Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2005-04-13

A Flexible Circuit-Switched Communication Network for FPGA-Based SOC Design

Clint Richard Hilton
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

 Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation

Hilton, Clint Richard, "A Flexible Circuit-Switched Communication Network for FPGA-Based SOC Design" (2005). *All Theses and Dissertations*. 312.
https://scholarsarchive.byu.edu/etd/312

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

A FLEXIBLE CIRCUIT-SWITCHED COMMUNICATION NETWORK FOR FPGA-BASED SOC DESIGN

by

Clint R. Hilton

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

August, 2005

Copyright © 2005 Clint R. Hilton

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Clint R. Hilton

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Brent E. Nelson, Chair
Date	Michael J. Wirthlin
Date	Doran K. Wilde

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Clint R. Hilton in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Brent E. Nelson
Chair, Graduate Committee

Accepted for the Department

Michael A. Jensen
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, Ira A. Fulton College
of Engineering and Technology

ABSTRACT

A FLEXIBLE CIRCUIT-SWITCHED COMMUNICATION NETWORK FOR FPGA-BASED SOC DESIGN

Clint R. Hilton

Department of Electrical and Computer Engineering

Master of Science

As FPGA densities continue to improve, single chips are becoming capable of implementing larger and more complex systems. Even today these systems may include several processors working in conjuction with a handful of other standard interfaces or custom modules. Additional system complexity naturally leads to added complexity throughout the different design and implementation stages. Attempting to design such a system while maintaining high performance and within a reasonable time frame is becoming more and more difficult.

Architectural design approaches ranging from direct module interconnection to sophisticated bus schemes have been used to build such systems, all with their own trade-offs. Often direct module interconnection results in the best overall performance but at the cost of design time and flexibility. Bus schemes on the other hand attempt to simplify the integration of the different hardware modules and allow for a more modular design approach. However, since the bus is a single shared interconnection medium, a practical limit is placed on the system's acheivable throughput. A relatively new architectural approach to system design involves a network-based

communication infrastructure. A network-based interconnect scales much better than the shared bus and provides a potential increase in system throughput capabilites.

An effective approach would be one that can provide the throughput capabilities of direct interconnect, the modular design advantages of the shared bus, and the flexibility to adapt to different system requirements while maintaining lightweight communication.

A design infrastructure that attempts to meet these requirements has been developed. This infrastructure is based on a circuit-switched network architecture. The circuit-switching aspect allows two nodes, or modules, to temporarily establish a direct and dedicated connection for high-throughput data transfer. The network-based topology allows this to occur without tying up all the interconnect resources as other routes can be used to connect the other nodes. Each node is connected to the network via a well-defined interface therefore allowing for modular design. Flexibility is built into the architecture to accommodate many different topology configurations. Lightweight protocols and handshaking mechanisms are used to establish node-to-node connections, and initiate and terminate data transfers.

Two different example applications have been implemented with this network-based interconnect: one that involves the use of a single resource that must be shared among different modules, and another that has high system bandwidth requirements and dynamically schedules the use of functionally identical resources. These implementations were then compared against that of a bus-based approach. Both applications illustrate the effectiveness of this network architecture in SoC implementation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The number of transistors that can be packed onto a single chip continues to increase at an exponential rate [1]. This increase in chip density clearly leads to an increase in the amount of logic that can be incorporated on the chip. More available logic resources allow for the implementation of larger and more complex systems. As a result the notion of a *System on Chip* (SoC) has taken form and become a popular research topic.

In the context of this work, an SoC is a system of interconnected components that are implemented on a single chip. SoCs often consist of a heterogeneous mix of components that include one or more general purpose microprocessors. Traditionally, systems often required complex board-level design since each component was implemented on a separate chip (ASSP, ASIC). Today however, similar systems can be targeted to a single chip as illustrated in Figure 1.1. Integrating everything onto a single chip often results in improvements in power consumption, manufacturing costs, and in many cases design time.

As these systems continue to grow in size and complexity, incorporating effective communication between components in a reasonable time-frame is becoming increasingly difficult. A significant amount of work has been done in an attempt to standardize SoC communication in order to facilitate intellectual property (IP) reuse and reduce design time. Shared buses and packet-switched network architectures are two of the more commonly proposed standard interconnects. This work compares these different approaches and presents an alternative interconnection scheme that

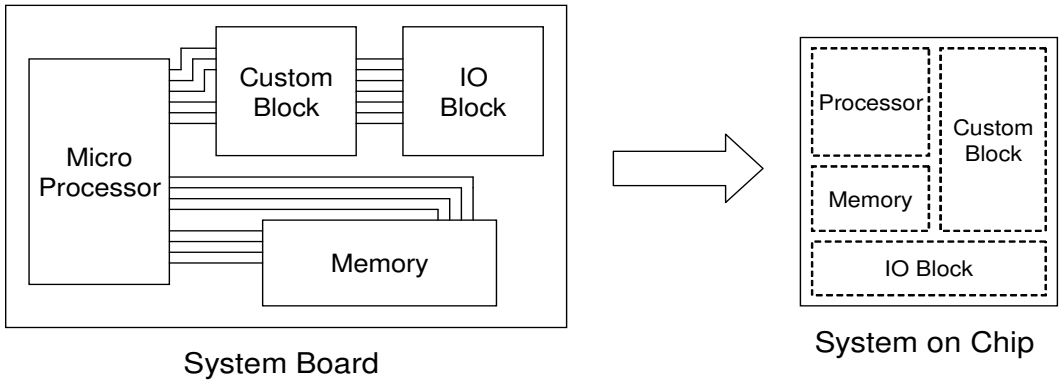

Figure 1.1: System on Chip Mapping

consists of a flexible circuit-switched rather than packet-switched network infrastructure. The advantages of this approach will be explained throughout this work.

1.1 Programmable SoCs

A special family of chips known as Field Programmable Gate Arrays (FPGAs) are among those that have taken advantage of increasing chip densities. FPGAs are chips whose function can be defined at run-time through the downloading of a configuration bitstream. Because of the amount of logic resources available on today's FPGAs, entire systems can be implemented thereon. The combination of fast reconfigurability and vast resource availability make FPGAs an attractive target for SoC implementation. For these reasons the research presented in this work is targeted to FPGAs.

Though FPGAs are often larger, slower, and less energy efficient than their ASIC counterparts, FPGA-based SoC design provides several important advantages as identified in [2]:

- **Fast Time-to-Market.** The process of implementing a circuit onto an FPGA simply requires the generation and downloading of a configuration bitstream. Since this bitstream is generated with software tools and can be downloaded onto the FPGA in just seconds, these systems can quickly be tested and made

ready for production. ASICs on the other hand require a mask to be generated and then shipped off for fabrication, which can take months, before being able to test the actual hardware implementation.

- **Lower Manufacturing Costs.** This bitstream generation process is not only fast, it is also inexpensive. Improvements can be made to the system design and a new bitstream generated without any additional cost. Fabrication of ASICs however, is extremely expensive. Unless the product is intended for mass production, many cannot even afford the manufacturing costs.

- **Flexibility.** The flexibility provided by FPGAs through reconfiguration is what set them apart from traditional ASICs. This allows for system updates as improvements are made and bugs are identified and corrected. The ability to completely change the system's operation on the fly is a valuable feature for many application domains. Several FPGA families also come equipped with the ability to perform *partial* reconfiguration, where only a specific portion of the circuitry is modified on the fly while the rest remains busy performing its defined task. This opens up a completely new design paradigm that involves the capabilities of dynamic module replacement.

1.1.1 Dynamic Module Replacment

In order to understand where dynamic module replacement might prove useful, it is first important to discuss the different components that might make up an SoC. In many system-based applications a CPU, or a group of CPUs, make up the primary control for the system. The CPU core communicates with the other modules of the system to complete the desired task. These other modules may consist of standard interfaces such as a UART controller or memory interface, or they may be completely custom circuits that perform some specialized computation.

The advantage of dynamic module replacement arises when certain modules are used during specific time periods throughout the system's active life. In such cases these modules can be plugged in only when needed, and then replaced by something

that would prove more useful during the times the module is not needed. Likewise, if certain responsibilities of the system change over time, more appropriate modules can be plugged into the system to meet its changing needs.

1.2 Ideal Architectural Approach

In considering the challenges of system design, an ideal SoC architecture would be one that includes the following characteristics:

- **High system throughput.** Fast and unimpeded data transfer between modules.
- **System flexibility.** Architecture configurability that enables the construction of numerous topologies to meet the specific system requirements.
- **Lightweight communication.** Interconnect mechanism doesn't severely limit the resources available for system task modules and simple protocols for low-overhead communication.
- **Modular design.** Design approach that facilitates reuse of IP blocks with as little concern for the interconnect mechanism as possible.

1.3 Common Architectural Approaches

Many different architectural approaches have been proposed to meet the challenges of complex SoC design. Some of the more common SoC architectural approaches and their respective trade-offs can be categorized as follows:

- **Direct Module Interconnect.** Shown in Figure 1.2, this approach consists of modules that are directly connected in a custom manner so data can be transferred between modules exactly as needed. As a result, systems designed using this approach tend to achieve higher throughput, lower latency, and in most cases require fewer resources. However, since this is a more custom approach, design time is lengthened, and flexibility is reduced. Whenever a module is

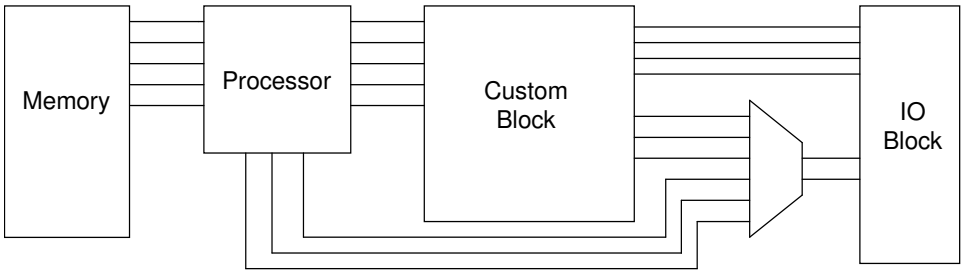

Figure 1.2: Direct Module Interconnect

modified during the design phase, the entire interconnection associated with that module may also require modification.

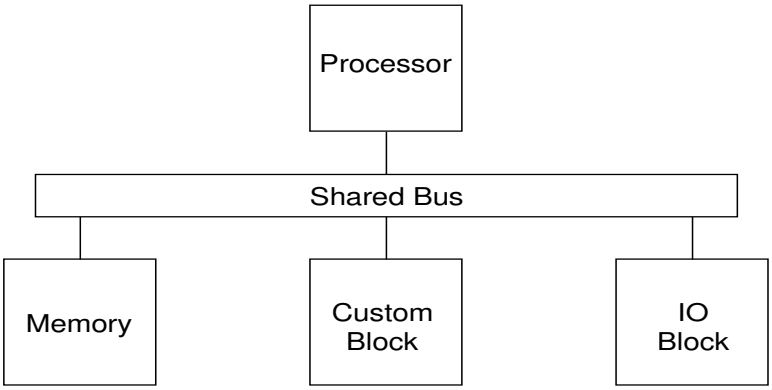

Figure 1.3: Memory-mapped Peripheral Bus

- **Memory-mapped Peripheral Bus.** By nature this approach, shown in Figure 1.3, is often processor-centric and uses a shared memory/peripheral bus to allow for communication between modules. Because there is a fixed interface to the shared bus, this architecture lends itself to a modular design approach therefore resulting in reduced design time and increased flexibility. The use of a single shared bus however, places a limit on the achievable system throughput

and can significantly increase the worst-case latency. As the system increases in size the bus becomes an even more severe bottleneck.

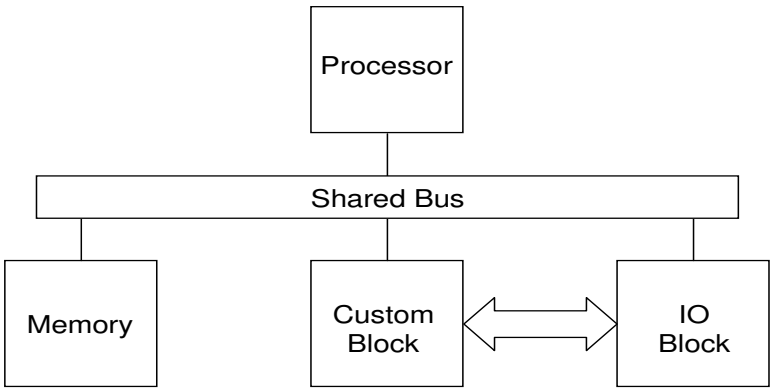

Figure 1.4: Bus/Direct-interconnect Hybrid

- **Bus/Direct-interconnect Hybrid.** One of the more common approaches currently used is a combination of the previous two design strategies. This hybrid approach, shown in Figure 1.4, generally uses a shared peripheral bus for all communication involving processor(s) and/or memory interface(s). For other intra-module communication, direct interconnect may be used. In many cases this approach takes advantage of the strengths of both strategies. The ability to have modules directly connected may remove the shared bus as the throughput bottleneck, and at the same time reduce average bus access latency. As previously mentioned, this direct interconnect can have adverse effects on the design time and flexibility of the system design.

- **Network on Chip Interconnect.** There are many different variations of network topologies that have been proposed to serve as the interconnect for SoCs. Most consist of a 2-D mesh-based topology, similar to that shown in Figure 1.5,

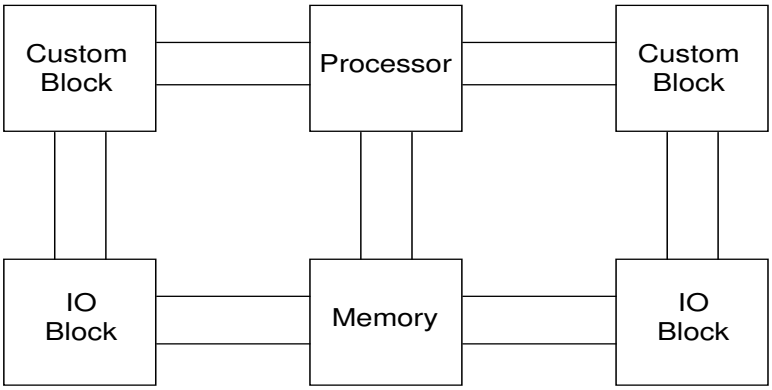

Figure 1.5: Network on Chip Interconnect

and perform packet switching using wormwhole or cut-through routing. In general, a network interconnect provides several advantages over the bus architecture. Probably the most important advantage is in its scalability. Systems are only going to get larger and more complex as time goes on, and bus architectures will likely not accommodate the increasing communication demands. Network architectures may soon become the dominant interconnection mechanism. In addition to their scalability, their potential for high-throughput data transfer is a feature that may help to propel them to the forefront of SoC architectures. In many cases network-based interconnect also results in a more energy-efficient system since the data does not need to be broadcast to every module of the system. However, there are a few concerns with the network architectures that have been proposed. The main disadvantages result from the overhead incurred by the networking infrastructure. The routing circuitry tends to tie up a significant portion of the available resources. Also the packet-switched nature of these architectures cause problems for applications that require heavy data flow between modules. The process of forming, parsing, and buffering packets incurs substantial overhead. Because of these weak points, none of the proposed network architectures have become widely accepted for use in SoC design.

1.4 Programmable Network on Chip

An SoC architecture has been created that attempts to match the characterists of the ideal SoC. This architecture is based on a circuit-switched rather than packet-switched networking infrastructure. A router, or series of routers, is used as the interconnect medium for the system's modules. Lightweight protocols and handshaking mechanisms are used to establish dedicated point-to-point connections between two modules to allow for high-throughput data transfer. Other modules can simultaneously establish point-to-point connections since there is no single shared bus but a series of possible routes between modules.

A significant amount of flexibility has been built into this architecture to allow for the creation of numerous network topologies. Due to the programmable nature of the proposed network architecture, this infrastructure will be referred to as Programmable Network on Chip (PNoC). A modular design approach can be used since each module has a well-defined port interface to this network. This often results in improved design time and increased design flexibility as it becomes much easier to modify or replace modules during the design process.

Another important benefit of a fixed communication interface is the ability to support dynamic module replacement. Unfortunately the current design flows as provided by FPGA vendors for partial reconfiguration are not well-developed. Generating dynamic module bitstreams, though possible, is a relatively difficult task. The work described here doesn't actually demonstrate the dynamic replacement feature but support for it is built into the architecture.

1.5 Structure of This Work

In the chapters that follow, current SoC architectures will be identified, and the PNoC will be described in more detail. The following is an outline of what is contained in this work:

- Related work in this area is presented and analyzed.
- A general technical description of the PNoC is provided.

- A discussion of the network router.
- An explanation of network modules and their design requirements.
- The details of the CPU network interface and its corresponding software usage are described.
- Two example applications are presented and the features they exploit are identified. Their PNoC implementation is also compared to that of a more standard approach.
- A conclusion of this work is provided.

The appendix provides a step-by-step tutorial for constructing a system using the PNoC architecture.

Chapter 2

Background

A significant amount of work has been done that has helped to inspire and motivate some of the architectural decisions that have gone into this research. The first section of this chapter describes different advancements that have been realized in SoC design. The second section of the chapter analyzes different network-based approaches that have been proposed and their respective contributions. The third section briefly describes the status of partial reconfiguration for Xilinx FPGAs. The fourth and final section introduces JHDL (a Java-based Hardware Description Language), which was used in the implementation of this work.

2.1 Relevant Work in SoC Design

In recent years SoC design has become a viable solution for the implementation of a variety of complex systems. Initially, and still in many cases these SoCs are designed in a custom manner and are specific to the application at hand. Many researchers realize however, that as chip densities continue to increase, and systems grow in size and complexity, the design of such systems in a reasonable time-frame will not likely be feasible. As a result, the push for a more general or standard interconnection mechanism that will allow for a modular design approach is critical for future SoC design.

A common architecture that is based on a processor-centric system is that of the shared bus, shown previously in Figure 1.3. Example bus architectures that have become well established include the ARM-based AMBA bus [3] and IBM's CoreConnect [4]. Since the shared bus is a standard interconnection medium, it allows for a

modular design approach that facilitates IP reuse [5]. Given a well-defined interface to the bus, each module can be effectively designed, tested, and debugged independently. However, a strictly bus-based architecture has serious limitations since the modules of the system are confined to a single bus for all data transfers. A more favorable bus architecture is one that supports multiple split buses through the use of bus bridges. This can allow multiple bus transactions to occur simultaneously as long as they occur on different bus segments. However, split-bus architectures affect the design simplicity and flexibility of the system as changes to the system components may also require changes to the overall bus structure.

Another bus-based approach that is often attractive is the bus/direct-interconnect hybrid architecture (refer back to Figure 1.4). Such an architecture is provided in both Xilinx's Embedded Development Kit (EDK) [6] and Altera's System On Programmable Chip (SOPC) Builder [7]. Both platforms use a shared bus architecture, but maintain the design flexibilty that allows for modules to be directly interconnected as well. The shared bus is primarily used for communication involving the CPU or memory interface. The direct interconnect capabilities allow for high-throughput data transfer between modules.

In such systems the CPU is often used to coordinate the scheduling of tasks and then perform certain functions that are not time-critical while the hardware modules execute the compute-intensive tasks. Since the modules can pass data through direct connections, the shared bus is often eliminated as the system's performance bottleneck. In addition to increasing the system throughput capabilities, this hybrid architecture also scales better than a typical bus approach because only the modules that need to communicate with the CPU actually need to connect to the bus. The direct connection capabilities, however, do complicate the design process. This custom interconnect reduces the modularity of the system so more time must be spent verifying the integration of the system modules.

2.2 Relevant Work in NoC Design

As the trend for larger and more complex SoCs continues, several important design considerations arise. In [8] a few of these are identified and discussed. One that received particular attention involves system clocking schemes. In many current SoC architectures, the systems are implemented using a single clock source. As chips become more dense, and systems become more complex, this simple clocking scheme will not suffice. Instead a globally asynchronous – locally synchronous scheme was proposed to meet the needs of today's growing systems.

In addition to clocking, it was also suggested that the limited scalabilty of bus architectures will eventually give rise to network-based interconnects. The shared-medium bus architectures are effective for systems requiring only a handful of interconnected modules, but for larger systems the bus becomes a critical limiting factor. On-chip networks that provide multiple interconnecting routes between modules are an attractive solution for increasingly complex SoCs.

2.2.1 Packet-Switching Architectures

One of the early network architectures proposed for use in SoCs was developed at Stanford and presented in [9]. In making their case for a network-based approach, it was pointed out that networks would be preferred to buses because they have higher potential bandwidth as they are capable of supporting multiple concurrent data transactions. Their proposed architecture consists of a 2-D folded torus topology where each module includes a small area reserved for networking logic so that packets can be routed among all modules on the network.

Another group [10], proposed a similar 2-D mesh topology that requires a routing switch for each module on the network as shown in Figure 2.1. They named their approach CLICHE ($Chip-Level\ Integration\ of\ Communicating\ Heterogeneous\ Elements$) as they emphasized the notion of heterogeneous system design. They argue that many systems perform a variety of tasks and therefore require a heterogeneous mix of modules to make up the system. Though their architecture is highly scalable, they concede that their architecture is unsuitible for certain heavy data flow

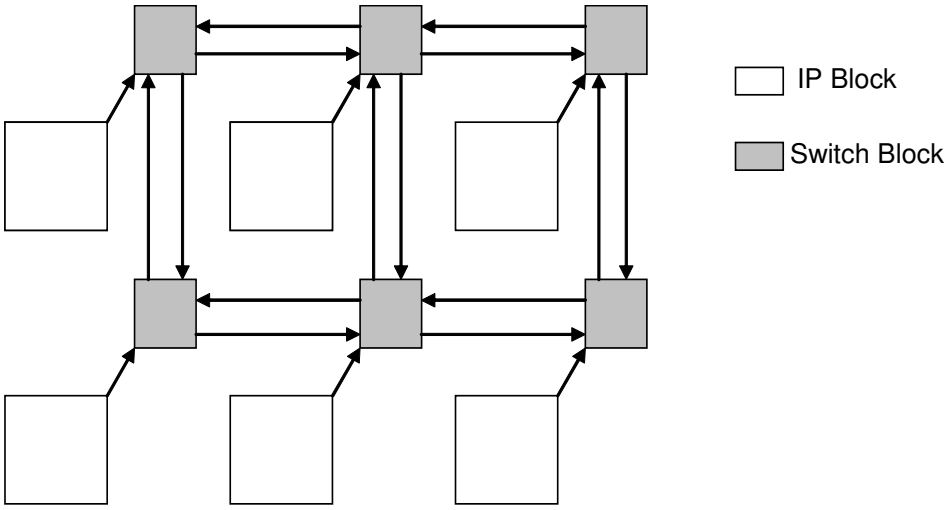

Figure 2.1: The CLICHE Architecture

systems due to its limited performance which comes as a result of its message passing overhead.

Research published in [11], at the University of British Columbia, proposed a completely different network topology in an attempt to create an effective SoC architecture. This topology, shown in Figure 2.2, is described as a butterfly fat-tree graph where the leaves of the tree represent the IP blocks and the vertices correspond to the network switches. The motivation for this topology is to minimize wire delay for the system. This architecture helps to reduce global wire lengths between nodes and allows for better interconnect delay predictions. Their argument as to why bus architectures are so limited is due to the bus's long wire delays. With their proposed architecture, the minimal wire delays would serve as the major source for improved overall system performance and scalability.

2.2.2 FPGA-Specific Architecture

The network architectures previously described are targeted to general SoC platforms. A few network-based SoC architectures have been developed that specifically target FPGAs. One such architecture was proposed by a group from Belgium

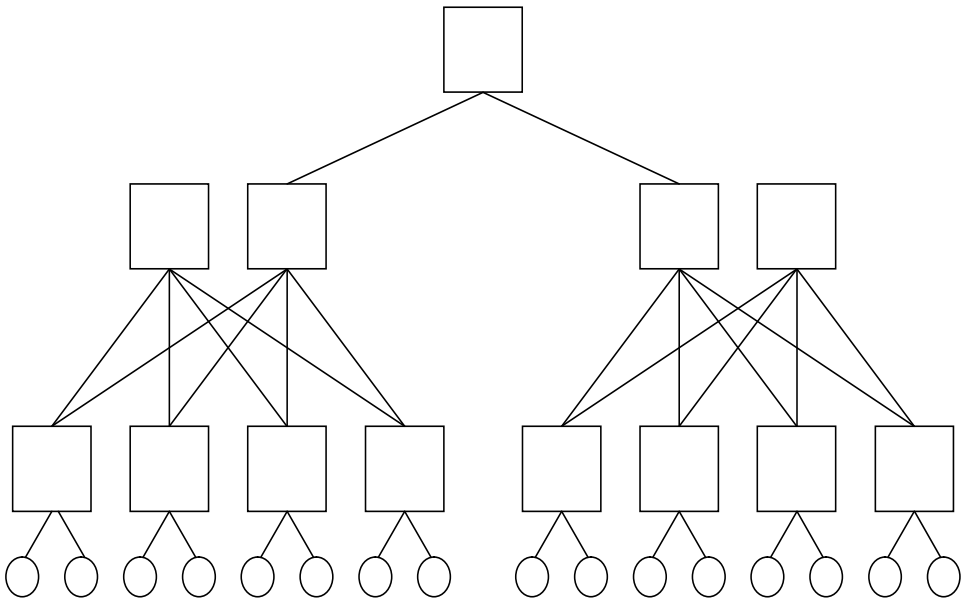

Figure 2.2: Butterfly Fat-Tree Network

[12]. The topology for this architecture is similar to that proposed in [9]. It consists of a 2-D torus topology and performs packet switching through the use of wormhole routing. This work was among the first in publication to use partial reconfiguration for dynamic module replacement in the context of network-on-chip design. With a fixed routing network established, partial reconfiguration could be used to dynamically replace individual modules on the network, allowing for dynamic hardware multi-tasking. A more detailed description of this process is provided in Section 2.3.

2.2.3 Packet Switching vs Circuit Switching

All of the network architectures previously mentioned make use of packet switching, and surprisingly few even mention the possibility of a circuit-switched approach. Two groups: [13] and [14], provide strong arguments for the use of circuit-switched networks in SoC implementations.

The major difference between packet-switched and circuit-switched networks is in how the data is transferred between modules [15]. In packet-switched networks,

data packets, also called *datagrams*, consist of fixed-length blocks of data that contain routing information and are independently routed through the network to the desired destination. In circuit-switched networks, a dedicated connection path (i.e. *virtual circuit*) between two modules is established so the raw data can be freely transferred between the modules.

Packet-switched networks often allow for greater resource utilization as many packets can be in flight at a given instant. They are well-suited to systems that exhibit short, bursty data traffic that can easily be formed into packets [14]. However, in general they require more sophisticated congestion control and packet processing. Large buffers are often required to queue up packets awaiting the availability of routing resources.

Circuit-switched networks, on the other hand, are connection-based, meaning that a module must first go through the process of establishing a dedicated connection path to the desired destination before transferring data. This can introduce undesirable latency, but once the connection is made, high-throughput data transfer between the modules can be guaranteed, and the data latency is extremely predictable. In general, circuit-switched networks are better suited for data traffic that is longer in duration as the connection setup time becomes negligible [14]. A common argument against circuit switching is its potential for significant underutilization of the communication links. Connections may be established in which the communication remains idle for an extended period of time. This can result in inefficient resource utilization if other modules need access to that communication link.

Both [13] and [14] propose fixed circuit-switched network topologies and focus on link scheduling techniques to maximize communication link utilization. This research presents a flexible circuit-switched network that uses a low-overhead connection setup process to accommodate both long and short data traffic and improve link utilization. A circuit-switched approach was chosen due to its relative simplicity, low overhead, modular design, and intra-module throughput capabilities.

2.3 Status of Xilinx Partial Reconfiguration

Partial reconfiguration as currently supported by Xilinx is explained in [16]. This capability allows one or more configuration frames to be reconfigured while the rest of the chip remains active. Special bus macros are used to retain connectivity between the logic being reconfigured and the active portion. Such a feature allows for the design of a static routing fabric and a series of reconfigurable nodes or modules. The modules retain their connection to the routing fabric during reconfiguration through the use of these bus macros. Because of the frame structure of current Xilinx devices, there are some design limitations when attempting to support partial reconfiguration. The reconfigurable modules must meet the following requirements:

- Since reconfiguration frames are column-based, the module's height must always be the full height of the device.
- Since the reconfiguration frames are each four slices wide, the module must be horizontally placed on four-slice boundaries and its width must be a multiple of four slices.
- All logic resources that lie within the boundary of the reconfigurable module are part of that module's reconfiguration frame. This includes all block RAMs, multiplier blocks, TBUFs, IOBs, and routing resources. This makes generic module-based bitstream generation a difficult task since the configuration frames are not identical.

Generating a bitstream for a module that meets these requirements involves specific floorplanning. This can be done at the HDL level through the use of low-level LOC calls to constrain the tools during the placement phase. The alternative is to design the module using a low-level tool such as FPGA Editor. Neither of these solutions are very attractive for large or complex modules. Unfortunately there is currently no high-level constraint parameters that can be applied to a module at the top-level to keep it constrained within a specified reconfiguration frame.

Once a partial bitstream has been generated, it can be loaded onto the FPGA through the use of the Xilinx internal reconfiguration access port (ICAP). ICAP consists of an 8-bit input data bus and an 8-bit output data bus that allows internal logic to reconfigure the device's configuration memory, therefore modifying the hardware's functionality. Work presented by a group at Xilinx [17] introduces a software API for accessing the ICAP on a Virtex-II device. Through these API calls the system software can perform the dynamic module insertion/replacement.

As mentioned in Section 2.2.2, the research presented in [12] uses partial reconfiguration to provide dynamic module support. Their work uses the ICAP API calls to perform the desired hardware replacement. In order for their system's modules to meet the reconfiguration requirements previously listed, their 2-D torus was folded into a 1-D torus and bus macros were incorporated at the module port interfaces. A floorplan of their system is shown in Figure 2.3.

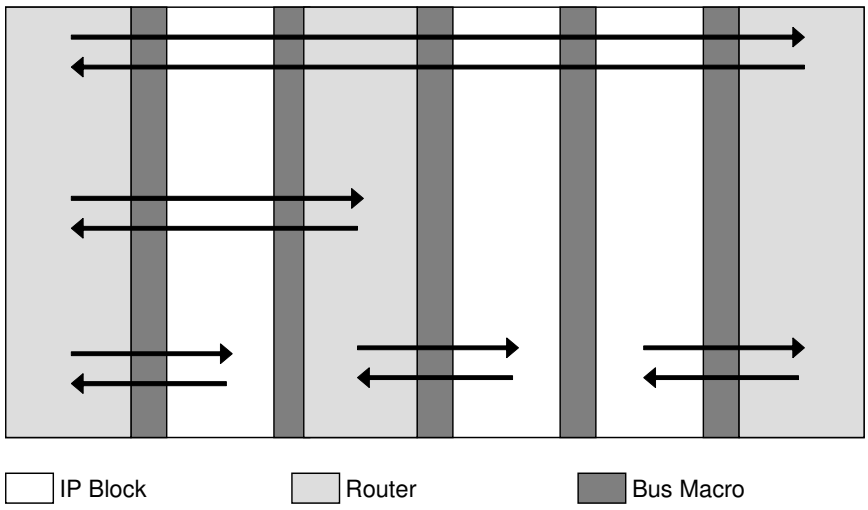

Figure 2.3: Xilinx System Floorplan for Dynamic Module Support

As mentioned previously, because of the difficulty in generating useful partial reconfiguration bitstreams, this work doesn't demonstrate the use of dynamic module replacement, but provides architectural support for that capability once partial bitstream generation becomes more widely supported.

2.4 JHDL

The actual implementation of this work was done using JHDL and its associated simulation tools. JHDL, developed at Brigham Young University and presented in [18], consists of a library of Java classes that provide an object-oriented approach to structural FPGA circuit design. Also included in this library is a graphical simulation environment and EDIF netlister. The following features of JHDL led to its selection as the principle design tool for this work:

- **Standard programming language** - Java, a well-developed, widely-used programming language with extensive documentation, significantly facilitates the construction of parameterizable circuits.
- **Debugging capabilities** - the circuit generation process can be easily monitored and debugged with the use of traditional software debugging tools and print statements. A native Java simulation tool that uses the Swing libraries allows for seamless integration between circuit generation and circuit simulation.
- **Polymorphism** - the ability for network modules to extend the NetInterface class simplifies the design of user modules and their integration with the network infrastructure.

2.5 Summary

Much work has been done recently in the area of SoC design. As custom approaches have been dominant in SoC design up to this point, two alternatives have been growing in popularity: bus-based and network-based architectures. Shared bus architectures have gained momentum since on-chip processor cores have become increasingly common in SoC implementations. This processor-centric infrastructure

allows for a modular design flow that simplifies SoC design. Network architectural approaches have been growing in popularity as chip densities continue to increase and SoCs grow in size and complexity. The network's scalability and potential bandwidth improvements make them an attractive alternative.

Several network architectures have been identified, most of which use packet switching with wormhole routing. The network architecture proposed in this work is based on a circuit-switched communication mechanism to allow for localized high-throughput data transfer. Extensive flexibility is provided to allow for numerous system configurations and run-time dynamic module replacement. Lightweight protocols and modular-based design simplify the creation of systems components. JHDL has been selected as the design tool for this system as its features further simplify the design of parameterizable system modules.

Chapter 3

Programmable Network on Chip: General Description

The goal of this work is to create an effective interconnection framework for use in SoC design. Two major aspects set this work apart from that which has been previously done. First is the circuit-switched approach that this architecture implements, which can guarantee high-throughput data transfer between two modules that share a dedicated connection. The second characteristic is the flexibility built into this architecture that simplifies the creation of numerous system topologies while maintaining lightweight communication. Several pieces are required to make such a system work. This chapter is dedicated to giving a general description of these pieces and illustrates how they come together to make an effective network architecture. The following chapters provide a more detailed explanation of the actual implementation for each of these pieces.

3.1 Circuit-Switched Architecture

As discussed previously, circuit switching provides advantages over packet switching in that once two modules have set up a dedicated connection between each other, they are guaranteed a period of unimpeded data transfer. This takes advantage of an assumption made on general heterogeneous systems – which is that most of the critical high-throughput data transfer occurs between specific nodes, and is not uniform among all modules of the system. Circuit switching can allow the critical nodes to establish dedicated connections to more effectively increase the system's throughput capabilities.

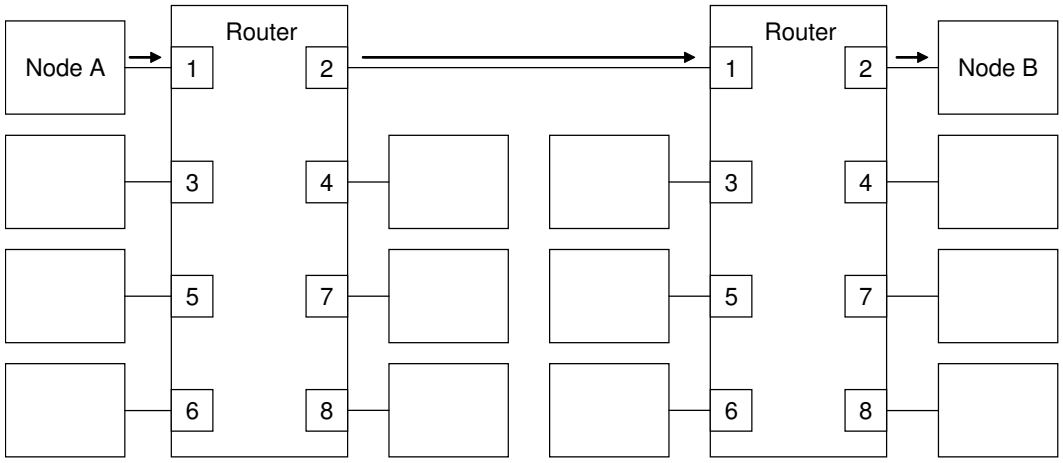

Figure 3.1: Example PNoC Topology

The proposed network topology consists of a series of subnets where each subnet consists of a single router along with several network modules or nodes. Figure 3.1 shows how a simple system might be connected using this architecture. The architectural features that define the circuit-switched nature of this network are built into the system's routers. A lightweight handshaking mechanism is used to establish the dedicated connection between modules. The node that desires to establish the connection, referred to as the master node (Node A in the figure), sends a request to the router. The router services the request and determines which port is associated with the desired target node (Node B). In this example the connection request is forwarded on to the second router since the target node (Node B) resides in its subnet. The second router then processes the request to identify the port connected to the target node. Once that port (port 2) becomes available, the router establishes the dedicated connection and informs the master that the connection has been made. The master and slave are then free to transfer data as necessary.

Once the master has determined that the data transfer is complete, a port release command is sent to the router and the connection is terminated. The released ports then become available for use by other modules. A timeout mechanism has also been incorporated into the router that monitors the requests for a busy port. If a

pending request awaits the release of a busy port, a *pend* signal will be issued to that connection's master node. The master node can monitor this signal to detect when a timeout occurs, at which point the master issues a release command to allow the pending module access to the shared resource. If the master node still needs access to the port it can re-issue a connection request command and wait for the port to become available again. It is through this time-multiplexing mechanism that no single module can completely tie up a shared resource. This timeout mechanism is optional since there may be some cases where a master node cannot allow interruption of its dedicated connection.

3.1.1 Network Routers

In this architecture the network is divided up into one or more subnets. The router serves as the central module for each subnet on the network. Each router can be configured with up to eight ports that serve as the interface by which modules are connected to the router.

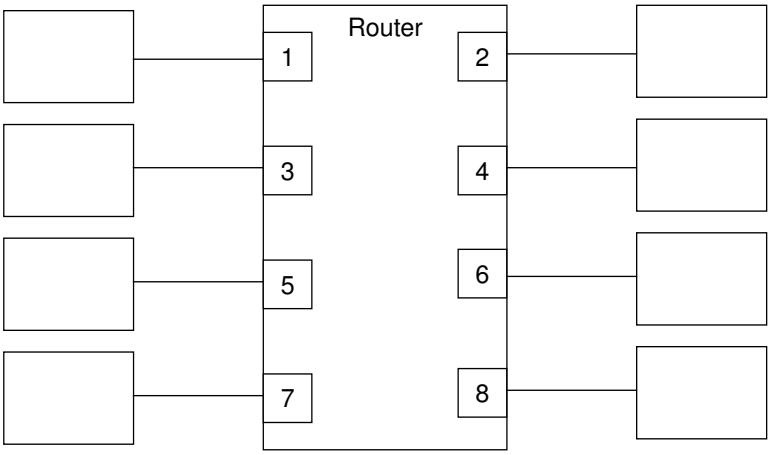

Figure 3.2: A Single Router System

In systems where several subnets are required, some of those ports must be reserved for intra-router connections so that data can be transferred between subnets. The data widths for these ports are parameterizable. The configurability of these routers allows for the creation of a topology that is specialized for the application at hand. The manner in which modules are connected to the routers is also flexible. In designs that involve only a few modules, a single router can be used with all ports available for module connectivity as shown in Figure 3.2.

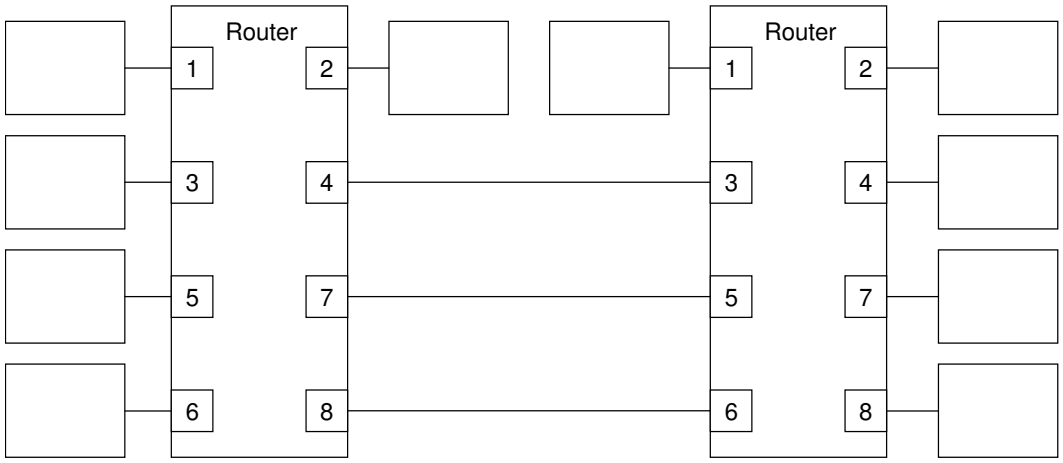

Figure 3.3: High Intra-Router Bandwidth

On the other hand, in systems that involve many modules and require significant cross-subnet communication, several of the router's ports can be connected to neighboring routers to increase the intra-subnet bandwidth. An example of such a system is shown in Figure 3.3.

As mentioned in the previous section, the routers are responsible for the establishment of dedicated connections between modules. In making these connections, the router performs two different phases of arbitration. The first phase involves arbitration for access to the routing table. When a connection request is received from a master node, it is accompanied by a destination address. The router arbitrates

which master's request to process and then performs a table look-up to see which port is associated with the selected master's target address. Module addresses in this system are function based. For example, two modules that perform the exact same task can be configured to have the same source address. In this way it is possible to increase performance if multiple modules often perform a common task and there exist multiple such resources. Since there may be multiple instances of a particular target address it is likewise possible that there exist multiple ports associated with that address. This is where the second phase of arbitration occurs. The router selects just one of the ports in order to establish the dedicated connection, and leaves the other open for another module to access.

Since the routing table is the means by which the router associates modules and ports, in order to support dynamic module replacement there must exist a mechanism for updating the routing table. This initially occurs at startup. Each module sends a router update request to its associated subnet router. The router services these requests similar to a connection request and modifies its table. Then, when a module is to be replaced via partial reconfiguration, the dynamic module controller, whether it be software based or a dedicated hardware block, is responsible for clearing the module to be replaced from the routing table. Once the new module has been configured into the network it must send a router update request to update the routing table with its source address. When the router services a router update request it forwards that request on to all routers with which it is connected. In this manner each router on the network knows exactly which ports are associated with a particular module address.

Figure 3.4 illustrates how a recently configured module with a source address of 5 updates the routers on the network. Represented in the figure is a three-stage update process. In the first stage the updating node issues a router update request to its subnet router. During the second stage, the router updates its table by modifying the entry at index 5 (the node's source address). The data in the table is a mask value where each bit corresponds to one of the router's ports. In this example, a 0x01 is written at index 5 for the first router since port 1 was the source of the update request. Also during stage 2, the router forwards the update request to the

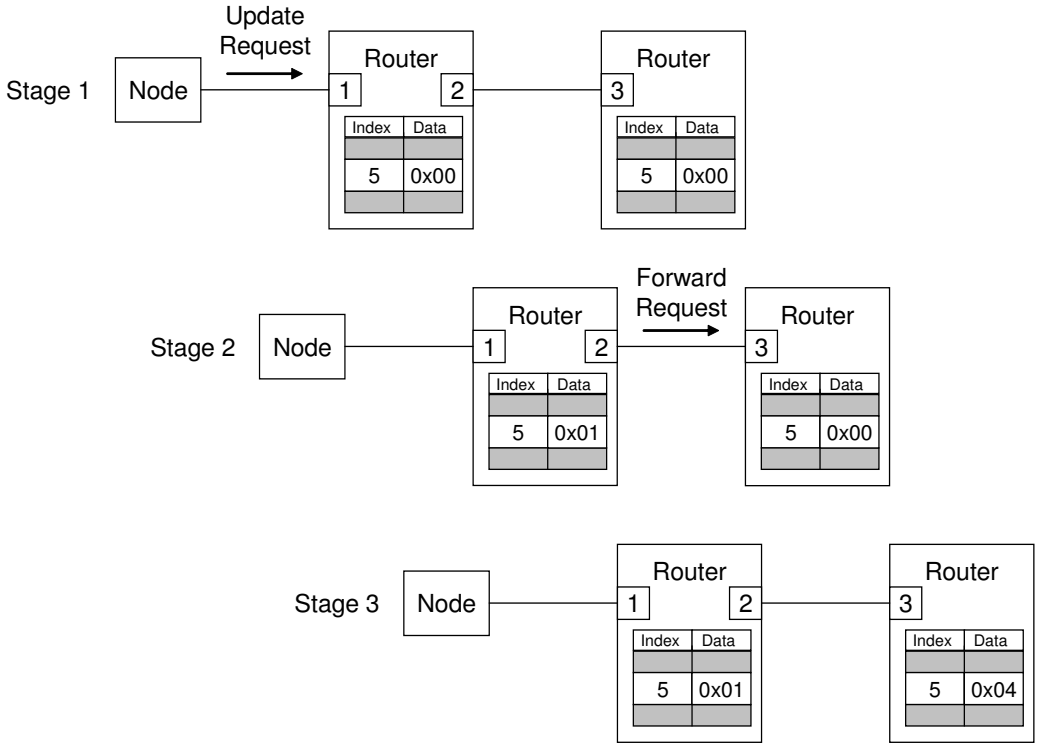

Figure 3.4: Routing Table Updates

other router(s) in the network. The neighboring router updates its table during stage three, which involves writing the value 0x04 at index 5. A 0x04 is written as it maps to port 3 (0x04: the third bit is a 1). This process continues until each router has received the update and is capable of associating the node address 5 with one of its port interfaces.

3.1.2 Network Modules

A network module can consist of practically any hardware task block. This might range from something as simple as a single adder to something as complex as an entire subsystem. The only qualification for a network module is that it must require some sort of communication with other network modules and it must fit on the target device. Good candidates for network modules are those blocks that can be shared by several different hardware tasks. Some examples might include

standard interfaces such as memory controllers, UART controllers, and other off-chip communication interfaces. Special custom modules may also fit these requirements. If several hardware tasks require the use of an FFT block or some common custom computation, that computation can be implemented as a network module and can therefore be shared among the modules that require its services. Another hardware block that is likely a good candidate to be implemented as a network module is a general purpose processor. In many systems, a CPU is the module responsible for controlling the scheduling of tasks and, as a result, requires connectivity to all the hardware modules involved.

Once a hardware task block has been identified as a viable network module it must be appropriately interfaced to the network. As a result, each network module consists of two parts: the hardware task block and the network interface. All modules connected to the network must conform to a well-defined top-level interface so that dynamic module connectivity can be supported. The network interface portion of the module is responsible for performing router update requests and connection requests. It is also responsible for the implementation of congestion control and crossing clock domains if required by the network implementation. Both of these issues will be addressed later in this chapter.

3.1.3 CPU Interfaces

In this system, CPU modules are a special type of network module. In order for a CPU to effectively communicate with other modules on the network, there are two types of memory mapping that must take place. The first is a mapping for establishing network connections, and the second is a mapping for network-based data transfers. The CPU's connectivity to the network is directly controlled by the software. The software uses a *network access* pointer to initiate and terminate connections to other modules on the network. Similarly, the software then uses a *network data* pointer to transfer data to the desired network module. Chapter 6 explains the details of the CPU network interface.

3.2 Modular Design Flow

One of the design advantages that is shared by both network and bus architectures is that of a modular design flow. Both approaches take advantage of a well-defined interconnection mechanism that allows the designer to focus on the functionality of a module and reduce the effort spent on the design and debug of the interconnect mechanism. In both systems the modules must conform to the top-level port interfaces and obey the associated communication protocol, but in general this can be verified effectively at the module level.

Another advantage that comes as a result of this module-based infrastructure is flexibility both at the design phase and during run-time. Design time flexibility simply involves the notion that top-level module instantiation is similar among all modules. Replacing modules at design time only requires modifications to the module instantiations. Little or no change is required to the interconnect. The only possible change is that of port parameterization and involves no extra design. Run-time module replacement is a little more involved but can be a powerful feature for a system. Dynamic module replacement requires the use of partial reconfiguration as explained in section 2.3.

3.3 Clocking Scheme

The clocking scheme envisioned for this network architecture involves a single global clock for the network backbone (i.e. the system routers) and then separate clocks that are local to the modules of the system. For a small network topology a single system clock could be used, but the capability to handle multiple clock domains is vital for large systems. The boundaries for clock domains are located at the node interfaces, where FIFOs are used to accommodate the differing data rates.

3.4 Data Flow Control

Supporting multiple data rates on chip implies the potential for overflowing buffers. If a master node is capable of sending data at a rate that exceeds the slave node's ability to receive that data, it won't take long before the slave's buffer is filled

and data is lost. In order to prevent such occurrences, a flow-control mechanism has been built into the architecture. This mechainsm involves the use of a clear-to-send (CTS) signal in conjuction with the node interface FIFOs. The CTS signal can either be manually disabled by the node module or it will automatically be disabled when the node interface's receive FIFO is *almost full*. The *almost full* flag is used so that data already in flight can be safely stored in the FIFO.

In order for flow control to be fully supported however, the CTS signal must be taken into account in the module design. The sending module must monitor the CTS signal and stall data tranfers until CTS is again raised.

3.5 Summary

The strengths of this architecture lie in its circuit-switching advantages and system flexibility. The circuit-switching nature of this architecure allows for localized high-throughput data transfer between nodes. The flexibility includes that from both design and run-time perspectives and comes as a result of a standard interconnection interface and router/module parameterization.

The routers are the pieces responsible for managing the dedicated connections between modules, and the modules themselves contain network interfacing circuitry that aid in the handling of asynchronous data flow. A special CPU interface is used to connect a general processor to the network infrastructure, and its network access is controlled by the software. The chapters that follow describe in detail the implemenation for each of these major components.

Chapter 4

PNoC Router Description

The router is the core of this network architecture. This chapter describes the implementation details of the router and explains the major components involved. The flexibility built into the router is explained, identifying its parameterizable features. Then, a description is provided as to how the routing table works and how it is updated as the system changes. Finally, the process of point-to-point connection control is explained.

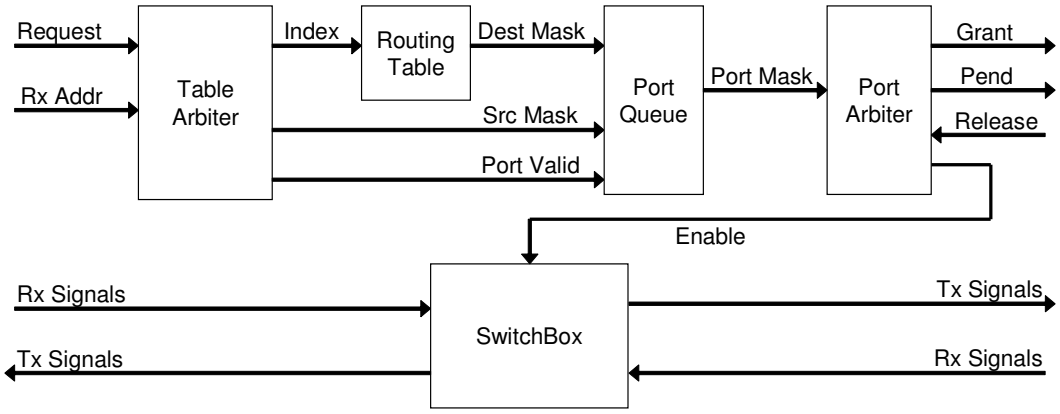

Figure 4.1: Router Block Diagram

4.1 Router Component Overview

The major components of the router are shown in the block diagram of Figure 4.1 and described as follows:

- **Table Arbiter** - The table arbiter receives connection requests and schedules access to the routing table when mulitple requests are received on the same clock cycle. This block is also responsible for managing the routing table update requests.
- **Routing Table** - The routing table maps network module addresses to ports that may be used to establish connections between modules.
- **Port Queue** - This queue is used to maintain the connection request order while the requests await availability of the target port(s).
- **Port Arbiter** - Once the target port(s) becomes available, the port arbiter establishes the desired connection and issues the appropriate *grant* signals. This block also monitors the *release* signals for the disabling of connections.
- **Switch Box** - The switch box forms the actual connections between modules by enabling tri-state buffers that allow the Rx signals to drive the appropriate Tx signals.

4.2 Parameterizable Features

The flexibility that is built into the routers is key to enabling the construction of a wide variety of systems. Two major pieces of the router design are parameterizable therefore allowing for numerous system topologies. First is parameterization of the port interfaces that connect to the router. Second is the flexible way that routers may be connected to each other and to the other modules of the system.

4.2.1 Port Interface

Each router is built with a parameterizable number of port interfaces with up to a maximum of eight ports. The number of port interfaces indicates how many

different modules can be connected to the router. The signals that define each port interface as seen by the router are listed in Table 4.1.

Table 4.1: Router Port Interface Signals

Signal Name	Direction	Description
request	input	initiates either a router update request or a connection request
release	input	initiates a connection release
grant	output	indicates to the network module that its connection request has been granted
sl_grant	output	indicates a connection has been established with the target node as a slave
pend	output	indicates that another module is requesting access to the destination port
rx_data[X:0]	input	rx data bus of parameterizable width
rx_addr[Y:0]	input	rx address bus of parameterizable width
rx_rnw	input	rx read-not-write signal
rx_valid	input	indicates valid rx data, address, and rnw signals
rx_cts	input	rx clear-to-send signal
tx_data[X:0]	output	tx data bus of parameterizable width
tx_addr[Y:0]	output	tx address bus of parameterizable width
tx_rnw	output	tx read-not-write signal
tx_valid	output	indicates valid tx data, address, and rnw signals
tx_cts	output	tx clear-to-send signal

The widths of the *rx_data*, *tx_data*, *rx_addr*, and *tx_addr* signals are parameterizable up to a maximum width of 32 bits. Since ports of differing widths may be connected to each other, by convention the least significant bits of the larger bus are tied to the smaller bus signals. For example, if a 32-bit *rx_data* from one port connnects to an 8-bit *tx_data* of another port only the least significant 8 bits of *rx_data* will actually be connected to *tx_data*.

Care must be taken when parameterizing the bus widths for a router's ports when that router is to be used with dynamic module replacement. Once a router's port widths have been set they are fixed and cannot be dynamically modified.

4.2.2 Router Connectivity

The other flexible aspect of the routers is in how they can be connected to the other modules of the system, including other routers. A topology can be constructed that best meets the specific system's needs. If a system requires more bandwidth between adjacent routers, mulitple ports can be used to connect the routers. Also if a system would benefit from multiple instances of a particular module, the routers have the capability built-in to establish a connection with whichever one is avaliable at the time. The details of how this works will be described in Section 4.4.2.

Depending on the FPGA used, this system-level flexibility may be compromised some when support for dynamic module replacement is desired. Because of the design restrictions identified in Section 2.3 to support dynamically replaceable modules, the network, implemented on Xilinx FPGAs, is limited to a column-based topology. Such a system may look similar to that shown in Figure 4.2.

4.3 Routing Table

An important part of the router is its routing table. This table is responsible for mapping a given module address to the appropriate port interface. When a master node requests a connection to a given target node, the address associated with that target node is used as the index to the routing table. The output of the routing table is a mask value that identifies the ports that may be used to establish a route to the target node. Each bit of this mask value maps to one of the router's port interfaces.

4.3.1 Table Updates

Upon startup, and whenever a change is made to the network through dynamic module replacement, the routing tables are updated via router update requests. The

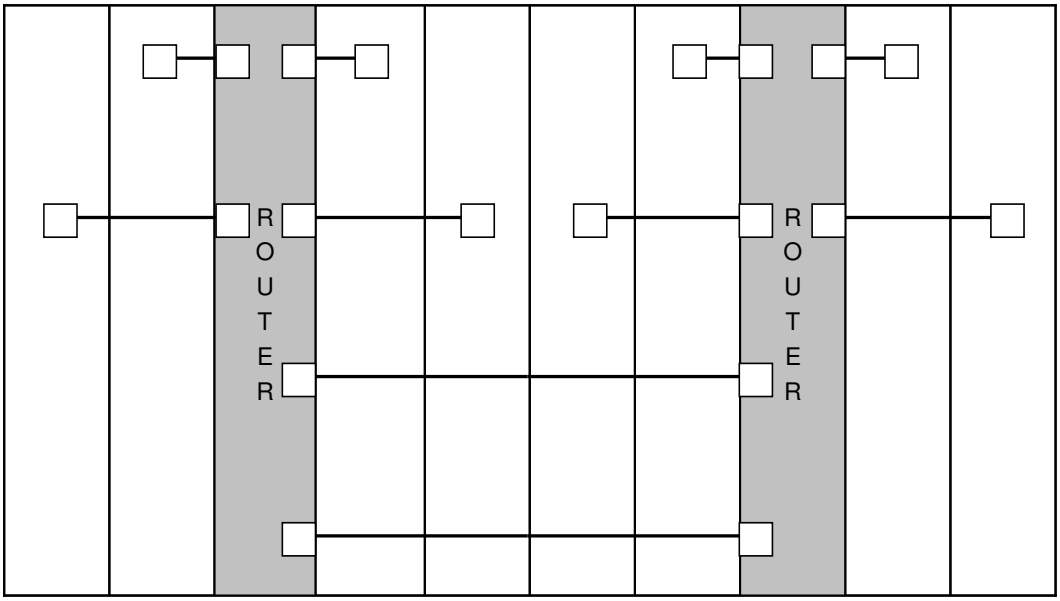

Figure 4.2: Topology for Dynamic Module Support

modules are responsible for adding themselves to and removing themselves from the routing table of the router that lies in their subnet.

These router update requests are implemented as connection requests addressed to the router. Routers maintain a fixed source address value of 0x00. Whenever a router receives a request from one of its port interfaces that is addressed to 0x00, it is recognized as a router update request. If the corresponding *rx_rnw* signal is low then the request is to add the module to the routing table, otherwise the request is to remove the module. When adding a module to the routing table, the *rx_data* signal represents the module's source address and is used to index the routing table. A read-modify-write is performed in which the table output is logically or'd with the port mask that made the router update request. Upon completion of the update request, the updating node's *grant* signal is raised and that node can begin requesting network connections.

Once a router update request has been serviced, that request is forwarded on to all adjacent routers until each router on the network has updated its table

appropriately. Each router is initially configured with routing table entries for all of its adjacent routers so that routers know through which ports to forward the router update requests. In order to prevent a flooding of router update requests, the routers check to see if the updating entry is already in the routing table. If so the router does not forward the request.

4.3.2 Illustrative Example

A simple example will be used to illustrate how a router update request is processed. The timing waveform in Figure 4.3 will be used to help describe each step of the process.

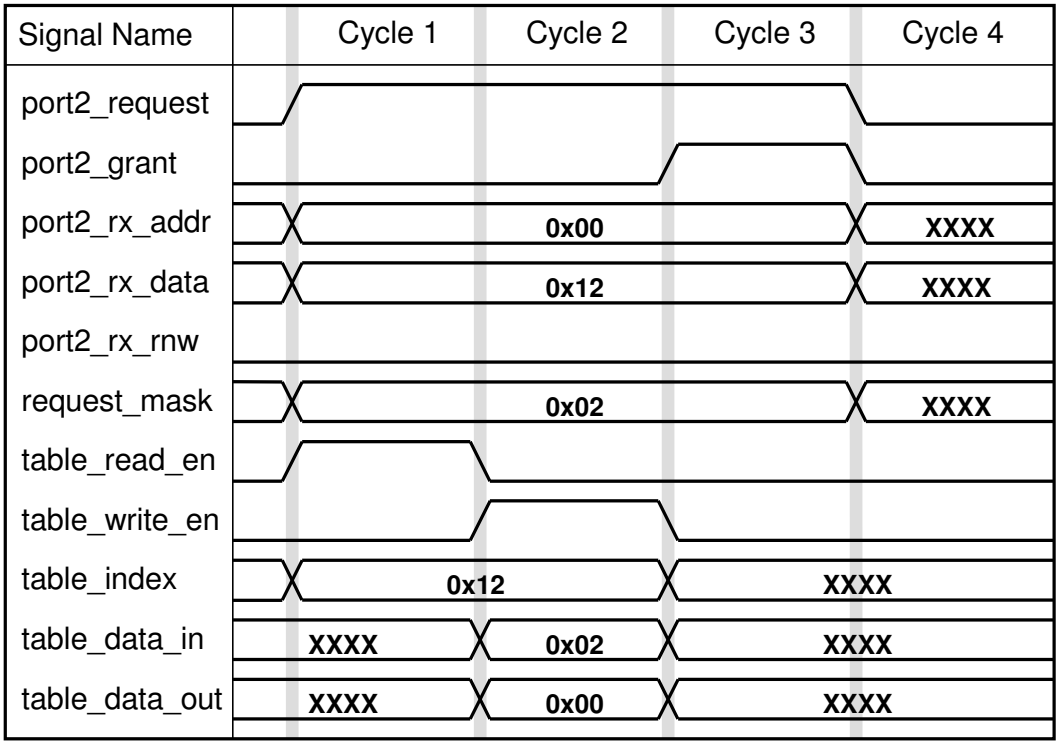

Figure 4.3: Router Update Process

In this example a module with a source address of 0x12 is connected to the router's port-2 interface. During clock cycle 1, this module initiates the router update request by raising its $port2\_request$ line while addressing the router ($port2\_rx\_addr = 0x00$) and presenting its source address on the data bus ($port2\_rx\_data = 0x12$). During this cycle the $port2\_rx\_data$ is used to index the routing table and begin the read-modify-write process. At clock cycle 2 the output of the routing table becomes valid ($table\_data\_out = 0x00$) and is logically or'd with the $request\_mask$ value. The result ($table\_data\_in = 0x02$) is then written to the routing table. This is how the module's address gets associated with the appropriate port interface. The result of this process is the value $0x02$ stored at location $0x12$. Then, when a module makes a connection request in which the desired target module has address $0x12$, the router will associate that address with the port-2 interface and establish the desired connection.

4.4 Connection Process

The primary function of the routers is to manage the establishment of dedicated connections between modules. Since multiple ports can make requests during the same clock cycle, this process involves a few levels of arbitration. The first is responsible for allowing access to the routing table and is controlled by the table arbiter. The second, controlled by the port arbiter, is in determining which port to use to establish the connection. Once a connection has been established the router serves as a single pipeline stage for the data transfer between the connected modules.

4.4.1 Table Arbitration

The routing table can only service one request at a time. Since multiple port requests can be issued on the same clock cycle, a table arbitration scheme is needed to determine which port gets access to the table. This arbitration scheme is implemented in a round-robin-like fashion. A series of ring counters are used to assign priorities to each port's *request* line. The request with the highest priority is given access to the router for that given transaction. The priority ring counter is then incremented,

and the next high priority request is determined. This process continues in similar fashion as long as there are pending requests.

Since the connection requests only require access to the routing table for one cycle, the maximum table access latency is equal to the number of ports in the router. An example will be used to illustrate the table arbitration for two requests that are received on the same clock cycle. The block diagram in Figure 4.4 illustrates the modules involved in the connection process and the corresponding timing diagram is shown in Figure 4.5.

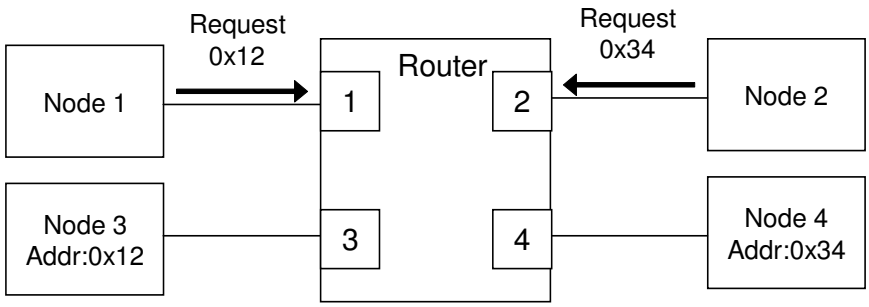

Figure 4.4: Table Arbitration Process - Block Diagram

In this example, Node 1 requests a connection to Node 3, and Node 2 requests a connection to Node 4. Since these requests are issued simultaneously, the router's port interfaces 1 and 2 receive the requests on the same cycle. Port 1 involves a connection request to a module with an address of 0x12 while port 2 addresses 0x34. At the time that these requests occur, during clock cycle 1, port 1 is assigned a priority level of 1 while port 2 is assigned priority level 2. Therefore port 1 is of higher priority and gets access to the routing table first. This is evident during clock cycle 2 where the *table_index* is 0x12 which is the $port1\_rx\_addr$ value. On the next clock cycle port 2 is granted access as the *table_index* is 0x34, the $port2\_rx\_addr$ value.

The module addressed as 0x12 (Node 3) is connected to port 3, and the module addressed as 0x34 (Node 4) is connected to port 4. These address/port associations

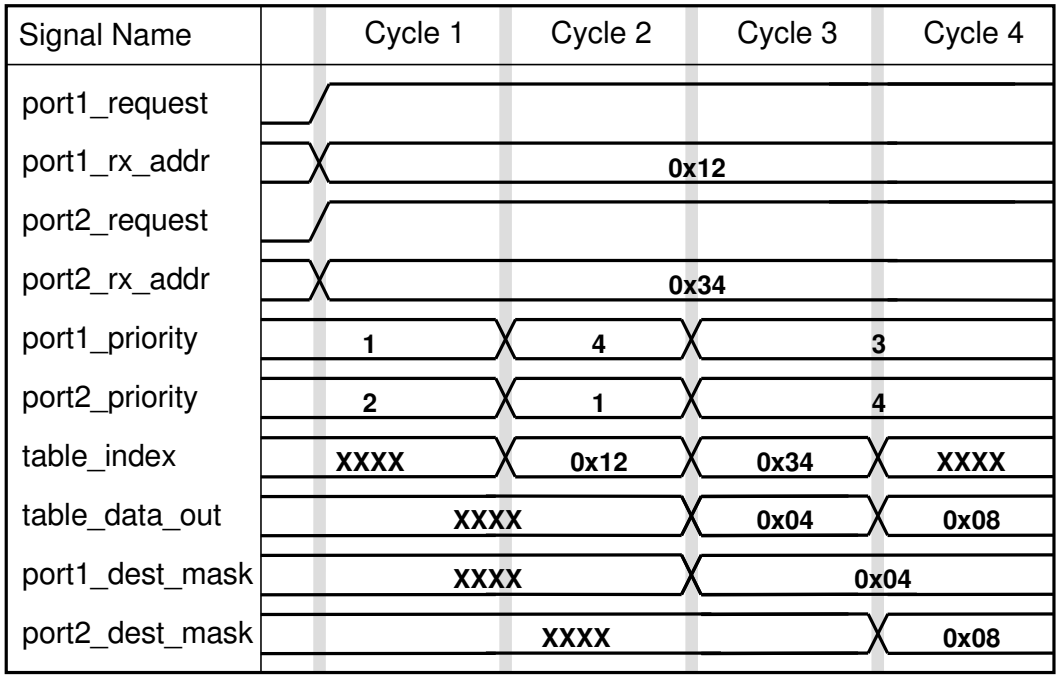

Figure 4.5: Table Arbitration Process - Timing Diagram

are determined by the router during clock cycles 3 and 4. During clock cycle 3, one cycle after port 1's table access, the *table_data_out* value of 0x04 maps to port 3 as the third bit is a 1. Similarly during clock cycle 4, one cycle after port 2's table access, the *table_data_out* value is 0x08 which maps to port 4. These mapping mask values are later used by the port arbiter to enable the connections between Node 1 and Node 3, and between Node 2 and Node 4.

4.4.2 Port Arbitration

Once a port has successfully accessed the routing table, the table's output is used to determine with which port the requesting node must connect in order to establish a route to the target node. As mentioned previously, the output of the routing table is a port mask value that consists of a bit for each port associated with that router. Each bit in that mask value that is a 1 represents a port that may be used to establish the desired connection.

Once this destination mask has been obtained, the router must determine whether any of the corresponding destination ports are available. A port is unavailable if (a) it is currently used in a previously established connection or (b) if the node attached to it has its *tx_cts* held low, indicating that the node is not ready to accept a connection. Since it is possible that none of the destination ports may be available, each port has a request queue associated with it. A requesting port waits its turn in the queue until the destination port becomes available. At that point a simple arbitration scheme is used to determine which of the available destination ports to use, and the connection is established. A *grant* signal is then issued to the requesting node and the *sl_grant* is issued to the target node to indicate that a dedicated connection is in effect.

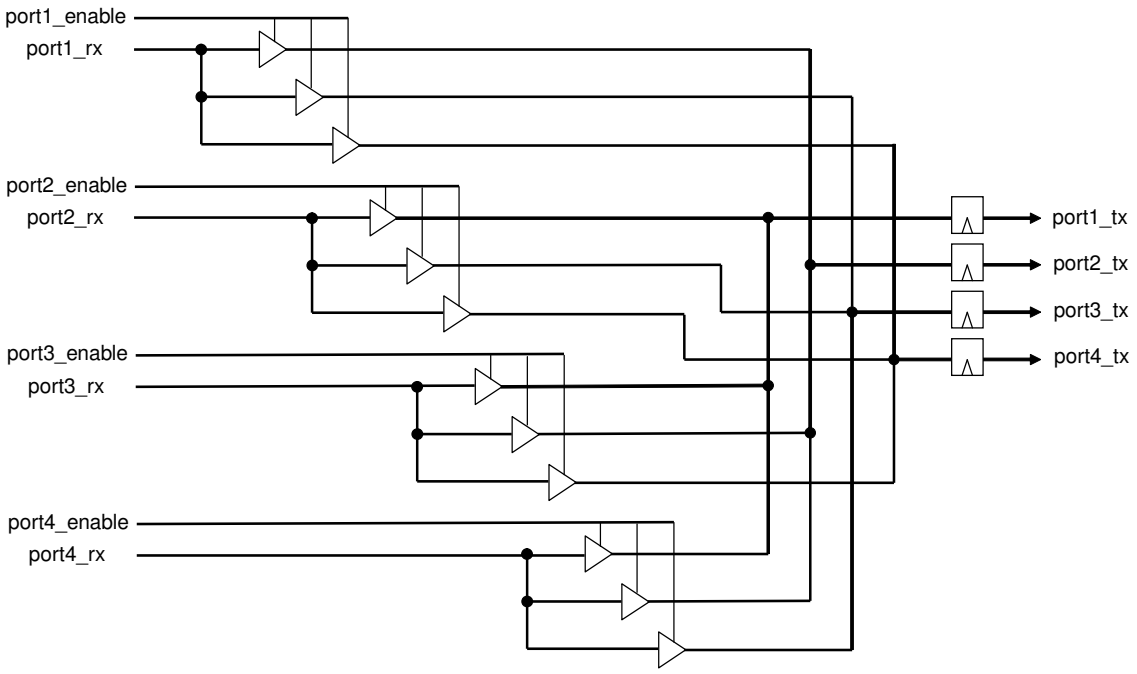

Figure 4.6: Switch-box Hardware Diagram

The actual connection circuitry is implemented as a tri-state enabled switch-box. Each set of *port_rx* signals, which includes the *rx_data*, *rx_addr*, *rx_rnw*, *rx_valid*, and *rx_cts* signals, has a one-hot *enable_mask* bus associated with it to enable one of the tri-state buffers that correspond to the target *port_tx* signals. Figure 4.6 shows how this would look where there exists four port interfaces to the router.

4.4.3 Termination Process

There are two instances in which a dedicated connection might be terminated through the assertion of the *release* signal: upon completion of the data transfer, or at the signaling of an optional timeout mechanism. In both cases it is the node that established the connection that must initiate the release.

When the desired data transfer is complete, the master node issues a release command to the router by raising the *release* line. This *release* signal disables the dedicated connection and clears the *grant* signals between the two ports. In situations where a potential hogging of a port might be detrimental to the operation of the system, the master node may monitor a timeout request signal (*pend*) from the router and comply with the assertion of the *release* signal to allow a pending connection to be established.

4.4.4 Illustrative Example

A basic example will be used to further illustrate how the router's connection process is realized. In this example it is assumed that there are four port interfaces to the router, and that the module at port 1 desires to connect to port 3, and the module at port 2 desires to connect to either port 3 or port 4. Once the requests have been received by the router, the table arbitration process, as explained in Section 4.4.1, is used to associate each request with its desired destination port mask. For this example the destination port mask for port 1 is 0x04 and the destination port mask for port 2 is 0x0C. Figure 4.7 illustrates the connection process that takes place after the table access has occured.

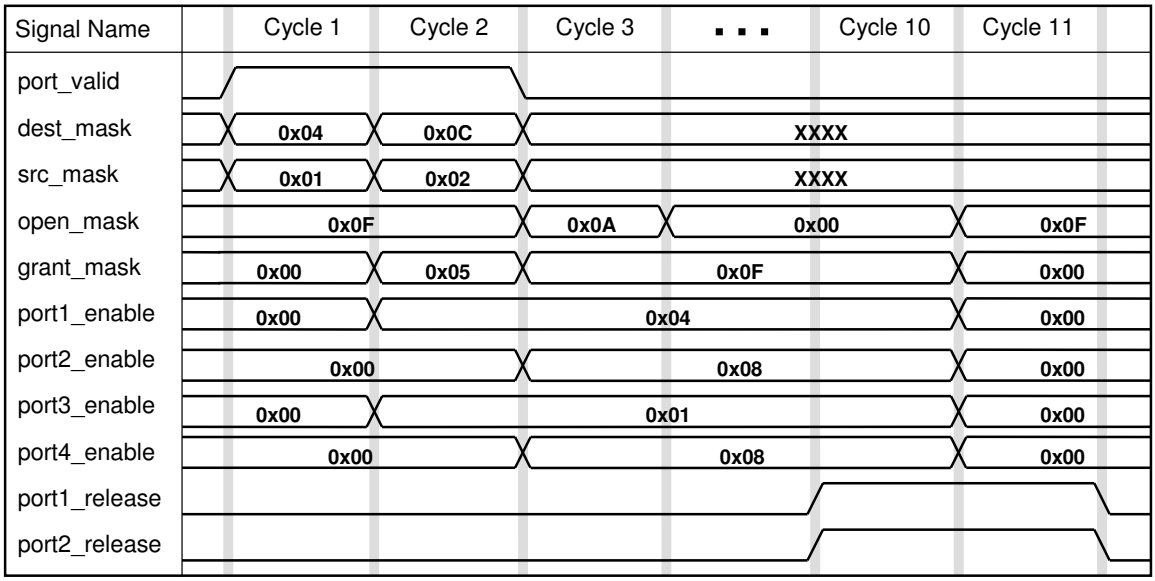

Figure 4.7: Port Connection Process

The destination port mask for port 1 becomes valid during clock cycle 1 ($dest\_mask = 0x04$). At this point the connection request is loaded into a queue to wait for the availability of port 3, as determined by the *open_mask* value. Since initially all the ports are available ($open\_mask = 0x0F$), the destination port can be selected, and the connection established. Therefore during clock cycle 2, the *grant* and *sl_grant* signals for ports 1 and 3 respectively are raised ($grant\_mask = 0x05$), and the tri-state switch is enabled appropriately ($port1\_enable = 0x04$).

Also during clock cycle 2, the next valid request is received: port 2's connection request to either port 3 or port 4 ($dest\_mask = 0x0C$). This request is entered into the queue, and the arbiter determines that one of the target ports (port 4) is available. So, on the next clock cycle (cycle 3), the request is removed from the queue and the *grant* and *sl_grant* signals for port 2 and port 4 are raised ($grant\_mask = 0x0F$) along with the corresponding tri-state enable signal ($port2\_enable = 0x08$).

At this point data can be freely tranferred between the connected modules (ports 1 & 3 and ports 2 & 4). Once the data transfer is complete, the master

nodes terminate the connection with assertion of the *release* signals. These can be serviced simultaneously as shown during cycle 10, where both the *port*1_*release* and *port*2_*release* signals are raised. On the following cycle, the *grant* and *sl_grant* signals are lowered, and the tri-state enable signals are disabled. The ports at that point become available for other connection requests.

4.5 Summary

The router is the core of this virtual circuit based network architecture. It is responsible for establishing the dedicated point-to-point connections between the system modules. The routing table is used to associate the router's ports with the network addressable modules. Table arbitration ensures that requests are serviced effectively even as multiple requests are received simultaneously. Since multiple ports can map to a single network address, a port arbiter is used to isolate one that is available for connection. The actual connection between modules is implemented as a tri-state enabled switch box. This mechanism allows for multiple concurrent connections, where each connection provides unimpeded data transfer between modules.

Chapter 5

PNoC Module Interface

One of the principle goals of this work is to facilitate the design of complex systems through modular design and a simple interface to the communication medium. This chapter describes in detail the node interface infrastructure that addresses these issues. The first two sections detail the node's port interface to the network and the hardware required. The last two sections provide the details involved in establishing point-to-point connections and transferring data, both as master and slave.

5.1 Node Interface

Modules that connect to the network do so via a well-defined port interface. This interface is defined by the signals in Table 5.1 as seen from the module. As mentioned previously, the rx_data, rx_addr, tx_data, and tx_addr widths are parameterizable to a maximum of 32 bits.

Table 5.1: Node Port Interface Signals

Signal Name	Direction	Description
request	output	initiates either a router update request or a connection request
release	output	initiates a connection release
grant	input	indicates to the network module that its connection request has been granted
sl_grant	input	indicates that a connection has been established with this node as a slave
pend	input	indicates that another module is requesting access to the module's current destination port
rx_data[X:0]	input	rx data bus of parameterizable width
rx_addr[Y:0]	input	rx address bus of parameterizable width
rx_rnw	input	rx read-not-write signal
rx_valid	input	indicates valid rx data, address, and rnw signals
rx_cts	input	rx clear-to-send signal
tx_data[X:0]	output	tx data bus of parameterizable width
tx_addr[Y:0]	output	tx address bus of parameterizable width
tx_rnw	output	tx read-not-write signal
tx_valid	output	indicates valid tx data, address, and rnw signals
tx_cts	output	tx clear-to-send signal

The node interface is responsible for issuing router update requests, managing asynchronous boundaries, and coordinating network data transfers. Figure 5.1 identifies the hardware needed to effectively integrate a module (hardware task block) with the network. The router FSM controls the router handshaking signals responsible for issuing router updates and connection requests. The Rx and Tx FIFOs provide support for data flow control. This section describes these components in detail and explains their role in this architecture.

5.1.1 Router Updates

In systems where dynamic module replacement is desired, each module that connects to the network infrastructure must implement a simple router update request sequence. This consists of a simple state machine that issues a router update request

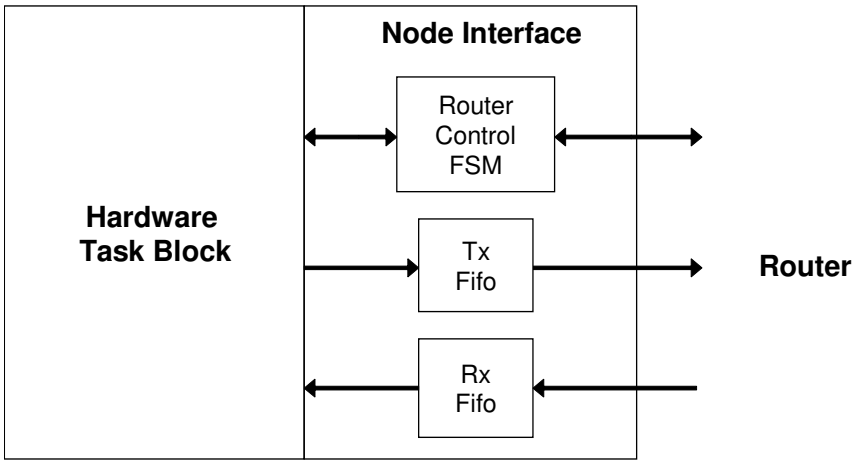

Figure 5.1: Node Interface Hardware

and monitors the *grant* signal to determine when the request has been completed. As shown in Figure 5.2, the router request involves the raising of the *request* signal with *tx_addr* set to 0x00 and *tx_rnw* driven low. The *tx_data* signal must be set to the module's source address – in this example a source address of 0x25 is used. These signals are held until the *grant* signal is raised, indicating the table update has completed.

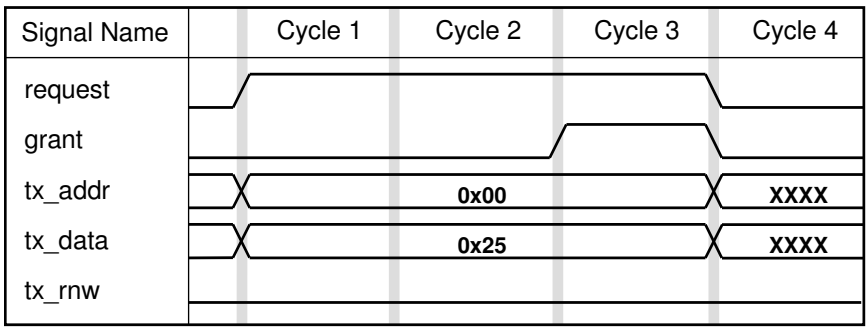

Figure 5.2: A Module's Router Update Request

If dynamic module replacement is not desired, the routing tables may be initialized at design time with the necessary address/port mappings.

5.1.2 Interface FIFOs

Depending on the system timing characteristics, certain network modules may require the use of interface FIFOs. These FIFOs are necessary at asynchronous boundaries, which occur under two conditions:

- **Clock boundaries** - whenever a module runs at a clock rate different from that of its subnet router.
- **Data-rate boundaries** - during data transmission, when the receiving module cannot keep up with the data transmission rate.

This asynchronous flow control is dictated by the CTS signal. During data transmission between modules capable of different data rates, where the transmitting module is the faster of the two, the receiver's FIFO begins to fill. Once the FIFO reaches $almost\ full$ status, the tx_cts signal is lowered and is propagated to the transmitting module's rx_cts signal. In order for this asynchronous flow control to work, the transmitting node must stall until its rx_cts is again raised, indicating that the receiving module is capable of accepting more data.

5.2 Data Transfer Process

This section describes the data transfer process first from the perspective of the master node and then from the perspective of the slave node. An example is then used to illustrate the complete read and write sequences. For simplicity, all examples provided assume the use of a single common clock for all nodes/routers involved.

5.2.1 Master Node Data Transfer

The module responsible for initiating a data transfer, the master node, must first issue a connection request to the router, as shown in Figure 5.3.

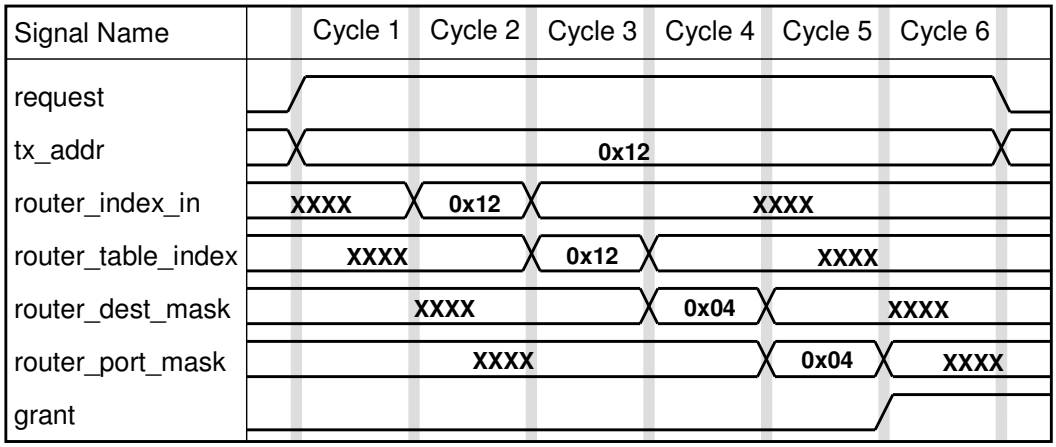

Figure 5.3: A Module's Connection Request

Similar to the router update request, the connection request involves the raising of the *request* signal as the *tx_addr* is set to the target module's network address ($tx\_addr = 0x12$). These signals are held until the *grant* signal is raised. The master node's *grant* signal remains high throughout the duration of the connection. The slave node sees a similar *sl_grant* signal (not shown) so that it knows a connection to it has been established. This connection process has a minimum latency of six clock cycles as shown in the figure. If for some reason a target node is not ready to accept a connection request, it may deny connections by holding its *tx_cts* line low.

Once the *grant* signal goes high, the *request* should be lowered so that data transfer can begin. Data transfers in this network are similar to pipelined memory accesses. Write operations occur when the *tx_valid* signal is raised, the *tx_rnw* goes low, and the *tx_data* and *tx_addr* buses are driven appropriately. Since the data transfer occurs on a dedicated connection path, there is no need for an acknowledge signal or similar handshaking. Also, as long as the *rx_cts* signal is high, writes can occur on consecutive clock cycles.

A read operation occurs when the *tx_valid* and *tx_rnw* signals go high, and the *tx_addr* signal is driven appropriately. Read requests are pipelined, meaning that the request only needs to be presented for one clock cycle and not held until the data is

read back. This allows for reads to occur on successive clock cycles. The data comes back with the *rx_valid* going high, and *rx_data* representing the requested data.

As described in Section 5.1, if data flow control is required then the *rx_cts* signal must be monitored during the data transmission process. Similarly, if connection timeout is to be supported, the master node must also monitor the *pend* signal. The master may either completely ignore it and potentially block other connections indefinitely, or the master may respond after a determined number of clock cycles where the *pend* signal has remained high. In this case, the master must issue a connection release command, and once the *grant* has been lowered, it may then again attempt to connect to the target module. In this manner, shared resources can be effectively time-multiplexed between modules.

5.2.2 Slave Node Data Transfer

Data transfers on the slave end are simpler. No connection process is necessary since that is taken care of by the master node. Data transmission is detected by monitoring the *rx_valid* signal. For write operations the *rx_rnw* is held low, and the *rx_addr* and *rx_data* values are read in and used by the slave node.

Read operations are identified by a high *rx_rnw* signal. The *rx_addr* value is read in and the requested data is produced. Upon availability of the requested data, the slave asserts the *tx_valid* signal and some number of clock cycles later presents the data on the *tx_data* bus.

5.2.3 Illustrative Examples

Two simple examples will be used to show how network write and network read operations are performed. In these examples the nodes involved have already successfully established dedicated connections. Also for this example it is assumed that the data transfer latency between nodes is two cycles, though this will vary based on the number of routers between the modules. First, the write process will be examined. In this example the master node writes the value 0x01 to address 0x22 of

the slave node. In the second example the master node reads from the slave node at address 0x22.

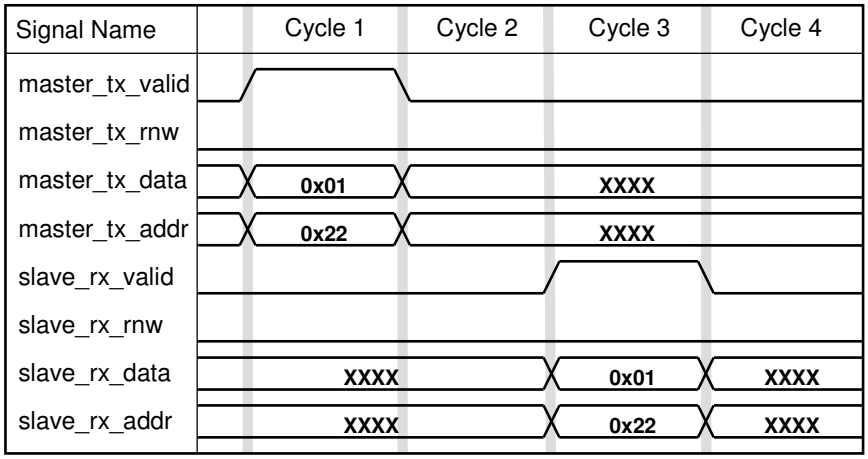

Figure 5.4: Master Node Write Sequence

Figure 5.4 illustrates the write process. At clock cycle 1 the master node initiates the write by raising its *tx_valid* signal, holding the *tx_rnw* low, and presenting the values 0x01 and 0x22 on the *tx_data* and *tx_addr* buses respectively. A few clock cycles later, at clock cycle 3, the slave node receives the write request as its *rx_valid* goes high. At this point the slave's *rx_data* and *rx_addr* values are valid and the data can be read in and used by the slave node.

Figure 5.5 illustrates the read process. At clock cycle 1 the master node initiates the read by raising the *tx_valid* and *tx_rnw* signals and presenting the value 0x22 on the *tx_addr* bus. At clock cycle 3 the slave node receives the read request as *rx_valid* goes high. On the next cycle the slave node responds by asserting its *tx_valid* signal and presenting the requested data, 0x01, on its *tx_data* bus. At clock cycle 6 this data is received by the master node, completing the read sequence.

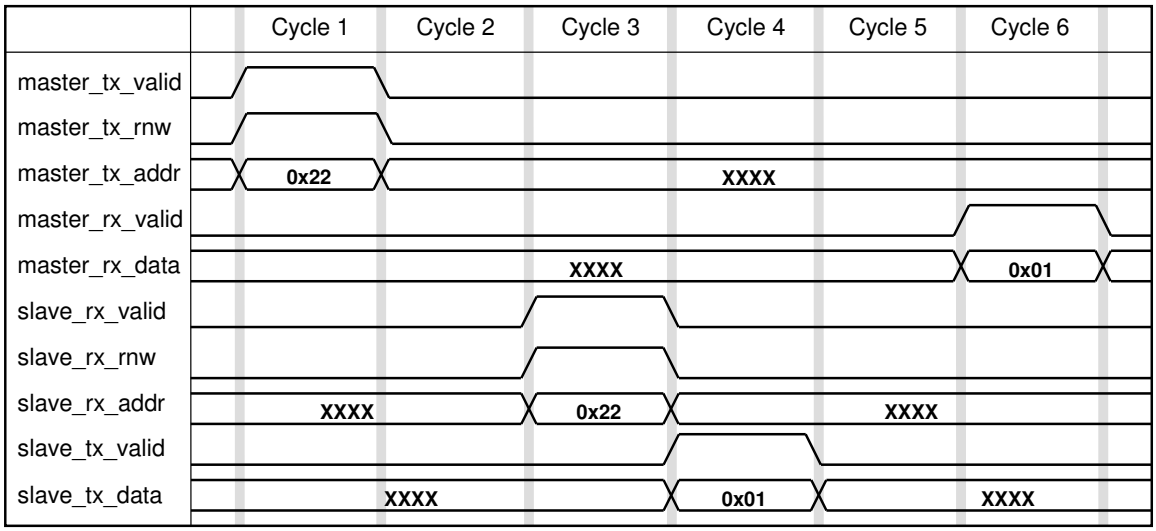

Figure 5.5: Master Node Read Sequence

5.3 Summary

The network modules for this architecture are comprised of two major pieces: the network interface circuitry and the module's functional hardware. The interface circuitry is configurable and may include such things as asynchronous data flow FIFOs or router update state machines. It must include some handling of the network signals required for any data transaction in which the module may be involved. This network port interface is fixed and the handshaking involved is relatively simple, allowing for a straightforward and modular design of the network nodes.

Chapter 6

PNoC CPU Interface

A special type of module that often plays an important role in systems is a general purpose processor. The interface for most processors is generally designed to communicate with memory rather than a network so a special interface is required to effectively attach a processor to this framework. This chapter explains the special interfacing required both in hardware and software that enables the processor's interaction with the network. An example is then provided that illustrates the processor's communication with the network.

6.1 Memory-Mapped Interfacing

When possible, the processor node should contain both the processor and its associated program memory as shown in Figure 6.1. In systems where this is not feasible, the processor node and the memory interface node should reside on the same subnet in order to reduce memory access latency.

Connection of a general purpose processor or CPU to the network infrastructure requires two levels of interfacing: first, interfacing the CPU's memory-based interface to a network-based one, then interfacing that to the network's required port interface. The first interface requires a special memory mapping scheme to access the network. The second interface is just a general network module interface as explained in Chapter 5. These two levels of interfacing are illustrated in Figure 6.1.

There are two memory mappings that are required to effectively interface the processor with the network. The first is a network access mapping and the second is a network data mapping. The network access mapping is used to make connection

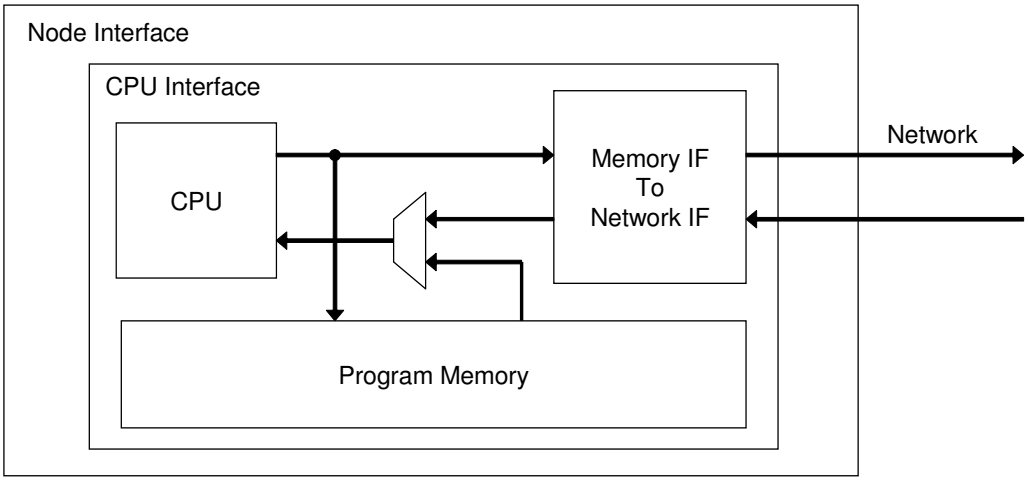

Figure 6.1: Network CPU Interfacing

requests and releases. The network data mapping is used in network-based data transfers. When a read or write request is issued by the processor, a simple address decoding is applied to determine if the access involves a network request/release, a network data transfer, or a local memory access. If the transaction is a network connection request/release then that request is forwarded on to the router and the desired connection is established/terminated. Similarly if the transaction is a network data access, the data request is forwarded to the router and from there on to the slave node. In order for a network data transaction to occur, a network access request must have previously been sent and the connection with the slave node established.

6.2 Network CPU Software

As with all modules, in order to transmit data on the network, the processor must first establish a connection with the target node. For the processor node this connection process is controlled in software. A *network access* pointer that corresponds to the network access memory map is used to initiate and terminate connections. Similarly, a *network data* pointer that corresponds to the network data memory map is used to initiate the network data transfer. The processor can also be accessed as a slave through the use of an interrupt line and corresponding interrupt

service routine. Sections 6.2.1 and 6.2.2 provide detailed examples of how the software accesses the network as master and as slave respectively. These examples are simply used to illustrate how software *may* be used to access the network. Since network access is done through basic memory mapping, the actual software implementation could be done many different ways. The only hard requirements are listed as follows:

- Two memory mappings are needed: one for network access requests, and one for network data transfers.
- Each software memory mapping must match the corresponding memory map used in the CPU network interface hardware.
- A connection request must be completed before a network data transfer can take place.
- Issuing a connection request requires writing the value 1 to the network access address, and a connection release requires writing the value 2.
- When a connection is established between the CPU and another module, a release must be issued before a new connection can be requested.

6.2.1 CPU As Master

As a master node, the processor is responsible for establishing the connection prior to initiating the data transfer. Sample code shown in Figure 6.2 will be used to illustrate how this process occurs in software and how that translates to the hardware interface. As seen in the example code, there are a few $\#define$'s that represent important system parameters.

```
#define NET_ACCESS_ADDR   0xFE000000
#define NET_DATA_ADDR     0xFF000000

#define NET_REQUEST   1
#define NET_RELEASE   2

#define NODE_ADDR     6

int *network_ptr = (int*)NET_ACCESS_ADDR;
int *data_ptr = (int*)NET_DATA_ADDR;

void networkRequest(int node_addr);
void networkRelease(int node_addr);

int main() {
        int read_val;

        networkRequest(NODE_ADDR);
        *(data_ptr+1) = 6;
        read_val = *(data_ptr+2);
        networkRelease(NODE_ADDR);

}

void networkRequest(int node_addr) {
        *(network_ptr + node_addr) = NET_REQUEST;
        while( !*(network_ptr + node_addr) );
}

void networkRelease(int node_addr) {
        *(network_ptr + node_addr) = NET_RELEASE;
        while( *(network_ptr + node_addr) );
}
```

Figure 6.2: Network CPU Interfacing in Software

- **NET_ACCESS_ADDR** - the address that maps to network access requests.
- **NET_DATA_ADDR** - the address that maps to network data requests.
- **NET_REQUEST** - the value representing a request command.
- **NET_RELEASE** - the value representing a release command.
- **NODE_ADDR** - the network address for the target node.

On source code line 9 the *network_ptr* variable is initialized to point to *NET_ACCESS_ADDR*. On line 10 the *data_ptr* is initialized to point to *NET_DATA_ADDR*. Inside the *main*() function the first call at line 18 is a *networkRequest()* function call that makes a request to connect to the target node. The *networkRequest()* function does two things. It first writes a 1 (*NET_REQUEST*) to address 0xFE000006 (*NET_ACCESS_ADDR + NODE_ADDR*). Then it waits in a loop reading from address 0xFE000006 until the connection grant is received.

Figure 6.3 shows what the corresponding interface hardware sees. During clock cycle 1 the *cpu_data_addr* is 0xFE000006, and the *cpu_data_we* line goes high. This is detected as a network access, and since *cpu_data_out* is 0x00000001 the access is decoded as a network request. On the next clock cycle (cycle 2) the request is sent on to the router with the *tx_addr* value of 0x06, as obtained from the low byte of the *cpu_data_addr* bus. These signals are held until the *grant* signal is received from the router indicating that the requested connection has been established. During clock cycle 8, the *cpu_data_re* line is asserted and the *cpu_data_addr* is again set to 0xFE000006. This is decoded as a network access read which enables the reading of the *grant* signal. On clock cycle 8 the *grant* signal is high and read in by the processor on the following cycle ($cpu\_data\_valid = 1$ & $cpu\_data\_in = 1$).

Now that the connection with the target node has been established, the data transfer can be initiated. This occurs at line 19 where a network write is followed by a read. The write consists of a 6 being written to address 0xFF000001. Then the read requests data from address 0xFF000002.

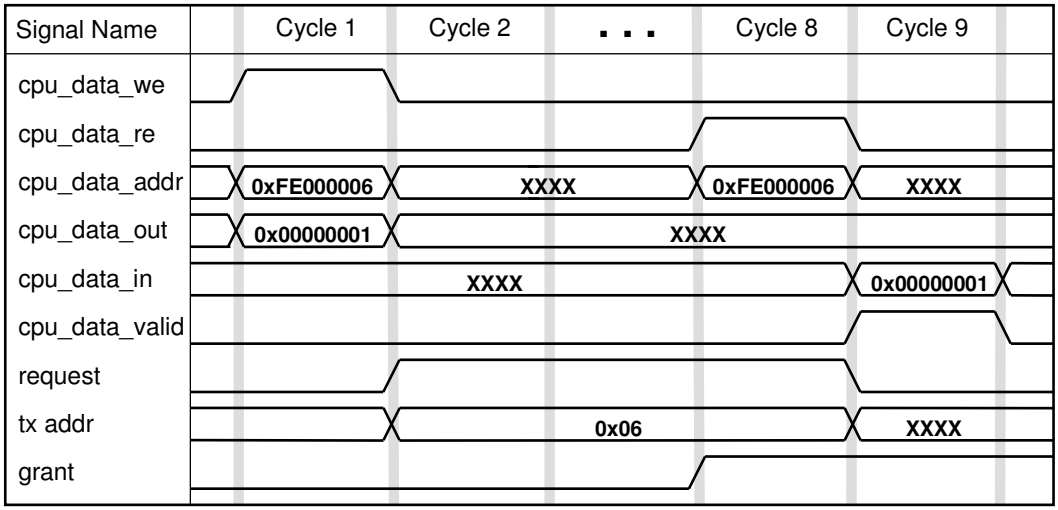

Figure 6.3: CPU Interface - Connection Request

Figure 6.4 shows the corresponding data transaction sequence as it occurs in the hardware interface. During clock cycle 1 the *cpu_data_we* signal goes high, *cpu_data_addr* is 0xFF000001, and the *cpu_data_out* value is 6. This is decoded as a network write of value 6 to network address 0x01. On clock cycle 2 this write request is forwarded to the router ($tx\_valid = 1$, $tx\_addr = 0x01$, $tx\_data = 0x06$). On the following cycle the request is received by the target node. Also on clock cycle 3 the *cpu_data_re* signal is asserted, and *cpu_data_addr* is set to 0xFF000002. This is decoded as a network read from address 0x02. Again this request is sent to the router and from there on to the target node. The node responds during clock cycle 7 with a data value of 6. On clock cycle 8 this value is received by the processor node and can be used by the software.

The final piece of code is at line 21, where the *networkRelease*() function is called. Similar to the *networkRequest*() function, the release consists of two parts. The first is the writing of a 2 ($NET\_RELEASE$) to address 0xFE000006. Then the software waits in a $while - loop$ until the *grant* signal goes low.

Figure 6.5 show what happens in hardware during this release process. During clock cycle 1 *cpu_data_we* is asserted, *cpu_data_addr* is set to 0xFE000006, and the

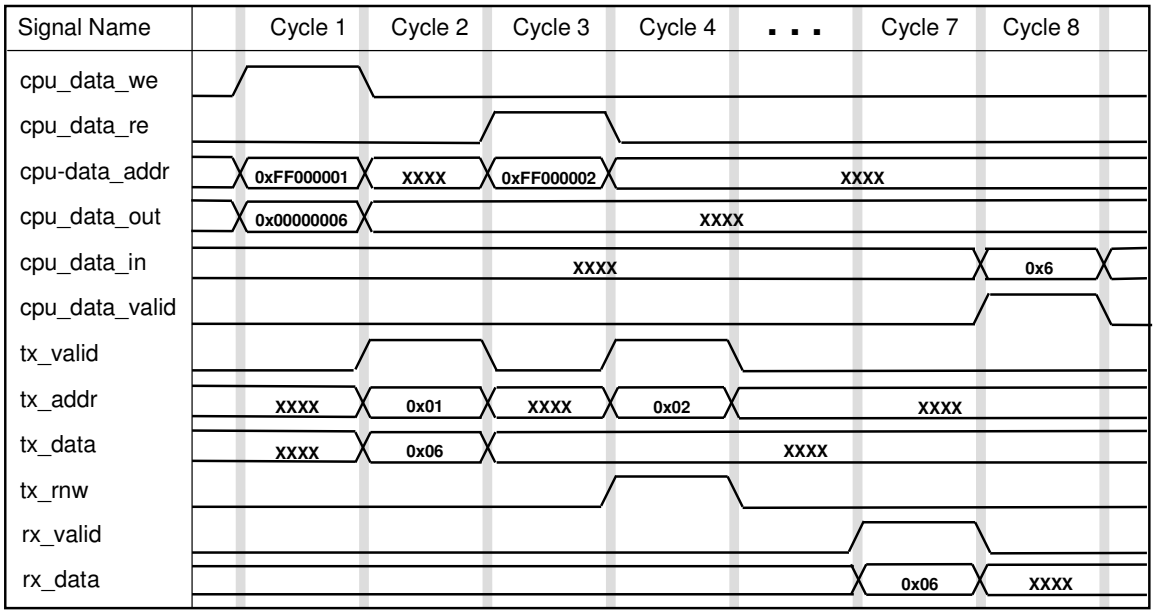

Figure 6.4: CPU Interface - Data Transfer

cpu_data_out value is 0x02. This is decoded as a network release command which is forwarded to the router on the following cycle. On clock cycle 4 the router responds by lowering the *grant* signal. Also on clock cycle 4 *cpu_data_re* is asserted, and *cpu_data_addr* is again set to 0xFE000006. This combination is decoded as a network access read and enbles the reading of the *grant* signal which is read into the processor on the following cycle.

6.2.2 CPU As Slave

The processor can be accessed as a slave through the service of an interrupt. This process occurs when a node on the network establishes a connection with the processor node and then issues a write request while presenting its source address on the *tx_data*. This request gets decoded by the processor interface as an interrupt and the interrupt line is raised. The processor then jumps to the interrupt service routine and from there reads in the master node's source address to determine which node issued the interrupt and responds accordingly.

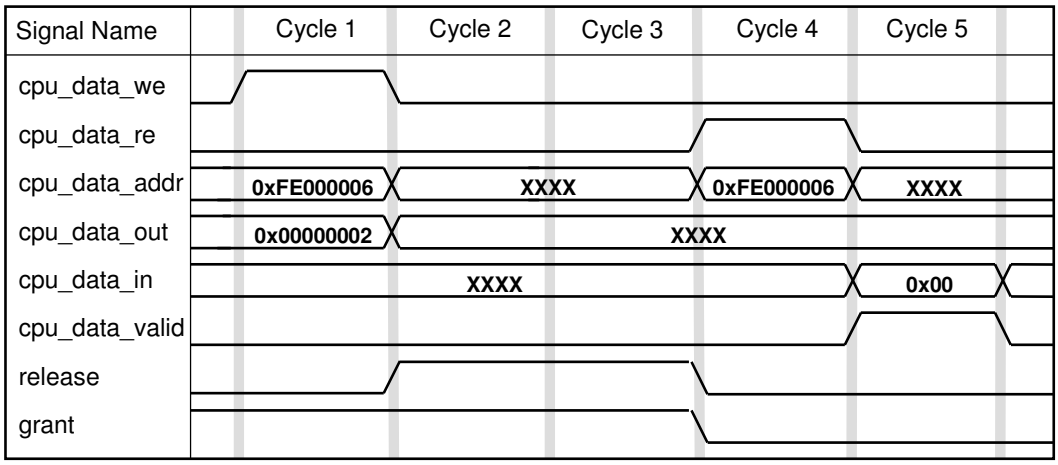

Figure 6.5: CPU Interface - Connection Release

6.3 Summary

General purpose processors often serve an important role in systems. In order to integrate a processor into this network infrastructure, a special interface converts a processor's memory-based interface to a network-compatible one. This conversion process involves two memory mappings: one for network access requests and the other for network data transactions. Both the network accesses and the network data transfers are controlled by the software through memory-mapped pointers.

Chapter 7

PNoC Implementation Results

The PNoC router, module interface, and CPU interface have each been isolated and run through the Xilinx ISE place-and-route tools for analysis purposes. This chapter reports the implementation results for each of these components, each with varying parameter settings. The numbers provided in the tables show the values reported by the Xilinx tools when targeting a Virtex-II Pro device. Also provided is a comparison between the PNoC and a representative packet-switched architecture.

7.1 PNoC Router Results

Table 7.1 shows the implementation results for several different router configurations. The top half of the table represents results for a router with 8-bit data and address widths at each port interface where 2, 4, and 8 port interfaces were used. The bottom half of the table reflects the results for 32-bit data and address widths. In each case 1 block RAM was required for the routing table.

Table 7.1: Router Results

# of Ports	Data/Addr Width	Area (slices)	Speed (MHz)
2	8	83	160
4	8	249	152
8	8	1,113	145
2	32	131	154
4	32	366	141
8	32	1,305	129

As indicated by the table, the size and speed of the router is more heavily impacted by the number of ports rather than the data and address widths. This is due to the fact that all table and port arbitration circuitry is directly related to the number of port interfaces. The number of connection request FIFOs (for connections awaiting availability of the destination port) is equal to the number of port interfaces in the router. As a result, the size of the router grows substantially as the number of ports increase.

7.2 PNoC Module Interface Results

Table 7.2 shows the implementation results for the module interface hardware. The basic configuration includes no circuitry for asynchronous data flow or timeout handling. It consists of a simple state machine for managing connection requests and circuitry for registered inputs. The full configuration, on the other hand, includes circuitry for full functionality: asynchronous data flow, timeout handling, and connection request management.

Table 7.2: Module Interface Results

Data/Addr Width	Basic (slices)	Basic (brams)	Full (slices)	Full (brams)
8	27	0	143	4
16	43	0	147	4
32	75	0	155	4

Full-functionality module interfaces require noticeably more hardware than the basic interface implementations. This is primarily due to the ciruitry required to implement the four asynchronous data flow FIFOs: the Rx data, Rx address, Tx data, and Tx address FIFOs.

Each FIFO is implemented with a dual-ported block RAM, and read and write address counters. The timeout handling hardware is simply a counter that counts the duration of an asserted *pend* signal.

7.3 PNoC CPU Interface Results

Table 7.3 shows the implementation results for the CPU interface hardware. Again the basic configuration includes no circuitry for asynchronous data flow or timeout handling. It consists of circuitry for converting CPU memory-mapped accesses to the corresponding network accesses. Again, the full configuration includes the circuitry required for asynchronous data flow control and timeout handling. These values are based on the results for a 32-bit processor.

Table 7.3: CPU Interface Results

Configuration	Slices	BRAMs
Basic	102	0
Full	190	4

In addition to the hardware required for basic module interfacing described in the previous section, the CPU interface also contains address decoding circuitry to identify network accesses verses normal program memory accesses. It also contains special hardware to handle the network-module interrupt requests.

7.4 Network Architecture Comparison

Unfortunately, few authors have published implementation results for their proposed network-based architectures. As a result, it is difficult to perform direct comparisons with other network approaches. One that has done so, however, and appears to be representative of other packet-switched approaches is presented in [19].

In order to make a meaningful comparison between their architecture and the PNoC architecture a few assumptions had to be made:

- The system under comparison consists of 8 network modules targeted to a Xilinx Virtex-II Pro device, where the packet-switched architecture uses distributed routers while the PNoC architecture uses a single central router. The communication between nodes consists of 16-bit data channels. These systems are illustrated in Figure 7.1.

- Based on the packet-switched topology shown in the figure, the system requires 4 3-port routers and 4 4-port routers. According to the implementation results published in [19], a 3-port router requires 250 slices and a 4-port router requires 350 slices. It is also assumed that each router also uses 1 block RAM for the output buffers.

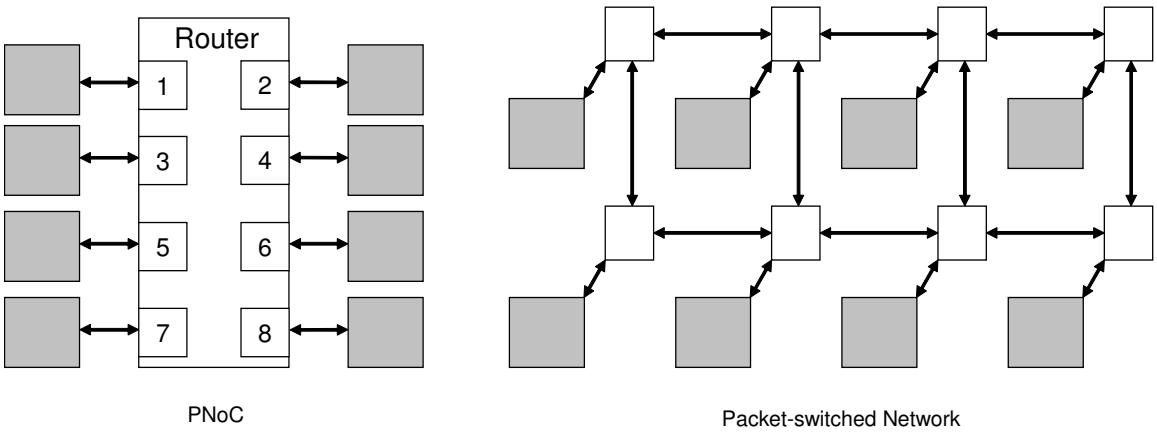

Figure 7.1: Network Architectures

Table 7.4 shows how the PNoC architecture compares with this packet-switched architecture based on the assumptions previously mentioned. The table compares resource usage and maximum clock rates for the systems' routers.

Table 7.4: Network Comparison Results

Network Architecture	Slices	BRAMs	Clock Rate
Packet-Switched	2,400	8	50 MHz
PNoC	1,223	1	134 MHz

These results reflect the complexity required in packet-switched networks to process, route, and buffer packets. The simplicity of the circuit-switched PNoC architecture not only reduces the hardware costs by over 2×, in this example it also increased the clock rate by almost 3×, resulting in an area×time improvement of over 5×.

7.5 Summary

One of the goals of this work was to create a lightweight communication framework. The results presented in this chapter illustrate the overhead required for using the PNoC architecture. The comparisons made against a representative packet-switched architecture suggest that the PNoC architecture is both lightweight and capable of relatively fast clock rates.

Chapter 8

PNoC Test Applications

Two example applications have been created to illustrate the effectiveness of this network architecture. Each has been implemented using both the PNoC architecture and the Xilinx EDK [6], which consists of a shared bus architecture based on IBM's CoreConnect [4]. This chapter explains the test applications and their implementation on both platforms. Also a comparison of the performance results are analyzed.

The first application involves an autonomous robot with on-board vision. The objective of this application is to integrate a robot soccer defense algorithm on the robot so that it searches for the soccer ball with the on-board vision. Once the ball is located, the robot quickly moves to the ball in attempt to clear it away from the goal. The implementation of this application involves two nodes that access a shared memory. As the camera writes a new video frame to memory the processor reads from memory the previously stored frame to decide what action to perform. Since there is not a significant amount of intra-node communication, this application should work equally well on both platforms.

The second application is an image binarization algorithm that uses hierarchical thresholding to segment the image data. The binarization is used to compress the image data while mimimizing the amount of important information lost in the process. This algorithm requires a significant amount of concurrent data processing and bandwidth between the computation modules. This algorithm is much more suited to a network architecture since multiple data transactions can occur simultaneously.

In both applications a single global clock is used since the systems are relatively small. For the PNoC implementation this eliminates the need for module interface FIFOs since all transactions occur on the same clock. Also, since no dynamic module replacement is used in either application, the PNoC's router configuration is simplified as no table update circuitry is needed.

8.1 Autonomous Robot

The autonomous robot used in this application consists of a control board that rests on top of a three-wheeled base. Attached to the board is a small video camera to provide on-board vision. The control board uses a Virtex-II 1000 FPGA (part: xc2v1000-4) that is responsible for controlling both the wheel motors and the camera. The objective of the robot is to identify the location of a white ball and keep it away from the robot's goal. Once the ball has been identified, the robot activates the motors and drives towards the ball in attempt to direct it away from the goal.

The autonomous robot application involves the use of four top-level modules that interface to the connection medium as shown in Figure 8.1.

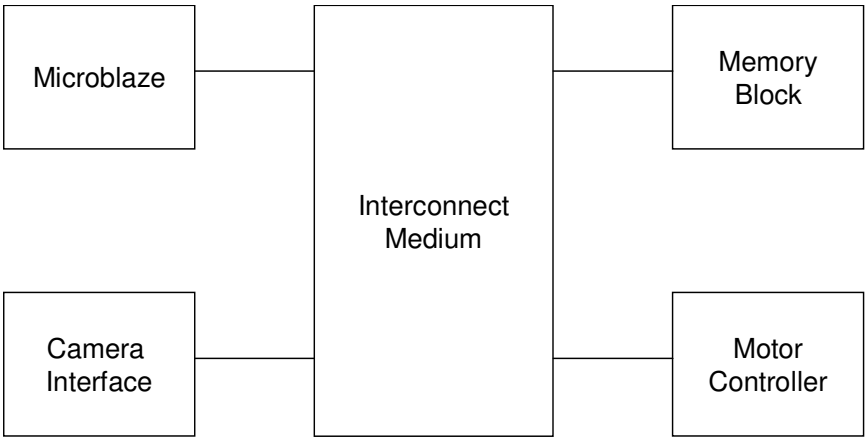

Figure 8.1: Robot Top-Level Modules

- **Microblaze Processor & Local Memory** - this module serves as the primary control for the system. It consists of a Microblaze processor along with its associated program memory. The software is responsible for sending commands to the motor controller to control the movement of the robot. Also a simple ball tracking algorithm is implemented in the software that runs on this processor.

- **On-chip Memory Block** - this module is a block RAM-based memory in which the video frame data is stored.

- **Camera Interface** - this module interfaces the FPGA logic to the on-board video camera. Hardware is also built into this module that captures video frames as directed by the software, performs intensity segmentation, and then sends the segmented frame to the on-chip memory block.

- **Motor Controller** - this module serves as the interface between the FPGA and the robot's wheel motors. A hardware PID loop is part of this module to provide precise control of the robot's movement.

8.1.1 General Implementation Details

Control for the system originates with the processor, which begins by issuing a frame-capture request to the camera interface module. Once the camera module receives this command, it begins a single frame capture. As the pixels are received, the camera module extracts only the luminance value (the Y value in the YCrCb video data) and performs segmentation based on this intensity value. All pixels with intensity values over a specified threshold are segmented to a 1 while all other pixel values are set to 0. These segmented values are then sent to the on-chip memory block. After the entire frame has been received, segmented, and then loaded into memory, the camera module raises its *done* status bit.

The processor, which polls on the *done* status bit, initiates another frame-capture request and begins processing the frame previously loaded in memory. Double buffering is used to allow the software to process a frame of video data while the next frame gets captured. The software uses vertical projection to locate the white ball.

Since the ball is white, it should be the dominant feature in the segmented image. Each column of the current frame is summed, and the column with the greatest sum represents the center of mass for the ball in the horizontal direction. This information is used to determine the direction the robot must navigate to center the ball in its field of view. Once that is done the processor sends the appropriate commands to the motor controller to move forward towards the ball. If the ball is not detected in the current frame, the robot pivots to get a new field of view and awaits the arrival of the next frame.

The main design challenge in this system is in coordinating accesses to the shared memory while maintaining the data-rate necessary to keep up with the desired video frame rate. This memory sharing is accomplished in slightly different ways for each architecture.

8.1.2 Shared Bus Implementation

The bus-based implemenation was done using the Xilinx EDK version 6.3 and contains a Microblaze processor and OPB bus running at 75 MHz. In this implementation both the processor and camera module access memory through a series of single-word bus transactions as needed. If ever the two require access to the memory simultaneously, the bus arbiter assigns priorities to the requests, grants bus access to that of highest priority, and forces the other to stall until the granted transaction is completed. Since the required bandwidth for this system is limited by the slow video frame rate (30 frames per second) there is ample time for sharing of the bus and memory without performance degradation or data loss.

8.1.3 Network Implementation

In the PNoC implementation the entire system runs off a single 75 MHz clock. The Microblaze processor accesses memory through a series of single-word network transactions as requested by the software. The camera module on the other hand, first buffers up 16 bytes of segmented pixel data before establishing a connection and transferring the data to the memory node. This is done to reduce the network

connection setup overhead and make more efficient use of the communication medium. If the processor requires access to the memory while the camera module is bursting the pixel data, the processor's request stalls and waits for the camera module to terminate the connection.

8.1.4 System Comparisons

Both system implementations were able to keep up with the video frame rate and perform the desired task equally well. There were slight differences in the amount of hardware resources required and maximum clock rate allowed for each system. Table 8.1 shows the number of slices required for each of the system modules and illustrates the area and speed differences between the two implementations. These numbers were obtained from the Xilinx ISE place-and-route tools.

Table 8.1: Robot System Comparison

Parameter	Shared Bus	PNoC
Microblaze	565	565
Camera Interface	246	246
Motor Controller	441	441
Communication Overhead	305	526
Total Slices	1,557	1,778
Maximum Clock Rate	78 MHz	88 MHz

As shown in this table, the network architecture does introduce some overhead in terms of hardware resources (14% increase), but provides a 13% faster clock due to the shorter wire delays. Most of the PNoC resource overhead came from the router, which required 488 slices. Since the system was run off a single clock, no module interface FIFOs were needed, therefore reducing the module interface overhead. The design times for each system were very similar since both the bus and PNoC architectures each provide a standard interface to the communication medium.

8.2 Image Binarization

The image binarization involves the implementation of an algorithm that requires much higher bandwidth utilization than the robot application. Again, this implementation was done using both the shared-bus architecture and the PNoC architecture. The image binarization algorithm, developed specifically for this work, uses hierarchical thresholding to quantize a grayscale image to binary black and white values. This type of algorithm is used to compress, and often clean up noisy hand written documents that have been digitized for archival purposes. The objective of the algorithm used here was not intended to provide the best binarization. Instead it was created for the purpose of illustrating important advantages of the PNoC architecture. It was used because of its highly parallel structure that maps nicely to a hardware implementation that provides high bandwidth capabilities. It involves computing median values at three different levels of hierarchy to be used as quantization threshold values. These levels are identified as follows:

- **Global level** - the highest level, uses the pixel data of the entire image.
- **Block level** - the middle level, uses the pixel data of a partitioned section of the image.
- **Window level** - the lowest level, uses the pixel data of a single window.

For certain types of documents or images a simple global thresholding is effective, however most images that contain considerable noise or inconsistent background intensities require a more sophisticated algorithm. This algorithm maintains the simplicity of the global thresholding algorithm but also incorporates local image information before performing the final quantization. This algorithm involves the following steps:

1. Compute the median value for the entire image and use that to compute the global threshold value where $global\_thresh = median + median/4$.

2. For each block of data, determine its darkest pixel value and compare that against the global threshold. If it is lower (darker) than the threshold, then that block contains valid data and is processed further (step 3). Otherwise the entire block of data is set to a white value.

3. A valid block is divided into smaller windows and each window is then quantized based on the window's threshold value. Each pixel that is darker than the threshold is set to black – all others are set to white.

4. Steps 2 and 3 are repeated until every block has been processed and the complete quantized image has been produced.

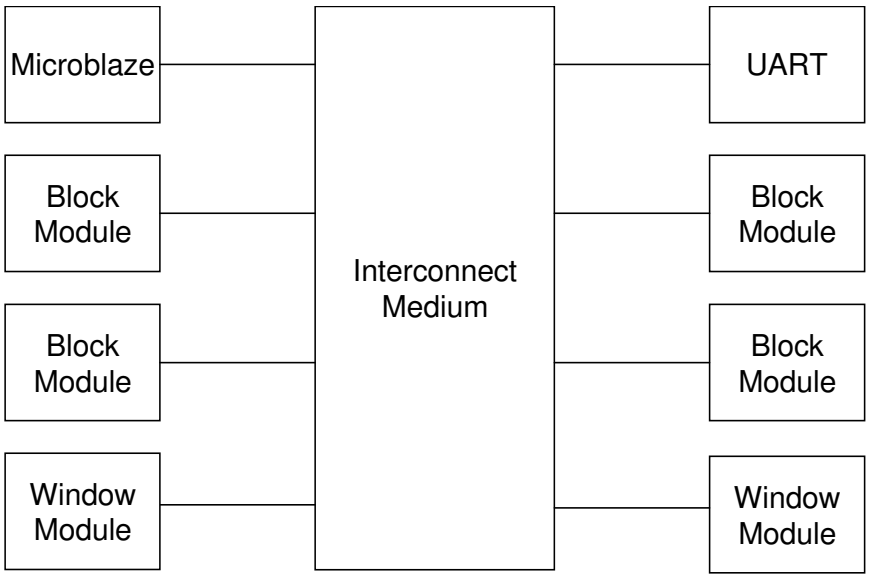

Figure 8.2: Binarization Top-Level Modules

This application, targeted to a Virtex-II Pro FPGA (part: xc2vp30-ff860-7), is illustrated in Figure 8.2 and consists of four different module types. As shown in the figure, there are a different number of block and window modules (4 block modules &

2 window modules). This is due to the projected need for each in the overall system computation. The module types and their functional descriptions are as follows:

- **Microblaze Processor & Program Memory** - this module serves as the primary control for the system. It is responsible for computing a global threshold value for the entire image and then manages the distribution of the image blocks to the block thresholding modules.
- **UART** - this module is used to enable the uploading and downloading of the original and final images between the FPGA and a host computer via the RS232 connector.
- **Block Thresholding Modules** - these modules compute block-level threshold values, and if valid data is detected within a block, smaller windows of the data are sent to the window processing modules.
- **Window Processing Modules** - these modules quantize each pixel of the window based on the window's threshold value.

8.2.1 General Implementation Details

The binarization algorithm gets mapped to this system as follows:

1. The processor, controlled by software, reads in the original image data via the UART.
2. As the data is received, the global median and threshold values are computed.
3. The image is divided into blocks and these blocks, along with the global threshold value, are sent to available block nodes one at a time.
4. Each block node computes its own local minimum, median, and threshold values which are used to identify valid blocks and later valid windows.
5. If the block node detects valid data within the block then that block is further divided into windows. The block node forwards these windows, along with the

block's threshold value, to an available window node. If no valid data is found in a given block, all of its pixel values are set to white and read back by the processor.

6. Each window node computes its own local minimum, median, and threshold values. The minimum value is compared against the parent block's threshold value to determine if the window is valid. If the window is valid each pixel is quantized against the window threshold value. Otherwise all pixels are set to white.

7. Once the window node has completed its quantization process, the quantized values are sent back to the parent block node.

8. Once the block node has processed each of its windows, the quantized results are read back by the processor.

9. Once the processor has received the quantized results for each block, the final quantized image is sent out the UART.

For this particular experiment block sizes of 2,400 bytes were used with 10×15 (150 bytes) windows. When a block is found that contains valid data, each window is sent to the window processing module. The quantized window data becomes valid one cycle after the last byte of has been received by the window module. The quantized values are then read back to the block module. As a result each window processed requires 300 data transactions (150 writes to and 150 reads from the window module). Complete block processing therefore requires 4,800 data transactions (2,400 total writes and 2,400 total reads).

The main design challenge in this system is in coordinating the transfer of image data between the different hierarchical nodes. Also, since there is a different number of block nodes verses window nodes, there must be some arbitration or scheduling involved to allow a block node access to an available window node when needed. These issues are addressed in significantly different ways between the two architectures.

8.2.2 Shared Bus Implementation

Two different bus-based implementations were completed using Xilinx EDK version 6.3. Each uses a Microblaze processor and OPB bus running at 100 MHz. For both implementations, all data transactions must be time-multiplexed since all modules share a common communication medium. There are two extremes as to how this may be done:

- **Single-beat Transfers** - The reads and writes are simple single-word transactions. This is easiest to implement, but doesn't make effective use of the available bandwidth since bus arbitration cycles precede each data transfer.
- **Burst Transfers** - Bus arbitration occurs only once for x number of words to be transferred. This can result in much higher throughput, but since the shared bus must be locked, no other modules can communicate on the bus during the burst.

The first bus-based implementation uses single-beat transfers. It results in a considerably slower system, but allows other modules to communicate on the bus without considerable delays. The second implementation uses burst transfers that lock the bus during the entire window processing phase. This results in a much faster system, but stalls all other pending accesses to the bus.

In both implementations, and in bus architectures in general, there is no built-in way of arbitrating access to the available window modules. This module-level arbitration must be incorporated into the modules themselves. For this experiment, both implementations were done using simple static scheduling of the window modules. Each window module is shared by two predetermined block modules.

8.2.3 Network Implemenation

The PNoC implementation was completed with the use of a single clock running at 100 MHz. The PNoC architecture is well-suited to this type of system. Multiple block-to-window module data transfers can occur simultaneously as multiple connections can be active at a given instant. All transactions are essentially

burst transactions, but due to the networking infrastructure, routing resources remain available for other system modules to communicate.

The dynamic routing capability of this network also plays an important role in this system. When a block module requests a connection to a window module, that connection can be established with whichever one becomes available first. No additional hardware is required by the system designer to poll for available window modules – the choice of which module to use is made by the router. Further, if none of the window modules is available, the connection request is placed in a queue until the module becomes available. This allows for significant system flexibility – additional block and/or window modules can be added to the system without requiring any modification to the block or window modules.

8.2.4 System Comparisons

The designs were downloaded to the Xilinx XUP Virtex-II Pro Development Board. A simple hardware timer was incorporated into each of the designs to count the number of clock cycles required by each system implementation. Each were set up in such a way as to remove the software overhead from the computation time. The four block modules were loaded, and then the computation/communication of the block/window processing was timed using the hardware timer. This was done to analyze the system performance during maximum data transfer, where all four block modules contend for the services of the two window modules. As mentioned earlier, each valid block requires 4,800 data transfers. For this experiment, that implies 19,200 total data transfers. Table 8.2 shows how the implementations compared in terms of maximum clock rate, hardware utilization, and the execution time required for processing the four blocks.

As illustrated in the table, though the PNoC implementation introduces a 29% area increase verses the bus implementations, its performance is considerably better. The single-beat version of the bus implementation requires 20× more cycles for execution than the PNoC implementation. This occurs as a result of the bus arbitration that takes place for each data transfer, resulting in around 10 cycles per

Table 8.2: Binarization System Comparison

Parameter	Shared Bus (single-beat)	Shared Bus (burst)	PNoC
Microblaze	565	565	565
UART	52	52	52
Block Module	148x4	148x4	148x4
Window Module	627x2	627x2	627x2
Communication Overhead	389	401	1,222
Total Slices	2,852	2,864	3,685
Maximum Clock Rate	108 MHz	98 MHz	124 MHz
Execution Cycle Count	198,113	20,919	9,977

transaction. By allowing the block modules to lock the bus and burst the data during window processing, the difference is reduced to about 2×. This factor of 2× is a direct result of the PNoC's ability to support multiple simultaneous data transfers (in this case 2, one for each window module). The locking of the bus in the burst implementation prevents any commuincation between the CPU and other system modules. The PNoC architecture, on the other hand, provides both high throughput and open CPU communication.

Also of note is that the PNoC's clock rate is 15% and 27% faster than the bus implementations. This is consistent with the results seen in the robot system implementation and is a result of the shorter wire delays found in the network architecture. Taking into account the maximum clock rates results in overall PNoC speedups of 23× and almost 3× verses the single-beat and burst bus implementations respectively.

These results also illustrate the channel utilization efficiency of the PNoC architecture. Both the single-beat and burst bus implementations involved a single channel and attained 10% and 92% effective utilization. The PNoC implementation on the other hand, involved the use of two channels and achieved 96% utilization on each.

The design simplicity for the PNoC implementation and the single-beat bus implementations were very similar. Implementing the burst implementation however,

was a little more complicated. It required the use of some additional bus signals that enable the master's locking of the bus and bursting of data. Similarly some additional design had to be incorporated into the slave modules as well. They had to be modified to monitor these additional bus signals and respond differently to accommodate the burst transfers. They had to be specially designed to handle both burst and single-beat data transfers, whereas in the PNoC both burst and single-beat data transfers are handled in the exact same manner.

8.3 Summary

Two applications were used to compare the PNoC architecture against a widely used bus architecture. The first application involved an autonomous robot system, which was a relatively small system that required low-bandwidth intra-module communication. In this system two modules accessed a shared memory for the acquisition and processing of video frames. Due to the low-bandwidth requirements for the system, both the bus and PNoC implementation were equally effective in performing the desired task.

The second application involved a slightly larger system that had much higher bandwidth requirements. Four block modules required the services of two window processing modules to perform image binarization. In this experiment, 4,800 data transfers were needed by each block module to complete the task. Two different bus-based implementations were completed: a single-beat data transfer version and a bursting version. The single-beat implementation was much slower but allowed for open CPU communication. The burst implementation was much faster but locked the bus, temporarily preventing any other intra-module communication. The PNoC implementation provided both high-throughput data transfer while at the same time allowing open CPU communication. Even against the bursting bus implementation the PNoC provided almost a 3× performance improvement due to its use of multiple channels, higher clock rate, and more efficient channel utilization. Analysis of these experiments indicate the PNoC's viability as an effective alternative to bus architectures.

Chapter 9

Conclusion

9.1 Summary

The increase in chip densities has increased the potential for systems to be integrated onto a single chip, commonly referred to as *System on Chip* (SoC). Today these systems are capable of integrating one or more general purpose processors with numerous other system hardware modules. As chip densities continue to increase these systems will similarly continue to grow in size and complexity. One of the major design challenges that results is in maintaining effective communication between the system modules.

Shared bus architectures are common in current systems that involve processor-centric control. These architectures often provide the processor with a memory-mapping scheme to access the system modules as bus peripherals. Bus architectures have been found to be effective in their role as a system communication medium. Modular-based design can be used, which allows for relatively short and more predictable design-time. The major weaknesses of the shared bus architecture are its limited bandwidth and poor scalability. Only two modules can communicate on the bus at a given instant and, as more modules are integrated into the system, the bus becomes a signficant performance bottleneck.

Several network-based architectures have been proposed with the arguments that they significantly improve scalability, provide higher bandwidth capabilities, and maintain a modular design flow. Nearly all of the network architectures researched

suggest a packet-switched communication framework. The weaknesses of these networks is in the overhead required to process, route, and buffer the data packets. A significant amount of hardware resources are needed to perform these tasks, and in some cases it has been acknowledged that their networks had difficulty with heavy data flow applications from a performance standpoint.

This work proposes another alternative that involves a lightweight, flexible circuit-switched rather than a packet-switched network architecture. Circuit-switching, in the context of this work, involves the establishment of dedicated connections between system modules so that unimpeded data transfer can occur. Once the desired data transfer is complete for a given connection, that connection is released and other modules can then request access to the available resources.

This circuit-switched architecture provides some important advantages to the packet-switched networks. Since data is transferred only after a dedicated connection has been established, there is no need for the formation and buffering of packets. The raw data can simply be forwarded along the connected route. This reduces the communication overhead as packet processing is not required. Implementation comparisons were made between the PNoC and a representative packet-switched architecture. These comparisons illustrated the relative simplicity of the PNoC architecture and strengthen its claim as a fast, lightweight network architecture.

A considerable amount of flexibility has been incorporated into this architecture. The system designer has full control of the system connectivity and the network topology. Also built into this architecture is support for dynamic module replacement. Dynamically configured modules can issue update requests to the routers so that the routing tables maintain updated address/port associations.

9.2 Conclusions

The proposed architecture, designed using JHDL, was used in the implementation of two test applications. The first application was an embedded system for the control of an automomous robot. The second involved a data processing algorithm

that performs image binarization. Both systems were also implemented using the Xilinx EDK, which uses a shared bus architecture.

Analysis results for the robot test application showed that both architectures were equally effective for the low-bandwidth system implementation. The network architecture allowed for a slightly higher maximum clock rate but also required a few more resources as part of the routing overhead.

The second, higher-bandwidth application clearly favored the PNoC architecture. In the PNoC-based system the maximum clock rate was higher and the execution cycle count considerably lower than for the shared bus implementations. This was due to the network's ability to allow for multiple simultaneous data transfers.

These tests indicate that this circuit-switched network is capable of matching the performace of a bus architecture for simple embedded systems without incurring significant hardware overhead. It has also proven to be superior to the bus architecture for applications that require considerable system bandwidth and scalability. This work has successfully produced a flexible, and relatively lightweight circuit-switched architecture that allows for high-throughput data transfer between the modules that are critical in the overall system performance.

9.3 Future Work

An important continuation to this work involves exploring the use of multiple routers and subnets, and understanding the kinds of topologies that map well to different system communication requirements. Then, formally testing it against other proposed network architectures could be done to verify its relative effectiveness for large-scale system design. This is currently a difficult task since most of the proposed architectures are not readily accessible. Though, that may soon change as the demand for scalable communication continues to increase.

Another area of future work that may prove valuable is further develpment of the PNoC's dynamic module support. This may involve the design of a tool that aids in the creation of module-based partial reconfiguration bitstreams. This would allow for the creation of systems in which the requirements change over time and where

dynamic module replacement can be used to make more efficient use of the available resources.

Appendix

Appendix A

PNoC Tutorial

Presented here is a tutorial that walks through the process of building a simple *Hello World* example system. This system is created using JHDL and consists of a Microblaze processor node and a UART node that are connected to the PNoC router as shown in Figure A.1. This tutorial first describes how the hardware for this system is created. Then a corresponding *Hello World* software program is explained. This example simply illustrates one of many ways that such a system might be implemented.

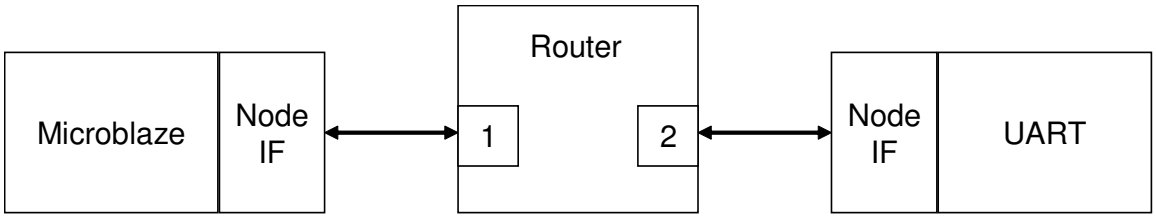

Figure A.1: Hello World System

A.1 Hello World Hardware Design

This hardware design tutorial assumes that both the Microblaze and UART cores are complete and ready to integrate into the PNoC system. A $NetInterface$ class has been created to simplify PNoC module integration. Though the $NetInterface$

is used in the creation for each of the modules in this example, its use is not required. A module designer could manually implement all of the required interfacing hardware for one or more of the system modules. This section describes the $NetInterface$ class and explains how both the Microblaze and UART nodes use it to interface to the network. Also explained is the creation of the top-level hardware system.

A.1.1 The NetInterface Class

The $NetInterface$ class manages all of the node's top-level port interface signals and provides an abstracted set of signals to the module, simplifying its integration to the network. This interface can be configured (by boolean flags passed to its constructor) to include Rx and Tx interface FIFOs, timeout detection circuitry, a router update FSM, and registered inputs and outputs. PNoC modules simply extend this class to take advantage of its features.

A.1.2 Network Module Design: A UART Example

A $UartNode.java$ file is provided in A.4.1 that illustrates how the UART core might be used with the $NetInterface$ to interface to the network. As shown in the code, the $UartNode$ extends $NetInterface$ and passes all the top-level port signals to it through the $super()$ constructor call. The $NetInterface$ connects all of the top-level network port signals, but the UART-specific rx and tx signals must be separately added to the port interface and connected from within the $UartNode$.

The $UartNode$ reads in the rx signals provided by the $NetInterface$ to interpret what commands to issue to the UART core. For example, in this system, a send request is defined by a write to rx_addr = 0x00, where the rx_data is the value to be sent. Similarly, a receive request is defined as a read from rx_addr = 0x00, while a read from rx_addr = 0x01 is interpreted as a status read request. The requested data is connected to the tx_data bus.

Since the $UartNode$ never needs to act as a master node, its $request$, $release$, tx_addr, and $dest\_addr$ lines are tied to ground.

A.1.3 Network CPU Design: A Microblaze Example

The *MicroblazeNode*, shown in A.4.2, also extends *NetInterface*, which provides its top-level interface to the network. However, as described in chapter 6, there is another level of interfacing that is needed. *MicroblazeNode* instances a module called *NetMicroblaze*, shown in A.4.3, which is responsible for converting the Microblaze memory-based interface to a PNoC compatible one. It is in the *NetMicroblaze* class that the network address decoding takes place, which identifies network accesses and network data transfers.

NetMicroblaze instances both the *Microblaze* core and its associated program memory, *LocalMemory*. Since *Microblaze* is not available in a JHDL implementation, it is instanced as a blackbox, shown in A.4.4. Also included in the *NetMicroblaze* class is circuitry that drives the network connection *request* and *release* lines as controlled by the software program.

A.1.4 Top-Level System Design

The top-level system file, *HelloWorld.java*, is shown in A.4.5. It is in this file where the system parameters are defined, the system clocks are generated, and the top-level router and network nodes are instantiated. This system has a *port_count* of 2 since there are only 2 nodes that connect to the network: the Microblaze processor and UART. The data and address widths for the Microblaze are both 32 bits, while the data and address widths for the UART are 8 bits and 1 bit respectively.

A DCM is used to generate the clocks used in this system. For this example, the *MicroblazeNode* and *NetRouter* run at the native clock rate of 100 MHz, while the *UartNode* runs at 50 MHz. Therefore, the *clk*0 output of the DCM drives both the *net_clk* and the *cpu_clk*, while the *clk_dv* drives the *uart_clk*.

As shown in A.4.5, the *NetRouter* is configured without the use of table update hardware since dynamic module replacement support is not needed in this system. Instead, the table is initialized with the values provided in the *table_init* array, which map the *MicroblazeNode* address 1 to port interface 1 and the *UartNode* address 2 to port interface 2.

The *MicroblazeNode* instantiation includes the network access and network data mask parameters. For this system the *net_access_mask* is 0x80000000 and the *net_data_mask* is 0xC0000000. The *MicroblazeNode* is assigned a source address of 1 and does not require the use of interface FIFOs since it runs at the same clock rate as the *NetRouter*.

The *UartNode* is assigned a source address of 2 and does require the use of interface FIFOs since it runs at a clock rate different from that of the *NetRouter*.

A.2 Hello World Software

The software that runs on this system is shown in A.4.6. The NET_ACCESS_MASK and NET_DATA_MASK values are consistent with those used in the top-level hardware file (*HelloWorld.java*). Network macros (*mNetworkRequest*, *mNetworkRelease*, *mNetworkWrite*, and *mNetworkRead*) are used to access the network. The function of this program is to print the string "Hello World!" to the UART to be received and read by a host computer. To do this the software first requests access to the UART by issuing a network request addressed to the *UartNode*. Once access is granted, the *printString*() function sends the desired string, one character at at time. The connection is then released as the software issues a network release command.

A.3 Building the System

The details of building such a system are device and processor specific and are not discussed in detail here. However, the hardware and software makefiles used to build this system, targeted to a Xilinx Virtex-II Pro device (xcv2p30), are provided in A.4.7 and A.4.8 respectively to be used as a reference.

A.4 Hello World Source Code

A.4.1 UartNode.java

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class UartNode extends NetInterface {

  public UartNode( Node parent, String name,
                   Wire net_clk, Wire node_clk, Wire reset,
                   Wire net_request, Wire net_release,
                   Wire net_grant, Wire net_pend,
                   Wire net_rx_data, Wire net_rx_addr, Wire net_rx_rnw,
                   Wire net_rx_valid, Wire net_rx_cts,
                   Wire net_tx_data, Wire net_tx_addr, Wire net_tx_rnw,
                   Wire net_tx_valid, Wire net_tx_cts,
                   int src_addr, int timeout_val,
                   Wire rx, Wire tx,
                   int clock_rate, int baud_rate ) {

    super( parent, name,
           net_clk, node_clk, reset,
           net_request, net_release, net_grant, net_pend,
           net_rx_data, net_rx_addr, net_rx_rnw, net_rx_valid, net_rx_cts,
           net_tx_data, net_tx_addr, net_tx_rnw, net_tx_valid, net_tx_cts,
           src_addr, timeout_val );

    addPort( out("tx", 1) );
    addPort( in("rx", 1) );

    connect( "rx", rx );
    connect( "tx", tx );

    gnd_o( dest_addr );
    gnd_o( request );
    gnd_o( release );

    gnd_o( tx_addr );
    gnd_o( tx_rnw );
    vcc_o( tx_cts );

    Wire read_status = and( rx_valid, rx_rnw, rx_addr.gw(0), "read_status" );
    Wire read_en = and( rx_valid, rx_rnw, not(rx_addr.gw(0)), "read_en" );

    Wire enable = and( rx_valid, not(rx_rnw), "enable" );
    Wire din = regce( rx_data, enable, "data" );
    Wire send = reg( enable, "send" );
    Wire valid = wire( 1, "valid" );
    Wire recv = regre( vcc(), read_en, valid, "read_data" );
    Wire dout = wire( 8, "dout" );
    Wire ready = wire( 1, "ready" );
    new Uart( this, reset, din, send, recv, rx, tx, dout, valid, ready,
              clock_rate, baud_rate );

    Wire status = concat(gnd(7), ready, "status");

    mux_o( status, dout, recv, tx_data );
    mux_o( read_status, valid, recv, tx_valid );

  } // end UartNode()

} // end class UartNode
```

A.4.2 MicroblazeNode.java

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class MicroblazeNode extends NetInterface {

  public MicroblazeNode( Node parent, String name,
                         Wire net_clk, Wire node_clk, Wire reset,
                         Wire net_request, Wire net_release,
                         Wire net_grant, Wire net_pend,
                         Wire net_rx_data, Wire net_rx_addr, Wire net_rx_rnw,
                         Wire net_rx_valid, Wire net_rx_cts,
                         Wire net_tx_data, Wire net_tx_addr, Wire net_tx_rnw,
                         Wire net_tx_valid, Wire net_tx_cts,
                         int src_addr, int timeout_val,
                         boolean use_rx_fifos, boolean use_tx_fifos,
                         int net_access_mask, int net_data_mask,
                         int mem_size ) {

    super( parent, name, net_clk, node_clk, reset,
           net_request, net_release, net_grant, net_pend,
           net_rx_data, net_rx_addr, net_rx_rnw, net_rx_valid, net_rx_cts,
           net_tx_data, net_tx_addr, net_tx_rnw, net_tx_valid, net_tx_cts,
           src_addr, timeout_val, use_rx_fifos, use_tx_fifos );

    new NetMicroblaze( this, reset, rx_valid, rx_rnw, rx_data, rx_addr,
                       grant, cpu_request, cpu_release, dest_addr,
                       tx_valid, tx_rnw, tx_data, tx_addr, tx_cts,
                       net_access_mask, net_data_mask, mem_size );

    new CPUTimeoutFSM( this, timeout, grant,
                       timeout_request, timeout_release);

    or_o( cpu_request, timeout_request, request_in );
    regre_o( vcc(), request_in, grant, request );

    or_o( cpu_release, timeout_release, release_in );
    regre_o( vcc(), release_in, not(grant), release );

  } // end MicroblazeNode()

} // end class MicroblazeNode
```

A.4.3 NetMicroblaze.java

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class NetMicroblaze extends Logic {

  public static CellInterface cell_interface[] = {

    in( "reset", 1 ),
    in( "rx_valid", 1 ),
    in( "rx_rnw", 1 ),
    in( "rx_data", 32 ),
    in( "rx_addr", 32 ),
    in( "grant", 1 ),
    out( "cpu_request", 1 ),
    out( "cpu_release", 1 ),
    out( "dest_addr", 8 ),
    out( "tx_valid", 1 ),
    out( "tx_rnw", 1 ),
    out( "tx_data", 32 ),
    out( "tx_addr", 32 ),
    out( "tx_cts", 1 )

  };

  public NetMicroblaze( Node parent, Wire reset,
                        Wire rx_valid, Wire rx_rnw,
                        Wire rx_data, Wire rx_addr, Wire grant,
                        Wire cpu_request, Wire cpu_release,
                        Wire dest_addr, Wire tx_valid, Wire tx_rnw,
                        Wire tx_data, Wire tx_addr, Wire tx_cts,
                        int net_mask_value, int net_data_mask_value,
                        int mem_size ) {

    super( parent );

    connect( "reset", reset );
    connect( "rx_valid", rx_valid );
    connect( "rx_rnw", rx_rnw );
    connect( "rx_data", rx_data );
    connect( "rx_addr", rx_addr );
    connect( "grant", grant );
    connect( "cpu_request", cpu_request );
    connect( "cpu_release", cpu_release );
    connect( "dest_addr", dest_addr );
    connect( "tx_valid", tx_valid );
    connect( "tx_rnw", tx_rnw );
    connect( "tx_data", tx_data );
    connect( "tx_addr", tx_addr );
    connect( "tx_cts", tx_cts );

    Wire interrupt = wire( 1, "interrupt" );
    Wire i_data = wire( 32, "i_data" );
    Wire i_ready = wire( 1, "i_ready" );
    Wire i_wait = wire( 1, "i_wait" );
    Wire i_fetch = wire( 1, "i_fetch" );
    Wire i_addr = wire( 32, "i_addr" );
    Wire i_addr_valid = wire( 1, "i_addr_valid" );
    Wire d_data_in = wire( 32, "cpu_data_in" );
    Wire d_data_out = wire( 32, "cpu_data_out" );
    Wire d_wait = gnd();
    Wire d_addr = wire( 32, "cpu_data_addr" );
    Wire d_addr_valid = wire( 1, "cpu_addr_valid" );
    Wire d_data_we = wire( 1, "d_data_we" );
    Wire d_data_re = wire( 1, "d_data_re" );
    Wire d_data_be = wire( 4, "d_data_be" );
```

```
    Wire d_ready = wire( 1, "d_ready" );
    new Microblaze( this, reset, interrupt, i_data,
                    i_ready, i_wait, i_fetch, i_addr, i_addr_valid,
                    d_data_in, d_data_out, d_data_we, d_data_re, d_data_be,
                    d_ready, d_wait, d_addr, d_addr_valid );

    Wire mem_fetch = and( i_fetch, i_addr_valid );
    Wire mem_data_re = wire( 1, "mem_data_re" );
    Wire mem_data_we = wire( 1, "mem_data_we" );
    Wire mem_data_out = wire( 32, "mem_data_out" );
    Wire mem_data_valid = wire( 1, "mem_data_valid" );
    new LocalMemory( this, mem_fetch, i_addr, i_data, i_valid,
                     mem_data_re, mem_data_we, d_addr, d_data_out,
                     mem_data_out, mem_data_valid, mem_size );

    Wire cpu_data_we = and( d_data_we, d_addr_valid );
    Wire cpu_data_re = and( d_data_re, d_addr_valid );
    Wire cpu_data_en = or( cpu_data_we, cpu_data_re );

    Wire net_mask = constant( 32, net_mask_value );
    Wire net_access = and( d_addr.gw(31), not(d_addr.gw(30)) );

    Wire net_data_mask = constant( 32, net_data_mask_value );
    Wire net_data_access = and( d_addr.gw(31), d_addr.gw(30) );

    Wire net_active = wire( 1, "net_active" );
    Wire net_access_request = and( net_access, cpu_data_we );
    Wire net_data_request = and( net_data_access, net_active, cpu_data_en );

    and_o( net_access, cpu_data_we, d_data_out.gw(0), cpu_request );
    and_o( net_access, cpu_data_we, d_data_out.gw(1), cpu_release );

    regre_o( cpu_request, net_access_request, cpu_release, net_active );

    Wire net_enable = buf( d_addr.gw(31) );
    Wire data_mem_re = and( cpu_data_re, not(net_enable) );
    Wire data_mem_we = and( cpu_data_we, not(net_enable) );

    Wire net_data_sel = wire( 1, "net_data_sel" );
    Wire net_data = concat( gnd(31), grant );
    Wire net_data_out = mux( rx_data, net_data, net_data_sel );
    Wire net_data_valid = mux( rx_valid, vcc(), net_data_sel );
    regre_o( net_access, cpu_data_re, net_data_valid, net_data_sel );

    Wire cpu_data_sel = regre( net_enable, cpu_data_re, net_data_valid );
    Wire cpu_data_valid = mux( mem_data_valid, net_data_valid, cpu_data_sel );
    mux_o( mem_data_out, net_data_out, cpu_data_sel, d_data_in );

    or_o( cpu_data_valid, reg(cpu_data_we), d_ready );

    and_o( net_active, rx_rnw, rx_valid, interrupt );

    Wire dest_addr_in = buf( d_addr.range(9, 2) );
    regce_o( dest_addr_in, cpu_request, dest_addr );

    Wire tx_addr_in = and( not(net_data_mask), d_addr );
    concat_o( gnd(2), tx_addr_in.range(31, 2), tx_addr );

    buf_o( cpu_data_re, tx_rnw );
    buf_o( net_data_access, tx_valid );
    vcc_o( tx_cts );

  } // end NetMicroblaze()

} // end class NetMicroblaze
```

A.4.4 Microblaze.java

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class Microblaze extends Logic {

  public static CellInterface cell_interface[] = {

    in( "reset", 1 ),
    in( "interrupt", 1 ),
    in( "instruction", 32 ),
    in( "i_ready", 1 ),
    in( "i_wait", 1 ),
    out( "i_fetch", 1 ),
    out( "i_addr", 32 ),
    out( "i_addr_valid", 1 ),
    in( "data_in", 32 ),
    out( "data_out", 32 ),
    out( "data_we", 1 ),
    out( "data_re", 1 ),
    out( "data_be", 4 ),
    in( "d_ready", 1 ),
    in( "d_wait", 1 ),
    out( "d_addr", 32 ),
    out( "d_addr_valid", 1 )

  };

  public Microblaze( Node parent,
                     Wire reset, Wire interrupt, Wire instruction,
                     Wire i_ready, Wire i_wait, Wire i_fetch,
                     Wire i_addr, Wire i_addr_valid,
                     Wire data_in, Wire data_out,
                     Wire data_we, Wire data_re, Wire data_be,
                     Wire d_ready, Wire d_wait,
                     Wire d_addr, Wire d_addr_valid ) {

    super( parent );

    connect( "reset", reset );
    connect( "interrupt", interrupt );
    connect( "instruction", instruction );
    connect( "i_ready", i_ready );
    connect( "i_wait", i_wait );
    connect( "i_fetch", i_fetch );
    connect( "i_addr", i_addr );
    connect( "i_addr_valid", i_addr_valid );
    connect( "data_in", data_in );
    connect( "data_out", data_out );
    connect( "data_we", data_we );
    connect( "data_re", data_re );
    connect( "data_be", data_be );
    connect( "d_ready", d_ready );
    connect( "d_wait", d_wait );
    connect( "d_addr", d_addr );
    connect( "d_addr_valid", d_addr_valid );

  } // end Microblaze()


  public String getCellName() {
    return "microblaze_wrapper";
  }


  public boolean isBlackBox() {
```

```
    return true;
  }

} // end class Microblaze
```

A.4.5 HelloWorld.java

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex2.*;

public class HelloWorld extends Logic {

  public static int port_count = 2;
  public static int dwidths[] = {32, 8};
  public static int awidths[] = {32, 1};

  public static CellInterface cell_interface[] = {

    in( "clk", SINGLE_BIT ),
    in( "reset", SINGLE_BIT ),
    in( "rx", SINGLE_BIT ),
    out( "tx", SINGLE_BIT )

  };

  public HelloWorld( Node parent, Wire clk, Wire reset,
                     Wire rx, Wire tx ) {

    super( parent );

    connect( "clk", clk );
    connect( "reset", reset );
    connect( "rx", rx );
    connect( "tx", tx );


    ////////////////////////////////////////////////////////////////
    //////////////////  Generate System Clocks  ////////////////////
    ////////////////////////////////////////////////////////////////
    Wire clk0 = wire( 1, "clk0" );
    Wire clk_in = wire( 1, "clk_in" );
    Wire clk_fb = wire( 1, "clk_fb" );
    Wire clk_dv = wire( 1, "clk_dv" );
    Wire net_clk = wire( 1, "net_clk" );
    Wire cpu_clk = wire( 1, "cpu_clk" );
    Wire uart_clk = wire( 1, "uart_clk" );

    new ibufg( this, clk, clk_in );
    new bufg( this, clk0, clk_fb );
    new bufg( this, clk0, net_clk );
    new bufg( this, clk0, cpu_clk );
    new bufg( this, clk_dv, uart_clk );

    Cell sys_dcm = new dcm( this,
                            clk_in, clk_fb, gnd(), gnd(), gnd(), gnd(), gnd(),
                            clk0, nc(), nc(), nc(), nc(), nc(), clk_dv, nc(),
                            nc(), nc(), nc(8), nc() );

    sys_dcm.addProperty("CLKIN_PERIOD", "10ns");
    sys_dcm.addProperty("CLKDV_DIVIDE", "2");


    ////////////////////////////////////////////////////////////////
    ////////////////  Create the Network Signals  //////////////////
    ////////////////////////////////////////////////////////////////
    Wire request[] = new Wire[port_count];
    Wire release[] = new Wire[port_count];
    Wire grant[] = new Wire[port_count];
    Wire pend[] = new Wire[port_count];
    Wire rx_data[] = new Wire[port_count];
    Wire rx_addr[] = new Wire[port_count];
```

```
Wire rx_rnw[] = new Wire[port_count];
Wire rx_valid[] = new Wire[port_count];
Wire rx_cts[] = new Wire[port_count];
Wire tx_data[] = new Wire[port_count];
Wire tx_addr[] = new Wire[port_count];
Wire tx_rnw[] = new Wire[port_count];
Wire tx_valid[] = new Wire[port_count];
Wire tx_cts[] = new Wire[port_count];

for( int i = 0; i < port_count; i++ ) {
  String prefix = "port"+i+"_";
  int dwidth = dwidths[i];
  int awidth = awidths[i];
  request[i] = wire( SINGLE_BIT, prefix+"request" );
  release[i] = wire( SINGLE_BIT, prefix+"release" );
  grant[i] = wire( SINGLE_BIT, prefix+"grant" );
  pend[i] = wire( SINGLE_BIT, prefix+"pend" );
  rx_data[i] = wire( dwidth, prefix+"rx_data" );
  rx_addr[i] = wire( awidth, prefix+"rx_addr" );
  rx_rnw[i] = wire( SINGLE_BIT, prefix+"rx_rnw" );
  rx_valid[i] = wire( SINGLE_BIT, prefix+"rx_valid" );
  rx_cts[i] = wire( SINGLE_BIT, prefix+"rx_cts" );
  tx_data[i] = wire( dwidth, prefix+"tx_data" );
  tx_addr[i] = wire( awidth, prefix+"tx_addr" );
  tx_rnw[i] = wire( SINGLE_BIT, prefix+"tx_rnw" );
  tx_valid[i] = wire( SINGLE_BIT, prefix+"tx_valid" );
  tx_cts[i] = wire( SINGLE_BIT, prefix+"tx_cts" );
}


////////////////////////////////////////////////////////////////
//////////////////  Instance the NetRouter  ////////////////////
////////////////////////////////////////////////////////////////
setDefaultClock( net_clk );

boolean table_update_enable = false;
int table_init[] = {0x00,0x01,0x02};
new NetRouter( this,
               request, release, grant, pend,
               rx_data, rx_addr, rx_rnw, rx_valid, rx_cts,
               tx_data, tx_addr, tx_rnw, tx_valid, tx_cts,
               table_update_enable, table_init );


////////////////////////////////////////////////////////////////
//////////////////  Instance the CPUNode  //////////////////////
////////////////////////////////////////////////////////////////
setDefaultClock( cpu_clk );

int cpu_addr = 1;
int cpu_timeout_value = 0;
boolean cpu_use_rx_fifos = false;
boolean cpu_use_tx_fifos = false;
int net_access_mask = 0x80000000;
int net_data_mask = 0xC0000000;
int cpu_mem_size = 64;
new MicroblazeNode( this, "MicroblazeNode",
                    net_clk, cpu_clk, reset,
                    request[0], release[0], grant[0], pend[0],
                    tx_data[0], tx_addr[0], tx_rnw[0],
                    tx_valid[0], tx_cts[0],
                    rx_data[0], rx_addr[0], rx_rnw[0],
                    rx_valid[0], rx_cts[0],
                    cpu_addr, cpu_timeout_value,
                    cpu_use_rx_fifos, cpu_use_tx_fifos,
                    network_mask, network_addr_mask,
                    cpu_mem_size );
```

```
    ////////////////////////////////////////////////////////////////
    ////////////////  Instance the UartNode  ///////////////////////
    ////////////////////////////////////////////////////////////////
    setDefaultClock( uart_clk );

    int uart_addr = 2;
    int uart_timeout_value = 0;
    boolean uart_use_rx_fifos = true;
    boolean uart_use_tx_fifos = true;
    int uart_clk_freq = 50000000;
    int uart_baud_rate = 115200;
    new UartNode( this, "UartNode",
                  net_clk, uart_clk, reset,
                  request[1], release[1], grant[1], pend[1],
                  tx_data[1], tx_addr[1], tx_rnw[1],
                  tx_valid[1], tx_cts[1],
                  rx_data[1], rx_addr[1], rx_rnw[1],
                  rx_valid[1], rx_cts[1],
                  uart_addr, uart_timeout_value,
                  uart_use_rx_fifos, uart_use_tx_fifos,
                  rx, tx, uart_clk_freq, uart_baud_rate );

  } // end HelloWorld()

} // end class HelloWorld
```

A.4.6 HelloWorld.c

```
////////////////////////////////////////////////////////////////////////
//////////////////////  Network Access Defines  ////////////////////////
////////////////////////////////////////////////////////////////////////
#define NET_ACCESS_MASK 0x80000000
#define NET_DATA_MASK   0xC0000000

#define NET_REQUEST 0x01
#define NET_RELEASE 0x02


////////////////////////////////////////////////////////////////////////
//////////////////////  Network Access Macros  ////////////////////////
////////////////////////////////////////////////////////////////////////
#define mNetworkIn(net_addr) \
  (*(volatile int *)(net_addr))

#define mNetworkOut(net_addr, value)  \
  (*(volatile int *)(net_addr) = value)

#define mNetworkRequest(node_addr) \
  mNetworkOut(NET_ACCESS_MASK+(node_addr<<2), NET_REQUEST); \
  while(!mNetworkIn(NET_ACCESS_MASK))

#define mNetworkRelease(node_addr) \
  mNetworkOut(NET_ACCESS_MASK, NET_RELEASE); \
  while(mNetworkIn(NET_ACCESS_MASK))

#define mNetworkRead(data_addr) \
  (*(volatile int *)(NET_DATA_MASK+(data_addr<<2)))

#define mNetworkWrite(data_addr, value) \
  (*(volatile int *)(NET_DATA_MASK+(data_addr<<2)) = value)


////////////////////////////////////////////////////////////////////////
//////////////////////  UART Defines & Functions  //////////////////////
////////////////////////////////////////////////////////////////////////
#define UART_ADDR           0x02
#define UART_STATUS_OFFSET  0x01
#define UART_READY_MASK     0x01
void printString(char* value);
void sendByte(int value);


////////////////////////////////////////////////////////////////////////
//////////////////////  Function Definitions  //////////////////////////
////////////////////////////////////////////////////////////////////////
int main() {

  mNetworkRequest(UART_ADDR);
  printString("Hello World!\r\n");
  mNetworkRelease(UART_ADDR);

} // end main()


void printString(char* value) {
  int c, i = 0;

  while( (c = value[i++]) != '\0' )
    sendByte(c);

} // end printString()
```

```
void sendByte(int value) {

  while( mNetworkRead(UART_STATUS_OFFSET) != UART_READY_MASK );
  mNetworWrite(0, value);

} // end sendByte()
```

A.4.7 Software Makefile

```
SW_FILE = HelloWorld

CC = mb-gcc -mno-xl-soft-mul -mno-xl-soft-div
AS = mb-as
AR = mb-ar
LD = mb-ld -relax -N -T default.link
DUMP = mb-objdump -D

all : $(SW_FILE).lst

$(SW_FILE).lst : $(SW_FILE).c
        $(CC) -c $(SW_FILE).c -o $(SW_FILE).o
        $(AS) init.S -o init.o
        $(LD) init.o $(SW_FILE).o libxil.a -o $(SW_FILE).elf
        $(DUMP) $(SW_FILE).elf > $(SW_FILE).lst
```

A.4.8 Hardware Makefile

```
HW_FILE = HelloWorld
HW_PART = xc2vp30-ff896-7

all: $(HW_FILE).bit

$(HW_FILE).bit : $(HW_FILE).edn
        ngdbuild -p $(HW_PART) $(HW_FILE)
        map -detail $(HW_FILE)
        par -ol 5 -w $(HW_FILE).ncd $(HW_FILE).par.ncd
        trce -v 100 -a $(HW_FILE).par.ncd
        bitgen -w -g startupclk:jtagclk $(HW_FILE).par.ncd
        mv *.par.twr $(HW_FILE).twr
        mv *.par.bit $(HW_FILE).bit

$(HW_FILE).edn : $(HW_FILE).ucf
        jikes +Pno-naming-convention *.java
        java tb_$(HW_FILE) -netlist

$(HW_FILE).ucf : $(HW_FILE).elf
        data2mem -bm $(HW_FILE).bmm -bd $(HW_FILE).elf
                 -o u temp.ucf -p $(HW_PART) -i
        cat temp.ucf >> $(HW_FILE).ucf
```

Bibliography

[1] G. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, no. 8, 1965.

[2] M. Abramovici, C. Stroud, and M. Emmert, "Using Embedded FPGAs for SoC Yield Improvement," in *Proceedings of the Design Automation Conference*. DAC 02, 10-14 June 2002, pp. 713–724.

[3] "ARM, AMBA Specification," ARM, Tech. Rep., 1999, revision 2.0.

[4] "CoreConnect, CoreConnect Bus Architecture," IBM Cooperation, Tech. Rep., 1999.

[5] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen, "Overview of Bus-Based System-on-Chip Interconnections," in *Proceedings of the IEEE International Symposium on Circuits and Systems*. ISCAS 02, 26-29 May 2002, pp. II–372 – II–375 vol.2.

[6] [Online]. Available: http://www.xilinx.com

[7] [Online]. Available: http://www.altera.com

[8] L. Beninni and G. D. Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, pp. 70–78, 2002.

[9] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the Design Automation Conference*. DAC 01, 18-22 June 2001, pp. 684–689.

[10] S. Kumar and A. Jantsch, "A Netork on Chip Architecture and Design Methodology," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI.* ISVLSI 02, 25-26 April 2002, pp. 105–112.

[11] C. Grecu, P. P. Pande, A. Ivanov, and R. Saleh, "A Scalable Communication-Centric SoC Interconnect Architecture," in *Proceedings of the 5th International Symposium on Quality Electronic Design*, 2004, pp. 343–348.

[12] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs," in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications.* FPL 02, September 2002, pp. 795–805.

[13] D. Wiklund and D. Liu, "SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[14] J. Liu, L.-R. Zheng, and H. Tenhunen, "A Circuit-Switched Network Architecture for Network-on-Chip," in *Proceedings of the International Symposium on System-on-Chip*, September 2004, pp. 55–58.

[15] L. Peterson and B. Davie, *Computer Networks: A Systems Approach*, 2nd ed. San Francisco: Morgan Kaufmann, 2000, ch. 3, pp. 170–186.

[16] D. Lim and M. Peattie, "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations," Xilinx Corporation, Tech. Rep., 17 May 2002, XAPP290 (v1.0).

[17] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition.* DATE 03, 2003.

[18] P. Bellows and B. Hutchings, "JHDL-an HDL for Reconfigurable Systems," in *Proceedings of FPGAs for Custom Computing Machines.* FCCM 98, 15-17 April 1998, pp. 175–184.

[19] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Highly Scalable Network on Chip for Reconfigurable Systems," in *Proceedings of the International Symposium on System-on-Chip*, November 2003, pp. 79–82.