

NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Packet-Based Transaction Interconnect Fabric for FPGA Systems on Chip

by

Kent Orthner

Thesis submitted to the Faculty of the Department of Electronics
at Carleton University, Ottawa in partial fulfillment of
the requirements for the degree of

Master of Applied Science

in

Electrical Engineering

March 9th, 2009

Department of Electronics, Carleton University

Ottawa, Ontario

Copyright © 2009, Kent J. Orthner



Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-60222-5
Our file Notre référence
ISBN: 978-0-494-60222-5

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

■+■
Canada

Abstract

As the logic available in FPGA devices grows, traditional bus interconnects are struggling to keep up, and higher performance interconnects are required.

This thesis presents introduces "Merlin", a flexible, high performance, and light weight approach to on-chip FPGA interconnect, which includes a collection of packet based primitives, and a way to map transactions to packets, and algorithms for connecting components such that they provides the services required of a transaction interconnect.

A prototype implementation is reported, describing the components that compose the interconnect, the interfaces between them, the algorithms used to assemble the interconnect, the development and testing methodology, and a proof of concept system.

Results show that for large systems, a Merlin interconnect provides higher frequency with less logic than the interconnect tools available from FPGA vendors. In addition, Merlin provides lower latency through the network than other FPGA NoC implementations, while costing less resources, at a higher frequency.

Acknowledgments

A tremendous amount of thanks goes to my wife Miki and my son Kaz. The work presented in this thesis would not have been possible without their love, support, encouragement, and patience. I also owe a debt of gratitude to my mother for teaching me the importance of education, and that I really could do anything I wanted to, and to my father for setting me upon my career path and pointing me in the right direction.

I would also like to thank my advisor, Dr. Maitham Shams at Carleton University for his support and advice in the completion of this thesis.

Table of Contents

Abstract.....	ii
Acknowledgments.....	iii
Table of Contents.....	iv
1. Introduction	1
1.1 Motivation	2
1.2 Statement of Thesis	4
1.3 Approach	4
1.4 Thesis Organization	5
2. Background and Related Work.....	7
2.1 On Chip Interconnects.....	7
2.1.1 Streaming Connections.....	8
2.1.2 Transaction Connections	9
2.1.3 Interconnects	9
2.1.4 Transaction Interconnect Protocols.....	11
2.1.5 Transaction Interconnect Services	15
2.1.6 Transaction Interconnect Topologies	21
2.1.7 Arbitration.....	28
2.1.8 Burst Transactions.....	31
2.2 Networks on chip	33

2.2.1	On-chip and off-chip networks differences and similarities.....	34
2.2.2	Network Switching Algorithms	37
2.2.3	Topologies	38
2.3	Field Programmable Gate Arrays	43
2.3.1	FPGA Structure.....	44
2.3.2	FPGA Design Flow	46
2.3.3	FPGA Considerations for NoC design	47
2.4	Related Work	49
3.	Merlin Architecture	54
3.1	Objectives	54
3.2	Feature Set	56
3.3	Architecture Overview	58
3.4	Interfaces	61
3.4.1	Transaction Interfaces	62
3.4.2	Transaction Packet Interfaces	67
3.4.3	Packet Transport Interfaces	73
3.5	Components	74
3.5.1	Master Network Interface Component	75
3.5.2	Slave Network Interface	78
3.5.3	Width Adapter.....	79
3.5.4	Burst Adapter	79
3.5.5	Twin Fabric.....	81
3.5.6	FIFOs.....	83
3.5.7	Register Stage	83
3.6	Assembly Algorithms.....	83
4.	Merlin Prototype Implementation.....	87
4.1	Motivation	87

4.2	Methodology	88
4.2.1	Software Development and Testing	89
4.2.2	Component Generation.....	89
4.2.3	System Representation	92
4.2.4	Tool Integration.....	93
4.3	Component Implementation	94
4.3.1	Twin Fabric component.....	94
4.3.2	Master Network Interface	99
4.4	Component Verification	104
4.4.1	Verification Coverage	107
4.5	Transform Implementation and Verification.....	109
4.5.1	Find Avalon MM Domains Transform.....	109
4.6	Simple Processor Test System.....	112
4.6.1	Implementation	114
4.6.2	Test Program.....	117
4.6.3	Test Flow	117
4.7	Measurements & Results.....	119
4.7.1	Crossbar Switch Performance Measurement.....	119
4.7.2	Network Interface Performance Measurement	122
4.7.3	Comparison to SOPC Builder Generated Interconnect.....	124
4.7.4	Comparison to other Network On Chip Implementations.....	127
5.	Conclusion	134
5.1	Summary.....	135
5.2	Contributions	136
5.3	Future Work.....	137
	References	138

Table of Figures

Figure 1 - Streaming connections.....	8
Figure 2 - Logical view of a simple embedded system	9
Figure 3 - Physical view of a simple embedded system.....	10
Figure 4 - Shared Bus Topology	22
Figure 5 - Shared Bus Topology Multiplexer Implementation.....	23
Figure 6 - Two-level Hierarchical Bus	24
Figure 7 - Segmented hierarchical bus	25
Figure 8 - Cross-connect Topology.....	26
Figure 9 - Message-dependant deadlock example [25]	36
Figure 10 - 2D Mesh Topology.....	39
Figure 11 - 2D Torus Topology	39
Figure 12 - 2-ary 4-flies butterfly network topology.....	40
Figure 13 - 2D Flattened butterfly.....	41
Figure 14 - 5 Stage Clos Network.....	42
Figure 15 - 4 Master, 12 Slave Command & Response Butterfly Networks.....	43
Figure 16 - Stratix II FPGA Block Diagram [36]	44
Figure 17 - Stratix II Logic Element [36]	45
Figure 18 - FPGA Design Flow [37].....	46
Figure 19 – Performance summary for existing solutions	50
Figure 20 - Interconnect Implementation	59
Figure 21 - Merlin Interfaces	61
Figure 22 - Simple Avalon MM Transactions [47]	65

Figure 23 - Avalon MM Write Burst Transaction [47]	65
Figure 24 - Avalon MM Read Burst Transaction [47]	66
Figure 25 - Example Write Command Packet	70
Figure 26 - Example Read Burst Command Packet	70
Figure 27 - Example Read Burst Response Packet	71
Figure 28 - Merlin Structure	75
Figure 29 - Effect of Domain Definition Algorithm	84
Figure 30 - Effect of Network Creation Algorithm	85
Figure 31 - Effect of Network Topology Creation Algorithm	86
Figure 32 - Twerp Algorithm Pseudo- Code	90
Figure 33 - Twerp SystemVerilog Source Code Example	91
Figure 34- Twerp SystemVerilog Output	91
Figure 35 - Structural Module Hierarchy	92
Figure 36 – 2 in x 2 out Combinatorial Crossbar Switch	95
Figure 37 - Multiplexer Implementation	97
Figure 38 - Arbitration Implementation	98
Figure 39 - Example arbitration	99
Figure 40- Master Network Interface	99
Figure 41 - Width Adaptation	102
Figure 42 - Slave Network Interface	103
Figure 43 - Slave Agent	104
Figure 44 - JUnit Simulation Test Run	106
Figure 45 - JUnit Simulation Output	107
Figure 46 - Pre-transformation system	110
Figure 47 - Result of Find Avalon MM Domains Transform	111
Figure 48 - Result of Implement Fabric From Empty SM Transform	112
Figure 49 - Simple Processor System	113
Figure 50 - Simple Processor System in SOPC Builder	115
Figure 51 - Photograph of Simple System Test Board	115
Figure 52 - Simple Processor System Resource Utilization	116

Figure 53 - Console output from Simple Processor System Test Program	118
Figure 54 - Crossbar Switch Frequency as a function of the Number of Outputs.	120
Figure 55 - Crossbar Switch Frequency as a function of the Number of Inputs.....	121
Figure 56 - Crossbar Switch LUT Utilization (LUTs).	121
Figure 57 - Crossbar Switch Register Utilization (Registers).....	122
Figure 58 - Master Network Interface Frequency & Logic Utilization.....	123
Figure 59 - Slave Network Interface Frequency & Logic Utilization.....	123
Figure 60 - Slave Network Interface Frequency & Logic Utilization.....	125
Figure 61 - Merlin vs. SOPC Builder Interconnect Frequency.....	126
Figure 62 - Merlin vs. SOPC Builder Resource Utilization	127
Figure 63 - Comparison with other Network on Chip implementations	128
Figure 64 - Resource Utilization Comparison	129
Figure 65 - Frequency Comparison.....	130
Figure 66 - Latency Comparison.....	132

Table of Acronyms

AHB	<i>Advanced High performance Bus.</i> An on-chip bus standard, and member of the AMBA family of interconnects.
AMBA	<i>Advanced Microcontroller Bus Architecture.</i> A family of on-chip interface standards released by ARM, which has become the de facto standard for on-chip communication.
APB	<i>Advanced Peripheral Bus.</i> An on-chip bus standard, and member of the AMBA family of interconnects.
ASB	<i>Advanced System Bus.</i> An on-chip bus standard, and member of the AMBA family of interconnects.
ASIC	<i>Application Specific Integrated Circuit.</i>
AXI	<i>Advanced eXtensible Interface.</i> An on-chip bus standard, and member of the AMBA family of interconnects.
BFM	<i>Bus Functional Model.</i>
CPU	<i>Central Processing Unit.</i>
DDR	<i>Dual Data Rate.</i> A common off-chip memory standard.
FIFO	<i>First In First Out.</i> A FIFO is a type of queue that has the characteristics of returning data in the same order as it was added.
FPGA	<i>Field Programmable Gate Array.</i>
HDL	<i>Hardware Description Language.</i>
IDE	<i>Integrated Development Environment.</i>
IP	<i>Intellectual Property.</i> An IP core is a pre-verified pre-designed circuit block that can be used to build an application

JTAG	<i>Joint Test Access Group.</i> JTAG is used to refer to the IEEE 1149.1 standard for IC test access ports.
LE	<i>Logic Element.</i> The basic building block of an Altera FPGA.
LUT	<i>Look Up Table.</i>
MNI	<i>Master Network Interface.</i>
NI	<i>Slave Network Interface.</i>
NoC	<i>Network on Chip.</i>
OCP	<i>Open Core Protocol.</i> OCP is an on-chip interface specification that enables standardized definitions of an IP core's on-chip interfaces.
PLB	<i>Processor Local Bus.</i> The high-performance member of the CoreConnect family of on-chip busses defined by IBM.
PLL	<i>Phase Locked Loop.</i>
RAM	<i>Random Access Memory.</i>
RTL	<i>Register Transfer Level.</i> RTL refers to HDL code that is written to be synthesized into silicon logic.
SNI	<i>Slave Network Interface.</i>
SoC	<i>System on Chip.</i>
SRAM	<i>Static Random Access Memory.</i>
TCL	<i>Tool Control Language.</i> A computer programming language that is common used in the Electronic Design Industry.
QoS	<i>Quality of Service.</i>
UART	<i>Universal Asynchronous Receiver Transmitter.</i>
USB	<i>Universal Serial bus.</i>

1. Introduction

Since FPGAs were first introduced, FPGA usage has increased from simple glue logic applications to entire System on Chip (SoC) designs. FPGAs are common components in embedded systems, and often implement the entire embedded system, including CPUs, on-chip memories, and complex peripherals. The interconnect technology used by FPGA system-on-chip implementations has followed the interconnect trends used by Application Specific Integrated Circuits (ASICs), and printed circuit boards before that, beginning first with shared bus implementations, and moving towards implementations supporting multiple masters, heterogeneous components, and concurrent transactions.

Integrated Circuit (IC) manufacturing technology is expected to soon provide us with in excess of 4 billion transistors on a single chip [1], and FPGAs will soon support in excess of five hundred thousand logic elements [2]. Synthesis and compiler development is not able to keep pace with this growth [1], leaving system designers facing the so-called *productivity gap*, the gap between the capability of modern FPGAs and ICs, and a designer's ability to integrate and manage the designs that fill them. The reuse of more and more complex on-chip intellectual property (IP) blocks, or components has been the

main mechanism to increase productivity, and it is expected that by 2010, 40% of all logic size will be composed of reused design blocks [1]. A scalable and flexible inter-design block communications strategy is required to harness the power of these diverse and ever more complex reusable design blocks. Network on chip technology promises to provide a flexible platform that addresses these needs [3].

1.1 Motivation

As the size of Field Programmable Gate Arrays (FPGAs) approach a million logic elements, the number of components on a device keeps growing, and FPGA designs with dozens of processors are no longer unheard of. The traditional bus-based interconnect with centralized arbitration will no longer be able to manage the complex connections that this rich set of components will demand. Connections between components will have to consist of multiple pipeline stages [4], and components will need the ability to autonomously initiate data transfer without knowing if they've been granted access to the destination. To support this, the communication network will need to be fully distributed, with little or no global coordination [5].

The two leading FPGA companies have a combined FPGA market share of over 80% [6], and both provide solutions for FPGA developers designing system-on-chip solutions. Altera's SOPC Builder product uses the Altera proprietary "Avalon" protocol to connect on-chip components to one another [7]. This protocol provides *slave-side arbitration*, which supports maximum concurrency, often at the expense of resource utilization and achievable frequency, while allowing the system designer little control over implementation. Xilinx's PlatformStudio product uses IBM's CoreConnect protocol for

on-chip communications [8], an industry standard protocol developed to interface to the PowerPC processor family. CoreConnect is targeted to ASIC designs, and was not originally intended for FPGA implementations.

Neither of these solutions provides system-on-chip designers with the performance and flexibility that FPGA fabrics promise. A high-performance FPGA-optimized flexible and modular interconnect is required to support FPGA-based system on chip designs for the coming years.

A Network-on-Chip (NoC) approach to on-chip FPGA interconnect promises to be a flexible and high-performance strategy, allowing the interconnect to take advantage of the flexibility of FPGA devices. Existing bus-based interconnects are not appropriate for chips with a high component count, because of the diverse and dynamic communication requirements of advanced current and future applications. Networks on a chip are emerging as an alternative to existing on-chip interconnects because they structure and manage global wires, are scalable when compared to traditional buses, decouple communications from computation, and use individual protocol layers that each provide a well-defined interface, decoupling service usage from implementation [9]. NOC implementations for FPGAs that have been reported include complex network nodes that achieve unsatisfactory performance and that aren't appropriate for an FPGA cost model.

1.2 Statement of Thesis

This thesis introduces a flexible, high performance, and light weight approach to on-chip interconnect for System-on-Chip (SoC) Intellectual Property (IP) integration. The interconnect includes a collection of packet based primitives, a way to map transactions to packets, and algorithms for connecting components such that they provides the services required of a transaction interconnect. The resultant interconnect is able to support transactions issued by transaction interface specifications, such as AMBA's AHB, Wishbone, Altera's Avalon, and OCP. A prototype interconnect implementation and a simple processor system are used to prove that the of concept of using Network-On-Chip primitives to support an FPGA SoC is sound. This same prototype implementation provides higher frequency with lower logic utilization and lower latency than the interconnect generation tools available from FPGA vendors and other FPGA NoC implementations.

1.3 Approach

To accomplish this objective, we investigate consider the interconnects available from FPGA vendors system design tools, and from FPGA NoC implementations that have been published in the literature.

We then develop the architecture of a new high-performance and modular interconnect solution for FPGA-based system on chips. This interconnect, named "Merlin", provides a flexible feature set to support a wide variety of applications while optimizing for the relatively high cost and low performance of FPGA logic. It takes advantage of NoC interconnect techniques, such as using packets to encapsulate commands and responses.

Merlin uses packets to encapsulate commands and responses, but unlike some other NoC implementations, uses separate packet networks for commands and responses [4] [10]. We detail the different layers of functionality that compose a Merlin interconnect, and present some design tradeoffs considered in the creation of this architecture. We also describe the components that implement the required functionality, the packet and transaction interfaces between component, and the algorithms used to implement an application specific interconnect implementation given a description of desired connectivity.

To demonstrate Merlin, a prototype interconnect generator is implemented, including all the required components and algorithms for a proof-of-concept system. Along with the implementation, we outline the toolset and component development and testing methodology. We use this prototypical implementation to generate a proof-of-concept system including a processor core with instruction and data masters, and a number of peripherals such as an on-chip memory core for program and data storage, and a Universal Synchronous Receiver Transmitter (UART) core for communications. Finally, this proof-of-concept system will be tested in FPGA hardware by running a simple test program on the processor. We also generate a number of synthetic systems that vary the number of masters and slaves in the system, as well as other variables, in order to come up with performance measurements that can be compared to previous implementations.

1.4 Thesis Organization

This thesis is divided into 5 major chapters. Chapter 1 is the present chapter, and describes the motivation and objectives of this work. Chapter 2 provides background

information for this work, including a discussion of FPGA technology, SoC interconnects, and NoC technology on FPGAs. Chapter 3 proposes an architecture for a new FPGA optimized on-chip interconnect that features higher performance and modularity, and that supports the considerable flexibility that FPGAs allow, while allowing for low-cost implementations. Chapter 4 describes a prototype implementation of this new interconnect architecture, including a design methodology and testing framework, a proof-of -concept design, and measurement results. Chapter 5 summarizes the research, and discusses future work.

2. Background and Related Work

In this section, we provide background information about on chip interconnects for SoC designs, focusing on interconnect types, topologies, arbitration algorithms, and burst transactions. We then present Network-on-Chips, discussing how they can be applied to on-chip interconnects, and how they are different from traditional inter-chip networks. finally, we introduce FPGAs, and compare & contrast them with ASICs, and discuss the some additional considerations for FPGA interconnects.

2.1 On Chip Interconnects

On-chip interconnects are used to connect different intellectual property (IP) cores within a chip, where a *core* is defined as a pre-designed, pre-verified silicon circuit block that can be used to build an application [11]. There are many different types of cores available to today's system designers, including analog and digital cores, processor cores, communication interface cores, and DSP cores. It has become a common with both FPGA and ASIC development to approach the design problem as an assembly of IP cores, where some cores are particular to the given design, and some are standard cores, including those available off-the-shelf from IP core providers. A system, however, is not

made up of cores alone; it must include a mechanism for the cores to communicate with one another.

Most inter-core communication takes one of two forms: as unidirectional streams of data from a core to an adjacent core, or as transactions issued by masters and responded to by slaves. Additional forms of communication within an on-chip system include clock and reset connectivity, interrupts, and connections for test & measurement. In this thesis, we focus on transaction connections, and how they can be supported by using NoC techniques, taking advantage of streaming connections in the implementation.

2.1.1 Streaming Connections

Streaming connections such as those shown in Figure 1 provide a unidirectional flow of data from a source interface to a sink interface, and support a backpressure mechanism to allow the sink to refuse data from the source. Streaming connections are often used in processing chains, where a flow of data from a source such as a video camera is passed through a series of processing blocks that apply transformations to the data, eventually to a sink. Streaming connections are also typically used in telephony and data communications equipment, where packets or frames are received from a physical interface, are processed, and are then sent on their way via a different physical interface.

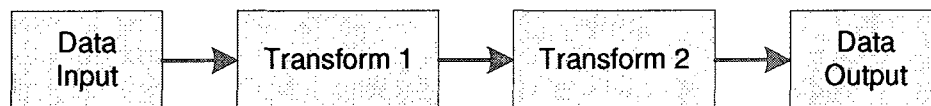


Figure 1 - Streaming connections

2.1.2 Transaction Connections

With transaction-based communication, masters initiate transactions such as read and write, and slaves respond to them by returning the requested read data, by accepting the write data, or by returning an acknowledgement or error. Masters typically are connected to multiple slaves, and slaves are often connected to multiple masters. A system with transaction connections is shown in Figure 2.

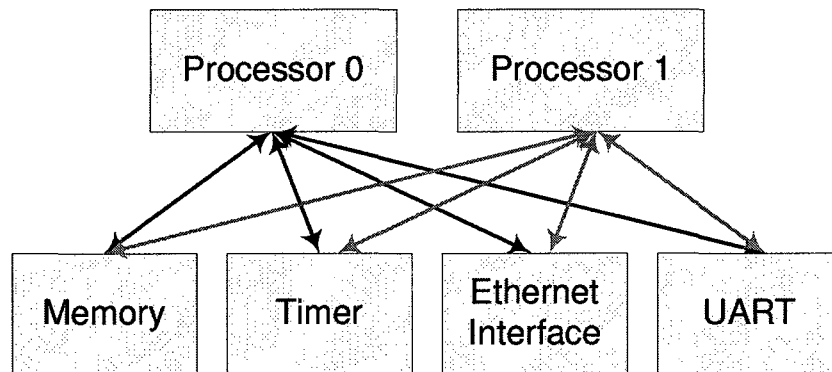


Figure 2 - Logical view of a simple embedded system

2.1.3 Interconnects

Both of the connection types described above represent not a physical connection between two cores, but a logical one. To implement the system shown in Figure 2, a system designer would not use a processor that had a master interface for each of the four slaves, nor would he or she use a memory with a slave interface for each master. Instead, each master core exposes a single master interface, and each a slave core exposes a single slave interface.

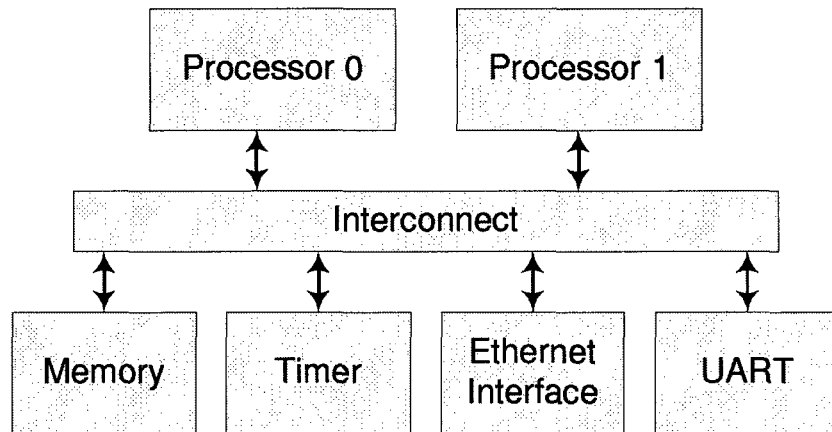


Figure 3 - Physical view of a simple embedded system

An interconnect such as that shown in Figure 3 is used to provide for the logical communications between components, abstracting away implementation details. The functional components communicate with the interconnect component, and hence with each other, via *interfaces*. *Connections* are used to provide communication channels between interfaces. Connections are point-to-point, with exactly one start and one end. *Abstract connections* are connections that are implemented using other components, and where the interfaces they connect do not necessarily share any wires. For example, Figure 2 shows an abstract connection between Processor 0 and the memory whose realization is shown in Figure 3 as a physical connection from Processor 0 to the interconnect, and another from the interconnect to the memory. The connections shown in Figure 3 are themselves implemented in an FPGA by connections between logic elements across multiple configurable routing blocks.

2.1.4 Transaction Interconnect Protocols

To increase the opportunities for a core to be included in different systems, core designers often design their cores such that the interfaces adhere to standard protocols, allowing them to be easily integrated with interconnects that supports that protocol. In preparation for a discussion of transaction interconnect services, we introduce a number of interconnect protocols and specifications that exist in the industry today.

2.1.4.1 AMBA Family of Protocols

The Advanced Microcontroller Bus Architecture (AMBA) family of protocols from ARM has become the de facto standard for on-chip ASIC communication [12]. In 1996, ARM introduced the first members of the family, the Advanced System Bus (ASB) and Advanced Peripheral Bus (APB) protocols. The Advanced High Performance Bus (AHB), a single clock-edge version of ASB, was introduced in 1999, followed by Multi-Layer AHB, which supports hierarchical bus topologies. In 2003, ARM introduced the Advanced eXtensible *Interface* (AXI), a feature-rich protocol that supports high throughput multi-master systems with high concurrency. In 2006, AHB-Lite was released, an enhancement to AHB that frees AHB masters from any knowledge of the topology or arbitration implementation. The following provides a brief discussion of each member of the AMBA interconnect family.

Advanced Peripheral Bus (APB) - APB is a low cost synchronous interface to low-bandwidth peripherals, optimized for power consumption and reduced interface complexity [12].

Advanced System Bus (ASB) - ASB was introduced at the same time as APB, and is a higher performance bus that supports multiple co-existing masters, pipelined operation, and burst transfers. There is a central arbiter that accepts requests and grants access to a single master. The granted master initiates transfers on the bus, and the slave provides a transfer response back to the bus master. ASB has been largely replaced by AHB.

Advanced High Performance Bus (AHB) - AHB is newer than ASB, with higher performance, but is in many ways a simplification of ASB. It supports multiple co-existing masters, pipelined operation, burst transfers, and split transactions. Like ASB, AHB supports data paths of up to 128 bits, and there is a central arbiter that grants access to the entire bus to a single master. Unlike ASB, AHB uses a non-tristate implementation, simplifying component design.

Multi-Layer AHB - Multilayer AHB, introduced in 2001, is a refinement to AHB that uses a more complex topology scheme to support multiple parallel paths between masters and slaves in a system [13]. It supports the development of complex multi-master systems with complex interconnects, while still supporting components with AHB masters and slaves.

AHB-Lite - AHB-Lite is a further refinement of AHB that supports a single bus master. AHB-Lite transfers are the same as AHB transfers, except that there is no requirement for arbitration signals, simplifying the master interface. “Multi-Layer AHB-Lite” is AHB-Lite where each master has a private bus. Multi-Layer AHB Lite supports full concurrency, where if two masters try to access the same slave at the same time, one of

the masters will be backpressured by the interconnect until the other has completed it's transaction.

The difference between Multi-Layer AHB-Lite and AHB is that with Multi-Layer AHB-Lite, the system topology is not specified. It is instead left as a consideration for the system designer, allowing greater freedom of implementation. In addition, master implementations are simpler because there is no requirement for them to deal with signals to and from the arbiter.

Advanced eXtensible Interface (AXI) - The AMBA AXI protocol is a high-performance interface definition with a number of features that make it suitable for high-speed interconnect. The defining characteristic of AXI is that it uses five channels to carry out transactions, each with independent handshaking and backpressure. The presence of five different channels allows for a maximum of concurrency, while their independence allows for simplicity of implementation [14].

2.1.4.2 Wishbone

The wishbone specification is a public domain system-on-chip interconnect protocol originally developed by Silicore, now owned by OpenCores.org. The goals of the wishbone specification are to provide a standard, flexible open-source interface definition to enforce compatibility between IP cores [15]. The opencores.org website includes a wide selection of freely available cores for inclusion in ASIC & FPGA SoC designs.

2.1.4.3 OpenCore Protocol (OCP)

OCP was originally developed by Sonics, and has been donated to OCP-IP, a non-profit company promoting the OCP protocol as a standard to ensure creation and integration of interoperable components. OCP supports more than just read and write interfaces; it also supports configurable sideband control signaling and test harness signals. OCP strives to define protocols to unify all of the inter-core communication [16].

The OCP transaction protocol supports synchronous unidirectional signaling, pipelined transfers, burst support, atomic transactions, sideband signals for flow control, and tag, thread, and connection identifiers to support out-of order transactions. OCP has a small set of required signals with a wide range of optional signals and parameters to define the interface on any given component, allowing cores to only implement the subset of the protocol that applies to them.

2.1.4.4 Avalon

The Avalon Memory Mapped specification was first released by Altera with the SOPC Builder system integration tool. Avalon was originally developed to connect peripherals to Altera's Nios & Nios II soft processor cores, was designed from the ground up to target FPGAs, and has considerable flexibility in its configuration. The Avalon specification describes a configurable interface between a master or slave and the interconnect, allowing a component designer to use only the signals and parameters that are applicable to their component.

2.1.4.5 CoreConnect

The CoreConnect family of on-chip busses was developed by IBM for the PowerPC 405 processor core. It consists of three-levels: the processor local bus (PLB), the on-chip peripheral bus, and the device control register bus, each of which supports read and write transactions. CoreConnect supports data bus widths of 32 bits and higher, uses separate read and write data signals, and allows multiple masters. It provides for high performance features such as pipelining, split transactions and burst transfers [17].

Beginning in 2000, Xilinx began promoting CoreConnect for FPGA peripheral components [18], and it has become the standard interface for Xilinx embedded processor designs, including those using the MicroBlaze soft processor and the PowerPC Hard IP processor [8].

2.1.5 Transaction Interconnect Services

To support transaction connections, a transaction interconnect provides *transaction services*. The definition of these services is a contract between the components and the interconnect, and the implementation of the interconnect determines how each service is provided. For transaction connections, the interconnect is responsible for seeing that each command and its data gets to the selected component, and that the response and its data gets back to the master. Services provided by a transaction interconnect typically include the following.

Transaction Types - Typical interfaces support transaction types such as posted write, where once the write command is issued, it is assumed complete by the master, and non-

posted reads and writes, where read data or a write acknowledgement is returned for every command. For higher throughput, interconnects also often support burst transactions, where multiple cycles of read or write data are transferred with a single command.

Address Decoding - In most embedded systems, a master selects a given slave by reading or writing to an address within the slave's address range. In order to correctly route the transaction, the interconnect decodes the address, converting it into a destination ID. The least significant bits are carried unchanged to the slave, and are used by the slave to address an internal register or memory cell.

Arbitration - If the system contains multiple masters, there is the possibility that more than one master simultaneously requests access to a shared resource, such as a slave interface or bus segment. To handle this, an interconnect provides an arbitration service, deciding which master gets access to the resource.

Command and Write Data Routing - Once the destination slave interface has been determined, and the master has won the arbitration, the interconnect routes the command, and in the case of a write command, the data that goes along with it, to the correct slave interface.

Response Routing - Once the slave has received the command and responded to it, the response must be routed back to the master. The response typically consists of read data or a write acknowledgement.

Adaptation - Adaptation refers to the service by which differences between master and slave interface implementation are accounted for. In many systems, the data widths of master and slave interfaces differ, with simple components having data widths of as narrow as 8 bits, and high-throughput components having data widths wider than 128 bits. A transaction interconnect might provide a data width adaptation service, allowing components with different data widths to connect to one another.

Clock Crossing - In many systems, different master and slave interfaces may be on different clock domains. In this case, the interconnect must include clock crossing logic that allows the command issued on one clock domain to be received by a slave on another clock domain, and the response from the slave on the second clock domain to be received by the master on the first.

Atomic Operations - Atomic operations are those that must occur together without intervening transactions from another master. With a read-modify-write operation, for example, a master reads a register from a peripheral, modifies a bit-field, and writes back to the register. If a second master were to do a read-modify-write operation on a different field of the same register, and if its read were between the first master's read and its write back, the second master's write-back would effectively undo the first master's write, leading to functional incorrectness.

To support atomic transactions, most protocols allow each master to assert a lock signal to indicate that the current transfer should not be separated from the following.

OCP and AXI also support an exclusive read/conditional write mechanism, where a master can perform a *read linked* operation, which sets a reservation monitor for the location. Requests from other threads are not blocked, but may clear the reservation monitor. A normal write from the original thread will clear the reservation. The same master later performs a *conditional write* operation to the same location. If the reservation is no longer in place, then the write operation is not performed, and a failure indication is returned to the master. If the write operation is successful, then the reservation is cleared.

Transaction Ordering and Threading Support - Simple interconnect architectures are such that the response to commands are always returned to masters in the same order that they are issued: the interconnect and the slaves do not rearrange transactions to increase efficiency. For a shared bus architecture, a read operation that cannot be served right away results in wait states that not only reduce the throughput of the requesting master, but tie up the entire bus until the request can be serviced.

For bus-based protocols such as AMBA's AHB, a *split transaction* allows the slave being read to indicate that the data is not yet available, so that the master tries again later, freeing up the bus for other masters to access other slaves while the original slave fulfills the request for the original master.

For protocols that support concurrency like Avalon, OCP, and AXI, a split transaction isn't a useful mechanism, since other masters are free to transact with other slaves while the first master is waiting for read data. These protocols allow masters to have multiple outstanding requests, and the order of the responses to different masters can be

rearranged for throughput or efficiency. DDR memory, for example, is most efficient when reading and writing data to the same memory bank. A DDR controller that is free to rearrange data reads and writes to minimize bank switching can support higher memory throughput. Typically the responses to a given master, or to a given thread on that master, must be returned in the same order they were issued.

OCP and AXI support a *tag* mechanism for out-of-order transaction completion. If a master issues transactions with the same tag, the interconnect must return the responses in the same order as they were issued. Transactions from different tags can be returned in any order, with the restriction that tagged transactions from the same thread cannot be re-ordered if the transaction is for overlapping address spaces. A master that doesn't support threads or tags is treated as a master with one thread and tag.

Cache and Protection Support - Some protocols use extra signals to protect against illegal transactions. ASB, AHB, CoreConnect, OCP, and AXI all support signals to indicate the type of transfer, so that slaves can indicate if the transaction type is supported. Typically, it is up to the slaves or the interconnect to indicate an error if the master attempts an unsupported transaction. The protection related transaction types include the following.

Instruction vs. data – Indicates to the slave or fabric if the transaction is a data or instruction fetch. Instruction transactions to non-memory slaves, or to non-instruction regions of a memory can cause an exception.

User vs. privileged access - Indicates the processing mode. A privileged processing mode typically has a greater level of access within a system.

Secure vs. non-secure – Indicates to the slave the transaction mode when a greater degree of differentiation between processing modes is required.

Functional vs. debug –Indicates to the slave if the access is during normal operation or during debug mode. This is particularly useful to suppress read or write side effects (such as popping data from a FIFO) while debugging.

Some protocols also provide specific signals for system level caching and performance enhancement, in a manner similar to protection. The type of information signaled by caching and performance enhancement signals include the following:

Bufferable –This flag indicates that a write transaction can be buffered for an arbitrary number of cycles before reaching it's final destination. It also implies that a bridge or system level cache can return a response to the master on behalf of the ultimate destination. If a transaction is not bufferable, then the write response can be provided only by the final destination.

Cacheable - Indicates that the transaction at the final destination does not have to match the original transaction, allowing a number of different writes to be merged together, or reads to be pre-fetched.

Allocate – For cacheable transactions, one bit for read and bit one for write indicates if the address in the transaction should be allocated for future read and write transactions.

Write Through Storage - If asserted, write accesses must write-through to memory instead of only writing to the cache.

Memory Coherent Storage - Specifies whether the access must be performed such that it maintains memory coherence with the rest of the system.

Errors - A number of protocols allow the interconnect or slave devices to signal errors, such as data bus or address parity errors, read or write errors, *decode* errors where there is no slave at the indicated address, slave errors where the command has reached the slave, and the slave is returning an error condition, or a *timeout* where no response is received before a timer expires. It is important that the error has enough information to allow the master to determine which of the possibly many outstanding transactions caused the error.

2.1.6 Transaction Interconnect Topologies

The interconnect topology defines the physical structure of the interconnect, and has a significant impact on the performance that a transaction layer can offer. This section describes some of the more common SoC interconnect topologies.

2.1.6.1 Shared Bus

The shared bus, shown in Figure 4, is one of the earliest on-chip communication mechanisms, and is modeled after inter-chip processor busses. It is one of the simplest ways to move on-chip data [19].

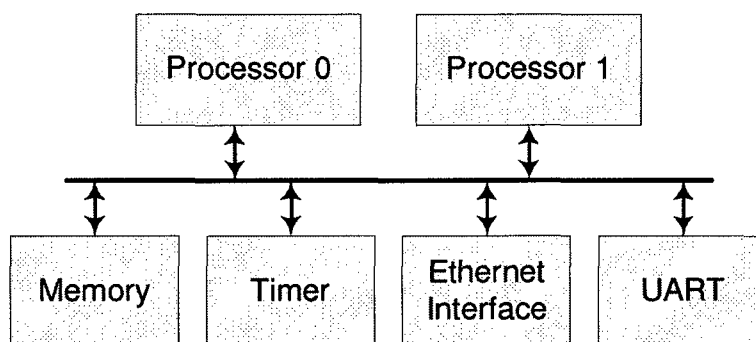


Figure 4 - Shared Bus Topology

Original shared bus implementations such as the AMBA ASB bus are similar to traditional off-chip busses in that control signals are fanned out to all slaves, and tristate data signals are driven by the active master during writes, and by the selected slave during reads. When multiple masters are used, each master asserts a request signal to the central arbiter, which makes a scheduling decision before the transfer can begin [12]. Using tristate signals limits the maximum operating frequency because of the capacitive loading of high fan-out signals. In addition, to allow time for the master or slave to finish driving the bus, idle cycles are inserted every time a new master is granted access. Tristate bus implementations have been largely replaced by multiplexer-based implementations, and tristate signals have not been available in mainstream FPGA device families for years.

Unlike a tristate implementation, typical multiplexer-based implementations like AHB, shown in Figure 5, uses multiplexers controlled by the arbiter and address decoder to route data and control, and tristate drivers are required. The capacitive loading and energy consumption is reduced because only the multiplexers drive high fan-out signals.

In addition, there is no chance of multiple drivers driving a signal, so idle cycles for tri-state driver turn-off time is not required.

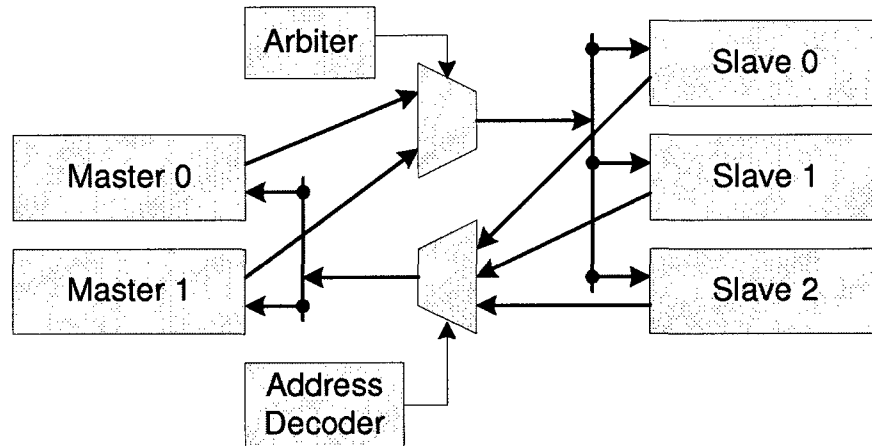


Figure 5 - Shared Bus Topology Multiplexer Implementation

Although shared bus structures support multiple masters, a master executing a transaction is driving data and control to all the devices on the bus, forcing other masters to wait until the transaction is complete. If a processor has separate interfaces for instruction and data, this topology prevents an instruction fetch and a data operation from occurring simultaneously. Because high fan-out and deep multiplexer logic add delays, shared bus structures tend to not be very scalable. The physical wires also tend to be long, adding more delay.

2.1.6.2 Hierarchical Bus

A hierarchical bus topology such as that shown in Figure 6 is a refinement of the shared bus, with bridges to connect busses at different hierarchy levels. Instead of having all of the slave interfaces on the same bus segment, seldom-accessed slaves are placed on low-performance bus segments, and bridges are used to connect the segments.

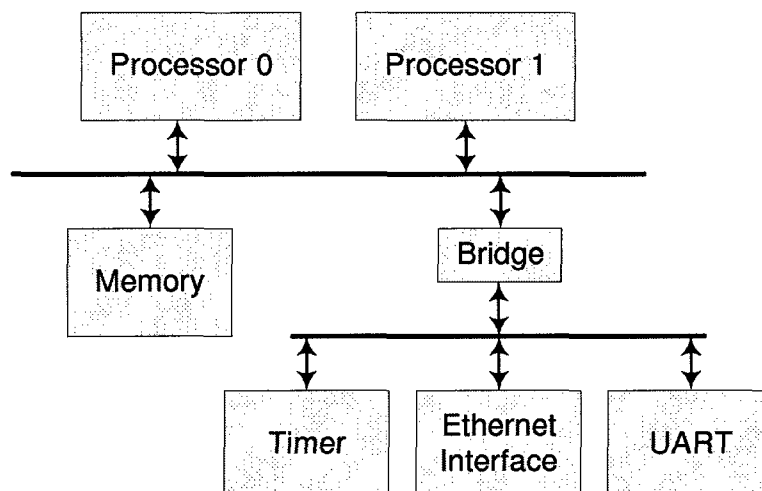


Figure 6 - Two-level Hierarchical Bus

Hierarchical busses reduce the number of components on any one segment, lowering fan-out and logic delays, and allowing the bus frequency to be increased. With low performance components on a separate bus segment, the logic to adapt to a narrower data width or to a slow clock domain can be instantiated once at the bridge, and can be shared by all of the components on the low performance segment. Since the components on the low performance segment are accessed less often, the additional latency is often worth the performance gain.

2.1.6.3 Segmented Bus

The segmented bus topology is similar to the hierarchical bus topology, except that different segments can have masters, slaves, or a combination. Bidirectional bridges are used to allow masters on one segment to access slaves on other segments. When a master accesses a slave on a segment other than its own, both segments are tied up for the duration of the transfer. If there is a master on the segment being accessed, that master cannot proceed until the other transaction is complete.

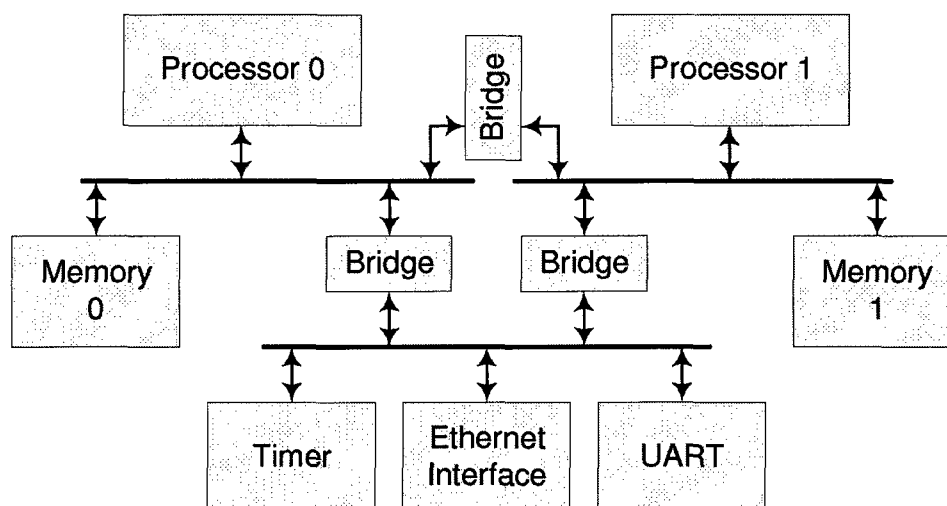


Figure 7 - Segmented hierarchical bus

The topology shown in Figure 7 is referred to as a *segmented hierarchical* topology. Here, each master has its own high-performance segment, and uses a bridge to access a shared low-performance segment or another processor's segment. Careful segmentation results in a system where a master on a given segment spends most of its time accessing slaves on the same segment. As long as two masters on different segments are accessing slaves on their own segment, operations can occur concurrently, and throughput is increased.

The cost of a segmented bus is higher than a non-segmented bus, because address decoding and arbitration logic is replicated. Performance for a segmented bus with N masters is up to N times that of a shared bus, as long as the masters seldom access the same resource.

2.1.6.4 Cross-Connect

A full cross-connect topology such as that shown in Figure 8 can be thought of as a segmented bus where every master and slave has its own bus segment, and there is a bridge from every bus segment to every other bus segment. As long as no two masters are trying to access the same slave at a given instant, full concurrency can be supported. In addition, all connections are point-to-point, reducing fan-in and fan-out to one, and minimizing capacitive loading.

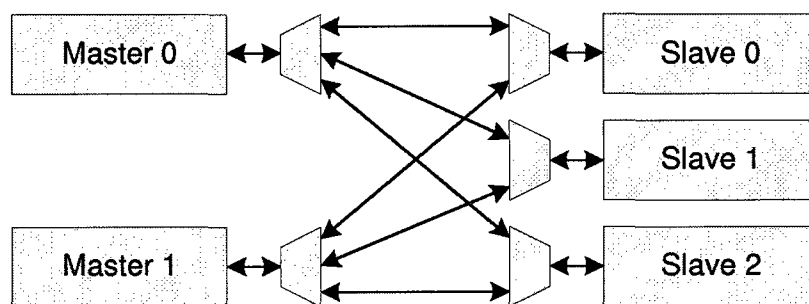


Figure 8 - Cross-connect Topology

Full cross-connects are low latency and non-blocking, but have the disadvantage of being expensive because the arbitration and multiplexing logic is replicated for every slave interface [19]. *Slave-side arbitration* is a cross-connect topology implementation where the arbitration logic is implemented independently for each slave interface, allowing a different arbitration algorithm at each slave.

2.1.6.5 Separation of Command, Response, and Data

Modern on-chip transaction specifications such as OCP and AXI separate the command, response, and data portions of the interface to allow for greater throughput and flexibility in implementation.

With OCP, master interfaces are connected to matching slave interfaces on the interconnect using three channels. For every write command a master pushes to the command channel, it pushes data to the write data channel. For read commands, a response is eventually pushed back to the master via the response channel.

AXI, goes one step further, and has five channels instead of three, each with an identical but independent handshake. This allows for higher performance because any given master or slave can be writing and reading at the same instant. The five channels are listed below.

Write Address Channel - Carries all of the address and control information required for a write transaction.

Write Data Channel - Carries all of the write data for a write transaction.

Write Response Channel - Provides a way for the slave to respond to write transactions. All write transactions require a response.

Read Address Channel - Carries all of the address and control information required for a write transaction.

Read Response Channel - Conveys both the read data and read response information from the slave back to the master.

2.1.7 Arbitration

Arbitration algorithms are used to determine which masters get access to a given shared resource. Arbitration algorithms can be complex, and given a set of masters competing for a resource, it is not uncommon for the arbitration logic to take a number of clock cycles to determine which master wins the resource. *Arbitration cost* can be defined as the number of clock cycles it takes to make an arbitration decision. An arbitration cost of zero refers to an arbitration scheme where the result of the arbitration is delivered in the same cycle as the requests. This section briefly describes the most common arbitration algorithms.

Fixed Priority - With a *fixed priority* arbitration scheme, the relative priority of each of the masters is constant, such that a higher priority master always wins arbitration when competing with lower priority masters. The advantage of fixed priority arbitration is the simplicity of its implementation and the low cost in terms of device resources.

Starvation refers to the condition where the arbiter prevents a given master from getting even minimal access to a shared resource. With fixed priority arbitration, as long as a high priority master is trying to access a slave, there is no possibility for the low priority master to transfer any data at all. In a typical embedded system, it is desirable for even low priority masters to be allocated a non-zero share of the total available bandwidth.

Round Robin - Round robin is a best effort arbitration scheme where each master gets a turn to use the resource. If a master misses its turn, it must wait until the next turn. Most round robin arbitration schemes implement *work conserving round robin*, where if a master doesn't take its turn, the next master can use the turn instead of it going to waste. As with other best effort schemes, the bandwidth available to any given master cannot be guaranteed. Both round robin and work conserving round robin arbitration schemes are relatively easy to implement and have a low cost.

When bursts are supported, round-robin arbitration is unfair, because there is no way to ensure that masters get the same throughput; they only get the same number of turns. If one master issues large write bursts every turn while another issues only small bursts, the first master gets a larger share of the available bandwidth.

Weighted Round Robin - *Weighted round robin* is an extension of simple round robin that assigns a weight to each master contending for the resource, and the number of turns that each master gets is proportional to its weight.

Weighted Fair Queue - *Weighted fair queuing* is a packet scheduling implementation intended to approximate ideal *generalized processor sharing*. With generalized processor sharing, each master has a separate transaction queue, and at any given time all the active masters have a portion of the available throughput according to their weight [20]. Since each master has its own queue, a master with a significant amount of data to send can coexist with other masters while still allowing them to be guaranteed a certain throughput [21]. Weighted fair queuing can approximate fairness between masters with different burst characteristics.

Fixed Time Division Multiplexing - With *fixed time division multiplexing* arbitration, each arbitration point maintains an arbitration schedule where each unique master has a reserved timeslot. The schedule can be such that some masters have more timeslots than others, providing them with higher throughput. Bursting masters can send only a portion of a burst in each timeslot, providing true fairness at the expense of complexity at the slaves due to the management of partial bursts in each slot. Fixed time domain multiplexing arbitration schemes commonly implement a mechanism that allows other masters to use cycles not used by the scheduled master.

Lottery - In lottery arbitration systems, processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets can be non-uniform to grant a given master more tickets to provide it with a higher chance of selection.

Multi-level Arbitration - If the requirements of an embedded system cannot be met by a single arbitration scheme, the system can implement *multi-level arbitration*, where the winner of the first arbitration scheme is an arbiter itself, which then goes on to select its own winner.

Consider, for example, the case of a resource with four high priority masters and four low priority masters. The two first stage round-robin arbiters might select from among the high priority masters and among the low priority masters respectively, and the second stage fixed priority arbiter selects the first stage high-priority arbiter over the low-priority arbiter whenever the high-priority arbiter is requesting.

Specifications such as CoreConnect's PLB allows each master to signal the priority of each transfer as one of four priority levels. The master can change the priority of the request at any time during the address cycle.

2.1.8 Burst Transactions

A *burst* is the term given to a transaction that includes more than a single unit of data. Burst transactions are used to amortize across multiple transfers the cost of an expensive operation such as an arbitration or an off-chip access. A burst write sent by a master typically includes the command, the address to write to, and multiple cycles worth of data as a single transaction unit. The response to a burst read command is typically multiple cycles worth of data returned to the master as a single unit.

Consider a system where the arbiter can deliver one decision every two clock cycles, where the throughput of single cycle transfers is limited to one transfer every two cycles. If each master sends a burst with two or more cycles of data for every arbitration decision, then a transfer every cycle can be achieved.

Bursts can also be either *precise* or *imprecise*. With precise bursts, the burst length provided at the beginning of the burst defines precisely the length of the burst: the master will send no more and no less than the number of transfers indicated. With imprecise bursts, the burst size is just a hint from the master regarding how long the transaction will continue, and only the last transfer, with an end of burst flag, can be interpreted with complete certainty [22].

From the interconnect network's point of view, a burst is simply a mechanism to reserve the connection to a given resource. Upon beginning a write burst, the interconnect guarantees to the master that it will not be interrupted until the burst is complete. From a slave's point of view, bursts provide advance knowledge of how much data to expect or provide.

One of the primary considerations for a burst implementation is the ordering of addresses within the burst. The most common burst address sequences are described below.

Fixed – Fixed address bursting simply refers to the case where all of the data in the burst is destined for the same address, such as when pushing data into a FIFO.

Incrementing – One of the most common burst addressing sequences is the *incrementing burst*, where each subsequent data transfer is to the address following that of the previous cycle.

Wrapping - Memory devices commonly implement a *wrapping burst*, where the address wraps back to the previous burst boundary when it reaches the end of a burst boundary. This is equivalent to incrementing only the low order bits [23].

Wrapping bursts are efficient for processors with data or instruction caches. When a processor requests data that is not in the cache, the cache controller reads enough data from the memory to fill the entire cache line. For a processor with a cache line size of 64 bytes, a cache miss causes 64 bytes to be read from memory. If the processor happens to have wanted data from address 12 when the cache miss occurred, an incrementing addressing burst issues read addresses 0, 4, 8, 12, 16, 20, 24, 28, such that the data that

the processor is waiting for isn't available until the fourth datum is returned. With wrapping bursts, the issued address order is 12, 16, 20, 24, 28, 0, 4, and 8, such that the data that the processor is waiting for is the first returned.

Interleaved – With *interleaved addressing*, also referred to as *exclusive-OR addressing*, the address for subsequent accesses within the burst is calculated by XORing the address with a binary counter. As with wrapped bursts, the first memory location is accessed first, but it is then followed by the memory location whose address is common in all but the low order bit. This is followed by accesses whose addresses share all but the second-most low order bit. An interleaved access pattern returns the requested word first as with a wrapping burst, but arranges the data to minimize the number of accesses to an external memory, such as a DDR memory, increasing memory efficiency.

2.2 Networks on chip

Networks on chip are poised to revolutionize on-chip interconnects, especially for large System-on-Chips with many components. The basic idea behind Network On Chips is borrowed from traditional networks, and envisions on-chip router-based networks supporting packetized communication. Cores access the network by means of proper interfaces, and have their packets forwarded to the destination through a multi-hop routing path [24].

Network-on-chip technologies promise to address needs facing system-on-chip integrators by treating the system as a network of components [5]. The use of a well defined protocol stack allows the separation of computation from communication [25]. The different layers of a protocol stack are responsible for providing services to upper

layers in the same fashion as traditional local area or wide area networks. This approach has the advantages of structure and modularity, reducing resources and routing congestion by sharing wires. An on-chip network defines a standard network interface that components can plug into in much the same manner as cards plugging into a backplane bus. Networks also typically provide higher performance than buses because they are simpler to implement, and can support concurrent communications [26].

2.2.1 On-chip and off-chip networks differences and similarities

The result of years of research into off-chip networks serves as a good starting point for investigation into on-chip networks. The following is a list of differences and similarities between on-chip and off-chip networks.

Design-time specialization – The single most important characteristic of on-chip networks is the applicability of design-time specialization. Off-chip networks must be future-proofed, as the network interfaces and protocols must be designed such that they can support the needs of applications not yet dreamt of. With on-chip networks, the designer only needs to support the application that the chip is providing. The network layers and the services they provide can be tailored to and optimized for exactly the needs of the system being developed [25], and if a newer chip in the future needs different network services, it can implement a different network.

Resource Constraints - On-chip networks differ from traditional inter-chip networks in that the resource constraints are much different. In off-chip networks, memory and computation power is relatively inexpensive, and processors are often used to implement

protocol stacks, while links are exceedingly constrained, and off-chip networks carry as much data as possible over a minimum of wires. In on-chip networks, the computing power available at each node is limited, but the number of links between nodes might number in the hundreds [25].

Flow control and Buffering - The networking nodes in inter-chip networks typically finish receiving an entire packet before beginning to send it to the next node [27]. With on-chip networks, the scarcity of resources dictates that per-node storage is kept to a minimum. Fortunately, on-chip wires are much shorter and much more reliable than off-chip networks, allowing a reduction of buffer space and error detection and correction logic [25].

Reliable Communication - In traditional networks, the network links themselves are treated as unreliable, and the data link layer drops data at any time. A significant amount of processing power goes into ensuring reliable communications between nodes in the face of this unreliability [28]. On-chip wires provide a reliable communication medium, so on-chip networks can avoid the processing and overhead associated with providing reliability.

Data ordering – Typical off-chip networks do not maintain packet ordering, meaning that the packets from one node to another may arrive in a different order than they were sent. For NoC network implementations, deterministic routing can be used to ensure that all packets from a source to a sink follow the same path, guaranteeing that they arrive in the same order.

Synchronization – With an on-chip network, all of the network nodes can share synchronization due to the small area of the network. Global synchronization allows for routing algorithms that can provide Quality of Service (QoS) guarantees [29].

Deadlock – Deadlock is the condition when two nodes cannot progress because each of them is waiting for some action from the other. The deadlocks that can occur in NoCs consist of *routing-dependent deadlocks*, where there is a cyclic dependency of resources created by the packets on the various paths in the network, and *message-dependent deadlocks*, which occur when interactions and dependencies are created between different message types at network endpoints when they share resources. For regular topologies, the use of restricted routing functions is an effective way to avoid routing-dependent deadlocks [25].

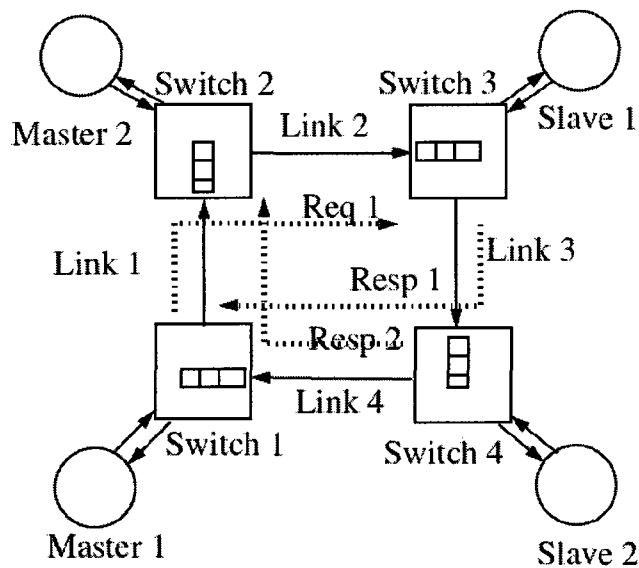


Figure 9 - Message-dependant deadlock example [25]

An example of message dependent deadlock is shown in Figure 9, where Master 1 is waiting for Slave 1 to accept request Req 1, and Slave 1 is trying to send a response to a

previous request, Resp 1. Resp 1 cannot be sent, however, since Resp 2 is using Link 4, and Resp 2 cannot finish because Req 1 is blocking Link 1. In a network consisting of masters requests and slave responses, message dependant deadlock can be avoided by maintaining separate virtual networks, with separate buffers, for commands and responses [25].

2.2.2 Network Switching Algorithms

One of the key functions of an on-chip network is to get packet from a source to a destination, often across a number of links. Switching algorithms that provide this function can be grouped into four classes:

Circuit switching - Common in telecom applications, *circuit switching* is when a path or circuit is reserved prior to the transmission of data, and is maintained until the transmission is complete, at which time it is torn down. With circuit switching, the network resources are kept busy for the duration of the communication, and the time to set up a path can cause sizeable latency.

Packet switching – With *packet switching*, the data stream is split into a number of packets, and the packets are sent individually across the network. At each network switch or router, the packet is completely buffered, and is held there while the node determines the next hop. This approach is also called *store-and-forward*, and can introduce significant non-deterministic latency due to the buffering at each node [5].

Cut-through switching – *Cut-through* switching is similar to packet switching, except that each node forwards the packet as soon as the amount of data received for a packet

reaches the *cut-through threshold* [30]. This reduces the latency seen by the packet, and the buffering required at each node, but increases the probability of blocking, since a packet may occupy several nodes.

Wormhole Switching – Wormhole switching further segments packets into *flow control units*, or *flits*. The first flit in a packet reserves each node along the path to the destination, and the node is reserved until the last flit in the packet has been sent on its way. Wormhole switching requires minimal buffering at each node, and reduces the transport latency, but like cut-through switching, one packet may occupy several nodes along its path.

2.2.3 Topologies

The topic of network on chip topologies has been researched significantly, and different topologies have been found to have different advantages depending on the application. In this section, we present some of the most common NoC topologies.

2.2.3.1 Bidirectional Mesh

The bidirectional mesh, shown in Figure 10, is a regular two-dimensional grid topology with bidirectional connections between adjacent nodes. The mesh topology has been favored in NoCs because of its regularity and linear area growth with the number of nodes: there is always one switch per node [31] [32]. Meshes have a relatively large average network distance, which increases transaction latency and power consumption. Moreover, meshes tend to have concentrated areas of high traffic within the grid, which reduces concurrency.

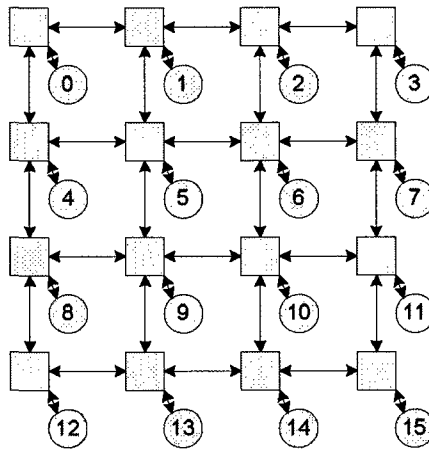


Figure 10 - 2D Mesh Topology

2.2.3.2 2D Torus

A 2D torus such as that shown in Figure 11 extends a mesh by including wrap-around links between the rightmost and leftmost nodes on a given row, and the top and bottom nodes on a given column. This reduces average network latency and traffic concentration, at the expense of additional resources for the additional links [33].

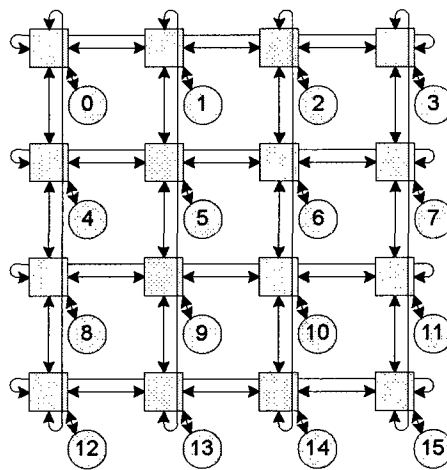


Figure 11 - 2D Torus Topology

2.2.3.3 Butterfly

A k -ary n -flies *butterfly network*, also known as a *fat tree*, is a network with n stages of k -radix switches. A four stage butterfly network with 2-radix switches is shown in Figure 12. The switches and connections between them are arranged such that there is only one path from any input to any output. Conflicts in a butterfly network cause network saturation, while some of the connections cannot be serviced [32]. Increasing the radix of the switches increases cost, but reduces conflicts and lowers the network latency.

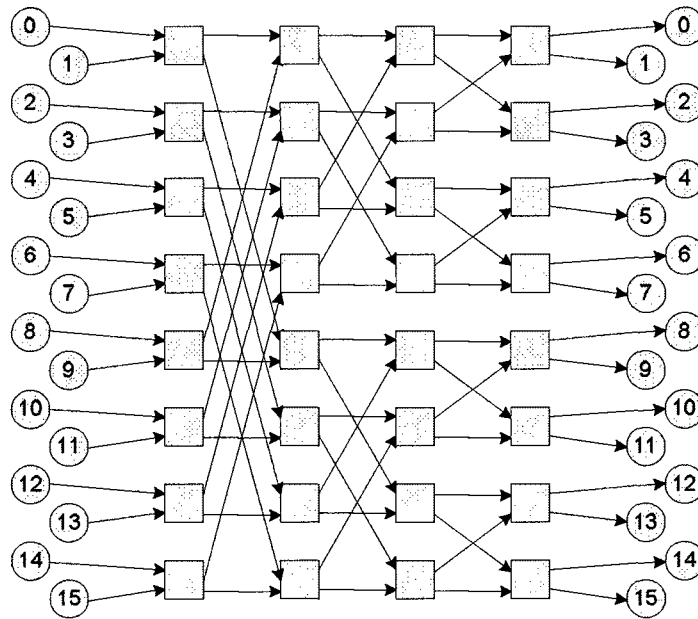


Figure 12 - 2-ary 4-flies butterfly network topology.

2.2.3.4 Flattened Butterfly

A flattened butterfly network can be constructed by taking a butterfly network and collapsing the nodes on a given row into a single switch, with higher radix, as shown in Figure 13. A flattened butterfly network requires fewer switches, but with a higher radix

switches, and supports multiple paths between any master and slave, which can reduce conflicts. Some connections also see a lower network latency.

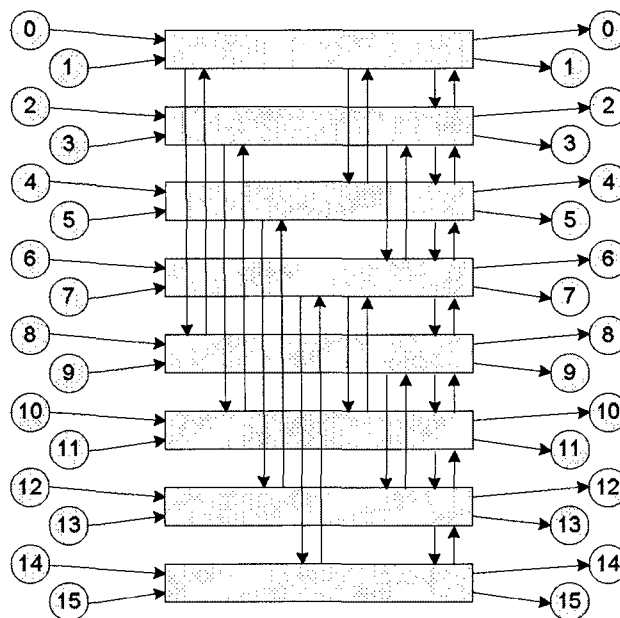


Figure 13 - 2D Flattened butterfly

2.2.3.5 Clos

A Clos network is a multi-stage network with an odd number of stages, equivalent to two back-to-back butterfly networks. The input network can route data from an input to any middle stage, and the output network can route data from any middle stage to any output stage [34]. A Clos network has high latency than the other networks considered so far, but provides for multiple paths between any master and slave, which can reduce conflicts.

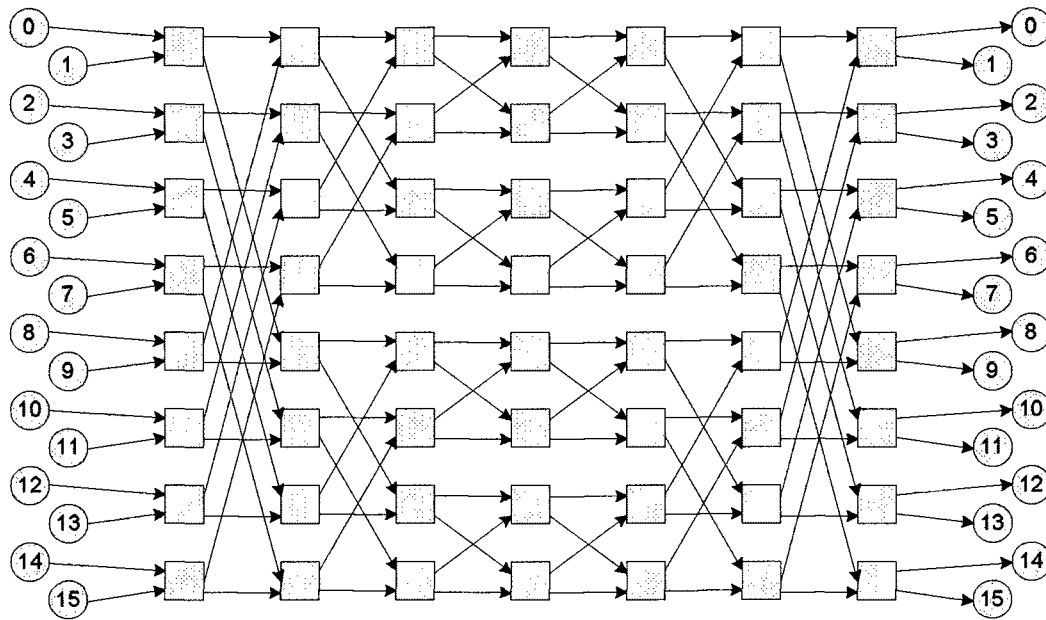


Figure 14 - 5 Stage Clos Network

2.2.3.6 Separating Command & Response Networks

Most of the NoCs proposed to date treat each IP core as a single node in the network that has a single bidirectional channel for the sending and receiving of data, and allows every node in the network to send and receive packets to and from every other node on the network. If the NoC is being used to support transactions, the network requirements can be simplified considerably by considering that master commands are only sent to slaves, and slave response packets are only sent to masters. If we remove the connections that are not used, this leaves us two completely independent networks, one for commands, and one for responses.

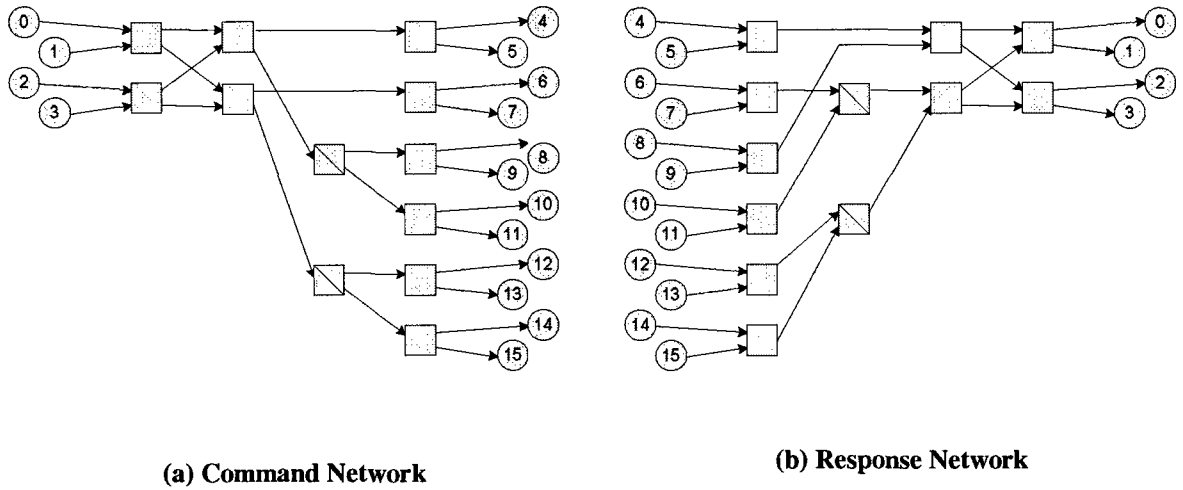


Figure 15 - 4 Master, 12 Slave Command & Response Butterfly Networks

For example, if the 16 node network butterfly network shown in Figure 12 happened to consist of four masters and 12 slaves, it could be implemented as the two networks shown in Figure 15. Instead of the original 32 2x2 switches, each network now requires only 10 switches, and two merge or join primitives, saving approximately a third of the network resources. In addition, congestion is reduced compared to the original, since only a portion of the traffic is on each network, and some nodes now see only 2 hops of latency, instead of three. Maintaining separate networks for commands and responses also prevents NoC deadlock [35].

2.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a mainstream technology for systems requiring custom logic. While FPGAs were originally used as glue logic between other devices, they are now used for system-on-chip designs in their own right.

2.3.1 FPGA Structure

FPGAs typically consist of a largely homogenous array of *logic elements* (LEs) with programmable routing resources used to connect them [36]. The Stratix II device whose block diagram is shown in Figure 16 has its LEs grouped into *Logic Array Blocks* (LABs), as shown. At power up, an FPGA pulls a configuration bit stream from a memory device such as a flash memory, or is actively programmed with a bit stream by a processor. The bit stream populates an SRAM memory array that controls the function of each of the LEs and the routing resources themselves.

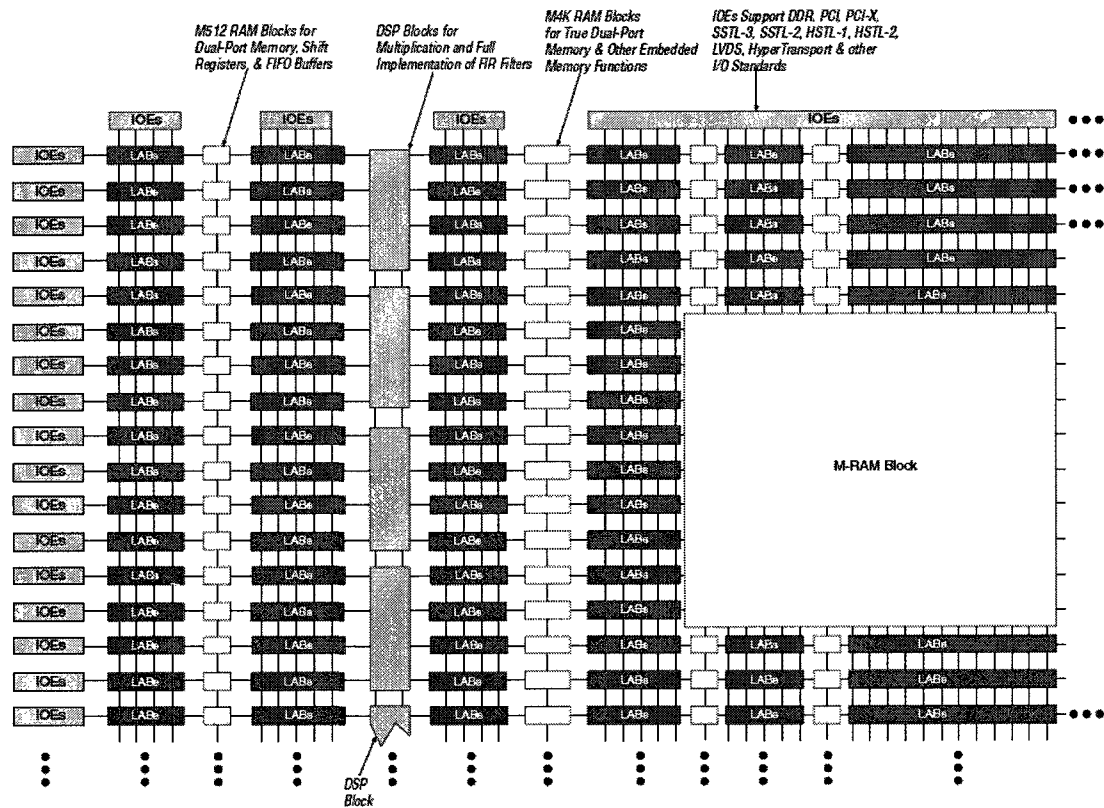


Figure 16 - Stratix II FPGA Block Diagram [36]

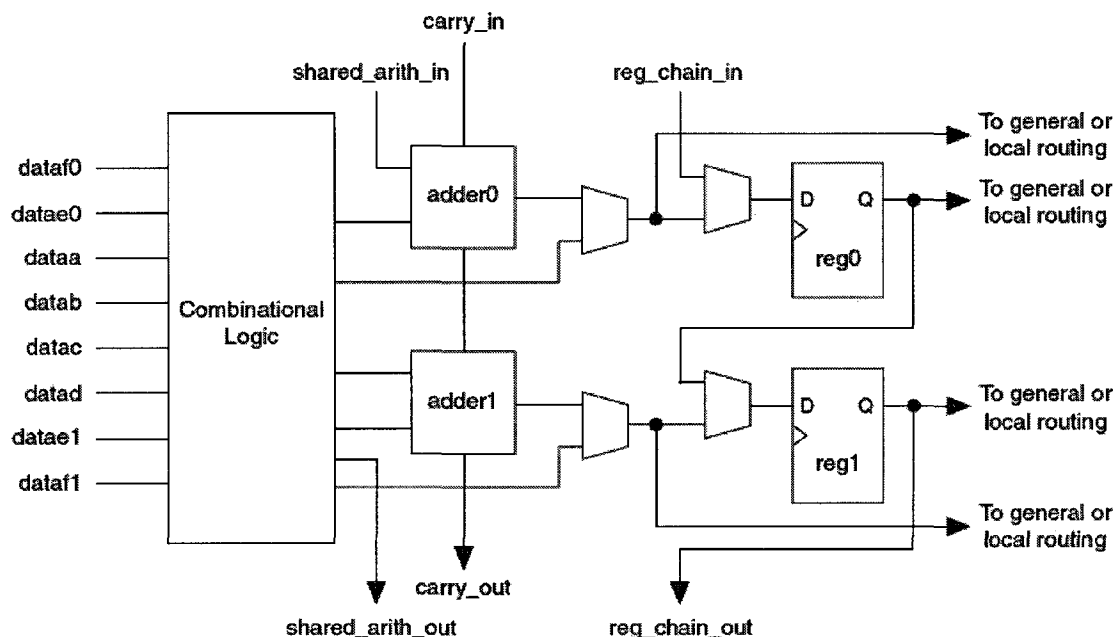


Figure 17 - Stratix II Logic Element [36]

The simplified structure of a single logic element is shown in Figure 17. The combinational logic shown typically consists of a number of *look up tables* (LUTs) that treat the data input as bits of a memory address connected to a memory whose outputs become the outputs of the combinational logic. One of the roles of the configuration bit stream is to populate this LUT memory. In addition to the LUTs, each LE includes high performance carry chain logic for arithmetic operations, and a number of registers to optionally register the data at the output of the LE. The multiplexers shown in Figure 17 are also configured by the configuration bit stream, and are used to control the routing of data as the chip is operating.

The cost of an FPGA is usually higher per unit cost than custom ASICs, and FPGAs typically suffer from higher power consumption and lower throughput. Nonetheless, FPGAs are a popular choice because there is no Non Recurring Engineering (NRE) fee

for masks and tooling, and because a single FPGA can be reprogrammed repeatedly during system development, eliminating the risk of expensive ASIC re-spins when errors are found.

2.3.2 FPGA Design Flow

The FPGA design flow shown in Figure 18 is very similar to an ASIC design flow. Most of the design is done using a hardware description language such as VHDL or Verilog, and the HDL design is simulated with a testbench to ensure functional correctness. A synthesis tool is then used to synthesize the logic elements and connections required to implement the design, and a place and route tool is used to fit the design in a specific FPGA. Static timing analysis is used to ensure timing closure, and a timing simulation can optionally be used to verify functional correctness with timing information [37].

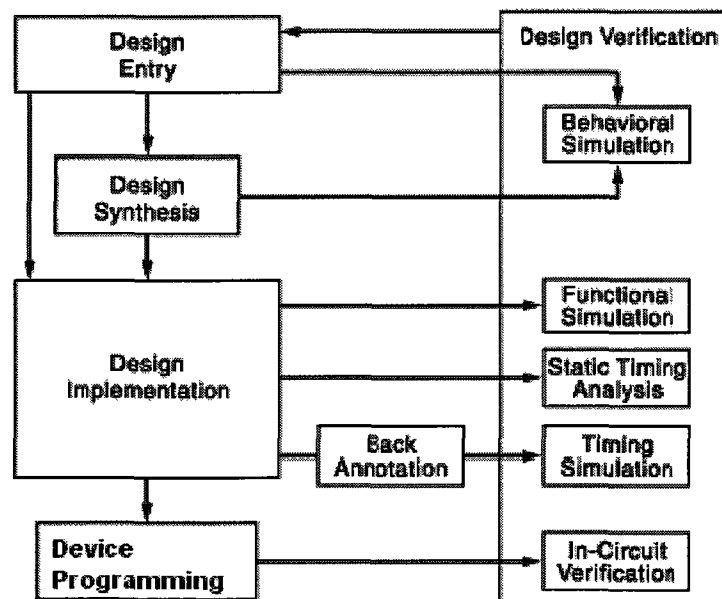


Figure 18 - FPGA Design Flow [37]

The biggest advantage of an FPGA over an ASIC is that the design can be downloaded to the target device and tested in real hardware immediately after the implementation tools have finished compiling the design [37]. RTL simulations can be performed just as with ASICs, but being able to configure an actual device and provide real-time stimulus in the lab gives a great advantage during verification. When a design is targeted to an FPGA device, the timing analysis and fitting results from the tool chain are complete. With ASICs, designers must become used to piecemeal results as the separate steps of I/O insertion, boundary scan insertion, test circuitry insertion and block integration are performed. In FPGAs, due to a highly integrated environment, everything needed for the design is present.

2.3.3 FPGA Considerations for NoC design

When designing a System on Chip to be implemented on an FPGA, the system designer needs to tailor the interconnect to account for FPGA characteristics, such as the differing cost of resources and the differences in design flow.

2.3.3.1 Logic Resources

When considering an interconnect for an FPGA, one has to consider the resources available in the FPGA, and the cost associated with them. Unlike ASIC devices, where the Non-Recurring Engineering (NRE) charge associated with a design is a primary consideration, FPGAs have a minimal NRE charge, since they are programmed after being purchased, and can be re-programmed indefinitely. The cost per individual device, on the other hand, is considerably more expensive.

With FPGA devices, the logic available in a device is available whether or not it is used, and seldom is all of the logic an FPGA used. For example, if a given FPGA design uses 75% of the available Logic Elements, it might use only 60% of the registers, a small fraction of the routing resources, and possibly none of the embedded memory; the rest of the device goes to waste. According to [39], FPGAs use an average of 35x more logic than an ASIC for an equivalent design.

Because of the relative cost of FPGA logic, one of the primary considerations for an FPGA targeted NoC is logic utilization. This means that an FPGA NoC can't afford heavyweight routers or switches. An FPGA NoC must provide its services while consuming a minimum of logic.

2.3.3.2 Routing Resources

One of the proposed benefits of NoCs for on-chip communication is that wires can be used more efficiently by sharing them between many communication flows, such that while one client is idle, other clients can use the network resources [26]. On the other hand, modern FPGAs are relatively rich in routing resources; the device usually has enough routing resources to connect whatever logic the designer decides to use. In the case of FPGA NoCs, the logic required to share the wires may be more than the cost of the wires themselves.

2.3.3.3 Floorplanning

Another difference between the FPGA design flow and the ASIC design flow is that manual floorplanning is not part of the standard FPGA design flow. Anecdotal evidence

suggests that designers that manually floorplan their FPGA often end up with worse performance than if they let the automated tools run without a floorplan. Early in the NoC design flow presented in [26], the die is divided into IP core tiles and network logic. This way of thinking is especially appropriate to a topology like a 2D mesh or torus, where the network nodes are physically arranged in a 2D space. One of the problems found by the creators of the SoCBus mesh-based NoC was that the NoC nodes were placed arbitrarily rather than according to the mesh layout they had in mind while designing it [38]. When designing an FPGA-based NoC, it is important to consider the logical topology and its effects on resource utilization, transport latency, and link congestion, but the physical layout of the topology on the device is a secondary consideration.

2.4 Related Work

In this section, we look at current solutions to on-chip FPGA interconnects, including the SoC integration tools provided by Altera and Xilinx, and FPGA based NoC implementations found in the literature. Figure 19 provides a summary of the frequency, resource utilization in terms of Look up Tables (LUTs), and the latency of a read operation for a number of interconnect implementations. Note that since different implementations have used different FPGA families with different performance characteristics, frequency is not a reliable indicator of interconnect performance. Also, the effective amount of logic in a LUT in different devices families varies by as much as 25%, so those measurements should be taken as approximate as well. The information at [40] was used to convert Xilinx Logic Cells to Altera LUTs for comparison purposes.

<i>Size (Mstrs x Slaves)</i>	<i>Source</i>	<i>Device Family</i>	<i>Datapath (bits)</i>	<i>Freq. (MHz)</i>	<i>Read Latency (Cycles)</i>	<i>LUTs</i>
2, 2	SOPC Builder (Cross-connect)	Stratix III	32	322	1	182
	Aethereal [41] (2D Mesh)	Virtex II	32	18	≥ 3	3200
3, 3	SOPC Builder (Cross-connect)	Stratix III	32	230	1	359
4, 4	SOPC Builder (Cross-connect)	Stratix III	32	199	1	741
	Xilinx PLB (Bus)	VirtexIIP	64	145	3	604
	Nostrum [42] (2D Mesh)	--	64	--	110	4945
	[43] (2D Mesh)	Virtex II	8	--	≥ 3	5652
	[44] (Cross-connect, no NI)	Virtex 4	36	272	6	780
	[32] (Unidirectional Mesh)	Virtex II	32	200	16	648
5, 5	[32] (Bidirectional Mesh)	Virtex II	32	200	8	1206
7, 7	SoCBus [38] (2D Mesh)	Virtex II	32	80		4700
8, 8	SOPC Builder (Cross-connect)	Stratix III	32	138	1	3230
	Xilinx PLB (Bus)	Virtex-IIP	64	144	3	1178
16, 16	SOPC Builder (Cross-connect)	Stratix III	32	91	1	11838
	Xilinx PLB (Cross-connect)	Virtex-IIP	64	135	3	2221

Figure 19 – Performance summary for existing solutions

Altera's SOPC Builder is a tool that generates a transaction interconnect to support a user-specified-system. The interconnect uses a cross-connect with per-slave arbitration, which allows full concurrency. The SOPC Builder results presented in Figure 19 were arrived at by creating & compiling a number of designs with simple master and slave components, and then subtracting from the total the resources attributed to those masters

and slaves. The biggest problem with the SOPC Builder solution is its scalability, as can be seen in the figure. As the number of masters and slaves increases from 4 to 16, the frequency drops from 199 MHz to 91 MHz, and the resource usage increases 15x from 740 LUTs to 11'838 LUTs.

The Xilinx PLB performance results were taken from [43]. The Xilinx PLB Arbiter solution provides a 64-bit shared bus with a centralized arbiter, and consists of a bus control unit, a watchdog timer, and separate address, write, and read data path units with three-cycle arbitration. With this centralized arbiter, the bandwidth available per master goes down as the number of masters increase, making large designs with many components impractical.

A number of research papers have implemented 2D mesh networks on FPGAs. In [42], a 2x2 Network on chip using the Nostrum protocol as described in [46] is implemented in a Stratix II FPGA. The network interface components for a single interface in this network includes timeout and retransmission logic, and includes 7 finite state machines and 8 FIFOs. The resource utilization is 4945 LUTs, larger than a complete 8x8 SOPC Builder interconnect, while the latency it provides for read transactions is on the order of 110 clock cycles. The NoC FPGA interconnects in [43] and [44] also each present a NoC using 2D mesh topologies, where each node in the mesh has expensive per-network hop routing. Both of these have logic utilization higher than the equivalent full cross-connect from SOPC Builder, with higher latency due to the multi-hop network.

In [41], an Aethereal NoC with a 32 bit data path is prototyped on a Virtex II FPGA. This design supports two nodes, each with a master and slave interface. The authors

report that the network uses about 5% of the logic available on the device, which translates into about 3200 LUTs, while operating at only 18MHz. This is an example of using an NoC targeted at ASIC implementation on an FPGA; the frequency is 17 times slower than that provided by the SOPC Builder tool, and requires 17 times the resources.

In [32], the author implements a low-overhead, high-performance packet-switched network based on out split & merge primitives that supports 8 nodes, runs at 166 MHz, and uses approximately 20'000 LUTs. Included is an extensive analysis that considers different network topologies given an FPGA cost model. A subset of the presented findings is included in Figure 19, where it can be seen that the logic utilization for a 4 input 4 output switch box alone based on this work is slightly less than the complete 4x4 interconnect from SOPC Builder, and suffers from considerably more latency.

The XPipes NoC, presented in [24], is a library of composable & tunable soft macros for the support of masters and slaves communicating via the OCP protocol. The XPipesCompiler tool can be used to assemble the XPipes primitives to provide an application specific NoC, and supporting arbitrary topologies. XPipes was designed with reliability in mind, and includes per-node CRC-based error detection with link-level acknowledgement & retry, and the switch implementation has both input buffers and output buffers, making it high latency and resource intensive, too heavy-weight for FPGAs. Although the XPipes macros themselves are not appropriate for FPGA designs, the approach providing a library of composable IP cores is very applicable to FPGAs, as it allows the FPGA System designer to use NoC components to tailor the network & topology to the application.

From our review of current on-chip FPGA interconnect solutions from industry, and of solutions presented in the literature, it is clear that there is not yet a compelling solution to the on-chip FPGA interconnect problem. A light weight, flexible, and high performance interconnect that is appropriate for FPGA costs and resource limitations is required.

3. Merlin Architecture

This section presents Merlin, a flexible, high performance, and light weight approach to on-chip FPGA interconnect. Merlin includes a collection of packet based primitives, a way to map transactions to packets, and algorithms for connecting components such that they provides the services required of a transaction interconnect. This chapter is organized into 5 sections. The first two sections describe the objectives of the Merlin interconnect, and the feature set that it needs to implement. The third section provides an overview of the architecture, including the protocol stack model. The fourth and fifth sections describe in detail the interfaces, primitives, and algorithms used to implement the interconnect.

3.1 Objectives

There are a number of goals and objectives to be met by the Merlin Architecture described here. The goals and objectives include:

Resource Efficiency – Because of the comparatively high cost of resources in an FPGA, it is important that the interconnect is able to use resources efficiently. In particular, the interconnect should not use resources to support a given feature if

the feature isn't required by the application.

Performance – Merlin must support high performance systems, such that the interconnect does not become the performance bottleneck for reasonable topologies. What it means to support high performance depends on what the system itself is being optimized for. If the system is being designed for minimum software execution time on a given processor, Merlin must be able to provide a high frequency or low latency interconnect.

Flexibility and Configurability – Since Merlin is a general purpose interconnect not targeted to any specific application, flexibility is of the utmost importance. The interconnect must be able to support a varying number of master and slave connections, various address and data widths, and different combinations of supported transactions. The interconnect must also allow system designers to make engineering tradeoffs, such as selecting an interconnect solution that can operate at high frequency at the expense of resource utilization and latency.

Scalability - It is expected that as FPGAs continue to grow in size and complexity, the demands placed upon the system interconnect will also grow. Merlin must be able to scale well with system sizes.

Heterogeneity - Merlin must support a mix of different masters and slaves in the same system, and must allow for adaptation between them. This includes masters and slaves with different data widths, with different burst characteristics, and with different timing characteristics.

3.2 Feature Set

The interconnect described herein is intended to support the transaction services required by the Avalon MM , AMBA AHB, AMBA AXI, OCP, and Wishbone protocols, so that it can be used to connect cores using any of these interfaces. As such, it must support the functionality described below.

Data width adaptation – The data width of a master or slave must be a power-of-two number of bytes. Where a master and slave have different widths, the interconnect must be able to perform the conversion.

Fixed and variable wait states – Slaves are able to declare the timing of their interface to use either fixed wait states, where the number of wait states for read and write transactions are defined at system generation time, or variable wait states, where a slave output signal is used to indicate to the fabric that the transaction must be held off.

Pipelined reads – Pipelined slaves return the data independently of the issuance of the read command itself. The read response may be several cycles later.

Burst transactions – Merlin must support burst read and write transactions, where the burst addressing can be sequential, interleaved, or to a fixed address. If the master and slave support different burst mechanisms, a conversion mechanism must be available.

Zero-cycle arbitration – The Merlin interconnect must be able to make arbitration decisions using purely combinatorial logic, such that an arbitration decision is

rendered in the same cycle that requests are made. Making an arbitration decision in zero cycles means that the arbiter logic is combinatorial, and is often on the critical path with the address decoding and the data path multiplexing logic. Merlin must allow be able to support multi-cycle arbitration, which moves the arbitration logic off the critical path, and allows for higher frequency operation.

Transaction Ordering –Merlin must be able to accommodate masters that require responses to commands to be returned in the same order they were issued. If the network can be relied upon to deliver packets in the same order as they are issued, the transaction layer needs concern itself only with ensuring that responses to commands that have been issued to different slaves are in order.

Clock Domain Crossing – Merlin must be able to support masters and slaves on different clock domains, by providing light weight clock domain crossing logic. A dual clock FIFO clock crosser that which buffers the commands and responses should also be available to support higher throughputs at the expense of higher logic utilization.

Pipeline Stage Insertion – For systems with a large number of masters and/or slaves, the achievable frequency is often less than that desired by the system designer. To allow for higher frequency operation, Merlin must provide a mechanism to break long critical paths into smaller segments, to allow for higher performance.

Error Responses – Merlin must include the ability to return error indications to a master, such that the master is aware when a transaction failed.

Timeouts – Merlin must be able to support timeouts, where a transaction is automatically ended and a error code returned to the master when there is no response to a transaction.

Flexible Arbitration Mechanism – System designers often need custom arbitration algorithms to suit their applications. For example, the arbitration for a DDR memory slave might consider the DDR memory banks that are currently open, and accepts transactions so as to maximize memory bandwidth. Merlin must allow user-provided arbiter components in place of the default ones.

3.3 Architecture Overview

The Merlin architecture is a layered architecture where the master and slave are peers, occupying the same position on the protocol stack. The benefits of a protocol-stack approach is that the implementations of components at different layers of the stack are independent of components at other layers. This means that the implementation of a switch component can be changed without affecting the other components, as long as they continue to provide packet transport functionality.

Another significant benefit is that the components at different layers can be verified independently of one another. It is possible to test the implementation of a switch fabric with arbitrary packets, without requiring components to source and sink the packets.

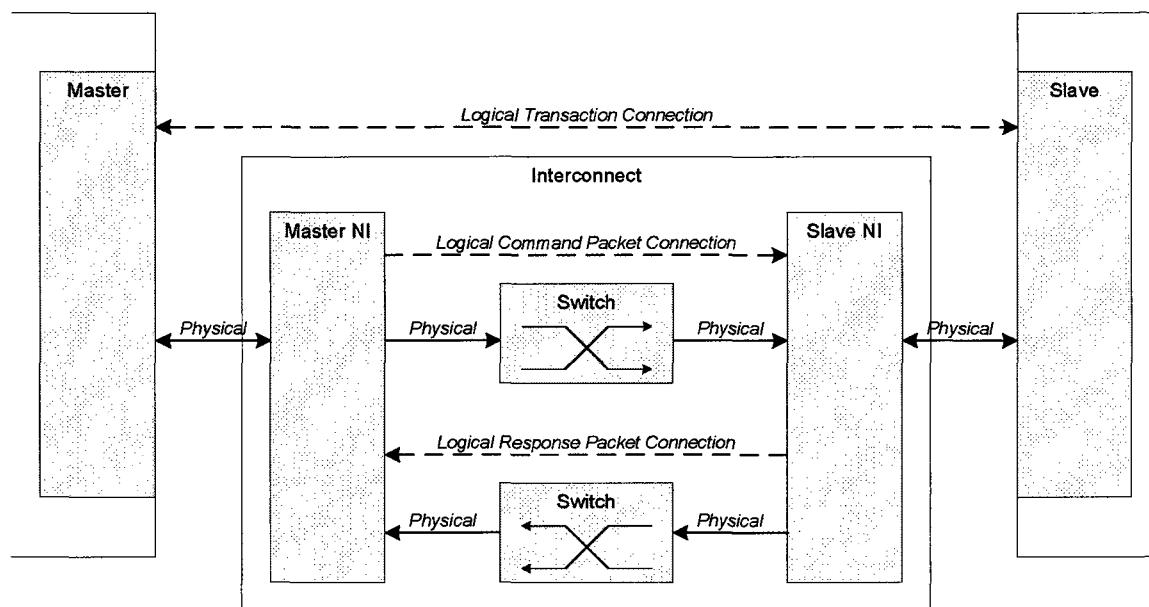


Figure 20 - Interconnect Implementation

Merlin's structure is shown in Figure 20. The master and slave communicate with each other using the Logical transaction connection made available by the interconnect, which allows the master to write to and read from the slave. From the master's point of view, it is communicating directly with the slave via the logical transaction connection. For example, when the master issues a read command, the slave receives the read command and responds with read data, perhaps after asking the master to wait. The master sees and responds to the wait indication, accepting the read data when it's available. Neither the master nor the slave are aware of the interconnect, whose purpose is to provide the logical transaction connection.

In a similar fashion, the Master Network Interface (NI) and Slave Network Interface in Figure 20 are peers; the Master NI sends command packets to the Slave NI, and the Slave

NI sends response packets to the Master NI. Again, neither of them are aware of how the packets are being transported.

It is important to note that the different layers of the protocol stack cannot depend on the specific implementation of the lower layer. For example, the network interface does not care how the packet transport mechanism works, as long as the packets are transported. This allows the characteristics of the packet transport layer, such as switch topology, pipelining, and arbitration, to be changed to suit the application without affecting the design of the master, slave, or network interfaces.

With Merlin, parameters guiding the detailed implementation of each component is determined at interconnect generation time, as a function of the characteristics of the system being developed. The command packet switch, for example, might support a 32 bit data path in one implementation, and a 64-bit data path in another. The response packet switch might be a purely combinatorial switch, optimizing for latency, or it might be heavily pipelined to achieve a high operating frequency. Likewise, implementations of Network Interfaces components depend on the system itself. In one system, the NIs may not need to support bursting at all, where in another system, they may need to support both interleaved and line wrapped bursts. In fact, since the NIs communicate with each other independently of the implementation of the switch, and since the switch carries packets without knowing what's in them, the packet format between the master NI and the slave NI can be different for different systems, or even for different segments in the same system.

3.4 Interfaces

Figure 21 illustrates the interfaces that are present in the Merlin interconnect. Starting from the outside, the master presents a *transaction interface*, with which it issues transactions and collects responses. Likewise, the slave uses a transaction interface to accept and respond to commands. The interfaces on the switch side of the master NI and slave NI are identical, and are *Transaction Packet interfaces*. The Master NI and Slave NI must agree on the details of this interface type. The switch components see only *Packet Transport interfaces*. They accept packets on one side, and ensure delivery of those packets to the other side of the switch.

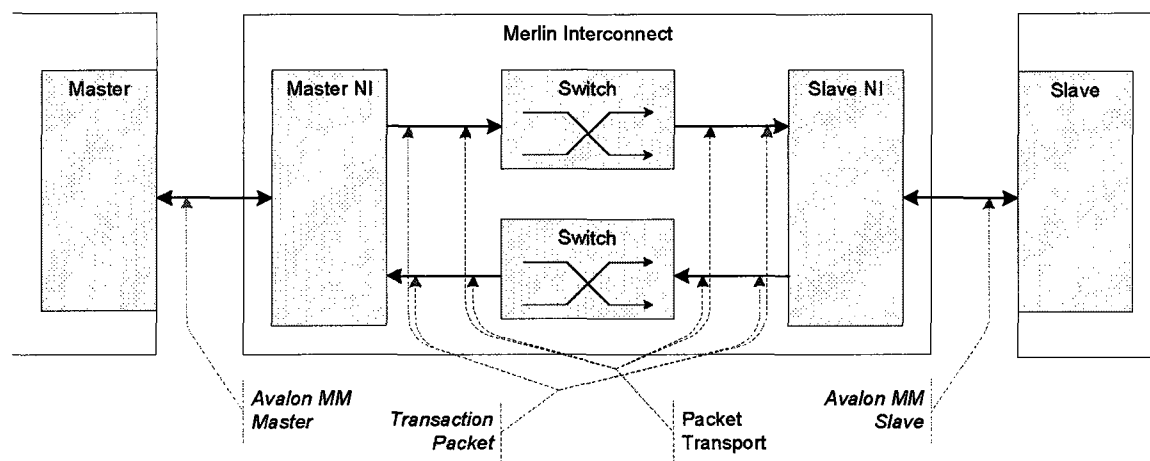


Figure 21 - Merlin Interfaces

The astute observer will note that the transaction packet interface from the Master NI, and the packet transport interface on the switch to which it is connected are different interface types. Likewise the packet transport interface on the right side of the switch is different from the input of the Slave NI it is connected to. All of the information communicated between the master NI and the slave NI is carried as part of the data of a

packet; as long as the master and slave NI components agree on the data format inside the packet, they don't care how the packet is transported.

3.4.1 Transaction Interfaces

Masters and slave interfaces connect to the Merlin interconnect via Avalon MM Master and Slave interfaces, which are defined according to the Avalon Memory Mapped Specification [47]. The transaction interface is the interface that masters and slaves interact with to issue commands and responses, and is implemented such that masters and slaves can carry on transactions in a peer-to-peer fashion. It accepts commands from its master interfaces, and drives those commands to slave interfaces, collects responses from the slaves, and returns the responses to the masters.

3.4.1.1 Avalon MM Signals

The nature of the Avalon interface specification is such that there are a multitude of signal combinations and parameters that an interface can provide; the component designer can use active high or active low versions of many of the signals, can provide fixed wait-state responses instead of signaled wait state responses, and can support only a subset of the interface features, if required. The following is a list of the standard Avalon Memory Mapped signals; for more details, see reference [47].

Read – The single-bit *read* signal is asserted by the master while a read is being issued, and is held until the read has been accepted.

Write - The single-bit *write* signal is asserted by the master during the beginning of a write operation, and is held until the write has been accepted.

Address - *Address* is a multi-bit signal driven by the master to indicate the destination for the transaction. This signal determines what slave interface the transaction gets routed to, and determines what data within the slave interface is being written to or read from.

Writedata - The *writedata* signal carries the data from the master during write transactions.

Readdata - The *readdata* signal carries the data returned to the master during read transactions.

Byteenable - *Byteenable* is a multi-bit signal with a single bit for each data byte in the interface. During a write transaction, the n^{th} bit in *byteenable* is asserted to indicate that the n^{th} byte in *writedata* contains valid data, and that the other bytes in the *writedata* signal should be ignored. Likewise, for read operations, *byteenable* indicates which bytes in *readdata* are being fetched by the master; the other bytes will all be ignored.

Burstcount - *Burstcount* is an optional multi-bit signal that indicates the size of a burst. *Burstcount* is only valid during the first cycle of a burst, and indicates the size of the burst in units of cycles to be transferred. Section 3.4.1.2 describes Avalon MM bursts in more detail.

Waitrequest - *Waitrequest* is a signal in the slave-to-master direction that indicates that a transaction cannot be accepted. If a master attempts to write to a

slave while it is busy, the slave will assert *waitrequest*. The master must then wait until *waitrequest* is deasserted to continue.

Readdatavalid – The *readdatavalid* signal is used to indicate to a master when the data from a previous read is available. *Readdatavalid* is asserted when the *readdata* signal is valid. Once the master issues a read command, it must be ready to accept the read data at any time.

Begintransfer – *Begintransfer* is a convenience signal asserted for the slave at the beginning of every transfer. This signal is generated by the Avalon interconnect, and is not provided by the master.

Beginbursttransfer – *Beginbursttransfer* is a convenience signal for the slave asserted at the beginning of every burst. This signal is generated by the Avalon interconnect, and is not provided by the master.

3.4.1.2 Avalon MM Transactions

This section describes the most important Avalon MM transactions that Merlin supports. Although information about these transactions is available from reference [47], they are described here because of their importance to the Merlin interconnect.

Figure 22 illustrates simple Avalon MM read and write transactions. To begin a transaction, the *read* or *write* signal is asserted by the master. If the slave is not ready for the transfer, it responds within the same cycle by asserting *waitrequest*. For a write transaction, the transaction is complete as soon as *waitrequest* is deasserted for a cycle. For a read transaction, there are two cases: if the interface does not support

readdatavalid, the *waitrequest* signal is only deasserted once valid read data is available. If the slave does support the *readdatavalid* signal, then deassertion of *waitrequest* signal that the read command has been accepted; the read response is signaled independent of the read command.

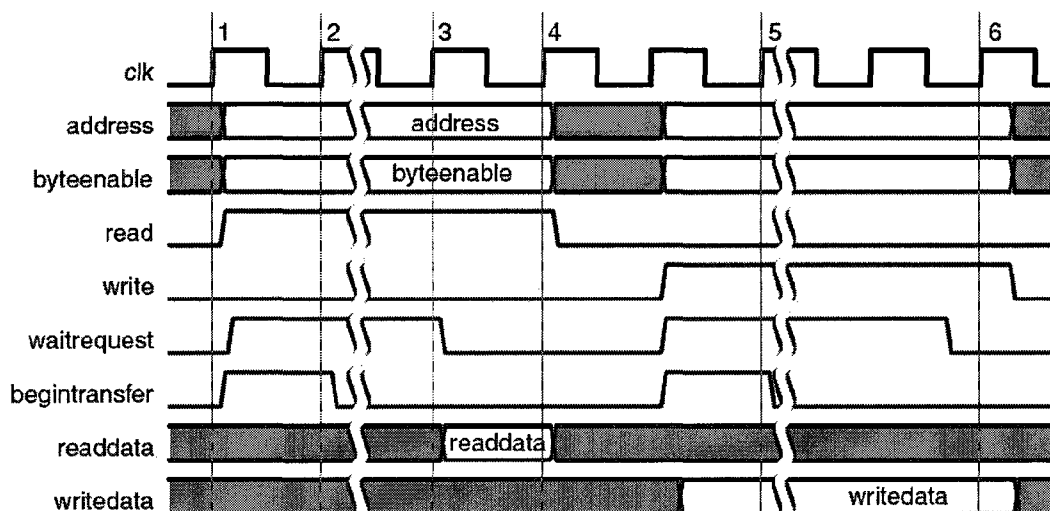


Figure 22 - Simple Avalon MM Transactions [47]

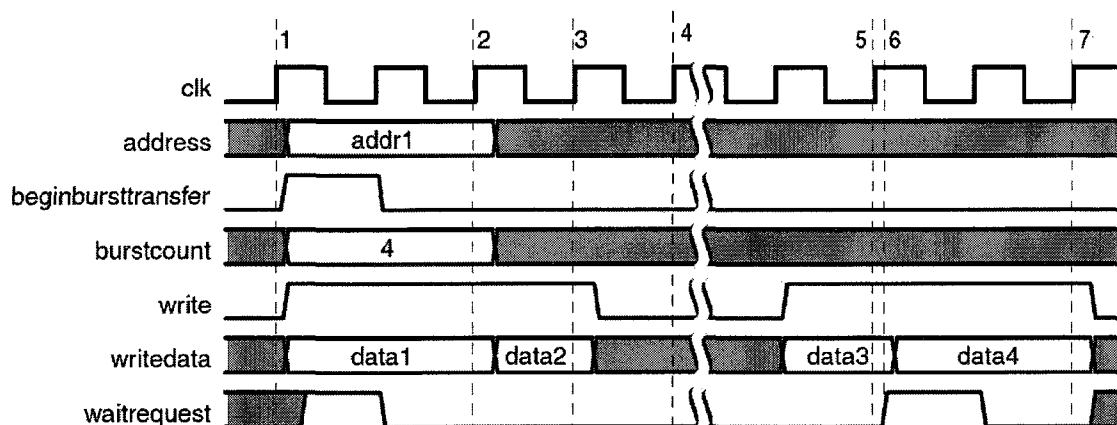


Figure 23 - Avalon MM Write Burst Transaction [47]

Figure 23 illustrates a write burst as seen by an Avalon MM slave interface. As with the previous case, the write transaction is issued with the assertion of the *write* signal. During the same cycle, the *burstcount* and *address* signals are valid. The *burstcount* signal indicates the length of the burst, and the transaction is only complete once *burstcount* words of data have been transferred. The slave is able to pause a burst at any time by asserting *waitrequest*, and the master may pause a burst in the middle by deasserting the *write* signal. Regardless of how the transaction is paused, it does not complete until *burstcount* words have been transferred. Note that the slave can only rely on the *burstcount* and *address* signals during the first cycle of the transfer.

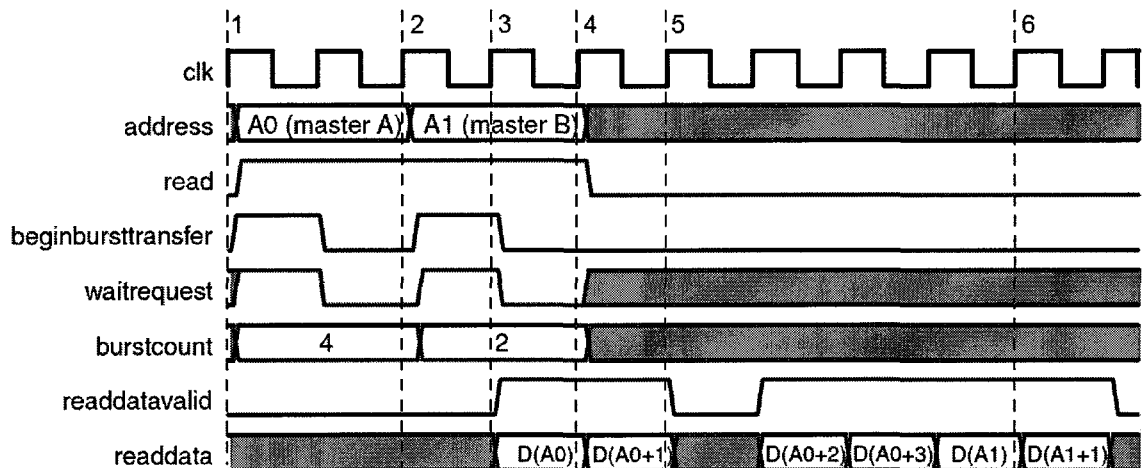


Figure 24 - Avalon MM Read Burst Transaction [47]

Figure 24 illustrates a read burst as seen by an Avalon MM slave interface. The read transaction is issued with the assertion of the *read* signal, and the *address* and *burstcount* signals are valid while *read* is. Like the previous two transactions, the slave can delay accepting the read command by asserting *waitrequest*. Once the slave has accepted the read command, it must return *burstcount* words of data to the master. It does this by

asserting the *readdatavalid* signal with valid read data every time a cycle of read data is available. As shown in the illustration, the slave can pause the return of data for a given transaction. There is no way for a master to pause the data being returned by a slave once it has been requested.

3.4.2 Transaction Packet Interfaces

Transaction packet interfaces carry information about a transaction command or response as the payload of a packet. Command interfaces are completely compliant with the packet transport specification described below in section 3.4.3; they support all of the packet transport signals, and define the data signal to be a number of fields for carrying different pieces of information. A given transaction packet can be either a command or a response; the packet format for the two is the same. A transaction packet contains information such as the transaction type and address, the write or read data, the byte enable signals, and the packet's destination.

Because of the flexibility demanded by the interconnect, the interface itself is parameterizable, such that the widths of fields within the packet are a function of the characteristics of the masters and slaves being connected. Two components can communicate across a packet transaction interface only if they agree about the width of each of the fields. Components such as a width adapter can be used to convert from a transaction packet interface with one set of parameters to another.

3.4.2.1 Packet Fields

The fields in the payload of a command or response transaction packet include the following:

Data – The data being written or returned in the case of write and read transaction, respectively. This consists of a number of n-bit symbols, where both the width of the symbols and the number carried per cycle are dependant on parameters.

Byteenable – *Byteenable* contains a bit for each symbol in the data field. For writes, each bit in the *byteenable* signal indicates that the corresponding symbol is being written to. For reads, each bit in *byteenable* indicates that the corresponding symbol is being read from.

Address – The byte address of the transaction. A portion of the address is used to determine what slave to send the packet to, and a portion is passed onto the slave itself for internal addressing.

Size – The *size* field carries the remaining size of the transaction, in bytes, rounded up to be a multiple of the data width. For multi-cycle burst transactions, the size is always the remaining size of the burst. If byte enables indicate unused data bytes in the middle of the burst, the burst size indicates the number of bytes until the last cycle in the burst, regardless of byte enables.

Command – The *command* field indicates the command to be executed at the slave, such as read, posted write, etc.

Destination –The *destination* field indicates the packet’s destination. In the case of a packet containing a command, this is the slave ID, an identifier that uniquely identifies the slave interface that the command is destined for. In the case of a packet containing a response, this is the master ID, which uniquely identifies the master that this response is intended for.

Source – The source field indicates where the packet came from. This contains the master ID of the master that issues a command, or the Slave ID of the slave that is returning a response.

3.4.2.2 Packet Format

The packet is constructed such that for packets that contain multiple cycles worth of data, each cycle contains all of the information required to do a single cycle command.

The example shown in Figure 25 is a 6-byte write burst to address 0x010040, where symbols are 8 bits wide. Throughout the packet, the ‘src’ field indicates that this command is from master 1, ‘dest’ indicates that it’s to slave 3, and ‘cmd’ is ‘WR’, indicating a posted write. During the first cycle of the packet, labeled cycle 0, the burst size field indicates that there are 8 bytes remaining in the transaction. The 32 bit data field is used to send the data ‘01 02 03 04’, with all bytes valid. On the second cycle, the master and command are the same, but there are now only four bytes left in the transaction, and the packet fragment is destined for address 0x010044. The data field includes enabled data ‘05’ and ‘06’, as well as two bytes that are disabled. Note that

even though the last cycle only includes 2 bytes of valid data, the size field is four, because there are a total of four enabled and non-enabled bytes.

	cmd (3b)	dest (4b)	src (4b)	size (6b)	address (32b)	byteen (4b)	data (32b)
Cycle 1	WR	3	1	4	0x010044	0x3	xx xx 06 05
Cycle 0	WR	3	1	8	0x010040	0xf	04 03 02 01

Figure 25 - Example Write Command Packet

Figure 26 and Figure 27 illustrate a read burst command and response, respectively. The command packet is four cycles long, with one cycle to request each four-byte transaction. Throughout the packet, the destination field indicates that this is for slave #3, and the source field indicates that this is from master #1. During the first cycle, the size field indicates that the remaining number of bytes being requested is 16, and this field decrements throughout the burst command, while at the same time increments the address. The data field contains no data, since this is a read command.

	cmd (3b)	dest (4b)	src (4b)	size (6b)	address (32b)	byteen (4b)	data (32b)
Cycle 3	RD	3	1	4	0x01000c	0x3	xx xx xx xx
Cycle 2	RD	3	1	8	0x010008	0xf	xx xx xx xx
Cycle 1	RD	3	1	12	0x010004	0xf	xx xx xx xx
Cycle 0	RD	3	1	16	0x010000	0xf	xx xx xx xx

Figure 26 - Example Read Burst Command Packet

The read response packet is nearly identical to the command packet. The command, size, address, and byteenable fields are the same as in the original packet, and the source and

destination fields are reversed, since this packet is now from the slave and to the master. The data field is now filled with the read data.

	cmd (3b)	dest (4b)	src (4b)	size (6b)	address (32b)	byteen (4b)	data (32b)
<i>Cycle 3</i>	RD	1	3	4	0x01000c	0x3	xx xx DD CC
<i>Cycle 2</i>	RD	1	3	8	0x010008	0xf	BB AA 99 88
<i>Cycle 1</i>	RD	1	3	12	0x010004	0xf	77 66 55 44
<i>Cycle 0</i>	RD	1	3	16	0x010000	0xf	33 22 11 00

Figure 27 - Example Read Burst Response Packet

3.4.2.3 Performance and resource utilization tradeoffs

At a first glance, there are a couple of things about the packet format described above that seem inefficient. This section describes some of the tradeoffs that were considered in the design of this packet format.

Consider first the packet size field. If the first cycle in a 32-bit interface indicates that the transaction contains 16 bytes, the number of bytes in each of the remaining cycles can be easily derived, and doesn't need to be repeated for every cycle. The fact that the information is presented in every cycle may look to be inefficient for throughput. If this were a traditional packet interface such as Ethernet, where data is all serialized, then this would indeed be true; sending the same information multiple times within the packet would be inefficient. But in this case, each field within the packet has it's own signals to travel on. If subsequent cycles weren't to carry this information, the signals would either

be unused, which isn't any more efficient, or the fields would be reused to carry other information.

If the fields were reused to carry other information, then in addition to the extra logic to multiplex data onto and off of those signals, each cycle of the transaction would not be able to stand alone, and multiple cycles would have to be received before enough was known about the transaction to present it to the slave, increasing effective latency. Depending on what information was carried, this might also make single-cycle transactions impossible, since only some of the required information would be present in each cycle.

Simply leaving the fields unused would be less work for the master network interfaces, since they wouldn't need to increase the address field and decrease the size field throughout the packet. If the masters don't do this computation, however, each slave would need to do it, because each slave still needs to know the address that each word of data goes to or comes from, and they need to know the remaining burst size. In typical systems, the slaves outnumber the masters. Requiring masters to replicate control information in subsequent cycles of a command packet complicates the master's network interface design at the expense of increased logic, but reduces the complexity of the slaves.

It is also true that not all of the fields in the packet are strictly necessary for all slaves and/or masters that receive them. For example, masters typically don't need to know the source, size, address, or byte enable fields of the response packet. With modern FPGA synthesis tools, the net list optimization algorithm removes all logic that contributes to

nodes that aren't used by downstream logic [48] [49]. Defining these fields in the packet format makes the information available to components that need it, while the synthesis optimization algorithms ensure that the price for the logic isn't paid if the information isn't used.

3.4.3 Packet Transport Interfaces

The packet transport interface is used to transfer packets between two components on the FPGA. The transport interface is agnostic to the information being carried in the packet itself. One side of the transport mechanism's interfaces acts as a packet *sink*, accepting packets from another component, while the other interface acts as a *source*, providing packets to another component. The packet transport interface used by Merlin is based on the Avalon Streaming interface described in [47].

3.4.3.1 Signals

The following signals are used by the Packet Transport interface to transfer packets between a source and a sink.

Ready - Ready is a single-bit signal from the sink back to the source, asserted by the sink when it is able accept data. Nothing is transferred on clock cycles where ready is deasserted.

Valid - Valid is a single bit signal from the source to the sink indicating when the source has valid data. Nothing is transferred on cycles when valid isn't asserted by the source. A transfer of data and control signals only occurs when both the sink asserts ready and the source asserts valid.

Startofpacket - *Startofpacket* is asserted high to mark the cycle containing the start of the packet.

Endofpacket - *Endofpacket* is asserted high to mark the cycle containing the end of the packet.

Channel - *Channel* is a multi-bit signal indicating the destination for the data carried in the current cycle. While the Avalon Streaming specification allows packet interleaving where a complete or partial packet can be sent for a given channel before switching to a packet segment for a different channel, Merlin only transfers complete packets. This simplifies the implementation of the Slave and Master network interfaces, and guarantees that a burst transaction encapsulated in a packet isn't interrupted. To allow the high performance arbiter implementation described in section 4.3.1.3, Merlin uses one-hot coding of the packet's destination, where each bit in the channel signal is a packet destination, and only the bit for the intended destination is asserted high.

Data - The *data* signal carries the packet payload being transferred by the interface. The width of this data can be parameterized.

3.5 Components

The Merlin interconnect is composed of a number of individual components that support the interfaces described above, such that they can be interchanged and connected in different ways. This section describes each of the components, and describes its functionality and the services it provides. Although not mentioned in the individual

sections, the design of most components are such that a command or a response shows up at the component's output in the same cycle that it arrives at its input, allowing a purely combinatorial (zero clock cycle) delay through the entire interconnect. Timing-specific components such as the register stage and FIFO can be used to break the long combinatorial path, allowing for higher frequency, at the expense of higher latency. The components typical of a Merlin system are shown in Figure 28.

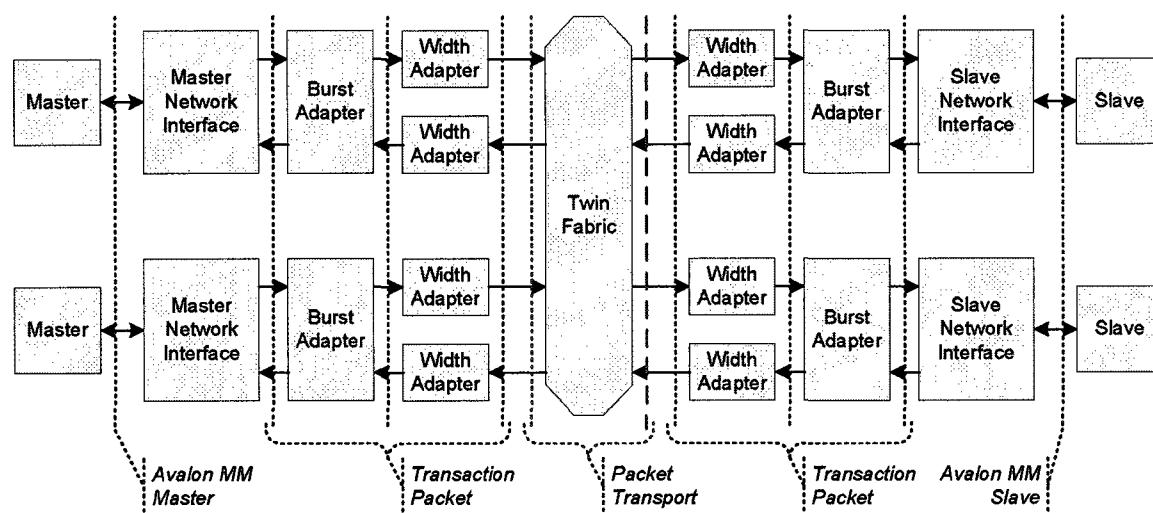


Figure 28 - Merlin Structure

3.5.1 Master Network Interface Component

There is a Master Network Interface (MNI) component such as that shown in Figure 28 for every master in the system. The MNI is responsible for accepting a command from a master, and converting it into a packet to transport to a slave's network interface. It is also responsible for accepting a response packet, and converting that into a response to the master's request. The services provided by the MNI are listed below. It is worth noting that any given system implementation may only require a subset of the services

available, in which case support for services can be disabled, decreasing implementation cost.

Protocol Translation - Interfaces such as Avalon MM have a multitude of options, such as whether control signals are active high or low, and whether or not to take advantage of optional features such as pipelined reads and bursts. The MNI is responsible for translating the variation used by the master into a common format.

Command Packet Creation - The MNI is responsible for constructing a command packet such as that described in section 3.4.2. To keep latency to a minimum, it is important that the first cycle of the packet can be constructed combinatorially, such that the cycle that the master presents the transaction to the network interface is the same cycle that the beginning of the packet is pushed into the downstream component.

Address Decoding – Masters send commands in terms of write and read transactions to a given address. The address decode logic in the MNI encapsulates all of the information about the master's address map. This consists of a combinatorial function that uses the address bits to determine the endpoint for the transaction, and drives the channel signals indicating this end node.

Timing Control – The MNI must provide for the timing control of the master. This means causing the master to wait when the network isn't ready for a new command, and for indicating to a master when read data and other responses are available.

Transaction Ordering Control – It is common for masters to require that responses to commands are received in the order in which they were issued. For masters with this

requirement, the MNI must guarantee that the response order requirements are met.

Since the packet transport layer guarantees that command and response packets are delivered in order, and the slave guarantees that it will respond to commands in the order that they were received, the Master NI needs only to endure that responses from commands issued to two different slaves are returned to the master in order. The requirements can be met by never allowing commands that might result in out of order responses to be issued, or by reordering out-of-order responses for the masters' consumption.

Response Packet Termination - The MNI receives a response packet such as that described in section 3.4.2, and presents the master with response information and timing control. For posted writes, there is no response packet, and the response termination function does nothing. For non-posted write transactions, the response consists of only a transaction complete indication, and the *waitrequest* signal to the master is held asserted high until the response is received. For non-posted reads and posted reads, the *waitrequest* and *readdatavalid* signals respectively are high until the response packet with the read data is received.

Timeout Watchdog - The timeout watchdog starts a timer every time a command packet is issued. If the timer runs out before the corresponding response packet is received, the transaction is cancelled, and the master is informed via the bus error signal.

Bus error detection and reporting – It is up to the MNI to provide for bus error detection and reporting, such as the case of a master issuing a command to an address with no slave, or a transaction timeout expiring before an expected response has been received.

3.5.2 Slave Network Interface

This section describes the implementation of the Slave Network Interface (SNI) component shown in Figure 28. This component is responsible for accepting command packets from the fabric, driving those packets to the slave using transaction layer signals, and collecting the response from the slave to create the response packet. The responsibility of the SNI includes the following:

Protocol Translation – As with the MNI, the SNI must translate the variation used by the master into a common format that can be used by the network.

Command Packet Termination - The command packet termination logic accepts and decodes the command packet, and drives the command to the slave.

Response Packet Creation – The SNI creates a response packet such as that described in section 3.4.2 to send back to the master. For posted writes, there is no response packet, and the response creation logic does nothing. For non-posted write transactions, as soon as the packet is accepted by the slave, a response packet is send back with an unused data field. For read transactions, a response packet containing the read data is sent back as soon as the data is available.

Response Destination Synthesis – Unlike in the master case, a slave does not provide the address of the master to send the response to; it simply responds to the command. It is up to the SNI to keep track of which master sent the command, and to ensure that the response goes to the right place.

Timing Control – The Network Interface must provide for the timing control of the slave. This corresponds to driving the control signals at the appropriate time, and responding to signals such as *waitrequest* and *readdatavalid*. In addition, with most slave implementations, once a command has been issued to the slave, the response cannot be backpressured, meaning that the SNI must be able to store the response temporarily in case the fabric is busy.

3.5.3 Width Adapter

The role of a width adapter in a Merlin system is to convert command or response packets at one data width into command or response packets at a different data width. For example, a network interface for a 32-bit master will issue command packets with 32 bits of the data signal reserved for write data. If the configuration of the fabric is such that the write data portion of the packet is 64 bits wide, then data width translation is necessary.

The width adapter, in addition to rearranging the write or read data in a packet, must ensure that the resulting addresses and burst count indications are correct. Consider, for example, a 32 bit to 16 bit width adapter. If a packet arrives containing a single 32-bit write to address 0x800, the width adapter must issue a packet on the output side containing a 16 bit write to address 0x800 followed by a 16 bit write to address 0x802.

3.5.4 Burst Adapter

In a heterogeneous system, there will be a mix of masters and slaves that support different burst characteristics, where a *burst* is simply a collection of transfers that are

being carried out as a unit. With a Merlin interconnect, bursts are used to accomplish two things. First, all transactions within a burst are kept together, and are not interrupted by other transactions, and second, at any point within a burst, the component receiving the burst knows how much data remains in the burst. Components such as DDR memory controllers operate more efficiently when they read or write multiple cycles to a single memory bank. In order to realize this efficiency gain, the memory controller needs to know at the beginning of a burst how much data will be available.

The role of a burst adapter is essentially to convert from one burst type or burst size to another burst type or burst size. For example, if a master issues an 8-cycle write burst to a slave that doesn't support bursting, the burst adapter must break the burst into individual transactions. If the master instead issued an 8-cycle read burst, the burst adapter would have to issue 8 individual transactions, and would also have to collect the eight responses into a single response packet.

The services provided by a burst adapter include the following:

Command segmentation – When considering the burst adaptation required for a command, there are three cases to consider.

(a) Burst size and addressing type match - In the first case, the addressing type in the arriving burst matches that of the slave, and the size of the burst is less than the slave's maximum burst size. In this case, there's no work for the burst adapter to do.

(b) Burst size mismatch - In the second case, the master and slave agree on the burst addressing scheme, but the burst from the master is larger than the slave's maximum burst size. In this case, the burst must be segmented into smaller bursts.

(c) Addressing scheme mismatch - In the third case, the addressing of the burst set by the master is different than that expected by the slave. For example, the master might have issued an 8 cycle line-wrapped burst starting at word three, which wraps back to word 0 after word 7. If the slave happens to do sequential bursts, it will expect an 8-cycle burst starting at offset 3 to continue until offset 11. In this case, the burst adapter segments the original burst into two smaller bursts; one 5-cycle burst starting at address 3, and a second 3-cycle burst starting at word 0.

Response reassembly – In the case of read bursts, if the master requested a read burst of eight cycles, and the burst was broken up into smaller parts, the burst adapter must reassemble the slave burst responses into a single burst response for the master.

3.5.5 Twin Fabric

The role of the twin fabric in Figure 28 is to route command and response packets from masters to slaves and back. The 'twin' prefix comes from the fact that the twin fabric really implements two unidirectional fabrics. There is an input on the command fabric for every master, and an output for every slave. Likewise, there is an input on the response fabric for every slave, and an output for every master. The twin fabric provides

packet transport services for communicating between network interfaces. The services provided by the transport layer include the following.

Unidirectional Packet Transport – The twin fabric provides unidirectional packet transport from input to output interfaces. The user of the fabric can push packets into any input interface, and the transport layer will deliver that packet to the output interface indicated by the channel signal.

Arbitration and Backpressure – If multiple inputs are sending packets to the same output interface, the fabric performs arbitration such that the output interface gets the packet from the winner of the arbitration. The input interface that loses the arbitration is backpressured until it can send its packet. If an output interface is busy for any reason, the transport layer manages the backpressure and control signals such that any input interfaces that are trying to send packets to a backpressured output interface are themselves backpressured appropriately.

Source to sink packet ordering – The fabric guarantees that the packets sent by a source to a sink will be delivered in the order in which they were sent. Unless arbiterlock is used, the fabric does not guarantee that packets from other inputs will not be delivered between packets from the original input.

Complete Packets – The fabric guarantees that output interfaces will never see incomplete packets. If an input interface wins arbitration, then the transport layer will allow it to send a packet to the selected output interface until the packet is complete. An in-flight packet is never interrupted for another packet.

3.5.6 FIFOs

Although not shown in Figure 28, First In First Out queues, or *FIFOs*, can be used between components to buffer command and response packets. FIFOs can be used when the data width of the master is less than the common data width of the network. Without a FIFO, a narrow master might win arbitration for the slave of interest, but only provide data slowly across the network, tying up network resources. With the FIFO, the command message can be in the FIFO in its entirety before arbitration is granted. Additionally, dual clock FIFOs can be used to support masters and slaves on different clock domains.

3.5.7 Register Stage

The register stage component is intended solely to break a critical path into multiple segments to allow for higher frequency operation, at the expense of latency. In order to ensure that the critical path is broken in both the data and backpressure direction with no throughput penalty, the register stage must store two cycles worth of data, and there can be no combinatorial path from the input side of the register to the output side, and vice versa. Addition of a register state, of course, means that the path no longer supports zero-cycle latency.

3.6 Assembly Algorithms

To assemble a Merlin interconnect, a collections of algorithms are used to manipulate the original system description, which captures the designers intent, replacing it with a system in which the interconnect is fully implemented. Instead of having a single

monolithic algorithm that implements the entire system, a number of small algorithms are used, each of which gets the system closer to its end implementation.

This section describes some of the potential algorithms that would belong to a full Merlin implementation.

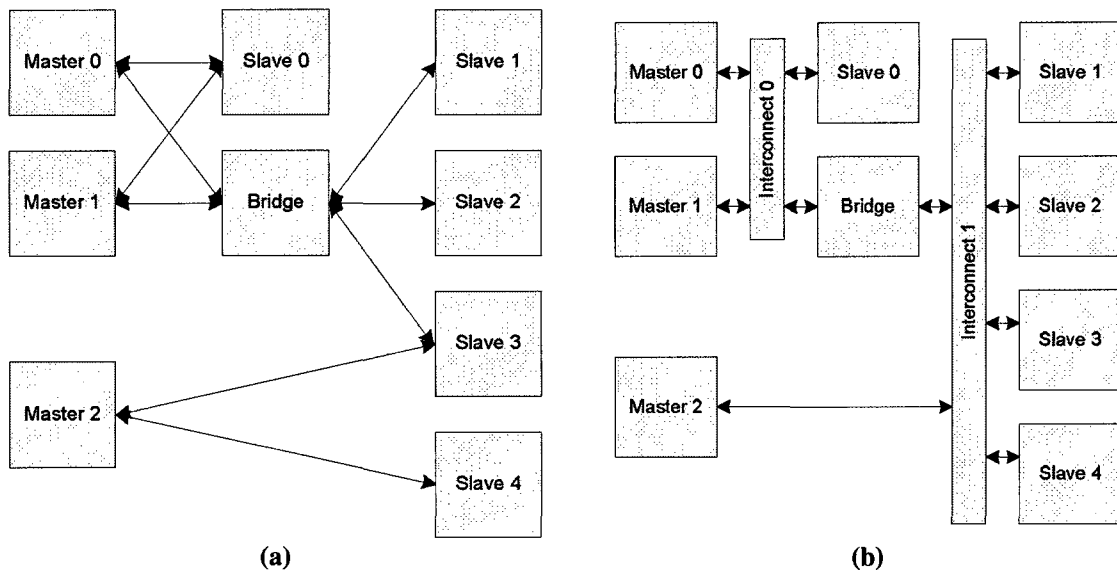


Figure 29 - Effect of Domain Definition Algorithm

Interconnect Domain Definition - The role of an interconnect domain algorithm is to determine how many interconnects are needed, and how they should connect to the modules in the system. Consider, for example, the system shown in Figure 29(a), where a pair of masters are both connected to slave 0 and a bridge, the bridge is connected to three components, and a third master is connected to one of the components behind the bridge and an additional slave component. The Interconnect Domain Definition algorithm would determine that one interconnect needs to support master 0 and master 1, while a second one is needed to support the bridge and master 2. The algorithm would

thus return the system shown in Figure 29(b), which has the interconnect domains defined.

Network Creation - For each interconnect block, a Network Creation Algorithm could insert and configure the fabric all of the network interfaces, and return a system that has the twin fabric and network interfaces, and that provides the desired packet connectivity, such as that shown in Figure 30.

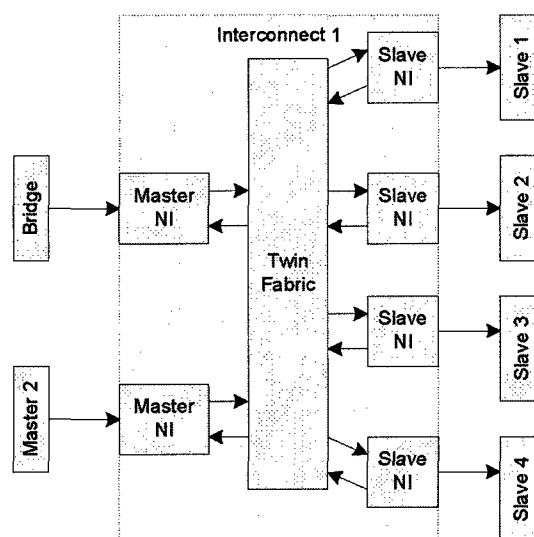


Figure 30 - Effect of Network Creation Algorithm

Network Topology Creation- In the case of the preceding example, Master 2 never needs to communicate with slave 1, allowing a topology that doesn't include that connection. The Network Topology creation algorithm takes into consideration the requirements of the twin fabric, and assembles the network topology using the available primitives. For example, Figure 31 illustrates a topology that supports interconnect 1 from the system in Figure 29(a) using simple split & join primitives.

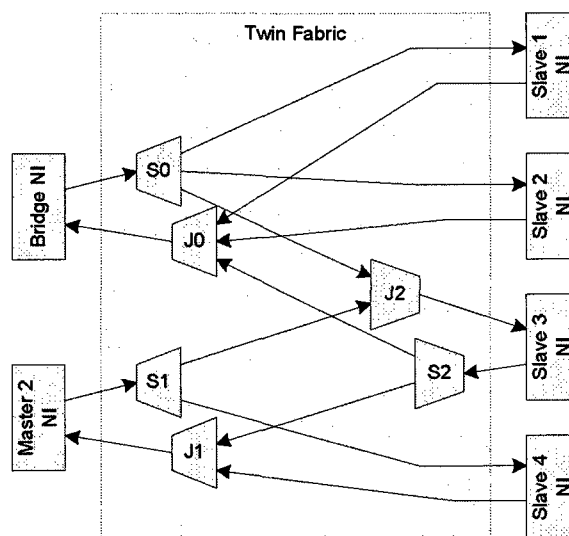


Figure 31 - Effect of Network Topology Creation Algorithm

Clock Crossing Logic Insertion - A clock crossing logic insertion algorithm would be responsible for adding clock crossing logic such as handshake modules or a dual clock FIFOs to the network to take care of the case where masters and slaves are on different clock domains. Considering the system from Figure 29(a), if master 2 and slave 4 happened to be on a different clock domain than the other components being supported by that interconnect, the clock crossing logic insertion algorithm might put a dual clock FIFO between S1 and J2, and between S2 and J1 from Figure 31. This would allow command and response packets to be exchanged between master 2 and slave 3, even though they are different clock domains.

4. Merlin Prototype Implementation

This chapter presents a prototype Merlin implementation which demonstrates a single-processor system communicating with a number of slaves over a Merlin interconnect. The motivation for this prototype is presented first, followed by the design methodology. The implementation of individual components and of system assembly algorithms is presented in the third and fourth sections, respectively. A description of a proof-of-concept system is then presented, followed by measurement results and comparisons with other FPGA interconnect implementations..

4.1 Motivation

A flexible packet-based transaction interconnect such as that described in the previous section presents a challenging design, both from a logic design point of view, and from a software development point of view. Due to the number of components, their complex interaction, and the myriad of ways that they can be arranged, it doesn't make sense to develop and verify a complete Merlin-based interconnect synthesis solution without first doing a prototype implementation. This prototype serves multiple purposes:

- Serves as a proof of concept, demonstrating that a working FPGA system can

indeed be implemented using a packet-based transaction interconnect.

- Demonstrates that the resource utilization and performance of such a system is reasonable.
- Provides a platform upon which to perform experiments that determine optimal topologies and tradeoffs.
- Allows for performance comparison with other FPGA interconnect implementations.

4.2 Methodology

To produce a working Merlin prototype, methodologies in a number of areas had to be developed or adopted. This section describes methodologies used in the development of this Merlin prototype.

Since Merlin has a number of highly parameterizable components, a methodology for running extensive parameterizable simulations was required. Many of the Merlin components are parameterizable beyond that supported by modern Hardware Description Languages (HDLs) such as SystemVerilog and VHDL, so some way of programmatically generating custom HDL for individual components based on their parameters was needed. To verify the functional correctness of these components, a way to programmatically generate simulation test benches was also needed. Given that at Merlin's core is a collection of components and the connections between them, there needs to be a way of representing collections of modules and connections, and a way of rendering these collections as HDL. There also needs to be a way to transform an

original system description, consisting of components of masters and slaves and the desired connections between them, into a realized system that implements the interconnect and required adapters.

4.2.1 Software Development and Testing

Since the development of Merlin included a considerable amount of software development, the software development and testing methodology was important. Merlin software is developed in Java, using Test Driven Development techniques. Test Driven Development is a style of development where the developer maintains an extensive suite of programmer tests, no code is released unless it has associated tests, the developer writes the tests first, and the tests determine what code gets written [50]. As Merlin code was written, functional tests were written as well, usually before the code itself. In order to execute the tests quickly and easily, the JUnit unit testing framework was used. JUnit is a simple, open source framework to write and run repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include assertions for testing expected results, test fixtures for sharing common test data, and test runners for running tests [51]. JUnit is included with the Eclipse development platform, which was used for Merlin development.

4.2.2 Component Generation

Some of the components developed for Merlin are more parameterizable than can be accommodated in standard hardware description Languages (HDLs) such as the Very High Speed Integrated Circuit Hardware Description Language (VHDL) or

SystemVerilog. For example, an address decoder component depends highly on the address map seen by the master, and it is not possible to encode the address map as a collection of SystemVerilog parameters or VHDL generics. To overcome this limitation, a pre-processing flow was developed to generate HDL as needed.

This flow, called ‘Twerp’, consists of two parts. The first part is a Java model for each component, and the second is a Twerp SystemVerilog file, which is pre-processed to generate the HDL. To generate a Twerp module, an instance of the component's Java model is created and parameterized. Once the component has been parameterized, the `getVerilog()` method is used to generate HDL specific to the parameterization. The `getVerilog()` method simply calls the Twerp function, passing in an array of parameter name / value pairs, and a Twerp SystemVerilog file to pre-process, and then collects the result.

The Twerp function itself works by executing the following pseudo-code:

```
Create a new source file that includes definitions of all of the
component parameters as variables;

For every line in the original file {
    If the line begins with '@@', {
        append it to the new source file with the '@@' removed.
    } else {
        append it to the new source file as a print statement.
    }
}

Execute the new source file, and collect the output.
```

Figure 32 - Twerp Algorithm Pseudo- Code


```

always @* begin
    in_ready          = out_ready;
    out_valid         = in_valid;
    out_data          = in_data;
    out_startofpacket = in_startofpacket;
    out_endofpacket   = in_endofpacket;
    address            = in_data[CP_ADDRESS_L:CP_ADDRESS_R];

    out_channel = 0;
    @@ for (set i 0) ($i < $num_slaves) {incr i} {
        if ((address >= $param(s${i}_start) ) && (address < $param(s${i}_end))) begin
            out_channel[$i] = 1;
            out_data[CP_SLAVEID_L:CP_SLAVEID_R] = $i;
        end
    }
end

```

Figure 33 - Twerp SystemVerilog Source Code Example

```

always @* begin
    in_ready          = out_ready;
    out_valid         = in_valid;
    out_data          = in_data;
    out_startofpacket = in_startofpacket;
    out_endofpacket   = in_endofpacket;
    address            = in_data[CP_ADDRESS_L:CP_ADDRESS_R];

    out_channel = 0;
    if ((address >= 0 ) && (address < 1024)) begin
        out_channel[0] = 1;
        out_data[CP_SLAVEID_L:CP_SLAVEID_R] = 0;
    end
    if ((address >= 10240 ) && (address < 11264)) begin
        out_channel[1] = 1;
        out_data[CP_SLAVEID_L:CP_SLAVEID_R] = 1;
    end
    if ((address >= 11264 ) && (address < 12288)) begin
        out_channel[2] = 1;
        out_data[CP_SLAVEID_L:CP_SLAVEID_R] = 2;
    end
    if ((address >= 22528 ) && (address < 23552)) begin
        out_channel[3] = 1;
        out_data[CP_SLAVEID_L:CP_SLAVEID_R] = 3;
    end
end

```

Figure 34- Twerp SystemVerilog Output

For example, Figure 33 shows the SystemVerilog code that implements the address decoder logic, before the Twerp pre-processing stage, and Figure 34 illustrates a small portion of the resulting SystemVerilog code after Twerp processing. The lines in the source code that begin with ‘@@’ define a for loop with index variable \$i which iterates

through each slave, such that the output channel and slave ID fields in the data packet are a function of the address input.

4.2.3 System Representation

In order to perform interesting system transformations, or to render complicated test benches as HDL, a way to represent collections of modules and connections was required. To satisfy this need, the hierarchy of Java objects shown in Figure 35 was developed to represent all the pieces of structural HDL, and this class hierarchy was then used to generate test systems for component testing, as well as for performance analysis.

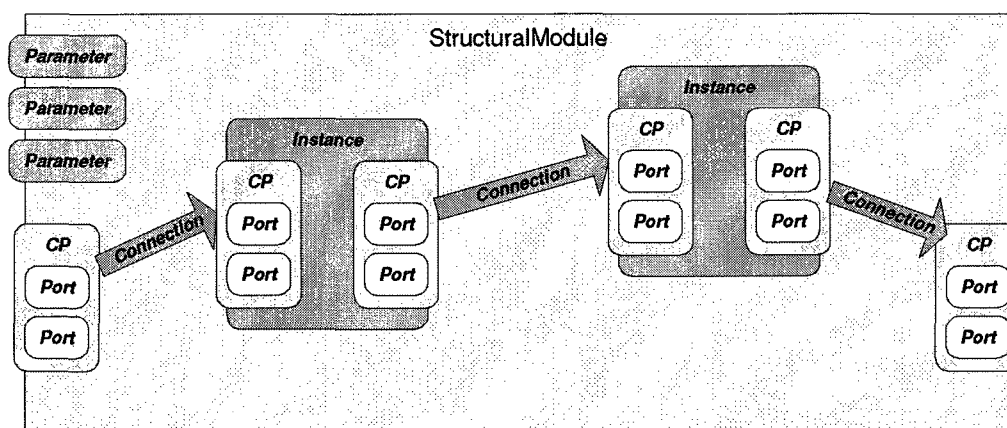


Figure 35 - Structural Module Hierarchy

The components in this class hierarchy are as follows:

Structural Module – A class representing an HDL module composed of instances and connections, with optional parameters.

Parameter – A simple key/value pair representing a SystemVerilog parameter or VHDL generic.

Connection Point (CP) – An object representing an interface on a component.

Each Connection Point has a type, such as “Avalon MM”, a direction, such as “Master”, and a collection of Ports.

Port – An object representing an input or output port in VHDL or Verilog. Each Port has a name, direction width, and role. A port’s role is the role it plays in the connection point, such as ‘*writedata*’. The port’s name does not have to match it’s role.

Instance – The instance of a component within a Structural Module. An Instance object itself has Connection Point objects, which in turn have Port objects.

Connection – Represents a connection between to connection points. Connections themselves can have parameters. The Avalon MM connection between a master and a slave, for example, has a base address parameter that defines the base address of the slave as seen by the master.

4.2.4 Tool Integration

This Merlin implementation relied heavily on Altera's SOPC Builder tool, a tool for SoC design capture and generation. Within the SOPC Builder tool, the system designer can add a number of components to the system, parameterize them as appropriate for the designs, and then specify the connections between them. As part of the connection specification, the system designer makes decisions such as the address map as seen by each master, the relative priority of different masters when accessing the slave, interrupt priorities for each interrupts source, etc. When the designer generates the system, SOPC

Builder generates the a top level file that instantiates all of the desired modules, parameterized appropriately, as well as an implementation of the interconnect which provides for the desired connectivity. As part of generation, SOPC Builder emits a *sopcinfo* file, which contains an XML description of the system, formatted such that it is machine readable.

Once a system has been generated using SOPC Builder, the prototype Merlin tool reads the *sopcinfo* file to determine the designer's top level system and desired functionality. It creates an in-memory model of the system, and then runs the assembly algorithms described in section 4.5. From there, it generates a top-level HDL file that instantiates all of the user-specified components and the interconnect component. It finally generates the interconnect component by generating and instantiating all the required Merlin components a described in 4.3.

4.3 Component Implementation

This section describes in detail the implementation of each of the components that were implemented as part of this prototype.

4.3.1 Twin Fabric component

The twin fabric component used to transport packets consists of a pair of combinatorial crossbar switches, each of which requires multiplexer and arbiter components.

4.3.1.1 Combinatorial Crossbar Switch Implementation

The key component of the transport layer implementation is the packet aware combinatorial crossbar switch, shown in Figure 36. Each instance of an $n \times m$ combinatorial switch has n input interfaces and m output interfaces, and provides connectivity from any input interface to any output interface. As shown in the figure, arbitration for the combinatorial crossbar switch is distributed, such that each output interface has an arbiter independent of all the other outputs. This allows different arbiter implementations to be used for different output interfaces within a single combinatorial switch. In addition to the arbitration block, each output has a multiplexer that routes the appropriate data to the output.

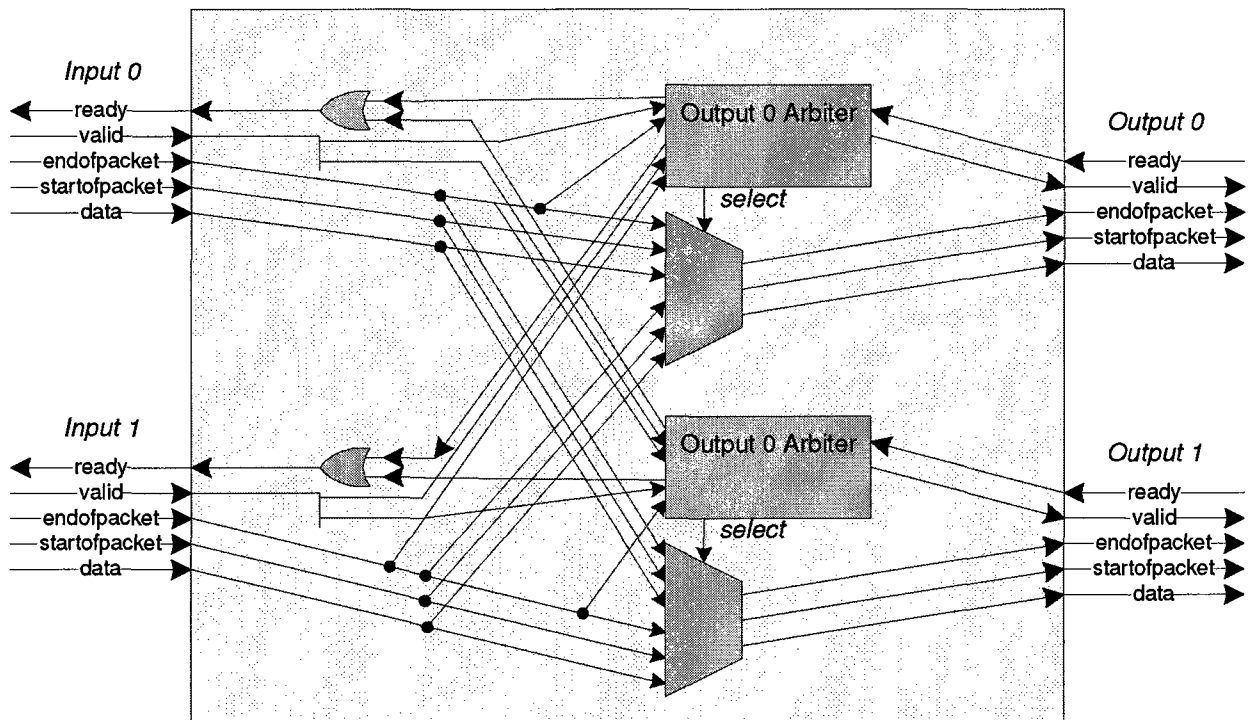


Figure 36 – 2 in x 2 out Combinatorial Crossbar Switch

With reference to Figure 36, each packet transfer begins with the presentation of data and the assertion of valid at an input interface. Valid is a vector with one bit for each potential destination, with at most one bit asserted at any give time. Bit 0 is routed to the arbiter for output 0, and bit 1 is routed to the arbiter for output 1, such that the arbiter for output N sees bit N of every input valid vector. The arbiter makes a scheduling decision, and indicates the decision to the multiplexer via a one-hot encoded select vector. The arbiter asserts ready back to the selected input such that it knows that the data it presented has been sampled, and that it can move on to the next data item. The arbiter also has as an input the endofpacket signal from each input interface, so that once the arbiter has selected an input interface, it can hold the selection constant until an entire packet has been transferred.

4.3.1.2 Multiplexer Implementation

Since the select input to the multiplexer is a one-hot coded vector, and not a binary coded select value, the structure of the multiplexer itself is implemented as a sum-of-and functions, as shown in Figure 37. One-hot coding of the multiplexer's select input was selected over binary encoding for the following reasons: First, the address decoding logic upstream of the multiplexer ends up with a single bit output for each slave, such that a pair of comparators can be used for each slave seen by a master, leading to a very efficient address decoder implementation. Second, the arbiter logic described below can take advantage of the FPGA's carry-chain, which has logic that has been optimized to allow for a large fan-in with a minimum performance penalty.

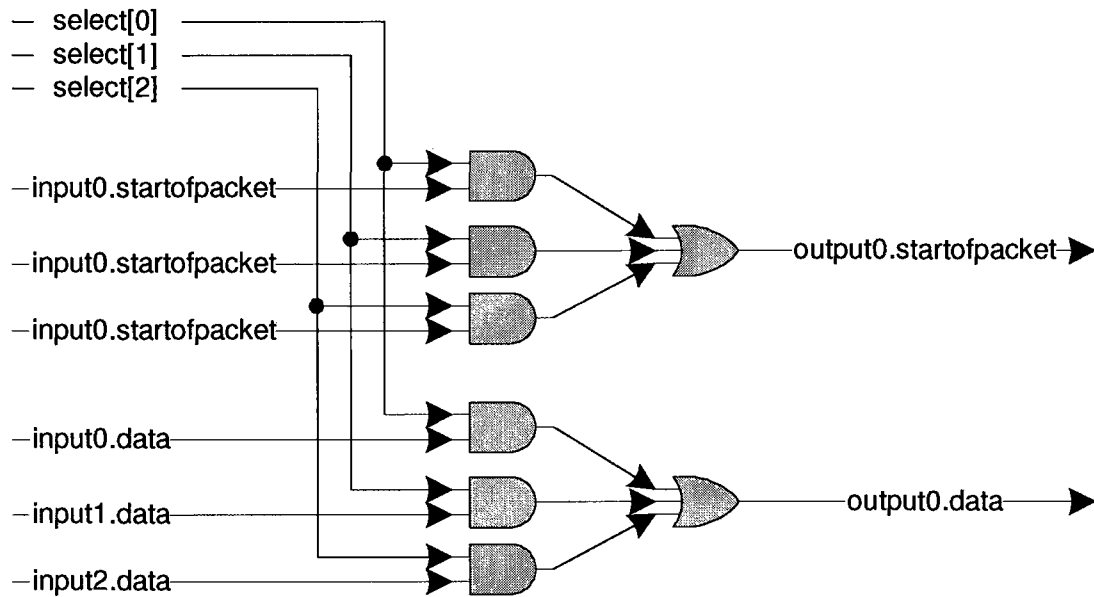


Figure 37 - Multiplexer Implementation

4.3.1.3 Arbiter Implementation

The arbiter takes as inputs a *valid* vector that includes all the sources currently trying to provide data for the output, a one-bit *endofpacket* signal that indicates when the current packet transfer is complete, and a one-bit ready signal that indicates when the resource can accept data. It provides as output a *select* vector that indicates which of the inputs are currently selected, with a bit for each of the input interfaces, indicating when the selected resource is ready for their input.

The arbiter also has clock and reset inputs, so the state used to make an arbitration decision can be maintained. On every cycle, the arbiter considers all inputs, and makes a decision for that clock cycle; the arbiter is not designed to support pipelined operation. On any given cycle, the arbiter's decision is only used if there is not currently a packet

being transported. A signal used to indicate that a packet is in flight is set as soon as a decision is made, and is cleared when the output *ready* signal, the selected *valid* bit, and the selected *endofpacket* bits are all true.

Modern FPGAs have a dedicated high speed carry chain between logic elements to allow arithmetic operations such as addition to incur a minimum of routing latency [52] [53]. By using an addition operation in the core of the arbiter, the arbitration structure shown in Figure 38 takes advantage of the FPGA fast carry chain routing, providing results optimized for FPGAs, while at the same time allowing for an arbitration decision in a single clock cycle.

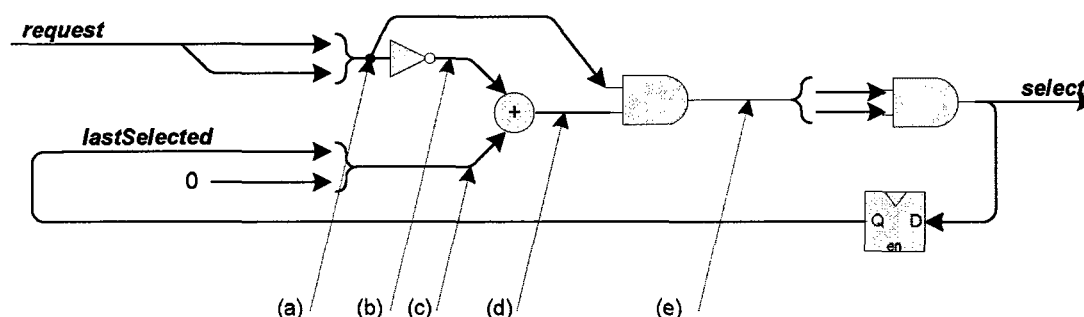


Figure 38 - Arbitration Implementation

Figure 39 provides an example of the arbitration algorithm shown in Figure 38 for an 8 bit arbiter. In this example, the previous selected output was 4, represented by a *lastSelected* vector of 0'b00010000, and inputs 0, 3, 4, and 7 are all requesting access, as represented by the request vector of 8'b10001101. The signals in Figure 38 are annotated to match the rows in Figure 39. As shown in the illustration, since the last input to have been granted access is input 4, the next input to win the arbitration is input 7.

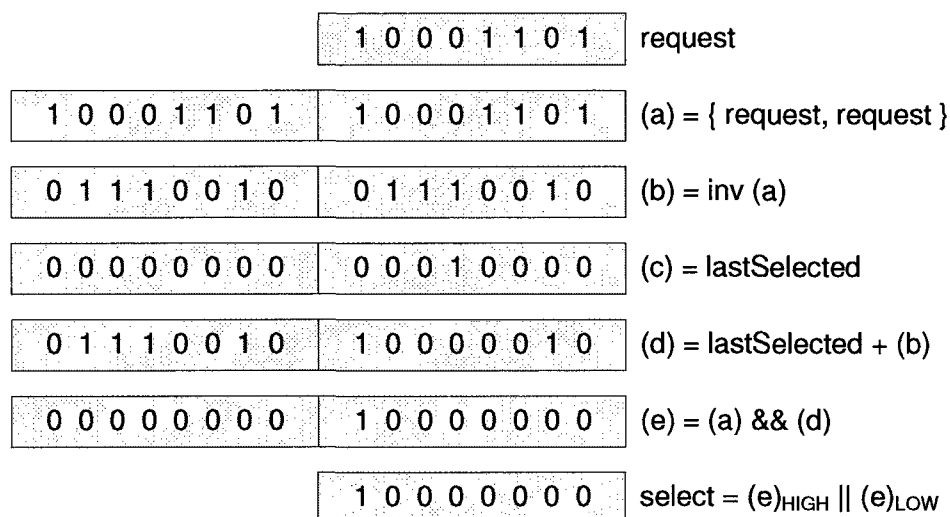


Figure 39 - Example arbitration

4.3.2 Master Network Interface

Figure 40 illustrates the subcomponents that make up the master network interface. The interface to the left is the Avalon MM interface that connects to the master that this NI is providing services for. To the right are the command packet and response packet interfaces.

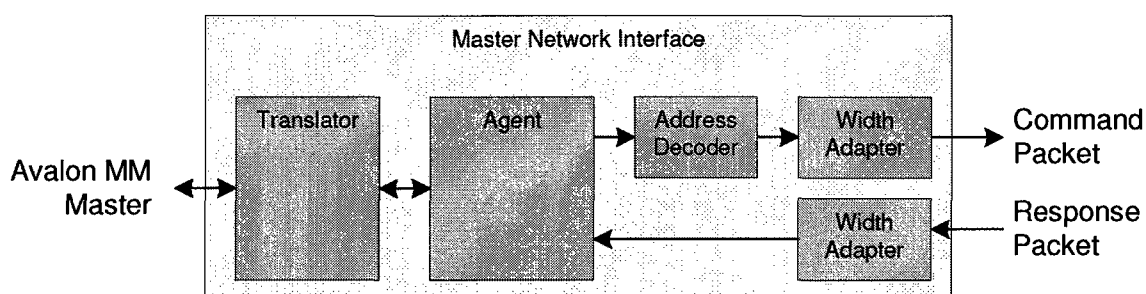


Figure 40- Master Network Interface

The master on the left drives transactions using Avalon MM bus signals such as *read* and *write*, an *address* signal to indicate the target for the transactions, a *byteenable* signal,

unidirectional *readdata* and *writedata* signals, and timing control signals such as *waitrequest* and *readdatavalid*. The Master NI turns the command into a command packet on the right, and accepts a response packet from the right to provide a response within the Avalon MM interface on the left. The subcomponents of the Master network interface are as described below.

4.3.2.1 Translator

The translator component is a conceptually simple component that converts the specific combination of signals used by the master into the standard set of Avalon MM signals described in section 3.4.1.1. The translation includes synthesizing active high read and write signals out of any combination of *read*, *write*, and *chipselect*, and inverting active low versions of all signals if necessary. It also ties unused signals off to the appropriate value, such as tying the *burstcount* signal to ‘1’ if the master doesn’t support bursts.

Although any given instance of the translator is simple, it has a significant parameterization space, and the ports it presents to its parent depend on this parameterization. To allow for this, the Twerp mechanism described in section 4.2.2 was used to generate the HDL.

4.3.2.2 Master Agent

The Agent component translates the master’s transactions into command packets, and accepts response packets on behalf of the master. When the *write* signal is asserted, the agent takes the *address*, *data*, and *byteenable* signals from the transaction interface, and creates a command packet according to the rules of the command interface. In the case

of a write command, the agent deasserts *waitrequest* as soon as the command packet has been accepted, allowing the master to continue with the next transaction. In the case of a non-pipelined read, the Agent waits for a response before deasserting *waitrequest*, and presents the read data from the response packet to the master using the *readdata* signal.

The Agent is one of the more complicated components, but its parameterization space is relatively small. Signal widths are parameterized, and a pair of boolean parameters determines if it is being used with a bursting master, a pipelined master, or just a simple master. As such, this was developed in SystemVerilog, and didn't require a generation mechanism like Twerp.

4.3.2.3 Address Decoder

The address decoder looks at the address field of the command packet that it receives, fills in the slave ID field of the outgoing command packets, and drives the packet interface's *channel* signal to the appropriate value. The parameterization space is essentially any imaginable memory map. A parameterization space of this magnitude can't be implemented in SystemVerilog or VHDL, so Twerp was used here as well.

4.3.2.4 Width Adapter

A width adapter is used when the master and slave components have differing data widths. With the example shown in Figure 41, if a master has a 64 bit data width, and one of the slaves has a 32 bit data width, a width adapter can be used to convert wide command packets into narrow command packets, and to convert narrow response packets

into wide response packets. Since the fields used by response packets and command packets are the same, the same width adapter component can be used in both places.

Although a width adapter adheres to the standard interface described in section 3.4.2 and can be used anywhere in the system that width adaptation is required, width adapters were placed within the master network interface to reduce the clutter in the generated top level. Width adapter parameterization consists only of defining the widths of each of the fields in the input and output transaction packet. For this purpose, SystemVerilog without Twerp is sufficient.

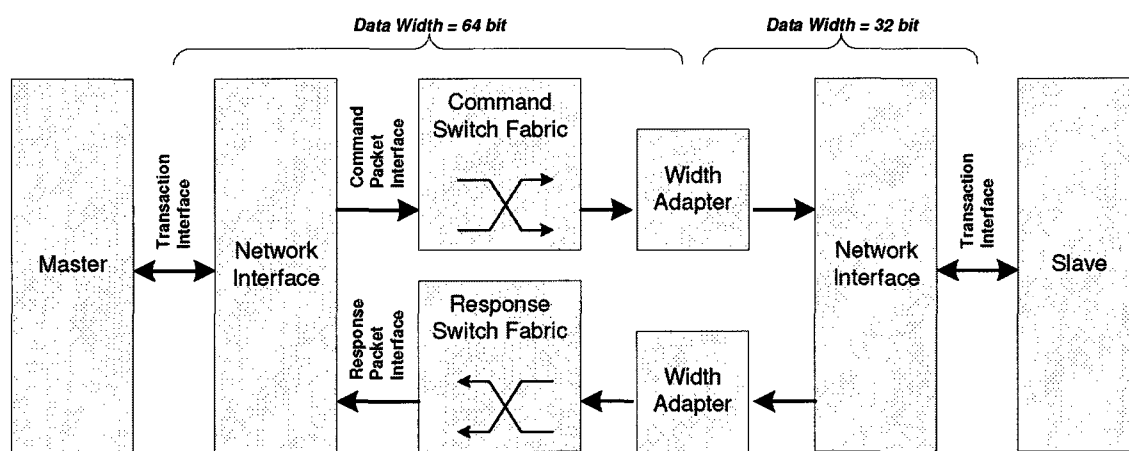


Figure 41 - Width Adaptation

4.3.2.5 Slave Network Interface

Figure 42 illustrates the subcomponents that make up the slave network interface. The interfaces to the left are the command packet and response packet interfaces from the fabric, and the interface to the right is the Avalon MM interface that connects to the slave. The width adapter components are exactly the same as those used in the Master

NI. The slave's translator is different than the master's translator, but it serves the same function, and is developed in the same manner, and so is not discussed further here.

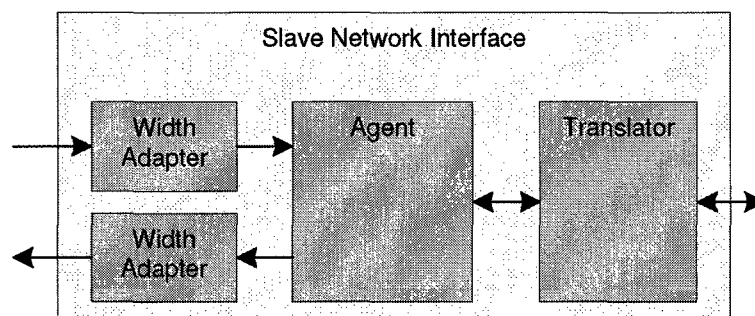


Figure 42 - Slave Network Interface

4.3.2.6 Slave Agent

The Slave Agent translates the command packets into transactions on the Avalon MM bus for the slave, and creates a response packet to send back to the master from the slave's response. When the command arrives, the agent asserts either the read signal or *write* signal to the slave, and drives the *address*, *writedata*, *byteenable*, and *burstcount* signals with data from the command packet. As soon as *waitrequest* is deasserted by the slave, the agent asserts the command interface's *ready* signal, indicating that the cycle has been accepted.

In order to create the response packet, the implementation of the Slave Agent includes two FIFO structures. One FIFO is used to store the command as it arrives from the command interface. As described in section 3.4.1, pipelined and bursting slaves do not respond with data while they are accepting the command, but instead respond with data a couple of cycles later with the assertion of the *readdatavalid* signal. Therefore a FIFO is needed to store the portion of the command that gets returned in the response packet.

The second FIFO is used to store response data, if needed. Unlike Avalon MM, Merlin does not guarantee that the response channel is available to receive read data any time the slave has read data to send. The response packets must go through arbitration for the master just like the master's command went through arbitration for the slave. To accommodate this, a FIFO is used to hold the response data when the response packet interface is not ready. The response packet is thus a combination of data from the two FIFOs, and is not valid until there is valid data in both.

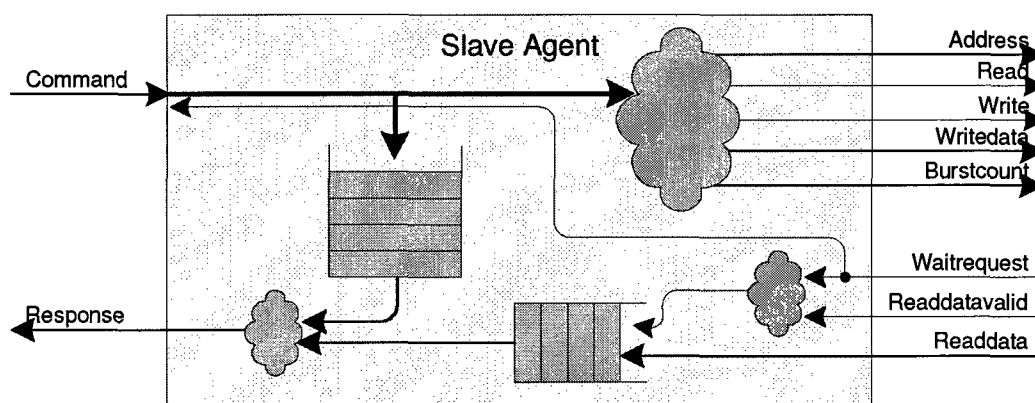


Figure 43 - Slave Agent

4.4 Component Verification

Before using any Merlin components in a functional system, they need to be verified in simulation. Since many Merlin components are highly parameterizable, a flexible test mechanism was developed. Unlike traditional ASIC or FPGA component verification, the verification coverage of parameterizable components exists in two dimensions: the parameterization dimension and the functionality dimension. To completely test any given component, the parameterization space must be covered by selecting appropriate

parameterizations to test, and for each parameterization, the functionality space must be covered in simulation.

To accommodate this, parameterizable Avalon MM Bus Functional Models (BFMs) were developed. The parameterization for both the master and slave BFM includes *data* width, *address* width, *burstcount* width, pipelined support, and burst support. The master BFM gives the user the ability to issue any Avalon MM command, including invalid ones, and to collect the response, including the ability to count the number of wait states and the time taken for pipelined *readdata* responses.

Once the BFMs were available and tested, a framework that called the Modelsim HDL simulator from Java was developed, allowing the JUnit framework to be used to handle the complexity of these highly parameterizable components. Each JUnit test run tests a single variation of the component under test. Since the component is parameterizable, the testbench must be generated to suit the component, and the BFMs must be parameterized appropriately as well. For example, to test the twin fabric component, a source BFM needs to be instantiated for every input, a sink BFM needs to be instantiated for every output, and the input and output BFMs have to be configured to match the fabrics ports. Each JUnit test generates the component and its test bench, and calls the Modelsim simulator to run the tests.

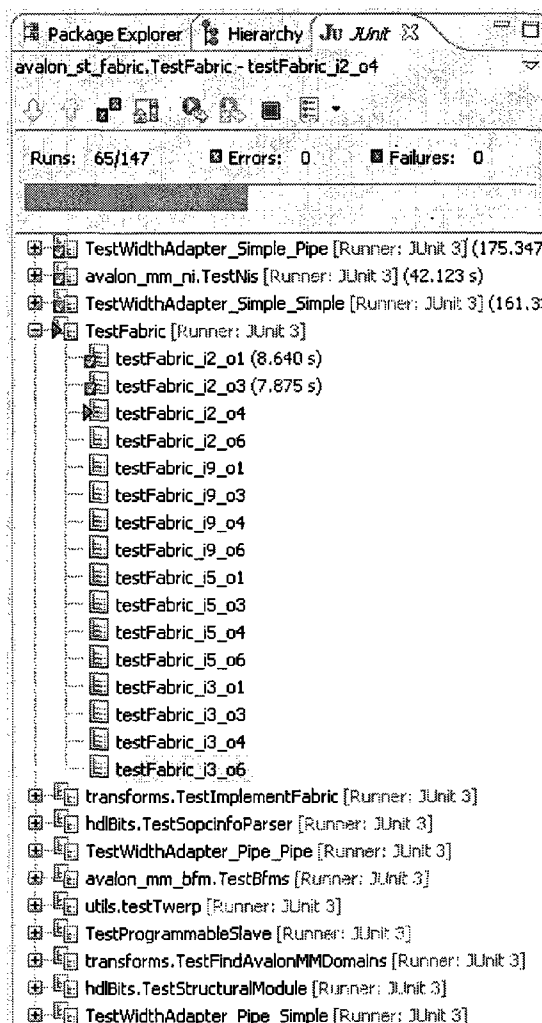
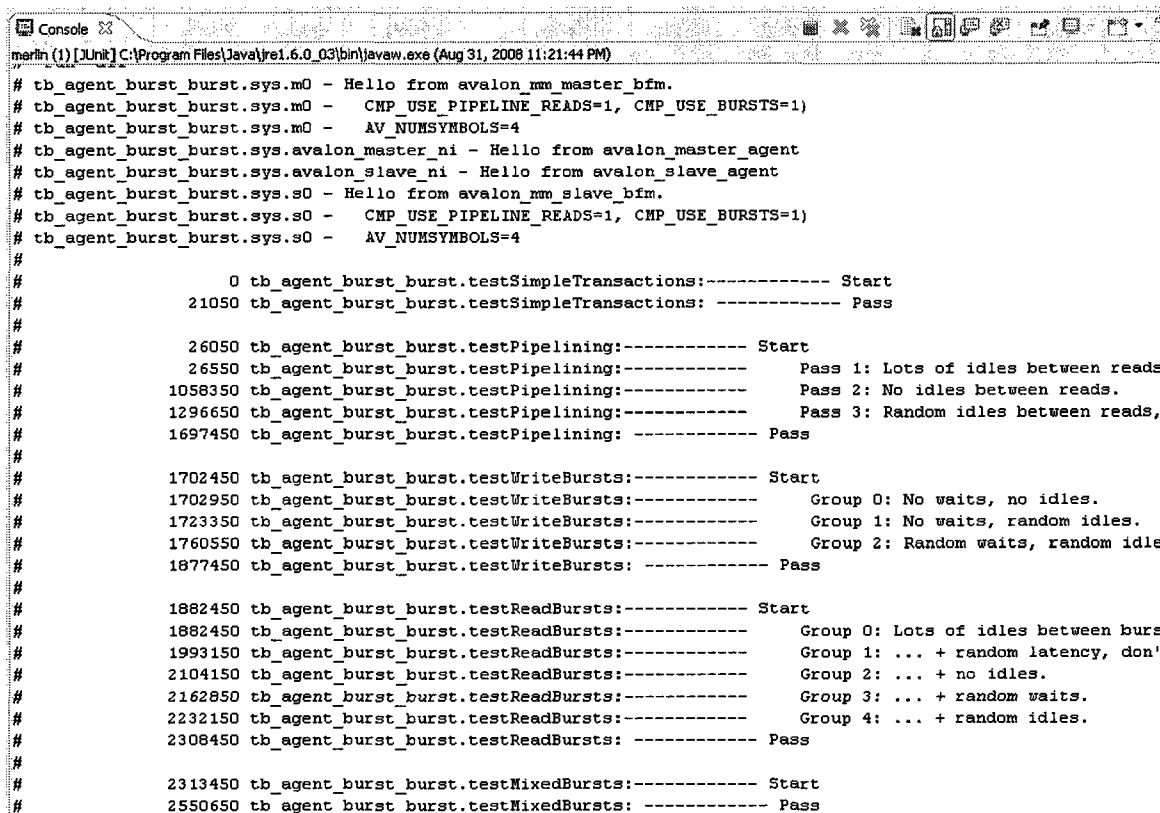


Figure 44 - JUnit Simulation Test Run

The tests run by the simulator were constructed to be the SystemVerilog equivalent of JUnit tests: each simulation run would run a number of discrete tests, each of which tested some function of the component, and would return pass or fail. In most cases, the tests involved a degree of randomness in the generation of test vectors. In the case of the cross connect, for example, the BFM at the each input would create random packets, and the testbench would ensure that the data was received correctly by the output BFM. Figure 44 shows an in-progress JUnit / Modelsim Test run, and Figure 45 shows the

resulting output. Each of the items on the left in Figure 44 is one of the variations being tested.



```

marlin (1) [JUnit] C:\Program Files\Java\jre1.6.0_03\bin\javaw.exe (Aug 31, 2008 11:21:44 PM)

# tb_agent_burst_burst.sys.m0 - Hello from avalon_mm master_bfm.
# tb_agent_burst_burst.sys.m0 - CMP_USE_PIPELINE_READS=1, CMP_USE_BURSTS=1)
# tb_agent_burst_burst.sys.m0 - AV_NUMSYMBOLS=4
# tb_agent_burst_burst.sys.avalon_master_ni - Hello from avalon_master_agent
# tb_agent_burst_burst.sys.avalon_slave_ni - Hello from avalon_slave_agent
# tb_agent_burst_burst.sys.s0 - Hello from avalon_mm slave_bfm.
# tb_agent_burst_burst.sys.s0 - CMP_USE_PIPELINE_READS=1, CMP_USE_BURSTS=1)
# tb_agent_burst_burst.sys.s0 - AV_NUMSYMBOLS=4
#
#           0 tb_agent_burst_burst.testSimpleTransactions:----- Start
#       21050 tb_agent_burst_burst.testSimpleTransactions: ----- Pass
#
#           26050 tb_agent_burst_burst.testPipelining:----- Start
#       26550 tb_agent_burst_burst.testPipelining:-----      Pass 1: Lots of idles between reads
#      1058350 tb_agent_burst_burst.testPipelining:-----      Pass 2: No idles between reads.
#      1296650 tb_agent_burst_burst.testPipelining:-----      Pass 3: Random idles between reads,
#      1697450 tb_agent_burst_burst.testPipelining: ----- Pass
#
#       1702450 tb_agent_burst_burst.testWriteBursts:----- Start
#       1702950 tb_agent_burst_burst.testWriteBursts:-----      Group 0: No waits, no idles.
#       1723350 tb_agent_burst_burst.testWriteBursts:-----      Group 1: No waits, random idles.
#       1760550 tb_agent_burst_burst.testWriteBursts:-----      Group 2: Random waits, random idle
#       1877450 tb_agent_burst_burst.testWriteBursts: ----- Pass
#
#       1882450 tb_agent_burst_burst.testReadBursts:----- Start
#       1882450 tb_agent_burst_burst.testReadBursts:-----      Group 0: Lots of idles between burs
#       1993150 tb_agent_burst_burst.testReadBursts:-----      Group 1: ... + random latency, don'
#       2104150 tb_agent_burst_burst.testReadBursts:-----      Group 2: ... + no idles.
#       2162850 tb_agent_burst_burst.testReadBursts:-----      Group 3: ... + random waits.
#       2232150 tb_agent_burst_burst.testReadBursts:-----      Group 4: ... + random idles.
#       2308450 tb_agent_burst_burst.testReadBursts: ----- Pass
#
#       2313450 tb_agent_burst_burst.testMixedBursts:----- Start
#       2550650 tb_agent_burst_burst.testMixedBursts: ----- Pass

```

Figure 45 - JUnit Simulation Output

4.4.1 Verification Coverage

The following is a brief discussion of the tests that were created to verify that merlin components operated correctly.

BFM Verification - To test the Bus Functional Models for functional correctness, a set of three BFM tests were created, one for burst capable BFMs, pipeline capable BFMs, and simple BFMs. For each set, the master BFM was directly connected to a slave BFM. The test consisted of subtests using randomly generated transactions to cover the

following things: simple (non-bursting) transactions, pipelining transactions with a random number of wait states and idle responses, and mixed burst transactions with random sizes, a random number of idle cycles between commands, and random wait request cycles and response latency.

Fabric Verification - To verify that the twin fabric was operating correctly, a behavioral model of both the arbiter component and the fabric was developed along with a simulation test bench. The test bench has a parameterizable number of input and output ports, as well as a parameterizable width. It instantiates the fabric being tested alongside the behavioral model, and injects random packets into every interface in parallel. To verify correctness, the test bench verifies that the output on every port of the fabric being tested matches the behavioral model on every clock cycle. The test suite varies the number of inputs and number of outputs between 1 port and 32 ports, to get coverage over the large parameterization space.

Network Interface Verification - To verify that the network interfaces are functionally correct, a testbench was created that tested a master NI and slave NI together. The master BFM drives random transactions into the master network interface, sends packets out on its command packet interface. The command packet interface is connected to the slave NI, which in turn drives transactions to the slave BFM, which checks them and provides responses. The responses are converted into response packets by the slave NI, and are sent to the master NI, which responds to the master BFM, which checks that the responses match the expected responses. Nine individual test runs are required to test

that all combinations of simple, pipelined, and burst master and slave network interfaces are verified.

Width Adapter - To verify the width adapter components, the width adapter test bench includes a master BFM and it's NI at one data width connected to a slave BFM and it's NI at a different width, with width adapters connected between them for both the command and response paths. For each of the nine combinations of master & slave interface types described above, the master and slave widths were each allowed to be every power of 2 value between 8 bits and 64 bits, for a total of 225 test runs. Each run included a random set of bursting and non-bursting transactions, as before.

4.5 Transform Implementation and Verification

Once all of the components are available, a software algorithm must be able to take a high-level description of a system with the desired connectivity, and transform that into a realized system that consists only of connected components.

To accomplish this transformation, a pair of cooperating transform algorithms were developed. Each transform takes as its input a Structural Module object as described in section 4.2.3 that represents desired connectivity, and returns a collection of Structural Modules that are further along the way to a final implementation.

4.5.1 Find Avalon MM Domains Transform

The role of the first transform, the “Find Avalon MM Domains” transform, is to divide the system construction problem into a number of domains, and to define what needs to be done by each domain.

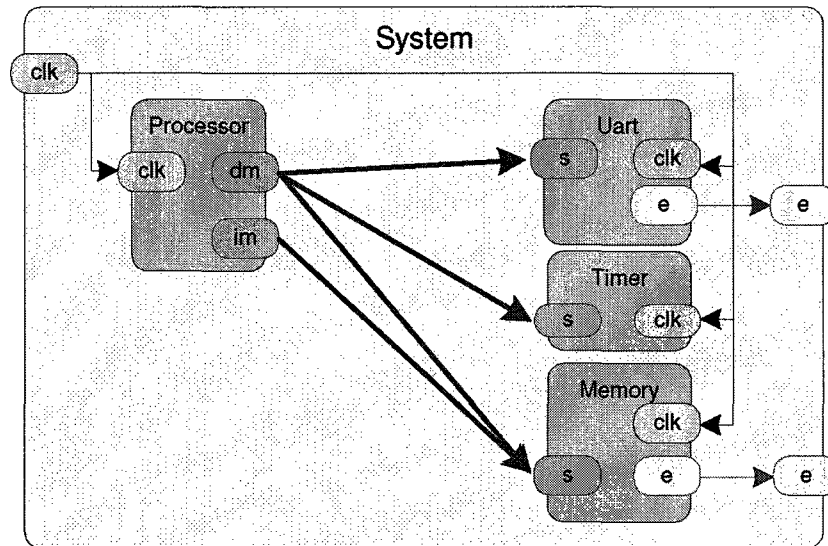


Figure 46 - Pre-transformation system

To illustrate this transformation, consider the simple system presented in Figure 46. This system consists of an on-chip processor with a clock input, a data master (dm), and an instruction master (im) interface. There is also on the same system a Universal Asynchronous Receiver Transmitter (UART) component, a timer component, and a memory controller that interfaces to off-chip memory. All three of these components have a clock input and a single slave interface. The UART and memory component also has an export (e) interface to export some signals to physical pins on the device. This is implemented in software as a Structural Module object, with an Instance object for each of the components in the system. The connections shown from the data master to all slaves, and from the instruction master to the memory slave are logical connections expressing the desire of the system designer; they are not yet physical connections that can be realized as wires on the FPGA.

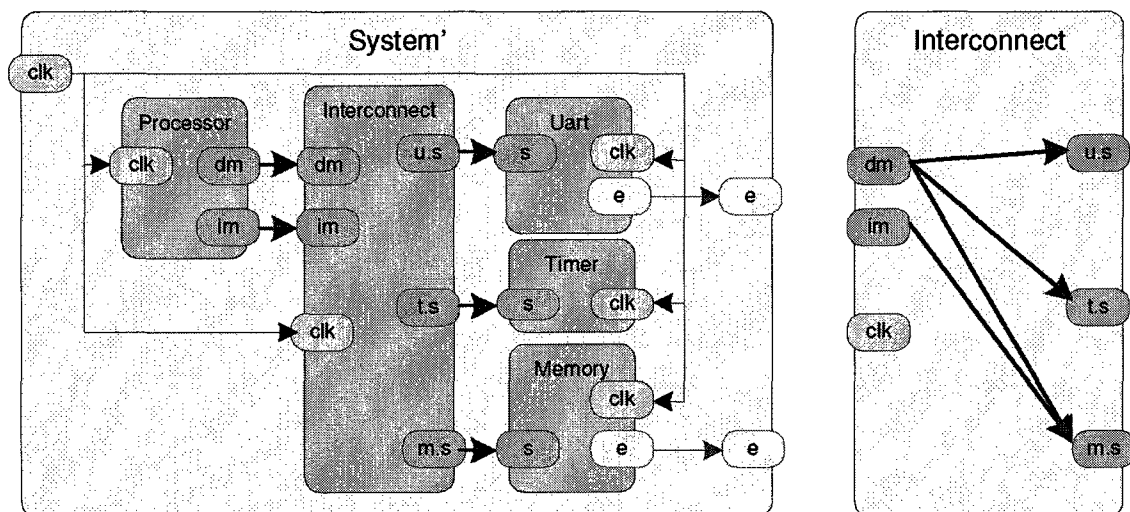


Figure 47 - Result of Find Avalon MM Domains Transform

Figure 47 illustrates the result of running the Find Avalon MM Domains transform. This transform returns two new Structural Module systems. The first one, labeled *system'* ("system prime"), is the new top-level system with the high-level point-to-multipoint connections replaced with simple point-to-point connections and an interconnect. Each of the master and slave interfaces has a connection to a newly created component in the center, called the *interconnect*.

At this stage, the interconnect component includes only the interfaces on its periphery and the connections between them; the components used to realize this connectivity have yet to be determined. The connections between the interfaces represent, at this point, the desired connectivity to be supported by the interconnect block. Unlike the *system'* Structural Module, which can now be rendered as SystemVerilog, the interconnect component has to go through a second transformation before it is complete.

4.5.1.1 Implement Fabric From Empty SM Transform

The second transformation accepts as input the interconnect component from the previous transform. Using the interfaces around the periphery of the component and the connections between them as a statement of the desired connectivity, it returns the interconnect' ("interconnect prime") Structural Module shown in Figure 48. The components used in this interconnect are themselves Structural Modules consisting of the building blocks described previously. Once this transformation is complete, the entire system can be rendered to SystemVerilog, and simulated or compiled into an FPGA.

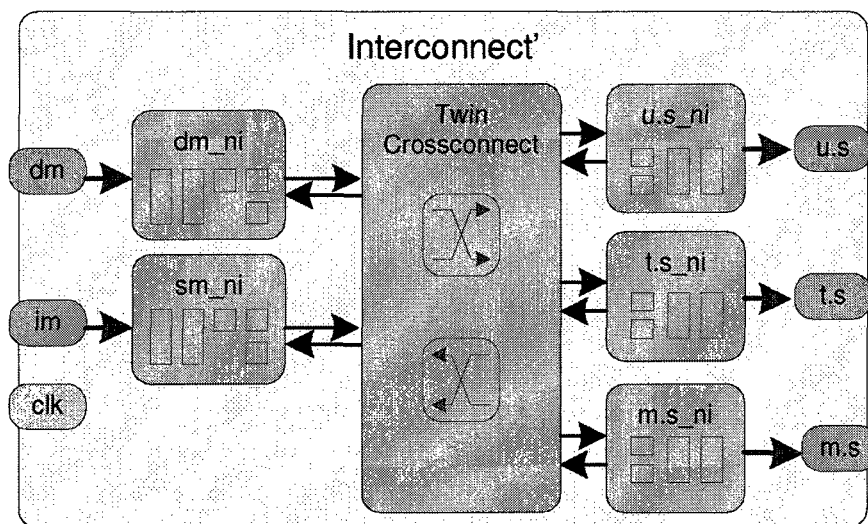


Figure 48 - Result of Implement Fabric From Empty SM Transform

4.6 Simple Processor Test System

To demonstrate the prototype Merlin implementation working in a real system, small microprocessor-based system was developed and tested with simple software. As shown in Figure 49, this system consists of a Nios II soft core microprocessor, a PLL, a JTAG

UART, and an On-chip RAM component. A description of each of these components follows.

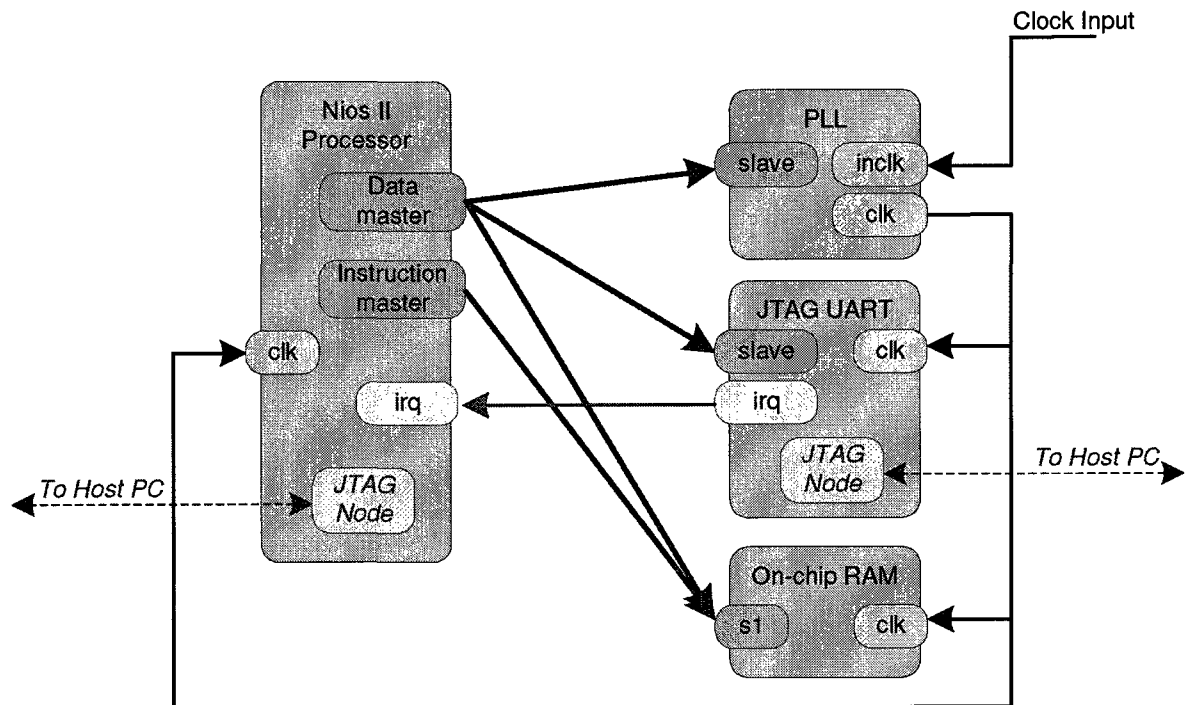


Figure 49 - Simple Processor System

Nios II Processor – The Nios II Processor is a general purpose 32-bit RISC processor core with a full 32-bit instruction set, data path, and address space [54]. As shown, the Nios II processor includes an internal JTAG node, allowing a host PC to control and debug the processor without using FPGA logic.

PLL – The Phase Locked Loop (PLL) component generates an output clock with a controllable phase and frequency relative to an input clock. The PLL component includes an Avalon slave interface to support run-time configuration of the phase and frequency by allowing the setting of phase and counter settings.

JTAG UART – The JTAG UART is a component that is used to send and receive data via the on-chip JTAG mechanism instead of via the serial connection typical of most UART devices. This allows the Host PC to send and receive data from the device under test as though it were using a UART and Serial port. The interface and register set presented to the internal system logic are the same as a traditional UART component. All processor communication in this system uses the JTAG UART.

On-Chip RAM – The on-chip RAM component instantiates one or more of the RAM blocks present inside the FPGA, and presents an Avalon Slave Interface for reading and writing the memory. In this system, the on chip RAM is used to store all of the processor's program code, as well as data storage.

4.6.1 Implementation

The system was captured using Altera's SOPC Builder design tool. A screenshot of the system as captured in SOPC Builder is presented in Figure 50. When a system is generated using the SOPC Builder tool, the '.sopcinfo' XML report file describing the system is written to disk in addition to the generated HDL. To generate this system's interconnect using Merlin, a Java class was implemented to read this sopcinfo file and convert it into a Structural Module so that it could be passed to the Merlin Transform algorithms described in section 4.4.

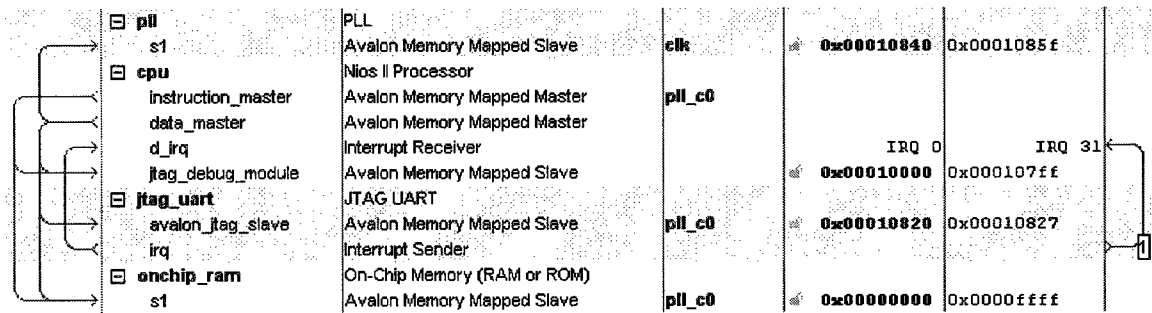


Figure 50 - Simple Processor System in SOPC Builder.

Once Merlin had run, and the HDL files were written to disk, the system was compiled into an Altera Stratix 1S10 device with a C6 speed grade, on a Nios II 1S10 development board. Figure 51 shows a photograph of the development board used for this test.

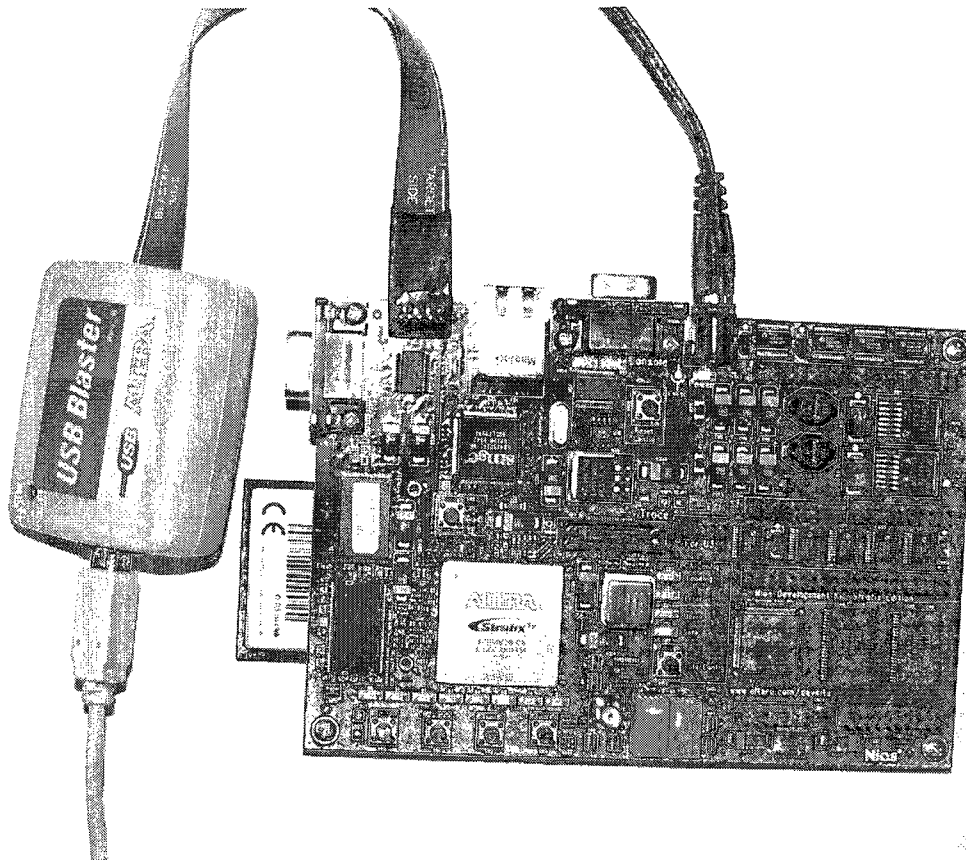


Figure 51 - Photograph of Simple System Test Board

Analysis & Synthesis Resource Utilization by Entity				
	Compilation Hierarchy Node	Logic Cells	LC Registers	Memory Bits
1	[-] sys	4159 (20)	1831	591664
2	[-] cpu:cpu	2202 (1819)	1074	65664
40	[-] high_res_timer:high_res_timer	89 (89)	70	0
41	[-] interconnect:interconnect	1485 (0)	465	688
42	[-] cpu_data_master_ni:cpu_data_master_ni	20 (0)	1	0
43	[-] lavalon_master_agent:cpu_data_master_ni_agent	15 (15)	1	0
44	[-] cpu_data_master_ni_decoder:cpu_data_master_ni_decoder	5 (5)	0	0
45	[-] cpu_instruction_master_ni:cpu_instruction_master_ni	4 (0)	0	0
48	[-] cpu_itag_debug_module_ni:cpu_itag_debug_module_ni	317 (0)	122	256
55	[-] high_res_timer_s1_ni:high_res_timer_s1_ni	98 (0)	47	0
60	[-] interconnect_fabric:interconnect_fabric	398 (0)	41	0
61	[-] interconnect_fabric_cmd:command	255 (202)	25	0
67	[-] interconnect_fabric_rsp:response	143 (91)	16	0
70	[-] itag_uart_avalon_itag_slave_ni:itag_uart_avalon_itag_slave_ni	280 (0)	106	176
76	[-] onchip_ram_s1_ni:onchip_ram_s1_ni	318 (0)	122	256
77	[-] lavalon_slave_agent:onchip_ram_s1_ni_agent	311 (75)	116	256
82	[-] onchip_ram_s1_ni_trans:onchip_ram_s1_ni_trans	7 (7)	6	0
83	[-] pll_s1_ni:pll_s1_ni	50 (0)	26	0
84	[-] lavalon_slave_agent:pll_s1_ni_agent	10 (5)	4	0
85	[-] lavalon_slave_agent_fifo:command_fifo	5 (5)	4	0
86	[-] pll_s1_ni_trans:pll_s1_ni_trans	3 (3)	3	0
87	[-] width_adapter:pll_s1_ni_cmdwal	37 (37)	19	0
88	[-] itag_uart:itag_uart	191 (54)	108	1024
110	[-] onchip_ram:onchip_ram	32 (32)	32	524288
113	[-] pll:pll	0 (0)	0	0
116	[-] sld_hub:sld_hub_inst	140 (38)	82	0

Figure 52 - Simple Processor System Resource Utilization

Figure 52 shows the hierarchy and post-synthesis resources utilization after synthesizing in the Quartus II tool. As shown in the figure, the entire system was 4159 Logic Cells, and 591664 memory bits. Of this, the Merlin interconnect consumed 1485 logic cells (36%), and 688 memory bits (0.1%). The interconnect itself include the command and response fabric, which combined were responsible for 398 Logic Cells, 27% of the total resources used by Merlin. The remaining logic cells and all of the interconnect memory bits were consumed by the network interfaces.

4.6.2 Test Program

To test the functional correctness of this system, we constructed a test that verified that communication between the CPU's instruction and data masters, the on chip memory, and the JTAG UART component was working correctly. The test itself consists of three sub-tests that test that the memory is working correctly, and it prints the results to the console, verifying the operation of the JTAG UART component. The three sub-tests are as follows:

Data Bus Test - In a single memory location, this test tests each bit of the data bus one at a time, setting it to a one, and then to a zero.

Address Bus Test - For each address bit in the address range of interest, a pattern is written to memory with the address bit set, and a different pattern is written to memory with it cleared. Both locations are then checked for correctness.

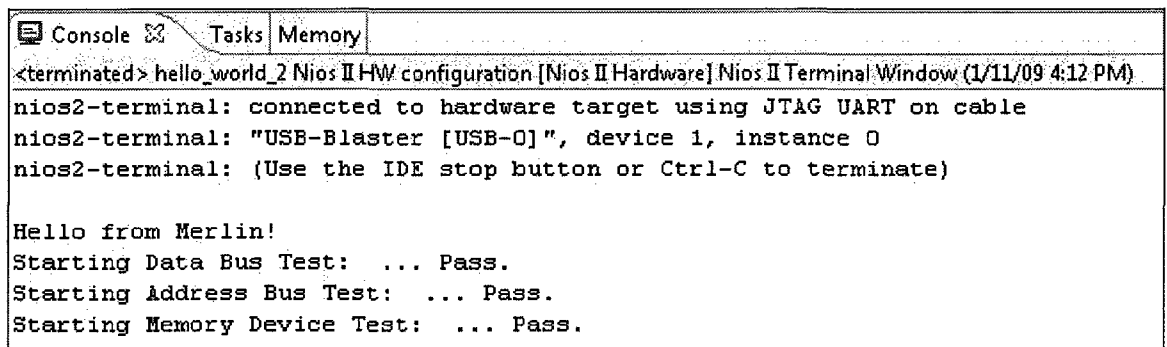
Memory Contents Test - To verify that the memory itself is working correct, we verify that each location in memory is capable of holding both a zero and a one. To do this, for each location in memory, a value derived from the address and its inverse is written to memory and verified.

4.6.3 Test Flow

The functional test consists of a very simple test running on the processor that prints a message to the JTAG UART. The host PC is connected to the development board's

JTAG chain using an Altera USB Blaster, and the Nios II IDE is used to download and run the software. The entire test flow consists of the following steps:

1. Starting with the .sopcinfo file output from SOPC Builder, generate the Merlin top level and interconnect files.
2. Compile the design to a FPGA bitstream using the Quartus II software.
3. Download the bitstream from the Quartus II Programmer to the FPGA across the JTAG Chain using the USB Blaster. Once this is complete, the FPGA starts up automatically, and the system with the processor now exists in soft logic.
4. Compile the test Nios II software in the Nios II IDE.
5. By controlling the soft processor via the JTAG chain, download the compiled software into the On-chip memory.
6. Start the processor. The processor begins executing at the reset vector address, which is in the on-chip memory. After executing the initialization code, the processor executes the test. The result of the test is written to the console via the JTAG UART device.
7. Observe on the Nios II IDE console that the software has been downloaded to the processor, that the processor has started executing, and that all of the tests have passed.



```

<terminated> hello_world_2 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (1/11/09 4:12 PM)
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from Merlin!
Starting Data Bus Test: ... Pass.
Starting Address Bus Test: ... Pass.
Starting Memory Device Test: ... Pass.
  
```

Figure 53 - Console output from Simple Processor System Test Program

4.7 Measurements & Results

In this section, we present some performance and resource utilization measurements, such as the frequency of the network as a function of number of masters and slaves, and the effect of different network interface options on the achievable frequency. We also consider the effect on logic resources of varying the number of masters & slaves. Finally, we use the performance results to discuss some tradeoffs of different types of network topologies. Note that all of the measurements in this section are using an Altera Stratix III 3S50, in a C2 speed grade.

4.7.1 Crossbar Switch Performance Measurement

For this series of tests, we varied the number of inputs and outputs supported by an crossbar switch between 1 and 32. The crossbar itself supports a combinatorial data path, so for the purposes of performance measurement, it was instantiated within a wrapper module that added to register stages to every input and output signal. In addition, every signal other than clock and reset was marked as a virtual pin, so that it was not routed to a physical pin on the device. In all cases, physical synthesis, register re-timing, and non-standard optimizations were disabled in Quartus. These optimizations typically provide an additional 10 to 15% performance gain when enabled. When the same experiment was done with 4 data bits instead of 32, the operating frequency was not significantly affected.

Figure 54 and Figure 55 show the frequency of the crossbar switch when varying the number of inputs and outputs. Each line shown Figure 54 represents a fixed number of inputs, as the number of outputs are varied. As shown in the figure, as the number of

outputs increases, the frequency that can be achieved by the switch drops, but not sharply. This is to be expected, since increasing the number of outputs only replicates logic and increases the fan-out of data signals.

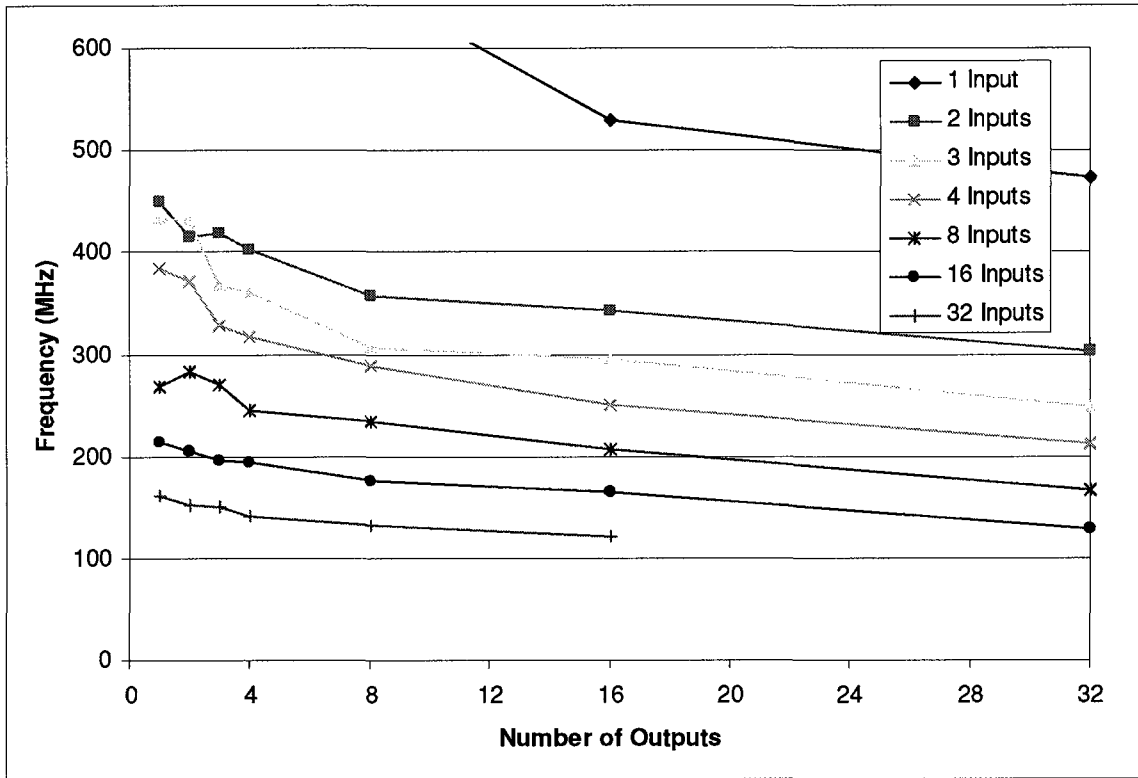


Figure 54 - Crossbar Switch Frequency as a function of the Number of Outputs.

Figure 55 shows the same data as Figure 54, but arranged such that each line represents a constant number of outputs with a varying number of inputs. Here, we see a sharp decrease in the system frequency as the number of inputs increases. This is also to be expected, since as the number of inputs increases, the arbitration logic for each output must consider more requesters, and all of the data path multiplexing becomes deeper. In this device family, a operating frequency of 200 MHz can be achieved for any number of outputs, as long as the number of inputs per crossbar switch is kept to 4 or less.

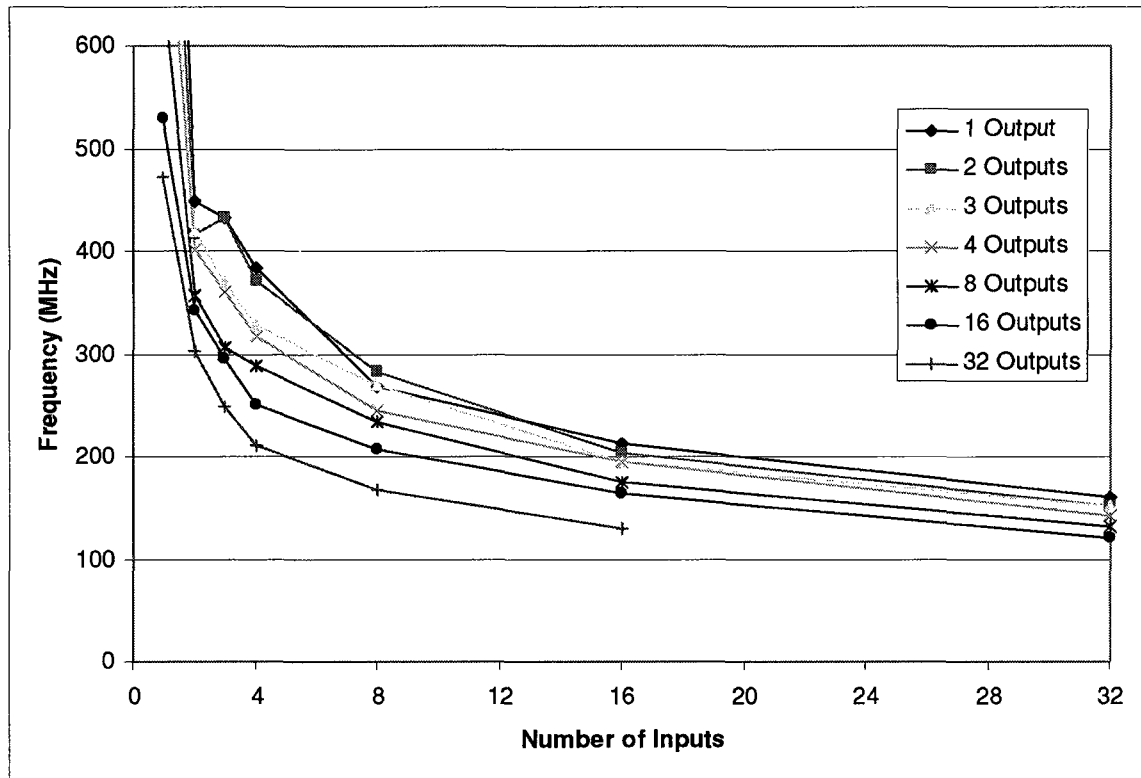


Figure 55 - Crossbar Switch Frequency as a function of the Number of Inputs.

Figure 56 and Figure 57 show the Look-up-table (LUT) and Register utilization for the cross-connect as a function of the number of inputs and number of outputs.

		# Outputs						
		1	2	3	4	8	16	32
# Inputs	1	4	7	10	15	28	59	133
	2	21	40	60	83	160	327	682
	3	96	193	294	405	859	1990	5070
	4	106	211	319	442	934	2146	5394
	8	144	277	420	585	1205	2704	6636
	16	323	634	960	1330	2703	6132	15236
	32	657	1283	1941	2702	5653	12864	32000

Figure 56 - Crossbar Switch LUT Utilization (LUTs).

		# Outputs						
		1	2	3	4	8	16	32
# Inputs	1	2	4	6	8	16	32	64
	2	5	10	16	20	40	80	160
	3	7	14	21	28	56	113	225
	4	9	19	27	36	72	146	288
	8	17	35	51	68	137	275	553
	16	33	66	103	133	264	531	1057
	32	65	131	196	263	520	1052	2080

Figure 57 - Crossbar Switch Register Utilization (Registers).

4.7.2 Network Interface Performance Measurement

In this section, we consider the effect of a number of different parameters on the master and slave network interfaces. Both the master and slave network interfaces support three modes, called *burst*, *pipelined*, and *simple*, which are controlled through synthesis time parameters. The pipeline mode is used to connect to masters or slaves that support pipelined reads, but not burst transactions, and the simple mode is for master and slave interfaces that support neither pipeline reads nor bursts. The burst mode network interface is used to connect to a master or slave that supports bursting. The bursting network interface requires the least logic, it does not need to terminate burst transactions on behalf of the slave; the slave is able to terminate the bursts itself. As with the previous section, all of the results shown here are using a Stratix III 3S50 device, in a C2 speed grade, and all interfaces are have 32 bits of data and 8 bits of address.

Figure 58 shows the frequency and resource utilization of a Master network interface in each of its three synthesizable modes. Although static timing analysis gives us the numbers shown in the figure, the clock frequency on this family of FPGAs is actually

limited to 400 MHz. The numbers in Figure 58 are thus useful when comparing logic implementation & performance, but do not represent what is really achievable on an FPGA. With the Merlin packet format described in 3.4.2, the Master network interface has very little work to do to convert transactions into command packets, and response packets back into transaction responses, allows it to achieve such as high frequency.

<i>Mode</i>	<i>Freq. (MHz)</i>	<i>LUTS</i>	<i>Registers</i>
<i>Simple</i>	1050	5	1
<i>Pipelined</i>	1126	2	0
<i>Burst</i>	656.6	44	20

Figure 58 - Master Network Interface Frequency & Logic Utilization

<i>Mode</i>	<i>Freq. (MHz)</i>	<i>LUTS</i>	<i>Registers</i>
<i>Simple</i>	299.94	39	91
<i>Pipelined</i>	323.31	39	91
<i>Burst</i>	331.56	42	91
<i>Simple (0 Cycle)</i>	138.06	116	91
<i>Pipelined (0 Cycle)</i>	163.19	115	91
<i>Burst (0 Cycle)</i>	183.25	111	91

Figure 59 - Slave Network Interface Frequency & Logic Utilization

In addition to the three modes of operation, Slave network interfaces also optionally support zero cycle responses. With this mode of operation, the Slave network interface is able to return a response packet on the same cycle it receives the associated command packet. In order to do this, it adds logic that bypasses the command and response FIFOs depending on the instantaneous response timing of the slave. This has the effect of joining the path through the command network and the path through the response network into one long combinatorial path, which has a significant effect on the

performance. Figure 59 shows the frequency and resource utilization of a Slave network interface for all three modes of operation, with and without the zero cycle support option enabled.

4.7.3 Comparison to SOPC Builder Generated Interconnect

Here, we compare generated SOPC Builder interconnect instances to Merlin interconnect instances. In all cases, SOPC Builder's interconnect is combinatorial, meaning that it supports read transactions where the read data can be returned to the master the cycle after it was issued. With Merlin, two different implementations are considered. With the first implementation, we place a register boundary between the master network interface and the packet network, and between the packet network and the slave network interface, leading to a read latency of 5 cycles. With the second, there are no registers, the path is purely combinatorial, and the read latency is one cycle, just like the SOPC Builder interconnect. The size and frequency of the master network interface reported in Figure 58 does not include the master's address decoder, which is responsible for determining which slave a given transaction goes to. To determine the size of the address decoder, a small design containing only the address decoder was created. The combinatorial logic for the address decoder was found to be so small that it was implemented 'for free' in the I/O blocks of the design.

To determine the size of the Merlin solution, the resources for the appropriate number of master and slave network interfaces are added to the resources for the appropriately sized crossbar switch fabric, as reported in the previous sections. The frequency of the register boundary Merlin switch is determined by taking the worst frequency of all of the

components independently. The frequency for the non-registered version is determined by adding the delays of all of the components in series. The size and frequency of each solution is shown in Figure 60.

<i>Size</i>	<i>Interconnect</i>	<i>Freq. (MHz)</i>	<i>LUTS</i>
2x2	SOPC Builder	322	182
	Merlin, Registered	300	212
	Merlin, Non-Registered	137	
3x3	SOPC Builder	230	359
	Merlin, Registered	300	552
	Merlin, Non-Registered	132	
4x4	SOPC Builder	199	741
	Merlin, Registered	300	786
	Merlin, Non-Registered	125	
8x8	SOPC Builder	138	3230
	Merlin, Registered	233	1893
	Merlin, Non-Registered	109	
16x16	SOPC Builder	91	11838
	Merlin, Registered	165	7508
	Merlin, Non-Registered	92	

Figure 60 - Slave Network Interface Frequency & Logic Utilization

Figure 61 and Figure 62 compare the frequency and logic utilization of the Merlin and SOPC Builder interconnects for a number of interconnect sizes. We can see that for a very small 2x2 network, the SOPC Builder generated fabric is the fastest, but as soon as there are three masters and three slaves, the registered Merlin implementation has higher frequency. Up to and including a network with four masters and four slaves, the frequency of the Merlin interconnect is limited to 300 MHz by the bursting slave network interface, while beyond that, the number of terms feeding into the arbitration logic in the cross connect dominates, and the frequency drops. We can also see that the frequency of the SOPC Builder system falls off quickly as masters and slaves are added to the system, and that both the registered and non-registered Merlin implementations fall off at a

slower rate. Once the size of the network reaches 16 master and 16 slaves, even the non-registered Merlin interconnect can support a frequency comparable to the SOPC Builder interconnect. Using Merlin primitives, it is possible to keep the frequency high by building a larger system out of smaller cross-connect components with register boundaries.

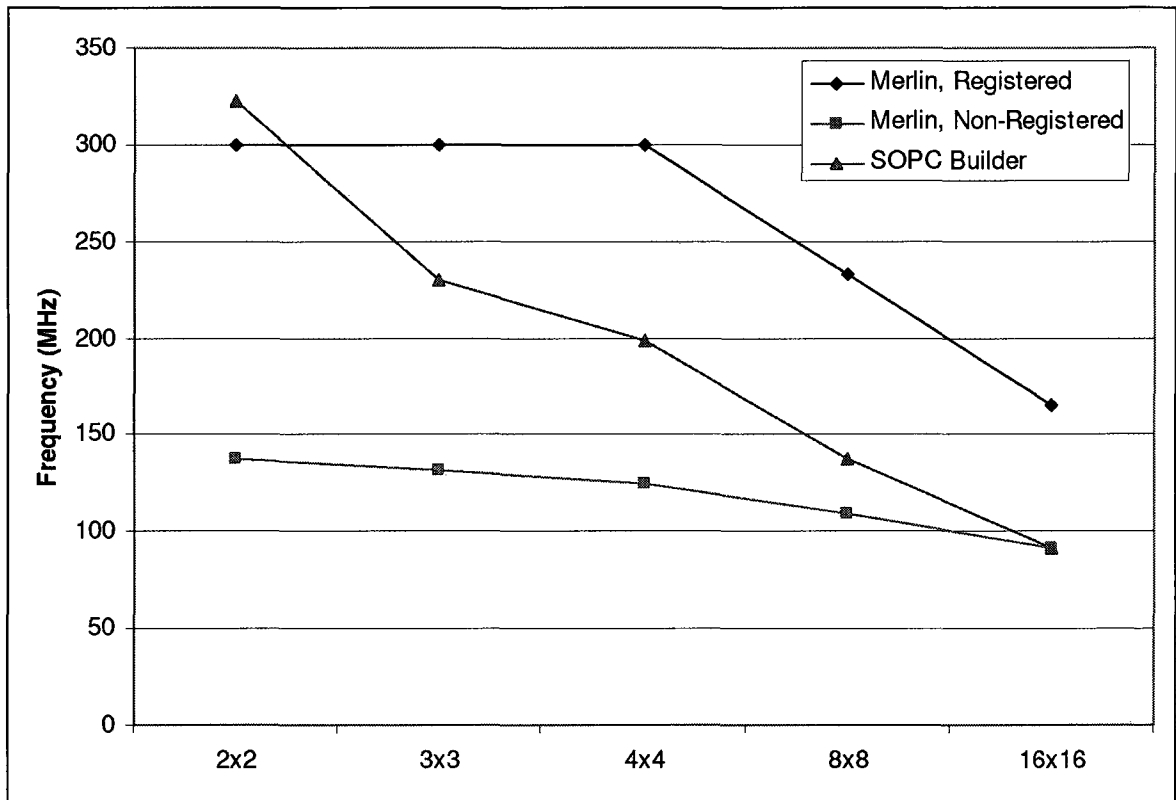


Figure 61 - Merlin vs. SOPC Builder Interconnect Frequency

Regarding logic utilization, we can see that for small network sizes, the SOPC Builder solution is the same or smaller than the equivalent Merlin interconnect, because the size of the Merlin solution is dominated by the network interfaces. Beyond a 4x4 network, however, Merlin becomes more efficient, using only 63% of the logic used by SOPC Builder at the 16 master, 16 slave node.

While SOPC Builder generated interconnect may be faster and smaller when a system contains a small number of masters and slaves, the network approach provides faster and smaller systems as the system size grows, even if the underlying packet network supports full concurrency with a single stage cross-bar interconnect. Merlin primitives could also be used to provide multi-stage or non fully concurrent networks, increasing performance and reducing resources even further.

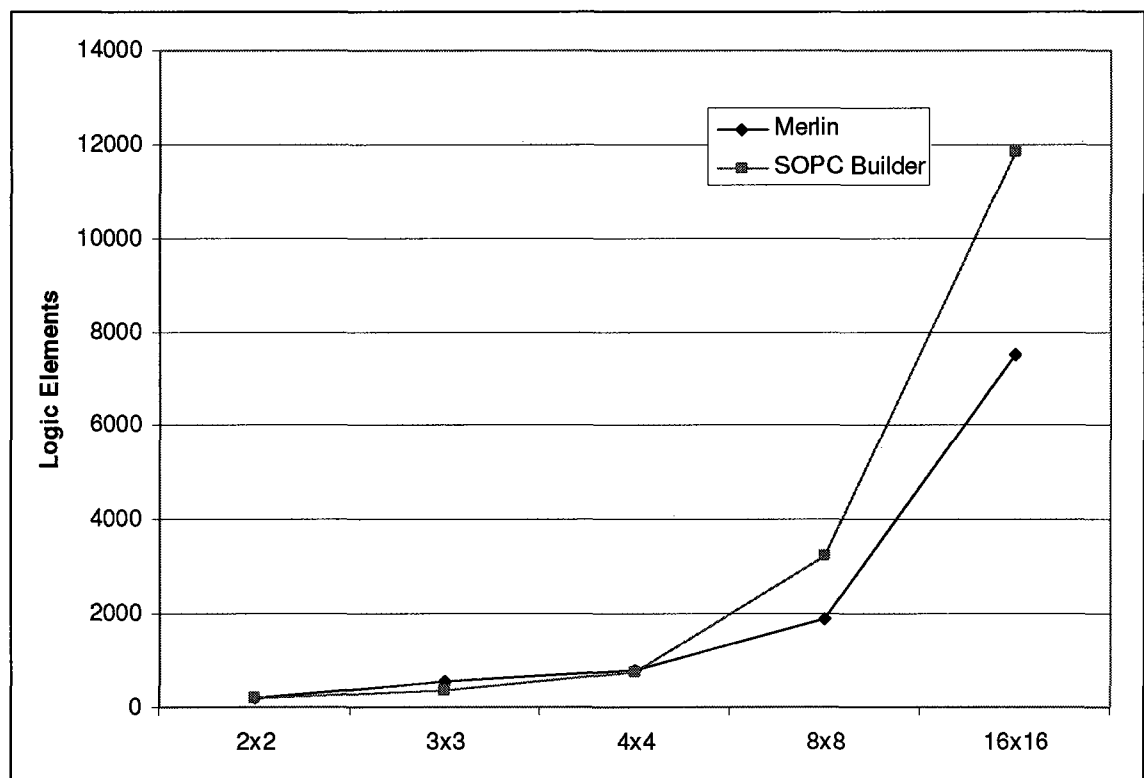


Figure 62 - Merlin vs. SOPC Builder Resource Utilization

4.7.4 Comparison to other Network On Chip Implementations

Here, we compare Merlin performance to that of other network-on-chip implementations.

Figure 63 shows how a 2 master, 2 slave and 4 master, 4 slave Merlin implementation

compares to other equivalent network on chip implementations. Keep in mind that because different NoC implementations use different device families and speed grades, frequency results are unreliable.

<i>Size (Mstrs, Slaves)</i>	<i>Source</i>	<i>Device Family</i>	<i>Datapath (bits)</i>	<i>Freq. (MHz)</i>	<i>Read Latency (Cycles)</i>	<i>LUTs</i>
2, 2	Aethereal [41] (2D Mesh)	Virtex II	32	18	≥ 5	3200
	Merlin (Registered)	Stratix III	32	300	5	212
	Merlin (Non-Registered)	Stratix III	32	137	1	212
4, 4	Nostrum [42] 2D Mesh	--	64	--	110	4945
	[43] (2D Mesh)	Virtex II	8	--	≥ 5	5652
	[44] (Cross-connect, no NI)	Virtex 4	36	272	6	780
	[32] (Unidirectional Mesh)	Virtex II	32	200	16	648
	Merlin (Registered)	Stratix III	32	300	5	786
	Merlin (Non-Registered)	Stratix III	32	125	1	786

Figure 63 - Comparison with other Network on Chip implementations

Figure 64 shows the resource utilization, in terms of Look Up Tables (LUTs) for a Merlin implementation vs. other NoC implementations. As shown, a Merlin interconnect with master and slave network interfaces has a lower resource utilization than any of the other implementations presented. The networks presented in [32] and [44] report comparable resource utilization, but they include only the packet network, and do not include the network interface components that perform the command and response packet encapsulation.

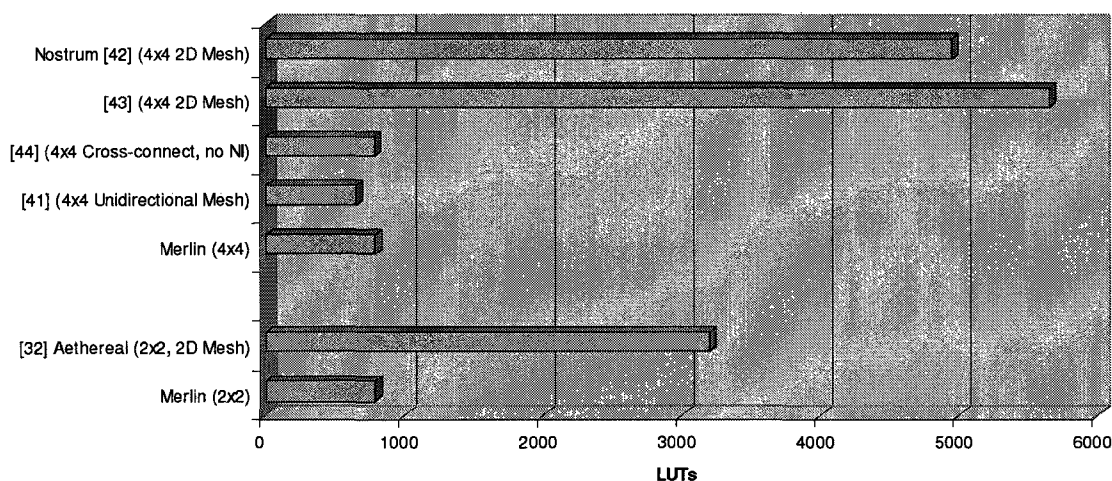


Figure 64 - Resource Utilization Comparison

There are a number of reasons why the Merlin implementation is smaller than other FPGA NoC implementations. Probably the most important is the fact that Merlin is intended to be a very light-weight, simple implementation, and targets FPGAs. The Nostrum interconnect presented in [42], for example, includes 7 finite state machines and 8 FIFOs at every node within the 2D mesh. The implementations presented in [43] and [44] also have expensive logic such as input FIFOs at every node. The Aethereal network presented in [41] was an NoC implementation targeted at ASICs, but prototyped on an FPGA. The resource utilization again reflects the fact that FPGA appropriate cost constraints were not kept in mind in the development of the network.

Another factor that reduces the size of the Merlin implementation with respect to other NoC implementations is that Merlin uses separate packet networks for commands and responses. Typical NoC topologies such as a 2D mesh or torus allow any node in the network to send a packet to any other node in the network. By considering that slaves

never send commands to masters, and that masters never send responses to slaves, the required connectivity can be significantly reduced. This connectivity reduction more than makes up for the fact that there are now two different networks, and the effect of this resources savings is amplified as the number of masters and slaves in the system grows. Separating the command and response networks has the additional benefit of preventing deadlock, since each network is unidirectional in either the master-to-slave or slave-to-master direction. This means that Merlin does not require any logic for deadlock avoidance.

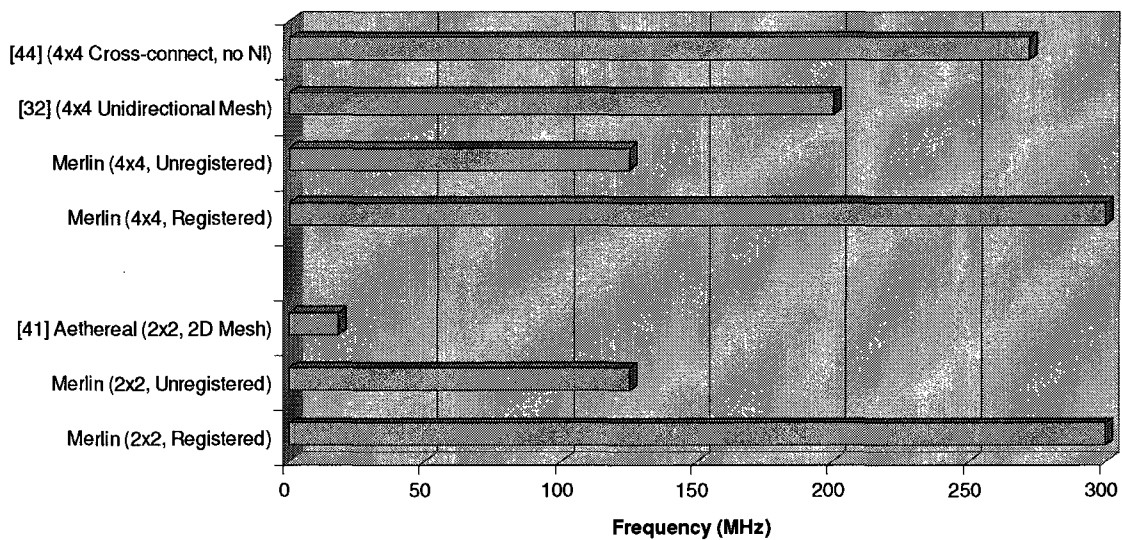


Figure 65 - Frequency Comparison

Figure 65 shows frequency of 2 master, 2 slave and 4 master, 4 slave registered and unregistered Merlin networks compared to other FPGA NoC implementations. As shown, the unregistered Merlin implementations can operate at less than half the frequency of the registered implementations. In the unregistered case, the logic for the

master and slave network interfaces is in the same combinational path as the packet network, allowing commands and responses to be transferred in one cycle each, but at the expense of frequency. In the registered case, there is a register stage between the master network interface and the packet network, and between the packet network and the slave interfaces. It happens that frequency for both the 2x2 and 4x4 systems is limited by the network interface, meaning that the network alone could achieve an even higher frequency.

Although the registered Merlin implementation achieved a higher frequency than any of the other NoC implementations, the performance testing was done on different FPGA families and speed grades, so the results are not really comparable. After the Merlin unregistered implementations, the next highest frequency was the 4x4 switch presented in [44]. This switch uses a FIFO for each input port, and an arbiter for each output port, allowing for full concurrency within the switch. The additional complexity of managing a FIFO for each input to each switch node possibly contributes to the reduced frequency compared to the Merlin network. Merlin does not provide buffering at every node, but instead implements wormhole routing, which allows a packet to be in flight across several nodes. This has the added benefit of reducing the latency seen by transactions crossing the network.

Other potential reasons why the Merlin packet fabric can achieve a higher frequency include the FPGA-optimized arbiter implementation, and the fact that the packet destinations from the address decoder are expressed as one-hot select vectors. One-hot select vectors simplify the implementation of the address decoder, as discussed in section

4.3.2.3, and allow the Merlin arbiter to take advantage of the high-performance carry chain logic available on modern FPGA device, as discussed in section 4.3.1.3.

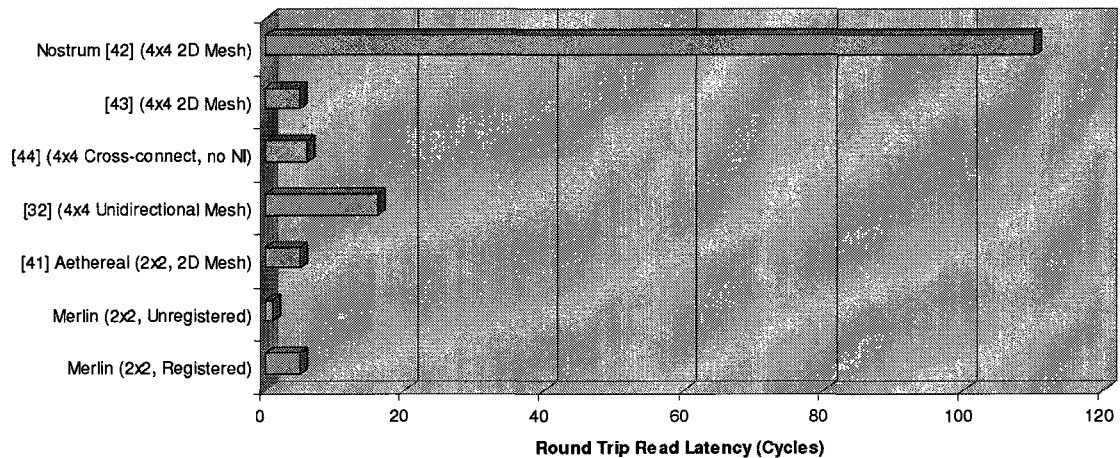


Figure 66 - Latency Comparison

The last important metric for an NoC implementation is the latency that it can provide. Figure 66 shows that the unregistered Merlin implementation can support reads with a latency of as low as one cycle, meaning that the read response can arrive at the master the cycle after the read command was issued. The registered Merlin implementation takes 5 cycles, since there is an extra cycle of latency for each of the master and slave network interfaces in each of the command and response directions. As a Merlin network gets larger, a tree of cross-connects can be used to maintain a high operating frequency at the expense of latency. It is straightforward to implement the network as an unbalanced tree such that latency critical connections see a minimal latency, while other connections go through multiple pipeline stages.

The 2D mesh implementations presented in [41] and [43] do not directly report the latency for read operations, but the best case latency can be deduced from the description of the topology. For both 2D mesh implementations, the best case is if the slave is immediately adjacent to the master. If the command is presented to the local network node on one cycle, it can be on the segment between the master's network node and slave's network node on the following cycle, and can be at the slave on the cycle after that. If the slave issues its response on the cycle immediately following, the response can be back at the master 5 cycles after the command was issued. It stands to reason that the latency is larger when a master and slave is not adjacent, and that this will be more common in large 2D Mesh networks.

The latency of the 4-port switch reported in [44] is 3 cycles, likely because of the input buffers on each port. For a cross-connect topology using these switches, the read latency would be at least 6 cycles, since the command and response packets each have to cross at least one switch. If the topology were instead a 2D mesh, then each master or slave is connected to a single network node, and the latency includes the cycles to cross two switches in each direction, for a total of at least 12 cycles.

Merlin also uses a very wide packet format to take advantage of the rich routing resources available on FPGAs. This allows a command packet and a response packet to be transferred in as few as one clock cycle each, which also keeps latency to a minimum.

5. Conclusion

As the popularity of FPGA devices continues to grow, so does the volume of logic available in a single FPGA device. Traditional bus-based interconnects with centralized arbitration mechanisms are struggling to keep up with the demands of today's FPGA developers, and more flexible and higher performance soft interconnects are required. A Network-on-Chip approach to on-chip FPGA interconnect promises to be a flexible and high-performance strategy, allowing the interconnect to take advantage of the flexibility of FPGA devices. Networks on a chip are scalable when compared to traditional buses, decouple communications from computation, and use individual protocol layers that each provide a well-defined interface, decoupling usage from implementation.

This work has presented a flexible, high performance, and light weight on-chip FPGA interconnect, built of packet based NoC primitives, a way to map transactions to packets, and algorithms for connecting components such that they provides the services required. It has reported a proof-of-concept system and demonstrated correct operation of a Merlin-based system, and compared the resulting performance to the interconnect generator available from Altera, and to a number of FPGA NoC implementations.

5.1 Summary

We have considered interconnects available from FPGA vendor system design tools, and from FPGA NoC implementations that have been published in the literature. We then presented the architecture of a new high-performance and modular interconnect solution for FPGA-based system on chips. This interconnect, named "Merlin", provides a flexible feature set to support a wide variety of applications while optimizing for the relatively high cost and low performance of FPGA logic. It takes advantage of NoC interconnect techniques, using packets to encapsulate commands and responses. The different layers of functionality that compose a Merlin interconnect have been presented, as well as some design tradeoffs that were considered in its creation. The components that implement the required functionality, the packet and transaction interfaces between component, and the algorithms used to implement an application specific interconnect implementation given a description of desired connectivity have all been described in detail.

To demonstrate Merlin, a prototype interconnect generator was implemented, including all the required components and algorithms for a proof-of-concept system. Along with the implementation, the toolset and component development and testing methodology was presented. The prototypical implementation was used to generate a proof-of-concept system including a processor core with instruction and data masters, and a number of peripherals such as an on-chip memory core for program and data storage, and a Universal Synchronous Receiver Transmitter (UART) core for communications. The proof-of-concept system was used to execute test software that demonstrates the interconnect operating correctly. For the proof-of-concept design, it was found that the interconnect consumed 36% of the total resources use by the design. By running a simple

test program on the processor, this system was demonstrated to operate correctly on an Altera 1S10 FPGA.

A number of synthetic systems were used to measure performance, so that Merlin could be compared to previous implementations. Measurement results have shown that Merlin interconnect provide better resource utilizations than other proposed FPGA NoCs, and that Merlin rivals the performance and resource utilization of a the interconnect solution from both Altera and Xilinx, the industry's two main FPGA providers. In addition, Merlin was found to provide lower latency through the network than other FPGA NoC implementations, while costing less FPGA resources.

5.2 Contributions

- An architecture for a new high-performance and modular interconnect solution for FPGA-based system on chips.
- A prototypical implementation of this new interconnect architecture, and a demonstration processor system.
- A proof-of-concept design demonstrating the prototype implementation working in hardware.
- A series of measurements that show that Merlin performance rivals FPGA vendors interconnect solutions, and is better than previously published FPGA NoC implementations.

5.3 Future Work

Networks On Chips are still rather new to the System On Chip development community, and are even newer to FPGA-based systems on chips. There are a number of questions about application specific Network On Chip interconnect generation that have yet to be answered. Some of the work yet to be done is listed below.

Performance Measurements – Presented here is an implementation and a proof of concept system. Before deploying Merlin based systems, more performance measurement and study is needed. Measurements yet to be taken include the effect of multi-stage arbitration, and the effect of burst adapters.

Additional Features – There are a number of features that are desirable for System-on-chip interconnects that Merlin does not yet support. An important example of this is a memory-aware arbiter that is able to make arbitration decisions knowing the type of memory that it is granting accesses to, and granting access based on addressing patterns.

Transform Development - There exists great opportunity for the development of transforms that solve the interconnect problem in a variety of different ways, such as with multi-hop networks, inter-component buffering, or even using token-ring based networks instead of a combinatorial fabric.

References

- [1] International Technology Roadmap For Semiconductors, “The International Technology Roadmap For Semiconductors: 2006 Update”, *The International Technology Roadmap For Semiconductors*, 2006. [Online]. Available: <http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm>. [Accessed: August 27, 2008].
- [2] Altera, “Accelerating High Performance Computing with FPGAs”, *Altera*, October 2007. [Online]. Available: <http://www.altera.com/literature/wp/wp-01029.pdf>. [Accessed: August 27, 2008].
- [3] Axel Jantsch and Hannu Tenhunen. “Will networks on chip close the productivity gap?” *Networks on Chip*, chap. 1, pp. 3-18. Kluwer Academic Publishers, February 2003.
- [4] P. P. Pande et al, “Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures”, *IEEE Trans. Computers*, vol. 54, no. 8, pp. 1025-1040, Aug. 2005.
- [5] L. Benini and G. DeMicheli, "Networks on Chips: A New Paradigm for Component-Based MPSoC Design," *IEEE Computer*, Jan. 2002, pp. 70-78

- [6] Forbes, “Altera And Xilinx Own The Market”, *forbes.com*, August, 2008. [Online] Available: http://www.forbes.com/investmentnewsletters/2006/08/03/Altera-Xilinx-Actel-gilder-in_cb_0803soapbox_inl_2.html. [Accessed: December 29, 2008]
- [7] Altera, “Quartus II Version 8.0 Handbook, Volume 4: SOPC Builder”, *Altera*, 2008. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii5v4.pdf. [Accessed: September 2, 2008].
- [8] Xilinx, “CoreConnect Technology, Product Info”, *Xilinx*. [Online]. Available: http://www.xilinx.com/products/ipcenter/dr_pcentral_coreconnect.htm. [Accessed: September 2, 2008].
- [9] Rijpkema, E.; Goossens, K.; Radulescu, A.; Dielissen, J.; van Meerbergen, J.; Wielage, P.; Waterlander, E., “Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip”, *Computers and Digital Techniques, IEE Proceedings* vol. 150, issue 5, pp. 294-302, 22 Sept. 2003.
- [10] P. Guerrier and A. Greiner, "A Generic Architecture for On-Chip Packet Switched Interconnections", *Proc. Design Automation and Test in Europe*, 2000.
- [11] R.K. Gupta and Y. Zorian, "Introducing Core Based System Design", *IEEE Design & Test of Computers*, vol.14, no. 4, October 1997.
- [12] ARM, “The Advanced Microcontroller Bus Architecture (AMBA) Specification, Rev 2.0”, *ARM*, 1999.
- [13] ARM, “Muti-layer AMBA high-performance Bus (AHB) Overview”, *ARM*, 2004.
- [14] ARM, “AMBA Advanced eXtensible Interface (AXI) Protocol v1.0”, *ARM*, 2001

- [15] Opencores, "SOC Interconnection: Wishbone", *Opencores.org*, 2002. [Online]. Available: <http://www.opencores.org/projects.cgi/web/wishbone/wishbone>. [Accessed: August 26, 2008].
- [16] Open Core Protocol International Partnership (OCP-IP) "Open Core Protocol Specification, Release 2.2", *OCP-IP*, 2006.
- [17] International Business Machines (IBM), "CoreConnect Bus Architecture White paper", *IBM*, 1999.
- [18] Matsumoto, C. "Altera, Xilinx hop diverging buses in SoC plans", *EETimes*, September, 2000. [Online]. Available: <http://www.eetimes.com/story/OEG20000925S0033>. [Accessed: August 29, 2008].
- [19] Vesa Lahtinen, Erno Salminen, Kimmo Kuusilinna, Timo D. Hämäläinen, "Comparison of Synthesized Bus and Crossbar Interconnection Architectures", *IEEE International Symposium on Circuits and Systems 2003*, Bangkok, Thailand, vol.5, pp. 433-436, May 25-28, 2003.
- [20] L. Kleinrock, "Queueing Systems, Volume 2: Computer Applications." *Wiley*, 1976.
- [21] Marsh, Ian. "Weighted Fair Queuing", *Swedish Institute of Computer Science*, 1998. [Online]. Available: http://www.sics.se/~ianm/WFQ/wfq_descrip/node21.html. [Accessed: September 2, 2008].
- [22] Open Core Protocol International Partnership (OCP-IP), "OCP-IP Frequently asked Questions – Bursting", *OCP-IP*, 2008. [Online]. Available: <http://www.ocpip.org/about/faqs/bursting/#01>. [Accessed: September 2, 2008].

- [23] “Understanding Burst Modes in Synchronous SRAMs”, *Cypress Semiconductor Corporation*, San Jose, CA, June 30, 1999.
- [24] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip", *IEEE Circuits and Systems*, vol.4, no.2, 2004.
- [25] A. Radulescu, K. G. W. Goossens, “Communication Services for Networks on Chip”, in Shuvra S. Bhattacharyya, Ed F. Deprettere, and Jrgen Teich, editors, *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, pages 193–213. Marcel Dekker, 2004..
- [26] W.J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks", *Proc. of the 38th Design Automation Conference (DAC)*, June 2001.
- [27] G. Bennett, “Designing TCP/IP Internetworks”, *VNR Communications Library*, 1995.
- [28] W. Stallings, “Data and Computer Communications, Fifth Edition”. *Prentice Hall*, 1997.
- [29] R. Fong “Improving Field-Programmable Gate Array Scaling Through Wire Emulation”, *Master's Thesis*, Blacksburg, VA, Sep 2004.
- [30] L. Tassiulas, “Cut-through witching, pipelining, and scheduling for network evacuation”, *IEEE/ACM Transactions on Networking*, vol. 7, issue 1, February 1999.
- [31] Pau, R., "A Configurable Router for Embedded Network-on-Chip Support in Field-Programmable Gate Arrays”, *Master's Thesis*, Queen’s University, Kingston, Ontario, Canada September 2008.

- [32] N. Kapre "Packet-Switched On-Chip FPGA Overlay Networks", Master's Thesis, California Institute of Technology, Pasadena, California, 2006.
- [33] J. Kim, J. Balfour, and W. Dally, "Flattened Butterfly Topology for On-Chip Networks", *IEEE Computer Architecture Letters*, vol.6, no.2, July 2007.
- [34] J. Kim, W. Dally, and D. Abts, "Adaptive Routing in High-Radix Clos Network", *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, Florida, 2006.
- [35] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip: Concepts, Architectures, and Implementations", *IEEE Design & Test of Computers*, Volume 22, Issue 5, Sept.-Oct. 2005
- [36] Altera, "Stratix II Device Handbook", Altera, 2007. [Online]. Available: http://altera.com/literature/hb/stx2/stratix2_handbook.pdf. [Accessed: September 2, 2008].
- [37] Mahmud, R. "An FPGA Primer for ASIC Designers", *EETimes*, 14 April 2004. [Online]. Available: <http://www.us.design-reuse.com/articles/article7654.html>. [Accessed: September 2, 2008].
- [38] A. Ehliar and D. Liu, "A Network on Chip based gigabit Ethernet router implemented on an FPGA", *Swedish System-on-Chip Conference (SSoCC)*, Kolmården, Sweden, May 2006.
- [39] I. Kuon, J. Rose, "Measuring the Gap Between FPGAs and ASICs", *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, February 2007.

- [40] Altera, "Device Comparison", Altera, 2008. [Online]. Available: http://www.altera.com/cgi-bin/device_compare.pl. [Accessed: January 16, 2009].
- [41] A. Kumar et al, "Reconfigurable Multi-Processor Network-on-Chip on FPGA", *Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging*, 2006.
- [42] D. Akerlund. "Implementation of a 2x2 NoC with Wishbone interface", *Master's Thesis*, Laboratory of Electronics and Computer Systems, Royal Institute of Technology, Sweden, 2005.
- [43] F. Moraes et al, "A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping", *International Conference on Very Large Scale Integration*, 2003.
- [44] A. Ehliar and D.Liu, "An FPGA based Open Source Network-on-Chip Architecture", *International Conference on Field Programmable Logic and Applications*, 2007.
- [45] Xilinx, "Processor Local Bus (PLB) Arbiter Design Specification", *Xilinx*, 2002. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/plb_v34.pdf. [Accessed: January 16, 2009].
- [46] M. Millberg, E. Nilsson, and R. Thid. "The Nostrum Protocol Stack and Suggested Services Provided by the Nostrum Backbone". November, 2002.
- [47] Altera, "Altera Avalon Interface Specifications, v1.0", *Altera*, March 2008. [Online]. Available: www.altera.com/literature/manual/mnl_avalon_spec.pdf. [Accessed: September 2, 2008].

- [48] Altera, “Quartus II Version 8.0 Handbook, Volume 2, Chapter 11: Netlist optimizations and Physical Synthesis”, *Altera*, 2008. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii52007.pdf. [Accessed: September 2, 2008].
- [49] Synplicity, “Synplify v7.0 Reference Manual”, *Synplicity*, October, 2001. [Online]. Available: http://www.synplicity.com/literature/pdf/synplify_ref_1001.pdf. [Accessed: September 2, 2008].
- [50] Astels, D. “Test Driven Development, a Practical Guide” , *Prentice Hall*, July, 2003.
- [51] Clark, M. “JUnit FAQ”, *JUnit.org*, Feb 2006. [Online]. Available: <http://junit.sourceforge.net/doc/faq/faq.htm>. [Accessed: August 29, 2008].
- [52] Altera, “Stratix II Device Handbook, Volume 1”, *Altera*, 2007. [Online]. Available: http://www.altera.com/literature/hb/stx2/stx2_sii51002.pdf. [Accessed: September 2, 2008].
- [53] Xilinx, “Achieving Higher System Performance with the Virtex-5 Family of FPGAs”, *Xilinx*, 2006. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp245.pdf. [Accessed: September 2, 2008].
- [54] Altera, “Nios II Processor Reference Handbook”, *Altera*, 2008. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf. [Accessed: September 2, 2008].