26-34.

34. **Xing, Xu, et al.** *Porting from Wishbone bus to Avalon bus in SoC design*. Xi'an, 2007. Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference on. Vol. 1, pp. 862-865.
35. **Minhass, Wajid Hassan, Öberg, Johnny and Sander, Ingo.** *Design and implementation of a plesiochronous multi-core 4x4 Network-on-Chip FPGA platform with MPI HAL support*. Stockholm, Sweden, 2009. Proceedings of the 6th FPGAworld Conference. pp. 52-57.
36. **Moraes, Fernando, et al.** *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. Issue 1, Amsterdam, The Netherlands : Elsevier Science Publishers B. V., 2004, Vol. 38. 0167-9260 .
37. **Pande, Partha Pratim, et al.** *Performance evaluation and design trade-offs for Network-on-Chip interconnect architectures*. Issue 8, 2005, Computers, IEEE Transactions on, Vol. 54, pp. 1025 - 1040.
38. **Sethuraman, Balasubramanian, et al.** *LiPaR: A light-weight parallel router for FPGA-based networks-on-chip*. Chicago, 2005. Proceedings of the 15th ACM Great Lakes symposium on VLSI. pp. 452 - 457.
39. **Dally, W.J. and Towles, B.** *Route packets, not wires: on-chip interconnection networks*. 2001. Design Automation Conference, 2001. Proceedings. pp. 684 - 689.
40. **Zeferino, C.A., et al.** *A study on communication issues for systems-on-chip*. 2002. Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on. pp. 121 - 126.

VITA AUCTORIS

Matt Murawski was born in Windsor, Ontario in 1985. In 2007, he earned his B.A.Sc at the University of Windsor in Electrical Engineering in Windsor, Canada. He is currently a candidate at the University of Windsor for the M.A.Sc program. His interests include digital hardware design, computers, embedded systems and FPGAs.