AN ANALYTICAL MODEL DESCRIBING THE PERFORMANCE OF APPLICATION-SPECIFIC NETWORKS-ON-CHIP ON FIELD-PROGRAMMABLE GATE ARRAYS

by

Jason Shek-Yen Lee
B.A.Sc, Simon Fraser University, 2007

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Applied Science
in the School
of
Engineering Science

APPROVAL

Name:	Jason Shek-Yen Lee
Degree:	Master of Applied Science
Title of Thesis:	An Analytical Model Describing the Performance of Application-Specific Networks-on-Chip on Field-Programmable Gate Arrays

Examining Committee:

Chair:

Dr. John Bird
Professor, School of Engineering Science

Dr. Lesley Shannon
Senior Supervisor
Assistant Professor, School of Engineering Science

Dr. Rick Hobson
Supervisor
Professor, School of Engineering Science

Dr. Rodney Vaughan
Internal Examiner
Professor, School of Engineering Science

Date Approved/Defended: *January 14, 2010*

Abstract

Modern Field-Programmable Gate Arrays (FPGAs) are now used to implement complex Systems-on-Chip (SoCs) and more recently Networks-on-Chip (NoCs). NoCs consist of computing nodes that are connected via switches or routers to a network of point-to-point links, which define its topology. Appropriate topology choices for Application-Specific Integrated Circuits (ASICs) have been investigated, but due to an FPGA's fixed interconnect fabric, these conclusions are not necessarily applicable. Our research investigates how a commercial FPGA's fixed interconnect and CAD flow constrain the performance of NoCs based on a set of design parameters. We develop an analytical model that predicts the performance for both homogeneous and heterogeneous NoCs with a geometric mean error of 4.68% for Xilinx Virtex 2 Pro, Virtex 4, Virtex 5, and Virtex 6 FPGAs, and with a geometric mean error of 5.12% for Altera Stratix III and Stratix IV FPGAs.

Keywords: Networks-on-Chip; NoCs; FPGAs; Analytical Model; Systems-on-Chip; SoCs; Performance; Topologies; Routability; Application-Specific; Homogeneous; Heterogeneous

Acknowledgments

While I was told to try to keep this section short, I cannot go without acknowledging the many people that have helped me in the past few years. First, I would like to thank my family for supporting me and dealing with my extended absences from home. I am especially thankful for my mom who never took an eye off of me and is always watching over me night and day. It is the best feeling in the world to always have a loving home to come back to whenever your research hits a wall...

To my endearing girlfriend Sami, thank you for listening to my complaints, dealing with my stress, and always having an opinion on what needs to be done. You were always there to put a smile on my face and I don't think I could have done it without your support. I love you!

To the friends I have made through university, I know that you will be my life long friends. We will hopefully continue to have parties at Cho's, bet on when Eric will graduate, and wonder if Jeff is asleep or awake. Thanks to Jamie who had many Indian lunches with me. I am determined to beat you in squash one day. To Danny H., who is always there for a helping hand and an intermittent round of golf. I am forever grateful that I have friends like you guys.

To my two awesome dogs, Bailey the Haveanese and Boomer the Chow cross Lab. You are my best companions and your unquestionable love cannot be thanked enough. Although you won't be able to read this, I promise I'll reward you with a fist full of dog treats.

I would also like to thank all my lab-mates for their support and humorous conversations. Jian, you helped me deal with so many problems when you already had a stack of them on your own, and I am in your debt. Ed, you helped me with all the little things, and although you might not realize it, I am very grateful for all the work you did. Dave, thanks for all the help you provided and the numerous projects that we have done. I will always be able to look to you when I want to get lost in a foreign country.

Last but not least, I would like to thank my supervisor, Dr. Lesley Shannon. I realize that this section might be too colloquial to your taste, but your attention to detail has allowed me to foster many skills that I lost in my early years of university. I have learned a great deal in my graduate studies, and a lot of it is because of you.

I would also like to acknowledge the financial support, as well as the equipment and software donations, that I have received from the following organizations that have made the research possible: the Canadian Microelectronics Corporation, the Natural Sciences and Engineering Research Council, Simon Fraser University, Xilinx, and Altera.

Contents

List of Figures

List of Tables

Glossary

AND - Average Node Degree

ASIC - Application-Specific Integrated Circuits

CAD - Computer-Aided Design

CLB - Configurable Logic Block

CU - Communication Unit

EDK - Embedded Development Kit

F_{base} - Base Frequency

F_{pred} - Predicted Frequency

FF - Flip Flop

FIFO - First-In-First-Out

FPGA - Field-Programmable Gate Array

FSL - Fast Simplex Link

GRD - Global Routing Demand

INT - intercept

IP - Intellectual Property

ISE - Integrated Software Environment

ISP - Internet Service Provider

k_{GRD} - Global Routing Demand Factor

k_{LRD} - Local Routing Demand Factor

LRD - Local Routing Demand

LUT - Look-Up Table

LW	Link Width
N	Number of Nodes
ND	Node Degree
NoC	Networks-on-Chip
P2P	Point-to-Point
RPM	Relatively Placed Module
RU	Resource Usage
SL	Slope
SoC	Systems-on-Chip
TDM	Time Division Multiplexing
WOT	Weighted Order Toggling

Chapter 1

Introduction

Computer networks allow various computing elements to communicate and share resources, wherein the network provides the connectivity between the different elements. Networks provide many levels of abstraction, ranging from local area networks (LANs) with a few processing elements to the world wide web's (WWW) millions of processing elements. The same communication infrastructure can be applied to any system with multiple elements. On a much smaller scale, networks are becoming increasingly attractive for use on Systems-On-Chip (SoC) design, where multiple computing elements are implemented on the same die.

As chip densities improve, Field-Programmable Gate Arrays (FPGAs) are being increasingly used to implement complex SoCs [47]. Due to the lesser logic densities of previous generations, the complexity of SoC implementations on FPGAs has been limited to a small number of computing elements. Therefore, a shared bus such as AMBA [40], CoreConnect [45], and WISHBONE [49] could be used as the communication infrastructure between multiple master and slave nodes. While appropriate for smaller systems [1], shared busses do not scale well for more complex systems. As system complexity increases, the number of possible masters and slaves on a given bus leads to increased bus contention, slowing down data transfer and limiting bandwidth. Thus, more complex communication structures, such as networks, are becoming increasingly attractive for Network-On-Chip (NoC) implementations. NoCs attempt to provide a solution to the growing complexity of SoCs by providing a highly customizable and scalable interconnect [6][35].

1.1 Motivation

When choosing an appropriate network topology for an application, multiple design factors need to be considered. Since each application is unique, there exists a problem of picking a desirable network from a large design space. For example, the NoC topology plays a role in the throughput, bandwidth, and latency of the entire systems. For systems designed on Application-Specific Integrated Circuits (ASICs), appropriate topology choices have been investigated [2][48]. Popular choices for ASICs are the mesh and torus topologies [35][14]. These topologies map well to an ASIC's two-dimensional implementation platform, providing control over the network's electrical characteristics. More complex topologies, such as the star and hypercube topologies, are generally not used on ASICs as they lead to increased chip area and an increasingly difficult routing task as the number of nodes grows.

While these conclusions may be true for ASICs, they do not necessarily hold for FPGAs. For ASICs, the structure and interconnect is completely defined by the designers and only the wires pertaining to the NoC are used and routed. In contrast, modern FPGAs are over-provisioned with routing; that is, FPGA architects provide significantly more routing than is needed for the "common case" to ensure a high fitting rate by the Computer Aided Design (CAD) tools for customer designs. Preliminary work by Shannon et al. showed that due to this over-provisioning, more complex topologies such as the star and hypercube are possible [51]. In fact, it suggested that in some cases these types of networks may be preferable to a mesh, since they have better network latency and bandwidth characteristics, yet can still be implemented easily on a modern FPGA. This result implies that designers have the increased freedom to select more complex topologies when implementing NoCs on FPGAs as opposed to ASICs. However, to leverage these findings, a more concrete understanding of the performance of various network topologies implemented on the fixed prefabricated routing of an FPGA is required.

1.2 Objective

While there has been research investigating the use of FPGAs for NoCs [9], little has been done to characterize the routability and performance of NoC architectures on FPGAs. Given a specific platform and network characteristics, the objective of this work is to quantify and develop an analytical model describing the routability and performance characteristics for both homogeneous and heterogeneous NoCs on a variety of FPGA devices. Various network topologies, using both homogeneous and heterogeneous nodes are investigated to quantify how specific NoC design parameters affect routability and performance in terms of maximum operating frequency.

1.3 Contributions

Our research provides a thorough analysis of the performance of FPGA-based NoCs. The contributions of this work is as follows:

1. An investigation into the effects of different NoC parameters (Resource Usage, Number of Nodes, Average Node Degree, Link Width, Heterogeneity) on system performance.
2. Development of an analytical model describing the maximum operating frequency of an NoC that encapsulates the effects of the CAD tools and network topology for both Xilinx and Altera FPGAs.
3. A preliminary investigation into using relatively placed modules (RPMs) to demonstrate if improvements in performance can be attained using this form of guided placement as opposed to purely "automatic" placement.

These contributions are important for two reasons. Firstly, it quantifies the effects of specific network parameters on performance, and thereby the suitability of network topologies for implementation on an FPGA. This is an important first step to understanding the flexibility and limitations of mapping application-specific network topologies to an FPGA's prefabricated routing interconnect using its commercial tool flow. Secondly, it provides guidance to a designer during early design space exploration, when a suitable network topology is being chosen.

1.4 Thesis Organization

This thesis is divided into 7 chapters. Chapter 2 discusses associated terminology and previous work related to our investigation of the implementation of NoCs on FPGAs. Chapter 3 describes the experimental methodology used in the investigation. Chapter 4 introduces the analytical framework and presents the general NoC performance trends. Chapter 5 describes the final derived analytical model for Altera and Xilinx FPGAs. Chapter 6 verifies our analytical model using a new set of benchmark circuits. Finally, Chapter 7 summarizes the conclusions of this work and possible areas for future work.

Chapter 2

Background

In order to fully characterize how varied NoC parameters affect system performance, diverse topologies are used to encapsulate the different network parameters. This chapter introduces the various network topologies used in our experiments and their properties, along with a thorough discussion of previous work done to investigate NoCs implemented on FPGAs.

2.1 Terminology

A network *topology* describes the connectivity between network nodes in an NoC. As shown in Figure 2.1, the topologies that are analyzed in our research include the (a)ring, (b)mesh, (c)star, (d)fully-connected, (e)torus, and (f)hypercube topologies, which are commonly used in many applications. In our experiments, the number of columns for the mesh topology is fixed at four. The size of the mesh topology is varied by increasing or decreasing the number of rows. Using these topologies, the experiments are further expanded by including both homogeneous or heterogeneous NoCs. A *homogeneous NoC* consists of identical network nodes, whereas a *heterogeneous NoC* consists of a set of different network nodes. A number of parameters are used to characterize a topology as shown in Table 2.1. For a given topology, the *diameter* of the topology is the maximum distance between any two nodes using existing links. The *link complexity* is the total number of links a topology requires. The *node degree*

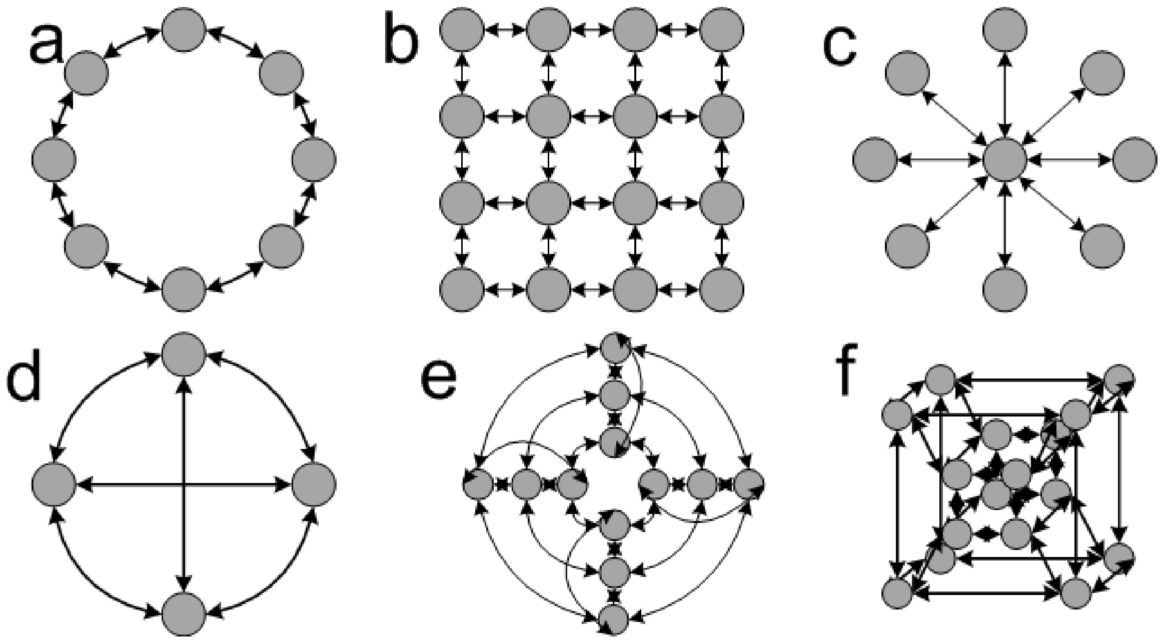

Figure 2.1: Network topologies

Table 2.1: Network topology characteristics

Topology	Diameter	Link Complexity	Node Degree	Regular
Ring	$n/2$	n	2	yes
Mesh	$2(n^{1/2} - 1)$	$2(n - n^{1/2})$	2,3,4	no
Star	2	$n - 1$	1,$n - 1$	no
Fully-Connected	1	$(n(n-1))/2$	$n - 1$	yes
Torus	$n/2$	n	4	yes
Hypercube	$\log_2 n$	$(n \log_2 n)/2$	$log_2 n$	yes

is the number of links from a node to its connected neighbours. Furthermore, a *regular* topology is defined as a topology where all nodes share the same node degree. Finally, the *average node degree* (not shown in Table 2.1) is the average number of links for all nodes. Also not shown is the *link width*, which is the number of bits in each link between network nodes.

In addition to these common topologies, the experiments also include random topologies, which are used to model application-specific topologies. Random topologies are used to model application-specific NoCs where connections between nodes are defined by application needs rather than using a topology in Figure 2.1 that could lead to unused connections. How random topologies are generated is discussed in Section 3.1.1.

2.2 Related Work

Research involving NoCs on FPGAs predominantly focuses on the design and implementation of NoCs. Network architectures investigate different switching elements to provide the best throughput and bandwidth. Changes to the FPGA architecture have also been suggested to better support NoCs as they become more popular. Furthermore, customized CAD tool flows have been developed to simplify the design process of NoCs. Our research focuses on characterizing NoC performance on existing FPGAs using our own customized tool flow. Using our results we develop an analytical model that describes the system performance as a function of various network parameters. This section describes previous research done for NoCs on FPGAs, along with some examples of analytical models for FPGAs.

2.2.1 Network Architecture

When designing FPGA-based NoCs, there are a number of design characteristics that are considered to optimize NoC performance. Numerous studies investigate different switch and router architectures that are targeted for use on FPGAs. Kapre investigates a high-performance, packet-switched, on-chip network [30]. Packet switching groups all transmitted data into suitable blocks to be routed over a shared network. In contrast Kapre's work, a parallel study by Mehta investigates highly-scalable, time-multiplexed, FPGA networks [44]. A time-multiplexed network relies on transmitting blocks of data from multiple nodes over the same network, sharing the resources over blocks of time. Both studies analyze the performance of their respective switch architecture for different network topologies and provide a measure of performance for both types of network interface on the Xilinx XCV4000. Kapre et al. then performs a direct comparison of the two switch architectures [31]. The study found that packet-switched networks typically outperform time-multiplexed systems for NoCs with under 100 nodes. Furthermore butterfly fat-trees are the best performing topology for both architectures.

Sethuraman et al. proposes a lightweight parallel router, which can support five simultaneous routing requests, with minimal overhead, implemented on a Xilinx XC2VP30 [54]. The router utilizes optimizations in XY routing and decoding logic that maximize the performance to area ratio. With minimal packet overhead, the router is implemented using 3x3 mesh networks and characterized for power and performance parameters. Bartic et al. investigates a topology adaptable communication network design [5]. A versatile network platform is introduced with the IP interfaces, ISP, and an operating system (OS) managing all network resources. This architecture allows the possibility of dynamically changing the packet routing for each IP, in every router, allowing a better balance in network traffic. However, the proposed architecture experiences a quadratic increase in resource utilization as the number of input signals increases. Zeferino et al. presents a router soft core for NoCs called RASoC [59]. The router architecture is used in the building of NoCs for embedded systems. The model relies on a parameterized VHDL module, which allows reuse of RASoC in the synthesis of NoCs with different sizes. However, significant overhead in resource utilization is reported. The overhead is attributed to the complicated switch design and routing algorithm implementation.

Bertozzi et al. designs a complete NoC infrastructure called Xpipes [7]. The system consists of a library of switches, network interfaces, and links, that are designed to be tuned to specific heterogeneous architectures. Links can be pipelined to a flexible number of stages to separate a link's latency from its throughput. Furthermore, a tool called XpipesCompiler is developed that automatically instantiates a customized NoC using its library of network components based on a given set of design parameters. Hilton et al. presents a flexible circuit-switched NoC for FPGA-based systems called PNoC [24]. In contrast to packet-switching, which uses a shared network resource, circuit-switching relies on establishing a connection between nodes before transmission can occur. While PNoC demonstrates a 23x speedup when compared with a shared bus implementation, a direct comparison between PNoC and a packet-switched network is not performed as a suitable packet-switched architecture was not developed yet.

These switch architectures, along with different bus architectures and proposed FPGA architectural changes, are compared by Mak et al. [43]. The investigation acknowledges that it is difficult to compare the different NoCs, as their respective implementations are often time applied to specific applications. The paper identifies that due to the unique requirements of different applications, there exists a problem of searching for optimal communication architectures from a huge design space since choices are often performed ad-hoc. Our work takes a step in this direction providing a method of predicting the performance of NoCs to enable easier design space exploration.

From all these investigations, NoCs on FPGAs are advocated as a promising solution to on-chip communication especially in Multi-Processor Systems-on-Chip (MPSoC) design. These investigations focus on the philosophy of "routing packets, not wires" [14][53], thus the network interface itself is the focus of much of the research. Selecting appropriate switch architectures, when used in conjunction with our work, can improve the means of selecting an optimized NoC model.

2.2.2 Architectural Changes

Since the routing and resources for FPGAs are fixed, possible architectural changes to FPGAs have also been investigated to improve the performance of NoCs. In terms of architectural changes, research has predominantly focused on changing the wiring types in FPGAs, and adding network switches or routers as embedded blocks. All studies aim to minimize area overhead, while maximizing data throughput and bandwidth for systems utilizing an NoC as its communication infrastructure.

Francis et al. shows that fine-grain, time-division-multiplexed wiring outperforms conventional wiring for networks on FPGAs [16][15]. Time-division multiplexed (TDM) wiring consists of FPGA wiring links that are shared amongst different IP blocks. This allows for increased capacity on individual wiring links while decreasing the silicon area. TDM wiring is scheduled serially within a discrete time slot, with the interconnect pipelined at a higher rate than the design to increase data throughput. The investigation shows that using TDM wiring can reduce the amount of FPGA configurable wiring by up to 83% and reduce the complexity of switch boxes, thus leading to an overall reduction in silicon area.

Goossens et al. illustrates a dedicated NoC interconnect fabric hardwired in an FPGAs [19]. By proposing a hardwired NoC, some area of the FPGA is lost to a fixed function. However, Goossens proposes that the loss of flexibility is outweighed by the reduced implementation costs and greater flexibility in dynamic partial reconfiguration. The investigation uses hardwired NoCs as functional interconnect between IP blocks and configuration interconnect that are used to transport data. The proposed hardwired NoC has a 10% overhead for IP sizes with approximately 1400 Look-Up Tables (LUTs).

Gindin et al. presents an NoC architecture based on a proposed routing scheme called Weighted Ordered Toggle (WOT) [18]. WOT utilizes simple, small-area, on-chip routers with low memory demands. The architecture consists of an island-style FPGA with configurable network interfaces at each "island," on the assumption that most applications implemented on this architecture will use a network style interface. Each network interface uses the WOT routing algorithm, which is a packet-switched network that toggles the flow of packets in the horizontal and vertical directions. Ahmadinia et al. describes a dynamic network-on-chip (DyNoC) consisting of a coarse-grain programmable fabric interlinked with circuit-switched busses called Reconfigurable Multiplexed Bus (RMBoC) and connected to each configurable region via on-chip switches [3]. A new routing algorithm is proposed such that the network-on-chip architecture can be realized with run-time configuration. Due to the complexity of the system, the on-chip switches consume a large area of the design. For a Xilinx Virtex-II 1000 device, 21% and 46% of logic resources are devoted to the on-chip switches of word-length 32-bit and 64-bit respectively. Similarly, Jovanovic et al. presents an island style architecture (CuNoC) with communication units (CU) placed at each configurable region [29]. Each CU consisted of a simple packet forwarding switch based on the priority-to-the-right rule. CuNoC is shown to have good performance and much less overhead when compared to DyNoC.

For all investigations, architectural changes show improved performance for applications that require NoCs. However, due to increased overhead, these changes may not necessarily be beneficial in other types of systems. Whereas these investigations propose possible changes to FPGA architecture to support NoCs, our objective is to understand how the existing interconnect fabric constrains NoC performance on commercial FPGAs. Thus the focus is on existing modern architectures and not on architectural changes.

2.2.3 CAD Tool Flows

In order to fully exploit FPGA resources, complex CAD algorithms are used to place and route NoCs on FPGAs. Research has been done to create automated design flows that target NoCs on FPGAs. Bertozzi et al. presents a complete synthesis flow, called NetChip, for customized NoC architectures [8]. The flow partitions the development work into topology mapping, selection, and generation. The entire flow implements a reusable and scalable network component's library called timespipe that is design-time tunable and customizable to achieve arbitrary topologies. Several case studies are presented showing the use of NetChip in generating NoCs.

Kumar et al. develops an automated design flow to instantiate Multi-Processor Systems on Chip (MPSoC), with an NoC communication scheme [34]. The NoC specifications are done on a high level of abstraction, relieving the designer of low level design. The flow is used to generate a set of sample designs verified on a Xilinx Virtex II 6000. In addition to the automatic generation, a run-time flow is presented that allows easy debugging and reconfiguration of the system via a host. Sethuraman et al. presents an algorithm which optimally maps custom routers called optiMap [55]. Each router consists of a multi-local port router capable of handling multiple logic cores in parallel. For a given NoC, optiMap finds the optimal number of routers, configuration of each router, optimal mesh topology and the final mapping of the NoC. In comparison to a single-local port version of the same router, the investigation observed an average of 36% area savings based on the Virtex 2 Pro 30.

A framework based on the Xilinx Embedded Development Kit (EDK) is also presented by Lukovix et al. [41]. The described framework represents a fully integrated design flow for fast generation of NoC-based MPSoCs on Xilinx FPGAs. In our research, we employ a similar tool flow to automatically generate a wide variety of NoC systems required for this investigation. However, our tool flow is capable of generating NoCs based on both Xilinx EDK and Altera Quartus CAD flows. A similar tool flow is also developed by Saldana et al [51]. The tool flow took an existing system generated for Xilinx EDK and changed the system files to describe the desired NoC system. The newly generated tool flow is developed in C and works independently of any existing systems. All file generation is hard coded into the tool flow to avoid any extraneous inputs. The details on the tool flow design are given in Section 3.2.

2.2.4 Previous Analytical Model Work

In the past, research has primarily evaluated FPGA architectures and applications using empirical analysis to find the best parameters to optimize performance. More recently, there has been a growing trend to express FPGA performance using analytical models to enable better design exploration when designing for FPGAs. Ahman et al. presents an analytical model that predicts interconnect requirements in FPGAs based on the number of look-up-tables (LUTs) in the FPGA [50]. This analytical model for two-dimensional FPGAs is calibrated using fully routed benchmark circuits and extended to three-dimensional

FPGAs. The investigation found that for FPGAs with more than 20,000 4-input LUTs, three-dimensional FPGAs can potentially reduce channel width, interconnect delay and power dissipation by up to 50%.

Lam et al. describes an analytical model that relates logic parameters to the area efficiency of an FPGA based on Rent's Rule [36]. The model relates the LUT and cluster size, and the number of inputs per cluster to the amount of logic that can be packed into each cluster. Due to the simplicity of the analytical model, the analysis can provide a tool for FPGA designers to better understand and guide the development of future FPGA architectures. An analytical model that relates FPGA architectural parameters with the average prerouting wirelength is presented by Smith et al. [57]. This model encompasses both homogeneous and heterogeneous FPGA architectures. For homogeneous FPGAs, the wirelength is related to the LUT and cluster size, and the number of inputs per cluster. For heterogeneous FPGAs, in addition to the parameters investigated for homogeneous FPGAs, the position of embedded blocks as well as the number of pins to each block is also considered. Much like the evaluations done in these works, our research focuses on developing an analytical model that enables designers to perform design space exploration for FPGA-based NoCs.

2.2.5 Analytical Models for NoCs

In this work, an analytical model describing the performance of an NoC is presented. Although a general analytical model describing the performance of a wide range of NoCs have not been investigated, several studies have investigated the performance of NoCs for specific applications. Lee et al. [37] describes an analytical model to predict the area, performance, and energy consumption of an MPEG-2 encoder application implemented on a Virtex2 3000 FPGA. The study analyzes the difference between using a network, point-to-point, or a shared bus for communication between nodes. It found that: 1) the performance of the NoC design is very close to the point-to-point (P2P) implementation, 2) the NoC implementation scales better in area than the P2P and bus-based implementations, and

3) the NoC implementation has lower energy consumption than the P2P and bus-based implementations.

Freitas et al. performs a study comparing the performance of a MPSoC system using bus-based communication to a mesh network [17]. The work utilizes MicroBlaze soft processors on a Xilinx Virtex 2 Pro 20 with 4-node and 16-node systems. The work concludes that MPSoCs benefits in performance when using NoCs versus bus-based systems. Similarly, Saldana et al. investigates the routability of multiprocessor network topologies on FPGAs [51]. Ring, star, mesh, hypercube, and fully connected topologies from 8 to 32 nodes are utilized to characterize the performance and area requirements on a Xilinx Virtex 4LX200. It is determined that all topologies except for fully connected performed well up to 32-nodes

The work in this thesis expands on the work done by Shannon et al. [51]. Preliminary results from this work is also presented in previous publications. Lee et al. presents the effects of node size, heterogeneity, and network size on NoC performance [38]. It is shown that the number of nodes has a greater affect on performance than node size and heterogeneity. Lee et al. further investigates NoC performance by developing an analytical model to predict the performance of NoCs due to local and global routing demand on Xilinx FPGAs [39]. The work presented in this thesis expands on these results [38] [39], utilizing both Xilinx and Altera FPGAs. Bandwidth, resource utilization, and the use of relatively placed modules are also explored in this thesis.

Chapter 3

Experiment Methodology

The objective of this research is to investigate how an NoC topology and its associated characteristics effects performance. Specifically, the focus is on the routing resources and how the NoC maps to an FPGA's fixed interconnect based on different NoC parameters. To do this, we utilize a vast exploration space, which includes homogeneous and heterogeneous topologies for a wide range of FPGA devices. The following chapter describes the NoC design used in our experiments, and the custom tool flow developed to automatically generate these NoCs.

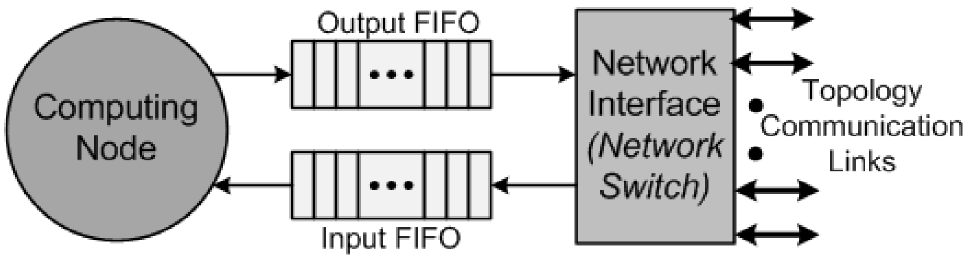
Output FIFO
Computing Node
Network Interface (Network Switch)
Topology Communication Links
Input FIFO

Figure 3.1: Network node

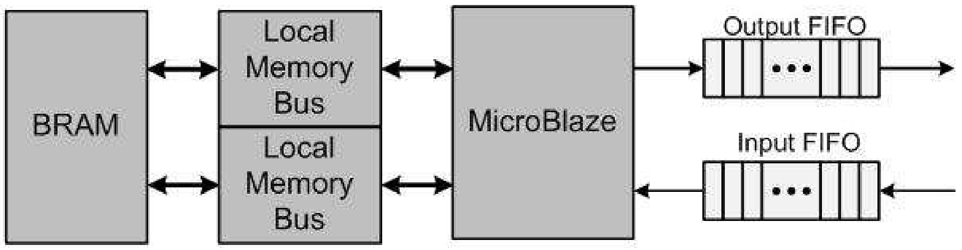

Figure 3.2: MicroBlaze computing node

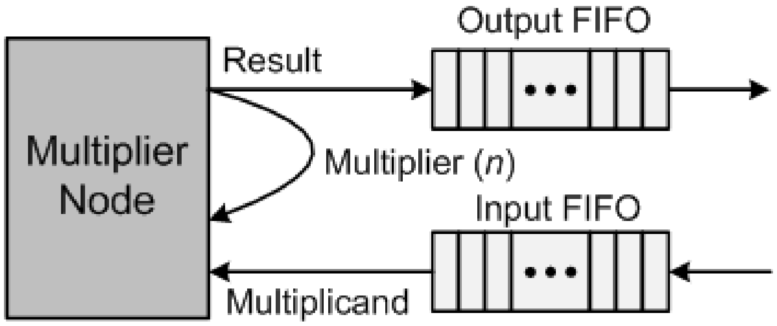

Figure 3.3: Multiplier computing node

3.1 Network Nodes

Section 2.1 illustrates the different network topologies, including common and application-specific topologies that are used in our experiments. Each network node, as shown in Figure 3.1, consists of a computing node, two FIFOs, and a network interface linked to the network interconnect using topology communication links. Each computing node communicates with the network interface through two, synchronous, 16-word-deep FIFOs. For Xilinx FPGAs, these FIFOs are implemented using Fast Simplex Links (FSLs) mapped to LUT-RAMs. For Altera FPGAs, these FIFOs are implemented using equivalent M-LABs, which are only supported by the Stratix III and Stratix IV families.

3.1.1 Computing Node

Two types of computing nodes are used; a MicroBlaze soft processor and a custom computing node. The MicroBlaze, as shown in Figure 3.2 is used to compare our newly updated tool

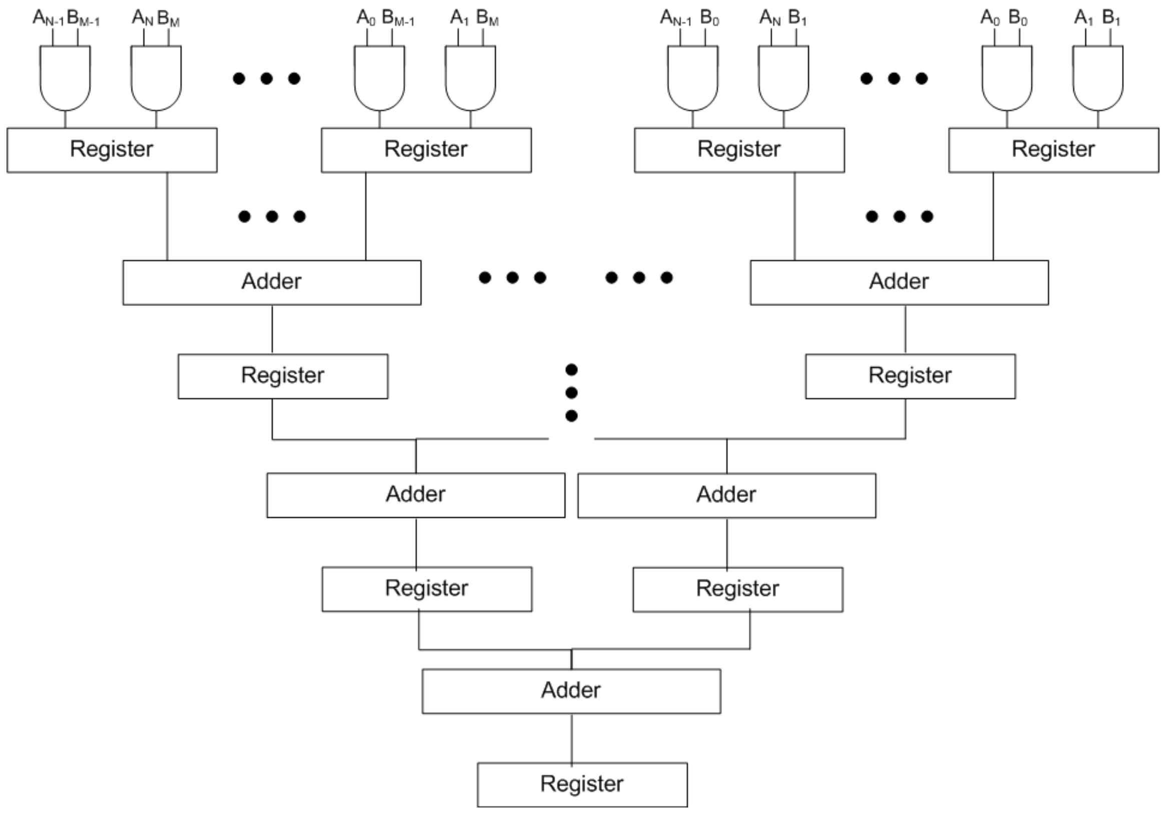

Figure 3.4: Multiplier architecture

Table 3.1: Homogeneous multiplier node types

Link Width	**uBlaze**	**MultHalf**		**MultBase**		**MultDouble**	
	LUTs	*Mult*	*LUTs*	*Mult*	*LUTs*	*Mult*	*LUTs*
48	629	2	347	4	658	6	1312
40		2	396	4	703	6	1487
32		4	371	6	694	8	1429
24		6	328	8	652	12	1393
16		8	311	12	614	16	1374

flow with the previous work [51]. Since our investigation focuses on the topology and network links (*topology communication links*), and not the computing nodes, the computing node has to meet a set of requirements that ensures that it does not contaminate the data obtained to characterize the links. Specifically, the computing node needs to: 1) not be the critical path in the design, 2) use only CLBs and no embedded blocks to ensure portability between Altera and Xilinx devices, and 3) ideally have all combinatorial operations registered in the same CLB to improve timing. The multiplier node in Figure 3.3 meets all these requirements. The multiplier consists of 2-bit partial-products pipelined at each stage as shown in Figure 3.4. Although actual network implementations would contain more complex computing nodes, such as a MicroBlaze soft processor, it is not helpful for this study. If a MicroBlaze is used as the compute node, then in small NoC systems, the critical path would be in the node itself, and not in the topology communication links. Furthermore, the link width cannot be changed for a MicroBlaze node and it is not portable to non-Xilinx FPGAs. In contrast, the multiplier node's size as well as link width can be adjusted, allowing us to experiment with different link widths, and topologies with different network node sizes. Therefore we only use the MicroBlaze in our new analysis to compare current state of the art CAD tools with previous work [51].

As shown in Figure 3.3, the *multiplicand* of the multiplier node is equal to the link-width, and the *multiplier* is equal to the lower n bits of the result. We consider both homogeneous and heterogeneous NoC types; in a homogenous network, all nodes are of the same size, while in a heterogeneous network, different node sizes exist. As the *multiplicand* always remains fixed to the link-width, the *multiplier* (n) is varied to scale the resource usage. For a given

Table 3.2: Heterogeneous multiplier node types

Link	**MultSmall**	**MultFull**	**MultLarge**
Width	*Range of Mults*	*Range of Mults*	*Range of Mults*
48	2,4	2,4,6,8	4,6,8
40	2,4	2,4,6,8	4,6,8
32	2,4,6	2,4,6,8,10	6,8,10
24	4,6,8	4,6,8,10,12,14	10,12,14
16	6,8,10	6,8,10,12,14,16	12,14,16

heterogeneous or homogeneous system, the link width remains fixed for all network nodes. Table 3.1 lists the multiplier sizes used in our homogeneous experiments. *Mult* is defined as the width (*n-bits*) of the multiplier, and *Link Width* is equal to the multiplicand width. We used five different link widths, with three different multiplier widths for each link-width. The baseline multiplier *MultBase* is chosen to have approximately the same LUT usage as a MicroBlaze on a Virtex 5. *MultHalf* is approximately half the size, and *MultDouble* approximately twice the size. As the link width changes, the multiplier width also needs to be adjusted to maintain approximately constant resource usage for the computing node. The percent variation in LUT usage for each node type is 7.12% with a standard deviation of 3.2%.

To generate heterogeneous NoCs, we kept the link width (multiplicand) fixed for the design and varied the network node size by scaling the multiplier bit-width. We use three types of heterogeneous NoCs (*MultSmall, MultFull, MultLarge*) generated using a range of multiplier node sizes defined by a minimum and maximum multiplier bit-width. The size of each multiplier node in a heterogeneous topology is chosen at random and uniformly distributed across the range of multiplier widths defined by the heterogeneous NoC type and fixed link width. Table 3.2 lists the range of sizes used in our heterogeneous experiments for varied link widths.

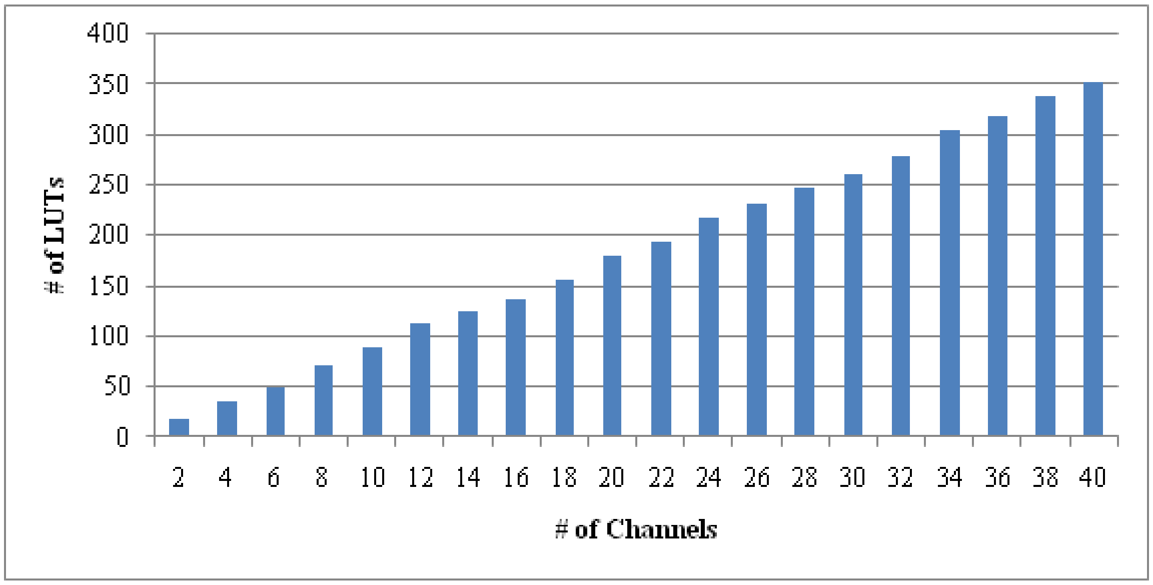

Figure 3.5: Network interface resource utilization

3.1.2 Network Interface

Since we are interested in the performance achieved by the CAD flow's ability to leverage the over-provisioned routing resources to implement the links that define the NoC topology, and not the NoC's performance in terms of bandwidth and throughput, we use a lightweight network switch. In order to isolate the performance of the network interface and subsequently the network node from the interconnect, the output from the network interface to the link is registered. The network switch is a lightweight packet switch that broadcasts an address control packet to all its linked neighbours. A receiving switch only reads the control packet and subsequent packets if the address matches its own and otherwise ignores it. These switches are only capable of sending to and receiving from their directly connected neighbours and are not capable of multi-hop communication. While extremely simple, this switch is sufficient for the purpose of our studies, which focuses on the performance of various topologies on an FPGA's fixed interconnect and not on the switch architecture or packet latency and throughput.

The size of each switch is defined by its number of ports. The NoC topology in turn

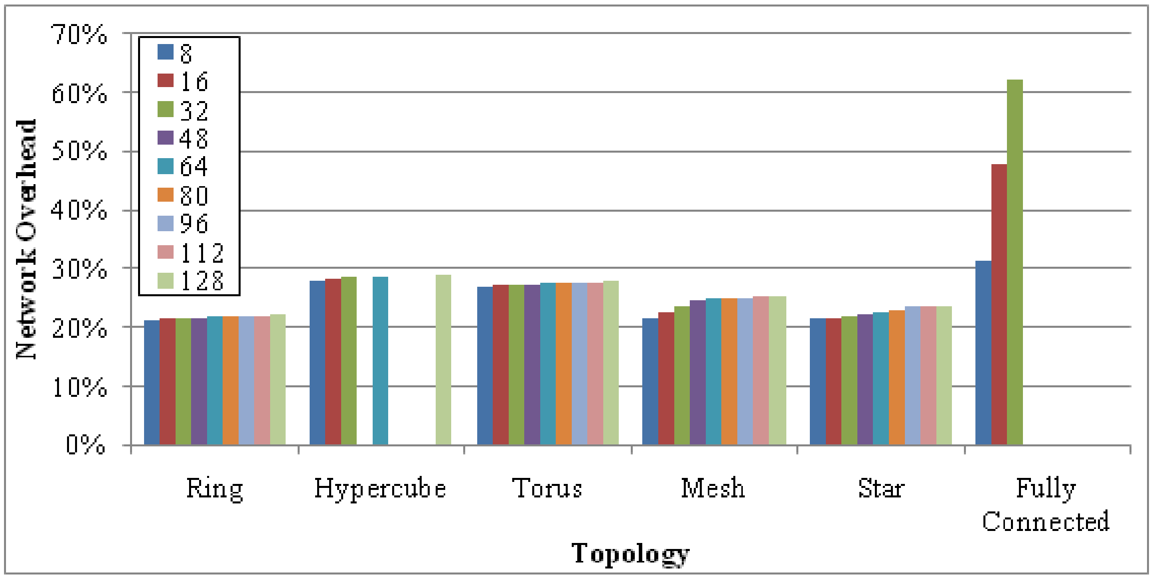

Figure 3.6: Network overhead on Xilinx Virtex 5 LX330

defines the number of ports required by each network node's switch. For example, a torus is a regular topology with all network nodes having a node degree of four. This means that each network interface has four ports or four channels. Each channel comprises two unidirectional links to form a bidirectional channel. In contrast, a mesh topology has network switches with node degrees of two, three and four. Therefore the system will require three different network switches with two, three, and four ports. All network nodes on the perimeter of the mesh require a network switch with three ports except the corner network nodes, which require a network switch with two ports. Finally, the interior network nodes require a network switch with four ports. Recall that in Figure 2.1 each topology is illustrated, and the network switches required by each network node can be determined by the number of links connected to a given node. As the number of links to the network switch grows, the resource utilization also increases linearly. Figure 3.5 shows the resource utilization (Number of LUTs) of different network interface sizes. This ensures that the total resource utilization of the network node (i.e. including the computing node, FIFOs, and network interface) increases linearly.

The network switch overhead is defined as the resource usage utilized by all network switches in the NoC topology as a percentage of total resources. In order to determine the network switch overhead of each topology, we measure the percentage of total resources occupied by the network connectivity. Figure 3.6 shows the network overhead for homogeneous ring, hypercube, torus, mesh, star, and fully connected topologies with MultBase nodes and 32-bit link widths.

As seen in Figure 3.6, for all topologies other than the fully connected, mesh, and star topologies, the network connectivity occupies a constant percentage of total resources as the number of nodes increase. This trend is expected since the resource usage of each individual network switch remains constant for the ring, hypercube, and torus topology. Thus when the NoC increases in the number of nodes, the total resource usage of all network switches should increase at the same rate as the total resources occupied by the entire NoC topology.

For small mesh topologies, the number of network switches with two or three ports is greater than the network switches with four ports. However, as a mesh gets bigger there are more four port network switches than two or three port network switches. This is shown by the increase in network overhead from 8 to 48 nodes due to an increase in four port nodes in relation to two and three port nodes. The network overhead eventually levels off for larger mesh topologies as the majority of switches utilize four ports. Furthermore, the slight increase in network overhead for the star topology is due to the increase in ports for the central node's network switch; and the large increase in network overhead for the fully connected topology is due to the increase in ports for all network nodes in the system.

3.2 Custom Tool Flow

In our experiments, we use approximately 3600 benchmark circuits that are generated using an automated benchmark circuit generator. The generator supports six of the common network topologies, previously shown in Figure 2.1, and application-specific topologies modeled using random topologies, with several different node sizes. The use of such a generator allows us to run many more experiments than would be possible using "real" benchmark circuits; this, in turn, allows us to isolate the impact of each NoC parameter on the overall

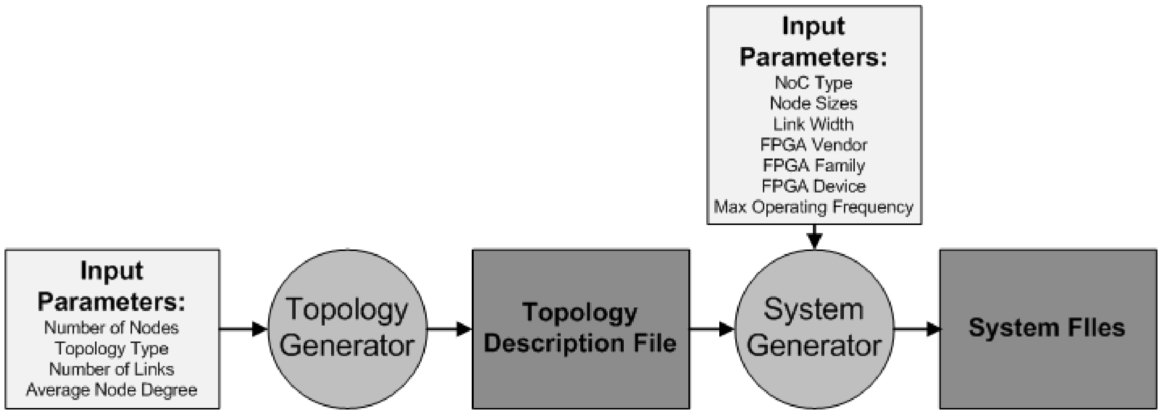

Figure 3.7: Custom tool flow

performance of the system.

Figure 3.7 shows the tool flow used to generate the benchmark circuits. The code for the *topology generator* is presented in Appendix A, and the code for the *system generator* is shown in Appendix B. The topology generator outputs a topology description file that defines the connectivity between each network node based on a given set of input parameters. These inputs define the NoC topology characteristics and are listed below:

- Number of Nodes - the number of nodes in a given system
- Topology Type - Torus, Hypercube, Ring, Mesh, Star, Fully Connected, Application-Specific (modelled using random topologies)
- Average Node Degree (optional, if topology is random) - the average number of links to all nodes

Common topologies have a predefined connectivity pattern, however application-specific topologies are not predefined. Therefore for application-specific topologies, given a fixed number of nodes, links are randomly generated between nodes until the average node degree is met and all nodes have at least one link. The topology description file illustrates the number of nodes, and the switch and node properties. The number of switches is determined by the topology type. If the topology is regular, than there is only one switch type. If the topology is not regular, than the number of switch types depend on the connectivity of all nodes. For example, as previously described, a mesh topology requires three switch types. Given the topology description file, the system generator produces the necessary files used by either the Altera and Xilinx CAD tool suites depending on the input parameters shown below:

- NoC Type - Homogeneous/Heterogeneous
- Node Sizes - Multiplier width or range
- Link Width - Width of point-to-point links between nodes (FIFOs)
- FPGA Vendor - Xilinx/Altera
- FPGA Family - Virtex 2 Pro, Virtex 4, Virtex 5, Virtex 6, Stratix III, Stratix IV
- FPGA Device - Specific device for a given FPGA family
- Maximum Operating Frequency - Target operating frequency used by the CAD tools

The output files include verilog files describing the NoC and run-time parameters needed by the tool suite. Using the system and topology generator, a wide range of NoCs can be generated. The topology generator can produce topologies of any size, and the system generator can create all necessary files required by the CAD tool suites to implement a wide variety of NoC designs. This provides the necessary framework to generate the infrastructure required by the investigation.

3.3 Testing Methodology

Our experiments aim to quantify and characterize the routability and performance of NoCs on a variety of FPGA architectures. Using our benchmark circuits, we generate approximately 3600 circuits to encapsulate the effects of different network parameters. These

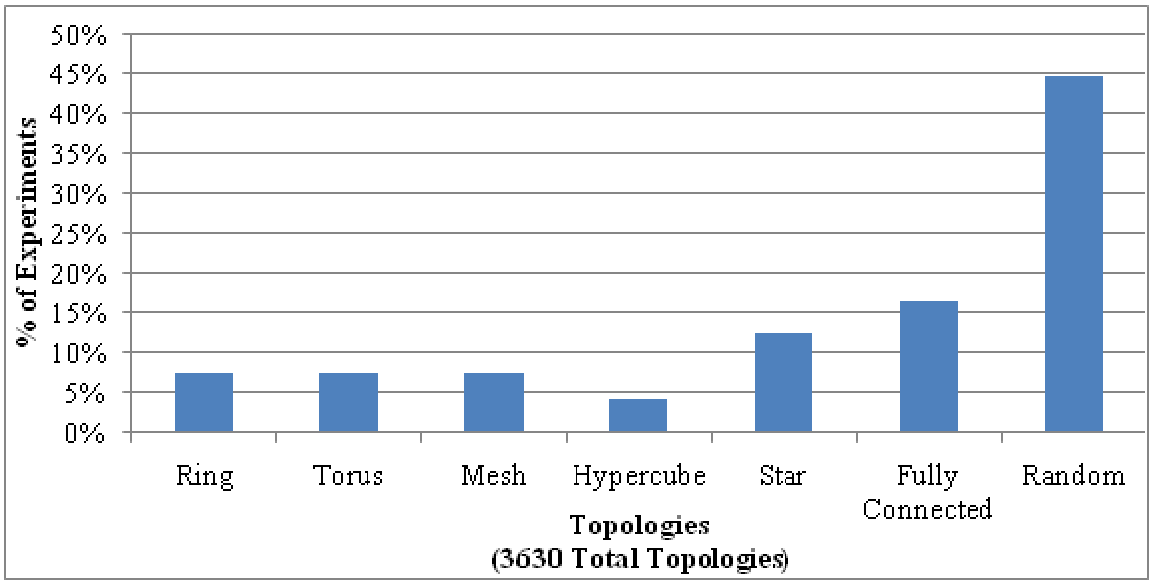

Figure 3.8: Distribution of experiments for each topology

circuits included ring, torus, mesh, hypercube, star, fully connected, and random topologies of varied link width and number of nodes. The distribution of experiments based on topology type in shown in Figure 3.8.

Since random topologies include systems with varying average node degree, these topologies constitute a greater percentage of total experiments. For the ring, torus, and mesh topologies, circuits are generated from 8 to 128 nodes at 8 node intervals. For the hypercube topology, there can only be 2^n nodes, thus there are fewer hypercube experiments. The fully connected and star topology is investigated in greater detail to pinpoint where systems fail to route. Thus circuits for these two topologies are generated from 8 to 128 nodes at 4 node intervals.

While having a wide range of NoCs is essential to characterizing NoC performance, we also synthesize these experiments onto multiple Xilinx and Altera FPGA device families. This allows us to demonstrate if the results are device or family specific. The devices that we use in our experiments are listed in Table 3.3. Column 1 lists the processing technologies of each device, and Columns 2 and 3 show the family and devices for Xilinx FPGAs and

Table 3.3: FPGA family and devices

Processing Technology	Xilinx		Altera	
	Family	**Devices**	**Family**	**Devices**
130nm	Virtex 2 Pro [25]	XC2VP100-6		
90nm	Virtex 4 [27]	XCV4LX200-11 XCV4LX160-11 XCV4LX100-11	Stratix II [11] (*Not Used*)	
60nm	Virtex 5 [26]	XCV5LX330-2 XCV5LX220-2 XCV5LX155-2	Stratix III [12]	EP3SL340-4 EP3SL200-4 EP3SL150-4
40nm	Virtex 6 [28]	XCV6LX760-2 XCV6LX365T-2 XCV6LX240T-2	Stratix IV [13]	EP4SE530-I4 EP4SE360-I4 EP4SE230-I4

Columns 4 and 5 show the family and devices for Altera FPGAs. While Xilinx and Altera FPGAs share the same processing technologies, their architectures are different, thus we expect some differences in performance. Since previous work also uses all Virtex 2 Pro FPGAs listed in Table 3.3, and the Virtex 4LV100 FPGA [51], we use the same FPGAs when performing our comparison with previous work.

All our benchmark circuits are synthesized for all the FPGA devices listed in Table 3.3. We found that while our model is the same for Xilinx and Altera FPGAs, the coefficients of our framework are unique between Xilinx and Altera FPGAs. Therefore, for our analytical model describing Xilinx FPGAs, our model coefficients are tuned by first running training experiments using Xilinx EDK 10.1.02 with the Virtex 2 Pro, Virtex 4, and Virtex 5 FPGAs as listed in Table 3.3. In order to evaluate the accuracy of our model, we use the Virtex 4, Virtex 5, and Virtex 6 FPGAs listed in Table 3.3, with EDK 11.2. For our Altera model, the training experiments use Altera Stratix III FPGAs, and the verification experiments use Stratix III and Stratix IV FPGAs listed in Table 3.3. Both sets of experiments are synthesized using Quartus 9.1. Only two Stratix devices are used since these devices are the only ones that are capable of mapping the network node's FIFOs to M-LABs.

In order to obtain accurate results, each design is synthesized multiple times. Initially for our Xilinx model, we ran experiments using the Xilinx Xplorer utility, which synthesizes designs using known place and route parameters to provide the best results. However, we found that the utility resulted in extremely long run times (on the order of 5-7 days) for each experiment, since the utility always runs to completion even when the maximum operating frequency converges before that point. Therefore, we only use the utility to form a baseline of comparison for the remaining experiments.

Rather than using the Xplorer utility, we are able to approximate Xplorer's process by synthesizing designs multiple times using different seeds, with the maximum operating frequency averaged over each run. The number of iterations is determined by repeating iterations until at least five iterations are run and the change in the average result over all runs is less than 5%. When the results do not converge (which occurs in less than 8% of our experiments), we set an upper bound on the experiments to ten iterations. This method has an average variation of 2.1% to the Xilinx Xplorer utility. A design is deemed unroutable if the design can not route for at least eight out of the ten iterations. This method results in shorter run times, since using our method limits the number of experiments once performance converges.

Chapter 4

Deriving an Analytical Framework for NoCs

The objective of this work is to create an analytical model that describes the maximum operating frequency of an NoC. Experimental results are used to derive the coefficients for these equations for a selected device. Our analysis focuses on how specific NoC parameters affect routability and performance of an NoC utilizing both homogeneous and heterogeneous topologies. The key parameters that are investigated include heterogeneity, resource usage, the number of nodes, average node degree, and link width of the topology.

Our overall approach is as follows. We arbitrarily chose an 8-node ring topology with a 32-bit link-width as a baseline architecture, and denote the maximum frequency of this baseline architecture implemented on a given FPGA as F_{base}. Therefore, the predicted frequency F_{pred} is quantified as a percentage performance of the baseline performance. For different NoC architectures, we then scale F_{base} using two factors. The framework is shown below:

$$F_{pred} = k_{GRD} \times k_{LRD} \times F_{base}, \tag{4.1}$$

where the terms k_{LRD} and k_{GRD} are functions of the NoC topology, the link width, and the number of nodes in the NoC. The first scaling factor in the above equation, k_{LRD}, models the impact of *local routing demand*. Local routing demand is defined by the routing requirements of a single network node. Network nodes that have a high number of links will tend to have more congestion around their network interfaces. The CAD tools resolve this congestion by either using non-direct routes, or by spreading out the logic related to those nodes with high degree. The impact is a decrease in the maximum frequency of the network; the magnitude of this decrease is encapsulated in k_{LRD}.

The second scaling factor in the above equation, k_{GRD}, models the impact of *global routing demand*. Global routing demand is characterized by the routing requirements of the entire NoC. Topologies that have a large number of links will require more wires. As the number of links increases, the difficulty in routing these connections increases, again leading to a reduction in the maximum frequency of the network. The magnitude of this decrease is encapsulated in k_{GRD}.

Our investigation aims to derive closed-form expressions for k_{LRD} and k_{GRD} using a combination of analytical derivations and empirical curve-fitting from experiments. Before experimentally deriving the specific constants to calibrate our model to Xilinx and Altera device families, we present the characteristics of our generated benchmark circuits. Using these characteristics, we illustrate that there are consistent performance trends, indicating that we can create an analytical model capable of predicting performance. First, an investigation is performed to compare our current generation scripts with previous work [51] to demonstrate how improvements in CAD tools might have improved routability in our new benchmarks. Since Altera and Xilinx FPGAs share the same routing fabric for their FPGAs, we expect that the analytical model is the same for all FPGAs. Only the coefficients are tuned for Altera FPGAs and Xilinx FPGAs, thus the Altera analytical model can be applied to all Stratix FPGAs, and the Xilinx analytical model can be applied to all Virtex FPGAs. Therefore, in order to verify this, we analyze the variation in performance for the same benchmark circuits amongst different Xilinx and altera FPGAs.

After analyzing the characteristics of our benchmark circuits in comparison with previous work and among different FPGAs, we present the general performance trends exhibited by specific NoC topologies that strain local and global routing demand on FPGAs. Resource usage is investigated first to demonstrate if NoC performance is dictated by resource usage alone or by the characteristics of the NoC. After, an analysis is performed to determine the effects of the network node alone by: 1) changing the number of nodes, 2) changing the node size, and 3) investigating heterogeneous and homogeneous network nodes. These results are then included in the investigation of local and global routing demand and its effects on NoC performance. The analysis in this chapter is highlighted using the Xilinx Virtex 5 LX330 and the Altera Stratix III 340 for simplicity. All results are consistent for different devices within the same vendor, which will be shown in Section 4.2. These two FPGAs are chosen since they have approximately the same resource availability; however, each vendor has a different FPGA architecture. The Altera Stratix III 340 has the equivalent of 338,000 4-input LEs and the Virtex 5 has 207,360 6-input LUTs, thus we will expect some performance differences between the two FPGAs. The results found in this section are applicable to all Xilinx and Altera FPGAs investigated.

4.1 Previous Research

Previous work developed a custom framework that only instantiated MPSoCs on Xilinx FPGAs[51]. This framework is restrictive in scope, thus in this work, a new framework is developed that can generate a wide range of NoCs including MPSoCs on both Xilinx and Altera FPGAs. Since our experiments utilize newer releases of the Xilinx CAD tools, we compare our generated benchmarks with previous benchmarks to determine how improvements in CAD tools might affect performance and routability.

Previous research showed that homogeneous multiprocessor NoCs exhibited unique trends on FPGAs compared to ASICs. The previous experiments used the MicroBlaze as the computing node and were synthesized to the Xilinx Virtex 2 Pro and Virtex 4 families of FPGAs using Xilinx EDK 7.1. We repeat these experiments with our tool flow using Xilinx EDK 10.1.02 for the ring, star, hypercube, and mesh topologies with 8, 16, and 32 nodes on a

Virtex 4 LX200. For the fully connected topologies, we performed experiments from 8-32 nodes at 4 node intervals. We used the MicroBlaze as the computing node and all IP core versions are the same as previous experiments.

Our new experiments use an average of 8% fewer resources than the previous experiments with a standard deviation of 1.1%. This can be attributed to CAD tool improvements in the newer releases of Xilinx's tool suites. Performance, in terms of max frequency, shows little change with an average improvement of 4.1% and standard deviation of 2.1%. These results exclude the fully connected topologies between 22 and 32 nodes that EDK 10.1.02 can now route, but were previously unroutable by EDK 7.1. Taking into consideration the changes due to an updated tool flow, our new framework is capable of generating MPSoCs that are consistent with previous results. We used the updated system generation scripts are used to produce all the NoCs aimed at characterizing different network parameters and their effects on performance. Before doing this, we first take a look at how the choice in FPGA device can effect performance.

4.2 Chip Independence

Since Xilinx and Altera FPGAs utilize the same routing fabric among their device families respectively, we expect that we should be able to apply the same analytical model. However, as they are two different FPGA vendors, we expect their respective fabrics to be different. Therefore, while the analytical model should be consistent, the coefficients defining the model have to be tuned to each vendor. In order to determine if this is true, we first look at the effects of changing device families when using the same network topologies. As explained in Section 3.3, a wide range of FPGAs are used to fully characterize the performance of NoCs on a number of platforms. Analyzing the performance results for different Xilinx and Altera FPGAs, we find that different devices and families within the same FPGA vendors exhibit similar trends in performance. When performance is normalized to an eight node ring topology for the selected family, performance has minimal variation between families and devices. Our training experiments involve the devices listed in Table 3.3. Figure 4.1

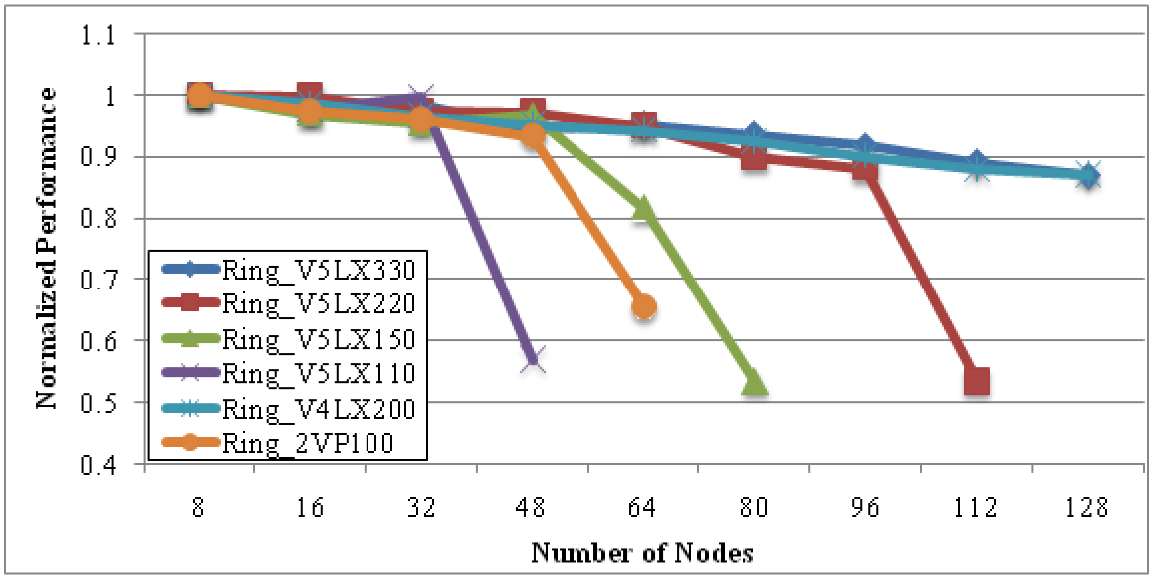

Figure 4.1: Ring topology on Xilinx FPGAs

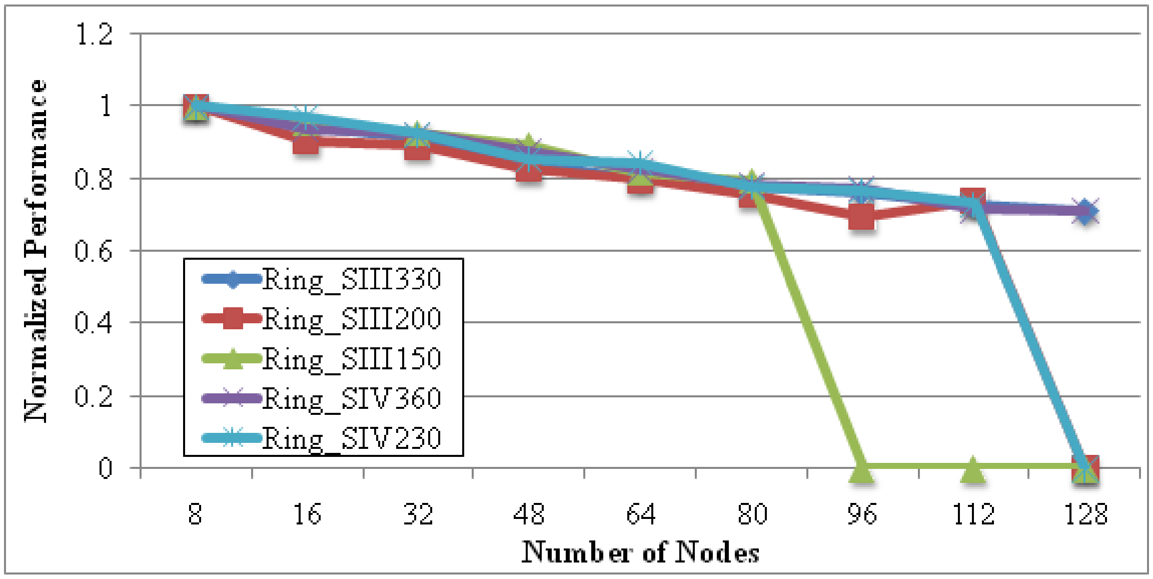

Figure 4.2: Ring topology on Altera FPGAs

Table 4.1: Performance variation for FPGA devices and families

	Xilinx		Altera	
	Variation	*Standard Deviation*	*Variation*	*Standard Deviation*
Family	3.5%	1.4%	4.1%	1.7%
Device	4.2%	1.6%	5.1%	1.2%

highlights this minimal variation by showing the ring topology for different Xilinx families and devices and Figure 4.2 shows them for different Altera families and devices.

For our Xilinx and Altera experiments, we find that there is minimal variation in performance when moving between different families and devices as shown in Table 4.1. Moving between different families exhibit smaller variations since the chosen devices between families have an equivalent amount of available resources. When moving between different devices, smaller devices have a higher percentage of global routing demand for larger designs. This is shown in Figures 4.1 and 4.2 by the dramatic drops in performance for smaller devices. Thus for very large systems on small devices, systems often fail to route or have very low performance compared to the same design on a larger device. Since performance has minimal variation between families and devices, the same conclusions drawn from our analysis can be applied to all Xilinx Virtex 2 Pro, Virtex 4, Virtex 5, and Virtex 6 FPGAs for our analysis of Xilinx FPGAs and all Altera Stratix III and Stratix IV FPGAs for Altera FPGAs.

4.3 Resource Usage

As seen in the previous sections, although trends between FPGA devices remain consistent up to a certain point, we see a drop of performance for smaller devices for a large number of nodes. We expect that this drop is due to resource usage. As a design becomes larger, the CAD tools distribute a design over the given FPGA fabric, and wires grow in length thus decreasing the maximum operating frequency of the system. However, long wire lengths are not always directly correlated with resource usage as the critical path could be a result of high local congestion, or poor design. In this section, we attempt to demonstrate the

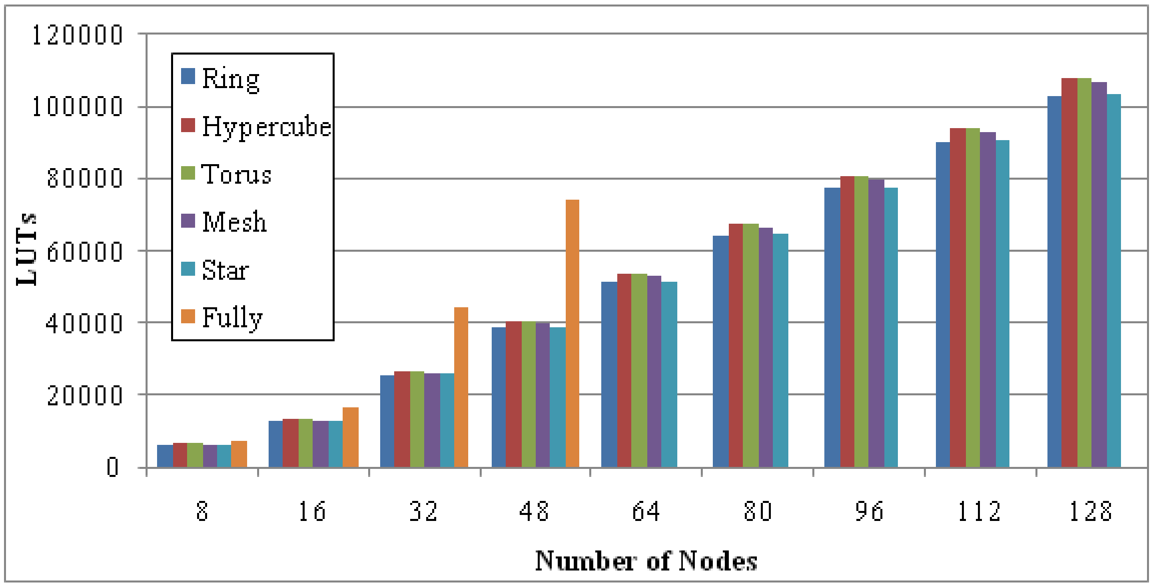

Figure 4.3: Logic utilization of NoC topologies on Xilinx Virtex 5 LX330

importance of resource usage on NoC performance.

In order to look at resource usage, we first look at how resource usage is impacted by different network topologies. Figure 4.3 shows the resource usage of the ring, hypercube, torus, mesh, star, and fully connected topologies with MultBase nodes and 32-bit link widths on Virtex 5 LX330 FPGAs. For all topologies other than the fully connected topologies, resource usage increases linearly as the number of nodes increase. However, for fully connected topologies, network connectivity grows exponentially as the number of nodes increase thus resource usage increases exponentially. Note that the 48 node fully connected topology is not routable, but we map the design to determine the total resource usage that is required. Fully connected topologies with greater than 48 nodes utilizes substantially more resources than other topologies, thus they are not shown.

Figure 4.4 shows the performance for the ring, torus, mesh, hypercube, star and fully connected topologies using the MultBase node on Virtex 5 LX330 FPGAs. Although the ring, torus, mesh, hypercube, and mesh topologies exhibit the same increase in resource usage, each topology experience different rates of performance decrease as the number of nodes

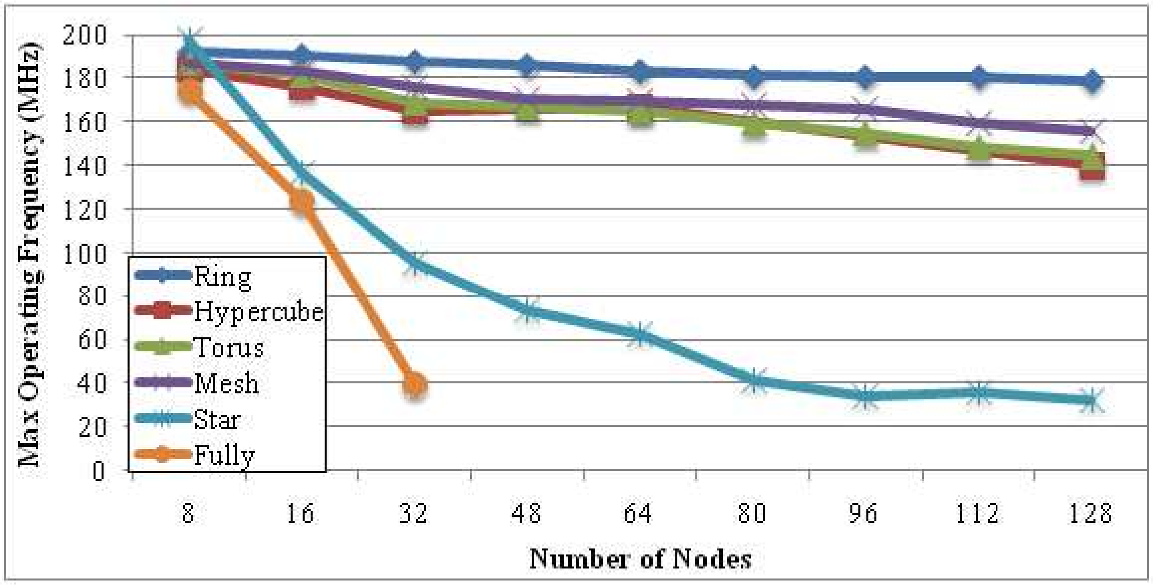

Figure 4.4: Xilinx Virtex 5 LX330 performance for all topologies

Table 4.2: Percentage resource usage of largest NoC system for Xilinx FPGAs

Topology	**Max Percentage**	
	Average	St. Deviation
Ring	84%	3.4%
Hypercube	77%	2.9%
Torus	79%	3.1%
Mesh	82%	3.7%
Average	**80%**	**3.23%**

increase. This suggests that there are multiple factors that contribute to the performance of an NoC, and not just resource usage alone.

Recall that in the previous section, topologies failed routing for larger designs on a smaller FPGA device. This suggests that while resource usage may not play a direct role in performance, it can effect the routability of an NoC topology. We analyze the limits of routability for the ring, torus, hypercube and mesh topologies for Xilinx and Altera FPGAs and show the results in Table 4.2 and 4.3. We did not analyze the star and fully connected topologies as they exhibited unique trends, which will be explained in Sections 4.5 and 4.6.

Table 4.3: Percentage resource usage of largest NoC system for Altera FPGAs

Topology	**Max Percentage**	
	Average	St. Deviation
Ring	71%	4.1%
Hypercube	72%	2.4%
Torus	70%	3.7%
Mesh	73%	2.8%
Average	**71%**	**3.25%**

In Table 4.2 and 4.3, the three homogeneous and three heterogeneous implementations of the topologies listed in Column 1 are analyzed on all devices to determine the percent resource usage for the largest designs that consistently route. The average maximum percent resource usage and standard deviation of the largest topologies that could route are given in Column 2 and 3. From these observations, we demonstrate that the largest NoC topologies that route on Xilinx FPGAs use approximately 80% resource usage and Altera FPGAs use approximately 71% resource usage. When resource usage is below these points, resource utilization has little to no effect on performance. With this in mind, the following sections analyze the effects of different network parameters on local and global routing demand on NoC performance.

4.4 Number of Nodes and Node Sizes

From the previous section, resource usage does not have a direct impact on performance until resource usage exceeds 80% for Xilinx FPGAs and 71% for Altera FPGAs. Above this point, topologies fail to route. Therefore for the purpose of our analytical model, we exclude resource usage and assume that the design is routable given a large routing fabric. In this section we discuss how the network node effects NoC performance by changing the number of nodes in the system and the node sizes.

As SoC complexity increases, larger and more complex computing nodes are used to perform application-specific functionality. Given the fixed FPGA interconnect, we analyze the effect of the number of nodes and node size by synthesizing the ring, torus, hypercube,

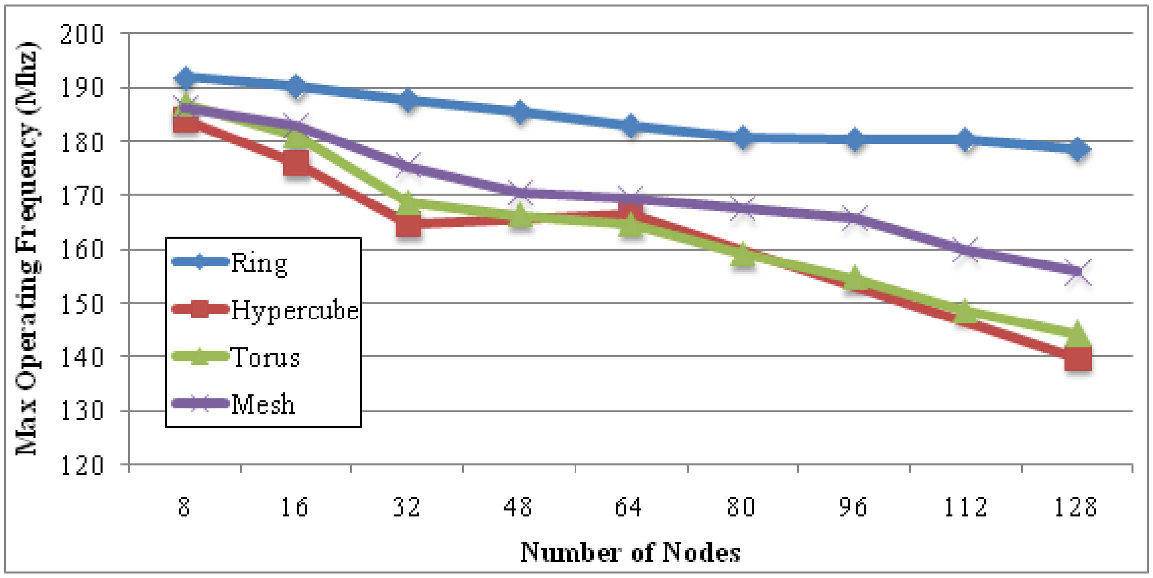

Figure 4.5: Xilinx Virtex 5 LX330 performance for homogeneous topologies

and mesh topologies from 8 to 128 nodes using both homogeneous and heterogeneous node types. The star and fully connected topologies are not analyzed here as they exhibit dramatic drops in performance in Figure 4.4 and represent a subset of topologies that have extreme local and global routing congestion.

Using a homogeneous MultBase node system, we show the performance for select topologies in Figures 4.5 and 4.6. The performance in terms of maximum operating frequency of the NoC degrades as the number of nodes increase. However, from this figure the reason for degradation cannot be isolated to one variable. The drop in performance can be attributed to either the number of nodes or how the number of nodes affect resource usage as NoCs with a greater number of nodes require more routing resources.

Another interesting observation from Figures 4.5 and 4.6 is that while the performance trends for each topology are the same, the absolute performance on the Altera Stratix III 340 is better than the Xilinx Virtex 5 LX330. In Table 4.4, the percent resource usage of a 128-node ring, torus, mesh, and hypercube topologies with MultBase nodes is shown. The same topology implemented on the Altera Stratix III 340 uses on average 15.2% less resources with a standard deviation of 4.3% compared to the Xilinx Virtex 5 LX330. This

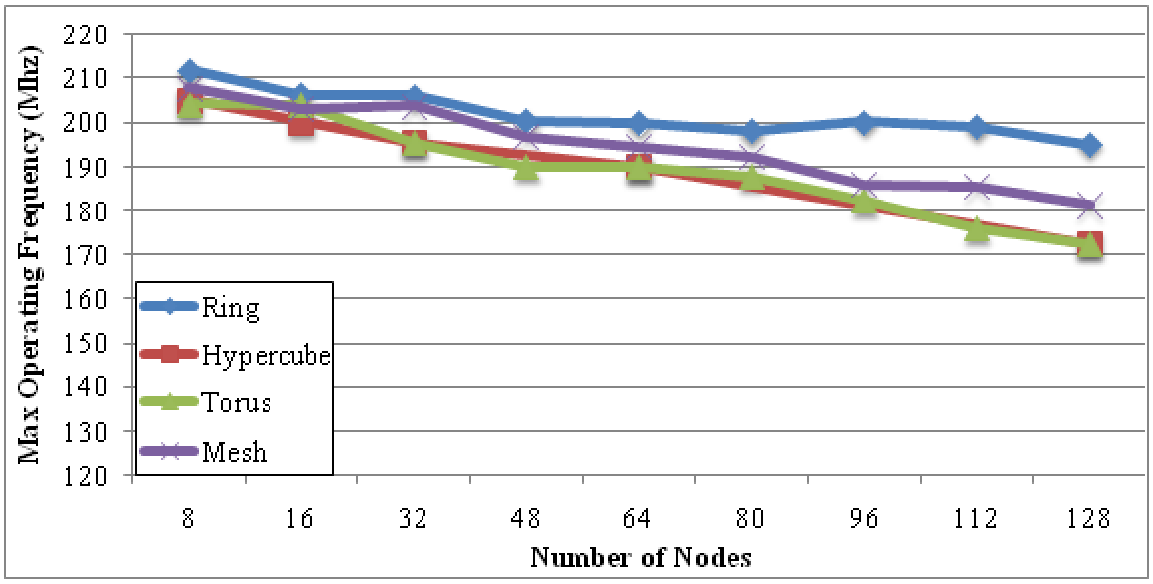

Figure 4.6: Altera Stratix III 340 performance for homogeneous topologies

Table 4.4: Percent resource usage for 128-node topologies

	Altera Stratix III 340	Xilinx Virtex 5 LX330
Ring	55.9%	69.7%
Torus	57.1%	74.5%
Mesh	56.8%	73.1%
Hypercube	57.9%	74.8%

can be attributed to the different FPGA architectures and the total available resources. The Altera Stratix III 340 was released later than the Xilinx Virtex 5 LX330 and uses a significantly different routing fabric, thus we expect the performances to be different. Furthermore, for our training experiments, our Altera experiments used Quartus 9.1, which was released later than Xilinx EDK 10.1. Therefore, the algorithms used in Quartus 9.1 may have improvements not seen in EDK 10.1. These performance differences are seen in all of our experiments. In order to determine if performance loss is due to the number of nodes in a system alone, we extended the experiments to include heterogeneous NoCs with varying ranges of node sizes.

The effects of heterogeneous node sizes are investigated by varying the node sizes for

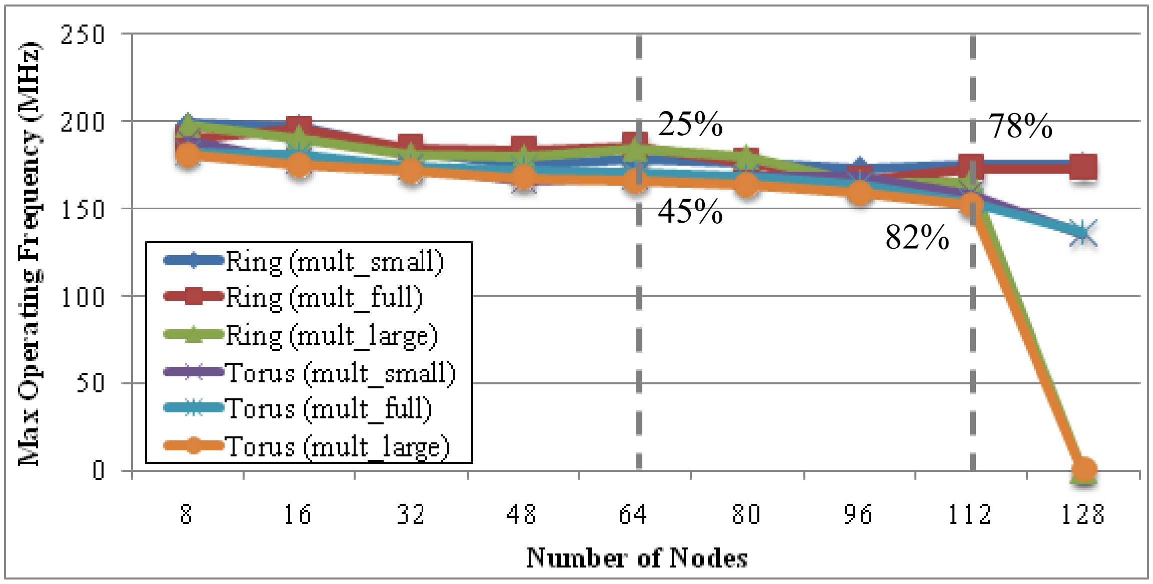

Figure 4.7: Xilinx Virtex 5 LX330 performance for heterogeneous topologies

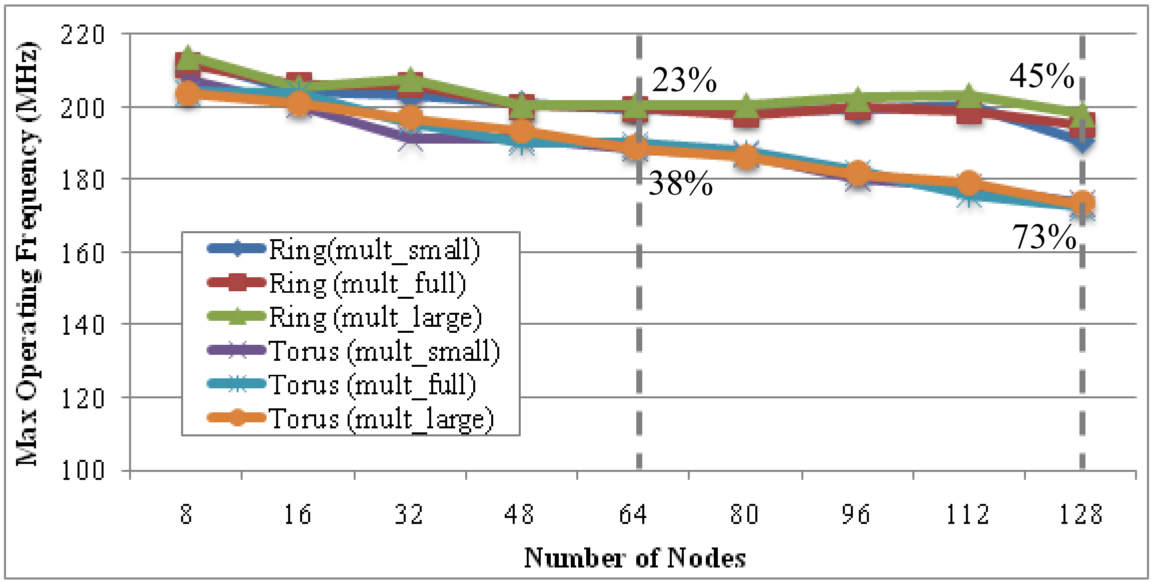

Figure 4.8: Altera Stratix III 340 performance for heterogeneous topologies

all topologies. Figures 4.7 and 4.8 show three different node size ranges for only the ring and torus topologies. We chose the ring topology as it attained the best performance and the torus had the worst (excluding the star and fully connected topologies). The hypercube topology had almost identical performance to the torus topology and the mesh topology had approximately 7.4% better performance than the torus topology on Xilinx FPGAs. The resource usage is also shown for NOCs with 64 nodes by the dotted line. For the ring topology, the percent resource usage for the smallest node range (MultSmall) is 25% and for the largest node range (MultLarge) is 45% for Xilinx FPGAs. Despite the large difference in resource usage, these two systems have the same maximum operating frequency. This suggests that for a given number of nodes, heterogeneity and node size does not affect the performance of an NoC.

From these observations, we conclude that node size matters only in as much as it increases resource usage. When resource usage is under 80% for Xilinx FPGAs and 71% for Altera FPGAs, node size has no effect; however, above this point the resource usage results in a dramatic decrease in performance and routing failure. This is consistent with our conclusions found in Section 4.3. The same conclusions are also verified for Altera FPGAs. Therefore, as long as the network node is isolated from the network using latched routers or switches, and is not the critical path in the design, heterogeneity and the choice in network nodes should not affect NoC performance.

Since performance loss is determined to be due to the number of nodes alone, and not resource usage, we quantify the performance loss due to increasing the number of nodes. Our analytical model uses an 8 node ring topology as the baseline. Therefore, we quantify the performance loss due to the number of nodes for a ring topology by normalizing to the maximum operating frequency of an eight node ring topology, and averaging the performance loss for our three heterogeneous and three homogeneous ring topologies. Other topologies exhibit the same trend, but have different rates of performance change due to other network parameters, which will be shown in later sections.

Figure 4.9 shows the percentage reduction in performance with errors bars from minimum

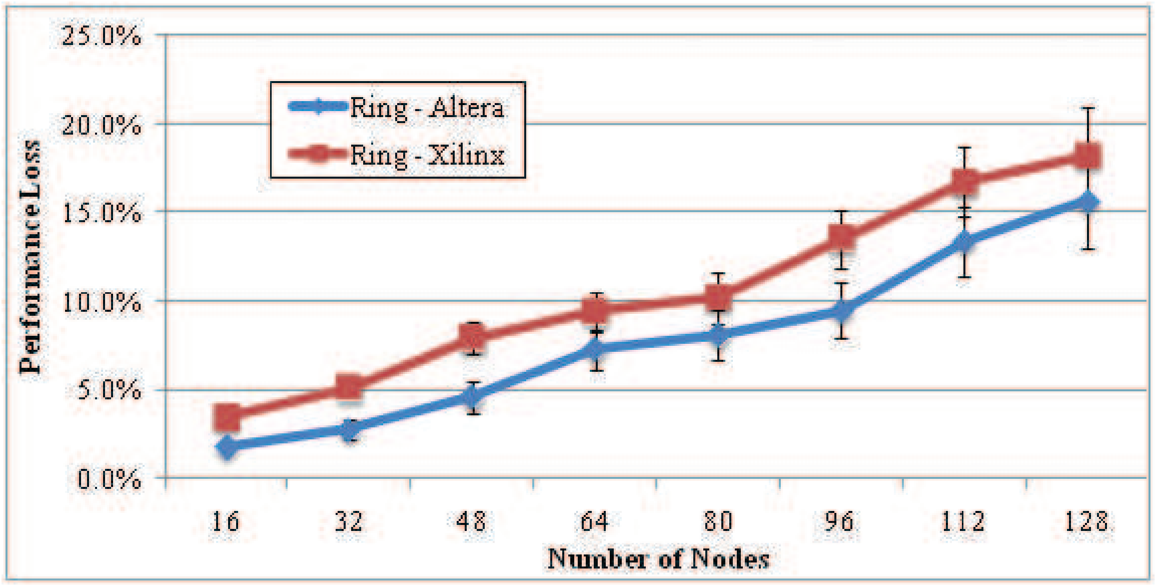

Figure 4.9: Performance loss due to change in number of nodes for a ring topology

to maximum loss in performance versus the number of nodes in a system for the Xilinx Virtex 5LX330 and Stratix III 340. As the number of nodes increases, the drop in performance has an approximately linear increase for the ring topology. From our observations, we conclude that: 1) topology has a greater effect on performance than resource usage, 2) the number of nodes is more important than the node size, and 3) resource usage only becomes significant as it approaches 80% for Xilinx FPGAs and 71% for Altera FPGAs, when larger NoCs consistently fail to route. While other NoC topologies show a similar linear decrease in performance, the rate of decrease is effected by different NoC parameters such as the local and global routing demand. The following sections will illustrate how the local and global routing demand of different topologies affect performance.

4.5 Local Routing Demand

The previous section quantifies how the number of nodes (N) affects the performance of the baseline ring topology. However, different topologies exhibit different rates of performance degradation, thus suggesting that NoC parameters differentiating these topologies

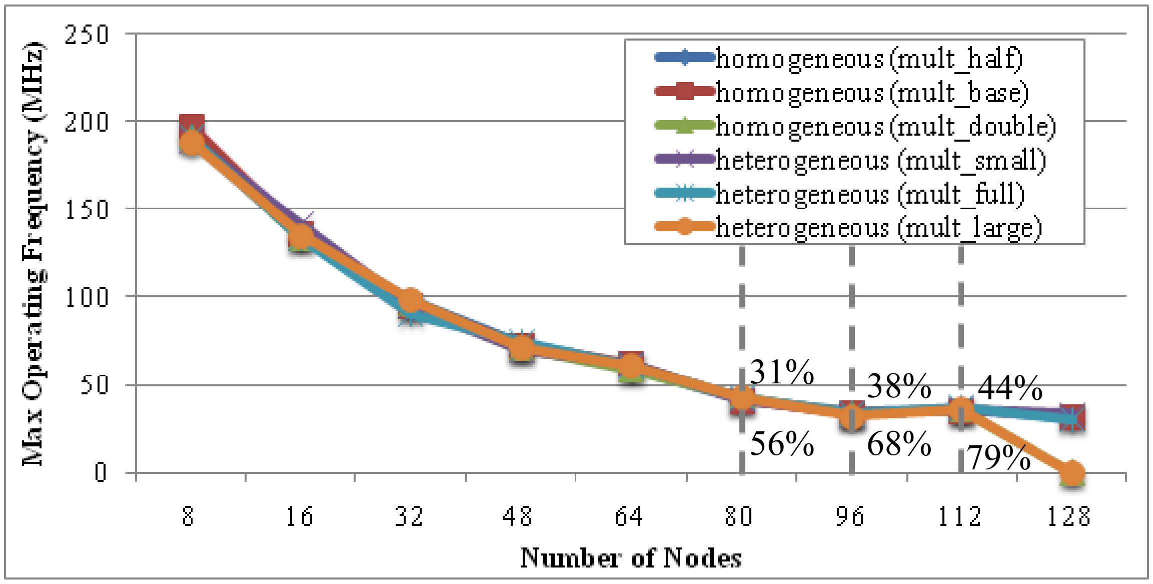

Figure 4.10: Star topology performance on Xilinx Virtex 5 LX330

further effect performance. These parameters can be viewered in terms of the local and global routing demand of the NoC. As described at the beginning of this chapter, the local routing demand factor (k_{LRD}) characterizes the impact of the routing requirements of a single network node. These routing requirements are related to the total number of wires connecting the node's network interface to the communication network. The total number of wires is dependent on both the node degree (ND) and link width (LW) of the node. In this section, we illustrate the effects of local routing demand on performance by isolating the effects due to node degree and link width. We use the star topology with a fixed 32 bit link width to demonstrate the effect of node degree on a single node; a star topology's central node has $ND = N - 1$, while for all other nodes $ND = 1$.

4.5.1 The Star Topology

Figures 4.10 and 4.11 show the performance of the star topology using our three homogeneous and three heterogeneous NoCs on Xilinx and Altera FPGAs. As previously stated, the node size for our heterogeneous star topologies is generated at random. Therefore, we

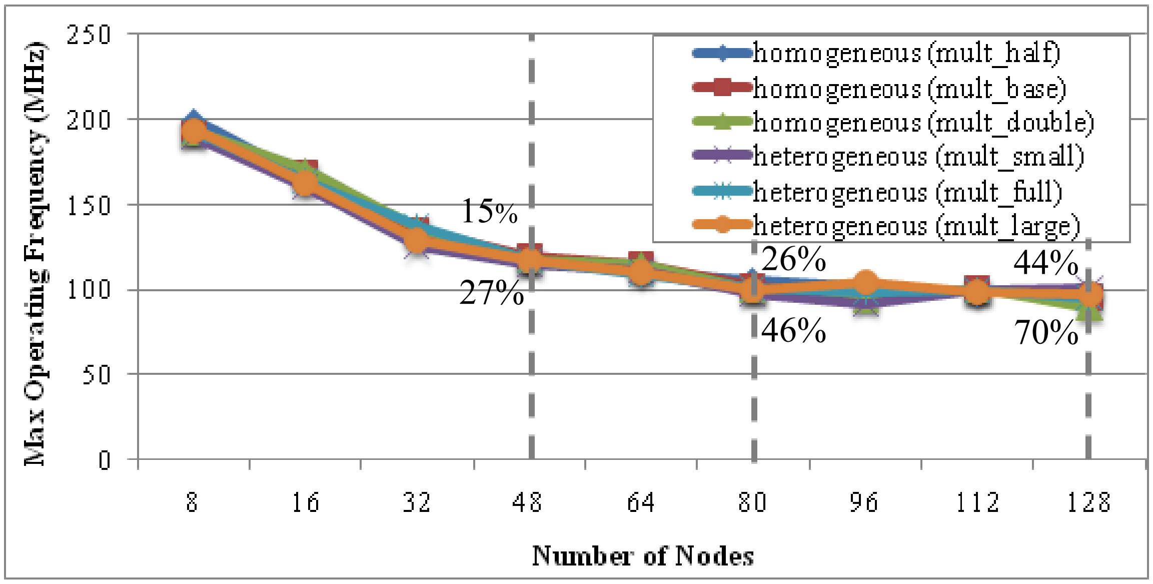

Figure 4.11: Star topology performance on Altera Stratix III 340

do not look at specific heterogeneous cases such as only increasing the size of the central node. As the graph shows, although the central node in a star has an extremely high node degree, very large star topologies are routable as long as there are sufficient resources for the overall system. As the number of nodes increases, more links are added to the central node. This increases the network interface's connectivity, causing the CAD tools to distribute the network interface across the FPGA fabric to enable multiple links to be routed to the central node. Using FPGA editor we analyzed the size of the central node's network switch as shown in Figures 4.12 and 4.13. Figure 4.12 shows the network interface of the central node for a 32-node star topology, and Figure 4.13 shows the network interface of the central node for a 64-node star topology. As the network interface spreads out, longer wires are used to successfully route the design causing a severe degradation in performance.

The numeric labels in Figures 4.10 and 4.11 indicate the logic resource usage of the star topology on a Xilinx Virtex 5 LX330 for 80, 96, and 112-node systems, and on a Altera Stratix III 340 for 48, 80, and 128-node systems. The labels shown above the performance line are for MultSmall NoCs and those below the line represent MultLarge NoCs. Although

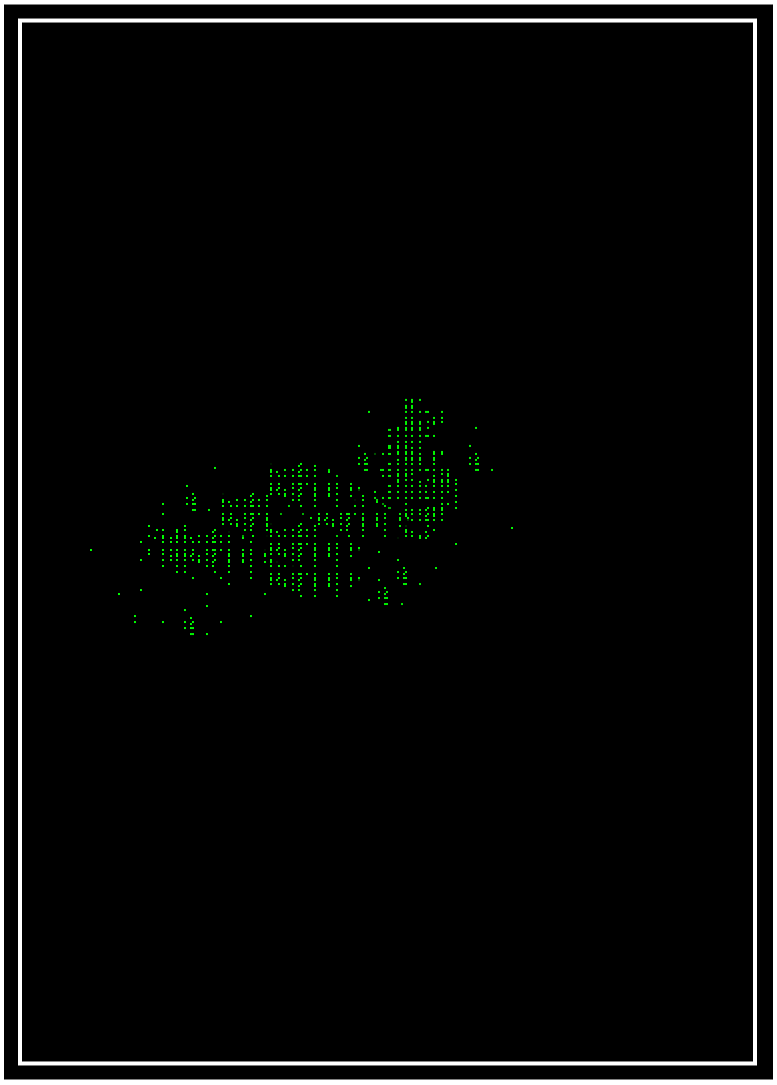

Figure 4.12: Network interface of 32-node star topology on Xilinx Virtex 5 LX330

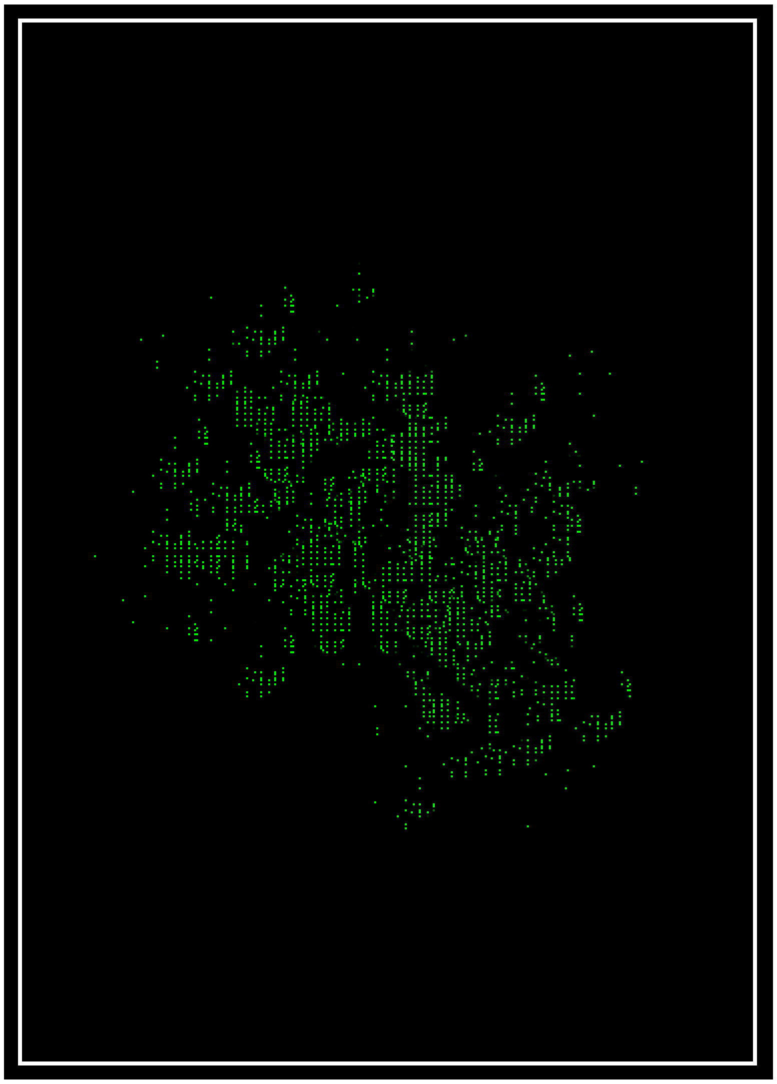

Figure 4.13: Network interface of 64-node star topology on Xilinx Virtex 5 LX330

there is a large difference in resource usage, the overall performance is roughly the same for all heterogeneous systems considered. This suggests that the resource usage does not have a significant impact on performance until routing fails at over $\approx 80\%$ resource usage for Xilinx FPGAs and $\approx 71\%$ for Altera FPGAs, which is consistent with the results from Section 4.4. Since resource usage does not impact performance, the loss in performance is primarily a function of how well the tools manage the wiring demand of the central node. The tools attempt to limit congestion by distributing the network interface of the central node. As a result, the node degree of the central node a significant impact on the topology's performance. However, due to the availability of global routing resources, this degradation eventually flattens out when the network interface is spread over almost the entire FPGA (approximately 64 nodes on the Virtex 5 LX330 and 48 nodes on the Stratix III 340) and larger star topologies will continue to be routable with constant performance, as long as there are sufficient logic resources.

For Altera FPGAs, the star topology flattens out at a higher operating frequency and at a lower number of nodes than for Xilinx FPGAs. This suggests that the Altera CAD tools attempt to limit congestion by distributing the central node's network interface earlier when compared to the Xilinx CAD tools. Furthermore, the subsequent higher operating frequency suggests that either the Altera CAD tools or routing fabric can better leverage global resources to reduce local routing demand.

4.5.2 Link Width

Link width plays an important role in local routing demand as it effects the number of wires each link needs to route. When changing the link width, the number of wires used to route a single link to a network node is directly proportional to the link width. To demonstrate the effect of varying link width on local routing demand for a fixed node degree over an increasing number of nodes, we use the ring and torus topologies.

As seen in Figures 4.14 and 4.15, increasing the link width has a significant impact on performance due to the increase in the number of wires required to connect two network

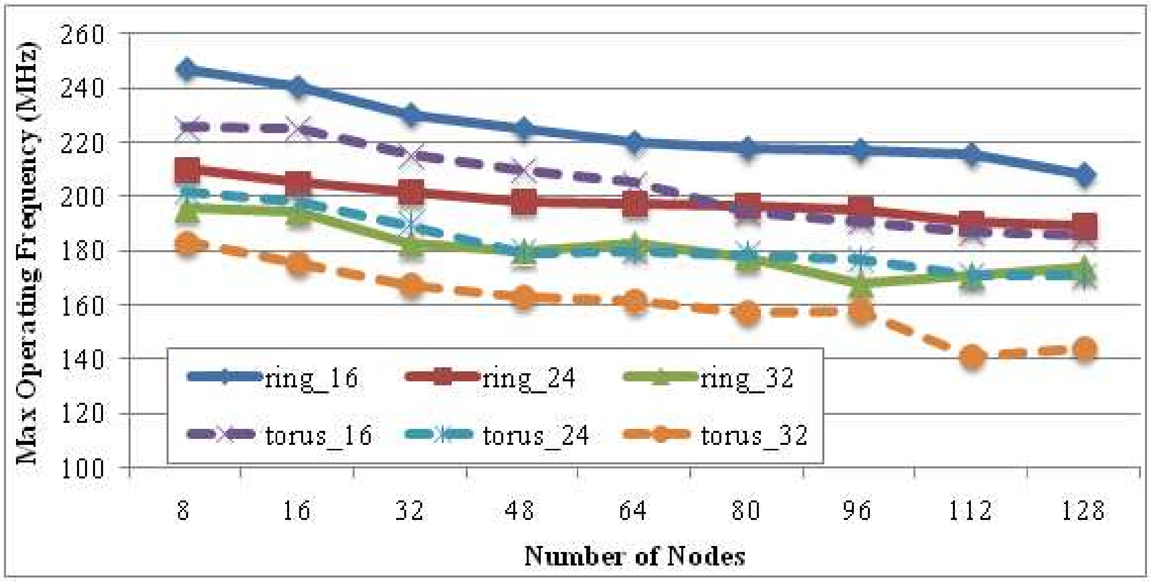

Figure 4.14: Varying link widths on Xilinx Virtex 5 LX330

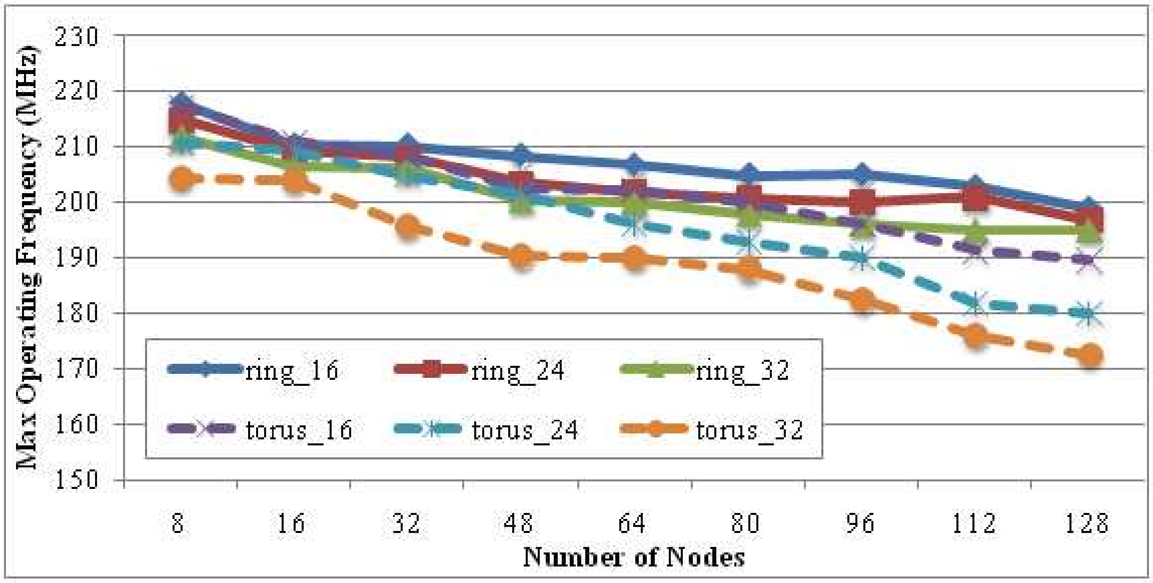

Figure 4.15: Varying link widths on Altera Stratix III 340

nodes to each other. Each line in Figures 4.14 and 4.15 illustrates the average performance of our three heterogeneous and three homogeneous ring and torus topologies from 8 to 128-nodes for a fixed link width of 16, 24, or 32-bits. We also ran experiments for 40 and 48-bit link widths but have omitted them here for simplicity. The dotted lines represent the torus topologies, and the solid lines are the ring topologies. As can be seen in Figures 4.14 and 4.15, generally each line is monotonically decreasing except for a few cases. This can be attributed to the CAD flow, as the CAD algorithms are random, and there still exists a certain amount of unpredictability in performance. However, with our exhaustive experiments, these small variations in performance can be effectively "averaged" out over a large exploration space.

As seen in Figures 4.14 and 4.15, as link width decreases, performance increases for both the ring and torus topology. For the ring topology, decreasing the link width from 32 to 24 bits on a Xilinx Virtex 5 LX330, results in an 8.1% performance increase, while a change from 32 to 16 bits results in a 14.2% increase. The torus topology exhibits slightly larger increases; a change in link width from 32 to 24 bits increases performance by 9.3%, and a change from 32 to 16 bits increases performance by 16.8%. As seen in the previous analysis, the node degree has a significant impact on performance. Therefore, since each node of the torus topology has twice the node degree of a ring, decreasing the link width by 8 bits for the torus topology, results in approximately two times the reduction in wires ($ND \times LW$) when compared to the same change for the ring topology. This results in link width having a greater impact on performance for topologies with higher node degrees.

4.5.3 Bandwidth

Another interesting observation from Figures 4.14 and 4.15 is that a ring topology with a 32-bit link width has a lower maximum frequency than the torus topology with a 16-bit link width. Both networks have the same number of incident wires to each node, and hence the local routing demand should be the same. Using FPGA Editor, we observe that the tools tend to route the wires in a single link using the same global route (along the same set of channels). Thus, networks with larger link widths create a harder routing problem. We investigate this further by analyzing the effect of changing the bandwidth to a single node. This is done using the ring and torus topology, since they are both regular topologies (all nodes have the same number of links). We increase the link width and vary the number of links between nodes to create custom ring and torus topologies.

Each topology is labelled *TopologyX_Y*, with *X* indicating the link width and *Y* indicating the number of total channels (2 unidirectinonal links) between each network node. For example *Ring16_2* is a ring topology with 16-bit link widths, but with two channels between each node. Therefore the baseline 32-bit ring topology is classified as *Ring32_1*. Using these custom topologies, we are able to reduce the link width while increasing the number of links between connected nodes. This allows us to isolate the effects due to individual links. Therefore, a *Ring16_2* and *Ring32_1* should have the same performance as the number of incident links to each node are the same.

The performance of the ring and torus topologies for a variety of link width combinations on a Xilinx Virtex 5 LX330 FPGA are shown in Figure 4.16. A system with 2x16 bit link widths, should have the same performance as a system with 1x32 bit link width since the same number of wires are used to connect two nodes. However, as shown in Figure 4.16, splitting links into smaller link widths tend to perform better than single links with large widths. Therefore, in order to maximize bandwidth, multiple small width links are a better choice than single links with large widths.

In this section, we discuss the effects of local routing demand on performance. Node degree has a significant impact on local routing demand by causing the CAD tools to distribute a single node to alleviate local congestion due to a high node degree. Furthermore, changing the link width has a direct impact on performance as it varies the number of wires connected to each node. The CAD tools seem able to better leverage available resources when the link width is small, thus resulting in systems with multiple links with small bit-widths performing better than systems with single large links.

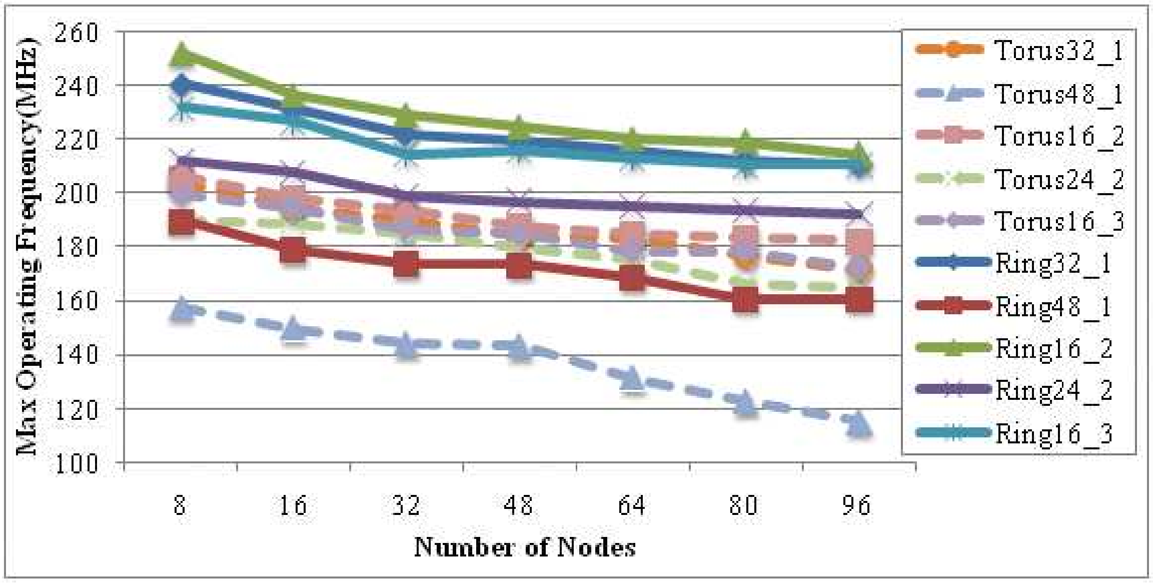

Figure 4.16: Bandwidth analysis for ring and torus topologies

4.6 Global Routing Demand

Local routing demand characterizes the effects of changing the network's routing requirements for a single network node. Recall that in Figures 4.10 and 4.11, local routing demand results in an initial rapid linear degradation in performance. Global routing demand leads to the eventual flattening of performance when the CAD tools leverage global resources to spread out all network links over the entire FPGA fabric. Therefore, as described in the beginning of this chapter, the global routing demand factor (k_{GRD}) characterizes the impact of the routing requirements of all network links. The total number of network links is equal to the number of nodes in the system (N) multiplied by the average node degree (AND). To show the effects due to changing the total number of links on performance, we vary AND and N using the fully connected topology (a regular topology where $AND = N - 1$).

Figures 4.17 and 4.18 show fully connected topologies with 32-bit link widths as a function of the number of nodes, along with a star topology with MultBase nodes as a reference point for comparison. As a fully connected topology grows in size, the impact on performance is severe as the total number of links increases quadratically. For up to 20 nodes

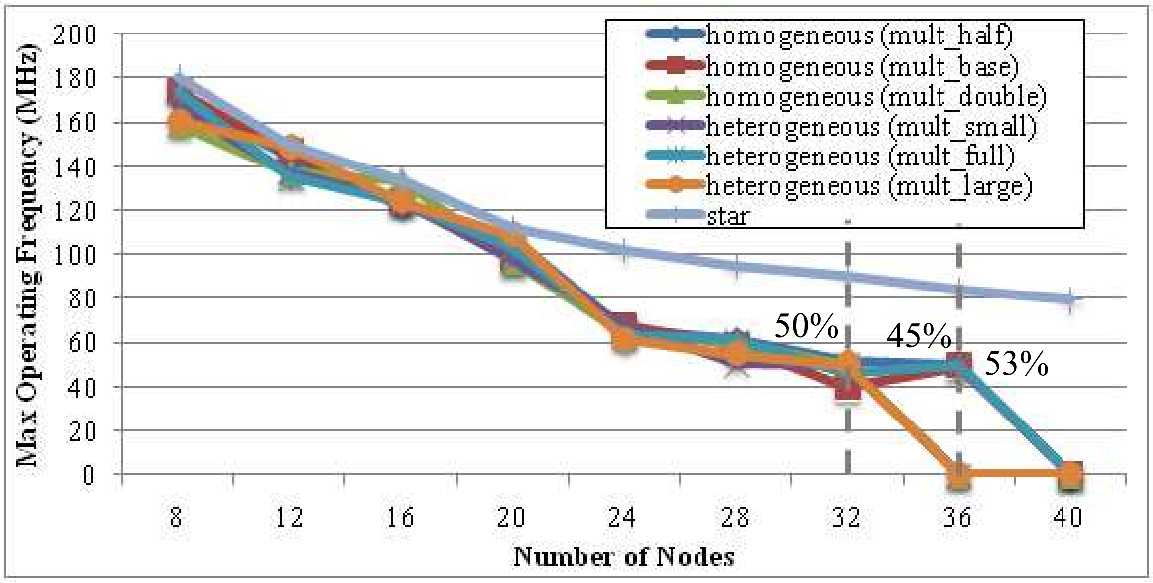

Figure 4.17: Fully connected topologies on Xilinx Virtex 5 LX330

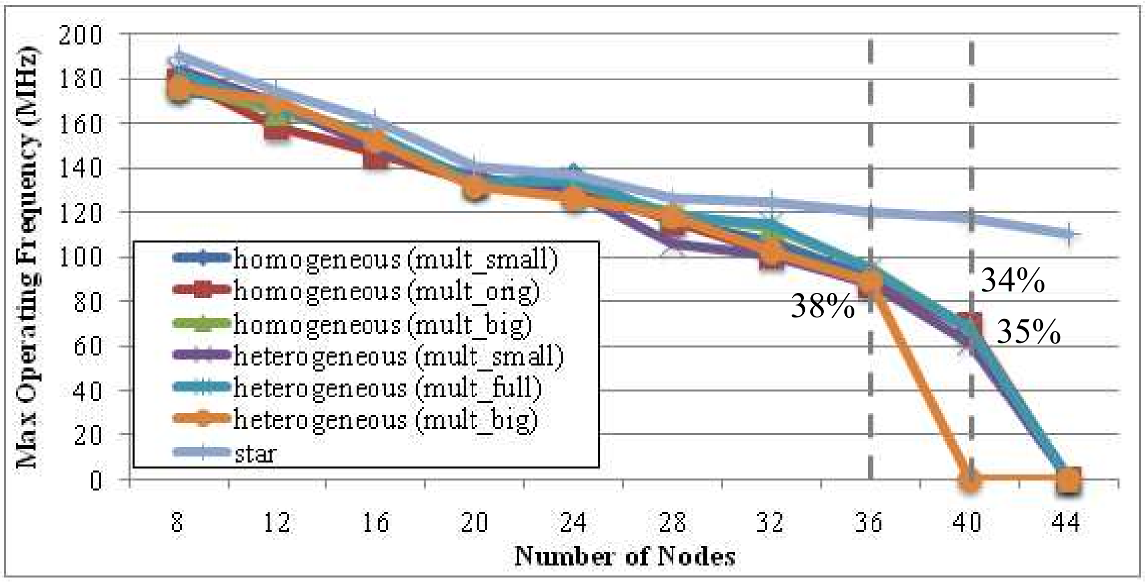

Figure 4.18: Fully connected topologies on Altera Stratix III 340

on the Xilinx Virtex 5 LX330, performance has a rapid linear degradation in performance. Performance then drops dramatically and flattens out at 24 nodes before routing eventually fails. The same trend can be seen on the Altera Stratix III 340. Up to 28 nodes, performance drops linearly and flattens out before routing failure. However, the fully connected topology on the Altera Stratix III 340 exhibit higher operating frequencies before routing failure, suggesting that the Altera CAD tools and routing architecture can better handle global routing demand.

As the average node degree increases up to 24 nodes, much like the star topology, the network interface for each node is distributed to allow the CAD tools to use the available routing resources. However, this is a much more difficult problem than that for a star topology since there are many more links in a fully connected network. Therefore, the designs become unroutable before running out of logic resources, as shown in Figure 4.17 for the Xilinx Virtex 5 LX330 where the largest fully connected topologies capable of routing uses only $\approx 50\%$ of the logic, and for the Altera Stratix III 340 in Figure 4.18 where the largest fully connected topology used $\approx 35\%$ of the logic. Fully connected topologies with 24 and 16-bit link widths exhibit the same trends, with a significant performance drop at 28 and 32 nodes and routing failure at 40 and 44 nodes respectively, on the Xilinx Virtex 5 LX330. For the Altera Stratix III 340, significant performance drops occur at 32 and 36 nodes, and routing failure at 44 and 48 nodes for 24 and 16-bit link widths respectively.

Comparing Figures 4.10 and 4.11 and Figures 4.17 and 4.18 reinforce the greater impact of global routing demand than local routing demand; we see that for the Xilinx Virtex 5 LX330, the performance of the two topologies varies by only 8.2% up to 20 nodes and then diverges as the star topology's performance flattens out and the fully connected topologies performance continues to decline. For the Altera Stratix III 340, the performance of the two topologies varies by 6.9% up to 28 nodes and then diverges. This suggests that up to these divergence points, the CAD tools are able to leverage the global routing resources to facilitate fully connected topologies. However, above this point the impact due to the global routing demand of all nodes outweighs that of the local routing demand of each node,

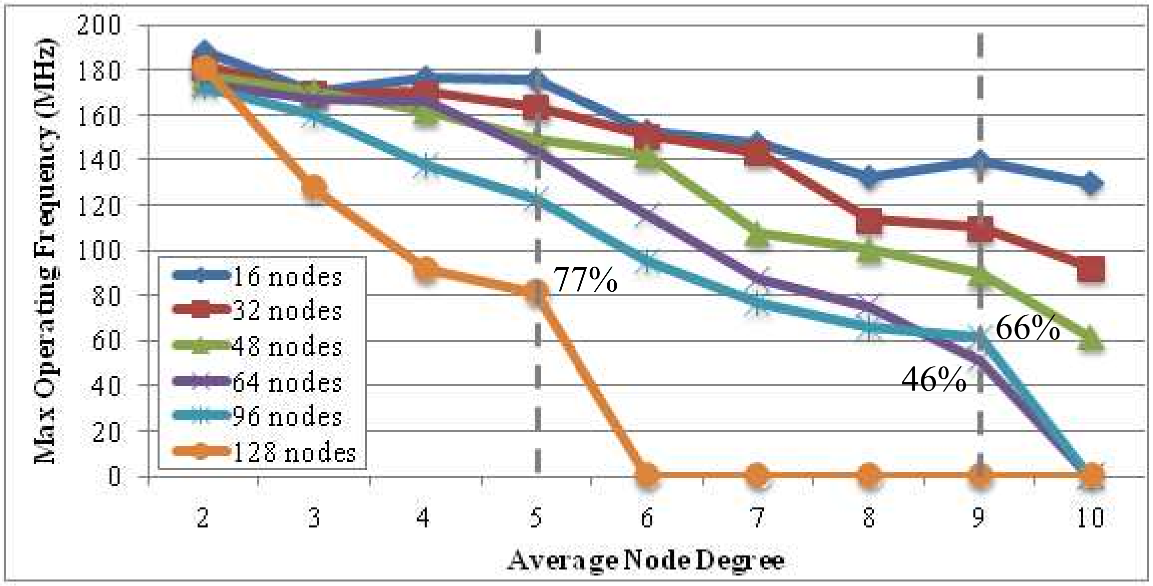

Figure 4.19: Heterogeneous random topologies on Xilinx Virtex 5 LX330

resulting in rapid performance decline and routing failure.

Fully connected topologies show that the total number of links have a significant impact on performance by varying the number of nodes *(N)* and average node degree *(AND)*, without isolating the two variables as they both increase at the same rate. In order to isolate the effects of average node degree from the number of nodes, we create and map benchmarks containing random topologies ranging from 16 to 128 nodes with average node degrees of 2 to 10. Figures 4.19 and 4.20 show the performance results for heterogeneous NoCs utilizing the MultFull range of node sizes with 32-bit link widths, where each line represents random topologies with a fixed number of nodes. The topologies with 16 nodes have the highest performance, which degrades as the number of nodes increases to 128. For a fixed number of nodes, the performance decreases almost linearly as the average node degree increases, until routing fails. The rate of degradation increases as the number of nodes increases, as shown by the increase in the slope's magnitude for each line in Figures 4.19 and 4.20. This is because for a fixed number of nodes, N links are added as the average node degree increases by one. Thus, we expect a greater drop in performance for systems with

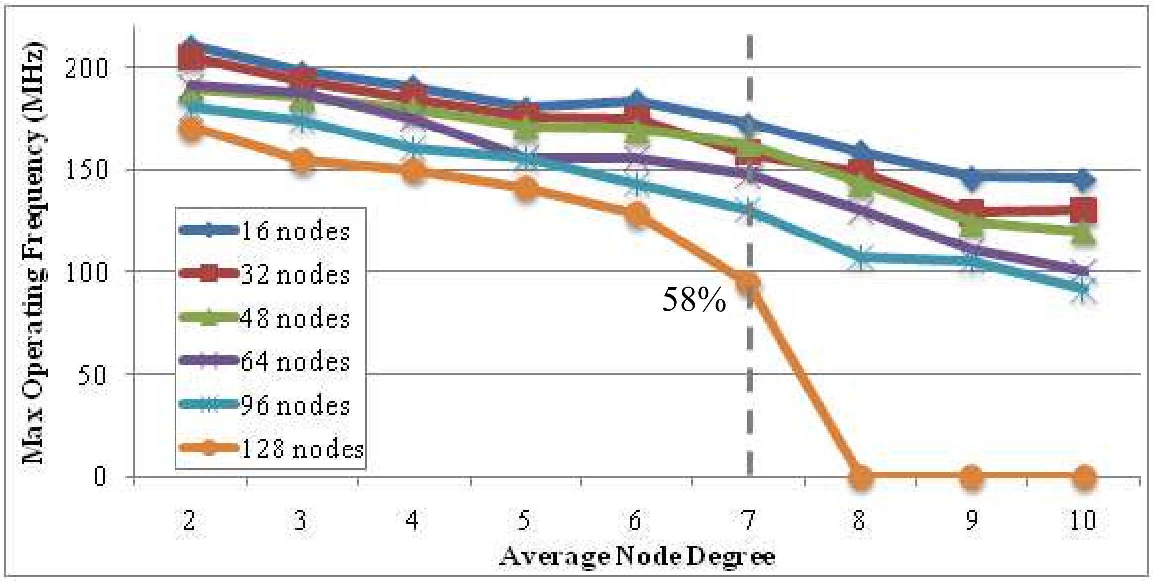

Figure 4.20: Heterogeneous random topologies on Altera Stratix III 340

more nodes. Furthermore, comparing Figures 4.19 and 4.20, we see that performance on the Altera Stratix III 340 exhibit less variations as shown by the “smoother” lines compared to the Xilinx Virtex 5 LX330. This cis due to the Altera CAD tool differences with the Xilinx CAD tools.

Much like the fully connected topology, systems with high average node degrees fail to route before all logic resources are utilized. For example on the Xilinx Virtex 5 LX330, for a 64-node random topology in Figure 4.19, the highest average node degree that consistently route is nine, using 46% of the logic. 96 and 128-node random topologies are routable with an average node degree of nine and five, respectively, but only use 66% and 77% of the total resources. As shown in Figure 4.20, For the Altera Stratix III 340, a 128-node random topology with an average node degree of seven only use 58% of total resource usage before routing fails at an average node degree of eight. This is because for a high average node degree, the stresses of global routing demand requirements on the CAD tools for the FPGA fabric cause routing to fail well before logic utilization approaches 71% for Altera FPGAs and 80% for Xilinx FPGAs. Therefore, using the results found in this section, we utilize *AND* and *N* to determine the effects that global routing demand has on network

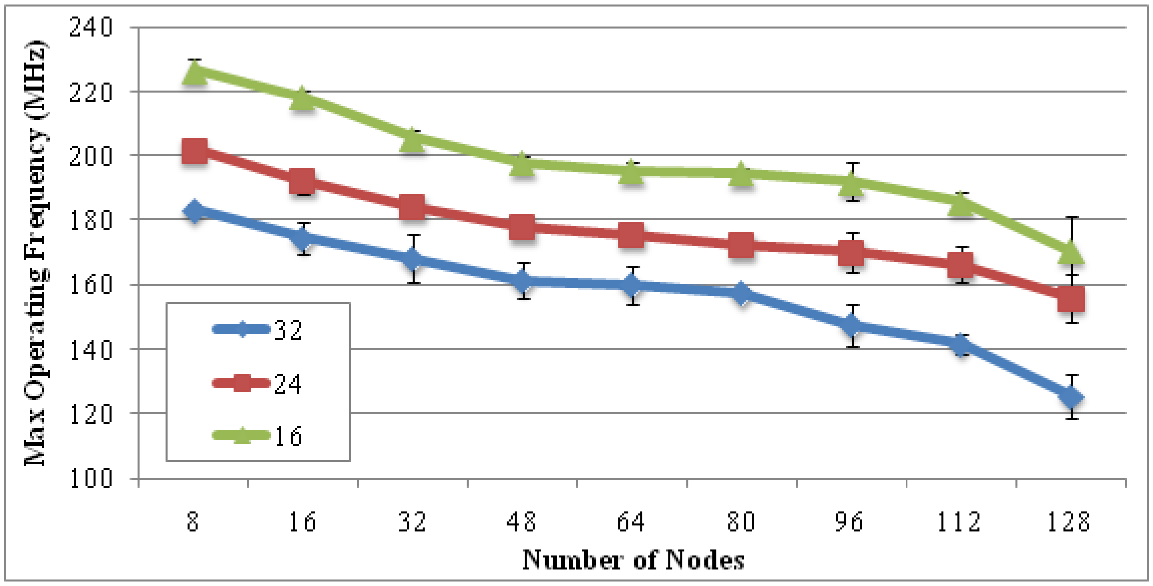

Figure 4.21: Performance of torus topology and random topologies on Xilinx Virtex 5 LX330 $(AND = 4)$

performance in our analytical model.

4.7 Regularity

In the previous sections, topologies such as the ring, torus, and fully connected topologies represent regular topologies where all network nodes have the same number of links. However, the vast majority of networks use irregular topologies such as the mesh or application-specific topologies where the node degree of each each network node is different. Many applications may benefit from an irregular topology, which is optimized specifically for that application. When quantifying the performance loss due to global and local routing demand, we did not take regularity into consideration when determining the parameters that affect performance. Therefore, in this section, we demonstrate that application-specific topologies modeled using random topologies have minimal variation in performance compared to regular topologies.

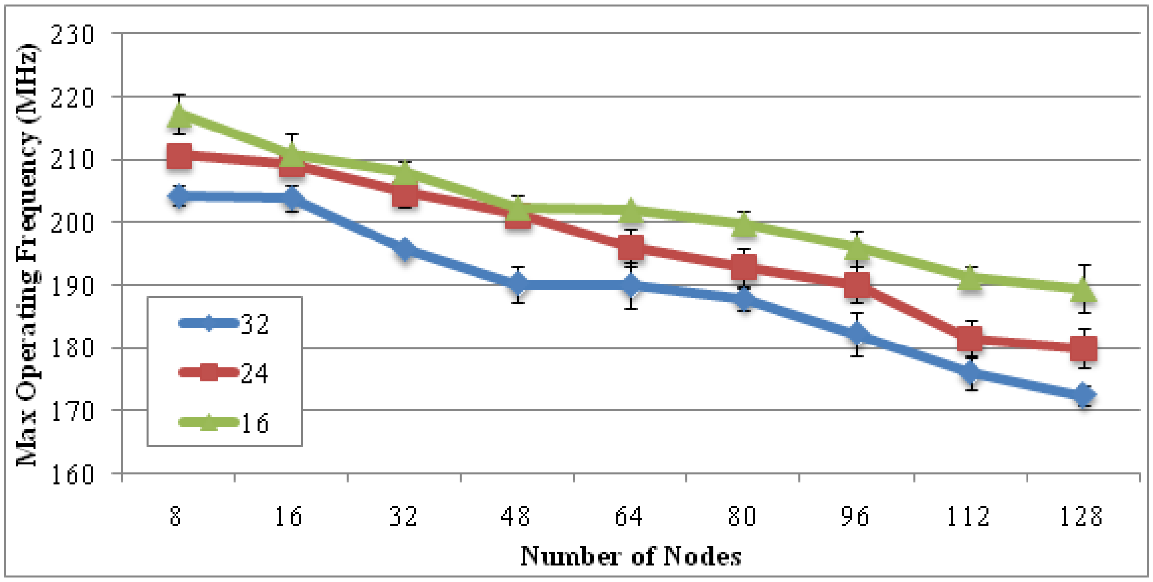

Figure 4.22: Performance of torus topology and random topologies on Altera Stratix III 340 ($AND = 4$)

Figures 4.21 and 4.22 illustrate the average performance of all heterogeneous and homogeneous torus topologies compared to ten random topologies with the same number of nodes *(N)*, the same average node degree *(AND)*, and the same link width *(LW)*. Each line corresponds to the performance of the torus topology for a fixed link width and the error bars represent the performance variation exhibited by the corresponding random topologies. The error bars show an average variation of 2.3% and a maximum variation of 4.8% for Xilinx FPGAs and an average variation of 2.1% and a maximum variation of 3.9% for Altera FPGAs. These results suggest that an expression written in terms of only the number of nodes *(N)*, average node degree *(AND)*, and link width *(LW)* can apply to irregular application-specific networks as well as regular networks. This allows our analytical model to encompass both regular and irregular topologies so long as they can be modelled by these parameters. The derivation and calibration of the factors k_{GRD} and k_{LRD}, in terms of these variables, are described in the following chapter.

Chapter 5

Analytical Model

Based on the results found in the past sections, we analytically derive the equations for k_{GRD} and k_{LRD} and tune the coefficients using experimental curve fitting to predict the maximum operating frequency of an NoC implemented on Xilinx and Altera FPGAs. As described in Section 3.3, a number of FPGAs are used in our training experiments to tune the coefficients of our model. The equations for k_{GRD} and k_{LRD} are presented in this section along with the empirical curve-fitting used to tune the coefficients of our model.

5.1 Xilinx Analytical Model

We use experiments run on three different Xilinx FPGA families as our training data. Each device family has a unique frequency, F_{base}, for the 8-node ring topology with 32-bit link widths that we use as our baseline architecture. For our training data, the baseline frequencies used in our model are 190MHz for Virtex 5, 145MHz for Virtex 4, and 110MHz for Virtex 2 Pro. Recalling from Equation 4.1, we choose to express the model in terms of k_{GRD} and k_{LRD} multiplied by the F_{base}. Therefore, the predicted frequency represents the remaining percentage of F_{base} after accounting for changes in global and local routing demand with respect to our baseline architecture. If we wish to "predict" the performance of our baseline 8-node ring topology with 32-bit link widths, we expect $k_{GRD} = k_{LRD} = 1$

and $F_{pred} = F_{base}$.

5.1.1 Local Routing Demand

Local routing demand (k_{LRD}) describes the performance impact from the perspective of a single network node. As discussed in Section 4.5, local routing demand is directly correlated with the node degree (ND) and link width (LW) of the network node. Since our focus is on application-specific topologies, we simplify our model by approximating the node degree of individual nodes in the topology as being equal to the average node degree ($ND \approx AND$). However, as discussed previously in Section 4.6, the average node degree also affects the global routing demand of the entire system. Therefore, we chose to only encapsulate the effects due to a change in link width and how it is magnified by average node degree in k_{LRD}. The overall effect of average node degree will be encapsulated in k_{GRD} and discussed in the following section.

Increasing the link width increases the number of wires to a node, causing the CAD tools to distribute the network interface, creating longer wire lengths and impacting performance. As average node degree increases, a fixed change in link width will result in more wires being added for higher average node degrees thus further impacting performance. Analytically, this corresponds to a linear equation for a change in link width. The magnitude of the slope will then change depending on the average node degree. Therefore k_{LRD} is expressed as a change in link width magnified by the average node degree. The expression for local routing demand is shown below:

$$k_{LRD} = LRD_{SL} \times \Delta LW + 1 \tag{5.1}$$

where ΔLW represents the change in link width given by ($\Delta LW = LW_{F_{pred}} - LW_{F_{base}}$ where $LW_{F_{base}} = 32$). If $\Delta LW = 0$ then there is no change in link width and ΔLW reduces to zero resulting in $k_{LRD} = 1$. The slope (LRD_{SL}) describes the rate of change of performance due to link width for a fixed average node degree.

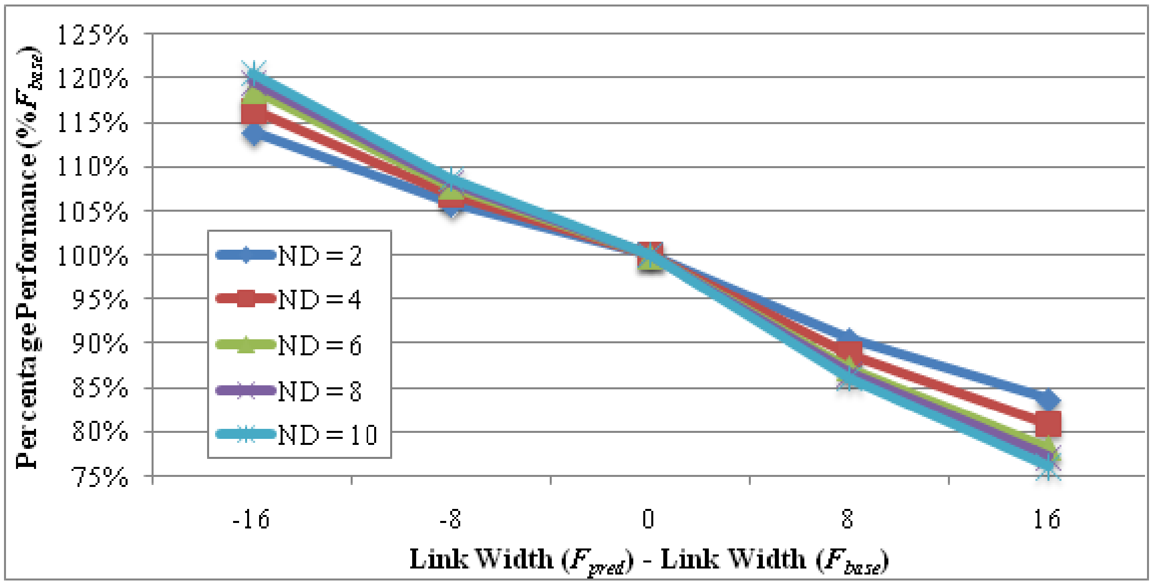

Figure 5.1: Performance variation due to a change in link width for Xilinx FPGAs

In order to calculate LRD_{SL}, Figure 5.1 shows a family of lines representing the performance of all topologies as a percentage of F_{base} with respect to changing the link width, where each line in the figure represents topologies with a constant average node degree. The horizontal axis indicates the change in link width from the 32-bit link width in the baseline architecture (i.e. $\Delta LW = LW_{F_{pred}} - 32$). For a fixed average node degree, a constant change in link width results in a constant change in the number of wires added to a network node. Since the number of wires is directly proportional to the link width ($numberofwires = ND \times LW \approx AND \times LW$), the impact on performance due to change in link width has the linear relationship given in Equation 5.1 for a constant value of AND.

As shown in Figure 5.1, adjusting the average node degree varies the impact of changing the link width on performance, corresponding to the slope of each line (LRD_{SL}). For example, if a node has $AND = 2$, then increasing the link width by 8 bits requires 16 additional wires to be routed to that node. However, if a node has $AND = 3$, then 24 more wires must be routed to that node. Therefore a topology with an average node degree of three would result in more wires being routed, and thus an increased local routing demand (on average), than the same change in link width for a topology with an average node degree of two. Thus, the higher the average node degree, the greater the effect link width has on performance. Since the value of LRD_{SL} is directly proportional to the average node degree, the slope remains constant for a fixed average node degree, and decreases linearly as the average node degree increases. We isolate the slope (LRD_{SL}) of each line in Figure 5.1 and map the change in slope as a linear relationship, deriving the slope and intercept through curve fitting as:

$$LRD_{SL} = (-1.21 \times 10^{-3} AND) - (4.62 \times 10^{-3}) \tag{5.2}$$

5.1.2 Global Routing Demand

Global routing demand (k_{GRD}) is characterized by the routing requirements of the entire system. Independent of network node, type, and size, global routing demand is directly affected by the total number of links in the system ($N \times AND$). In order to model these effects, we first consider how average node degree impacts performance and how the number of nodes magnifies this effect. An increase in average node degree results in more links being added to the topology since $\Delta Links = N \times \Delta AND$. As the number of nodes in a system increases, a change in average node degree results in even more links being added. The number of links affects performance by increasing the number of wires in the system, thus expanding the network over the fabric and effectively decreasing performance. Since k_{GRD} is directly correlated with the number of links, the effect due to k_{GRD} is shown below:

$$k_{GRD} = GRD_{SL} \times (AND - 2) + GRD_{INT} \tag{5.3}$$

The linear equation is characterized by a slope, GRD_{SL}, and an intercept, GRD_{INT}, which vary when the number of nodes change. Since k_{GRD} is equal to one for an eight node ring topology ($N = 8$, AND =2), the ($AND - 2$) terms reduces the slope to zero and the intercept is equal to one when $N = 1$. In order to determine GRD_{SL} and GRD_{INT}, we analyze the effect of varying average node degree for sets of NoCs with fixed numbers of

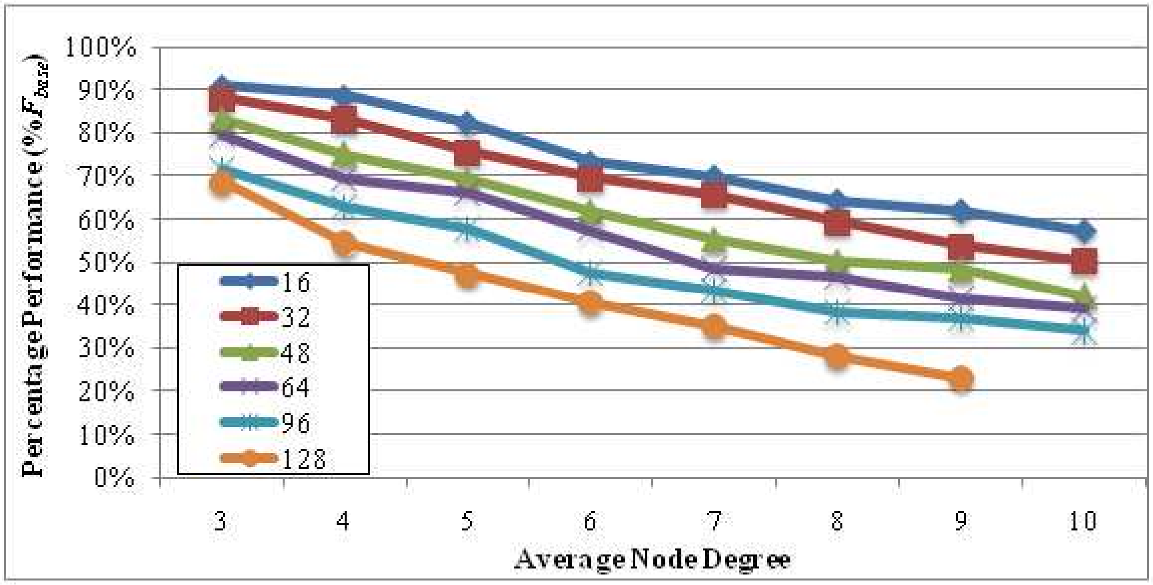

Figure 5.2: Performance loss due to average node degree for Xilinx FPGAs

nodes. Figure 5.2 shows the remaining percentage of F_{base} as a function of average node degree for a given number of nodes. Each line corresponds to a set of topologies with the same number of nodes and link width at varying average node degrees as expressed in Equation 5.3. For a fixed number of nodes, as the average node degree increases, a constant number of wires is added to the network. This number of wires is directly proportional to the average node degree, thus a change in average node degree from 2 to 3, or 4 to 5 will always result in the same number of additional wires being added to the system.

When moving between the different lines in Figure 5.2, the slope (GRD_{SL}) and intercept (GRD_{INT}) change to reflect the rate of performance loss (slope) and maximum possible performance (intercept) for that number of nodes. Since the slope (GRD_{SL}) describes the rate of performance decrease when changing the average node degree for a fixed N, it varies when N changes. For example, changing the average node degree of a system with 128 nodes should have a higher impact on performance than changing the average node degree of a 16-node system as there are a greater number of links added to the system. For a change in average node degree of one, adding 128 links results in significantly more network connectivity than 16 links. Analytically, this corresponds to the slope having a polynomial relationship to account for the increased connectivity in very large systems. In order to find the different values of GRD_{SL} for each number of nodes, we find the slope of each line in Figure 5.2. We determine that the slope has a square polynomial relation to the number of nodes and increases in magnitude when the number of nodes increases.

$$GRD_{SL} = (-2.48 \times 10^{-6} N^2) - (2.61 \times 10^{-4} N) - (3.58 \times 10^{-2}) \tag{5.4}$$

The slope (GRD_{SL}) takes the form of a square polynomial to account for the increased complexity of routing very large systems. For small systems (under 64 nodes), the N^2 term becomes negligible under 0.01) and a linear relationship approximates the effect of the CAD tools distributing the topology over the FPGA fabric. However, as systems and their corresponding network become very large, the tools are challenged to find available global resources, which cause a significant decrease in performance resulting in the N^2 term.

The intercept (GRD_{INT}) defines the fixed performance loss when $N = 8$ and $ND = 2$ and changes as the number of nodes varies. Increasing the number of nodes should result in a linear decrease in performance as adding one node to a topology results in a fixed gain in routing demand. In other words, going from 10 to 11 nodes or 100 to 101 nodes for a fixed AND results in the same increase in routing demand and consequently the same decrease in maximum performance. Therefore, we isolate GRD_{INT} for each line in Figure 5.2 and using curve fitting, we find that the value of GRD_{INT} decreases linearly as the number of nodes in the system increases according to the following relation:

$$GRD_{INT} = (-1.49 \times 10^{-3} N) + 1.01 \tag{5.5}$$

The intercept for Equation 5.3 results in $GRD_{INT} = 1$, when $N = 8$ to ensure that $k_{GRD} = 1$ for our baseline 8-node ring topology. By substituting the relations given in Equations 5.1 to 5.5 back into our original framework given in Equation 4.1, we obtain our final model for predicting the frequency of any regular or irregular topology. On a Xilinx FPGA, the accuracy of our model will be verified in Section 6.1.

5.2 Altera Analytical Model

In Chapter 4, the analytical model is shown to be extendable to all FPGAs. However, due to Altera FPGA's unique routing fabric, the coefficients of the model need to be recalibrated to account for these differences. The corresponding F_{base} for an 8-node ring topology with a 32-bit link width on a Stratix III FPGA is 210MHz. The coefficients for k_{GRD} and k_{LRD} are derived in this section along with a comparison with the Xilinx analytical model.

5.2.1 Local Routing Demand

As previously described, local routing demand, k_{LRD}, is characterized by the routing requirements of a single network node. The parameters that affect local routing demand are the average node degree (AND) and link width (LW). While the expression for k_{LRD} for Altera FPGAs is the same as for Xilinx FPGAs, the coefficients for the equation need to be recalibrated to reflect the different routing fabric and CAD tools. The expression for k_{LRD} is shown again here:

$$k_{LRD} = LRD_{SL} \times \Delta LW + 1 \tag{5.6}$$

Figure 5.3 shows a family of lines representing the performance of all topologies as a percentage of F_{base} with respect to changing the link width, where each line in the figure represents topologies with a fixed average node degree. The horizontal axis indicates the change in link width from 32-bits, and the vertical axis is the performance change compared to an 8-node ring topology.

In order to determine the expression for LRD_{SL}, the slope of each line in Figure 5.3 is isolated and found to change linearly as the average node degree changed. The expression for LRD_{SL} is identical to the Xilinx model, but has different slope and intercept coefficients. The expression for LRD_{SL} is shown below:

$$LRD_{SL} = (-1.39 \times 10^{-3} AND) + (1.62 \times 10^{-3}) \tag{5.7}$$

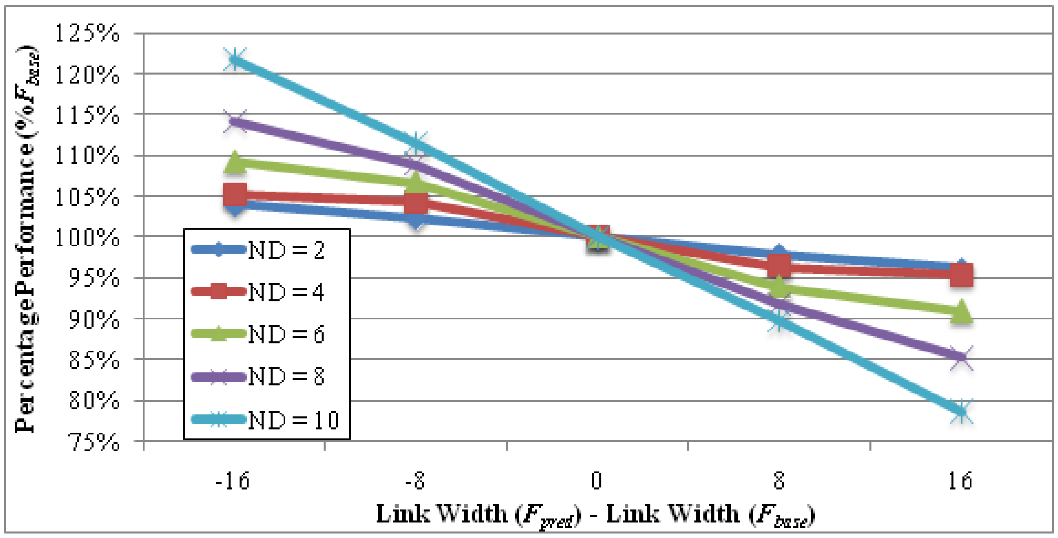

Figure 5.3: Performance variation due to a change in link width for Altera FPGAs

Compared to the Xilinx model, LRD_{SL} increases in magnitude faster for Altera FPGAs than Xilinx FPGAs. Therefore, as the average node degree increases, the performance variation due to a change in link width increases in magnitude at a greater rate than Xilinx FPGAs. These differences are due to a combination of the unique routing fabrics and CAD tool flows.

5.2.2 Global Routing Demand

Recall that global routing demand, k_{GRD}, is characterized by the routing requirements of the entire NoC. The parameters that affect global routing are the average node degree (AND) and the number of nodes (N). Much like for local routing demand, the expression for k_{GRD} for Xilinx FPGAs is also used for Altera FPGAs and only requires that the coefficients be recalibrated to account for different routing fabric and CAD tools. The equation for k_{GRD} is shown below:

$$k_{GRD} = GRD_{SL} \times (AND - 2) + GRD_{INT} \tag{5.8}$$

The linear equation is characterized by the slope, GRD_{SL}, and the intercept GRD_{INT}.

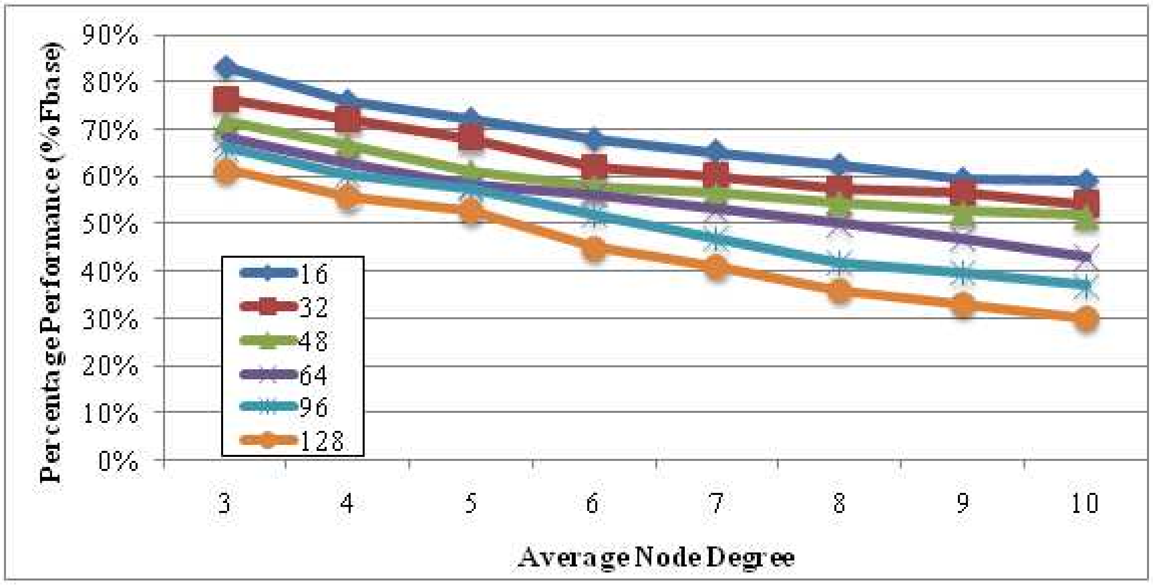

Figure 5.4: Performance loss due to average node degree for Altera FPGAs

In order to determine GRD_{SL} and GRD_{INT}, we analyze the effects of changing the average node degree for a variety of systems with a fixed number of nodes. Figure 5.4 shows a family of lines, with each line representing NoCs with a different fixed number of nodes. The horizontal axis is the average node degree, and the vertical axis represents the performance loss compared to an 8-node ring topology.

When moving between different lines, the slope GRD_{SL}, and the intercept GRD_{INT}, changes as the number of nodes change for a given NoC. The analytical model is the same as Xilinx FPGAs, thus the expression's form is the same, but the coefficients are different as Altera FPGAs exhibit slightly different changes between each line in Figure 5.4. The expressions for GRD_{SL} and GRD_{INT} are shown below:

$$GRD_{SL} = (-2.22 \times 10^{-6} N^2) - (1.47 \times 10^{-5} N) - (4.29 \times 10^{-2}) \tag{5.9}$$

$$GRD_{INT} = (-1.39 \times 10^{-3} N) + 1.01 \tag{5.10}$$

The slope and intercept expressions have different coefficients when compared to the

Xilinx model. Looking at GRD_{SL}, the Altera model exhibits a N^2 and N terms that are smaller in magnitude than the Xilinx model. This means that the rate of performance loss due to average node degree, does not change as much as the Xilinx model when the number of nodes increases in the system. This can be seen when comparing Figure 5.4 with Figure 5.2, as each line becomes considerably "steeper" for the Xilinx model when the number of nodes increases in Figure 5.2 compared to Figure 5.4. Furthermore, the GRD_{INT} expression for the Altera model has a slope that is slightly smaller in magnitude compared to the Xilinx model. Therefore, for an average node degree of two, when the number of nodes in an NoC increases to more than eight nodes, Xilinx FPGAs will exhibit a greater decrease in performance than Altera FPGAs.

As seen in the previous sections, performance on Altera III 340 generally have higher maximum operating frequency than the Xilinx Virtex 5 LX330. When comparing the Xilinx Virtex 5 LX330 and Altera Stratix III 340, the baseline 8-node ring topology with a 32-bit link width has a base frequency F_{base} of 190MHz and 210MHz respectively. The performance improvement on Altera FPGAs can be attributed to a combination of different CAD tool algorithms or the unique routing fabric. Since the architecture and algorithm are proprietary for both vendors, a direct comparison is not done here.

5.3 Summary

While the analytical models developed for Xilinx and Altera FPGAs have identical forms, the coefficients of our model are different for each FPGA vendor. This account for the performance fluctuations that are impacted by the different FPGA routing fabric and CAD tools of each vendor. Furthermore, differences in CAD tool suites can cause performance fluctuations as each suite can utilize different algorithms that may better leverage available resources. The derived analytical model is summarized and repeated here, along with the corresponding F_{base} for each family used in our training experiments in Table 5.1.

Table 5.1: Base frequencies for training experiments

Xilinx FPGAs		**Altera FPGAs**	
Family	F_{base}	Family	F_{base}
Virtex 2 Pro	110MHz		
Virtex 4	145MHz		
Virtex 5	190MHz	Stratix III	210MHz

Analytical Framework:

$$F_{pred} = k_{GRD} \times k_{LRD} \times F_{base} \tag{5.11}$$

Framework for effects due to Local and Global Routing Demand:

$$k_{LRD} = LRD_{SL} \times \Delta LW + 1 \tag{5.12}$$

$$k_{GRD} = GRD_{SL} \times (AND - 2) + GRD_{INT} \tag{5.13}$$

Effects due to Local and Global Routing demand for Xilinx FPGAs:

$$LRD_{SL} = (-1.21 \times 10^{-3} AND) - (4.62 \times 10^{-3}) \tag{5.14}$$

$$GRD_{SL} = (-2.48 \times 10^{-6} N^2) - (2.61 \times 10^{-4} N) - (3.58 \times 10^{-2}) \tag{5.15}$$

$$GRD_{INT} = (-1.49 \times 10^{-3} N) + 1.01 \tag{5.16}$$

Effects due to Local and Global Routing demand for Altera FPGAs:

$$LRD_{SL} = (-1.39 \times 10^{-3} AND) + (1.62 \times 10^{-3}) \tag{5.17}$$

$$GRD_{SL} = (-2.22 \times 10^{-6} N^2) - (1.47 \times 10^{-5} N) - (4.29 \times 10^{-2}) \tag{5.18}$$

$$GRD_{INT} = (-1.39 \times 10^{-3} N) + 1.01 \tag{5.19}$$

As previously stated, the expressions of our analytical model shown in Equations 5.11-5.19 are analytically derived. In order to tune our model to Xilinx and Altera FPGAs, we used empirical curve-fitting to determine the coefficients given in Equations 5.14- 5.19. In the following chapter, the expressions developed in this chapter are verified utilizing more modern Xilinx Virtex 6 FPGAs and Altera Stratix IV FPGAs. This will ensure that our model is extendable to the newest FPGAs from both FPGA vendors. Furthermore, an analysis on error will be performed to ensure that our model is capable of characterizing a majority of NoC topologies.

Chapter 6

Verification

Our analytical model provides an approximation of performance given the global and local routing demands of an NoC topology. Utilizing approximately 3600 experiments per device on both Altera and Xilinx FPGAs, we tune this analytical model, which is capable of predicting performance for a range of FPGA device families. In order to verify our model, we create 750 new benchmark circuits and synthesize them for the most recent Xilinx and Altera FPGAs. These new benchmark circuits consist of heterogeneous and homogeneous systems with 8 to 128 nodes using random topologies with average node degree from 2-10 and link widths of 16, 24, 32, 40, or 48-bits. In this section, we verify our analytical model and perform an error analysis to demonstrate where our model fails.

6.1 Xilinx Verification

To measure the accuracy of our model, we map the new benchmark circuits for Virtex 4, Virtex 5, and Virtex 6 FPGAs, resulting in 2250 data points. Since the Virtex 6 FPGA's recent release is not supported in Xilinx EDK 10.1.02, we use Xilinx EDK 11.2 to run our verification experiments. In order to see the improvements of new CAD tools and technologies, we ran several preliminary experiments on the Virtex 5 LX330 in EDK 11.2. The results show that the actual performance increased on average by 8.4% with a standard

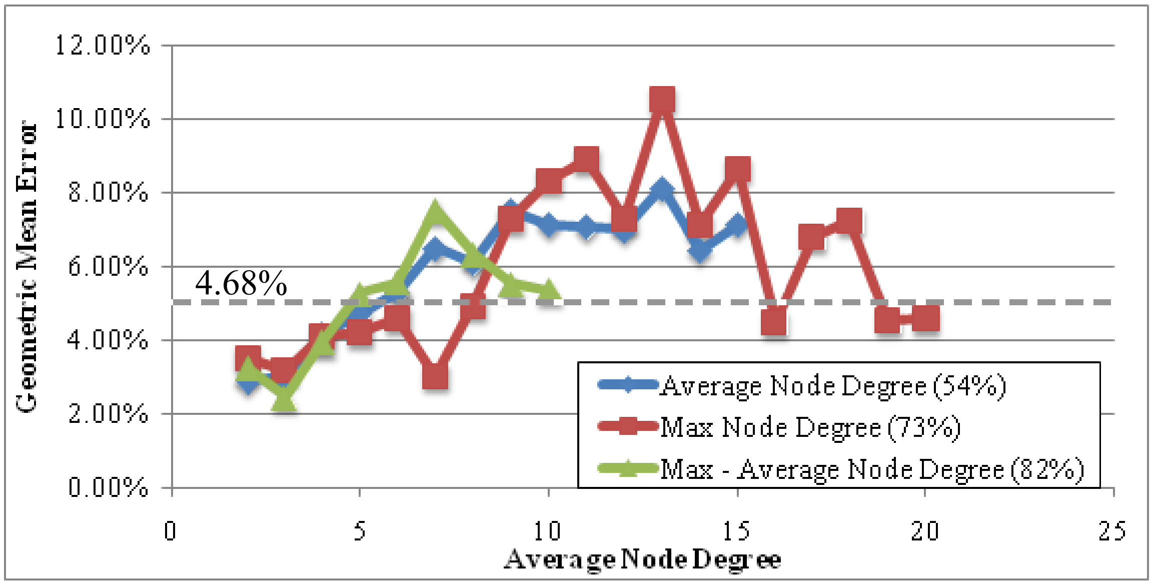

Figure 6.1: Geometric mean error as a function of node degree for Xilinx FPGAs

deviation of 3.2%. However, when normalized, the new CAD flow's performance only varied by 1.8% with a standard deviation of 1.4%.

Table **??** shows a sample set of the operating frequencies predicted by our model. The properties of each topology are shown by the number of nodes in Column 1, the average node degree in Column 2, and the link width in Column 3. Column 4 shows the predicted frequency, Column 5 presents the actual frequencies obtained from the CAD tools and Column 6 shows the geometric mean error for each NoC topology. Finally, Column 7 indicates the resource usage of each topology as a percentage of total available resources. The devices shown in Tables 6.1 and 6.2 have approximately the same number of LUTs. We use the geometric mean error as it weights the error's magnitude depending on the maximum operating frequencies. For example, an error of 5% between the actual and predicted frequencies has a greater significance for topologies with an actual frequency of 200MHz vs 50MHz. The new F_{base} for each device generated with EDK 11.2 is also listed in Tables 6.1 and 6.2. The overall error was found to be 4.68% with a standard deviation of 3.41%.

Table 6.1: Predicted operating frequencies for Xilinx Virtex 4 and Virtex 5 FPGAs

Virtex 4LX200		Base Frequency = 150MHz				
# of Nodes	*NodeDegree*	*Width*	*Predicted(MHz)*	*Actual(MHz)*	*Error*	*% RU*
16	5	48	105.1	110.1	4.58%	9%
16	8	24	116.8	113.6	2.73%	11%
32	3	16	154.5	150.4	2.68%	16%
32	6	32	112.5	114.7	1.94%	23%
32	8	16	107.4	111.5	3.61%	21%
48	3	16	149.4	143.6	4.01%	22%
48	5	24	123.9	116.7	6.16%	26%
48	8	32	87.4	88.2	0.89%	32%
64	5	32	107.5	103.5	3.86%	44%
64	6	24	106.8	97.5	9.48%	40%
64	7	40	78.5	80.4	2.37%	45%
96	4	24	113.4	108.7	4.28%	55%
96	8	48	43.3	47.1	8.01%	76%
96	9	16	54.4	52.8	18.8%	59%
128	3	32	107.8	92.9	15.9%	79%
128	5	24	83.9	84.9	1.21%	76%
128	6	40	56.3	53.1	5.91%	87%
Virtex 5LX220		Base Frequency = 200MHz				
# of Nodes	*NodeDegree*	*Width*	*Predicted(MHz)*	*Actual(MHz)*	*Error*	*% RU*
16	7	32	149.4	147.9	1.08%	10%
16	9	24	146.2	150.7	2.97%	9%
32	5	16	187.9	178.3	5.44%	16%
32	7	32	139.3	136.4	2.10%	20%
32	9	40	103.4	99.4	4.03%	28%
48	5	24	165.2	156.7	5.41%	25%
48	7	32	128.4	119.5	7.49%	31%
48	7	24	141.8	134.7	5.32%	26%
64	3	48	203.5	201.1	1.23%	33%
64	5	16	167.7	166.5	0.73%	32%
64	8	40	91.2	93.1	1.46%	44%
96	2	24	183.3	190.1	3.58%	41%
96	3	32	157.1	148.9	5.52%	54%
96	5	40	113.6	110.4	2.93%	59%
128	4	48	104.9	110.1	4.77%	68%
128	6	32	82.8	73.2	13.1%	75%
128	8	16	51.9	45.7	13.5%	79%

Table 6.2: Predicted operating frequencies for Xilinx Virtex 6 FPGA

Virtex 6LX240T		Base Frequency = 240MHz				
# of Nodes	*NodeDegree*	*Width*	*Predicted(MHz)*	*Actual(MHz)*	*Error*	*% RU*
16	3	32	225.6	232.2	2.85%	8%
16	7	24	197.9	196.5	0.76%	8%
32	3	16	247.2	256.9	3.78%	13%
32	4	24	221.2	223.3	0.98%	14%
32	6	40	163.1	162.2	0.58%	12%
48	3	32	211.3	199.4	5.97%	22%
48	5	24	198.3	189.5	4.58%	22%
48	7	16	186.2	182.2	2.20%	22%
64	2	32	219.8	218.1	0.82%	21%
64	3	48	177.2	180.4	1.80%	30%
64	7	24	154.8	160.9	3.80%	31%
96	3	16	213.3	215.3	0.95%	40%
96	4	40	156.1	153.1	1.94%	48%
96	9	24	78.5	72.5	8.19%	48%
128	4	32	148.1	125.8	17.8%	75%
128	5	40	113.3	115.1	1.61%	77%
128	6	16	118.2	92.5	27.9%	56%
Geometric Mean Error for all Virtex Experiments					**4.68%**	

As seen in Tables 6.1 and 6.2, our equation results in large errors for some topologies (see highlighted entries). To determine the relationship between error and NoC characteristics, we analyze the relationship between error and the values of *N*, *LW*, and *AND*. In Figure 6.1, we plot the geometric mean error as a function of average node degree, maximum node degree, and (max node degree - average node degree). We choose these parameters to see how our model encapsulates application-specific topologies. If a single node has much higher node degree, than the max node degree is much greater than the average node degree. Thus, local routing demand should have a large effect on performance as seen for the star topology. We also analyzed the variation in error in relation to *N* and *LW*. For these cases, the error remained small (1.3%) and never exceeded 7%, thus we do not show them here. The dotted line in Figure 6.1 shows the geometric mean error of all our benchmarks. While numerous data points lie above this line, the majority of our benchmarks resulted in points below this line. The percentage of systems below this line is given in the legend.

From Figure 6.1, we can see that the error increases minimally and remains less than 11% for all three cases, indicating that error is not directly correlated with the node degree as there are no significant trends. Thus our analytical model is still capable of accurately predicting application-specific topologies where max node degree may be greater than average node degree. However, not captured in this analysis is the star topology, which represents an extreme case when the maximum node degree is much larger than the average node degree. For the star topology, our predictor exceeds 25% error when the number of nodes is larger than 16. Addressing this is a topic of future work.

In Table **??**, all benchmarks in which the error was above 10% had a resource usage above 70%. Our current model does not include the resource usage as an input parameter. We plot our set of benchmark circuits by resource usage, and calculate the average error for each bin. The results are shown in Figure 6.2. For benchmarks with less than 65% resource usage, the geometric mean error is less than 10%; however above this point, the geometric mean error increases dramatically. These results suggest that to improve the accuracy of our equation, resource usage needs to be considered. Since our objective is

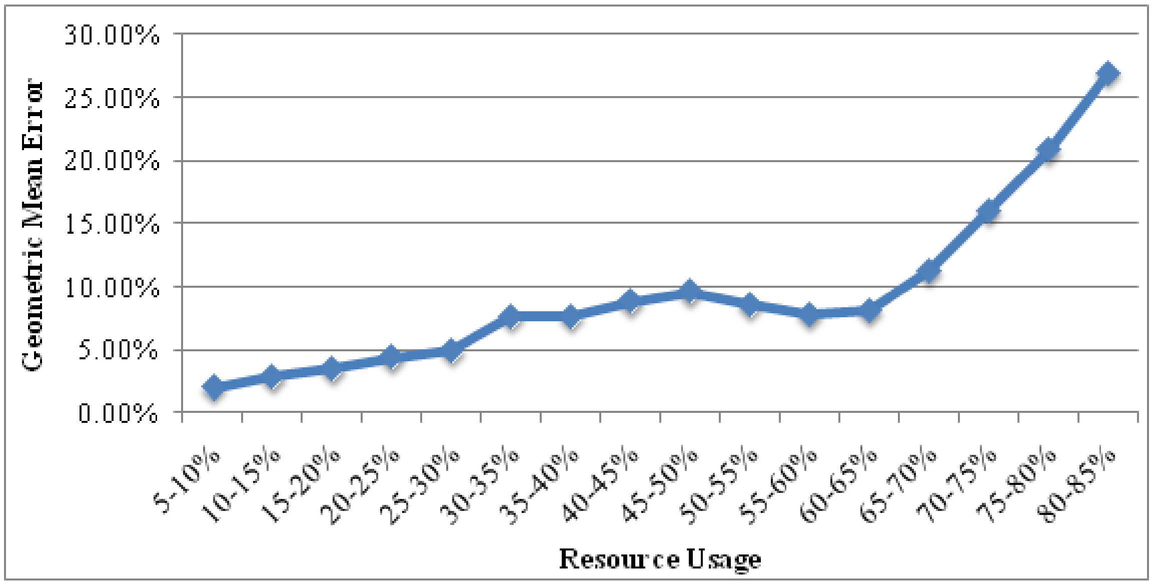

Figure 6.2: Geometric mean error as a function of resource usage for Xilinx FPGAs

to provide designers with a means of early design space exploration, including resource usage would not meet our objectives as it would require the designer to fully map the NoC. Therefore, provided the NoC has sufficient routing resources on a respective FPGA, our equation provides an accurate method of predicting the maximum frequency with no design time. The same verification experiments and error analysis are repeated in the next section for Altera FPGAs.

6.2 Altera Verification

We used the same methodology to verify our Altera tuned analytical model as we did for the Xilinx model. We verify our analytical model with the Stratix IV family of FPGAs and utilize the same benchmark circuits as the previous section. We use Stratix III and Stratix IV FPGAs in our verification experiments. Table 6.3 illustrates several benchmark circuits that are used to verify our model, and compares the actual frequency obtained from the Altera tool flow with the predicted frequency from our Altera analytical model. The properties of each topology are shown by the number of nodes in Column 1, the average

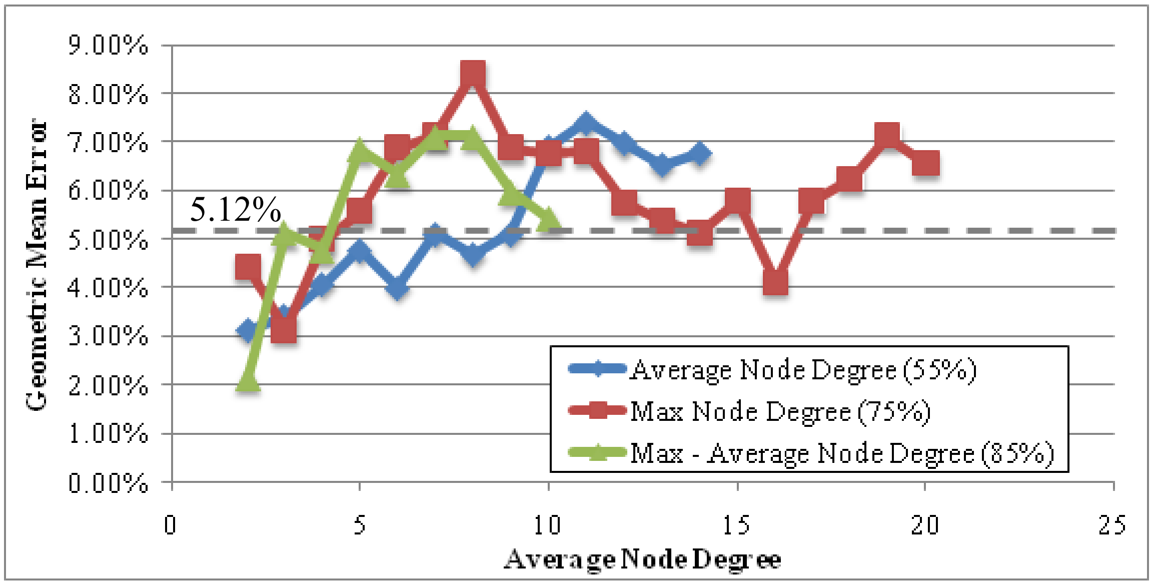

Figure 6.3: Geometric mean error as a function of node degree for Altera FPGAs

node degree in Column 2, and the link width in Column 3. Column 4 shows the predicted frequency, Column 5 presents the actual frequencies obtained from the CAD tools and Column 6 shows the geometric mean error for each NoC topology.

Each benchmark circuit is characterized by the number of nodes, average node degree, and link width. For our analytical model, the corresponding F_{base} for each family is 250MHz for Stratix IV FPGAs, and 210MHz for Stratix III FPGAs. The error is given for each topology in Table 6.3, and the geometric mean error of all verification experiments is found to be 5.12% with a standard deviation of 2.14%. While the error is approximately the same as the Xilinx model, the standard deviation is significantly less. Therefore, the variation in error for our Altera model is less than the Xilinx model. This can be seen in our Altera results through Chapter 4 and 5. Much of our results for Altera FPGAs exhibit more consistent trends with less unpredictability in performance.

In order to determine the cause of error from our tuned Altera analytical model, we analyze error in relation to N, AND, LW and resource usage. Similar to our Xilinx analytical model, the Altera analytical model did not show any significant relation between error and

Table 6.3: Predicted operating frequencies for Altera FPGAs

Stratix EP3SL200		Base Frequency = 210MHz				
# of Nodes	*NodeDegree*	*Width*	*Predicted(MHz)*	*Actual(MHz)*	*Error*	*% RU*
16	3	24	202.7	214.2	5.65%	9%
16	6	32	171.5	173.9	1.39%	10%
32	4	32	184.7	196.9	6.56%	%
32	7	48	136.8	130.4	4.70%	20%
32	9	16	163.9	165.9	1.26%	19%
48	2	32	198.2	202.3	2.03%	24%
48	5	24	178.2	181.8	2.01%	26%
48	6	16	179.2	177.2	1.14%	25%
64	2	24	195.4	209.5	7.20%	24%
64	5	32	165.8	161.9	2.35%	38%
64	7	40	137.7	128.4	6.73%	40%
96	5	16	169.1	185.5	9.66%	51%
96	6	40	138.2	135.1	2.24%	57%
96	10	24	118.9	134.4	13.0%	55%
128	3	32	164.9	171.2	3.86%	75%
128	6	32	135.4	118.4	12.6%	77%
128	8	24	124.6	118.9	4.59%	72%
Stratix EP4SE230		Base Frequency = 250MHz				
# of Nodes	*NodeDegree*	*Width*	*Predicted(MHz)*	*Actual(MHz)*	*Error*	*% RU*
16	4	16	240.1	223.1	7.08%	7%
16	7	32	139.3	182.8	5.47%	8%
32	2	24	243.9	258.4	5.93%	12%
32	8	16	203.8	186.8	8.36%	14%
32	10	40	139.7	140.8	0.79%	16%
48	3	24	229.8	222.7	3.08%	20%
48	7	32	181.5	172.7	4.86%	24%
48	9	16	187.9	173.2	7.82%	21%
64	4	48	195.1	201.1	3.12%	32%
64	6	24	196.5	181.9	7.47%	29%
64	8	32	164.4	153.9	6.41%	32%
96	3	16	216.6	222.1	2.55%	42%
96	7	24	173.4	180.2	3.92%	44%
96	10	24	141.6	140.9	0.49%	44%
128	3	16	204.5	210.2	2.79%	56%
128	5	32	172.9	157.8	8.76%	64%
128	8	40	127.2	131.1	3.08%	68%
Geometric Mean Error for all Stratix Experiments					**5.12%**	

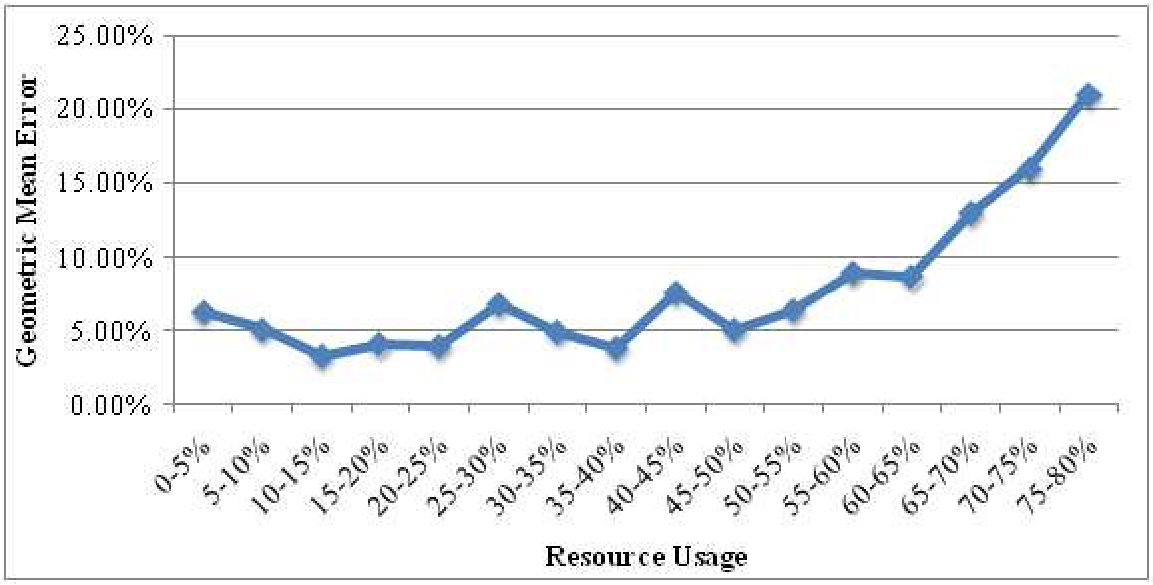

Figure 6.4: Geometric mean error as a function of resource usage for Altera FPGAs

N, *AND*, and *LW* as shown in Figure 6.3. For all three cases, error remains relatively constant and is always below 9.0%. However, when analyzing error in relation to resource usage, we found the same relation we found in the previous section. Figure 6.4 illustrates this relation.

In Figure 6.4, the error remains relatively constant up till 55% resource usage. Above this point, error increases dramatically up till 75-80% when systems start to fail routing. Much like our Xilinx model, this suggests that as designs become more congested, the CAD tools are unable to leverage available resources. This leads to systems with significantly lower performance than would be expected from our analytical model creating an exponential increase in error.

6.3 PlanAhead - Manual Placement of NoC Topologies

Our analytical model is used to provide a means of design space exploration without actual implementation on an FPGA. This allows a designer to choose an appropriate NoC topology for their application from a very large design space. However, the Xilinx model is derived using placement and routing tools by Xilinx. Xilinx's EDK and ISE CAD tool suites provide an automated synthesis tool that facilitates a simple design process. For NoC implementations, the individual blocks and connectivity need only to be defined for the tool suite to successfully implement the design.

While our analytical model provides an approximation of performance, we further look at the possibility of optimizing a design using guided placement. Since our model relies on the performance obtained using automated CAD tools, it is also possible to manually place nodes to see if a better result can be obtained. Since many NoC topologies can be flattened to two dimensional layouts, such as the ring, torus, and mesh topologies, the placement of individual network nodes should map well onto the two dimensional fabric of an FPGA. We look at manually placing these topologies using relatively placed modules (RPMs) to define the location of individual network nodes. We then see the impact on performance and if performance can be improved using these RPMs.

In order to facilitate these experiments, we use Xilinx PlanAhead 10.1, which allows a designer to define the location of individual blocks or nets in a design. First, a topology is run through the ISE tool flow, which iterates placement and routing ten times using different effort levels that are known to have the best results. Once this is complete, a manual placement of the topology is determined and the design is synthesized using Xilinx ExploreAhead, which re-iterates the CAD flow 10 times with different effort levels to try to optimize the maximum operating frequency. While manual placement can often achieve better results, the effort required often leads to extremely long design times as designers try to re-iterate manual designs to find the best solution. Therefore in our analysis, we only attempt an initial placement based on the layout of a two-dimensional NoC topology. Each node is assigned a location on the FPGA fabric, with each location having at least 10% more resources than is required by each node.

In order to "simulate" a congested design, we use designs that utilize approximately 80-90% resource usage when resource usage starts to have an affect on performance. Therefore to get this scenario, we used large designs on smaller FPGA chips. We chose 64-node ring,

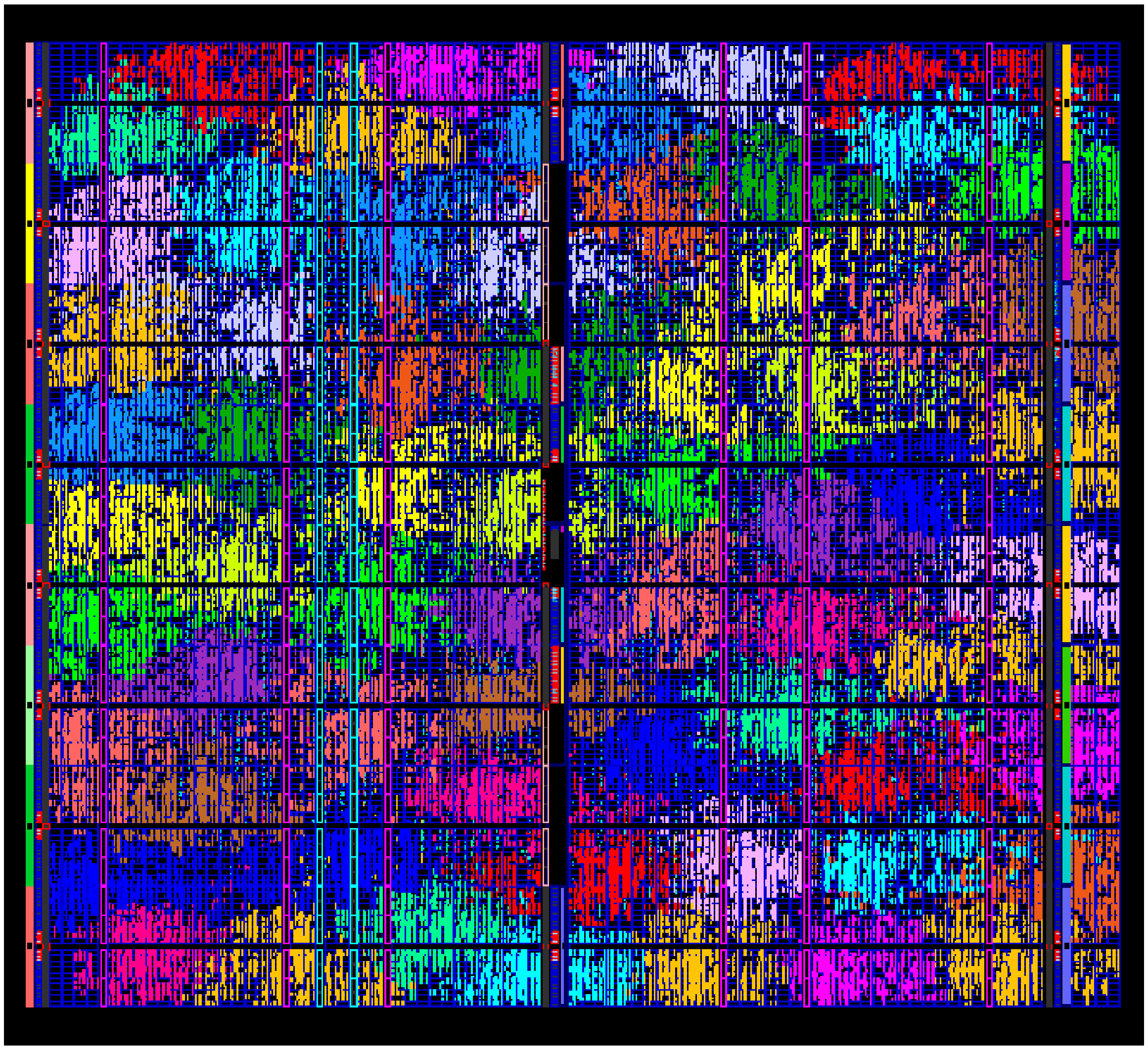

Figure 6.5: Example automatic placement of homogeneous 64-node mesh topology

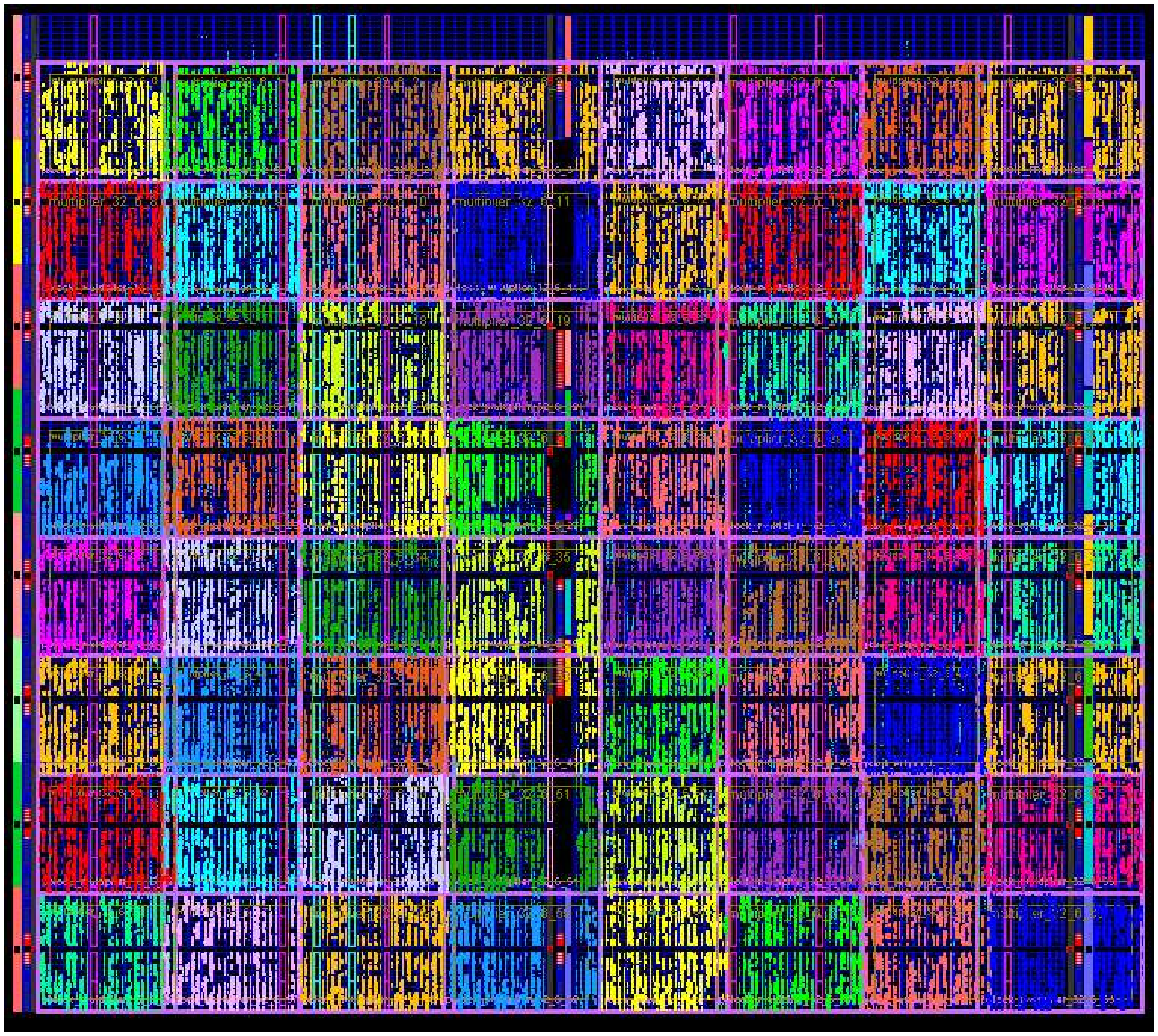

Figure 6.6: Example manual placement of homogeneous 64-node mesh topology

Table 6.4: Homogeneous MultBase topologies with manual placement

	Ring		Torus		Mesh	
Link Width	Auto	RPM	Auto	RPM	Auto	RPM
32	198MHz	214MHz	176MHz	187MHz	184MHz	200MHz
24	201MHz	220MHz	184MHz	191MHz	197MHz	215MHz
16	215MHz	232MHz	192MHz	200MHz	204MHz	229MHz
Avg % Improvement	8.48%		4.74%		10.1%	

Table 6.5: Heterogeneous MultBase topologies with manual placement

	Ring		Torus		Mesh	
Link Width	Auto	RPM	Auto	RPM	Auto	RPM
32	198MHz	208MHz	176MHz	181MHz	184MHz	200MHz
24	201MHz	207MHz	184MHz	190MHz	197MHz	205MHz
16	215MHz	221MHz	192MHz	194MHz	204MHz	214MHz
Avg % Improvement	3.61%		2.38%		5.89%	

torus, and mesh topologies implemented on a Virtex 5 LX150 for a variety of heterogeneous and homogeneous node types with 16, 24, and 32-bit link widths. Figure 6.5 shows an example automatic placement of a 64-node mesh topology with MultBase nodes and 32-bit link width. Using the same topology, we manually placed the design using the same FPGA and is shown in Figure 6.6.

Analyzing the performance of the two systems, we find that using relatively placed modules improves performance by 8.7%. This suggests that it is possible to improve the performance of certain topologies by using RPMs. Table 6.4 illustrates the performances for the ring, torus, and mesh topologies using the same methodology for automatic and manual placement as previously described.

The results in Table 6.4 show that the mesh topology exhibits the greatest improvement in performance of 12.6% when using RPMs as its structure maps extremely well to an island style FPGA. Other topologies also show performance improvements of 8.48% for the ring topology and 4.74% for the torus topology. The torus topology exhibits the smallest improvement due to the longer wires required to route a folded torus. We repeat the same experiments for heterogeneous systems, and list the results in Table 6.5.

As seen in Table 6.5, the performance improvement for manually placed heterogeneous

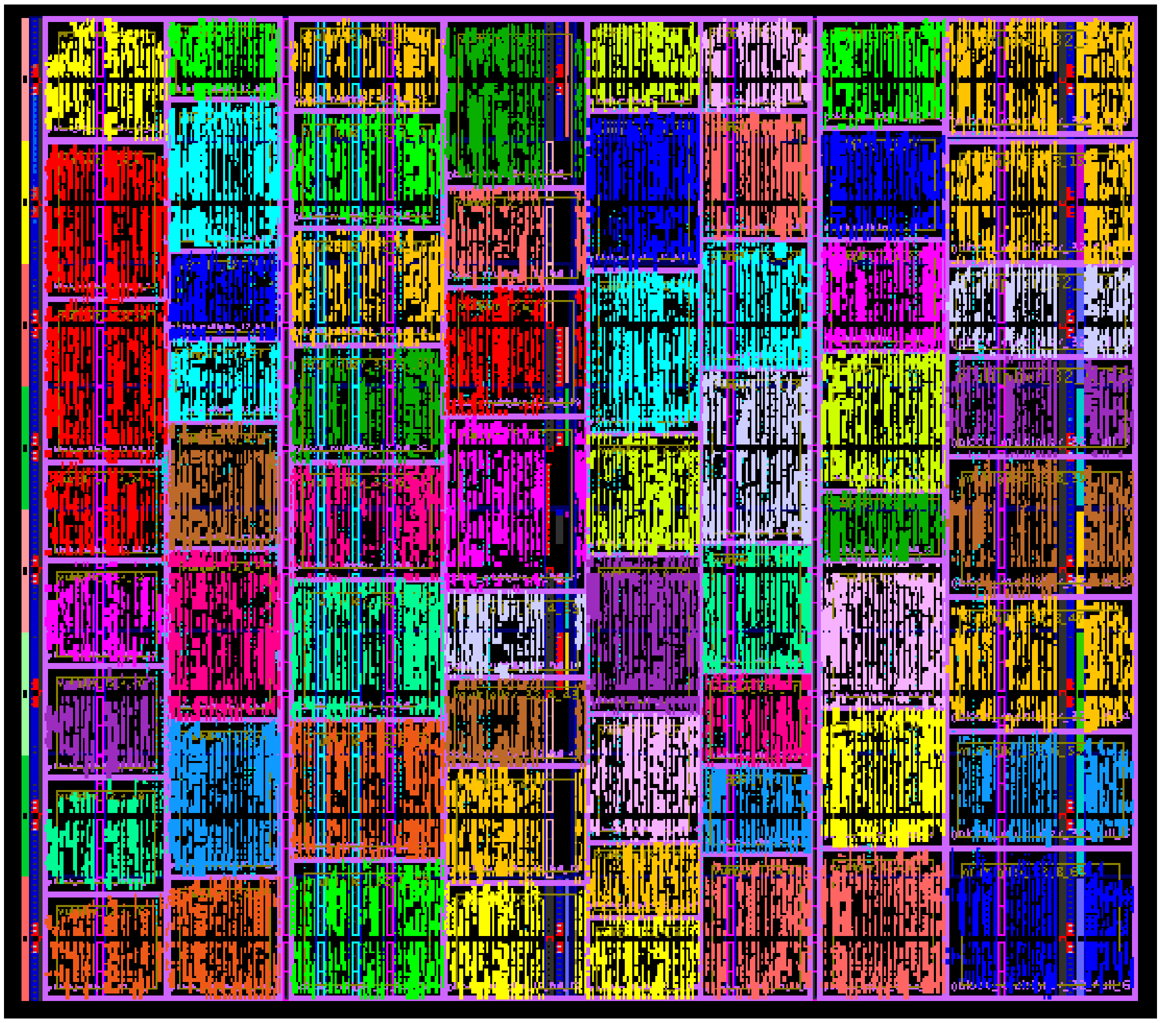
Figure 6.7: Example manual placement of heterogeneous 64-node mesh topology

topologies is less dramatic than for homogeneous topologies. However, all topologies do experience a small performance improvement. This is due to the fact that heterogeneous topologies are much more difficult to place manually due to the different node sizes. This can be seen in Figure 6.7, which shows a heterogeneous 64-node mesh topology with 32-bit link widths. Since each node requires different resource utilizations, part of the inherent two dimensionality of the mesh topology is lost. Thus, heterogeneous topologies do not exhibit as great an increase in performance as their homogeneous counterparts. The analysis shown here demonstrates that using relatively placed modules can improve performance by up to 12.6%. Therefore, when used in conjunction with our analytical model, the designer can take into consideration the possibility of using manual placement to further improve performance.

Chapter 7

Conclusion

In this work, our analysis demonstrates a concrete understanding of how NoCs perform on FPGAs. Performance is primarily dictated by the network connectivity so long as the network interface masks the performance of the network node. Therefore, by doing so, the network node only effects performance in so much as increasing the overall resource usage causing global resources to be stressed. Furthermore, our work shows that regularity has little effect on performance and irregular topologies modeled using application-specific topologies have the same performance characteristics as regular topologies.

Using these conclusions, we present an analytical model in the form of a simple equation that describes the maximum operating frequency (performance) of an NoC as a function of various network parameters related to the overall network topology. The predicted frequency is a function of the global and local routing demand of the topology, which in turn is affected by the number of nodes, average node degree and link width of the system. In order to characterize these parameters, random topologies are used to emulate application-specific NoCs. Although random topologies are used, our research shows that common topologies such as the torus and mesh topology exhibit almost identical performance results as random topologies with the same network parameters.

The analytical model is shown to be applicable to Xilinx Virtex 2 Pro, Virtex 4, Virtex 5, and Virtex 6 FPGAs along with Altera Stratix III and Stratix IV FPGAs. For our

Xilinx analytical model, the equation was shown to be accurate to within 4.68%, and for our Altera analytical model, within 5.12%. This model provides a measure of the effect of varying different topology parameters on NoC performance on FPGAs. Furthermore, it provides guidance to a designer during early design space exploration when a suitable network topology is being chosen.

A key observation from this work is that modern FPGAs contain enough routing to implement fairly complex NoCs. This opens the door to new system architectures based on application-specific NoCs rather than the more restricted mesh topologies that are typically used in ASIC SoC implementations. These application-specific NoCs can be tailored to the problem at hand, leading to an overall improvement in system-level performance measures. While our model is derived utilizing Altera and Xilinx's automatic CAD tool flow, further research utilizing Xilinx PlanAhead shows that with minimal effort, manual placement can further improve NoC performance. By taking advantage of topologies that are inherently two dimensional such as the ring, mesh, and torus topologies, manually placing network nodes in a defined pattern shows performance increases of up to 12.6%.

7.1 Future Work

The work presented in this thesis focuses on the derivation of a model capable of predicting the performance of an NoC using analytical modelling and empirical curve-fitting. The results presented herein suggest a number of possible research topics for future work. Firstly, our research focuses primarily on analyzing existing FPGA fabrics, and not on FPGA architectural changes that could benefit NoCs. However, we have begun to run experiments using versatile place and route (VPR) [42], a place and route tool developed at the University of Toronto, to determine if small architectural changes improve the performance of NoCs. However, the results obtained are not included in this thesis as they do not accurately model modern FPGAs. Modern FPGAs, such as the Virtex 6 and Stratix IV, have a hierarchical architecture that cannot be replicated in VPR. Furthermore, timing analysis utilizing VPR is still relatively new, and accurate results could not be obtained. With future releases of

VPR, this analysis may be possible; however, the current results are not included in this thesis due to these drawbacks.

There can also be significantly more work done in analyzing manual placement using RPMs. While topologies that exhibit two dimensional structures are analyzed using RPMs, other topologies such as hypercube, star, and application-specific topologies are not investigated. Due to their increased complexity, these topologies may not benefit from manual placement as the CAD tools may be more suited at finding an optimal placement. It would also be interesting to see why manual placement results in performance improvements; however, this would require a more in depth analysis of how the routing channels are used in the automatic placement of NoC topologies.

Another area of future work would be improving the accuracy of the analytical model developed in this thesis. Our model results in significant error whenever resource usage exceeds 80%. In order to alleviate this, resource usage needs to be considered in a more accurate model. This can be done by either fully mapping the design and inputting the resource usage or estimating the total resource usage based on the resource utilization of individual network nodes and network switches. Furthermore, our model does not encapsulate the star topology when the max node degree is much greater than the average node degree. Therefore, a future model should incorporate max node degree to account for the star topology into consideration. While the maximum operating frequency of NoC topologies is the focus in this thesis, other NoC performance metrics are of significant interest. Future models should consider bandwidth and throughput, which are very important performance parameters for NoC designers.

Bibliography

[1] M. Abramovici, C. Stroud, and M. Emmert. Using embedded FPGAs for SoC yield improvement. In *Proceedings of the 39th conference on Design automation*, pages 713–724. ACM New York, NY, USA, 2002.

[2] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and CA Zeferino. SPIN: a scalable, packet switched, on-chip micro-network. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 70–73, 2003.

[3] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S.P. Fekete, and J. van der Veen. A practical approach for circuit routing on dynamic reconfigurable devices. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, Montreal, Canada*, pages 84–90. Citeseer, 2005.

[4] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo. Contrasting a NoC and a traditional interconnect fabric with layout awareness. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, page 129. European Design and Automation Association, 2006.

[5] TA Bartic, JY Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *IEE Proceedings-Computers and Digital Techniques*, 152(4):467–472, 2005.

[6] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[7] D. Bertozzi and L. Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31, 2004.

[8] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, pages 113–129, 2005.

[9] G. Brebner and D. Levi. Networking on chip with platform FPGAs. In *2003 IEEE International Conference on Field-Programmable Technology (FPT), 2003. Proceedings*, pages 13–20, 2003.

[10] J. Chan and S. Parameswaran. NoCOUT: NoC topology generation with mixed packet-switched and point-to-point networks. In *Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 265–270. IEEE Computer Society Press Los Alamitos, CA, USA, 2008.

[11] Altera Corp. Stratix II Family Device Overview. 2007.

[12] Altera Corp. Stratix III Family Device Overview, 2009.

[13] Altera Corp. Stratix IV Family Device Overview, 2009.

[14] W.J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689, 2001.

[15] R. Francis, S. Moore, and R. Mullins. A Network of Time-Division Multiplexed Wiring for FPGAs. In *Proceedings of the Second ACM/IEEE International Symposium on NEtworks-on-Chip*, pages 45–44. IEEE Computer Society Washington, DC, USA, 2008.

[16] R.M. Francis and S.W. Moore. FPGAs with time-division multiplexed wiring: an architectural exploration and area analysis. 2009.

[17] H.C. Freitas, D.M. Colombo, F.L. Kastensmidt, and P.O.A. Navaux. Evaluating Network-on-Chip for Homogeneous Embedded Multiprocessors in FPGAs. In *IEEE International Symposium on Circuits and Systems*, pages 3776–3779, 2007.

[18] R. Gindin, I. Cidon, and I. Keidar. NoC-based FPGA: architecture and routing. In *Proceedings of the First International Symposium on Networks-on-Chip*, pages 253–264. IEEE Computer Society Washington, DC, USA, 2007.

[19] K. Goossens, M. Bennebroek, J.Y. Hur, and M.A. Wahlah. Hardwired Networks on Chip in FPGAs to Unify Functional and Configuration Interconnects. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 45–54. IEEE Computer Society Washington, DC, USA, 2008.

[20] J. Greenbaum, C.S. Inc, and CA San Jose. Reconfigurable logic in SoC systems. In *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, pages 5–8, 2002.

[21] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, page 256. ACM, 2000.

[22] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann. Dynamic Reconfiguration with hardwired Networks-on-Chip on future FPGAs. In *Proc. of the 15th Int. Conf. on Field Programmable Logic and Applications (FPL05), Tampere, Finland*, 2005.

[23] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, pages 166–173. Citeseer, 2000.

[24] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *IEE Proceedings-Computers and Digital Techniques*, 153(3):181–188, 2006.

[25] X. Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. *Xilinx Inc*, 2005.

[26] X. Inc. Virtex-5 Family Overview-LX, LXT, and SXT Platforms, 2007.

[27] X. Inc. Virtex-4 Data Sheets: Virtex-4 Family Overview, 2008.

[28] X. Inc. Virtex-6 Family Overview, 2009.

[29] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda. Cunoc: A scalable dynamic noc for dynamically reconfigurable fpgas. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 753–756, 2007.

[30] N. Kapre. *Packet-switched on-chip FPGA overlay networks.* PhD thesis, California Institute of Technology, 2006.

[31] N. Kapre, N. Mehta, M. DeLorimier, R. Rubin, H. Barnor, MJ Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 205–216, 2006.

[32] D. Kim, K. Lee, S. Lee, and H. Yoo. A reconfigurable crossbar switch with adaptive bandwidth control for networks-on-chip. In *IEEE International Symposium on Circuits and Systems*, volume 3, page 2369. IEEE; 1999, 2005.

[33] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens. A modular simulation framework for architectural exploration of on-chip interconnection networks. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 7–12. ACM New York, NY, USA, 2003.

[34] A. Kumar, A. Hansson, J. Huisken, and H. Corporaal. An fpga design flow for reconfigurable network-based multi-processor systems on chip. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6, 2007.

[35] S. Kumar, A. Jantsch, J.P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *IEEE Computer Society Annual Symposium on VLSI*, volume 102, pages 117–124, 2002.

[36] A. Lam, S.J.E. Wilton, P. Leong, and W. Luk. An analytical model describing the relationships between logic architecture and FPGA density. In *Intl Conf. on Field-Programmable Logic and Applications*, 2008.

[37] H.G. Lee, N. Chang, U.Y. Ogras, and R. Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):23, 2007.

[38] J. Lee and L. Shannon. The effect of number of nodes, heterogeneity, and node size on NoCs on FPGAs. In *Proceedings of Sixth Annual IEEE International Symposium on Field Programmable Technologies*, pages 58–64, 2009.

[39] J. Lee and L. Shannon. Predicting the Performance and Routability of FPGA based NoCs. In *Proceedings of Eighteenth Annual IEEE International Symposium on Field Programmable Gate Arrays*, 2010.

[40] ARM Limited. AMBA specification (rev 2.0). *ARM Limited*, 1999.

[41] S. Lukovic and L. Fiorin. An automated design flow for noc-based mpsocs on fpga. In *Rapid System Prototyping, 2008. RSP'08. The 19th IEEE/IFIP International Symposium on*, pages 58–64, 2008.

[42] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W.M. Fang, and J. Rose. VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 133–142. ACM, 2009.

[43] T.S.T. Mak, P. Sedcole, P.Y.K. Cheung, and W. Luk. On-FPGA communication architectures and design factors. In *Proceedings of the FPL*. Citeseer, 2006.

[44] N. Mehta. *Time-multiplexed FPGA overlay networks on chip*. PhD thesis, Citeseer, 2006.

[45] IBM Microelectronics. CoreConnect bus architecture, 1999.

[46] UY Ogras and R. Marculescu. Application-specific network-on-chip architecture customization via long-range link insertion. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, page 253. IEEE Computer Society, 2005.

[47] N. Ohba and K. Takano. An SoC design methodology using FPGAs and embedded microprocessors. In *Proceedings of the 41st annual Conference on Design Automation*, pages 747–752. ACM New York, NY, USA, 2004.

[48] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005.

[49] W.D. Peterson. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. *OpenCores. org*, 2002.

[50] A. Rahman, S. Das, A.P. Chandrakasan, and R. Reif. Wiring requirement and three-dimensional integration technology for field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 11(1):44–54, 2003.

[51] M. Saldaña, L. Shannon, and P. Chow. The routability of multiprocessor network topologies in FPGAs. In *Proceedings of the 2006 international workshop on System-level interconnect prediction*, page 56. ACM, 2006.

[52] E. Salminen, A. Kulmala, and TD Hamalainen. HIBI-based multiprocessor SoC on FPGA. In *IEEE International Symposium on Circuits and Systems, 2005. ISCAS 2005*, pages 3351–3354, 2005.

[53] C.L. Seitz. Let's route packets instead of wires. In *Proceedings of the sixth MIT conference on Advanced research in VLSI table of contents*, pages 133–138. MIT Press Cambridge, MA, USA, 1990.

[54] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri. LiPaR: A light-weight parallel router for FPGA-based networks-on-chip. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 452–457. ACM New York, NY, USA, 2005.

[55] B. Sethuraman and R. Vemuri. optiMap: a tool for automated generation of NoC architectures using multi-port routers for FPGAs. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, page 952. European Design and Automation Association, 2006.

[56] L. Shannon and P. Chow. Simplifying the integration of processing elements in computing systems using a programmable controller. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 63–72, 2005.

[57] A.M. Smith, S.J.E. Wilton, and J. Das. Wirelength modeling for homogeneous and heterogeneous FPGA architectural development. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 181–190. ACM, 2009.

[58] M.B. Stensgaard and J. Sparsø. Renoc: A network-on-chip architecture with reconfigurable topology. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 55–64. IEEE Computer Society, 2008.

[59] C.A. Zeferino, M.E. Kreutz, and A.A. Susin. RASoC: A router soft-core for networks-on-chip. In *Proceedings of the conference on Design, automation and test in Europe-Volume 3*. IEEE Computer Society Washington, DC, USA, 2004.

Appendix A

Topology Generator

As described in Section 3.2, the topology generator is used to generate a topology description file that defines the structure of the NoC. The user is prompted for the NoC type, the number of nodes, and topology specific parameters. The topology description file is used as an input to the system generator to produce all necessary files for synthesis. The script consists of two files: 1) main.c and 2) globals.c. The two files are shown in the following sections.

A.1 Main Function (main.c)

In "main.c", the script generates the appropriate topology description file based on the user's input.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "globals.h"

// Number of topologies used in script
#define P_NUM_TOPOLOGIES                7

// Used to generate a random number
int int_rand(int N){
    return(rand()/(int)(((unsigned)RAND_MAX + 1) / N));
}
```

```
// Generates the topology description file
void generate_topology(){

    // Topology File
    FILE *topology;

    int i,j,k;
    int NUM_SWITCH_TYPES;

    // Keeps track of channels used
    int channels[P_NUM_NODES];

    // Used for Random Topology Generator number of connections per node
    int node_num_connections[P_NUM_NODES];
    int node_congested;
    int connections;
    // connections per node, max number of connections is total # of nodes
    int node_connections[P_NUM_NODES][P_NUM_NODES];
    int num_rand_gen;
    int node, connection, connected, done, full;
    // Count types of nodes
    int num_node_sizes;
    int node_sizes[P_NUM_NODES];
    int node_size;
    int size_exists;
    int all_connected;
    int connected_congested_node;

    // Populate Arrays
    for (i=0;i<P_NUM_NODES;i++){
        channels[i] = 0;
        node_num_connections[i] = 0;
        node_sizes[i] = 0;
    }
    for (i=0;i<P_NUM_NODES;i++){
        for (j=0;j<P_NUM_NODES;j++){
            node_connections[i][j] = 0;
        }
    }

    // Node Types
    char topology_file[100];
    char num_nodes[100];
    char avg_connections[100];

    itoa(P_NUM_NODES,num_nodes,10);
    itoa(P_AVG_CONNECTION,avg_connections,10);

    // Generate Topology File Name
```

```
strcpy(topology_file, topology_names[P_TOPOLOGY]);
strcat(topology_file, "_");
strcat(topology_file, num_nodes);
// for random
if (P_TOPOLOGY == 6){
    strcat(topology_file, "_");
    strcat(topology_file, avg_connections);
}
strcat(topology_file, ".txt");

printf("\nTopology File Name = ");
printf(topology_file);
printf("\n");

topology = fopen(topology_file,"w");

// Generate the Topology Characteristics
//0 - Fully Connected
if (P_TOPOLOGY == 0){

    // Number of Switch Types
    // All Nodes connected to all other nodes
    NUM_SWITCH_TYPES = 1;

    //# of Nodes and Switch Types
    fprintf(topology,"%d\n", P_NUM_NODES);
    fprintf(topology,"%d ", NUM_SWITCH_TYPES);
    //only one switch type, each node connected to all other nodes
    fprintf(topology,"%d\n", P_NUM_NODES-1);

    //Node Properties
    for (i=0;i<P_NUM_NODES;i++){\
        // Node Number
        fprintf(topology, "%d ", i);
        // Node Size
        fprintf(topology, "%d ", P_NUM_NODES-1);

        //Connections
        for (j=0;j<P_NUM_NODES;j++){
            // connect to all nodes except current node
            if (j!= i){
                // current node to connect to
                fprintf(topology,"%d ", j);
                // connect to channel
                fprintf(topology,"%d ", channels[j]);
                // increment to next channel
                channels[j]++;
            }
        }
        fprintf(topology, "\n");
    }
//1 - Hypercube
}else if (P_TOPOLOGY == 1){

    // Number of Switch Types
    // Central Node, and all exterior nodes
    NUM_SWITCH_TYPES = 1;

    //# of Nodes and Switch Types
    fprintf(topology,"%d\n", P_NUM_NODES);
    // Switch Types
    fprintf(topology,"%d ", NUM_SWITCH_TYPES);
    // One Type.. 4 connections/Edge of Cube
    fprintf(topology,"%d\n", 4);

    for (i=0;i<P_NUM_NODES;i++){\
        // Node Number
        fprintf(topology, "%d ", i);
        // Node Size
        fprintf(topology, "%d ", 4);

        // Connect to Neighbours
        if (i%4==0){
            // Connect to Neighbours
            fprintf(topology,"%d ", i+1);
            fprintf(topology,"%d ", channels[i+1]);
            channels[i+1]++;
            fprintf(topology,"%d ", i+2);
            fprintf(topology,"%d ", channels[i+2]);
            channels[i+2]++;
        }else if((i-1)%4==0){
            // Connect to Neighbours
            fprintf(topology,"%d ", i-1);
            fprintf(topology,"%d ", channels[i-1]);
            channels[i-1]++;
            fprintf(topology,"%d ", i+2);
            fprintf(topology,"%d ", channels[i+2]);
            channels[i+2]++;
        }else if((i-2)%4==0){
            // Connect to Neighbours
            fprintf(topology,"%d ", i+1);
            fprintf(topology,"%d ", channels[i+1]);
            channels[i+1]++;
            fprintf(topology,"%d ", i-2);
            fprintf(topology,"%d ", channels[i-2]);
            channels[i-2]++;
        }else if((i-3)%4==0){
            // Connect to Neighbours
            fprintf(topology,"%d ", i-1);
            fprintf(topology,"%d ", channels[i-1]);
            channels[i-1]++;
            fprintf(topology,"%d ", i-2);
            fprintf(topology,"%d ", channels[i-2]);
            channels[i-2]++;
        }
        // Connect Up One Layer
        // Highest Layer
        if(i >= P_NUM_NODES-4){
            // Connect Lowest Layer
            fprintf(topology,"%d ", i-(P_NUM_NODES-4));
            // Connect to available channel
            fprintf(topology,"%d ", channels[i-(P_NUM_NODES-4)]);
            channels[i-(P_NUM_NODES-4)]++;
        }else{
            // Connect to Next Layer
            fprintf(topology,"%d ", i+4);
            // Connect to available channel
            fprintf(topology,"%d ", channels[i+4]);
            channels[i+4]++;
        }
        // Connect Down One Layer
        // Lowest Layer
        if(i < 4){
            // Connect Highest Layer
            fprintf(topology,"%d ", i+(P_NUM_NODES-4));
            // Connect to available channel
            fprintf(topology,"%d ", channels[i+(P_NUM_NODES-4)]);
            channels[i+(P_NUM_NODES-4)]++;
        }else{
            // Connect to Lower Layer
            fprintf(topology,"%d ", i-4);
            // Connect to available channel
            fprintf(topology,"%d ", channels[i-4]);
            channels[i-4]++;
        }
        fprintf(topology, "\n");
    }

//2 - Mesh
}else if (P_TOPOLOGY == 2){
    // Number of Switch Types
    // Central Node, and all exterior nodes
    NUM_SWITCH_TYPES = 3;

    //# of Nodes and Switch Types
    fprintf(topology,"%d\n", P_NUM_NODES);
    // Switch Types
    fprintf(topology,"%d ", NUM_SWITCH_TYPES);
    // Three Types
    // Corner, Edge, and Center
```

```
fprintf(topology,"%d\n", 4);
fprintf(topology,"%d\n", 3);
fprintf(topology,"%d\n", 2);

//Node Properties
for (i=0;i<P_NUM_NODES;i++){
    // Node Number
    fprintf(topology, "%d ", i);

    // Edges
    // Top Edge
    if (i < P_COLUMNS){
        //Top Left Corner
        if (i%P_COLUMNS == 0){
            // Two Connections
            fprintf(topology, "%d ", 2);
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
        //Top Right Corner
        }else if (((i+1)-P_COLUMNS)%P_COLUMNS == 0){
            // Two Connections
            fprintf(topology, "%d ", 2);
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
        }else{
            // Three Connections
            fprintf(topology, "%d ", 3);
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
        }
        // Bottom Edge
    }else if(i>=P_NUM_NODES-P_COLUMNS){

        //Bottom Left Corner
        if (i%P_COLUMNS == 0){
            // Two Connections
            fprintf(topology, "%d ", 2);
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
            // Top Neighbour
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i-P_COLUMNS]++;
        //Bottom Right Corner
        }else if (((i+1)-P_COLUMNS)%P_COLUMNS == 0){
            // Two Connections
            fprintf(topology, "%d ", 2);
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
            // Top Neighbour
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i+P_COLUMNS]++;
        }else{
            // Three Connections
            fprintf(topology, "%d ", 3);
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
            // Top Neighbour
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i-P_COLUMNS]++;
        }
    // Left Edge
    }else if (i%P_COLUMNS == 0){
        // Three Connections
        fprintf(topology, "%d ", 3);
        // Top Neighbour
```

```
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i-P_COLUMNS]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
        // Right Edge
        }else if (((i+1)-P_COLUMNS)%P_COLUMNS == 0){
            // Three Connections
            fprintf(topology, "%d ", 3);
            // Top Neighbour
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i-P_COLUMNS]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
        // Central Node
        }else{
            // Three Connections
            fprintf(topology, "%d ", 4);
            // Top Neighbour
            fprintf(topology, "%d ", i-P_COLUMNS);
            fprintf(topology, "%d ", channels[i-P_COLUMNS]);
            channels[i-P_COLUMNS]++;
            // Bottom Neighbour
            fprintf(topology, "%d ", i+P_COLUMNS);
            fprintf(topology, "%d ", channels[i+P_COLUMNS]);
            channels[i+P_COLUMNS]++;
            // Left Neighbour
            fprintf(topology, "%d ", i-1);
            fprintf(topology, "%d ", channels[i-1]);
            channels[i-1]++;
            // Right Neighbour
            fprintf(topology, "%d ", i+1);
            fprintf(topology, "%d ", channels[i+1]);
            channels[i+1]++;
        }
        fprintf(topology, "\n");
    }
```

```
//3 - Star
}else if (P_TOPOLOGY == 3){
    // Number of Switch Types
    // Central Node, and all exterior nodes
    NUM_SWITCH_TYPES = 2;

    //# of Nodes and Switch Types
    fprintf(topology,"%d\n", P_NUM_NODES);
    // Switch Types
    fprintf(topology,"%d ", NUM_SWITCH_TYPES);
    //two types
    // central node, and exterior nodes
    fprintf(topology,"%d ", P_NUM_NODES-1);
    fprintf(topology,"%d\n", 1);

    //Node Properties
    for (i=0;i<P_NUM_NODES;i++){
        // Node Number
        fprintf(topology, "%d ", i);

        // Central Node
        if (i==0){
            // Node Size
            fprintf(topology, "%d ", P_NUM_NODES-1);
            //Connections to all nodes
            for (j=0;j<P_NUM_NODES;j++){
                // connect to all nodes except current node
                if (j!= i){
                    // current node to connect to
                    fprintf(topology,"%d ", j);
                    // connect to channel 0
                    fprintf(topology,"%d ", 0);
                }
            }
        // Exterior Nodes
        }else{
            // Node Size
            fprintf(topology, "%d ", 1);
            // Connect to Central Node
            fprintf(topology,"%d ", 0);
            // Connect to available channel
            fprintf(topology,"%d ", channels[0]);
            channels[0]++;
         }
         fprintf(topology, "\n");
     }

//4 - Ring
}else if (P_TOPOLOGY == 4){
```

```
// Number of Switch Types
NUM_SWITCH_TYPES = 1;

//# of Nodes and Switch Types
fprintf(topology,"%d\n", P_NUM_NODES);
// Switch Types
fprintf(topology,"%d ", NUM_SWITCH_TYPES);
// one type, each connected to neighbours
fprintf(topology,"%d\n", 2);

//Node Properties
for (i=0;i<P_NUM_NODES;i++){
    // Node Number
    fprintf(topology, "%d ", i);

    // First Node
    if (i==0){
        // Node Size
        fprintf(topology, "%d ", 2);
        //Connected to Neighbours
        //+1 Neighbour (Channel 0)
        fprintf(topology,"%d ", i+1);
        fprintf(topology,"%d ", 0);
        // Last node
        fprintf(topology,"%d ", P_NUM_NODES-1);
        fprintf(topology,"%d ", 1);
    // All Other Nodes
    }else if (i==P_NUM_NODES-1){
        // Node Size
        fprintf(topology, "%d ", 2);
        //Connected to Neighbours
        // First Node (Channel 0)
        fprintf(topology,"%d ", 0);
        fprintf(topology,"%d ", 0);
        //-1 Neighbour (Channel 1) Last node
        fprintf(topology,"%d ", i-1);
        fprintf(topology,"%d ", 1);
    }else{
        // Node Size
        fprintf(topology, "%d ", 2);
        //Connected to Neighbours
        //+1 Neighbour
        fprintf(topology,"%d ", i+1);
        fprintf(topology,"%d ", 0);
        //-1 Neighbour
        fprintf(topology,"%d ", i-1);
        fprintf(topology,"%d ", 1);
    }
    fprintf(topology, "\n");
    }

    //5 - Torus
    }else if (P_TOPOLOGY == 5){
        // Number of Switch Types
        NUM_SWITCH_TYPES = 1;

        //# of Nodes and Switch Types
        fprintf(topology,"%d\n", P_NUM_NODES);
        // Switch Types
        fprintf(topology,"%d ", NUM_SWITCH_TYPES);
        // one type, each connected to neighbours
        fprintf(topology,"%d\n", 4);

        //Node Properties
        for (i=0;i<P_NUM_NODES;i++){
            // Node Number
            fprintf(topology, "%d ", i);
            // Node Size
            fprintf(topology, "%d ", 4);

            // First Ring
            if (i < P_RINGS){
                // First Node in Ring
                if (i==0){
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Last Node
                    fprintf(topology, "%d ", P_RINGS-1);
                    fprintf(topology, "%d ", channels[P_RINGS-1]);
                    channels[P_RINGS-1]++;
                    // Next Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                    // Last Ring
                    fprintf(topology, "%d ", P_NUM_NODES-P_RINGS);
                    fprintf(topology, "%d ", channels[P_NUM_NODES-P_RINGS]);
                    channels[P_NUM_NODES-P_RINGS]++;
                // Last Node in Ring
                }else if (i==P_RINGS-1){
                    // First Node
                    fprintf(topology, "%d ", 0);
                    fprintf(topology, "%d ", channels[0]);
                    channels[0]++;
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // Next Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                    // Last Ring
                    fprintf(topology, "%d ", P_NUM_NODES-1);
                    fprintf(topology, "%d ", channels[P_NUM_NODES-1]);
                    channels[P_NUM_NODES-1]++;
                // Other Nodes
                }else{
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // Next Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                    // Last Ring
                    fprintf(topology, "%d ", (P_NUM_NODES-P_RINGS)+i);
                    fprintf(topology, "%d ", channels[(P_NUM_NODES-P_RINGS)+i]);
                    channels[(P_NUM_NODES-P_RINGS)+i]++;
                }
            // Last Ring
            }else if (i > ((P_NUM_NODES-1)-P_RINGS)){
                // First Node in Ring
                if (i==P_NUM_NODES-P_RINGS){
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Last Node
                    fprintf(topology, "%d ", i+(P_RINGS-1));
                    fprintf(topology, "%d ", channels[i+(P_RINGS-1)]);
                    channels[i+(P_RINGS-1)]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // First Ring
                    fprintf(topology, "%d ", 0);
                    fprintf(topology, "%d ", channels[0]);
                    channels[0]++;
                // Last Node in Ring
                }else if (i==P_NUM_NODES-1){
```

```
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // First Node
                    fprintf(topology, "%d ", i-(P_RINGS-1));
                    fprintf(topology, "%d ", channels[i-(P_RINGS-1)]);
                    channels[i-(P_RINGS-1)]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // First Ring
                    fprintf(topology, "%d ", P_RINGS-1);
                    fprintf(topology, "%d ", channels[P_RINGS-1]);
                    channels[P_RINGS-1]++;
                // Other Nodes
                }else{
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // First Ring
                    fprintf(topology, "%d ", i-(P_NUM_NODES-P_RINGS));
                    fprintf(topology, "%d ", channels[i-(P_NUM_NODES-P_RINGS)]);
                    channels[i-(P_NUM_NODES-P_RINGS)]++;
                }
            // Central Ring
            }else{
                // First Node in Ring
                if (i%P_RINGS == 0){
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Last Node
                    fprintf(topology, "%d ", i+(P_RINGS-1));
                    fprintf(topology, "%d ", channels[i+(P_RINGS-1)]);
                    channels[i+(P_RINGS-1)]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // Next Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                // Last Node in Ring
                }else if ((i+1)%P_RINGS == 0){
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // First Node
                    fprintf(topology, "%d ", i-(P_RINGS-1));
                    fprintf(topology, "%d ", channels[i-(P_RINGS-1)]);
                    channels[i-(P_RINGS-1)]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // First Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                //Other Nodes
                }else{
                    // Previous Node
                    fprintf(topology, "%d ", i-1);
                    fprintf(topology, "%d ", channels[i-1]);
                    channels[i-1]++;
                    // Next Node
                    fprintf(topology, "%d ", i+1);
                    fprintf(topology, "%d ", channels[i+1]);
                    channels[i+1]++;
                    // Previous Ring
                    fprintf(topology, "%d ", i-P_RINGS);
                    fprintf(topology, "%d ", channels[i-P_RINGS]);
                    channels[i-P_RINGS]++;
                    // First Ring
                    fprintf(topology, "%d ", i+P_RINGS);
                    fprintf(topology, "%d ", channels[i+P_RINGS]);
                    channels[i+P_RINGS]++;
                }
            }
            fprintf(topology, "\n");
        }

    //6 - Random... Random connections...
    }else if (P_TOPOLOGY == 6){

        // Randomly generate Connections
```

```
// Number of Connections = Number of Nodes*Average Connections
num_rand_gen = P_NUM_NODES*P_AVG_CONNECTION/2;
for (i=0;i<num_rand_gen;i++){
    // pick a node that isn't full
    full = 0;
    while (!full){
        node = int_rand(P_NUM_NODES-1);
        if (node_num_connections[node] != P_NUM_NODES){
            full = 1;
        }
    }
    connected = 0;
    // pick a connection, must pick a new connection
    while(!connected){
        // pick a connection
        connection = int_rand(P_NUM_NODES-1);
        // assume it's a good connection
        connected = 1;
        for (j=0;j<P_NUM_NODES;j++){
            // if connection already exists
            // reset back to zero
            if(node_connections[node][j] == connection){
                connected = 0;
            }
        }
        // if connect to itself
        if (connection == node){
            connected = 0;
        }
    }
    // store connection -- bi-directional
    node_connections[node][node_num_connections[node]] = connection;
    node_connections[connection][node_num_connections[connection]] = node;
    // increment to next location to store, and increment how many connections made
    node_num_connections[node]++;
    node_num_connections[connection]++;
}

all_connected = 0;
while(!all_connected){
    // Check all nodes to ensure that they have connections
    for (i=0; i<P_NUM_NODES; i++){
        // if node has no connections
        if (node_num_connections[i] == 0){
            // CONNECT THE NODE to SOMETHING

            node_congested = 0;
            connections = 0;

            // Remove a connection from the most congested net
```

```
for (j=0;j<P_NUM_NODES;j++){
    // find most congested net
    if (node_num_connections[j] > connections){
        node_congested = j;
    connections = node_num_connections[j];
    }
}
// decrement from the most congested net
// decrement from congested node's connection
connected_congested_node = node_connections[node_congested]
        [node_num_connections[node_congested]-1];

// search through list and move congested node to end of list
for (k=0;k<node_num_connections[connected_congested_node]-1;k++){
    if (node_connections[connected_congested_node][k] == node_congested){
        // swap to end of list
        node_connections[connected_congested_node][k] = node_connections
                [connected_congested_node][node_num_connections
                [connected_congested_node]-1];
        node_connections[connected_congested_node][node_num_connections
                [connected_congested_node]] = node_congested;
    }
}
// decrement from congested node and connection
node_num_connections[connected_congested_node]--;
node_num_connections[node_congested]--; // this is correct

connected = 0;
// pick a connection, must pick a new connection
while(!connected){
    // pick a connection
    connection = int_rand(P_NUM_NODES-1);
    // assume it's a good connection
    connected = 1;
    for (j=0;j<P_NUM_NODES;j++){
        // if connection already exists
        // reset back to zero
        if(node_connections[i][j] == connection){
            connected = 0;
        }
    }
    // if connect to itself
    if (connection == i){
        connected = 0;
    }
}
// store connection -- bi-directional
node_connections[i][node_num_connections[i]] = connection;
node_connections[connection][node_num_connections[connection]] = i;
// increment to next location to store, and increment how many connections made
```

```
                node_num_connections[i]++;
                node_num_connections[connection]++;
            }
        }
        // Check all nodes to ensure that they have connections
        all_connected = 1;
        for (i=0; i<P_NUM_NODES; i++){
            if (node_num_connections[i] == 0){
                // repeat cycle if there are still empty connections
                all_connected = 0;
            }

        }
    }
    // Print out all node sizes in description file
    num_node_sizes = 0;

    // Store Node Sizes
    for (i=0;i<P_NUM_NODES;i++){
        // check node size
        node_size = node_num_connections[i];
        // assume size does not exist
        size_exists = 0;
        // scan through existing nodes
        for (j=0;j<num_node_sizes;j++){
            // if size already exists, then don't store
            if (node_size == node_sizes[j]){
                size_exists = 1;
            }
        }
        // if size did not exist
        if (size_exists == 0){
            // store new size
            node_sizes[num_node_sizes] = node_size;
            // move to next entry
            num_node_sizes++;
        }
    }


    // Print out all characteristics to the description file
    // Number of Switch Types
    NUM_SWITCH_TYPES = num_node_sizes;

    //# of Nodes and Switch Types
    fprintf(topology,"%d\n", P_NUM_NODES);
    // Switch Types
    fprintf(topology,"%d ", NUM_SWITCH_TYPES);

    for (i=0;i<NUM_SWITCH_TYPES;i++){
            fprintf(topology,"%d ", node_sizes[i]);
        }
        fprintf(topology,"\n");

        //Node Properties
        for (i=0;i<P_NUM_NODES;i++){
            // Node Properties

            // Node Number
            fprintf(topology, "%d ", i);

            // Node Size
            fprintf(topology, "%d ", node_num_connections[i]);
            for (j=0;j<node_num_connections[i];j++){
                // node connection
                fprintf(topology, "%d ", node_connections[i][j]);
                // Connect to available channel
                fprintf(topology,"%d ", channels[node_connections[i][j]]);
                channels[node_connections[i][j]]++;
            }
            fprintf(topology, "\n");
        }
    }

    fclose(topology);
}

/////////////////////////////////////////////////
// Main Function
/////////////////////////////////////////////////

int main(){

    while(1){

        // Reset control parameters
        P_TOPOLOGY = -1;
        P_NUM_NODES = -1;
        P_COLUMNS = -1;
        P_ROWS = -1;
        P_AVG_CONNECTION = -1;
        P_RINGS = -1;
        P_NUM_RINGS = -1;

        // Select Topology
        printf("Available Topologies:\n");
        printf("0 - Fully Connected\n");
        printf("1 - Hypercube\n");
        printf("2 - Mesh\n");
        printf("3 - Star\n");
        printf("4 - Ring\n");
        printf("5 - Torus\n");
        printf("6 - Random\n");

        // Ensure that user selects a topology
        while (P_TOPOLOGY < 0 || P_TOPOLOGY > 6){
            printf("Select Topology: ");
            scanf("%d", &P_TOPOLOGY);

            if (P_TOPOLOGY < 0 || P_TOPOLOGY > 6){
                printf("Invalid Selection, Choose Again\n");
            }
        }

        // Select Number of Nodes for fully, star and ring
        if (P_TOPOLOGY == 0 || P_TOPOLOGY == 3 || P_TOPOLOGY == 4){
            printf("Number of Nodes: ");
            scanf("%d", &P_NUM_NODES);
        // Error checking for Hypercube
        }else if(P_TOPOLOGY == 1){
            while (P_NUM_NODES%4 != 0){
                printf("Number of Nodes: ");
                scanf("%d", &P_NUM_NODES);
                if (P_NUM_NODES != 2 && P_NUM_NODES != 4  && P_NUM_NODES != 8  && P_NUM_NODES != 16
                        && P_NUM_NODES != 32  && P_NUM_NODES != 64  && P_NUM_NODES != 128){
                    printf("Number of Nodes must be 2^n\n");
                }
            }
        // Error checking for Mesh
        }else if(P_TOPOLOGY == 2){
            while (P_ROWS < 1){
                printf("Number of Rows in Mesh: ");
                scanf("%d", &P_ROWS);
                if (P_ROWS < 1){
                    printf("Must have more than one row\n");
                }
            }
            while (P_COLUMNS < 1){
                printf("Number of Columns in Mesh: ");
                scanf("%d", &P_COLUMNS);
                if (P_COLUMNS < 1){
                    printf("Must have more than one column\n");
                }
            }
            P_NUM_NODES = P_ROWS*P_COLUMNS;
        // Error checking for TOrus
        }else if (P_TOPOLOGY == 5){
            while (P_RINGS < 1){
                printf("Number of Nodes in Ring: ");
                scanf("%d", &P_RINGS);
```

```
                if (P_RINGS < 1){
                    printf("Must have more than 1 nodes in ring\n");
                }
            }
            while (P_NUM_RINGS < 1){
                printf("Number of Rings: ");
                scanf("%d", &P_NUM_RINGS);
                if (P_NUM_RINGS < 1){
                    printf("Must have more than 1 rings\n");
                }
            }
            P_NUM_NODES = P_NUM_RINGS*P_RINGS;
        }

        // Specify Parameters for Random Topology
        if (P_TOPOLOGY == 6){
            printf("Number of Nodes: ");
            scanf("%d", &P_NUM_NODES);
            while (P_AVG_CONNECTION < 0 || P_AVG_CONNECTION > P_NUM_NODES){
                printf("Average number of connections per node: ");
                scanf("%d", &P_AVG_CONNECTION);
                if (P_AVG_CONNECTION > P_NUM_NODES){
                    printf("Average Connections cannot exceed number of Nodes\n");
                }
                if (P_AVG_CONNECTION < 0){
                    printf("Invalid Selection, Choose Again\n");
                }
            }
        }

        printf("Generating Topology Description File\n");

        // Generate Topology
        generate_topology();
    }
}
```

A.2 Global Variables (globals.c)

The globals.c file is used to store the overall characteristics of the topology, which is used to generate the topology description file.

```
#define P_NUM_TOPOLOGIES                7

// Topologies
```

```
char topology_names[P_NUM_TOPOLOGIES][100] = {{"fully"},{"hypercube"},{"mesh"},{"star"},{"ring"},
            {"torus"},{"random"}};

// Properties
int P_NUM_NODES = -1; // Number of Nodes
int P_AVG_CONNECTION = -1;
int P_TOPOLOGY = -1;
int P_COLUMNS = -1;
int P_ROWS = -1;
int P_RINGS = -1;
int P_NUM_RINGS = -1;
```

Appendix B

System Generator

As described in Section 3.2, the system generator is used to produce all necessary files needed by the CAD tools to synthesize the NoC topology. The system generator requires several input files to operate. These files are listed below:

- constraints.txt - defines the FPGA device family, maximum operating frequency, and node types (heterogeneous or homogeneous)
- topologies.txt - lists the topology description files to use as inputs to the system generator
- ram_X - M-LAB RAM block used by Altera devices where X is the link width of the NoC topology

The system generator prompts the user for the type of system to generate. First, the user will be asked to generate an Altera or Xilinx system. If the system is a Xilinx system, the user will then decide to use either MicroBlaze or Multiplier nodes. The user will then be given an option to generate individual topologies, or all topologies listed in the 'topologies.txt' file.

The inputs files required by the program need to be in the same directory as the executable. The format of 'constraints.txt' is shown below:

```
Architecture: virtex5
Device: XC5VLX330
Package: FF1760
Clock: 250000
FSL_Width: 32
Multiplicand_Width: 6
Multiplicand_Width_Min: 4
Multiplicand_Width_Max: 8
```

The user can change the values associated with each parameter. Note that the current version of the system generator shown here can support all Xilinx and Altera devices. For Altera devices, the FPGA device family does not need to be specified here and 'altera' should be filled in for Architecture, Device, Package and Speed. The device family is specified at run time when Quartus is invoked. For the node properties, the system generator can only support even values for the FSL_Width and Multiplicand_Width. The Multiplicand_Width specifies the average value between the min and max values. If a homogeneous system is desired Multiplicand_Width = Multiplicand_Width_Min = Multiplicand_Width_Max. If a heterogeneous system is desired, Multiplicand_Width_Min must be less than Multiplicand_Width and Multiplicand_Width_Max must be greater than Multiplicand_Width. Furthermore, Multiplicand_Width_Max cannot exceed FSL_Width.

In order to specify the topologies to generate, an input file called 'topologies.txt' is used. The format of the file is shown below:

```
ring_8
ring_16
mesh_4
mesh_8
random_48_6
...
```

The user can list as many topology description files as they desire. All topology description files need to be in the same directory as the program itself. A folder with the same name as the topology, containing all necessary files is generated by the system generator.

If the user wishes to generate a system for Altera devices, a pre-generated ram block must be included in the same directory. The ram block can be generated using the MegaWizard tool, and must be specified to use only M-LAB RAM blocks. For our systems, the width of the ram block is equal to the FSL_Width and the depth is always set at 16. An example ram block used for a 16-bit FSL is shown below:

```
// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: ram_16.v
// Megafunction Name(s):
//         altsyncram
//
// Simulation Library Files(s):
//         altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 8.1 Build 163 10/28/2008 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2008 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.
```

```
// synopsys translate_off
‘timescale 1 ps / 1 ps
// synopsys translate_on
module ram (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input     clock;
    input   [15:0]  data;
    input   [3:0]  rdaddress;
    input   [3:0]  wraddress;
    input     wren;
    output  [15:0]  q;

    wire [15:0] sub_wire0;
    wire [15:0] q = sub_wire0[15:0];

    altsyncram  altsyncram_component (
                .wren_a (wren),
                .clock0 (clock),
                .address_a (wraddress),
                .address_b (rdaddress),
                .data_a (data),
                .q_b (sub_wire0),
                .aclr0 (1’b0),
                .aclr1 (1’b0),
                .addressstall_a (1’b0),
                .addressstall_b (1’b0),
                .byteena_a (1’b1),
                .byteena_b (1’b1),
                .clock1 (1’b1),
                .clocken0 (1’b1),
                .clocken1 (1’b1),
                .clocken2 (1’b1),
                .clocken3 (1’b1),
                .data_b ({16{1’b1}}),
                .eccstatus (),
                .q_a (),
                .rden_a (1’b1),
                .rden_b (1’b1),
                .wren_b (1’b0));
    defparam
        altsyncram_component.address_aclr_b = "NONE",
        altsyncram_component.address_reg_b = "CLOCK0",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_input_b = "BYPASS",
        altsyncram_component.clock_enable_output_b = "BYPASS",
        altsyncram_component.intended_device_family = "Stratix III",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 16,
        altsyncram_component.numwords_b = 16,
        altsyncram_component.operation_mode = "DUAL_PORT",
        altsyncram_component.outdata_aclr_b = "NONE",
        altsyncram_component.outdata_reg_b = "CLOCK0",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "MLAB",
        altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
        altsyncram_component.widthad_a = 4,
        altsyncram_component.widthad_b = 4,
        altsyncram_component.width_a = 16,
        altsyncram_component.width_b = 16,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "0"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: ECC NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Stratix III"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "256"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "1"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "1"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
// Retrieval info: PRIVATE: REGrren NUMERIC "1"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "0"
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "16"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Stratix III"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "16"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: RAM_BLOCK_TYPE STRING "MLAB"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING "DONT_CARE"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "4"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "4"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "16"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "16"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT NODEFVAL clock
// Retrieval info: USED_PORT: data 0 0 16 0 INPUT NODEFVAL data[15..0]
// Retrieval info: USED_PORT: q 0 0 16 0 OUTPUT NODEFVAL q[15..0]
// Retrieval info: USED_PORT: rdaddress 0 0 4 0 INPUT NODEFVAL rdaddress[3..0]
// Retrieval info: USED_PORT: wraddress 0 0 4 0 INPUT NODEFVAL wraddress[3..0]
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT VCC wren
// Retrieval info: CONNECT: @data_a 0 0 16 0 data 0 0 16 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 16 0 @q_b 0 0 16 0
// Retrieval info: CONNECT: @address_a 0 0 4 0 wraddress 0 0 4 0
// Retrieval info: CONNECT: @address_b 0 0 4 0 rdaddress 0 0 4 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16_bb.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16_waveforms.html TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL ram_16_wave*.jpg FALSE
// Retrieval info: LIB_FILE: altera_mf
//
```

B.1 Main (main.c)

The following code is the main function of the system generator. It queries the user for the input files and input parameters and utilizes a set of functions to generate the individual files needed by the Altera and Xilinx CAD tools. Below is the code for 'main.c' and all associated function calls.

```
#include <stdio.h>
#include "generate_xmp.h"
#include "generate_mhs.h"
#include "generate_mss.h"
#include "generate_switch_pcore.h"
#include "globals.h"
#include "generate_ucf.h"
#include "generate_opt.h"
```

```
#include "generate_mult.h"
#include "generate_fsl.h"

// Node Properties
int PLB_CONNECT;
int P_CONNECT_RS232;
int P_DCM_FREQUENCY;

// Xilinx CAD Tool Properties (Do Not Change)
int P_TIMING_CONSTRAINT = 1;   // don't change
int P_MAP_EFFORT = 1;
int P_PAR_EFFORT = 1;
int P_CONSTRAIN_BOARD = 0;      // don't change

// Control Variables
int P_NODE;
int P_NUM_NODE;
int Byte;
int init_multiplier;

int P_NODE_SIZE[P_MAX_NODES];  // Number of Connections for Node
int P_NODE_CONNECTIONS[P_MAX_NODES][P_MAX_NODES]; // Connections for Node
int P_NODE_CHANNELS[P_MAX_NODES][P_MAX_NODES]; // Connections for Node

int P_NUM_SWITCHES;               // Number of Switches
int P_SWITCHES[P_MAX_NODES];      // Number of types of switches

// Used to generate directory structures
char topology[100];
char directory[100];
char gen_directory[100];
char pcore_directory[100];
char switch_directory[100];
char multiplier_directory[100];
char switch_size[100];
char fsl_size[100];
char multiplicand_sizes[100];
char generate_file[100];
char hdl_directory[100];
char temp_path[100];

// Directory Structure Files
FILE *xmp_file, *mhs_file, *mss_file, *ucf_file, *v_switch, *v_switch_fsm, *switch_pao, *switch_mpd,
    *opt_file, *tcl_file;
FILE *topology_file, *system_file, *fsl_file, *inFile, *outFile;
FILE *multiplier_mpd, *multiplier_pao, *multiplier_v, *xbyx_mult_v, *xby2_mult_v, *two_mult_v;

// Used to generate tcl file that is used to automatically run Xilinx EDK
void generate_tcl(FILE *tcl_file){
    fprintf(tcl_file, "xload xmp system.xmp\n");
    fprintf(tcl_file, "xset enable_par_timing_error 0\n");
    fprintf(tcl_file, "run bits\n");
    fprintf(tcl_file, "exit\n");
}

// Generates all necessary CAD tool files for Xilinx and Altera tools
// Uses functions described in other sections of this code to generate
// individual files
void generate_system(int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH,
    int P_MULTIPLICAND_WIDTH_MIN, int P_MULTIPLICAND_WIDTH_MAX){

    int i,j,k;
    int multiplicand_width;

    P_DCM_FREQUENCY = P_CLK_FREQUENCY;

    // Change FSL and Multiplicand Width to strings
    itoa(P_FSL_WIDTH, fsl_size, 10);
    itoa(P_MULTIPLICAND_WIDTH, multiplicand_sizes, 10);
    strcpy(directory, topology);

    // Open topology description file and parse topology parameters
    strcat(topology, ".txt");
    topology_file = fopen(topology, "r");

    // P_NUM_NODES
    fscanf(topology_file, "%d", &P_NUM_NODES);

    // P_NUM_SWITCHES
    fscanf(topology_file, "%d ", &P_NUM_SWITCHES);

    // P_SWITCHES
    for (i=0; i<P_NUM_SWITCHES;i++){
        fscanf(topology_file, "%d ", &P_SWITCHES[i]);
    }

    // Node Properties
    for (i=0;i<P_NUM_NODES;i++){
        // temporary P_NODE
        fscanf(topology_file, "%d ", &P_NODE);
        // P_NODE_SIZE
        fscanf(topology_file, "%d ", &P_NODE_SIZE[P_NODE]);
        // P_NODE_CONNECTIONS
        // P_NODE_CHANNELS
        for (j=0;j<P_NODE_SIZE[P_NODE];j++){
            fscanf(topology_file, "%d %d ", &P_NODE_CONNECTIONS[P_NODE][j],
                &P_NODE_CHANNELS[P_NODE][j]);
        }
    }
    fclose(topology_file);
```

```
//////////////////////////////////////////////////////////////////
// Xilinx FPGAs
// Used to generate all Xilinx Files if chosen vendor is Xilinx
if (P_VENDOR == 1){

    // Generate Directory Structure
    mkdir(directory);
    // data directory
    strcpy(gen_directory, directory);
    strcat(gen_directory, "/data");
    mkdir(gen_directory);
    // etc directory
    strcpy(gen_directory, directory);
    strcat(gen_directory, "/etc");
    mkdir(gen_directory);
    // pcores directory
    strcpy(gen_directory, directory);
    strcat(gen_directory, "/pcores");
    mkdir(gen_directory);
    strcpy(pcore_directory, gen_directory);

    // Generate Multiplier Node Types else use MicroBlaze nodes
    if (P_NODE_TYPE == 1){
        // Multiplier
        for (i=0;i<2;i++){       // Two Types of multipliers Init and everything else
            // pcore directory
            // Generate standard size init multiplier
            if (i==0){
                // pcore directory
                strcpy(gen_directory, pcore_directory);
                strcat(gen_directory, "/init_multiplier_");
                itoa(P_FSL_WIDTH, fsl_size, 10);
                strcat(gen_directory, fsl_size);
                strcat(gen_directory, "_");
                itoa(P_MULTIPLICAND_WIDTH, multiplicand_sizes, 10);
                strcat(gen_directory, multiplicand_sizes);
                strcat(gen_directory, "_v1_00_a");
                mkdir(gen_directory);
                strcpy(switch_directory, gen_directory);
                // data directory
                strcpy(gen_directory, switch_directory);
                strcat(gen_directory, "/data");
                mkdir(gen_directory);
                // devl directory
                strcpy(gen_directory, switch_directory);
                strcat(gen_directory, "/devl");
                mkdir(gen_directory);
                // hdl directory
                strcpy(gen_directory, switch_directory);
                strcat(gen_directory, "/hdl");
                mkdir(gen_directory);
                // vhdl directory
                strcpy(hdl_directory, gen_directory);
                strcat(gen_directory, "/vhdl");
                mkdir(gen_directory);
                // verilog directory
                strcpy(gen_directory, hdl_directory);
                strcat(gen_directory, "/verilog");
                mkdir(gen_directory);
            }else{
                for (i = P_MULTIPLICAND_WIDTH_MIN;i<=P_MULTIPLICAND_WIDTH_MAX;i=i+2){
                    // pcore directory
                    strcpy(gen_directory, pcore_directory);
                    strcat(gen_directory, "/multiplier_");
                    itoa(P_FSL_WIDTH, fsl_size, 10);
                    strcat(gen_directory, fsl_size);
                    strcat(gen_directory, "_");
                    itoa(i, multiplicand_sizes, 10);
                    strcat(gen_directory, multiplicand_sizes);
                    strcat(gen_directory, "_v1_00_a");
                    mkdir(gen_directory);
                    strcpy(switch_directory, gen_directory);
                    // data directory
                    strcpy(gen_directory, switch_directory);
                    strcat(gen_directory, "/data");
                    mkdir(gen_directory);
                    // devl directory
                    strcpy(gen_directory, switch_directory);
                    strcat(gen_directory, "/devl");
                    mkdir(gen_directory);
                    // hdl directory
                    strcpy(gen_directory, switch_directory);
                    strcat(gen_directory, "/hdl");
                    mkdir(gen_directory);
                    //vhdl directory
                    strcpy(hdl_directory, gen_directory);
                    strcat(gen_directory, "/vhdl");
                    mkdir(gen_directory);
                    //verilog directory
                    strcpy(gen_directory, hdl_directory);
                    strcat(gen_directory, "/verilog");
                    mkdir(gen_directory);
                }
            }
        }
    }


    // Switches
```

```
for (i=0;i<P_NUM_SWITCHES;i++){
    strcpy(gen_directory, pcore_directory);
    // pcore directory
    strcat(gen_directory, "/switch");
    itoa(P_SWITCHES[i], switch_size, 10);
    strcat(gen_directory, switch_size);
    strcat(gen_directory, "_v1_00_a");
    mkdir(gen_directory);
    strcpy(switch_directory, gen_directory);
    // data directory
    strcpy(gen_directory, switch_directory);
    strcat(gen_directory, "/data");
    mkdir(gen_directory);
    // devl directory
    strcpy(gen_directory, switch_directory);
    strcat(gen_directory, "/devl");
    mkdir(gen_directory);
    // hdl directory
    strcpy(gen_directory, switch_directory);
    strcat(gen_directory, "/hdl");
    mkdir(gen_directory);
    //vhdl directory
    strcpy(hdl_directory, gen_directory);
    strcat(gen_directory, "/vhdl");
    mkdir(gen_directory);
    //verilog directory
    strcpy(gen_directory, hdl_directory);
    strcat(gen_directory, "/verilog");
    mkdir(gen_directory);
}

// Generate XMP File
strcpy(generate_file, directory);
strcat(generate_file, "/system.xmp");
xmp_file = fopen(generate_file, "w");
generate_xmp(xmp_file);
fclose(xmp_file);

// Generate MHS File
strcpy(generate_file, directory);
strcat(generate_file, "/system.mhs");
mhs_file = fopen(generate_file, "w");
generate_mhs(mhs_file, P_NODE_SIZE, P_NODE_CONNECTIONS, P_NODE_CHANNELS,
            P_CLK_FREQUENCY, P_NUM_NODES, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
            P_MULTIPLICAND_WIDTH_MIN, P_MULTIPLICAND_WIDTH_MAX, P_TIMING_CONSTRAINT);
fclose(mhs_file);

// Generate MSS File
strcpy(generate_file, directory);
strcat(generate_file, "/system.mss");
```

```
mss_file = fopen(generate_file, "w");
generate_mss(mss_file, P_NUM_NODES, P_NODE_SIZE, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
fclose(mss_file);

// Generate TCL File
strcpy(generate_file, directory);
strcat(generate_file, "/gen_bits.tcl");
tcl_file = fopen(generate_file, "w");
generate_tcl(tcl_file);
fclose(tcl_file);

// Generate UCF File
strcpy(generate_file, directory);
strcat(generate_file, "/data/system.ucf");
ucf_file = fopen(generate_file, "w");
generate_ucf(ucf_file, P_CLK_FREQUENCY, P_CONSTRAIN_BOARD, P_TIMING_CONSTRAINT);
fclose(ucf_file);

// Generate ETC Files
strcpy(generate_file, directory);
strcat(generate_file, "/etc/fast_runtime.opt");
opt_file = fopen(generate_file, "w");
generate_opt(opt_file, P_MAP_EFFORT, P_PAR_EFFORT);
fclose(opt_file);

// Generate Switch PCORES
for (i=0;i<P_NUM_SWITCHES;i++){
    strcpy(generate_file,pcore_directory);
    strcat(generate_file, "/switch");
    itoa(P_SWITCHES[i], switch_size, 10);
    strcat(generate_file, switch_size);
    strcat(generate_file, "_v1_00_a");
    strcpy(switch_directory, generate_file);
    // data files
    strcat(generate_file, "/data/switch");
    strcat(generate_file, switch_size);
    strcat(generate_file, "_v2_1_0.mpd");
    // generate mpd
    switch_mpd = fopen(generate_file, "w");
    strcpy(generate_file, switch_directory);
    strcat(generate_file, "/data/switch");
    strcat(generate_file, switch_size);
    strcat(generate_file, "_v2_1_0.pao");
    // generate pao
    switch_pao = fopen(generate_file, "w");
    generate_switch_data(P_SWITCHES[i], switch_mpd, switch_pao, P_FSL_WIDTH);
    fclose(switch_mpd);
    fclose(switch_pao);
    // verilog files
    strcpy(generate_file, switch_directory);
```

```
            strcat(generate_file, "/hdl/verilog/switch");
            strcat(generate_file, switch_size);
            strcat(generate_file, ".v");
            // generate switch
            v_switch = fopen(generate_file, "w");
            strcpy(generate_file, switch_directory);
            strcat(generate_file, "/hdl/verilog/switch_fsm");
            strcat(generate_file, switch_size);
            strcat(generate_file, ".v");
            // generate fsm
            v_switch_fsm = fopen(generate_file, "w");
            generate_switch_hdl(P_SWITCHES[i], v_switch, v_switch_fsm, P_FSL_WIDTH);
            fclose(v_switch);
            fclose(v_switch_fsm);
        }

        // Generate files for Multiplier Nodes
        if (P_NODE_TYPE == 1){
            // Generate all Multiplier Node Sizes Required
            for (i=0;i<((P_MULTIPLICAND_WIDTH_MAX-P_MULTIPLICAND_WIDTH_MIN)/2+2);i++){
                strcpy(generate_file,pcore_directory);

                // Generate init_multiplier (connected to output pins)
                // If this is the first node
                if (i==0){
                    strcat(generate_file, "/init_multiplier_");
                    multiplicand_width = P_MULTIPLICAND_WIDTH;
                // Else generate standard multiplier node
                }else{
                    strcat(generate_file, "/multiplier_");
                    multiplicand_width = P_MULTIPLICAND_WIDTH_MIN+2*(i-1);
                }
                // File structure of Multiplier node
                itoa(P_FSL_WIDTH, fsl_size, 10);
                strcat(generate_file, fsl_size);
                strcat(generate_file, "_");
                itoa(multiplicand_width, multiplicand_sizes, 10);
                strcat(generate_file, multiplicand_sizes);
                strcat(generate_file, "_v1_00_a");
                strcpy(multiplier_directory, generate_file);

                // File structure for data files
                if (i==0){
                    strcat(generate_file, "/data/init_multiplier_");
                }else{
                    strcat(generate_file, "/data/multiplier_");
                }
                strcat(generate_file, fsl_size);
                strcat(generate_file, "_");
                strcat(generate_file, multiplicand_sizes);
```

```
strcat(generate_file, "_v2_1_0.mpd");

// generate mpd
multiplier_mpd = fopen(generate_file, "w");
strcpy(generate_file, multiplier_directory);
if (i==0){
    strcat(generate_file, "/data/init_multiplier_");
}else{
    strcat(generate_file, "/data/multiplier_");
}
strcat(generate_file, fsl_size);
strcat(generate_file, "_");
strcat(generate_file, multiplicand_sizes);
strcat(generate_file, "_v2_1_0.pao");

// generate pao
multiplier_pao = fopen(generate_file, "w");
if (i==0){init_multiplier = 1;}else{init_multiplier = 0;}
gen_mult_data(multiplier_mpd, multiplier_pao, P_FSL_WIDTH,
    multiplicand_width, init_multiplier);
fclose(multiplier_mpd);
fclose(multiplier_pao);

// Generate Verilog Files
strcpy(generate_file, multiplier_directory);
// generate multipliers
if (i==0){
    strcat(generate_file, "/hdl/verilog/init_multiplier_");
}else{
    strcat(generate_file, "/hdl/verilog/multiplier_");
}
strcat(generate_file, fsl_size);
strcat(generate_file, "_");
strcat(generate_file, multiplicand_sizes);
strcat(generate_file, ".v");
multiplier_v = fopen(generate_file, "w");
if (i==0){init_multiplier = 1;}else{init_multiplier = 0;}
gen_multiplier(multiplier_v,P_FSL_WIDTH, multiplicand_width, init_multiplier);

// generate xbyx multiplier
strcpy(generate_file, multiplier_directory);
strcat(generate_file, "/hdl/verilog/xbyx_bit_multiplier.v");
xbyx_mult_v = fopen(generate_file, "w");
gen_xbyx(xbyx_mult_v, P_FSL_WIDTH, multiplicand_width);

// generate xby2 multiplier
strcpy(generate_file, multiplier_directory);
strcat(generate_file, "/hdl/verilog/xbytwo_bit_multiplier.v");
xby2_mult_v = fopen(generate_file, "w");
gen_xbytwo(xbyx_mult_v, P_FSL_WIDTH, multiplicand_width);
```

```
                // generate 2 bit multiplier
                strcpy(generate_file, multiplier_directory);
                strcat(generate_file, "/hdl/verilog/two_bit_multiplier.v");
                two_mult_v = fopen(generate_file, "w");
                gen_twobytwo(two_mult_v);
                fclose(multiplier_v);
                fclose(xbyx_mult_v);
                fclose(xby2_mult_v);
                fclose(two_mult_v);
            }
        }
    }

    ////////////////////////////////////////////////////////////////////////////
    // Altera FPGAs
    // Used to generate all Altera CAD tool files if chosen vendor is Altera
    else{
        // Generate Directory Structure
        mkdir(directory);

        // Generate all system verilog files
        strcpy(generate_file, directory);
        strcat(generate_file, "/system.v");
        system_file = fopen(generate_file, "w");
        generate_system_verilog(system_file, P_NODE_SIZE, P_NODE_CONNECTIONS, P_NODE_CHANNELS,
            P_NUM_NODES, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_MULTIPLICAND_WIDTH_MIN,
            P_MULTIPLICAND_WIDTH_MAX);
        fclose(system_file);

        // Generate custom FSL File
        strcpy(generate_file, directory);
        strcat(generate_file, "/fsl.v");
        fsl_file = fopen(generate_file, "w");
        gen_fsl(fsl_file, P_FSL_WIDTH);
        fclose(fsl_file);

        // Generate Switch Files
        for (i=0;i<P_NUM_SWITCHES;i++){

            // verilog files
            strcpy(generate_file, directory);
            strcat(generate_file, "/switch_");
            itoa(P_SWITCHES[i], switch_size, 10);
            strcat(generate_file, switch_size);
            strcat(generate_file, ".v");
            v_switch = fopen(generate_file, "w");
            strcpy(generate_file, directory);
            strcat(generate_file, "/switch_fsm_");
            strcat(generate_file, switch_size);
            strcat(generate_file, ".v");
            v_switch_fsm = fopen(generate_file, "w");

            // generate verilog files
            generate_switch_hdl(P_SWITCHES[i], v_switch, v_switch_fsm, P_FSL_WIDTH);
            fclose(v_switch);
            fclose(v_switch_fsm);
        }


        // Generate Multiplier Nodes
        for (i=0;i<((P_MULTIPLICAND_WIDTH_MAX-P_MULTIPLICAND_WIDTH_MIN)/2+2);i++){

            // Init_Multiplier and Standard Multiplier
            if (i==0){
                strcpy(generate_file, directory);
                strcat(generate_file, "/init_multiplier_");
                multiplicand_width = P_MULTIPLICAND_WIDTH;

            }else{
                strcpy(generate_file, directory);
                strcat(generate_file, "/multiplier_");
                multiplicand_width = P_MULTIPLICAND_WIDTH_MIN+2*(i-1);
            }
            itoa(P_FSL_WIDTH, fsl_size, 10);
            strcat(generate_file, fsl_size);
            strcat(generate_file, "_");
            itoa(multiplicand_width, multiplicand_sizes, 10);
            strcat(generate_file, multiplicand_sizes);
            strcat(generate_file, ".v");
            multiplier_v = fopen(generate_file, "w");

            if (i==0){init_multiplier = 1;}else{init_multiplier = 0;}
            gen_multiplier(multiplier_v,P_FSL_WIDTH, multiplicand_width, init_multiplier);

            // generate xbyx multiplier
            strcpy(generate_file, directory);
            strcat(generate_file, "/xbyx_bit_multiplier_");
            itoa(P_FSL_WIDTH, fsl_size, 10);
            strcat(generate_file, fsl_size);
            strcat(generate_file, "_");
            itoa(multiplicand_width, multiplicand_sizes, 10);
            strcat(generate_file, multiplicand_sizes);
            strcat(generate_file, ".v");
            xbyx_mult_v = fopen(generate_file, "w");
            gen_xbyx(xbyx_mult_v, P_FSL_WIDTH, multiplicand_width);

            // generate xby2 multiplier
            strcpy(generate_file, directory);
            strcat(generate_file, "/xbytwo_bit_multiplier_");
```

```
            itoa(P_FSL_WIDTH, fsl_size, 10);
            strcat(generate_file, fsl_size);
            strcat(generate_file, "_");
            itoa(multiplicand_width, multiplicand_sizes, 10);
            strcat(generate_file, multiplicand_sizes);
            strcat(generate_file, ".v");
            xby2_mult_v = fopen(generate_file, "w");
            gen_xbytwo(xbyx_mult_v, P_FSL_WIDTH, multiplicand_width);

            // generate 2 bit multiplier
            strcpy(generate_file, directory);
            strcat(generate_file, "/two_bit_multiplier.v");
            two_mult_v = fopen(generate_file, "w");
            gen_twobytwo(two_mult_v);
            fclose(multiplier_v);
            fclose(xbyx_mult_v);
            fclose(xby2_mult_v);
            fclose(two_mult_v);
        }

        fclose(topology_file);

        // copy Altera M-LAB RAM block, which is used for the custom FSL
        strcpy(generate_file, "ram_");
        strcat(generate_file, fsl_size);
        strcat(generate_file, ".v");
        inFile = fopen(generate_file, "rb");
        strcpy(generate_file, directory);
        strcat(generate_file, "/ram.v");
        outFile = fopen(generate_file, "wb");

        // Copy entire file
        while(1){
             if(Byte!=EOF){
                 Byte=fgetc(inFile);
                 fputc(Byte,outFile);
             }
             else{
                 break; // adds weird character at end of file
             }
         }
         Byte = 0;

         fclose(outFile);
         fclose(inFile);
    }
}

//////////////////////////////////////////////////////
// Main Function used to prompt user for all constraints
// and calls function to generate all necessary files
//
int main(){

    FILE *constraints;
    FILE *topologies;
    char ignore[100];
    int num_topology;

    int i,generate_all;

    // Generate for Xilinx or Altera?
    printf("Generate for Xilinx or Altera? (1-Xilinx, 0-Altera): ");
    scanf("%d", &P_VENDOR);

    // If Xilinx, prompt user for MicroBlaze or Multiplier Nodes
    if (P_VENDOR == 1){
        printf("Use Multiplier or MicroBlaze nodes? (1-Multiplier, 0-MicroBlaze): ");
        scanf("%d", &P_NODE_TYPE);
    }else{
        P_NODE_TYPE = 1;
    }

    printf("\nReading Constraints File\n");

    // Open Constraints File
    constraints = fopen("constraints.txt", "r");

    // Scan Constraint File for FPGA Architecture Parameters
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%s", &P_ARCH);
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%s", &P_DEVICE);
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%s", &P_PACKAGE);
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%s", &P_SPEED);

    // Clock Frequency
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%d", &P_CLK_FREQUENCY);

    // FSL and Node Parameters
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%d", &P_FSL_WIDTH);

    // force FSL width to 32 bits if using MicroBlaze nodes
    if (P_NODE_TYPE == 0){
        P_FSL_WIDTH = 32;
    }
    // Multiplier Properties
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%d", &P_MULTIPLICAND_WIDTH);
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%d", &P_MULTIPLICAND_WIDTH_MIN);
    fscanf(constraints, "%s", &ignore);
    fscanf(constraints, "%d", &P_MULTIPLICAND_WIDTH_MAX);
    fclose(constraints);

    // Generate all topologies from file?
    printf("Generate all topologies from 'topologies.txt'? (1-Yes, 0-No): ");
    scanf("%d", &generate_all);

    if (generate_all == 0){
    // Parse Topology
        while(1){
        printf("\nEnter topology benchmark name: ");
        scanf("%s", &topology);
        generate_system(P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_MULTIPLICAND_WIDTH_MIN,
            P_MULTIPLICAND_WIDTH_MAX);
        }
    }else{
        printf("Generating All Topologies\n");

        // Read in File
        topologies = fopen("topologies.txt", "r");
        fscanf(topologies, "%d", &num_topology);

        // Generate each topology
        for (i=0;i<num_topology;i++){
            fscanf(topologies, "%s", &topology);
            generate_system(P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_MULTIPLICAND_WIDTH_MIN,
                P_MULTIPLICAND_WIDTH_MAX);
        }

        fclose(topologies);
    }
}
```

B.2 Global Variables (globals.c)

The globals.c file is used to store the overall characteristics of the topology, which is used to generate the NoC topology.

```
// Read in from Description File
// FPGA Architecture Properties
int P_VENDOR;        // 1 = Xilinx, 0 = Altera;
int P_NODE_TYPE;     // 1 = Multiplier, 0 = MicroBlaze
char P_ARCH[100];
char P_DEVICE[100];
char P_PACKAGE[100];
char P_SPEED[100];

// Node Properties
int P_CLK_FREQUENCY;
int P_FSL_WIDTH;
int P_MULTIPLICAND_WIDTH;
int P_MULTIPLICAND_WIDTH_MIN;
int P_MULTIPLICAND_WIDTH_MAX;
int multiplicand_size[128];
int P_NUM_NODES;
```

B.3 Generate Multiplier PCORE (generate_mult.c)

The following functions are used to generate the individual HDL files and data files for a specified multiplier node size. The functions are capable of generating any even multiplier node size. The functions used are shown below:

```
#include <stdio.h>
#include "globals.h"

////////////////////////////////////////////////////////////////////////////////
// TWO BY TWO MULTIPLIER
// Bottom Level module for multiplier node 2bit by 2 bit multiplier
//
////////////////////////////////////////////////////////////////////////////////

void gen_twobytwo(FILE *twobytwo_mult){

    ////////////////////////////////////////////////////////////////////////////
    // Instantiate Module

    fprintf(twobytwo_mult, "‘timescale 1ns / 1ps\n");
    fprintf(twobytwo_mult, "/////////////////////////////////////////////////////////\n");
    fprintf(twobytwo_mult, "// twobytwo mult \n");
    fprintf(twobytwo_mult, "//    by: Jason Lee \n");
    fprintf(twobytwo_mult, "// \n");
    fprintf(twobytwo_mult, "/////////////////////////////////////////////////////////\n\n");
    fprintf(twobytwo_mult, "module two_bit_multiplier(\n\n");

    fprintf(twobytwo_mult, "\t multiplicand,\n");
    fprintf(twobytwo_mult, "\t multiplier,\n");
    fprintf(twobytwo_mult, "\t result\n\n");

    fprintf(twobytwo_mult, ");\n\n");

    ////////////////////////////////////////////////////////////////////////////
    // Port List

    fprintf(twobytwo_mult, "/////////////////////////////////////////////////////////\n");
    fprintf(twobytwo_mult, "// Ports\n\n");

    fprintf(twobytwo_mult, "\tinput\t\t\t[1:0]\t\t\tmultiplicand;\n");
    fprintf(twobytwo_mult, "\tinput\t\t\t[1:0]\t\t\tmultiplier;\n");
    fprintf(twobytwo_mult, "\toutput\t\t\t[3:0]\t\t\tresult;\n\n");

    ////////////////////////////////////////////////////////////////////////////
    // Wires and Registers
    fprintf(twobytwo_mult, "////////////////////////////////////////////////////////\n");
    fprintf(twobytwo_mult, "// Wires and Registers\n\n");

    fprintf(twobytwo_mult, "\twire\t\t\t[3:0]\t\t\tpartial_product_0;\n");
    fprintf(twobytwo_mult, "\twire\t\t\t[3:0]\t\t\tpartial_product_1;\n\n");

    fprintf(twobytwo_mult, "\twire\t\t\t[1:0]\t\t\tmultiplicand_0;\n");
    fprintf(twobytwo_mult, "\twire\t\t\t[1:0]\t\t\tmultiplicand_1;\n\n");

    ////////////////////////////////////////////////////////////////////////
    // Assigns

    fprintf(twobytwo_mult, "////////////////////////////////////////////////////////\n");
    fprintf(twobytwo_mult, "// Assigns\n\n");

    fprintf(twobytwo_mult, "\t// Multiplicands\n");
    fprintf(twobytwo_mult, "\tassign multiplicand_0[0] = multiplicand[0];\n");
    fprintf(twobytwo_mult, "\tassign multiplicand_0[1] = multiplicand[0];\n\n");

    fprintf(twobytwo_mult, "\tassign multiplicand_1[0] = multiplicand[1];\n");
    fprintf(twobytwo_mult, "\tassign multiplicand_1[1] = multiplicand[1];\n\n");

    fprintf(twobytwo_mult, "\t// Partial Products\n");
    fprintf(twobytwo_mult, "\tassign partial_product_0[1:0] = multiplier & multiplicand_0;\n");
    fprintf(twobytwo_mult, "\tassign partial_product_0[3:2] = 0;\n\n");

    fprintf(twobytwo_mult, "\tassign partial_product_1[0] = 0;\n");
    fprintf(twobytwo_mult, "\tassign partial_product_1[2:1] = multiplier & multiplicand_1;\n");
    fprintf(twobytwo_mult, "\tassign partial_product_1[3] = 0;\n\n");

    fprintf(twobytwo_mult, "\t// Result\n");
    fprintf(twobytwo_mult, "\tassign result = partial_product_0 + partial_product_1;\n\n");

    fprintf(twobytwo_mult, "endmodule\n");

    fclose(twobytwo_mult);

}

////////////////////////////////////////////////////////////////////////////////
// X BY TWO MULTIPLIER
// Second level module for multiplier, instantiates 2bit by 2bit multipliers
// to form Xbytwo bit multiplier depending on FSL width
//
////////////////////////////////////////////////////////////////////////////////

void gen_xbytwo(FILE *xbytwo_mult, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH){

    int P_NUM_STAGES;
    int P_NUM_MULTIPLIERS;
    int P_NUM_SUMS;
    int P_NUM_POINTS;
    int i,j,k;

    int num_points_per_stage[10];

    // Number of Stages
    if (P_FSL_WIDTH<4){
        P_NUM_STAGES = 0;
    }else if (P_FSL_WIDTH<8){
        P_NUM_STAGES = 1;
    }else if (P_FSL_WIDTH<14){
        P_NUM_STAGES = 2;
    }else if (P_FSL_WIDTH<26){
        P_NUM_STAGES = 3;
    }else if (P_FSL_WIDTH<48){
        P_NUM_STAGES = 4;
    }else if (P_FSL_WIDTH<98){
        P_NUM_STAGES = 5;
    }else if (P_FSL_WIDTH<=128){
        P_NUM_STAGES = 6;
    }

    P_NUM_MULTIPLIERS = P_FSL_WIDTH/2;

    ////////////////////////////////////////////////////////////////////////////
    // Instantiate Module

    fprintf(xbytwo_mult, "‘timescale 1ns / 1ps\n");
    fprintf(xbytwo_mult, "//////////////////////////////////////////////////\n");
    fprintf(xbytwo_mult, "//xbytwo mult \n");
    fprintf(xbytwo_mult, "//    by: Jason Lee \n");
    fprintf(xbytwo_mult, "// \n");
    fprintf(xbytwo_mult, "////////////////////////////////////////////////\n\n");

    if (P_VENDOR == 1){
        fprintf(xbytwo_mult, "module xbytwo_bit_multiplier(\n\n");
    }else{
        fprintf(xbytwo_mult, "module xbytwo_bit_multiplier_%d_%d(\n\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }

    fprintf(xbytwo_mult, "\tsys_clk,\n");
    fprintf(xbytwo_mult, "\tsys_rst,\n\n");

    fprintf(xbytwo_mult, "\tmultiplicand,\n");
    fprintf(xbytwo_mult, "\tmultiplier,\n");
    fprintf(xbytwo_mult, "\tresult,\n\n");

    fprintf(xbytwo_mult, "\tinput_data_valid,\n");
    //fprintf(xbytwo_mult, "\tready_for_data,\n");
    fprintf(xbytwo_mult, "\tdata_valid\n");
    fprintf(xbytwo_mult, ");\n\n");


    //////////////////////////////////////////////////////////////////////////////
    // Parameters

    fprintf(xbytwo_mult, "//////////////////////////////////////////////////////////\n");
    fprintf(xbytwo_mult, "// Parameters\n\n");

    fprintf(xbytwo_mult, "\tlocalparam P_MULTIPLIER_WIDTH\t= %d;\n", P_FSL_WIDTH);
    fprintf(xbytwo_mult, "\tlocalparam P_MULTIPLICAND_WIDTH\t= 2;\n");
    fprintf(xbytwo_mult, "\tlocalparam P_RESULT_WIDTH\t\t= P_MULTIPLIER_WIDTH
        +P_MULTIPLICAND_WIDTH;\n\n");

    //////////////////////////////////////////////////////////////////////////////
    // Ports

    fprintf(xbytwo_mult, "/////////////////////////////////////////////////////////\n");
    fprintf(xbytwo_mult, "// Ports\n\n");

    fprintf(xbytwo_mult, "\tinput\t\t\t\t\t\tsys_clk;\n");
    fprintf(xbytwo_mult, "\tinput\t\t\t\t\t\tsys_rst;\n\n");

    fprintf(xbytwo_mult, "\tinput\t\t[P_MULTIPLICAND_WIDTH-1:0]\t\tmultiplicand;\n");
    fprintf(xbytwo_mult, "\tinput\t\t[P_MULTIPLIER_WIDTH-1:0]\t\tmultiplier;\n");
    fprintf(xbytwo_mult, "\toutput\t[P_RESULT_WIDTH-1:0]\t\t\tresult;\n\n");

    fprintf(xbytwo_mult, "\tinput\t\t\t\t\t\t\tinput_data_valid;\n");
    //fprintf(xbytwo_mult, "\toutput\t\t\t\t\t\tready_for_data;\n");
    fprintf(xbytwo_mult, "\toutput\t\t\t\t\t\tdata_valid;\n\n");

    //////////////////////////////////////////////////////////////////////////////
    // States

    fprintf(xbytwo_mult, "/////////////////////////////////////////////////////////\n");
    fprintf(xbytwo_mult, "// States\n\n");

    fprintf(xbytwo_mult, "\tlocalparam\tIdle_State\t\t= 0;\n");
    fprintf(xbytwo_mult, "\tlocalparam\tMultiply_State\t\t= 1;\n");

    j = 2;
    for (i=0;i<P_NUM_STAGES-1;i++){
        fprintf(xbytwo_mult, "\tlocalparam\tAdd_State_%d\t\t= %d;\n",i,j);
        j++;
    }

    fprintf(xbytwo_mult, "\tlocalparam\tFinal_Add_State\t\t= %d;\n",j);
    fprintf(xbytwo_mult, "\tlocalparam\tTransmit_State\t\t= %d;\n\n",j+1);
```

```
////////////////////////////////////////////////////////////////////////////////
// Wires and Registers

fprintf(xbytwo_mult, "////////////////////////////////////////////////////////////\n");
fprintf(xbytwo_mult, "// Wires and Registers\n\n");

fprintf(xbytwo_mult, "\t// State Machine\n");
fprintf(xbytwo_mult, "\treg\t\t\t[3:0]\t\t\tmultiplier_state_cs;\n");
fprintf(xbytwo_mult, "\treg\t\t\t[3:0]\t\t\tmultiplier_state_ns;\n\n");

fprintf(xbytwo_mult, "\t// Generate More for Longer Lengths\n");
fprintf(xbytwo_mult, "\t// Multipliers\n");
for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\twire\t\t\t[1:0]\t\t\tmultiplicand_%d;\n",i);
    fprintf(xbytwo_mult, "\twire\t\t\t[1:0]\t\t\tmultiplier_%d;\n",i);
    fprintf(xbytwo_mult, "\twire\t\t\t[3:0]\t\t\tresult_%d;\n",i);
    fprintf(xbytwo_mult, "\treg\t\t\t[3:0]\t\t\tmultiplier_result_%d;\n\n",i);
}

fprintf(xbytwo_mult, "\t// pipeline stages\n");
fprintf(xbytwo_mult, "\treg\t\t\t[P_MULTIPLICAND_WIDTH-1:0]\t\t\tinput_multiplicand;\n");
fprintf(xbytwo_mult, "\treg\t\t\t[P_MULTIPLIER_WIDTH-1:0]\t\t\tinput_multiplier;\n\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\twire\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tpartial_product_%d;\n",i);
}
fprintf(xbytwo_mult, "\n");

// Partial Sum
P_NUM_POINTS = P_NUM_MULTIPLIERS;

for(i=0;i<P_NUM_STAGES-1;i++){

    // Number of Sums to do for this stage
    P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
    if (P_NUM_POINTS%2 ==1){
        P_NUM_SUMS++;
    }

    // Keep track of number of sums for this stage
    P_NUM_POINTS = P_NUM_SUMS;
    num_points_per_stage[i] = P_NUM_POINTS;

    for(j=0;j<P_NUM_SUMS;j++){
        fprintf(xbytwo_mult, "\treg\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tpartial_sum_%d_%d;\n",i,j);
    }
    fprintf(xbytwo_mult, "\n");
}
```

```
fprintf(xbytwo_mult, "\treg\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tresult;\n\n");

/////////////////////////////////////////////////////////////////////////////////////
// Assigns

fprintf(xbytwo_mult, "/////////////////////////////////////////////////////////////////\n");
fprintf(xbytwo_mult, "// Assigns\n\n");

// Multipler Inputs
fprintf(xbytwo_mult, "\t// Generate More for Longer Lengths\n");
for(i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\tassign multiplicand_%d = input_multiplicand;\n", i);
}
fprintf(xbytwo_mult, "\n");

for(i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\tassign multiplier_%d= input_multiplier[%d:%d];\n", i,2*i+1,2*i);
}
fprintf(xbytwo_mult, "\n");

// Partial Products
for(i=0;i<P_NUM_MULTIPLIERS;i++){

    if (i!=0){
        fprintf(xbytwo_mult, "\tassign partial_product_%d[%d:0] = 0;\n", i, 2*i-1);
    }
        fprintf(xbytwo_mult, "\tassign partial_product_%d[%d:%d] =
            multiplier_result_%d;\n", i,2*i+3,2*i,i);

    if (i!=P_NUM_MULTIPLIERS-1){
        fprintf(xbytwo_mult, "\tassign partial_product_%d[%d:%d] = 0;\n",i,P_FSL_WIDTH+1,2*i+4);
    }
        fprintf(xbytwo_mult, "\n");
}

fprintf(xbytwo_mult, "\t// Control Signals\n");
fprintf(xbytwo_mult, "\tassign data_valid = (multiplier_state_cs == Transmit_State);\n");
//fprintf(xbytwo_mult, "\tassign ready_for_data = (multiplier_state_cs == Idle_State);\n\n");

/////////////////////////////////////////////////////////////////////////////////////
// State Machine

fprintf(xbytwo_mult, "/////////////////////////////////////////////////////////////////\n");
fprintf(xbytwo_mult, "// State Machine\n\n");

// Register part of state machine
fprintf(xbytwo_mult, "\t// Register part of state machine\n");
fprintf(xbytwo_mult, "\talways @(posedge sys_clk)  \n");
fprintf(xbytwo_mult, "\tbegin\n");
fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1) // reset active high\n");
fprintf(xbytwo_mult, "\t\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\t\tmultiplier_state_cs <= Idle_State;\n");
fprintf(xbytwo_mult, "\t\t\tend\n");
fprintf(xbytwo_mult, "\t\telse\n");
fprintf(xbytwo_mult, "\t\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\t\tmultiplier_state_cs <= multiplier_state_ns;\n");
fprintf(xbytwo_mult, "\t\t\tend\n");
fprintf(xbytwo_mult, "\tend\n\n");

fprintf(xbytwo_mult, "\talways @(sys_rst or input_data_valid or multiplier_state_cs)\n");
fprintf(xbytwo_mult, "\tbegin\n");
fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\t\telse\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tcase(multiplier_state_cs)\n");
fprintf(xbytwo_mult, "\t\t\t\tIdle_State:\n");
fprintf(xbytwo_mult, "\t\t\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\t\t\tif (input_data_valid == 1'b1)\n");
fprintf(xbytwo_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Multiply_State;\n");
fprintf(xbytwo_mult, "\t\t\t\t\telse\n");
fprintf(xbytwo_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbytwo_mult, "\t\t\t\tend\n");
fprintf(xbytwo_mult, "\t\t\t\tMultiply_State:\n");
fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_state_ns <= Add_State_0;\n");

// Add Stages
for (i=0;i<P_NUM_STAGES-1;i++){

    if (i==P_NUM_STAGES-2){
        fprintf(xbytwo_mult, "\t\t\t\tAdd_State_%d:\n",i);
        fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_state_ns <= Final_Add_State;\n");
    }else{
        fprintf(xbytwo_mult, "\t\t\t\tAdd_State_%d:\n",i);
        fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_state_ns <= Add_State_%d;\n",i+1);
    }
}

fprintf(xbytwo_mult, "\t\t\t\tFinal_Add_State:\n");
fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_state_ns <= Transmit_State;\n");
fprintf(xbytwo_mult, "\t\t\t\tTransmit_State:\n");
fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbytwo_mult, "\t\t\t\tdefault: multiplier_state_ns <= multiplier_state_ns;\n");
fprintf(xbytwo_mult, "\t\t\tendcase\n");
fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\tend\n\n");

// multiplier && multiplicand
```

```
fprintf(xbytwo_mult, "\t// multiplier && multiplicand\n");
fprintf(xbytwo_mult, "\talways@(posedge sys_clk)\n");
fprintf(xbytwo_mult, "\tbegin\n");
fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tinput_multiplicand <= 0;\n");
fprintf(xbytwo_mult, "\t\t\tinput_multiplier <= 0;\n");
fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\t\telse\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tcase (multiplier_state_cs)\n");
fprintf(xbytwo_mult, "\t\t\tIdle_State:\n");
fprintf(xbytwo_mult, "\t\t\t\tif (input_data_valid == 1'b1)\n");
fprintf(xbytwo_mult, "\t\t\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\t\t\tinput_multiplicand <= multiplicand;\n");
fprintf(xbytwo_mult, "\t\t\t\tinput_multiplier <= multiplier;\n");
fprintf(xbytwo_mult, "\t\t\t\tend\n");
fprintf(xbytwo_mult, "\t\t\tdefault:\n");
fprintf(xbytwo_mult, "\t\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\t\tinput_multiplicand <= input_multiplicand;\n");
fprintf(xbytwo_mult, "\t\t\t\tinput_multiplier <= input_multiplier;\n");
fprintf(xbytwo_mult, "\t\t\tend\n");
fprintf(xbytwo_mult, "\t\t\tendcase\n");
fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\tend\n\n");

// Multiplier Results
fprintf(xbytwo_mult, "\t// Multiplier Results\n");
fprintf(xbytwo_mult, "\t// Generate More for Longer Lengths\n");
fprintf(xbytwo_mult, "\talways@(posedge sys_clk)\n");
fprintf(xbytwo_mult, "\tbegin\n");
fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbytwo_mult, "\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\t\t\tmultiplier_result_%d <= 0;\n",i);
}

fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\t\telse\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tcase (multiplier_state_cs)\n");
fprintf(xbytwo_mult, "\t\t\tMultiply_State:\n");
fprintf(xbytwo_mult, "\t\t\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\t\t\t\t\tmultiplier_result_%d <= result_%d;\n",i,i);
}

fprintf(xbytwo_mult, "\t\t\t\tend\n");
fprintf(xbytwo_mult, "\t\t\tdefault:\n");
fprintf(xbytwo_mult, "\t\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbytwo_mult, "\t\t\t\tmultiplier_result_%d <= multiplier_result_%d;\n",i,i);
}

fprintf(xbytwo_mult, "\t\t\tend\n");
fprintf(xbytwo_mult, "\t\t\tendcase\n");
fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\tend\n\n");

// Partial Sums
fprintf(xbytwo_mult, "\t// Partial Sums\n");
fprintf(xbytwo_mult, "\talways@(posedge sys_clk)\n");
fprintf(xbytwo_mult, "\tbegin\n");
fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbytwo_mult, "\t\tbegin\n");

// Partial Sums
P_NUM_POINTS = P_NUM_MULTIPLIERS;

for(i=0;i<P_NUM_STAGES-1;i++){

    // Number of Sums to do for this stage
    P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
    if (P_NUM_POINTS%2 ==1){
        P_NUM_SUMS++;
    }

    // Keep track of number of sums for this stage
    P_NUM_POINTS = P_NUM_SUMS;
    num_points_per_stage[i] = P_NUM_POINTS;

    for(j=0;j<P_NUM_SUMS;j++){
        fprintf(xbytwo_mult, "\t\t\tpartial_sum_%d_%d <= 0;\n",i,j);
    }
}
 fprintf(xbytwo_mult, "\n");

fprintf(xbytwo_mult, "\t\tend\n");
fprintf(xbytwo_mult, "\t\telse\n");
fprintf(xbytwo_mult, "\t\tbegin\n");
fprintf(xbytwo_mult, "\t\t\tcase (multiplier_state_cs)\n");


for(i=0;i<P_NUM_STAGES-1;i++){
    fprintf(xbytwo_mult, "\t\t\tAdd_State_%d:\n",i);
    fprintf(xbytwo_mult, "\t\t\t\tbegin\n");
```

```
        if (i==0){
            for(j=0;j<num_points_per_stage[i];j++){
                if(j==num_points_per_stage[i]-1 && P_NUM_MULTIPLIERS%2!=0){
                    fprintf(xbytwo_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_product_%d;\n",i,j,2*j);
                }else{
                    fprintf(xbytwo_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_product_%d+partial_product_%d;\n",i,j,2*j,2*j+1);
                }
            }

        }else{
            for(j=0;j<num_points_per_stage[i];j++){
                if(j==num_points_per_stage[i]-1 && num_points_per_stage[i-1]%2!=0){
                    fprintf(xbytwo_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_sum_%d_%d;\n",i,j,i-1,2*j);
                }else{
                    fprintf(xbytwo_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_sum_%d_%d+partial_sum_%d_%d;\n",i,j,i-1,2*j,i-1,2*j+1);
                }
            }
        }

        fprintf(xbytwo_mult, "\t\t\t\tend\n");
    }
    fprintf(xbytwo_mult, "\n");

    fprintf(xbytwo_mult, "\t\t\tdefault:\n");
    fprintf(xbytwo_mult, "\t\t\tbegin\n");

    // Partial Sums
    P_NUM_POINTS = P_NUM_MULTIPLIERS;

    for(i=0;i<P_NUM_STAGES-1;i++){

    // Number of Sums to do for this stage q
    P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
    if (P_NUM_POINTS%2 ==1){
        P_NUM_SUMS++;
    }

        // Keep track of number of sums for this stage
        P_NUM_POINTS = P_NUM_SUMS;
        num_points_per_stage[i] = P_NUM_POINTS;

        for(j=0;j<P_NUM_SUMS;j++){
            fprintf(xbytwo_mult, "\t\t\t\tpartial_sum_%d_%d <= partial_sum_%d_%d;\n",i,j,i,j);
        }
    }
    fprintf(xbytwo_mult, "\n");
    fprintf(xbytwo_mult, "\t\t\tend\n");
    fprintf(xbytwo_mult, "\t\t\tendcase\n");
    fprintf(xbytwo_mult, "\t\tend\n");
    fprintf(xbytwo_mult, "\tend\n");


    // Final Result
    fprintf(xbytwo_mult, "\t// Final Result\n");
    fprintf(xbytwo_mult, "\talways@(posedge sys_clk)\n");
    fprintf(xbytwo_mult, "\tbegin\n");
    fprintf(xbytwo_mult, "\t\tif (sys_rst == 1'b1)\n");
    fprintf(xbytwo_mult, "\t\tbegin\n");
    fprintf(xbytwo_mult, "\t\t\tresult <= 0;\n");
    fprintf(xbytwo_mult, "\t\tend\n");
    fprintf(xbytwo_mult, "\t\telse\n");
    fprintf(xbytwo_mult, "\t\tbegin\n");
    fprintf(xbytwo_mult, "\t\t\tcase (multiplier_state_cs)\n");
    fprintf(xbytwo_mult, "\t\t\tFinal_Add_State:\n");
    fprintf(xbytwo_mult, "\t\t\t\tbegin\n");


    fprintf(xbytwo_mult, "\t\t\t\t\tresult <= ");
    if(num_points_per_stage[P_NUM_STAGES-2]==2){
        fprintf(xbytwo_mult, "partial_sum_%d_0+partial_sum_%d_1;\n", P_NUM_STAGES-2,P_NUM_STAGES-2);
    }else if (num_points_per_stage[P_NUM_STAGES-2]==3){
        fprintf(xbytwo_mult, "partial_sum_%d_0+partial_sum_%d_1+partial_sum_%d_2;\n",
            P_NUM_STAGES-2,P_NUM_STAGES-2,P_NUM_STAGES-2);
    }


    fprintf(xbytwo_mult, "\t\t\t\tend\n");
    fprintf(xbytwo_mult, "\t\t\tdefault:\n");
    fprintf(xbytwo_mult, "\t\t\tbegin\n");
    fprintf(xbytwo_mult, "\t\t\t\tresult <= result;\n");
    fprintf(xbytwo_mult, "\t\t\tend\n");
    fprintf(xbytwo_mult, "\t\t\tendcase\n");
    fprintf(xbytwo_mult, "\t\tend\n");
    fprintf(xbytwo_mult, "\tend\n\n");

/////////////////////////////////////////////////////////////////////////////////////
// Sub-Modules

    fprintf(xbytwo_mult, "/////////////////////////////////////////////////////////\n");
    fprintf(xbytwo_mult, "// Sub-Modules\n\n");

    for (i=0;i<P_NUM_MULTIPLIERS;i++){
        fprintf(xbytwo_mult, "\ttwo_bit_multiplier two_bit_multiplier_%d (\n",i);
        fprintf(xbytwo_mult, "\t.multiplicand(multiplicand_%d), \n",i);
        fprintf(xbytwo_mult, "\t.multiplier(multiplier_%d), \n",i);
        fprintf(xbytwo_mult, "\t.result(result_%d)\n",i);
        fprintf(xbytwo_mult, "\t);\n\n");
    }

    fprintf(xbytwo_mult, "endmodule");

    fclose(xbytwo_mult);
}

////////////////////////////////////////////////////////////////////////////////
// X BY X MULTIPLIER
//
////////////////////////////////////////////////////////////////////////////////

void gen_xbyx(FILE *xbyx_mult, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH){

    int P_NUM_STAGES;
    int P_NUM_MULTIPLIERS;
    int P_NUM_SUMS;
    int P_NUM_POINTS;
    int i,j,k;

    int num_points_per_stage[10];

    // Number of Stages
    if (P_MULTIPLICAND_WIDTH<4){
        P_NUM_STAGES = 0;
    }else if (P_MULTIPLICAND_WIDTH<8){
        P_NUM_STAGES = 1;
    }else if (P_MULTIPLICAND_WIDTH<14){
        P_NUM_STAGES = 2;
    }else if (P_MULTIPLICAND_WIDTH<26){
        P_NUM_STAGES = 3;
    }else if (P_MULTIPLICAND_WIDTH<48){
        P_NUM_STAGES = 4;
    }else if (P_MULTIPLICAND_WIDTH<98){
        P_NUM_STAGES = 5;
    }else if (P_MULTIPLICAND_WIDTH<=128){
        P_NUM_STAGES = 6;
    }

    P_NUM_MULTIPLIERS = P_MULTIPLICAND_WIDTH/2;

    ////////////////////////////////////////////////////////////////////////////
    // Instantiate Module

    fprintf(xbyx_mult, "`timescale 1ns / 1ps\n");
    fprintf(xbyx_mult, "/////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "//xbyx mult \n");
    fprintf(xbyx_mult, "//    by: Jason Lee \n");
    fprintf(xbyx_mult, "// \n");
    fprintf(xbyx_mult, "////////////////////////////////////////////////////////\n\n");

    if (P_VENDOR == 1){
        fprintf(xbyx_mult, "module xbyx_bit_multiplier(\n\n");
    }else{
        fprintf(xbyx_mult, "module xbyx_bit_multiplier_%d_%d(\n\n", P_FSL_WIDTH,
            P_MULTIPLICAND_WIDTH);
    }

    fprintf(xbyx_mult, "\tsys_clk,\n");
    fprintf(xbyx_mult, "\tsys_rst,\n\n");

    fprintf(xbyx_mult, "\tmultiplicand,\n");
    fprintf(xbyx_mult, "\tmultiplier,\n");
    fprintf(xbyx_mult, "\tresult,\n\n");

    fprintf(xbyx_mult, "\tinput_data_valid,\n");
    fprintf(xbyx_mult, "\tready_for_data,\n");
    fprintf(xbyx_mult, "\tdata_valid,\n");
    fprintf(xbyx_mult, "\ttarget_ready_for_data\n");
    fprintf(xbyx_mult, ");\n\n");


    //////////////////////////////////////////////////////////////////////////////////////
    // Parameters

    fprintf(xbyx_mult, "//////////////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// Parameters\n\n");

    fprintf(xbyx_mult, "\tlocalparam   P_MULTIPLIER_WIDTH\t= %d;\n", P_FSL_WIDTH);
    fprintf(xbyx_mult, "\tlocalparam   P_MULTIPLICAND_WIDTH\t= %d;\n", P_MULTIPLICAND_WIDTH);
    fprintf(xbyx_mult, "\tlocalparam   P_RESULT_WIDTH\t\t=
        P_MULTIPLIER_WIDTH+P_MULTIPLICAND_WIDTH;\n\n");

    //////////////////////////////////////////////////////////////////////////////////////
    // Ports

    fprintf(xbyx_mult, "//////////////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// Ports\n\n");

    fprintf(xbyx_mult, "\tinput\t\t\t\t\t\tsys_clk;\n");
    fprintf(xbyx_mult, "\tinput\t\t\t\t\t\tsys_rst;\n\n");

    fprintf(xbyx_mult, "\tinput\t\t[P_MULTIPLICAND_WIDTH-1:0]\t\tmultiplicand;\n");
    fprintf(xbyx_mult, "\tinput\t\t[P_MULTIPLIER_WIDTH-1:0]\t\tmultiplier;\n");
    fprintf(xbyx_mult, "\toutput\t[P_RESULT_WIDTH-1:0]\t\t\tresult;\n\n");

    fprintf(xbyx_mult, "\tinput\t\t\t\t\t\t\tinput_data_valid;\n");
    fprintf(xbyx_mult, "\toutput\t\t\t\t\t\tready_for_data;\n");
    fprintf(xbyx_mult, "\toutput\t\t\t\t\t\tdata_valid;\n\n");
    fprintf(xbyx_mult, "\tinput\t\t\t\t\t\ttarget_ready_for_data;\n");

    /////////////////////////////////////////////////////////////////////////////////
    // States

    fprintf(xbyx_mult, "////////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// States\n\n");

    fprintf(xbyx_mult, "\tlocalparam\tIdle_State\t\t= 0;\n");
    fprintf(xbyx_mult, "\tlocalparam\tMultiply_State\t\t= 1;\n");

    j = 2;
    for (i=0;i<P_NUM_STAGES-1;i++){
        fprintf(xbyx_mult, "\tlocalparam\tAdd_State_%d\t\t= %d;\n",i,j);
        j++;
    }

    fprintf(xbyx_mult, "\tlocalparam\tFinal_Add_State\t\t= %d;\n",j);
    fprintf(xbyx_mult, "\tlocalparam\tTransmit_State\t\t= %d;\n\n",j+1);

    /////////////////////////////////////////////////////////////////////////////////
    // Wires and Registers

    fprintf(xbyx_mult, "///////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// Wires and Registers\n\n");

    fprintf(xbyx_mult, "\t// State Machine\n");
    fprintf(xbyx_mult, "\treg\t\t\t[3:0]\t\t\tmultiplier_state_cs;\n");
    fprintf(xbyx_mult, "\treg\t\t\t[3:0]\t\t\tmultiplier_state_ns;\n\n");

    fprintf(xbyx_mult, "\t// Generate More for Longer Lengths\n");
    fprintf(xbyx_mult, "\t// Multipliers\n");
    for (i=0;i<P_NUM_MULTIPLIERS;i++){
        fprintf(xbyx_mult, "\twire\t\t\t\t\t\t\t\t\t\t\t\t\tdata_valid_%d;\n",i);
        fprintf(xbyx_mult, "\twire\t\t\t[(P_MULTIPLIER_WIDTH+2)-1:0]\t\t\tresult_%d;\n",i);
        fprintf(xbyx_mult, "\treg\t\t\t[(P_MULTIPLIER_WIDTH+2)-1:0]
            \t\t\tmultiplier_result_%d;\n\n",i);
    }

    for (i=0;i<P_NUM_MULTIPLIERS;i++){
        fprintf(xbyx_mult, "\twire\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tpartial_product_%d;\n",i);
    }
    fprintf(xbyx_mult, "\n");

    // Partial Sum
    P_NUM_POINTS = P_NUM_MULTIPLIERS;

    for(i=0;i<P_NUM_STAGES-1;i++){
        // Number of Sums to do for this stage
        P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
        if (P_NUM_POINTS%2 ==1){
            P_NUM_SUMS++;
        }

        // Keep track of number of sums for this stage
        P_NUM_POINTS = P_NUM_SUMS;
        num_points_per_stage[i] = P_NUM_POINTS;

        for(j=0;j<P_NUM_SUMS;j++){
            fprintf(xbyx_mult, "\treg\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tpartial_sum_%d_%d;\n",i,j);
        }
        fprintf(xbyx_mult, "\n");
    }

    fprintf(xbyx_mult, "\treg\t\t\t[P_RESULT_WIDTH-1:0]\t\t\tresult;\n\n");


    ///////////////////////////////////////////////////////////////////////////////////
    // Assigns

    fprintf(xbyx_mult, "///////////////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// Assigns\n\n");

    // Partial Products
    for(i=0;i<P_NUM_MULTIPLIERS;i++){

        if (i!=0){
            fprintf(xbyx_mult, "\tassign partial_product_%d[%d:0] = 0;\n", i, 2*i-1);
        }
        fprintf(xbyx_mult, "\tassign partial_product_%d[%d:%d] =
            multiplier_result_%d;\n", i,2*i+(P_FSL_WIDTH+1),2*i,i);

        if (i!=P_NUM_MULTIPLIERS-1){
            fprintf(xbyx_mult, "\tassign partial_product_%d[%d:%d] =
                0;\n",i,P_FSL_WIDTH+P_MULTIPLICAND_WIDTH-1,2*i+(P_FSL_WIDTH+2));
        }
        fprintf(xbyx_mult, "\n");
    }

    fprintf(xbyx_mult, "\t// Control Signals\n");
    fprintf(xbyx_mult, "\tassign data_valid = (multiplier_state_cs == Transmit_State);\n");
    fprintf(xbyx_mult, "\tassign ready_for_data = (multiplier_state_cs == Idle_State);\n\n");

    ///////////////////////////////////////////////////////////////////////////////////
    // State Machine

    fprintf(xbyx_mult, "////////////////////////////////////////////////////////////////\n");
    fprintf(xbyx_mult, "// State Machine\n\n");
```

```
// Register part of state machine
fprintf(xbyx_mult, "\t// Register part of state machine\n");
fprintf(xbyx_mult, "\talways @(posedge sys_clk)    \n");
fprintf(xbyx_mult, "\tbegin\n");
fprintf(xbyx_mult, "\t\tif (sys_rst == 1'b1) // reset active high\n");
fprintf(xbyx_mult, "\t\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\t\tmultiplier_state_cs <= Idle_State;\n");
fprintf(xbyx_mult, "\t\t\tend\n");
fprintf(xbyx_mult, "\t\telse\n");
fprintf(xbyx_mult, "\t\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\t\tmultiplier_state_cs <= multiplier_state_ns;\n");
fprintf(xbyx_mult, "\t\t\tend\n");
fprintf(xbyx_mult, "\tend\n\n");

fprintf(xbyx_mult, "\talways @(sys_rst or input_data_valid or
    multiplier_state_cs or data_valid_0)\n");
fprintf(xbyx_mult, "\tbegin\n");
fprintf(xbyx_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbyx_mult, "\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbyx_mult, "\t\tend\n");
fprintf(xbyx_mult, "\t\telse\n");
fprintf(xbyx_mult, "\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\tcase(multiplier_state_cs)\n");
fprintf(xbyx_mult, "\t\t\t\tIdle_State:\n");
fprintf(xbyx_mult, "\t\t\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\t\t\tif (input_data_valid == 1'b1)\n");
fprintf(xbyx_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Multiply_State;\n");
fprintf(xbyx_mult, "\t\t\t\t\telse\n");
fprintf(xbyx_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbyx_mult, "\t\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\t\tMultiply_State:\n");
fprintf(xbyx_mult, "\t\t\t\t\tif (data_valid_0)\n");
fprintf(xbyx_mult, "\t\t\t\t\tbegin\n");

if (P_NUM_STAGES <= 1){
    fprintf(xbyx_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Final_Add_State;\n");
}else{
    fprintf(xbyx_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Add_State_0;\n");
}
fprintf(xbyx_mult, "\t\t\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\t\t\telse\n");
fprintf(xbyx_mult, "\t\t\t\t\t\tmultiplier_state_ns <= Multiply_State; \n");

// Add Stages
for (i=0;i<P_NUM_STAGES-1;i++){
    if (i==P_NUM_STAGES-2){
        fprintf(xbyx_mult, "\t\t\t\tAdd_State_%d:\n",i);
        fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_state_ns <= Final_Add_State;\n");
    }else{
        fprintf(xbyx_mult, "\t\t\t\tAdd_State_%d:\n",i);
        fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_state_ns <= Add_State_%d;\n",i+1);
    }
}

fprintf(xbyx_mult, "\t\t\t\tFinal_Add_State:\n");
fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_state_ns <= Transmit_State;\n");
fprintf(xbyx_mult, "\t\t\t\tTransmit_State:\n");
fprintf(xbyx_mult, "\t\t\t\tif (target_ready_for_data)\n");
fprintf(xbyx_mult, "\t\t\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(xbyx_mult, "\t\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\t\telse\n");
fprintf(xbyx_mult, "\t\t\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_state_ns <= Transmit_State;\n");
fprintf(xbyx_mult, "\t\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\t\tdefault: multiplier_state_ns <= multiplier_state_ns;\n");
fprintf(xbyx_mult, "\t\t\tendcase\n");
fprintf(xbyx_mult, "\t\tend\n");
fprintf(xbyx_mult, "\tend\n\n");

// Multiplier Results
fprintf(xbyx_mult, "\t// Multiplier Results\n");
fprintf(xbyx_mult, "\t// Generate More for Longer Lengths\n");
fprintf(xbyx_mult, "\talways@(posedge sys_clk)\n");
fprintf(xbyx_mult, "\tbegin\n");
fprintf(xbyx_mult, "\t\tif (sys_rst == 1'b1)\n");
fprintf(xbyx_mult, "\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbyx_mult, "\t\t\tmultiplier_result_%d <= 0;\n",i);
}

fprintf(xbyx_mult, "\t\tend\n");
fprintf(xbyx_mult, "\t\telse\n");
fprintf(xbyx_mult, "\t\tbegin\n");
fprintf(xbyx_mult, "\t\t\tcase (multiplier_state_cs)\n");
fprintf(xbyx_mult, "\t\t\tMultiply_State:\n");
fprintf(xbyx_mult, "\t\t\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbyx_mult, "\t\t\t\t\tmultiplier_result_%d <= result_%d;\n",i,i);
}

fprintf(xbyx_mult, "\t\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\tdefault:\n");
fprintf(xbyx_mult, "\t\t\tbegin\n");

for (i=0;i<P_NUM_MULTIPLIERS;i++){
    fprintf(xbyx_mult, "\t\t\t\tmultiplier_result_%d <= multiplier_result_%d;\n",i,i);
}

fprintf(xbyx_mult, "\t\t\tend\n");
fprintf(xbyx_mult, "\t\t\tendcase\n");
fprintf(xbyx_mult, "\t\tend\n");
fprintf(xbyx_mult, "\tend\n\n");

if (P_NUM_STAGES > 1){

    // Partial Sums
    fprintf(xbyx_mult, "\t// Partial Sums\n");
    fprintf(xbyx_mult, "\talways@(posedge sys_clk)\n");
    fprintf(xbyx_mult, "\tbegin\n");
    fprintf(xbyx_mult, "\t\tif (sys_rst == 1'b1)\n");
    fprintf(xbyx_mult, "\t\tbegin\n");

    // Partial Sums
    P_NUM_POINTS = P_NUM_MULTIPLIERS;

    for(i=0;i<P_NUM_STAGES-1;i++){

        // Number of Sums to do for this stage
        P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
        if (P_NUM_POINTS%2 ==1){
            P_NUM_SUMS++;
        }

        // Keep track of number of sums for this stage
        P_NUM_POINTS = P_NUM_SUMS;
        num_points_per_stage[i] = P_NUM_POINTS;

        for(j=0;j<P_NUM_SUMS;j++){
            fprintf(xbyx_mult, "\t\t\tpartial_sum_%d_%d <= 0;\n",i,j);
        }
    }
    fprintf(xbyx_mult, "\n");


    fprintf(xbyx_mult, "\t\tend\n");
    fprintf(xbyx_mult, "\t\telse\n");
    fprintf(xbyx_mult, "\t\tbegin\n");
    fprintf(xbyx_mult, "\t\t\tcase (multiplier_state_cs)\n");
    for(i=0;i<P_NUM_STAGES-1;i++){

        fprintf(xbyx_mult, "\t\t\tAdd_State_%d:\n",i);
        fprintf(xbyx_mult, "\t\t\t\tbegin\n");

        if (i==0){
            for(j=0;j<num_points_per_stage[i];j++){
```

```
                if(j==num_points_per_stage[i]-1 && P_NUM_MULTIPLIERS%2!=0){
                    fprintf(xbyx_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_product_%d;\n",i,j,2*j);
                }else{
                    fprintf(xbyx_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_product_%d+partial_product_%d;\n",i,j,2*j,2*j+1);
                }
            }

        }else{
            for(j=0;j<num_points_per_stage[i];j++){
                if(j==num_points_per_stage[i]-1 && num_points_per_stage[i-1]%2!=0){
                    fprintf(xbyx_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_sum_%d_%d\n",i,j,i-1,2*j);
                }else{
                    fprintf(xbyx_mult, "\t\t\t\t\tpartial_sum_%d_%d <=
                        partial_sum_%d_%d+partial_sum_%d_%d;\n",i,j,i-1,2*j,i-1,2*j+1);
                }
            }
        }
        fprintf(xbyx_mult, "\t\t\t\tend\n");
    }
    fprintf(xbyx_mult, "\n");

    fprintf(xbyx_mult, "\t\t\tdefault:\n");
    fprintf(xbyx_mult, "\t\t\tbegin\n");

    // Partial Sums
    P_NUM_POINTS = P_NUM_MULTIPLIERS;

    for(i=0;i<P_NUM_STAGES-1;i++){

        // Number of Sums to do for this stage q
        P_NUM_SUMS = (P_NUM_POINTS-(P_NUM_POINTS%2))/2;
        if (P_NUM_POINTS%2 ==1){
            P_NUM_SUMS++;
        }

        // Keep track of number of sums for this stage
        P_NUM_POINTS = P_NUM_SUMS;
        num_points_per_stage[i] = P_NUM_POINTS;

        for(j=0;j<P_NUM_SUMS;j++){
            fprintf(xbyx_mult, "\t\t\t\tpartial_sum_%d_%d <= partial_sum_%d_%d;\n",i,j,i,j);
        }
    }
    fprintf(xbyx_mult, "\n");

    fprintf(xbyx_mult, "\t\t\tend\n");
    fprintf(xbyx_mult, "\t\t\tendcase\n");
```

```
        fprintf(xbyx_mult, "\t\tend\n");
        fprintf(xbyx_mult, "\tend\n");

    }

    // Final Result
    fprintf(xbyx_mult, "\t// Final Result\n");
    fprintf(xbyx_mult, "\talways@(posedge sys_clk)\n");
    fprintf(xbyx_mult, "\tbegin\n");
    fprintf(xbyx_mult, "\t\tif (sys_rst == 1'b1)\n");
    fprintf(xbyx_mult, "\t\tbegin\n");
    fprintf(xbyx_mult, "\t\t\tresult <= 0;\n");
    fprintf(xbyx_mult, "\t\tend\n");
    fprintf(xbyx_mult, "\t\telse\n");
    fprintf(xbyx_mult, "\t\tbegin\n");
    fprintf(xbyx_mult, "\t\t\tcase (multiplier_state_cs)\n");
    fprintf(xbyx_mult, "\t\t\tFinal_Add_State:\n");
    fprintf(xbyx_mult, "\t\t\t\tbegin\n");

    if (P_NUM_STAGES <= 1){
        fprintf(xbyx_mult, "\t\t\t\t\tresult <= ");
        if(P_NUM_MULTIPLIERS==1){
            fprintf(xbyx_mult, "partial_product_0;\n");
        }else if (P_NUM_MULTIPLIERS==2){
            fprintf(xbyx_mult, "partial_product_0 + partial_product_1;\n");
        }else if (P_NUM_MULTIPLIERS==3){
            fprintf(xbyx_mult, "partial_product_0 + partial_product_1+partial_product_2;\n",
                P_NUM_STAGES-2,P_NUM_STAGES-2,P_NUM_STAGES-2);
        }
    }else{
        fprintf(xbyx_mult, "\t\t\t\t\tresult <= ");
        if(num_points_per_stage[P_NUM_STAGES-2]==2){
            fprintf(xbyx_mult, "partial_sum_%d_0+partial_sum_%d_1;\n", P_NUM_STAGES-2,P_NUM_STAGES-2);
        }else if (num_points_per_stage[P_NUM_STAGES-2]==3){
            fprintf(xbyx_mult, "partial_sum_%d_0+partial_sum_%d_1+partial_sum_%d_2;\n",
                P_NUM_STAGES-2,P_NUM_STAGES-2,P_NUM_STAGES-2);
        }
    }

    fprintf(xbyx_mult, "\t\t\t\tend\n");
    fprintf(xbyx_mult, "\t\t\tdefault:\n");
    fprintf(xbyx_mult, "\t\t\tbegin\n");
    fprintf(xbyx_mult, "\t\t\t\tresult <= result;\n");
    fprintf(xbyx_mult, "\t\t\tend\n");
    fprintf(xbyx_mult, "\t\t\tendcase\n");
    fprintf(xbyx_mult, "\t\tend\n");
    fprintf(xbyx_mult, "\tend\n\n");

///////////////////////////////////////////////////////////////////////////////////
// Sub-Modules
```

```
    if(P_VENDOR == 1){
        fprintf(xbyx_mult, "/////////////////////////////////////////////////////\n");
        fprintf(xbyx_mult, "// Sub-Modules\n\n");

        for (i=0;i<P_NUM_MULTIPLIERS;i++){
            fprintf(xbyx_mult, "\txbytwo_bit_multiplier xbytwo_bit_multiplier_%d (\n",i);
            fprintf(xbyx_mult, "\t.sys_clk(sys_clk), \n");
            fprintf(xbyx_mult, "\t.sys_rst(sys_rst), \n");
            fprintf(xbyx_mult, "\t.multiplicand(multiplicand[%d:%d]),\n", 2*i+1, 2*i);
            fprintf(xbyx_mult, "\t.multiplier(multiplier),              // from FSL\n");
            fprintf(xbyx_mult, "\t.result(result_%d), \n",i);
            fprintf(xbyx_mult, "\t.input_data_valid(input_data_valid), \n");
            fprintf(xbyx_mult, "\t.data_valid(data_valid_%d)\n",i);
            fprintf(xbyx_mult, "\t);\n\n");
        }
        fprintf(xbyx_mult, "endmodule");
    }else{
        fprintf(xbyx_mult, "/////////////////////////////////////////////////////\n");
        fprintf(xbyx_mult, "// Sub-Modules\n\n");

        for (i=0;i<P_NUM_MULTIPLIERS;i++){
            fprintf(xbyx_mult, "\txbytwo_bit_multiplier_%d_%d xbytwo_bit_multiplier_%d_%d_%d
                (\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,i);
            fprintf(xbyx_mult, "\t.sys_clk(sys_clk), \n");
            fprintf(xbyx_mult, "\t.sys_rst(sys_rst), \n");
            fprintf(xbyx_mult, "\t.multiplicand(multiplicand[%d:%d]),\n", 2*i+1, 2*i);
            fprintf(xbyx_mult, "\t.multiplier(multiplier),              // from FSL\n");
            fprintf(xbyx_mult, "\t.result(result_%d), \n",i);
            fprintf(xbyx_mult, "\t.input_data_valid(input_data_valid), \n");
            fprintf(xbyx_mult, "\t.data_valid(data_valid_%d)\n",i);
            fprintf(xbyx_mult, "\t);\n\n");
        }
        fprintf(xbyx_mult, "endmodule");
    }
    fclose(xbyx_mult);

}

////////////////////////////////////////////////////////////////////////////////
// MULTIPLIER
//
////////////////////////////////////////////////////////////////////////////////

void gen_multiplier(FILE *mult, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH, int init_multiplier){

    ////////////////////////////////////////////////////////////////////////////
    // Instantiate Module

    fprintf(mult, "‘timescale 1ns / 1ps\n");
    fprintf(mult, "////////////////////////////////////////////////////////////\n");
    fprintf(mult, "// multiplier \n");
    fprintf(mult, "//    by: Jason Lee \n");
    fprintf(mult, "// \n");
    fprintf(mult, "//////////////////////////////////////////////////////////\n\n");

    if (init_multiplier == 1){
        fprintf(mult, "module init_multiplier_%d_%d(\n\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }else{
        fprintf(mult, "module multiplier_%d_%d(\n\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }

    fprintf(mult, "\t FSL_Clk,\n");
    fprintf(mult, "\t FSL_Rst,\n\n");

    fprintf(mult, "\t FSL_S_Clk,\n");
    fprintf(mult, "\t FSL_S_Exists,\n");
    fprintf(mult, "\t FSL_S_Read,\n");
    fprintf(mult, "\t FSL_S_Data,\n");
    fprintf(mult, "\t FSL_S_Control,\n\n");

    fprintf(mult, "\t FSL_M_Clk,\n");
    fprintf(mult, "\t FSL_M_Full,\n");
    fprintf(mult, "\t FSL_M_Write,\n");
    fprintf(mult, "\t FSL_M_Data,\n");

    if (init_multiplier == 1){
        fprintf(mult, "\t FSL_M_Control,\n\n");
        fprintf(mult, "\t output_result,\n");
        fprintf(mult, "\t multiplicand\n\n");
    }else{
        fprintf(mult, "\t FSL_M_Control\n\n");
    }

    fprintf(mult, ");\n\n");

    //////////////////////////////////////////////////////////////////////////////////
    // Parameters

    fprintf(mult, "////////////////////////////////////////////////////////////////\n");
    fprintf(mult, "// Parameters\n\n");

    fprintf(mult, "\tlocalparam P_FSL_WIDTH = %d;\n",P_FSL_WIDTH);
    fprintf(mult, "\tlocalparam P_MULTIPLIER_WIDTH = %d;// from FSL\n",P_FSL_WIDTH);
    fprintf(mult, "\tlocalparam P_MULTIPLICAND_WIDTH = %d;\n",P_MULTIPLICAND_WIDTH);
    fprintf(mult, "\tlocalparam P_RESULT_WIDTH = P_MULTIPLIER_WIDTH+P_MULTIPLICAND_WIDTH;\n\n");

    //////////////////////////////////////////////////////////////////////////////////
    // Ports
```

```
fprintf(mult, "////////////////////////////////////////////////////////////////\n");
fprintf(mult, "// Ports\n\n");

fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_Clk;\n");
fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_Rst;\n\n");

fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_S_Clk;\n");
fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_S_Exists;\n");
fprintf(mult, "\toutput\t\t\t\t\t\t\t\tFSL_S_Read;\n");
fprintf(mult, "\tinput\t\t\t\t[%d:0]\t\t\t\tFSL_S_Data;\n",P_FSL_WIDTH-1);
fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_S_Control;\n\n");

fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_M_Clk;\n");
fprintf(mult, "\tinput\t\t\t\t\t\t\t\tFSL_M_Full;\n");
fprintf(mult, "\toutput\t\t\t\t\t\t\t\tFSL_M_Write;\n");
fprintf(mult, "\toutput\t\t\t\t[%d:0]\t\t\t\tFSL_M_Data;\n",P_FSL_WIDTH-1);
fprintf(mult, "\toutput\t\t\t\t\t\t\t\tFSL_M_Control; \n\n");

if (init_multiplier == 1){
    fprintf(mult, "\toutput\t\t\t\t[%d:0]\t\t\t\toutput_result;\n",P_FSL_WIDTH-1);
    fprintf(mult, "\tinput\t\t\t\t[%d:0]\t\t\t\tmultiplicand;\n",P_MULTIPLICAND_WIDTH-1);
}

/////////////////////////////////////////////////////////////////////////////////
// States

fprintf(mult, "//////////////////////////////////////////////////////////////////\n");
fprintf(mult, "// States\n\n");

fprintf(mult, "\tlocalparam Idle_State\t\t= 0;\n");
fprintf(mult, "\tlocalparam Load_State\t\t= 1;\n");
fprintf(mult, "\tlocalparam Calculate_State\t= 2;\n");
fprintf(mult, "\tlocalparam Transmit_State\t= 3;\n\n");

/////////////////////////////////////////////////////////////////////////////////
// Wires and Registers

fprintf(mult, "/////////////////////////////////////////////////////////////////\n");
fprintf(mult, "// Wires and Registers\n\n");

fprintf(mult, "\t// State Machine\n");
fprintf(mult, "\treg\t\t\t[1:0]\t\tmultiplier_state_cs;\n");
fprintf(mult, "\treg\t\t\t[1:0]\t\tmultiplier_state_ns;\n\n");


fprintf(mult, "\t// Multiplier\n");
if (init_multiplier == 0){
   fprintf(mult, "\treg\t\t\t[P_MULTIPLICAND_WIDTH-1:0]\t\tmultiplicand;\n");
}
fprintf(mult, "\treg\t\t\t[P_MULTIPLIER_WIDTH-1:0]\t\tmultiplier;\n\n");
fprintf(mult, "\twire\t\t\t\t\t\t\t\tinput_data_valid;\n");
fprintf(mult, "\twire\t\t\t\t\t\t\t\tready_for_data;\n");
fprintf(mult, "\twire\t\t\t\t\t\t\t\tdata_valid;\n");
fprintf(mult, "\twire\t\t\t[P_RESULT_WIDTH-1:0]\t\tresult;\n\n");

fprintf(mult, "\treg\t\t\t[%d:0]\t\t\tFSL_M_Data;\n\n", P_FSL_WIDTH-1);

////////////////////////////////////////////////////////////////////////////////
// Assigns

fprintf(mult, "//////////////////////////////////////////////////////////////////////////\n");
fprintf(mult, "// Assigns\n\n");

fprintf(mult, "\tassign FSL_S_Read = (multiplier_state_cs ==
    Idle_State && FSL_S_Exists == 1'b1);\n");
fprintf(mult, "\tassign FSL_M_Write = (multiplier_state_cs ==
    Transmit_State && FSL_M_Full == 1'b0);\n\n");

fprintf(mult, "\tassign input_data_valid = (multiplier_state_cs == Load_State);\n");
fprintf(mult, "\tassign target_ready_for_data = 1'b1; // not used for now\n\n");

if (init_multiplier == 1){
   fprintf(mult, "\tassign  output_result = FSL_M_Data; // Output pin from system\n\n");
}
if (P_VENDOR == 0){
   fprintf(mult, "\tassign  FSL_M_Control = (multiplier_state_cs == Calculate_State);\n\n");
}

////////////////////////////////////////////////////////////////////////////////
// State Machine

fprintf(mult, "//////////////////////////////////////////////////////////////////////////\n");
fprintf(mult, "// State Machine\n\n");

fprintf(mult, "\t// Register part of state machine\n");
fprintf(mult, "\talways @(posedge FSL_Clk)  \n");
fprintf(mult, "\tbegin\n");
fprintf(mult, "\t\tif (FSL_Rst == 1'b1) // reset active high\n");
fprintf(mult, "\t\t\tbegin\n");
fprintf(mult, "\t\t\t\tmultiplier_state_cs <= Idle_State;\n");
fprintf(mult, "\t\t\tend\n");
fprintf(mult, "\t\telse\n");
fprintf(mult, "\t\t\tbegin\n");
fprintf(mult, "\t\t\t\tmultiplier_state_cs <= multiplier_state_ns;\n");
fprintf(mult, "\t\t\tend\n");
fprintf(mult, "\tend\n\n");

// State Machine
fprintf(mult, "\t// State Machine\n");
fprintf(mult, "\talways @(FSL_Rst, FSL_S_Exists, ready_for_data, data_valid,
    FSL_M_Full, multiplier_state_cs)\n");
fprintf(mult, "\tbegin\n");
fprintf(mult, "\t\tif (FSL_Rst == 1'b1)\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\t\telse\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tcase(multiplier_state_cs)\n");
fprintf(mult, "\t\t\t\tIdle_State:\n");
fprintf(mult, "\t\t\t\tbegin\n");
fprintf(mult, "\t\t\t\t\tif (FSL_S_Exists == 1'b1)\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Load_State;\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(mult, "\t\t\t\tend\n");
fprintf(mult, "\t\t\t\tLoad_State:\n");
fprintf(mult, "\t\t\t\t\tif (ready_for_data == 1'b1)\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Calculate_State;\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Load_State;\n");
fprintf(mult, "\t\t\t\tCalculate_State:\n");
fprintf(mult, "\t\t\t\tbegin\n");
fprintf(mult, "\t\t\t\t\tif (data_valid == 1'b1)\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Transmit_State;\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Calculate_State;\n");
fprintf(mult, "\t\t\t\tend\n");
fprintf(mult, "\t\t\t\tTransmit_State:\n");
fprintf(mult, "\t\t\t\tbegin\n");
fprintf(mult, "\t\t\t\t\tif (FSL_M_Full == 1'b0)\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Idle_State;\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier_state_ns <= Transmit_State;\n");
fprintf(mult, "\t\t\t\tend\n");
fprintf(mult, "\t\t\t\tdefault: multiplier_state_ns <= multiplier_state_ns;\n");
fprintf(mult, "\t\t\tendcase\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\tend\n\n");

if (init_multiplier == 0){
// multiplicand
    fprintf(mult, "\t// multiplicand\n");
    fprintf(mult, "\talways@(posedge FSL_Clk)\n");
    fprintf(mult, "\tbegin\n");
    fprintf(mult, "\t\tif (FSL_Rst == 1'b1)\n");
    fprintf(mult, "\t\tbegin\n");
    fprintf(mult, "\t\t\tmultiplicand <= 7; // some arbitrary value\n");
    fprintf(mult, "\t\tend\n");
    fprintf(mult, "\t\telse\n");
    fprintf(mult, "\t\tbegin\n");
    fprintf(mult, "\t\t\tcase (multiplier_state_cs)\n");
    fprintf(mult, "\t\t\tCalculate_State:\n");
    fprintf(mult, "\t\t\t\tbegin\n");
    fprintf(mult, "\t\t\t\t\tif (data_valid == 1'b1)\n");
    fprintf(mult, "\t\t\t\t\t\tif (result[P_RESULT_WIDTH-1:P_RESULT_WIDTH-P_MULTIPLICAND_WIDTH]
        == 0)   // to prevent multiplicand from going to zero\n");
    fprintf(mult, "\t\t\t\t\t\t\tmultiplicand <= multiplicand - 1;\n");
    fprintf(mult, "\t\t\t\t\t\telse\n");
    fprintf(mult, "\t\t\t\t\t\t\tmultiplicand <= result[P_RESULT_WIDTH-1:
        P_RESULT_WIDTH-P_MULTIPLICAND_WIDTH];\n");
    fprintf(mult, "\t\t\t\t\telse\n");
    fprintf(mult, "\t\t\t\t\t\tmultiplicand <= multiplicand;\n");
    fprintf(mult, "\t\t\t\tend\n");
    fprintf(mult, "\t\t\tdefault:\n");
    fprintf(mult, "\t\t\tbegin\n");
    fprintf(mult, "\t\t\t\tmultiplicand <= multiplicand;\n");
    fprintf(mult, "\t\t\tend\n");
    fprintf(mult, "\t\t\tendcase\n");
    fprintf(mult, "\t\tend\n");
    fprintf(mult, "\tend\n\n");
}

// multiplier
fprintf(mult, "\t// multiplier\n");
fprintf(mult, "\talways@(posedge FSL_Clk)\n");
fprintf(mult, "\tbegin\n");
fprintf(mult, "\t\tif (FSL_Rst == 1'b1)\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tmultiplier <= 0;   // some arbitrary value\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\t\telse\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tcase (multiplier_state_cs)\n");
fprintf(mult, "\t\t\tIdle_State:\n");
fprintf(mult, "\t\t\t\tbegin\n");
fprintf(mult, "\t\t\t\t\tif (FSL_S_Exists == 1'b1)\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier <= FSL_S_Data;\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tmultiplier <= multiplier;\n");
fprintf(mult, "\t\t\t\tend\n");
fprintf(mult, "\t\t\tdefault:\n");
fprintf(mult, "\t\t\tbegin\n");
fprintf(mult, "\t\t\t\tmultiplier <= multiplier;\n");
fprintf(mult, "\t\t\tend\n");
fprintf(mult, "\t\t\tendcase\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\tend\n\n");
```

```
// FSL_M_Data
fprintf(mult, "\t// FSL_M_Data\n");
fprintf(mult, "\talways@(posedge FSL_Clk)\n");
fprintf(mult, "\tbegin\n");
fprintf(mult, "\t\tif (FSL_Rst == 1'b1)\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tFSL_M_Data <= 0;   // some arbitrary value\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\t\telse\n");
fprintf(mult, "\t\tbegin\n");
fprintf(mult, "\t\t\tcase (multiplier_state_cs)\n");
fprintf(mult, "\t\t\tCalculate_State:\n");
fprintf(mult, "\t\t\t\tbegin\n");
fprintf(mult, "\t\t\t\t\tif (data_valid == 1'b1)\n");
fprintf(mult, "\t\t\t\t\t\tFSL_M_Data <= result[P_FSL_WIDTH-1:0];\n");
fprintf(mult, "\t\t\t\t\telse\n");
fprintf(mult, "\t\t\t\t\t\tFSL_M_Data <= FSL_M_Data;\n");
fprintf(mult, "\t\t\t\tend\n");
fprintf(mult, "\t\t\tdefault:\n");
fprintf(mult, "\t\t\tbegin\n");
fprintf(mult, "\t\t\t\tFSL_M_Data <= FSL_M_Data;\n");
fprintf(mult, "\t\t\tend\n");
fprintf(mult, "\t\t\tendcase\n");
fprintf(mult, "\t\tend\n");
fprintf(mult, "\tend\n\n");

///////////////////////////////////////////////////////////////////////////////////////
// Sub-Modules

if(P_VENDOR==1){
    fprintf(mult, "///////////////////////////////////////////////////////////////\n");
    fprintf(mult, "// Sub-Modules\n\n");

    fprintf(mult, "\txbyx_bit_multiplier xbyx_bit_multiplier (\n");
    fprintf(mult, "\t .sys_clk(FSL_Clk), \n");
    fprintf(mult, "\t .sys_rst(FSL_Rst), \n");
    fprintf(mult, "\t .multiplicand(multiplicand), \n");
    fprintf(mult, "\t .multiplier(multiplier), \n");
    fprintf(mult, "\t .result(result), \n");
    fprintf(mult, "\t .input_data_valid(input_data_valid), \n");
    fprintf(mult, "\t .ready_for_data(ready_for_data), \n");
    fprintf(mult, "\t .data_valid(data_valid), \n");
    fprintf(mult, "\t .target_ready_for_data(target_ready_for_data)\n");
    fprintf(mult, "\t);\n\n");

    fprintf(mult, "endmodule\n");
}else{

    fprintf(mult, "///////////////////////////////////////////////////////////////\n");
    fprintf(mult, "// Sub-Modules\n\n");
```

```
        fprintf(mult, "\txbyx_bit_multiplier_%d_%d xbyx_bit_multiplier_%d_%d
            (\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(mult, "\t .sys_clk(FSL_Clk), \n");
        fprintf(mult, "\t .sys_rst(FSL_Rst), \n");
        fprintf(mult, "\t .multiplicand(multiplicand), \n");
        fprintf(mult, "\t .multiplier(multiplier), \n");
        fprintf(mult, "\t .result(result), \n");
        fprintf(mult, "\t .input_data_valid(input_data_valid), \n");
        fprintf(mult, "\t .ready_for_data(ready_for_data), \n");
        fprintf(mult, "\t .data_valid(data_valid), \n");
        fprintf(mult, "\t .target_ready_for_data(target_ready_for_data)\n");
        fprintf(mult, "\t);\n\n");

        fprintf(mult, "endmodule\n");
    }

    fclose(mult);
}

////////////////////////////////////////////////////////////////////////////////
// MULTIPLIER DATA FILES
//
////////////////////////////////////////////////////////////////////////////////

void gen_mult_data(FILE *multiplier_mpd, FILE *multiplier_pao, int P_FSL_WIDTH,
        int P_MULTIPLICAND_WIDTH, int init_multiplier){

    // MPD FILE
    fprintf(multiplier_mpd, "############################################################\n");
    fprintf(multiplier_mpd, "## MPD FILE\n");
    fprintf(multiplier_mpd, "##\n");
    fprintf(multiplier_mpd, "##########################################################\n\n");

    if (init_multiplier == 1){
        fprintf(multiplier_mpd, "BEGIN init_multiplier_%d_%d\n\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }else{
        fprintf(multiplier_mpd, "BEGIN multiplier_%d_%d\n\n", P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }

    fprintf(multiplier_mpd, "## Peripheral Options\n");
    fprintf(multiplier_mpd, "OPTION IPTYPE = PERIPHERAL\n");
    fprintf(multiplier_mpd, "OPTION IMP_NETLIST = TRUE\n");
    fprintf(multiplier_mpd, "OPTION HDL = VERILOG\n");
    fprintf(multiplier_mpd, "OPTION IP_GROUP = MICROBLAZE:PPC:USER\n\n");


    fprintf(multiplier_mpd, "## Bus Interfaces\n");
    fprintf(multiplier_mpd, "BUS_INTERFACE BUS = SFSL, BUS_TYPE = SLAVE, BUS_STD = FSL\n");
    fprintf(multiplier_mpd, "BUS_INTERFACE BUS = MFSL, BUS_TYPE = MASTER, BUS_STD = FSL\n\n");
    fprintf(multiplier_mpd, "## Generics for VHDL or Parameters for Verilog\n\n");

    fprintf(multiplier_mpd, "## Ports\n");
    fprintf(multiplier_mpd, "PORT FSL_Clk = "", DIR = I, BUS = SFSL:MFSL, SIGIS = CLK\n");
    fprintf(multiplier_mpd, "PORT FSL_Rst = OPB_Rst, DIR = I, BUS = SFSL:MFSL, SIGIS = RST\n");
    fprintf(multiplier_mpd, "PORT FSL_S_Clk = FSL_S_Clk, DIR = I, BUS = SFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_S_Exists = FSL_S_Exists, DIR = I, BUS = SFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_S_Read = FSL_S_Read, DIR = O, BUS = SFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_S_Data = FSL_S_Data, DIR = I, VEC = [%d:0],
        BUS = SFSL\n", P_FSL_WIDTH-1);
    fprintf(multiplier_mpd, "PORT FSL_S_Control = FSL_S_Control, DIR = I, BUS = SFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_M_Clk = FSL_M_Clk, DIR = I, BUS = MFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_M_Full = FSL_M_Full, DIR = I, BUS = MFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_M_Write = FSL_M_Write, DIR = O, BUS = MFSL\n");
    fprintf(multiplier_mpd, "PORT FSL_M_Data = FSL_M_Data, DIR = O, VEC = [%d:0],
        BUS = MFSL\n", P_FSL_WIDTH-1);
    fprintf(multiplier_mpd, "PORT FSL_M_Control = FSL_M_Control, DIR = O, BUS = MFSL\n\n");
    if (init_multiplier == 1){
        fprintf(multiplier_mpd, "PORT output_result = output_result, DIR = O, VEC =
            [%d:0]\n", P_FSL_WIDTH-1);
        fprintf(multiplier_mpd, "PORT multiplicand = multiplicand, DIR = I, VEC =
            [%d:0]\n\n", P_MULTIPLICAND_WIDTH-1);
    }

    fprintf(multiplier_mpd, "END\n\n");

    // PAO FILE

    fprintf(multiplier_pao, "##############################################################\n");
    fprintf(multiplier_pao, "## PAO FILE\n");
    fprintf(multiplier_pao, "##\n");
    fprintf(multiplier_pao, "##############################################################\n\n");

    if (init_multiplier == 1){
        fprintf(multiplier_pao, "lib init_multiplier_%d_%d_v1_00_a init_multiplier_%d_%d verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib init_multiplier_%d_%d_v1_00_a two_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib init_multiplier_%d_%d_v1_00_a xbytwo_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib init_multiplier_%d_%d_v1_00_a xbyx_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }else{
        fprintf(multiplier_pao, "lib multiplier_%d_%d_v1_00_a multiplier_%d_%d verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib multiplier_%d_%d_v1_00_a two_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib multiplier_%d_%d_v1_00_a xbytwo_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
        fprintf(multiplier_pao, "lib multiplier_%d_%d_v1_00_a xbyx_bit_multiplier verilog\n",
            P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
    }
}
```

B.4 Generate Switch PCORE (generate_switch.c)

The following functions are used to generate the network switch for each network node.

```
#include <stdio.h>
#include "globals.h"

// Generate MPD File
void generate_switch_mpd(int P_NODE_SIZE, FILE *switch_mpd, int P_FSL_WIDTH){

    int i;

    //////////////////////////////////////////////////////////////////////////////
    // Generate Switch MPD File
    // switch.mpd
    // Upper Level Wrapper for Switch Node
    //////////////////////////////////////////////////////////////////////////////

    fprintf (switch_mpd, "#############################################################\n");
    fprintf (switch_mpd, "##\n");
    fprintf (switch_mpd, "## Name     : switch%d\n", P_NODE_SIZE);
    fprintf (switch_mpd, "## Desc     : Microprocessor Peripheral Description\n");
    fprintf (switch_mpd, "##          : Automatically generated by PsfUtility\n");
    fprintf (switch_mpd, "##\n");
    fprintf (switch_mpd, "############################################################\n\n");

    fprintf (switch_mpd, "BEGIN switch%d\n\n", P_NODE_SIZE);

    // Peripheral Options
    fprintf (switch_mpd, "## Peripheral Options\n");
    fprintf (switch_mpd, "OPTION IPTYPE = PERIPHERAL\n");
    fprintf (switch_mpd, "OPTION IMP_NETLIST = TRUE\n");
    fprintf (switch_mpd, "OPTION HDL = VERILOG\n");
    fprintf (switch_mpd, "OPTION IP_GROUP = MICROBLAZE:PPC:USER\n\n\n");

    // Bus Interface
    fprintf (switch_mpd, "## Bus Interfaces\n");
    fprintf (switch_mpd, "BUS_INTERFACE BUS = SFSL, BUS_TYPE = SLAVE, BUS_STD = FSL\n");
    fprintf (switch_mpd, "BUS_INTERFACE BUS = MFSL, BUS_TYPE = MASTER, BUS_STD = FSL\n\n");

    // Parameters
    fprintf (switch_mpd, "## Generics for VHDL or Parameters for Verilog\n");
    fprintf (switch_mpd, "PARAMETER C_DEST_L = 0, DT = integer\n");
    fprintf (switch_mpd, "PARAMETER C_DEST_H = 0, DT = integer\n\n");
    // Ports
    fprintf (switch_mpd, "## Ports\n");
    fprintf (switch_mpd, "PORT FSL_Clk = \"\", DIR = I, BUS = SFSL:MFSL, SIGIS = CLK\n");
    fprintf (switch_mpd, "PORT FSL_Rst = OPB_Rst, DIR = I, BUS = SFSL:MFSL, SIGIS = RST\n");
    fprintf (switch_mpd, "PORT FSL_S_Clk = FSL_S_Clk, DIR = I, BUS = SFSL\n");
    fprintf (switch_mpd, "PORT FSL_M_Clk = FSL_M_Clk, DIR = I, BUS = MFSL\n");
    fprintf (switch_mpd, "PORT FSL_M_Data = FSL_M_Data, DIR = O, VEC = [%d:0],
        BUS = MFSL\n", P_FSL_WIDTH-1);
    fprintf (switch_mpd, "PORT FSL_M_Control = FSL_M_Control, DIR = O, BUS = MFSL\n");
    fprintf (switch_mpd, "PORT FSL_M_Write = FSL_M_Write, DIR = O, BUS = MFSL\n");
    fprintf (switch_mpd, "PORT FSL_M_Full = FSL_M_Full, DIR = I, BUS = MFSL\n");
    fprintf (switch_mpd, "PORT FSL_S_Data = FSL_S_Data, DIR = I, VEC = [%d:0],
        BUS = SFSL\n", P_FSL_WIDTH-1);
    fprintf (switch_mpd, "PORT FSL_S_Control = FSL_S_Control, DIR = I, BUS = SFSL\n");
    fprintf (switch_mpd, "PORT FSL_S_Read = FSL_S_Read, DIR = O, BUS = SFSL\n");
    fprintf (switch_mpd, "PORT FSL_S_Exists = FSL_S_Exists, DIR = I, BUS = SFSL\n");

    for (i=0;i<P_NODE_SIZE;i++){
        fprintf (switch_mpd, "PORT ch%d_in_data = "", DIR = I, VEC = [%d:0]\n",i, P_FSL_WIDTH-1);
        fprintf (switch_mpd, "PORT ch%d_in_ctrl = "", DIR = I\n",i);
        fprintf (switch_mpd, "PORT ch%d_in_exists = "", DIR = I\n",i);
        fprintf (switch_mpd, "PORT ch%d_in_read = "", DIR = O\n",i);
        fprintf (switch_mpd, "PORT ch%d_out_data = "", DIR = O, VEC = [%d:0]\n",i, P_FSL_WIDTH-1);
        fprintf (switch_mpd, "PORT ch%d_out_ctrl = "", DIR = O\n",i);
        fprintf (switch_mpd, "PORT ch%d_out_exists = "", DIR = O\n",i);
        fprintf (switch_mpd, "PORT ch%d_out_read = "", DIR = I\n\n",i);
    }
    fprintf (switch_mpd, "END\n\n");
}

////////////////////////////////////////////////////
// Generate Switch File
// Upper Level Wrapper for Switch Node

void generate_switch_v(int P_NODE_SIZE, FILE *v_switch, int P_FSL_WIDTH){
    int i;

    fprintf(v_switch, "‘timescale 1ns / 1ps\n");

    fprintf(v_switch, "////////////////////////////////////////////////////\n");
    fprintf(v_switch, "// Switch\n");
    fprintf(v_switch, "// # of Channels: %d\n", P_NODE_SIZE);
    fprintf(v_switch, "// \n");
    fprintf(v_switch, "////////////////////////////////////////////////////\n\n");

    ///////////////////////////////////////////////////////////////////////////
    // MODULE DEFINITION

    if(P_VENDOR == 1){
        fprintf(v_switch, "module switch%d(\n\n",P_NODE_SIZE);
    }else{
        fprintf(v_switch, "module switch_%d(\n\n",P_NODE_SIZE);
    }

    // single wire definition
    fprintf(v_switch, "\tFSL_Clk,\n");
    fprintf(v_switch, "\tFSL_Rst,\n\n");

    fprintf(v_switch, "\tFSL_S_Clk,\n");
    fprintf(v_switch, "\tFSL_M_Clk,\n\n");

    // Interface to MicroBlaze
    fprintf(v_switch, "\t// MicroBlaze Interface\n");
    fprintf(v_switch, "\tFSL_M_Data,\n");
    fprintf(v_switch, "\tFSL_M_Control,\n");
    fprintf(v_switch, "\tFSL_M_Write,\n");
    fprintf(v_switch, "\tFSL_M_Full,\n\n");

    fprintf(v_switch, "\tFSL_S_Data,\n");
    fprintf(v_switch, "\tFSL_S_Control,\n");
    fprintf(v_switch, "\tFSL_S_Read,\n");
    fprintf(v_switch, "\tFSL_S_Exists,\n\n");

    // Number of Channels
    for (i=0; i<P_NODE_SIZE; i++){
        fprintf(v_switch, "\t// Channel %d Interface\n", i);
        fprintf(v_switch, "\tch%d_in_data,\n", i);
        fprintf(v_switch, "\tch%d_in_ctrl,\n", i);
        fprintf(v_switch, "\tch%d_in_exists,\n", i);
        fprintf(v_switch, "\tch%d_in_read,\n\n", i);

        fprintf(v_switch, "\tch%d_out_data,\n", i);
        fprintf(v_switch, "\tch%d_out_ctrl,\n", i);
        fprintf(v_switch, "\tch%d_out_exists,\n", i);
        if (i == P_NODE_SIZE-1){                    // Last One
            fprintf(v_switch, "\tch%d_out_read\n\n", i);
        }
        else{
            fprintf(v_switch, "\tch%d_out_read,\n\n", i);
        }
    }

    fprintf(v_switch, ");\n\n");

    ////////////////////////////////////////////////////////////////////////////////
    // PARAMETERS

    fprintf(v_switch, "//////////////////////////////////////////////////////////\n");
    fprintf(v_switch, "// PARAMETERS\n\n");


    fprintf(v_switch, "\tlocalparam P_DATA_WIDTH = 32;\n\n");

    ////////////////////////////////////////////////////////////////////////////////
    // PORTS
```

```
fprintf(v_switch, "//////////////////////////////////////////////////////\n");
fprintf(v_switch, "// PORTS\n\n");

fprintf(v_switch, "\tparameter\t\t\t\t\t\t\t\t\t\tC_DEST_L = 0;\n");
fprintf(v_switch, "\tparameter\t\t\t\t\t\t\t\t\t\tC_DEST_H = 1;\n");

fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_Clk;\n");
fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_Rst;\n\n");

fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_S_Clk;\n");
fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_M_Clk;\n\n");

fprintf(v_switch, "\t// Input Data to MicroBlaze\n");
fprintf(v_switch, "\t// Writes Data to FSL connecting to MicroBlaze if data
                    on Switch is valid\n");
fprintf(v_switch, "\toutput\t\t[%d:0]\t\tFSL_M_Data;\n", P_FSL_WIDTH-1);
fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tFSL_M_Control;\n");
fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tFSL_M_Write;\n");
fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_M_Full;\n\n");

fprintf(v_switch, "\t// Output Data from MicroBlaze\n");
fprintf(v_switch, "\t// Reads Data from FSL Connecting to MicroBlaze
                    to all Channels\n");
fprintf(v_switch, "\tinput\t\t\t[%d:0]\t\tFSL_S_Data;\n", P_FSL_WIDTH-1);
fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_S_Control;\n");
fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tFSL_S_Read;\n");
fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tFSL_S_Exists;\n\n");

for (i=0;i<P_NODE_SIZE;i++){
    fprintf(v_switch, "\t// Channel %d\n",i);
    fprintf(v_switch, "\t// Input Data to MicroBlaze, reads data from other
            MicroBlaze's FSL\n");
    fprintf(v_switch, "\tinput\t\t[%d:0]\t\tch%d_in_data;\n", P_FSL_WIDTH-1,i);
    fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tch%d_in_ctrl;\n",i);
    fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tch%d_in_exists;\n",i);
    fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tch%d_in_read;\n\n",i);

    fprintf(v_switch, "\t// Output Data from MicroBlaze, reads data from
            MicroBlaze's FSL\n");
    fprintf(v_switch, "\toutput\t[%d:0]\t\tch%d_out_data;\n", P_FSL_WIDTH-1,i);
    fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tch%d_out_ctrl;\n",i);
    fprintf(v_switch, "\toutput\t\t\t\t\t\t\t\t\tch%d_out_exists;\n",i);
    fprintf(v_switch, "\tinput\t\t\t\t\t\t\t\t\t\tch%d_out_read;\n\n",i);
}

////////////////////////////////////////////////////////////////////////////////
// WIRES/REGISTERS

fprintf(v_switch, "//////////////////////////////////////////////////////\n");
fprintf(v_switch, "// WIRES/REGISTERS\n\n");

////////////////////////////////////////////////////////////////////////////////
// ASSIGNS

fprintf(v_switch, "////////////////////////////////////////////////////////\n");
fprintf(v_switch, "// ASSIGNS\n\n");

fprintf(v_switch, "\t// All Channels see the FSL output from uB\n");
fprintf(v_switch, "\t// Reads from the FSL if the address is corresponding to
    its own switch\n");
fprintf(v_switch, "\tassign FSL_S_Read = ");

for (i=0;i<P_NODE_SIZE;i++){
    if(i==P_NODE_SIZE-1){
        fprintf(v_switch, "ch%d_out_read;\n\n",i);
    }
    else{
        fprintf(v_switch, "ch%d_out_read || ",i);
    }
}

for (i=0;i<P_NODE_SIZE;i++){
    fprintf(v_switch, "\tassign ch%d_out_data = FSL_S_Data;\n",i);
    fprintf(v_switch, "\tassign ch%d_out_ctrl = FSL_S_Control;\n",i);
    fprintf(v_switch, "\tassign ch%d_out_exists = FSL_S_Exists;\n\n",i);
}


////////////////////////////////////////////////////////////////////////////////
// INSTANTIATIONS

fprintf(v_switch, "////////////////////////////////////////////////////////\n");
fprintf(v_switch, "// INSTANTIATIONS\n\n");

fprintf(v_switch, "\t// Looks at the input Channels and sees if address is
    corresponding to uB\n");
fprintf(v_switch, "\t// If it is, reads from the connected FSL Channel\n");
if(P_VENDOR == 1){
    fprintf(v_switch, "\tswitch_fsm%d #(\n",P_NODE_SIZE);
}else{
    fprintf(v_switch, "\tswitch_fsm_%d #(\n",P_NODE_SIZE);
}
fprintf(v_switch, "\t .C_DEST_L(C_DEST_L),\n");
fprintf(v_switch, "\t .C_DEST_H(C_DEST_H))\n");
fprintf(v_switch, "\tswitch_fsm%d(\n", P_NODE_SIZE);
fprintf(v_switch, "\t .clk(FSL_Clk),\n");
fprintf(v_switch, "\t .rst(FSL_Rst),\n");

for (i=0;i<P_NODE_SIZE;i++){
        fprintf(v_switch, "\t .ch%d_s_data(ch%d_in_data),\n",i,i);
        fprintf(v_switch, "\t .ch%d_s_control(ch%d_in_ctrl),\n",i,i);
        fprintf(v_switch, "\t .ch%d_s_read(ch%d_in_read),\n",i,i);
        fprintf(v_switch, "\t .ch%d_s_exists(ch%d_in_exists),\n",i,i);
    }

    fprintf(v_switch, "\t .ch_out_m_data(FSL_M_Data),\n");
    fprintf(v_switch, "\t .ch_out_m_ctrl(FSL_M_Control),\n");
    fprintf(v_switch, "\t .ch_out_m_write(FSL_M_Write),\n");
    fprintf(v_switch, "\t .ch_out_m_full(FSL_M_Full)\n");
    fprintf(v_switch, "\t);\n\n");

    fprintf(v_switch, "endmodule");

}

////////////////////////////////////////////////////////////////////////////////
// Generate Switch state machine
//

void generate_switch_fsm_v(int P_NODE_SIZE, FILE *v_switch_fsm, int P_FSL_WIDTH){

    int i;

    fprintf(v_switch_fsm, "`timescale 1ns / 1ps\n");

    fprintf(v_switch_fsm, "//////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// Switch_FSM\n");
    fprintf(v_switch_fsm, "// # of Channels: %d\n", P_NODE_SIZE);
    fprintf(v_switch_fsm, "// \n");
    fprintf(v_switch_fsm, "/////////////////////////////////////////////////////\n\n");

    ////////////////////////////////////////////////////////////////////////////////
    // MODULE DEFINITION

    if (P_VENDOR == 1){
        fprintf(v_switch_fsm, "module switch_fsm%d(\n\n",P_NODE_SIZE);
    }else{
        fprintf(v_switch_fsm, "module switch_fsm_%d(\n\n",P_NODE_SIZE);
    }

    // single wire definition
    fprintf(v_switch_fsm, "\tclk,\n");
    fprintf(v_switch_fsm, "\trst,\n\n");

    // Number of Channels
    for (i=0; i<P_NODE_SIZE; i++){
        fprintf(v_switch_fsm, "\t// Channel %d Interface\n", i);
        fprintf(v_switch_fsm, "\tch%d_s_data,\n", i);
        fprintf(v_switch_fsm, "\tch%d_s_control,\n", i);
        fprintf(v_switch_fsm, "\tch%d_s_read,\n", i);
        fprintf(v_switch_fsm, "\tch%d_s_exists,\n\n", i);
    }

    fprintf(v_switch_fsm, "\tch_out_m_data,\n");
    fprintf(v_switch_fsm, "\tch_out_m_ctrl,\n");
    fprintf(v_switch_fsm, "\tch_out_m_write,\n");
    fprintf(v_switch_fsm, "\tch_out_m_full\n\n");

    fprintf(v_switch_fsm, ");\n\n");

    ////////////////////////////////////////////////////////////////////////////////
    // PARAMETERS

    fprintf(v_switch_fsm, "////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// PARAMETERS\n\n");

    fprintf(v_switch_fsm, "\tlocalparam P_DATA_WIDTH = %d;\n", P_FSL_WIDTH);
    fprintf(v_switch_fsm, "\tlocalparam P_MSG_WIDTH = %d;\n", P_FSL_WIDTH/2);
    fprintf(v_switch_fsm, "\tlocalparam P_ADDR_WIDTH = %d;\n\n", P_FSL_WIDTH/2);

    ////////////////////////////////////////////////////////////////////////////////
    // PORTS

    fprintf(v_switch_fsm, "////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// PORTS\n\n");

    fprintf(v_switch_fsm, "\tparameter\t\t\t\t\t\t\t\t\tC_DEST_L = 0;\n");
    fprintf(v_switch_fsm, "\tparameter\t\t\t\t\t\t\t\t\tC_DEST_H = 1;\n");

    fprintf(v_switch_fsm, "\tinput\t\t\t\t\t\t\t\t\t\tclk;\n");
    fprintf(v_switch_fsm, "\tinput\t\t\t\t\t\t\t\t\t\trst;\n\n");

    for (i=0;i<P_NODE_SIZE;i++){
        fprintf(v_switch_fsm, "\t// Channel %d\n",i);
        fprintf(v_switch_fsm, "\tinput\t\t[P_DATA_WIDTH-1:0]\t\tch%d_s_data;\n",i);
        fprintf(v_switch_fsm, "\tinput\t\t\t\t\t\t\t\t\t\tch%d_s_control;\n",i);
        fprintf(v_switch_fsm, "\toutput\t\t\t\t\t\t\t\t\tch%d_s_read;\n",i);
        fprintf(v_switch_fsm, "\tinput\t\t\t\t\t\t\t\t\t\tch%d_s_exists;\n\n",i);
    }

    fprintf(v_switch_fsm, "\t// Output Channel\n");
    fprintf(v_switch_fsm, "\toutput\t[P_DATA_WIDTH-1:0]\t\tch_out_m_data;\n");
    fprintf(v_switch_fsm, "\toutput\t\t\t\t\t\t\t\t\tch_out_m_ctrl;\n");
    fprintf(v_switch_fsm, "\toutput\t\t\t\t\t\t\t\t\tch_out_m_write;\n");
    fprintf(v_switch_fsm, "\tinput\t\t\t\t\t\t\t\t\t\tch_out_m_full;\n\n");

    ////////////////////////////////////////////////////////////////////////////////
    // STATES
    fprintf(v_switch_fsm, "////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// STATES\n\n");

    fprintf(v_switch_fsm, "\tlocalparam Idle_State = 0;\n");
    fprintf(v_switch_fsm, "\tlocalparam Wait_State = 1;\n");
    fprintf(v_switch_fsm, "\tlocalparam Transmit_State = 2;\n\n");

    ///////////////////////////////////////////////////////////////////////////////
    // WIRES/REGISTERS

    fprintf(v_switch_fsm, "////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// WIRES/REGISTERS\n\n");

    fprintf(v_switch_fsm, "\t// State Machine\n");
    fprintf(v_switch_fsm, "\treg\t\t[1:0]\t\t\t\t\t\t\tswitch_state_cs;\n");
    fprintf(v_switch_fsm, "\treg\t\t[1:0]\t\t\t\t\t\t\tswitch_state_ns;\n\n");

    fprintf(v_switch_fsm, "\t// Registers\n");
    fprintf(v_switch_fsm, "\treg\t\t[P_MSG_WIDTH-1:0]\t\t\tch_out_msg_size;\n");
    fprintf(v_switch_fsm, "\treg\t\t[P_ADDR_WIDTH-1:0]\t\tch_out_src;\n\n");

    fprintf(v_switch_fsm, "\t// FSM Outputs\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tld_ch_out_src;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_ld_cntr;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_en_cntr;\n");
    fprintf(v_switch_fsm, "\treg\t\t[P_MSG_WIDTH-1:0]\t\t\tch_out_cntr;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_write;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_ctrl;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_read;\n");
    fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch_out_exists;\n\n");

    fprintf(v_switch_fsm, "\t// Message Exists\n");
    fprintf(v_switch_fsm, "\twire\t\t\t\t\t\t\t\t\t\tincoming_msg;\n\n");

    fprintf(v_switch_fsm, "\t// Multiplexed Read Message and Data\n");

    for (i=0;i<P_NODE_SIZE;i++){
        fprintf(v_switch_fsm, "\treg\t\t\t\t\t\t\t\t\t\tch%d_s_read;\n",i);
    }
    fprintf(v_switch_fsm, "\treg\t\t[P_DATA_WIDTH-1:0]\t\tch_out_m_data;\n\n");

    fprintf(v_switch_fsm, "\t// Wires\n");
    for (i=0;i<P_NODE_SIZE;i++){
        fprintf(v_switch_fsm, "\twire\t\t[P_ADDR_WIDTH-1:0]\t\tch%d_dest;\n",i);
        fprintf(v_switch_fsm, "\twire\t\t[P_MSG_WIDTH-1:0]\t\t\tch%d_size;\n",i);
        fprintf(v_switch_fsm, "\twire\t\t[P_DATA_WIDTH-1:0]\t\tch%d_data;\n",i);
        fprintf(v_switch_fsm, "\twire\t\t\t\t\t\t\t\t\t\tch%d_ctrl;\n\n",i);
    }

////////////////////////////////////////////////////////////////////////////////
```

```
// ASSIGNS

    fprintf(v_switch_fsm, "/////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// ASSIGNS\n\n");

    fprintf(v_switch_fsm, "\tassign ch_out_m_ctrl = ch_out_ctrl;\n\n");

    fprintf(v_switch_fsm, "\tassign ch_out_m_write = ch_out_write;\n\n");

    // Incoming Message Definition
    fprintf(v_switch_fsm, "\tassign incoming_msg = (");
    for (i=0;i<P_NODE_SIZE;i++){
        if (i==P_NODE_SIZE-1 && i!=0){
            fprintf(v_switch_fsm, "\t\t\t(ch%d_dest >= C_DEST_L && ch%d_dest <=
                C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1));\n\n",i,i,i,i);
        }
        else if (i==P_NODE_SIZE-1 && i == 0){
            fprintf(v_switch_fsm, "(ch%d_dest >= C_DEST_L && ch%d_dest <=
                C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1));\n\n",i,i,i,i);
        }
        else if (i == 0){
            fprintf(v_switch_fsm, "(ch%d_dest >= C_DEST_L && ch%d_dest <=
                C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1) ||\n",i,i,i,i);
        }
        else{
            fprintf(v_switch_fsm, "\t\t\t(ch%d_dest >= C_DEST_L && ch%d_dest <=
                C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1) ||\n",i,i,i,i);
        }
    }

    // Channel Assigns
    for (i=0;i<P_NODE_SIZE;i++){
        fprintf(v_switch_fsm, "\t///////////////////////////////////////////\n");
        fprintf(v_switch_fsm, "\t// CHANNEL %d\n",i);
        fprintf(v_switch_fsm, "\t///////////////////////////////////////////\n");
        fprintf(v_switch_fsm, "\tassign ch%d_ctrl = ch%d_s_control;\n",i,i);
        fprintf(v_switch_fsm, "\tassign ch%d_dest = ch%d_s_data[%d:%d]; //when ch%d_s_exists =
            '1' else (others=>'Z');\n",i,i,P_FSL_WIDTH-1, P_FSL_WIDTH/2,i);
        fprintf(v_switch_fsm, "\tassign ch%d_size = ch%d_s_data[%d:0];\n",i,i, P_FSL_WIDTH/2-1);
        fprintf(v_switch_fsm, "\tassign ch%d_data = ch%d_s_data; // when c%d_s_exists =
            '1' else (others=>'Z');\n\n",i,i,i);
    }

    ////////////////////////////////////////////////////////////////////////////////
    // MAIN CODE

    fprintf(v_switch_fsm, "/////////////////////////////////////////////////////\n");
    fprintf(v_switch_fsm, "// MAIN CODE\n\n");

    // STATE MACHINE
```

```
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\t// STATE MACHINE\n");
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\talways@(posedge clk)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tif (rst == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\tswitch_state_cs <= Idle_State;\n");
fprintf(v_switch_fsm, "\t\telse\n");
fprintf(v_switch_fsm, "\t\t\tswitch_state_cs <= switch_state_ns;\n");
fprintf(v_switch_fsm, "\tend\n\n");

fprintf(v_switch_fsm, "\talways@(switch_state_cs, ch_out_exists, ch_out_cntr,
    ch_out_m_full, incoming_msg)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tcase(switch_state_cs)\n");
fprintf(v_switch_fsm, "\t\t\tIdle_State:\n");
fprintf(v_switch_fsm, "\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\tif (incoming_msg == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Wait_State;\n");
fprintf(v_switch_fsm, "\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Idle_State;\n");
fprintf(v_switch_fsm, "\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\tWait_State:\n");
fprintf(v_switch_fsm, "\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\tif (ch_out_m_full == 1'b1 || ch_out_exists == 1'b0)\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Wait_State;\n");
fprintf(v_switch_fsm, "\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Transmit_State;\n");
fprintf(v_switch_fsm, "\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\tTransmit_State:\n");
fprintf(v_switch_fsm, "\t\t\t\tif (ch_out_cntr == 0)\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Idle_State;\n");
fprintf(v_switch_fsm, "\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\tswitch_state_ns <= Transmit_State;\n");
fprintf(v_switch_fsm, "\t\t\tdefault: switch_state_ns <= switch_state_cs;\n");
fprintf(v_switch_fsm, "\t\tendcase\n");
fprintf(v_switch_fsm, "\tend\n\n");

// FSM OUTPUTS
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\t// FSM Outputs\n");
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\talways@(switch_state_cs, ch_out_m_full, ch_out_exists,
    incoming_msg, rst)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tif (rst == 1'b1)\n");
fprintf(v_switch_fsm, "\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\tch_out_read <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\tch_out_write <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\tch_out_ctrl <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\tch_out_en_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\tld_ch_out_src <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\tch_out_ld_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\t\telse\n");
fprintf(v_switch_fsm, "\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\tcase(switch_state_cs)\n");
fprintf(v_switch_fsm, "\t\t\t\tIdle_State:\n");
fprintf(v_switch_fsm, "\t\t\t\tbegin\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tch_out_read <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_write <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_ctrl <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_en_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_ld_cntr <= 1'b0;\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tif (incoming_msg == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tld_ch_out_src <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tld_ch_out_src <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\tWait_State:\n");
fprintf(v_switch_fsm, "\t\t\t\tbegin\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tch_out_en_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tld_ch_out_src <= 1'b0;\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tif (ch_out_m_full == 1'b0 && ch_out_exists == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_write <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_read <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ctrl <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ld_cntr <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_write <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_read <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ctrl <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ld_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\tTransmit_State:\n");
fprintf(v_switch_fsm, "\t\t\t\tbegin\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tld_ch_out_src <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_ld_cntr <= 1'b0;\n\n");

fprintf(v_switch_fsm, "\t\t\t\t\tif (ch_out_m_full == 1'b0 && ch_out_exists == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_write <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_en_cntr <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_read <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ctrl <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_write <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_en_cntr <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_read <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\t\tch_out_ctrl <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\t\tdefault:\n");
fprintf(v_switch_fsm, "\t\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_read <= ch_out_read;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_write <= ch_out_write;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_ctrl <= ch_out_ctrl;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_en_cntr <= ch_out_en_cntr;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tld_ch_out_src <= ld_ch_out_src;\n");
fprintf(v_switch_fsm, "\t\t\t\t\tch_out_ld_cntr <= ch_out_ld_cntr;\n");
fprintf(v_switch_fsm, "\t\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\tendcase\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\tend\n\n");

// Channel Output
fprintf(v_switch_fsm, "\t// Channel Output Exists\n");
fprintf(v_switch_fsm, "\talways@(rst, ");
for(i=0;i<P_NODE_SIZE;i++){
    fprintf(v_switch_fsm, "ch%d_s_exists, ",i);
}
fprintf(v_switch_fsm, "ch_out_src)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tif (rst == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\tch_out_exists <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\telse\n");
fprintf(v_switch_fsm, "\t\tbegin\n");

fprintf(v_switch_fsm, "\t\t\tif (");
for (i=0;i<P_NODE_SIZE;i++){
    if (i==P_NODE_SIZE-1 && i==0){
        fprintf(v_switch_fsm, "(ch%d_s_exists == 1'b1 && ch_out_src == %d))\n",i,i);
    }else if (i==P_NODE_SIZE-1 && i!=0){
        fprintf(v_switch_fsm, "\t\t\t(ch%d_s_exists == 1'b1 && ch_out_src == %d))\n",i,i);
    }else if (i!=0){
        fprintf(v_switch_fsm, "\t\t\t(ch%d_s_exists == 1'b1 && ch_out_src == %d) ||\n",i,i);
    }else{
        fprintf(v_switch_fsm, "(ch%d_s_exists == 1'b1 && ch_out_src == %d) ||\n",i,i);
    }
```

```
}
fprintf(v_switch_fsm, "\t\t\t\tch_out_exists <= 1'b1;\n");
fprintf(v_switch_fsm, "\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\t\tch_out_exists <= 1'b0;\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\tend\n\n");

// Message Properties
fprintf(v_switch_fsm, "\t///////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\t// Message Properties\n");
fprintf(v_switch_fsm, "\t///////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\talways@(posedge clk)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tif (rst == 1'b1)\n");
fprintf(v_switch_fsm, "\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\tch_out_src <= 0;\n");
fprintf(v_switch_fsm, "\t\t\tch_out_msg_size <= 0;\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\t\telse\n");
fprintf(v_switch_fsm, "\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\tif (ld_ch_out_src == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\tbegin\n");

for(i=0;i<P_NODE_SIZE;i++){
    if (i==0){
        fprintf(v_switch_fsm, "\t\t\t\tif (ch%d_dest >= C_DEST_L && ch%d_dest <=
            C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1)\n",i,i,i,i);
        fprintf(v_switch_fsm, "\t\t\t\tbegin\n");
        fprintf(v_switch_fsm, "\t\t\t\t\tch_out_src <= %d;\n",i);
        fprintf(v_switch_fsm, "\t\t\t\t\tch_out_msg_size <= ch%d_size;\n",i);
        fprintf(v_switch_fsm, "\t\t\t\tend\n");
    }else{
        fprintf(v_switch_fsm, "\t\t\t\telse if (ch%d_dest >= C_DEST_L && ch%d_dest <=
            C_DEST_H && ch%d_s_control == 1'b1 && ch%d_s_exists == 1'b1)\n",i,i,i,i);
        fprintf(v_switch_fsm, "\t\t\t\tbegin\n");
        fprintf(v_switch_fsm, "\t\t\t\t\tch_out_src <= %d;\n",i);
        fprintf(v_switch_fsm, "\t\t\t\t\tch_out_msg_size <= ch%d_size;\n",i);
        fprintf(v_switch_fsm, "\t\t\t\tend\n");
    }
}

fprintf(v_switch_fsm, "\t\t\tend\n");
fprintf(v_switch_fsm, "\t\t\telse\n");
fprintf(v_switch_fsm, "\t\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\t\tch_out_src <= ch_out_src;\n");
fprintf(v_switch_fsm, "\t\t\t\tch_out_msg_size <= ch_out_msg_size;\n");
fprintf(v_switch_fsm, "\t\t\tend\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\tend\n\n");
```

```
// Counter
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\t// Counter\n");
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\talways@(posedge clk)\n");
fprintf(v_switch_fsm, "\tbegin\n");
fprintf(v_switch_fsm, "\t\tif (rst == 1'b1 || ch_out_ld_cntr == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\tch_out_cntr <= ch_out_msg_size;\n");
fprintf(v_switch_fsm, "\t\telse\n");
fprintf(v_switch_fsm, "\t\tbegin\n");
fprintf(v_switch_fsm, "\t\t\tif (ch_out_en_cntr == 1'b1)\n");
fprintf(v_switch_fsm, "\t\t\t\tch_out_cntr <= ch_out_cntr - 1;\n");
fprintf(v_switch_fsm, "\t\tend\n");
fprintf(v_switch_fsm, "\tend\n\n");

// Multiplexed Data
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n");
fprintf(v_switch_fsm, "\t// Multiplexed Read/Data\n");
fprintf(v_switch_fsm, "\t//////////////////////////////////////////\n\n");

for (i=0;i<P_NODE_SIZE;i++){
    fprintf(v_switch_fsm, "\t// Channel %d Read\n",i);
    fprintf(v_switch_fsm, "\talways@(rst,ch_out_read, ch_out_src)\n");
    fprintf(v_switch_fsm, "\tbegin\n");
    fprintf(v_switch_fsm, "\t\tif (rst == 1'b1)\n");
    fprintf(v_switch_fsm, "\t\t\tch%d_s_read <= 1'b0;\n",i);
    fprintf(v_switch_fsm, "\t\telse\n");
    fprintf(v_switch_fsm, "\t\tbegin\n");
    fprintf(v_switch_fsm, "\t\t\tif (ch_out_read == 1'b1 && ch_out_src == %d)\n",i);
    fprintf(v_switch_fsm, "\t\t\t\tch%d_s_read <= 1'b1;\n",i);
    fprintf(v_switch_fsm, "\t\t\telse\n");
    fprintf(v_switch_fsm, "\t\t\t\tch%d_s_read <= 1'b0;\n",i);
    fprintf(v_switch_fsm, "\t\tend\n");
    fprintf(v_switch_fsm, "\tend\n\n");
}

// Output Data
fprintf(v_switch_fsm, "\t// Output Data\n");
fprintf(v_switch_fsm, "\talways@(");
for (i=0;i<P_NODE_SIZE;i++){
    fprintf(v_switch_fsm, "ch%d_data, ",i);
}
fprintf(v_switch_fsm, "ch_out_write, ch_out_src, rst)\n");
fprintf(v_switch_fsm, "\tbegin\n");
for(i=0;i<P_NODE_SIZE;i++){
    if (i==0){
        fprintf(v_switch_fsm, "\t\tif (ch_out_write == 1'b1 && ch_out_src == %d)\n",i);
        fprintf(v_switch_fsm, "\t\t\tch_out_m_data <= ch%d_data;\n",i);
    }else{
        fprintf(v_switch_fsm, "\t\telse if (ch_out_write == 1'b1 && ch_out_src == %d)\n",i);
```

```
            fprintf(v_switch_fsm, "\t\t\tch_out_m_data <= ch%d_data;\n",i);
        }
    }
    fprintf(v_switch_fsm, "\t\telse\n");
    fprintf(v_switch_fsm, "\t\t\tch_out_m_data <= 0;\n");
    fprintf(v_switch_fsm, "\tend\n\n");

    fprintf(v_switch_fsm, "endmodule");
}

////////////////////////////////////////////////////////////////////////////////
// Generate Switch Data Files

// Generate Pao File
void generate_switch_pao(int P_NODE_SIZE, FILE *switch_pao){

    fprintf (switch_pao, "##############################################################\n");
    fprintf (switch_pao, "## Description:       Peripheral Analysis Order\n");
    fprintf (switch_pao, "#########################################################\n\n");

    fprintf (switch_pao, "lib switch%d_v1_00_a switch%d verilog\n", P_NODE_SIZE, P_NODE_SIZE);
    fprintf (switch_pao, "lib switch%d_v1_00_a switch_fsm%d verilog\n", P_NODE_SIZE, P_NODE_SIZE);

}

// Generate HDL Files
void generate_switch_hdl(int P_NODE_SIZE, FILE *v_switch, FILE *v_switch_fsm, int P_FSL_WIDTH){

    generate_switch_v(P_NODE_SIZE, v_switch, P_FSL_WIDTH);
    generate_switch_fsm_v(P_NODE_SIZE, v_switch_fsm, P_FSL_WIDTH);

}

// Generate Data Files
void generate_switch_data(int P_NODE_SIZE, FILE *switch_mpd, FILE *switch_pao, int P_FSL_WIDTH){

    generate_switch_mpd(P_NODE_SIZE, switch_mpd, P_FSL_WIDTH);
    generate_switch_pao(P_NODE_SIZE, switch_pao);
}
```

B.5 Generate Xilinx *.xmp file (generate_xmp.c)

The following functions are used to generate the system.xmp file required by the Xilinx CAD tools.

```
#include <stdio.h>
#include "globals.h"

//Generate Version Number
void gen_xmp_version(FILE *xmp_file){
    fprintf(xmp_file, "XmpVersion: 10.1.02\n");
    fprintf(xmp_file, "VerMgmt: 10.1.02\n");
    fprintf(xmp_file, "IntStyle: default\n");
}

// Generate File Locations
void gen_xmp_file_locations(FILE *xmp_file){
    fprintf(xmp_file, "MHS File: system.mhs\n");
    fprintf(xmp_file, "MSS File: system.mss\n");
    fprintf(xmp_file, "NPL File: projnav/system.ise\n");
    fprintf(xmp_file, "UcfFile: data/system.ucf\n");
}

// Generate Parameters
void gen_xmp_parameters(FILE *xmp_file){
    fprintf(xmp_file, "UserCmd1: \n");
    fprintf(xmp_file, "UserCmd1Type: 0\n");
    fprintf(xmp_file, "UserCmd2: \n");
    fprintf(xmp_file, "UserCmd2Type: 0\n");
    fprintf(xmp_file, "TopInst: system_i\n");
    fprintf(xmp_file, "GenSimTB: 0\n");
    fprintf(xmp_file, "InsertNoPads: 0\n");
    fprintf(xmp_file, "WarnForEAArch: 1\n");
    fprintf(xmp_file, "HdlLang: VHDL\n");
    fprintf(xmp_file, "Simulator: mti\n");
    fprintf(xmp_file, "SimModel: BEHAVIORAL\n");
    fprintf(xmp_file, "MixLangSim: 1\n");
    fprintf(xmp_file, "FpgaImpMode: 0\n");
    fprintf(xmp_file, "EnableParTimingError: 1\n");
    fprintf(xmp_file, "EnableResetOptimization: 0\n");
    fprintf(xmp_file, "ShowLicenseDialog: 1\n");
}

// Generate MicroBlazes
void gen_xmp_processors(FILE *xmp_file, int P_NUM_NODES){

    int i;
    char gen_xmp_microblaze[100];
    strcpy(gen_xmp_microblaze, "Processor: microblaze_");

    for (i=0;i<P_NUM_NODES;i++){
        fprintf(xmp_file, gen_xmp_microblaze); // declare uB
        fprintf(xmp_file, "%d", i);
        fprintf(xmp_file, "\nBootLoop: 0\n");
        fprintf(xmp_file, "XmdStub: 0\n");
    }
}

// Generate FPGA Architecture
void gen_xmp_architecture(FILE *xmp_file){
    fprintf(xmp_file, "Architecture: ");
    fprintf(xmp_file, P_ARCH);
    fprintf(xmp_file, "\n");
    fprintf(xmp_file, "Device: ");
    fprintf(xmp_file, P_DEVICE);
    fprintf(xmp_file, "\n");
    fprintf(xmp_file, "Package: ");
    fprintf(xmp_file, P_PACKAGE);
    fprintf(xmp_file, "\n");
    fprintf(xmp_file, "SpeedGrade: ");
    fprintf(xmp_file, P_SPEED);
    fprintf(xmp_file, "\n");
}

// Generate XMP File
void generate_xmp(FILE *xmp_file){
    gen_xmp_version(xmp_file);
    gen_xmp_file_locations(xmp_file);
    gen_xmp_architecture(xmp_file);
    gen_xmp_parameters(xmp_file);

    // Generate MicroBlaze nodes
    if (P_NODE_TYPE == 0){
        gen_xmp_processors(xmp_file, P_NUM_NODES);
    }
}
```

B.6 Generate Xilinx *.mhs file (generate_mhs.c)

The following functions are used to generate the system.mhs file required by the Xilinx CAD tools.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "globals.h"
#define P_MAX_CONNECTIONS_PER_NODE          128
#define P_MAX_NODES                     128

// Generate Random Number
int int_rand(int N){
    return(rand()/(int)(((unsigned)RAND_MAX + 1) / N));
}

// Function to generate MicroBlaze Nodes in *.mhs file
void generate_mhs_microblaze(FILE *mhs_file, int P_NODE, int PLB_CONNECT, char P_ARCH[100],
        int P_W_RESET){

    // Generate MicroBlaze HW Instance
    fprintf(mhs_file, "BEGIN microblaze\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = microblaze_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 7.10.c \n");
    fprintf(mhs_file, " PARAMETER C_FSL_LINKS = 1\n");
    fprintf(mhs_file, " PARAMETER C_FAMILY = ");
    fprintf(mhs_file, P_ARCH);
    fprintf(mhs_file, "\n");

    // Set MicroBlaze parameters
    fprintf(mhs_file, " PARAMETER C_INSTANCE = microblaze_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER C_USE_HW_MUL = 0\n");
    fprintf(mhs_file, " PARAMETER C_USE_MSR_INSTR = 0\n");
    fprintf(mhs_file, " PARAMETER C_USE_PCMP_INSTR = 0\n");
    fprintf(mhs_file, " PARAMETER C_AREA_OPTIMIZED = 1\n");
    // COnnect to PLB bus
    if (PLB_CONNECT == 1){
        fprintf(mhs_file, " BUS_INTERFACE DPLB = mb_plb\n");
        fprintf(mhs_file, " BUS_INTERFACE IPLB = mb_plb\n");
    }
    // Connect to Associated FSL Bus
    fprintf(mhs_file, " BUS_INTERFACE SFSL0 = fsl_v20_%d\n", P_NODE*2+1);
    fprintf(mhs_file, " BUS_INTERFACE DLMB = dlmb_%d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE ILMB = ilmb_%d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE MFSL0 = fsl_v20_%d\n", P_NODE*2);
    if (P_W_RESET == 1){
        fprintf(mhs_file, " PORT MB_RESET = mb_reset\n");
    }else{
        fprintf(mhs_file, " PORT MB_RESET = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");

    // Generate LMB/Controller/Memory
    // Generate Instruction Bus
    fprintf(mhs_file, "BEGIN lmb_v10\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = ilmb_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
    fprintf(mhs_file, " PORT LMB_Clk = sys_clk_s\n");  // Always connected to sys_clk_s
    if (P_W_RESET == 1){                           // Connect to Reset Controller
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");
    // Generate Data Bus
    fprintf(mhs_file, "BEGIN lmb_v10\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = dlmb_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
    fprintf(mhs_file, " PORT LMB_Clk = sys_clk_s\n");  // Always connected to sys_clk_s
    if (P_W_RESET == 1){                                // Connect to Reset Controller
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");
    // Generate DLMB Controller
    fprintf(mhs_file, "BEGIN lmb_bram_if_cntlr\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = dlmb_cntlr_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.10.a\n");
    fprintf(mhs_file, " PARAMETER C_BASEADDR = 0x00000000\n");
    fprintf(mhs_file, " PARAMETER C_HIGHADDR = 0x00001fff\n");
    fprintf(mhs_file, " BUS_INTERFACE SLMB = dlmb_%d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE BRAM_PORT = dlmb_port_%d\n", P_NODE);
    fprintf(mhs_file, "END\n\n");
    // Generate ILMB Controller
    fprintf(mhs_file, "BEGIN lmb_bram_if_cntlr\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = ilmb_cntlr_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.10.a\n");
    fprintf(mhs_file, " PARAMETER C_BASEADDR = 0x00000000\n");
    fprintf(mhs_file, " PARAMETER C_HIGHADDR = 0x00001fff\n");
    fprintf(mhs_file, " BUS_INTERFACE SLMB = ilmb_%d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE BRAM_PORT = ilmb_port_%d\n", P_NODE);
    fprintf(mhs_file, "END\n\n");
    // Generate BRAM
    fprintf(mhs_file, "BEGIN bram_block\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = lmb_bram_%d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
```

```
    fprintf(mhs_file, " BUS_INTERFACE PORTA = ilmb_port_%d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE PORTB = dlmb_port_%d\n", P_NODE);
    fprintf(mhs_file, "END\n\n");

    // Generate FSL Ports
    // Master and Slave
    fprintf(mhs_file, "BEGIN fsl_v20\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = fsl_v20_%d\n", P_NODE*2);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.11.a\n");
    fprintf(mhs_file, " PORT FSL_Clk = sys_clk_s\n");
    if (P_W_RESET == 1){
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");

    fprintf(mhs_file, "BEGIN fsl_v20\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = fsl_v20_%d\n", P_NODE*2+1);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.11.a\n");
    fprintf(mhs_file, " PORT FSL_Clk = sys_clk_s\n");
    if (P_W_RESET == 1){
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");

}

// Generate Multiplier Node Specifications
void generate_mhs_node(FILE *mhs_file, int P_NODE, int P_W_RESET, int P_FSL_WIDTH,
                            int P_MULTIPLICAND_WIDTH, int P_MULTIPLICAND_WIDTH_MIN,
                            int P_MULTIPLICAND_WIDTH_MAX){

    int multiplicand_width;

    // determine node size
    if (P_NODE == 0){
        multiplicand_width = P_MULTIPLICAND_WIDTH;
        multiplicand_size[P_NODE] = multiplicand_width;
    }else{
        multiplicand_width = int_rand((P_MULTIPLICAND_WIDTH_MAX -
            P_MULTIPLICAND_WIDTH_MIN)/2+1)*2+P_MULTIPLICAND_WIDTH_MIN;
        multiplicand_size[P_NODE] = multiplicand_width;
    }

    // Generate Multiplier
    if (P_NODE == 0){
        fprintf(mhs_file, "BEGIN init_multiplier_%d_%d\n", P_FSL_WIDTH, multiplicand_width);
        fprintf(mhs_file, " PARAMETER INSTANCE = init_multiplier_%d_%d_%d\n", P_FSL_WIDTH,
            multiplicand_width, P_NODE);
    }else{
        fprintf(mhs_file, "BEGIN multiplier_%d_%d\n", P_FSL_WIDTH, multiplicand_width);
        fprintf(mhs_file, " PARAMETER INSTANCE = multiplier_%d_%d_%d\n", P_FSL_WIDTH,
             multiplicand_width, P_NODE);
    }
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
    fprintf(mhs_file, " BUS_INTERFACE MFSL = fsl_v20_%d\n", P_NODE*2);
    fprintf(mhs_file, " BUS_INTERFACE SFSL = fsl_v20_%d\n", P_NODE*2+1);
    if (P_NODE == 0){
        fprintf(mhs_file, " PORT output_result = output_result_pin\n");
        fprintf(mhs_file, " PORT multiplicand = multiplicand_pin\n");
    }
    fprintf(mhs_file, "END\n\n");

    // Generate FSL Ports
    // Master and Slave
    fprintf(mhs_file, "BEGIN fsl_v20\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = fsl_v20_%d\n", P_NODE*2);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.11.a\n");
    fprintf(mhs_file, " PARAMETER C_FSL_DWIDTH = %d\n", P_FSL_WIDTH);
    fprintf(mhs_file, " PORT FSL_Clk = sys_clk_s\n");
    if (P_W_RESET == 1){                    // Connect to Reset Controller
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");

    fprintf(mhs_file, "BEGIN fsl_v20\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = fsl_v20_%d\n", P_NODE*2+1);
    fprintf(mhs_file, " PARAMETER HW_VER = 2.11.a\n");
    fprintf(mhs_file, " PARAMETER C_FSL_DWIDTH = %d\n", P_FSL_WIDTH);
    fprintf(mhs_file, " PORT FSL_Clk = sys_clk_s\n");
    if (P_W_RESET == 1){                   // Connect to Reset Controller
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");
}

// Generate PLB Bus
void generate_mhs_plb(FILE *mhs_file, int P_W_RESET){

    fprintf(mhs_file, "BEGIN plb_v46\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = mb_plb\n");
    fprintf(mhs_file, " PARAMETER HW_VER = 1.02.a\n");
    fprintf(mhs_file, " PORT PLB_Clk = sys_clk_s\n");
    if (P_W_RESET == 1){                              // Connect to Reset Controller
        fprintf(mhs_file, " PORT SYS_Rst = sys_bus_reset\n");
    }else{
        fprintf(mhs_file, " PORT SYS_Rst = sys_rst_s\n");
    }
    fprintf(mhs_file, "END\n\n");
}

// Generate UART
void generate_mhs_uart(FILE *mhs_file, int P_DCM_FREQUENCY, int P_CLK_FREQUENCY, int P_W_CLK){

    fprintf(mhs_file, "BEGIN xps_uartlite\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = RS232_Uart\n");
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
    fprintf(mhs_file, " PARAMETER C_BAUDRATE = 9600\n");
    fprintf(mhs_file, " PARAMETER C_DATA_BITS = 8\n");
    fprintf(mhs_file, " PARAMETER C_ODD_PARITY = 0\n");
    fprintf(mhs_file, " PARAMETER C_USE_PARITY = 0\n");

    if (P_W_CLK == 1){
        fprintf(mhs_file, " PARAMETER C_SPLB_CLK_FREQ_HZ = %d\n", P_DCM_FREQUENCY);
    }else{
        fprintf(mhs_file, " PARAMETER C_SPLB_CLK_FREQ_HZ = %d\n", P_CLK_FREQUENCY);
    }

    fprintf(mhs_file, " PARAMETER C_BASEADDR = 0x84000000\n");
    fprintf(mhs_file, " PARAMETER C_HIGHADDR = 0x8400ffff\n");
    fprintf(mhs_file, " BUS_INTERFACE SPLB = mb_plb\n");
    fprintf(mhs_file, " PORT RX = RS232_Uart_RX\n");
    fprintf(mhs_file, " PORT TX = RS232_Uart_TX\n");
    fprintf(mhs_file, "END\n\n");
}

// Generate Switch Specifications
void generate_mhs_switch(FILE *mhs_file, int P_NODE,int P_NODE_SIZE,
                int P_NODE_CONNECTIONS[P_MAX_CONNECTIONS_PER_NODE],
                int P_NODE_CHANNELS[P_MAX_CONNECTIONS_PER_NODE]){

    int i;

    fprintf(mhs_file, "BEGIN switch%d\n", P_NODE_SIZE);
    fprintf(mhs_file, " PARAMETER INSTANCE = switch%d_%d\n", P_NODE_SIZE, P_NODE);
    fprintf(mhs_file, " PARAMETER HW_VER = 1.00.a\n");
    fprintf(mhs_file, " PARAMETER C_DEST_L = %d\n", P_NODE);
    fprintf(mhs_file, " PARAMETER C_DEST_H = %d\n", P_NODE);
    fprintf(mhs_file, " BUS_INTERFACE MFSL = fsl_v20_%d\n", P_NODE*2+1);
    fprintf(mhs_file, " BUS_INTERFACE SFSL = fsl_v20_%d\n", P_NODE*2);

    for (i=0;i<P_NODE_SIZE;i++){
        // output ports associated to own switch
        fprintf(mhs_file, " PORT ch%d_out_read = switch_%d_ch%d_read\n", i, P_NODE, i);
        fprintf(mhs_file, " PORT ch%d_out_exists = switch_%d_ch%d_exists\n", i, P_NODE, i);
        fprintf(mhs_file, " PORT ch%d_out_ctrl = switch_%d_ch%d_ctrl\n", i, P_NODE, i);
        fprintf(mhs_file, " PORT ch%d_out_data = switch_%d_ch%d_data\n", i, P_NODE, i);

        // input ports connected to desired connection
        fprintf(mhs_file, " PORT ch%d_in_read = switch_%d_ch%d_read\n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        fprintf(mhs_file, " PORT ch%d_in_exists = switch_%d_ch%d_exists\n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        fprintf(mhs_file, " PORT ch%d_in_ctrl = switch_%d_ch%d_ctrl\n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        fprintf(mhs_file, " PORT ch%d_in_data = switch_%d_ch%d_data \n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
    }
    fprintf(mhs_file, "END\n\n");
}

// Generate Clock Generator (Optional Function)
void generate_mhs_clock_gen(FILE *mhs_file, int P_CLK_FREQUENCY, int P_DCM_FREQUENCY){

    fprintf(mhs_file, "BEGIN clock_generator\n");
    fprintf(mhs_file, " PARAMETER INSTANCE = clock_generator_0\n");
    fprintf(mhs_file, " PARAMETER HW_VER = 2.01.a\n");
    fprintf(mhs_file, " PARAMETER C_EXT_RESET_HIGH = 1\n");
    fprintf(mhs_file, " PARAMETER C_CLKIN_FREQ = %d\n", P_CLK_FREQUENCY);
    fprintf(mhs_file, " PARAMETER C_CLKOUT0_FREQ = %d\n", P_DCM_FREQUENCY);
    fprintf(mhs_file, " PARAMETER C_CLKOUT0_BUF = TRUE\n");
    fprintf(mhs_file, " PARAMETER C_CLKOUT0_PHASE = 0\n");
    fprintf(mhs_file, " PARAMETER C_CLKOUT0_GROUP = NONE\n");
    fprintf(mhs_file, " PARAMETER C_DCM0_CLKIN_PERIOD = 10.000000\n");
    fprintf(mhs_file, " PARAMETER C_DCM1_CLKIN_PERIOD = 10.000000\n");
    fprintf(mhs_file, " PARAMETER C_DCM2_CLKIN_PERIOD = 10.000000\n");
    fprintf(mhs_file, " PARAMETER C_DCM3_CLKIN_PERIOD = 10.000000\n");
    fprintf(mhs_file, " PORT CLKOUT0 = sys_clk_s\n");
    fprintf(mhs_file, " PORT CLKIN = dcm_clk_s\n");
    fprintf(mhs_file, " PORT RST = net_gnd\n");
    fprintf(mhs_file, " PORT LOCKED = Dcm_all_locked\n");
    fprintf(mhs_file, "END\n\n");
}

// Generate Reset Controller (Optional Function)
void generate_mhs_reset_controller(FILE *mhs_file, int P_W_CLK){

    fprintf(mhs_file, "BEGIN proc_sys_reset\n");
    fprintf(mhs_file, "PARAMETER INSTANCE = proc_sys_reset_0\n");
    fprintf(mhs_file, "PARAMETER HW_VER = 2.00.a\n");
    fprintf(mhs_file, "PARAMETER C_EXT_RESET_HIGH = 0\n");
    fprintf(mhs_file, "PORT Slowest_sync_clk = sys_clk_s\n");
    if (P_W_CLK == 1){
        fprintf(mhs_file, "PORT Dcm_locked = Dcm_all_locked\n");
    }
    fprintf(mhs_file, "PORT Ext_Reset_In = sys_rst_s\n");
    fprintf(mhs_file, "PORT MB_Reset = mb_reset\n");
    fprintf(mhs_file, "PORT Bus_Struct_Reset = sys_bus_reset\n");
    fprintf(mhs_file, "PORT Peripheral_Reset = sys_periph_reset\n");
    fprintf(mhs_file, "END\n\n");

}

// Generate Header used in MHS Function
void generate_mhs_header(FILE *mhs_file, int P_CLK_FREQUENCY, int P_W_CLK,
                int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH, int P_TIMING_CONSTRAINT){

    // Generate Header
    fprintf(mhs_file, "#########################################################\n");
    fprintf(mhs_file, "# MHS FILE\n\n");
    fprintf(mhs_file, "PARAMETER VERSION = 2.1.0\n\n");

    // Generate External Ports
    if (P_NODE_TYPE==1){
        fprintf(mhs_file, " PORT result_output= output_result_pin , DIR = O, VEC =
            [%d:0]\n", P_FSL_WIDTH-1);
        fprintf(mhs_file, " PORT  multiplicand_input = multiplicand_pin, DIR = I, VEC =
            [%d:0]\n\n", P_MULTIPLICAND_WIDTH-1);
    }else{
        fprintf(mhs_file, " PORT fpga_0_RS232_Uart_RX_pin = RS232_Uart_RX, DIR = I\n");
        fprintf(mhs_file, " PORT fpga_0_RS232_Uart_TX_pin = RS232_Uart_TX, DIR = O\n");
    }

    fprintf(mhs_file, " PORT sys_clk_pin = ");
    if (P_W_CLK == 1){
        fprintf(mhs_file, "dcm_clk_s,");
    }else{
        fprintf(mhs_file, "sys_clk_s,");
    }
    if (P_TIMING_CONSTRAINT == 1){
        fprintf(mhs_file, " DIR = I, SIGIS = CLK, CLK_FREQ = %d\n",P_CLK_FREQUENCY);
    }else{
        fprintf(mhs_file, " DIR = I, SIGIS = CLK\n");
    }
    fprintf(mhs_file, " PORT sys_rst_pin = sys_rst_s, DIR = I, RST_POLARITY = 1, SIGIS = RST\n\n");
}

// Main function used to generate MHS file, calls subfunctions to generate
// each section corresponding to individual elements
void generate_mhs(FILE *mhs_file, int P_NODE_SIZE[P_MAX_NODES], int P_NODE_CONNECTIONS
                        [P_MAX_NODES][P_MAX_NODES], int P_NODE_CHANNELS[P_MAX_NODES][P_MAX_NODES],
                        int P_CLK_FREQUENCY, int P_NUM_NODES, int P_FSL_WIDTH,
                        int P_MULTIPLICAND_WIDTH, int P_MULTIPLICAND_WIDTH_MIN,
                        int P_MULTIPLICAND_WIDTH_MAX, int P_TIMING_CONSTRAINT){

    int i,j;
    int P_CONNECTIONS[P_MAX_CONNECTIONS_PER_NODE];
    int P_CHANNELS[P_MAX_CONNECTIONS_PER_NODE];

    int P_W_CLK = 0;
    int P_W_RESET = 0;

    // Put this elsewhere
    int P_DCM_FREQUENCY = P_CLK_FREQUENCY;

    generate_mhs_header(mhs_file, P_CLK_FREQUENCY, P_W_CLK, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
            P_TIMING_CONSTRAINT);

    //Generate Multiplier nodes
    if (P_NODE_TYPE == 1){
        for (i=0;i<P_NUM_NODES;i++){
            if (i==0){
                generate_mhs_node(mhs_file, i, P_W_RESET, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
                            P_MULTIPLICAND_WIDTH_MIN, P_MULTIPLICAND_WIDTH_MAX);
            }else{
                generate_mhs_node(mhs_file, i, P_W_RESET, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
                            P_MULTIPLICAND_WIDTH_MIN, P_MULTIPLICAND_WIDTH_MAX);
        }
    }
    // Generate MicroBlaze Nodes
    }else{
        for (i=0;i<P_NUM_NODES;i++){
            if (i==0){
                generate_mhs_microblaze(mhs_file, i, 1, P_ARCH, P_W_RESET);
            }else{
                generate_mhs_microblaze(mhs_file, i, 0, P_ARCH, P_W_RESET);
            }
        }
        // Generate busses for MicroBlaze
        //generate buses
        generate_mhs_plb(mhs_file, P_W_RESET);

        //generate uart
        generate_mhs_uart(mhs_file, P_DCM_FREQUENCY, P_CLK_FREQUENCY, P_W_CLK);
    }

    //generate switch
    for (i=0;i<P_NUM_NODES;i++){

        for (j=0;j<P_NODE_SIZE[i];j++){
            P_CONNECTIONS[j] = P_NODE_CONNECTIONS[i][j];
            P_CHANNELS[j] = P_NODE_CHANNELS[i][j];
        }
        generate_mhs_switch(mhs_file, i, P_NODE_SIZE[i], P_CONNECTIONS, P_CHANNELS);
    }
    //generate clock generator
    if (P_W_CLK == 1){
        generate_mhs_clock_gen(mhs_file, P_CLK_FREQUENCY, P_DCM_FREQUENCY);
    }

    //generate reset controller
    if (P_W_RESET == 1){
        generate_mhs_reset_controller(mhs_file, P_W_CLK);
    }
}
```

B.7 Generate Xilinx *.mss file (generate_mss.c)

The following functions are used to generate the system.mss file required by the Xilinx CAD tools.

```
#include <stdio.h>
#include "globals.h"

#define P_MAX_NODES         128

// Generate MicroBlaze Drivers
void generate_mss_microblaze(FILE *mss_file, int P_NODE, int P_CONNECT_RS232){

    //Generate OS
    fprintf(mss_file, "BEGIN OS\n");
    fprintf(mss_file, " PARAMETER OS_NAME = standalone\n");
    fprintf(mss_file, " PARAMETER OS_VER = 2.00.a\n");
    fprintf(mss_file, " PARAMETER PROC_INSTANCE = microblaze_%d\n", P_NODE);
    if (P_CONNECT_RS232 == 1){
        fprintf(mss_file, " PARAMETER STDIN = RS232_Uart\n");
        fprintf(mss_file, " PARAMETER STDOUT = RS232_Uart\n");
    }
    fprintf(mss_file, "END\n\n");

    //Generate uB
    fprintf(mss_file, "BEGIN PROCESSOR\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = cpu\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.11.b\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = microblaze_%d\n", P_NODE);
    fprintf(mss_file, " PARAMETER COMPILER = mb-gcc\n");
    fprintf(mss_file, " PARAMETER ARCHIVER = mb-ar\n");
    fprintf(mss_file, "END\n\n");

    //Generate LMB Controller
    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = bram\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.00.a\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = dlmb_cntlr_%d\n", P_NODE);
    fprintf(mss_file, "END\n\n");

    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = bram\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.00.a\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = ilmb_cntlr_%d\n", P_NODE);
    fprintf(mss_file, "END\n\n");

    //Generate BRAM Block
    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, "PARAMETER DRIVER_NAME = generic\n");
    fprintf(mss_file, "PARAMETER DRIVER_VER = 1.00.a\n");
    fprintf(mss_file, "PARAMETER HW_INSTANCE = lmb_bram_%d\n", P_NODE);
    fprintf(mss_file, "END\n\n");
}

// Generate Driver for Multiplier Node
void generate_mss_node(FILE *mss_file, int P_NODE, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH){

    //Generate Multiplier
    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = generic\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.00.a\n");
    if (P_NODE == 0){
        fprintf(mss_file, " PARAMETER HW_INSTANCE = init_multiplier_%d_%d_%d\n",
            P_FSL_WIDTH, multiplicand_size[P_NODE], P_NODE);
    }else{
        fprintf(mss_file, " PARAMETER HW_INSTANCE = multiplier_%d_%d_%d\n",
            P_FSL_WIDTH, multiplicand_size[P_NODE], P_NODE);
    }
    fprintf(mss_file, "END\n");
}

// Generate Driver for Custom Switch
void generate_mss_switch(FILE *mss_file, int P_NODE, int P_NODE_SIZE){

    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = generic\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.00.a\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = switch%d_%d\n", P_NODE_SIZE, P_NODE);
    fprintf(mss_file, "END\n\n");
}

// Generate Driver for UART
void generate_mss_uart(FILE *mss_file){

    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = uartlite\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.13.a\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = RS232_Uart\n");
    fprintf(mss_file, "END\n");

}

// Generate Driver for Clock
void generate_mss_clock_gen(FILE *mss_file){

    fprintf(mss_file, "BEGIN DRIVER\n");
    fprintf(mss_file, " PARAMETER DRIVER_NAME = generic\n");
    fprintf(mss_file, " PARAMETER DRIVER_VER = 1.00.a\n");
    fprintf(mss_file, " PARAMETER HW_INSTANCE = clock_generator_0\n");
    fprintf(mss_file, "END\n");

}

// Generate Header for MSS File
void generate_mss_header(FILE *mss_file){

    fprintf(mss_file, "########################################################\n");
    fprintf(mss_file, "# MSS FILE\n\n");
}

// Main function that generates MSS file using subfunctions to generate
// individual elements
void generate_mss(FILE *mss_file, int P_NUM_NODES, int P_NODE_SIZE[P_MAX_NODES],
            int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH){

    int i;
    int P_W_CLK = 0;

    // generate header
    generate_mss_header(mss_file);

    // generate multiplier
    if (P_NODE_TYPE == 1){
        for (i=0;i<P_NUM_NODES;i++){
            if (i==0){
                generate_mss_node(mss_file, i, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
            }else{
                generate_mss_node(mss_file, i, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);
            }
        }
    }else{
        // generate MicroBlaze
        for (i=0;i<P_NUM_NODES;i++){
            if (i==0){
                generate_mss_microblaze(mss_file, i, 1);
            }else{
                generate_mss_microblaze(mss_file, i, 1);
            }
        }
    }

    // generate switches
    for (i=0;i<P_NUM_NODES;i++){
        generate_mss_switch(mss_file, i, P_NODE_SIZE[i]);
    }

    // generate uart
    //generate_mss_uart(mss_file);
    // generate clock
    if (P_W_CLK == 1){
        generate_mss_clock_gen(mss_file);
    }
}
```

B.8 Generate Xilinx *.opt file (generate_opt.c)

The following functions are used to generate the fast_runtime.opt file required by the Xilinx CAD tools.

```
#include <stdio.h>

// Generate option file, which characterizes the CAD tool's run-time
// parameters
void generate_opt(FILE *opt_file, int P_MAP_EFFORT, int P_PAR_EFFORT){

    char map_effort[100];
    char par_effort[100];

    // Define Map Effort Level
    if (P_MAP_EFFORT == 1){
        strcpy(map_effort, "-ol high;");
    }else{
        strcpy(map_effort, "-ol std;");
    }

    // Define PAR Effort Level
    if (P_PAR_EFFORT == 1){
        strcpy(par_effort, "-ol high;");
    }else{
        strcpy(par_effort, "-ol std;");
    }

    fprintf(opt_file, "FLOWTYPE = FPGA;\n");
    fprintf(opt_file, "###############################################################\n");
    fprintf(opt_file, "## Filename: fast_runtime.opt\n");
    fprintf(opt_file, "##\n");
    fprintf(opt_file, "## Option File For Xilinx FPGA Implementation Flow for Fast\n");
    fprintf(opt_file, "## Runtime.\n");
    fprintf(opt_file, "## \n");
    fprintf(opt_file, "## Version: 4.1.1\n");
    fprintf(opt_file, "###############################################################\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Options for Translator\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Type \"ngdbuild -h\" for a list of ngdbuild command line options\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "Program ngdbuild \n");
    fprintf(opt_file, "-p <partname>;   # Partname to use - picked from xflow commandline\n");
    fprintf(opt_file, "-nt timestamp;   # NGO File generation. Regenerate only when\n");
    fprintf(opt_file, "                 # source netlist is newer than existing \n");
    fprintf(opt_file, "                 # NGO file (default)\n");
    fprintf(opt_file, "-bm <design>.bmm # Block RAM memory map file\n");
    fprintf(opt_file, "<userdesign>;    # User design - pick from xflow command line\n");
    fprintf(opt_file, "-uc <design>.ucf;# ucf constraints\n");
    fprintf(opt_file, "<design>.ngd;    # Name of NGD file. Filebase same as design filebase\n");
    fprintf(opt_file, "End Program ngdbuild\n\n");

    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Options for Mapper\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Type \"map -h <arch>\" for alist of map command line options\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "Program map\n");
    fprintf(opt_file, "-o <design>_map.ncd; # Output Mapped ncd file\n");
    fprintf(opt_file, "-w;           # Overwrite output files.\n");
    fprintf(opt_file, "-pr b;               # Pack internal FF/latches into IOBs\n");
    fprintf(opt_file, "#-fp <design>.mfp;   # Floorplan file\n");
    fprintf(opt_file, map_effort);
    fprintf(opt_file, "       # Overall Effort\n");
    fprintf(opt_file, "\n-timing;\n");
    fprintf(opt_file, "<inputdir><design>.ngd;  # Input NGD file\n");
    fprintf(opt_file, "<inputdir><design>.pcf;  # Physical constraints file\n");
    fprintf(opt_file, "END Program map\n\n");

    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Options for Post Map Trace\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Type \"trce -h\" for a list of trce command line options\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "Program post_map_trce\n");
    fprintf(opt_file, "-e 3;     # error report limited to 3 items per constraint\n");
    fprintf(opt_file, "#-o <design>_map.twr;  # Output trace report file\n");
    fprintf(opt_file, "-xml <design>_map.twx;# Output XML version of timing report\n");
    fprintf(opt_file, "#-tsi <design>_map.tsi; # Produce Timing Interaction report\n");
    fprintf(opt_file, "<inputdir><design>_map.ncd;  # Input mapped ncd\n");
    fprintf(opt_file, "<inputdir><design>.pcf;      # Physical constraints file\n");
    fprintf(opt_file, "END Program post_map_trce\n\n");

    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Options for Place and Route\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Type \"par -h\" for alist of par command line options\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "Program par\n");
    fprintf(opt_file, "-w;                 # Overwrite existing placed and routed ncd\n");
    fprintf(opt_file, par_effort);
    fprintf(opt_file, "       # Overall Effort\n");
    fprintf(opt_file, "<inputdir><design>_map.ncd;  # Input mapped NCD file\n");
    fprintf(opt_file, "<design>.ncd;                # Output placed and routed NCD\n");
    fprintf(opt_file, "<inputdir><design>.pcf;      # Input physical constraints file\n");
    fprintf(opt_file, "END Program par\n\n");

    fprintf(opt_file, "#\n");
```

```
    fprintf(opt_file, "# Options for Post Par Trace\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "# Type \"trce -h\" for a  list of trce command line options\n");
    fprintf(opt_file, "#\n");
    fprintf(opt_file, "Program post_par_trce\n");
    fprintf(opt_file, "-e 3;    #  error report limited to 3 items per constraint\n");
    fprintf(opt_file, "#-o <design>.twr;  # Output trace report file\n");
    fprintf(opt_file, "-xml <design>.twx; # Output XML version of the timing report\n");
    fprintf(opt_file, "#-tsi <design>.tsi;  #Timing Specification Interaction report\n");
    fprintf(opt_file, "<inputdir><design>.ncd; # Input placed and routed ncd\n");
    fprintf(opt_file, "<inputdir><design>.pcf; # Physical constraints file\n");
    fprintf(opt_file, "END Program post_par_trce\n\n");

}
```

B.9 Generate Xilinx *.ucf file (generate_ucf.c)

The following functions are used to generate the system.ucf file required by the Xilinx CAD tools.

```
#include <stdio.h>

void generate_ucf(FILE *ucf_file, int P_CLK_FREQUENCY, int P_CONSTRAIN_BOARD,
                int P_TIMING_CONSTRAINT){

    int i;

    char clk_pin[100];
    char rst_pin[100];
    char rx_pin[100];
    char tx_pin[100];
    char period[100];

    int P_CLK_PERIOD;

    // Calculate clock period from clock frequency
    P_CLK_PERIOD = (P_CLK_FREQUENCY/100000);
    P_CLK_PERIOD = 10000000/P_CLK_PERIOD;
    itoa(P_CLK_PERIOD, period, 10);

    // Default Pin Locations, not used right now
    strcpy(clk_pin, "AH15;");
    strcpy(rst_pin, "E9;");
    strcpy(rx_pin, "AG15;");
    strcpy(tx_pin, "AG20;");

    fprintf(ucf_file, "#############################################\n");
    fprintf(ucf_file, "## UCF File\n\n");

    // If constraining output pins to specific locations, use above
    // pin locations
    if (P_CONSTRAIN_BOARD == 1){
        fprintf(ucf_file, "Net sys_clk_pin TNM_NET = sys_clk_pin;\n");
        fprintf(ucf_file, "Net sys_clk_pin LOC = ");
        fprintf(ucf_file, clk_pin);
        fprintf(ucf_file, "\n");
        fprintf(ucf_file, "Net sys_clk_pin IOSTANDARD=LVCMOS33;\n");
        fprintf(ucf_file, "Net sys_rst_pin LOC = ");
        fprintf(ucf_file, rst_pin);
        fprintf(ucf_file, "\n");
        fprintf(ucf_file, "Net sys_rst_pin IOSTANDARD=LVCMOS33;\n");
        fprintf(ucf_file, "Net sys_rst_pin PULLUP;\n\n");
    }
    // Otherwise, only constraint is the max clock frequency
    fprintf(ucf_file, "## System level constraints\n");
    fprintf(ucf_file, "Net sys_clk_pin TNM_NET = sys_clk_pin;\n");
    if (P_TIMING_CONSTRAINT == 1){
        fprintf(ucf_file, "TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin ");
        fprintf(ucf_file, period);
        fprintf(ucf_file, " ps;\n");
    }
    fprintf(ucf_file, "Net sys_rst_pin TIG;\n");
}
```

B.10 Generate FSL for Altera FPGAs (generate_fsl.c)

The following functions are used to generate the system level wrapper for the FSL link used by Altera FPGAs. These functions are only used for Altera systems, as Xilinx systems have predefined IP cores that can be instantiated in the system.mhs files.

```
#include <stdio.h>

// Used to generate wrapper for equivalent FSL FIFO for Altera FPGAs
void gen_fsl(FILE *fsl_file, int P_FSL_WIDTH){

    fprintf(fsl_file, "///////////////////////////////////\n");
    fprintf(fsl_file, "// FSL\n");
    fprintf(fsl_file, "///////////////////////////////////\n\n");

    // Define output ports
    fprintf(fsl_file, "module fsl(\n");
    fprintf(fsl_file, "\tFSL_Clk,\n");
    fprintf(fsl_file, "\tSYS_Rst,\n");
    fprintf(fsl_file, "\tFSL_Rst,\n\n");

    fprintf(fsl_file, "\tFSL_M_Clk,\n");
    fprintf(fsl_file, "\tFSL_M_Data,\n");
    fprintf(fsl_file, "\tFSL_M_Control,\n");
    fprintf(fsl_file, "\tFSL_M_Write,\n");
    fprintf(fsl_file, "\tFSL_M_Full,\n\n");

    fprintf(fsl_file, "\tFSL_S_Clk,\n");
    fprintf(fsl_file, "\tFSL_S_Data,\n");
    fprintf(fsl_file, "\tFSL_S_Control,\n");
    fprintf(fsl_file, "\tFSL_S_Read,\n");
    fprintf(fsl_file, "\tFSL_S_Exists\n\n");

    fprintf(fsl_file, "\t);\n\n");

    fprintf(fsl_file, "\t///////////////////////////////////\n");
    fprintf(fsl_file, "\t// PARAMETER\n\n");

    fprintf(fsl_file, "\tlocalparam C_FSL_DWIDTH = %d;\n", P_FSL_WIDTH);
    fprintf(fsl_file, "\tlocalparam C_FSL_DEPTH = 16;\n");
    fprintf(fsl_file, "\tlocalparam C_FSL_ADDR = 4;\n");
    fprintf(fsl_file, "\tlocalparam C_RST_POLARITY = 1;\n\n");

    fprintf(fsl_file, "\t///////////////////////////////////\n");
    fprintf(fsl_file, "\t// Input Output Ports \n\n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_Clk;\n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tSYS_Rst;  \n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_Rst;  \n\n");

    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_M_Clk;\n");
    fprintf(fsl_file, "\tinput\t\t[C_FSL_DWIDTH-1:0]\t\t\tFSL_M_Data;\n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_M_Control;\n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_M_Write;\n");
    fprintf(fsl_file, "\toutput\t\t\t\t\t\t\tFSL_M_Full;\n\n");

    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_S_Clk;\n");
    fprintf(fsl_file, "\toutput\t\t[C_FSL_DWIDTH-1:0]\t\t\tFSL_S_Data;\n");
    fprintf(fsl_file, "\toutput\t\t\t\t\t\t\tFSL_S_Control;\n");
    fprintf(fsl_file, "\tinput\t\t\t\t\t\t\tFSL_S_Read;\n");
    fprintf(fsl_file, "\toutput\t\t\t\t\t\t\tFSL_S_Exists;\n\n");

    // Input and Output Addresses
    fprintf(fsl_file, "\t////////////////////////////////////\n");
    fprintf(fsl_file, "\t// Wires and Registers\n\n");

    fprintf(fsl_file, "\treg\t\t\t\t\t\t\tFSL_M_Full;\n");
    fprintf(fsl_file, "\treg\t[C_FSL_ADDR-1:0]\t\t\tRead_Address;\n");
    fprintf(fsl_file, "\treg\t[C_FSL_ADDR-1:0]\t\t\tWrite_Address;\n\n");

    // FSL Control Logic
    fprintf(fsl_file, "\t////////////////////////////////////\n");
    fprintf(fsl_file, "\t// Combinatorial Logic\n\n");

    fprintf(fsl_file, "\tassign FSL_S_Exists = Write_Address != Read_Address;\n");
    fprintf(fsl_file, "\tassign FSL_S_Control = FSL_M_Control;\n");

    fprintf(fsl_file, "\t////////////////////////////////////\n");
    fprintf(fsl_file, "\t// Sequantial Logic\n\n");

    // FSM for Control Logic used on FSL
    fprintf(fsl_file, "\talways @ (posedge FSL_Clk)\n");
    fprintf(fsl_file, "\tbegin\n");
    fprintf(fsl_file, "\t\tif (FSL_Rst || SYS_Rst == C_RST_POLARITY)\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tFSL_M_Full <= 0;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\t\telse\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tif ((Write_Address == Read_Address-1) ||
            (Read_Address == 0 && Write_Address == C_FSL_DEPTH-1))\n");
    fprintf(fsl_file, "\t\t\t\tFSL_M_Full <= 1;\n");
    fprintf(fsl_file, "\t\t\telse\n");
    fprintf(fsl_file, "\t\t\t\tFSL_M_Full <= 0;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\tend\n\n");

    // Read Address
```

```
    fprintf(fsl_file, "\talways @ (posedge FSL_Clk)\n");
    fprintf(fsl_file, "\tbegin\n");
    fprintf(fsl_file, "\t\tif (FSL_Rst || SYS_Rst == C_RST_POLARITY)\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tRead_Address <= 0;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\t\telse\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tif (FSL_S_Read && FSL_S_Exists)\n");
    fprintf(fsl_file, "\t\t\t\tRead_Address <= Read_Address+1;\n");
    fprintf(fsl_file, "\t\t\telse\n");
    fprintf(fsl_file, "\t\t\t\tRead_Address <= Read_Address;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\tend\n\n");

    // Write Address
    fprintf(fsl_file, "\t// Write_Address\n");
    fprintf(fsl_file, "\talways @ (posedge FSL_Clk)\n");
    fprintf(fsl_file, "\tbegin\n");
    fprintf(fsl_file, "\t\tif (FSL_Rst || SYS_Rst == C_RST_POLARITY)\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tWrite_Address <= 0;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\t\telse\n");
    fprintf(fsl_file, "\t\tbegin\n");
    fprintf(fsl_file, "\t\t\tif (FSL_M_Write && !FSL_M_Full)\n");
    fprintf(fsl_file, "\t\t\t\tWrite_Address <= Write_Address+1;\n");
    fprintf(fsl_file, "\t\t\telse\n");
    fprintf(fsl_file, "\t\t\t\tWrite_Address <= Write_Address;\n");
    fprintf(fsl_file, "\t\tend\n");
    fprintf(fsl_file, "\tend\n\n");

    // Instantiate M-LAB RAM block used in FIFO
    fprintf(fsl_file, "\tram ram (\n");
    fprintf(fsl_file, "\t\t.clock(FSL_Clk), \n");
    fprintf(fsl_file, "\t\t.data(FSL_M_Data),       // INPUT\n");
    fprintf(fsl_file, "\t\t.rdaddress(Read_Address), \n");
    fprintf(fsl_file, "\t\t.wraddress(Write_Address), \n");
    fprintf(fsl_file, "\t\t.wren(FSL_M_Write && !FSL_M_Full), \n");
    fprintf(fsl_file, "\t\t.q(FSL_S_Data),      // OUTPUT\n");
    fprintf(fsl_file, "\t);\n\n");

    fprintf(fsl_file, "endmodule");
}
```

B.11 Generate System Wrapper for Altera FPGAs (generate_system.c)

The following functions are used to generate the high level system wrapper, which instantiates all the individual elements of the NoC topology. This function is only used for Altera FPGAs, as the Quartus tools requires a complete system description as an input. For Xilinx FPGAs, the experiments are run using Xilinx EDK, which acted as the high level wrapper of the NoC topology. Individual elements are instantiated using CAD tool specific files such as the *.xmp, *.mhs, *.mss, and *.ucf files.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define P_MAX_CONNECTIONS_PER_NODE          128
#define P_MAX_NODES                     128

int multiplicand_size[128];

// Generate individual multiplier node definitions
void generate_system_node(FILE *system_file, int P_NODE, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH,
                            int P_MULTIPLICAND_WIDTH_MIN, int P_MULTIPLICAND_WIDTH_MAX){

    int multiplicand_width;

    // determine node size
    if (P_NODE == 0){
        multiplicand_width = P_MULTIPLICAND_WIDTH;
        multiplicand_size[P_NODE] = multiplicand_width;
    }else{
        multiplicand_width = int_rand((P_MULTIPLICAND_WIDTH_MAX -
                P_MULTIPLICAND_WIDTH_MIN)/2+1)*2+P_MULTIPLICAND_WIDTH_MIN;
        multiplicand_size[P_NODE] = multiplicand_width;
    }

    // Generate Multiplier Instantiations
    if (P_NODE == 0){
        fprintf(system_file, "\tinit_multiplier_%d_%d init_multiplier_%d_%d_%d
                (\n", P_FSL_WIDTH, multiplicand_width, P_FSL_WIDTH, multiplicand_width, P_NODE);
        fprintf(system_file, "\t .output_result(result_output),// tie to output port\n");
        fprintf(system_file, "\t .multiplicand(multiplicand_input),// tie to output port\n");
    }else{
        fprintf(system_file, "\tmultiplier_%d_%d multiplier_%d_%d_%d (\n",
                P_FSL_WIDTH, multiplicand_width, P_FSL_WIDTH, multiplicand_width, P_NODE);
    }

    // Generate Associated FSL Ports
    fprintf(system_file, "\t .FSL_Clk(FSL_Clk), \n");
    fprintf(system_file, "\t .FSL_Rst(FSL_Rst), \n");
    fprintf(system_file, "\t .FSL_S_Clk(FSL_S_Clk_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Exists(FSL_S_Exists_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Read(FSL_S_Read_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Data(FSL_S_Data_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Control(FSL_S_Control_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Clk(FSL_M_Clk_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Full(FSL_M_Full_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Write(FSL_M_Write_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Data(FSL_M_Data_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Control(FSL_M_Control_%d) \n", P_NODE*2);
    fprintf(system_file, "\t);\n\n");

    // Generate FSL FIFOs
    fprintf(system_file, "\tfsl fsl_%d (\n", P_NODE*2);
    fprintf(system_file, "\t .FSL_Clk(FSL_Clk), \n");
    fprintf(system_file, "\t .SYS_Rst(SYS_Rst), \n");
    fprintf(system_file, "\t .FSL_Rst(FSL_Rst), \n");
    fprintf(system_file, "\t .FSL_M_Clk(FSL_M_Clk_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Data(FSL_M_Data_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Control(FSL_M_Control_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Write(FSL_M_Write_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_M_Full(FSL_M_Full_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Clk(FSL_S_Clk_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Data(FSL_S_Data_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Control(FSL_S_Control_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Read(FSL_S_Read_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Exists(FSL_S_Exists_%d)\n", P_NODE*2);
    fprintf(system_file, "\t);\n\n");

    fprintf(system_file, "\tfsl fsl_%d (\n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_Clk(FSL_Clk), \n");
    fprintf(system_file, "\t .SYS_Rst(SYS_Rst), \n");
    fprintf(system_file, "\t .FSL_Rst(FSL_Rst), \n");
    fprintf(system_file, "\t .FSL_M_Clk(FSL_M_Clk_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Data(FSL_M_Data_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Control(FSL_M_Control_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Write(FSL_M_Write_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Full(FSL_M_Full_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Clk(FSL_S_Clk_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Data(FSL_S_Data_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Control(FSL_S_Control_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Read(FSL_S_Read_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Exists(FSL_S_Exists_%d)\n", P_NODE*2+1);
    fprintf(system_file, "\t);\n\n");
```

```
}

// Generates Switch Instantiations
void generate_system_switch(FILE *system_file, int P_NODE,int P_NODE_SIZE,
            int P_NODE_CONNECTIONS[P_MAX_CONNECTIONS_PER_NODE],
            int P_NODE_CHANNELS[P_MAX_CONNECTIONS_PER_NODE]){

    int i;

    fprintf(system_file, "\tswitch_%d #(\n", P_NODE_SIZE);
    fprintf(system_file, "\t .C_DEST_L(%d),  \n",P_NODE);
    fprintf(system_file, "\t .C_DEST_H(%d))  \n",P_NODE);
    fprintf(system_file, "\tswitch_%d_%d (\n", P_NODE_SIZE, P_NODE);
    fprintf(system_file, "\t .FSL_Clk(FSL_Clk),  \n");
    fprintf(system_file, "\t .FSL_Rst(FSL_Rst),  \n");
    fprintf(system_file, "\t .FSL_M_Clk(FSL_M_Clk_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Data(FSL_M_Data_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Control(FSL_M_Control_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Write(FSL_M_Write_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_M_Full(FSL_M_Full_%d), \n", P_NODE*2+1);
    fprintf(system_file, "\t .FSL_S_Clk(FSL_S_Clk_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Data(FSL_S_Data_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Control(FSL_S_Control_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Read(FSL_S_Read_%d), \n", P_NODE*2);
    fprintf(system_file, "\t .FSL_S_Exists(FSL_S_Exists_%d), \n", P_NODE*2);

    // Connect all Switches
    for (i=0;i<P_NODE_SIZE;i++){

        // output ports associated to own switch
        fprintf(system_file, "\t .ch%d_out_data(switch_%d_ch%d_data),\n", i, P_NODE, i);
        fprintf(system_file, "\t .ch%d_out_ctrl(switch_%d_ch%d_ctrl), \n", i, P_NODE, i);
        fprintf(system_file, "\t .ch%d_out_exists(switch_%d_ch%d_exists), \n", i, P_NODE, i);
        fprintf(system_file, "\t .ch%d_out_read(switch_%d_ch%d_read), \n", i, P_NODE, i);

        // input ports connected to desired connection
        fprintf(system_file, "\t .ch%d_in_data(switch_%d_ch%d_data),\n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        fprintf(system_file, "\t .ch%d_in_ctrl(switch_%d_ch%d_ctrl), \n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        fprintf(system_file, "\t .ch%d_in_exists(switch_%d_ch%d_exists), \n", i,
            P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);

        if (i==P_NODE_SIZE-1){
            fprintf(system_file, "\t .ch%d_in_read(switch_%d_ch%d_read) \n", i,
                P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        }else{
            fprintf(system_file, "\t .ch%d_in_read(switch_%d_ch%d_read), \n", i,
                P_NODE_CONNECTIONS[i], P_NODE_CHANNELS[i]);
        }
    }
    fprintf(system_file, "\t);\n\n");
}

// Generate header for System file defining all input and output ports of system
void generate_system_header(FILE *system_file, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH){

    // Generate Header
    fprintf(system_file, "///////////////////////////////////////////////////\n");
    fprintf(system_file, "// system.v\n");
    fprintf(system_file, "////////////////////////////////////////////////////\n\n");

    fprintf(system_file, "module system(\n");
    fprintf(system_file, "\tresult_output,\n");
    fprintf(system_file, "\tmultiplicand_input,\n");
    fprintf(system_file, "\tsys_clk_pin,\n");
    fprintf(system_file, "\tsys_rst_pin\n");
    fprintf(system_file, ");\n\n");

    fprintf(system_file, "/////////////////////////////////////////////////////\n");
    fprintf(system_file, "// Parameters\n\n");

    fprintf(system_file, "\tlocalparam C_FSL_DWIDTH = %d;\n", P_FSL_WIDTH);
    fprintf(system_file, "\tlocalparam C_MULTIPLICAND_DWIDTH = %d;\n\n", P_MULTIPLICAND_WIDTH);

    fprintf(system_file, "////////////////////////////////////////////////////\n");
    fprintf(system_file, "// Input Output\n\n");

    fprintf(system_file, "\toutput\t[C_FSL_DWIDTH-1:0]\t\tresult_output;\n");
    fprintf(system_file, "\tinput\t\t[C_MULTIPLICAND_DWIDTH-1:0]\tmultiplicand_input;\n");
    fprintf(system_file, "\tinput\t\t\t\t\t\t\tsys_clk_pin;\n");
    fprintf(system_file, "\tinput\t\t\t\t\t\t\tsys_rst_pin;\n\n");

    fprintf(system_file, "///////////////////////////////////////////////////////\n");
    fprintf(system_file, "// Wires\n\n");

    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_Clk;\n");
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_Rst;\n");
    fprintf(system_file, "\twire\t\t\t\t\t\t\tSYS_Rst;\n\n");
}

// Generate all wires used by FSL Links
void generate_fsl_wires(FILE *system_file, int P_NODE, int P_FSL_WIDTH){

    // FSL Pins
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_M_Clk_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t[C_FSL_DWIDTH-1:0]\t\tFSL_M_Data_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_M_Control_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_M_Write_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_M_Full_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_S_Clk_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t[C_FSL_DWIDTH-1:0]\t\tFSL_S_Data_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_S_Control_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_S_Read_%d;\n",P_NODE);
    fprintf(system_file, "\twire\t\t\t\t\t\t\tFSL_S_Exists_%d;\n\n",P_NODE);
}

// Generate all wires used by Switches
void generate_switch_wires(FILE *system_file, int P_NODE,int P_NODE_SIZE){

    int i;

    for (i=0;i<P_NODE_SIZE;i++){
        fprintf(system_file, "\twire\t\t[C_FSL_DWIDTH-1:0]\t\tswitch_%d_ch%d_data;\n",P_NODE,i);
        fprintf(system_file, "\twire\t\t\t\t\t\t\tswitch_%d_ch%d_ctrl;\n",P_NODE,i);
        fprintf(system_file, "\twire\t\t\t\t\t\t\tswitch_%d_ch%d_exists;\n",P_NODE,i);
        fprintf(system_file, "\twire\t\t\t\t\t\t\tswitch_%d_ch%d_read;\n",P_NODE,i);
    }

    fprintf(system_file, "\n\n");

}

// Main function used to generate the top level system wrapper file defining
// the NoC for Altera FPGAs
void generate_system_verilog(FILE *system_file, int P_NODE_SIZE[P_MAX_NODES],
                int P_NODE_CONNECTIONS[P_MAX_NODES][P_MAX_NODES],
                int P_NODE_CHANNELS[P_MAX_NODES][P_MAX_NODES],
                int P_NUM_NODES, int P_FSL_WIDTH, int P_MULTIPLICAND_WIDTH,
                int P_MULTIPLICAND_WIDTH_MIN, int P_MULTIPLICAND_WIDTH_MAX){

    int i,j;
    int P_CONNECTIONS[P_MAX_CONNECTIONS_PER_NODE];
    int P_CHANNELS[P_MAX_CONNECTIONS_PER_NODE];


    // generate header, io ports
    generate_system_header(system_file, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH);

    //generate fsl_wires
    for (i=0;i<P_NUM_NODES*2;i++){
        generate_fsl_wires(system_file, i, P_FSL_WIDTH);
    }

    // generate switch wires
    for (i=0;i<P_NUM_NODES;i++){
        generate_switch_wires(system_file, i, P_NODE_SIZE[i]);
    }

    fprintf(system_file, "//////////////////////////////////////////////////\n");
    fprintf(system_file, "// Assigns\n\n");

    fprintf(system_file, "\tassign FSL_Clk = sys_clk_pin;\n");
    fprintf(system_file, "\tassign FSL_Rst = sys_rst_pin;\n");
    fprintf(system_file, "\tassign SYS_Rst = sys_rst_pin;\n\n");

    fprintf(system_file, "/////////////////////////////////////////////////\n");
    fprintf(system_file, "// Sub Components\n\n");

    //generate nodes
    for (i=0;i<P_NUM_NODES;i++){
        if (i==0){
            generate_system_node(system_file, i, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
                        P_MULTIPLICAND_WIDTH_MIN, P_MULTIPLICAND_WIDTH_MAX);
        }else{
            generate_system_node(system_file, i, P_FSL_WIDTH, P_MULTIPLICAND_WIDTH,
                        P_MULTIPLICAND_WIDTH_MIN, P_MULTIPLICAND_WIDTH_MAX);
        }
    }

    // generate switches
    for (i=0;i<P_NUM_NODES;i++){

        for (j=0;j<P_NODE_SIZE[i];j++){
            P_CONNECTIONS[j] = P_NODE_CONNECTIONS[i][j];
            P_CHANNELS[j] = P_NODE_CHANNELS[i][j];
        }
        generate_system_switch(system_file, i, P_NODE_SIZE[i], P_CONNECTIONS, P_CHANNELS);
    }

     fprintf(system_file, "endmodule\n\n");

}
```