A Scalable Approach to Multi-core Prototyping

Jamie David Newcomb

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. Cameron D. Patterson, Chair
Dr. R. Michael Buehrer
Dr. Patrick Schaumont

January 25, 2008
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: multi-core, multiprocessor array, FPGA, Xilinx, Aurora, multi-gigabit

A Scalable Approach to Multi-core Prototyping

Jamie David Newcomb

(ABSTRACT)

In recent years, multi-core processors and multi-processor networks have grown in popularity as a solution to the limits on increasing clock speed, rising power consumption, and the nanometer manufacturing processes. Multi-core processors and multi-processor networks are seen as the next step in the advancement of computational capabilities by way of concurrent processing. However, parallel software design is difficult due to the immaturity of scalable architectures and software development environments for multi-core hardware.

How should processors effectively and quickly pass information, with as little overhead as possible? What kind of communication architecture is best suited for parallelism? How can large-scale architectures be quickly produced, verified and properly utilized by software? Using commercially available FPGA development boards, Xilinx tools and components, this thesis offers a light-weight solution to these questions for effective, low-overhead, low-latency multi-core communication and fast prototyping of multi-processor networks for scalable processor arrays.

Dedication

So, I'll throw my life upon all that You are
'Cause I know You gave it all for me

Evermore my heart, my heart will say
Above all, I live for Your glory

Everlasting, Your light will shine when all else fades
Never ending, Your glory goes beyond all fame
And the cry of my heart is to bring You praise
From the inside out, Lord, my soul cries out

To the living God,
All I can say is that I hope I did
what you brought me here to do, Lord.
It wasn't easy! But maybe that was the point.

Acknowledgments

My sincere thanks go to the professors that comprise my committee:

Dr. Patterson, your willingness to see past my shortcomings and allow me to work on this project under your guidance is something for which I cannot thank you enough. Your wealth of knowledge and expertise in not only engineering but lifestyle and research practices will benefit me for years to come. Thanks for sticking with me and seeing my potential beyond my gradepoint average.

Dr. Buehrer, I also cannot thank you enough for our sometimes lengthy talks, whether about classes or engineering or relationships or faith, that have helped sustain me during my work here at Virginia Tech. Without your help in pointing me to *[nlcf]*, I likely would have struggled to find the community I needed when I moved here from my far-away home in Michigan. The simple act of listing off a few campus ministries has led me to an amazing place in my life, spiritually and academically.

Dr. Schaumont, thank you for pointing me in the direction of the Configurable Computing Lab and the work in which Dr. Patterson was involved. Without your guidance, I would have floundered in finding a research project that sparked my interests. Working with you as a TA, I found your personality and willingness to be of service to students inspiring. And always with a ready smile.

My heart-felt thanks go to my mom and dad for their unwavering support, unconditional love and willingness to help me see this through. I love you both very much, and I don't know what I would do without you. Thanks go to my brother Jason, for being the comic relief in the truest sense, and to whom I look as an image of manliness. The stories of traffic stops were the greatest parts of many of my days. I have to say, though, The State of Michigan should never have given you a gun. Thanks also to my sister Carrie and her family, for your support and allowing me to just relax and enjoy the company of my nephew and neice.

Thanks to my HomeGroup, the members of whom both past and present have provided me with that spiritual and mental help and guidance I needed as a fledgling Christian. As great models of faith, your friendship will forever be written on my heart. I always look forward to my favorite night of the week: Thursdays. I want to say so much more about how much you've all helped me grow in so many more ways than even I recognize, and to each and every one of you specifically, but the space on these pages doesn't allow it.

To my fiancée, Jessamyn, your constant support was essential to the fact that you read this thesis today. Without you, I would have given up a long time ago and none of this would have been finished. Without the people mentioned above, I would never have met you. Without your ability to help me to relax and unwind after long days in the lab, I would have gone insane. Without the love you've shown me, I wouldn't know what it means to be a servant in the manner that God's called us to be. I love you very much. I'm excited about our future lives together and what God has in store for us!

Many thanks go to my church family back home: Anne, Troy, Bethany and the rest of the family; DeeDee and Russ Tibbits; the Seitz family. What would I have done without your support and prayers during my time at VT? I couldn't ask for more spirit-filled and God-loving friends. Thanks for showing me who Jesus is, and why He means so much to you all — to all of us. Our discussions and worship are things I always look forward to, and will

continue to do so each time we meet. God has blessed me with your friendship in more ways that I can possibly count.

Thanks to the dudes of the CCM Lab; though I was usually quiet and kept to myself, I wouldn't have gotten very far had it not been for your willingness to answer any and all of the questions I had. Thanks for foozin'.

Go Hokies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Scientific computing applications often require complex calculations that force computer systems to the edge of their capabilities. The search for increased computational power has led to the advent of multi-core processors, otherwise known as chip-multiprocessors (CMPs). They are defined as several independent processor cores that are fabricated on the same die of silicon. As can be seen through reading current literature on the state of Moore's Law, single processors no longer satisfy the trend of achieving higher performance by increasing a processor's clock rate and decreasing the nanometer fabrication process. This leads to growing power consumption due to the physical limitations of integrated circuits. Multi-core processors are seen as a solution to these limitations [1–3].

Moore's Law was an optimistic observation and prediction of transistor growth in processors made by Intel co-founder Gordon E. Moore in 1965. At the lowest cost per transistor, the density of transistors can double between every 18 and 24 months. This observation has been translated into a closely followed goal for the microprocessor industry in the last 30 years. Figure 1.1 shows a graph of Intel's transistor growth rate for some of its processors

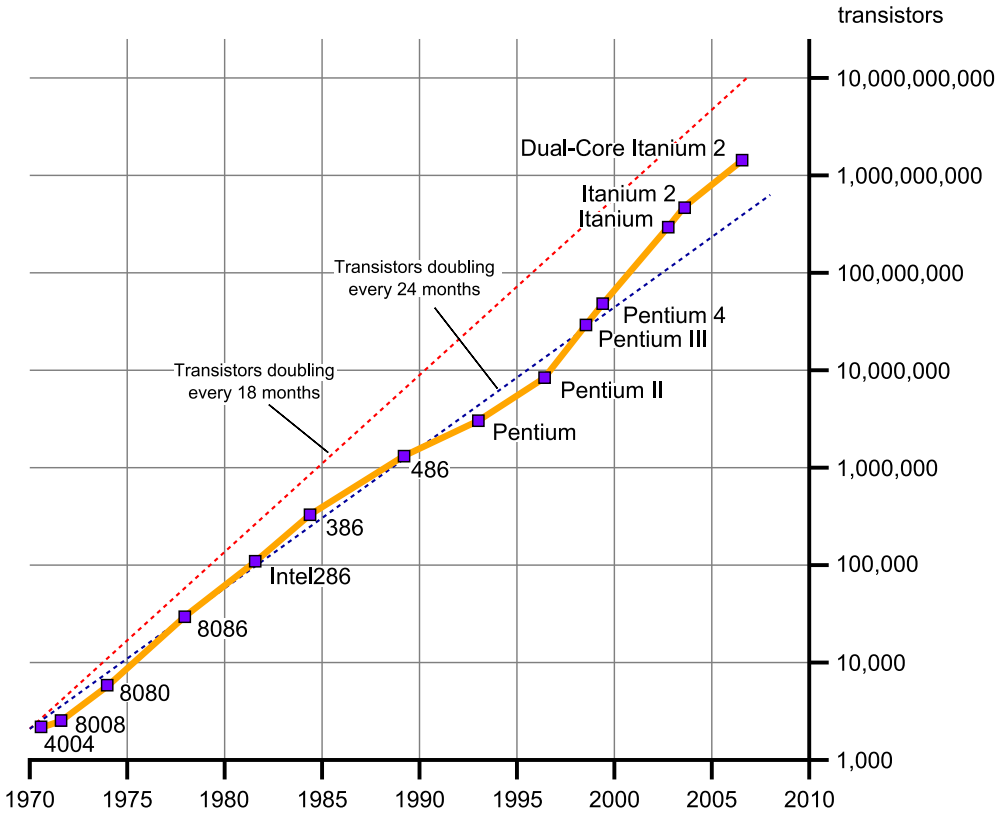


Figure 1.1: Moore's Law: Intel Processor Transistor Counts

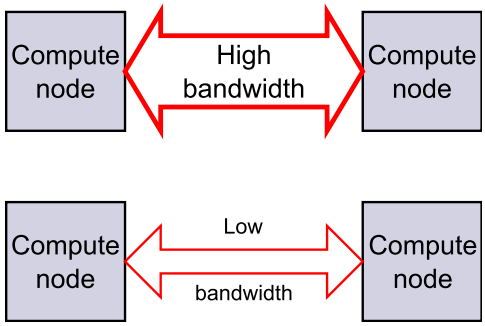
compared to the prediction set by Moore's Law.

As multiple core processors seek to match and possibly exceed Moore's Law, software applications must be written to take advantage of the increased computing potential. A large majority of software applications are sequential in nature, following a step-by-step algorithm. There is a need for greater parallelism within software to reduce processing time and advance software development. Current solutions for developing parallel software have been hindered by a lack of multi-core hardware platforms. In lieu of capable hardware, developers have turned to software simulation. While simulations provide excellent system visibility (one can define what is observed at any time), the results of a given simulation are primarily based on the quality of the simulator. Proper simulation can only be achieved through the

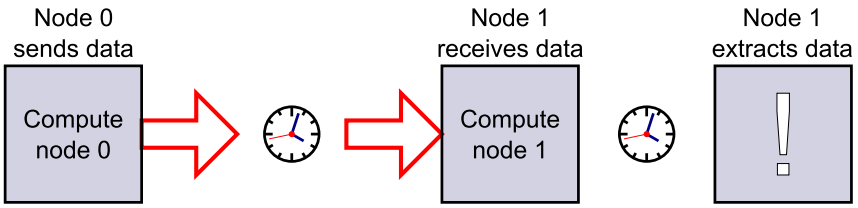
exact cycle-by-cycle reproduction of what the targeted hardware would produce. As the complexity of the targeted hardware increases, so does the time required to obtain a set of desired results. Because of the complexity of large multi-core networks, software simulation is insufficient for this level of software development.

Supercomputers and computing clusters have been another solution in terms of hardware that have allowed large-scale parallel software to be developed and analyzed. Each compute node is independent insofar as it is comprised of its own local memory and central processing unit (CPU) at the very least, much like separate personal desktop computers connected together in a network. However, such systems are expensive to build and maintain. A 1000-node system can likely cost several millions of dollars as evidenced by the costs of the supercomputers on the Top 500 Supercomputers List [4]. Communication between compute nodes is more costly in terms of latency and power consumption in such systems as compared to the shorter links found between processors printed on the same silicon die.

Bandwidth and communication latency are also major factors limiting performance in large networks. Figure 1.2 shows examples of bandwidth and latency. Bandwidth is the amount of data that can be transmitted during a certain amount of time. Many networks in use today have nodes connected by a common collection of wires, or *bus*. Overall network bandwidth is limited by the bandwidth of the bus. With nodes connected in a ring, connectivity among nodes is limited and reduces scalability. Both situations can result in performance bottlenecks in which nodes might be spending a significant amount of time waiting for information to reach them. Communication latency, the amount of time taken to complete a communication task, can be measured from the time in which information is sent from one node to the time when extracted data is processed at a desired destination node. Latency is primarily caused by the amount of overhead in software or communication protocols. Overhead is defined in many ways, but may be summarized in this context as the amount of processing time required before extracting useful data from communicated information.



(a) Bandwidth



(b) Latency

Figure 1.2: Limitations on Performance

Communication protocols enforce rules upon data to which software adheres by adding more information to data that the destination node must remove or ignore to obtain the original data.

An analogy might be how people travel in vehicles. To properly use a road, people use a vehicle to travel along the road to reach a destination, much like how data travels within the confines of a protocol along a communication bus. Bandwidth would be similar to the number of lanes a road might have. More vehicles can travel along wider roads, much like how more data can flow through a wider bus. Latency is similar to the amount of time it takes for a vehicle to travel to its destination and for the passengers to exit the vehicle (a combination of data transmission and data extraction). Within a computer, however, latency is dominated by the time taken to extract and interpret the data.

Therefore, relatively inexpensive hardware-based multi-core architectures with short interconnects and low-latency communication schemes are required to fulfill the needs of complex parallel software applications. However, there is currently no definitive multi-core architecture. How is parallelism best realized in hardware? What kind of communication can support and assist this type of processing? Can a reusable and adaptable multi-core architecture be developed in a manner that is portable across hardware platforms? A synergy between hardware and software must exist to reduce overall processing time for complex scientific applications.

This project is offered in the spirit (but not necessarily the means) of the suggestions provided in [5–9]. This thesis describes an FPGA-based solution for general application processing that highlights the potential to create a scalable and portable multi-core processor platform that promotes parallel software applications development using commercially-available FPGA development boards. The challenges of parallel software design for use on multi-core architectures, as expressed in [10], is lightly examined, as these two worlds often collide, and several research efforts exist elsewhere that focus on these challenges in far greater detail [11, 12].

1.2 Contributions

Multi-core architectures should provide:

1. scalability in the communication media as well as the network topology itself,
2. low-overhead and low-latency communication,
3. high bi-sectional bandwidth (a measure of the amount of data that can be passed between equal halves of a network during a single time interval),
4. sufficient parallelism for a given application,

5. the ability to be quickly prototyped,

6. portability among commercially-available FPGA development boards (i.e., no custom hardware required),

7. and lastly, a means to more easily develop and debug software.

The architecture presented in this thesis is a hardware-based solution with low-overhead, low-latency communication, balanced on-chip/off-chip bandwidth, and extensive debug facilities for the purpose of aiding parallel software development and developing parallel software design methodologies. Rapid prototyping of this architecture is possible through the use of commercially-available FPGA development boards. MicroBlaze™ (MB) 32-bit RISC processors, each with its own local memory accessed by a Local Memory Bus (LMB), are connected in a "nearest-neighbor" fashion by uni-directional, point-to-point, FIFO-style Fast Simplex Links (FSLs) in a two-dimensional mesh. Off-chip communications are provided by the Aurora serial communication protocol, occupying four Multi-Gigabit Transceivers (MGTs). The system was developed on and in its current form restricted to the resources of the Xilinx proprietary Virtex-II Pro XC2VP30 FPGA and the ML310 Embedded Development board, using the Xilinx Embedded Development Kit™ software tools. Applications were developed and debugged using a combination of a custom FSL-based RS-232 UART arbiter, the Xilinx MicroBlaze Debugger Module and corresponding Xilinx Microprocessor Debugger program, and the Xilinx internal signal viewer ChipScope™.

1.3 Thesis Organization

Chapter 2 describes multi-core in greater detail, including a few past and current research efforts in the area of multi-core hardware development and parallel computing.

Chapter 3 provides a general overview of the proposed multi-core design.

Chapter 4 discusses the details of the architecture and provides insight into the decisions that were made during the design process, including deliberate design omissions and the reasons why.

Chapter 5 offers insight on the software development and debugging process. The results of several software applications run on the hardware for comparison are provided.

Chapter 6 summarizes the results and contributions of this project. Suggestions for future work regarding this particular architecture are also discussed.

An Appendix and bibliography follows, providing details of the Aurora intellectual property (IP) core as well as the source code of the testbed applications.

Chapter 2

Background

Multi-core hardware technology is still in its infancy, and software design methodologies haven't been developed that fully utilize the parallelism available on multi-core hardware. In addition, current multi-core solutions lack proper communication efficiency as the number of processors scales up. While both situations require improvement, it is a matter of the chicken and the egg. How can software for multi-core be developed without the existence of proper hardware? What good is multi-core hardware without software to use it? This chapter describes a few background topics and discusses some related research in the area of multi-core hardware architectures.

2.1 Overview

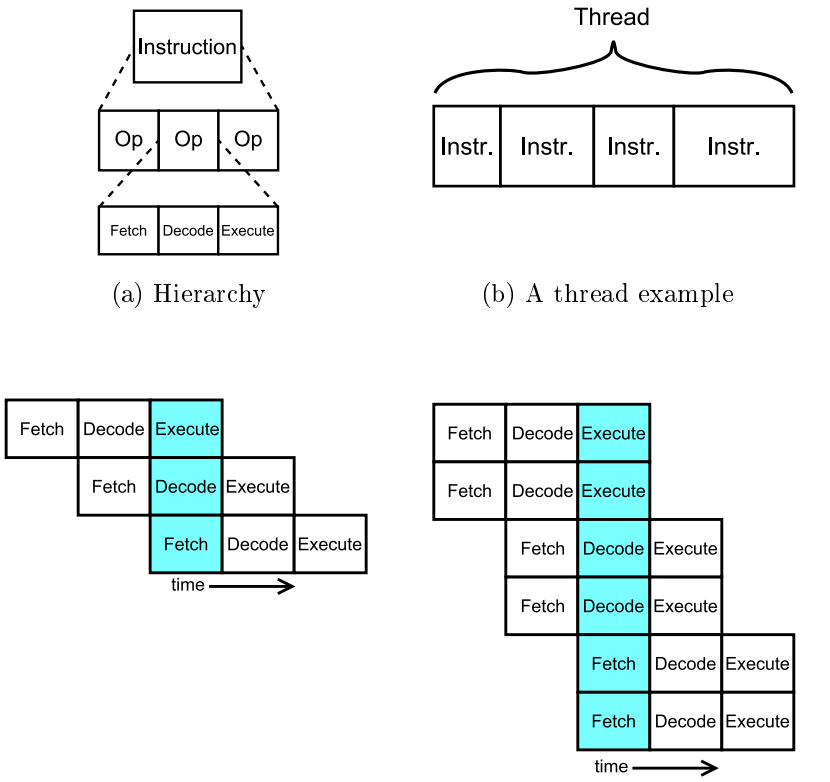
In the feature film *Bill & Ted's Excellent Adventure*, a student named Ox ends a speech with the following: "Everything is different, but the same... things are more moderner than before... bigger, and yet smaller... it's computers..." While ineloquent, this sentiment rings true. It *is* computers. Computers drive academia, help run corporations, and support the operations of governments. While this wasn't the case before the computer age dawned in the 20th century, there is now an insatiable need for faster processing that Moore's Law

states cannot be supplied without altering the current hardware landscape.

The vast majority of personal computers use a single processor. Some computers run on two or more individual processors that have separate sockets on the same motherboard (referred to as a multiprocessor computer). The research community has generally been relegated to building or buying clusters of these systems for complex scientific computing tasks. Recently, more home computers and workstations are being built with processors that contain multiple, yet independent, processor cores fabricated on the same piece of silicon. To be clear, a multi-core processor utilizes only a single motherboard socket. This single-socket processor contains several independent processing cores on its silicon die that may share some level of memory (referred to as *cache*) that stores temporary copies of frequently accessed data. A multiprocessor computer incorporates several separate uniprocessors that do not share a socket or a silicon die, but are embodied within a single computer system.

2.1.1 Intra-processor Parallelism

Processors perform tasks called *instructions* that are comprised of *operations*. An instruction might be "`add x and y`." An operation for this instruction might be "`move x into register 2`," in preparation for the addition to take place. Operations can be broken down into what the processor sees as *stages*; three of the most common include Fetch, Decode and Execute. More stages might be included depending on the processor's architecture. Parallelism in the form of *pipelining* is similar to an assembly line for the stages of an operation. For example, when an operation's Decode stage is processing, another operation's Fetch might begin. A *superscalar pipeline* would be similar to several assembly lines operating in tandem on several instructions. Some computer systems find these methods suffice for enabling parallelism. Others extend the analogy further in what are called *threads*. Threads are defined as a flow of control within an application. A thread combines several instructions together (whether processed in parallel or sequentially), and are defined explicitly by



(a) Hierarchy (b) A thread example

(c) a 3-stage Pipeline (d) Superscalar Pipeline (2× 3-stage)

Figure 2.1: Processing Terms

the application. Extending this idea yields a *process* that could potentially consist of several threads. Operating systems (OS) today are charged with scheduling the time during which threads are processed within the flow of an application.

Scientific computation generally involves complex algorithms that take a significant amount of time to process on a single processor. *Thread-level parallelism* (TLP) and *instruction-level parallelism* (ILP) are two of the primary methods by which today's computers provide concurrency to an application. Each of these abstractions can be combined to increase the level of parallelism. Separating parts of an application to run concurrently on several processors (a form of TLP), or separating tasks that can be run concurrently within the same processor

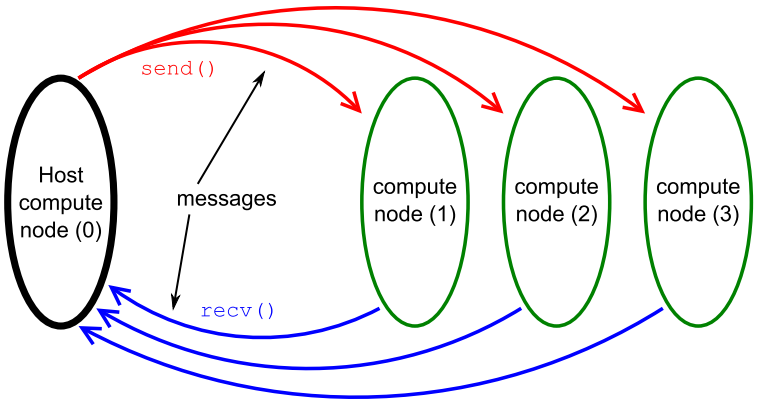
(a form of ILP), reduces the time taken to arrive at the solution. Multiple cores within a processor, as well as processors distributed among connected systems, aid in processing several threads at once. Likely, a combination of TLP and ILP is used: threads are communicated to processors by either shared memory or by message-passing protocols, and instructions within a thread are processed concurrently in superscalar pipelines.

2.1.2 Extra-processor Parallelism

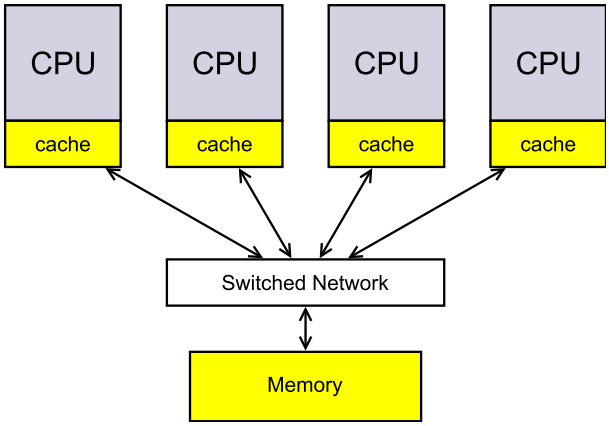
Figure 2.2 shows two common methods for enabling communication between independent processing entities (or compute nodes): *message passing* and *shared memory*. Message passing is the transmission of a set of instructions or data from one compute node to another. Shared memory is a storage medium that is accessible by several compute nodes; communication occurs through the access and modification of memory locations by two or more compute nodes.

A software package called the Message Passing Interface (MPI) enables software parallelism by distributing multiple instances of a program embedded with MPI communication functions to predefined nodes within a network, using *messages* to pass data from one node to another [13]. A message is a contiguous set of data generally consisting of the source node identification, the destination identification, the set of data that the destination(s) should process, and other fields that assist the communication process. Implementations of MPI with varying functionality are widely used by many of the supercomputers in the Top 500 Supercomputers list in lieu of software distinctly designed for parallelism. Generally, MPI works over Ethernet, but some implementations utilize a serial communication medium called InfiniBand [14] as well as other communication media.

Within an MPI environment (providing parallel task functionality), a *process* is an entity similar to compute nodes in a network. While multiple processes can be mapped to a single



(a) Message-Passing



(b) Shared Memory

Figure 2.2: Message-Passing and Shared Memory

node, the reverse is not true, and generally a single process corresponds to a single node. Process 0 would likely be mapped to the host compute node from which copies of the MPI-based program are transmitted to the other predetermined nodes. A *communicator* is defined as a group of processes (or nodes, in this context). Each process within a group is given a *rank* that provides an order of hierarchy among the group. Each process can communicate to other processes in a group through single, point-to-point *Send* and *Receive* functions as well as with collective one-to-many *Broadcast* and many-to-one *Reduce* functions. A "point-to-point" connection allows data to travel only between two interconnected nodes. More than one Communicator can exist. MPI can provide even more complex functionality for

parallelism.

The following code example prints a message to the host computer's screen for each compute node in the network.

```
#include    <mpi.h>
#include    <stdio.h>
#include    <string.h>

#define    MSG_SIZE    64
#define    TAG    0

int main(int argc, char *argv[])
{
    // declare variables
    int   numnodes, me, i;    // number of nodes, node id, generic "i"
    char  message[MSG_SIZE];  // the message itself inside each communication
    char  str[32];            // a character string, if needed
    MPI_Status  status;       // MPI's struct for status information

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numnodes);   // MPI_COMM_WORLD is a common
    MPI_Comm_rank(MPI_COMM_WORLD, &me);         // Communicator descriptor

    if (me == 0)
    {
        printf("I'm processor %d\r\n", me);
        for (i=1; i<numnodes; i++)
        {
            // this node will wait until a message is received
            MPI_Recv(message, MSG_SIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &status);
            printf("From node %d: %s\r\n", me, message);
        }
    }
    else
    {
        sprintf(str, "I'm node %d, ready for processing.\r\n", me);
        strcat(message, str);
        MPI_Send(message, MSG_SIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    }

    MPI_Finalize();
```

```
    return 0;
}
```

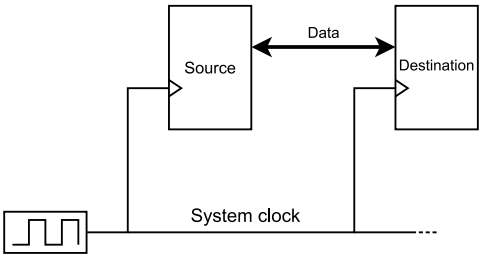
Shared memory is another widely-used method for allowing access to data among several processors. Figure 2.2(b) shows how processors share memory. Communication occurs by storing information for another node in a location that is accessible by the intended node. Sending mail to someone is a proper analogy: a letter awaits the intended recipient in an accessible location such as a mailbox. As mentioned before, *cache* is an intermediate memory bank that is generally closely linked to a processor for storing frequently accessed data . A processor looks in its cache before resorting to requesting data from latency-heavy external memory. However, data is accessible by all of the nodes in the network from a central location (assuming all are connected to a single bank of memory). A danger exists if preventative measures are not taken to protect data with a method such as locking memory locations from alteration while in use by a processor. Failure to do so might result in data corruption or a system crash. Deadlock is a situation in which two or more nodes are suspended indefinitely while waiting for another node to complete a task on data that it cannot access due to the locking mechanism. Shared memory limits the scalability of a network by increasing memory contention as the number of nodes connected to the shared memory increases. Memory access times in a single computer can take up to tens of clock cycles to complete. When the memory exists off-chip, either on the same motherboard as the processor or in a separate computer altogether, this latency can be as high as hundreds of clock cycles.

The most efficient communication between processors is not found in current message passing techniques or shared memory, both of which incur high overhead in either data encapsulation in software or hardware access times. These technologies are used due to a lack of a more efficient communication architecture. They were designed assuming systems would have high overhead and latency. Shared-memory systems benefit from ease of programming (data is kept in one central location accessible by all nodes) as compared to distributed-memory systems that primarily use message passing for node intercommunication (data is kept on each

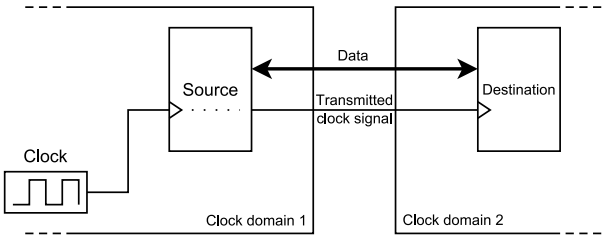
node separately, requiring the exchange of information through messages and synchronization between nodes). The performance improvement methods in the past allowed memory access *times* to decrease by increasing the *speed* of the system clock. In relative terms, however, the number of clock cycles needed to access data remain the same. This trend has been seen as acceptable, but clearly begs for redesign [10]. In large networks and buses, long wires abound and many times require data to be repeated before arriving at its destination to avoid losing or corrupting data during transmission. Longer wires require more power to sustain data integrity. Shorter wires yield lower latency and less power consumption [1]. Performance gains can no longer be obtained by increasing system clock rates.

2.1.3 The Move to Serial Communication

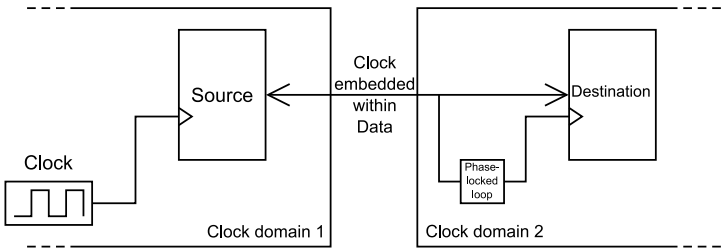
In systems requiring communication, synchronization between nodes is a crucial aspect of system performance. Synchronization is a preventative measure against a lack of communication context. There are three primary methods for synchronization between communication partners: system-wide, source-based and self-synchronization. Figure 2.3 shows examples of each scheme. A system-synchronous mechanism uses a common clock or synchronization source between all communication partners. In the past, delays along the clock path were considered trivial until processor speed increases made the communication delays more visible. Source-synchronous systems generate a clock signal and transmit it along with the data through a separate channel, simplifying delay problems that once plagued system-synchronous models. In this model, the destination compute node processes the data received from the communication channel according to the clock domain provided by the source. However, this results in an increased number of clock domains in the system and increases complexity. Finally, self-synchronous designs encode the clock domain within the transmitted data from which the received clock and data must be decoded. System communication complexity is reduced and transmission can scale to greater speeds. Generally, self-synchronous designs are serial in nature. Communication overhead is slightly increased due to the necessary inclusion of non-data bits embedded within the transmission stream that act as indicators for separate



(a) System-sync



(b) Source-sync



(c) Self-sync

Figure 2.3: Synchronization Schemes

data words.

Parallel input/output (I/O) links require a greater number of wires and pins than their serial counterparts. Data-alignment problems occur at the receiving end, limiting transmission speeds. These result in increased cost for using parallel I/O. Differential serial communication reduces the number of wires, decreases electromagnetic interference and provides far greater switching speeds. For simplicity of design and high-bandwidth communication links, it is clear that serial I/O provides a favorable solution.

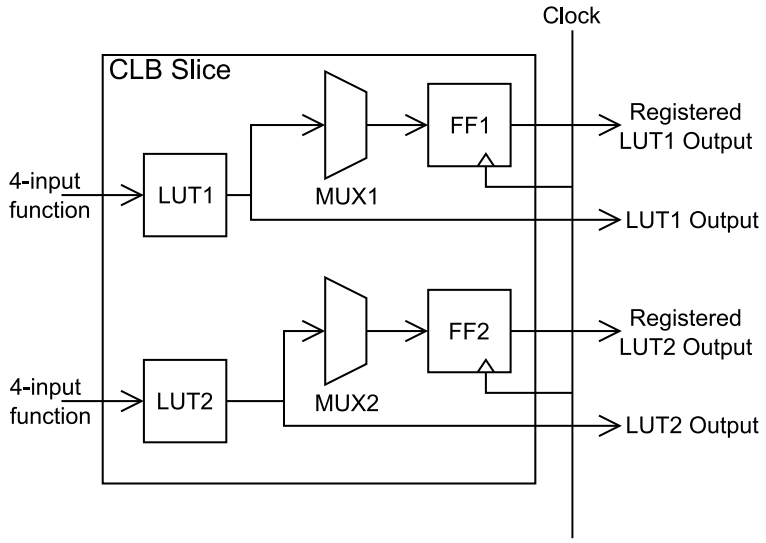


Figure 2.4: An example of a single Slice of a Xilinx Configurable Logic Block

2.1.4 FPGAs

Field Programmable Gate Arrays (FPGA) were first developed in the 1980s as a solution to the cost of fabricating custom chips. Today, they are used in many academic and corporate fields for developing hardware that can be corrected or enhanced, thereby reducing costs incurred by design errors. Standard (or universal) logic printed as integrated circuits (IC) cannot be altered after fabrication without destroying or severing a part of the circuit. Application-Specific Integrated Circuits (ASIC) are popular fabrication solutions for custom chips that differ from standard logic in the enhancements provided by their optimization to a particular application or hardware design. However, in the same way that standard logic chips are unalterable, the ASIC design cannot be changed once it is printed on a silicon wafer. This is not the case for FPGAs.

The primary logic unit of a modern Xilinx FPGA is referred to as a Configurable Logic Block (CLB), comprised of one or more *slices* that each contain small 4- or 6-input function generators called Look-Up Tables (LUTs), several small switching devices called multiplexers

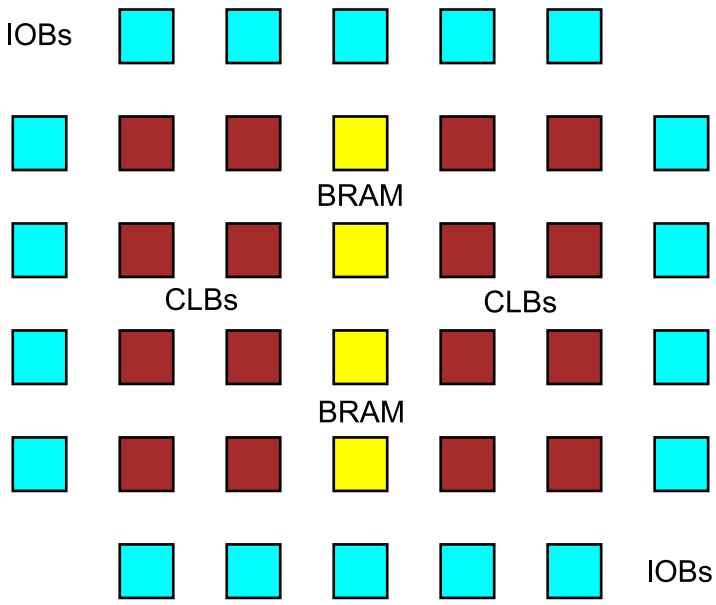


Figure 2.5: FPGA: tiled CLBs and BRAM surrounded by I/O banks

(MUXes), sequential storage elements called Flip-Flops (FFs), and other basic logic gates to enhance connectivity between CLBs. Figure 2.5 shows a basic physical layout of CLBs, BRAM and I/O banks (IOBs) that comprise a Xilinx Virtex FPGA. "Sequential" in this context refers to storage operations based on a periodic clock signal. Due to the configurability and different implementations of the CLB in Xilinx FPGAs, the CLB block diagram in Figure 2.4 is not indicative of all possible component interconnections, but provides an overview of its general hierarchy. Each block is designed to be configured in tandem with the other CLBs on the FPGA fabric to create nearly any type of digital hardware. A user can specify any arbitrary n-input boolean function to be implemented within the LUTs with lower delay than cascaded logic gates. The switching capabilities of MUXes can be used to combine LUTs in a single CLB into boolean functions of up to 8 inputs in the Virtex-II Pro FPGAs. FFs are useful for storing data and system state information. An FPGA is "reconfigurable" in that the parameters defining each configuration can be changed at a later time, potentially producing an entirely different hardware design.

Top-of-the-line development boards that feature FPGAs are generally more expensive than common personal desktop and laptop computers one might find in a home or office. This is justified by the number of hardware designs a single development board can encapsulate in its FPGA. Smaller development boards exist that are quite affordable (often costing far less than a common desktop computer) and provide functionality similar to their larger counterparts. Xilinx provides various hardware components and software tools to combine IP components into a working system. FPGAs are ideal for fast prototyping of nearly any hardware design, including multiprocessor networks.

Emulation versus Simulation

It should be made clear that FPGAs provide hardware emulation. Emulation is defined as striving to match the operation of the original object or process. In this case, emulation is provided by hardware. The CLBs within the FPGA fabric match the logic functions found in a statically fabricated IC of the same circuit design. Simulation, on the other hand, is the attempt to reproduce a subset of functions of the original object or process in software. In other words, a simulation only needs to present a similar end result of a target hardware design. Simulators are commonly run on an entirely different set of hardware than the target hardware.

In the recent past, general processing was used to determine the viability of hardware designs through software simulation due to the scarcity of multi-core architectures. There are clear advantages in software simulation, such as the availability of capable platforms and wide system observability. Given enough resources, a personal desktop computer can run a simulation. It is true that there is generally a greater number of computers that can run a simulator than there are available multi-core architecture platforms. While simulation is still a useful design practice of small hardware circuits and small-scale multi-core research, simulations run more slowly as design complexity increases. Therefore, software simulation

is insufficient in providing timely and reliable results that FPGA-based hardware emulation can provide for large hardware designs. For example, cycle-accurate simulators offer simulations of microarchitectures on a clock-cycle-by-clock-cycle basis. Each step of this type of simulator presents results that imitate the results that the target hardware would produce in one clock cycle. Accuracy is increased versus other types of simulators, but also incurs more computational overhead due to its complexity. Large-scale hardware design simulation does not suit such a simulator very well.

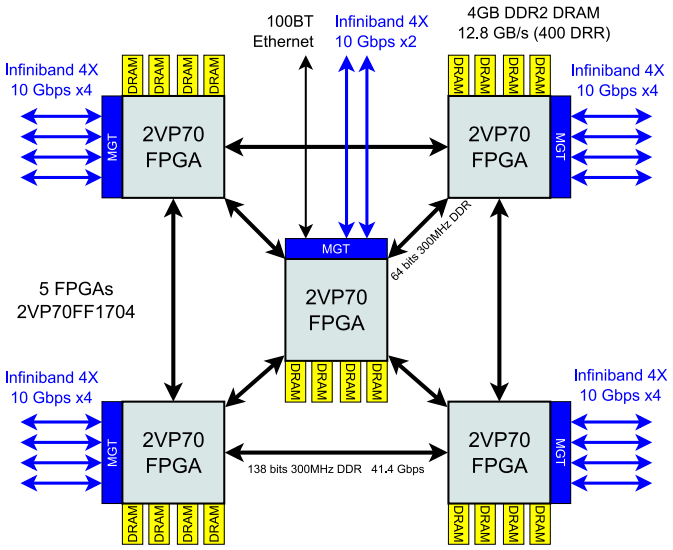
2.2 Related Work

It is arguably clear that software depends on hardware. Parallel software development is handicapped in the absence of hardware capable of efficient parallelism. How is this efficiency defined? Perhaps, only time (and some trial-and-error) will tell. Several research projects currently underway are investigating multiprocessor systems using various methods. Most of the research projects mentioned here use FPGAs in one way or another as evidenced by the ability of the FPGA to assume different hardware designs. FPGAs also offer very low power consumption relative to other general processors of comparable size, a fact touted by several of the following research projects as a means to further the use of FPGAs in multi-core research by reducing the cost of ownership.

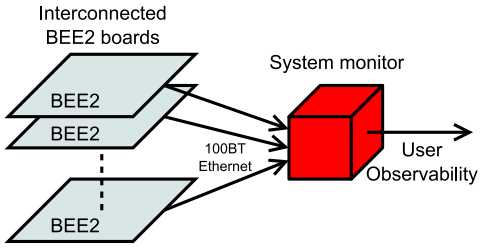
UC Berkeley's Research Accelerator for Multiple Processors (RAMP) project aims at eventually providing a reusable, quickly prototyped and massively scaled multiprocessor platform to universities and businesses. Ideally, the project would emulate the performance of supercomputers for custom hardware and software designs [15, 16]. All variants (Red, White and Blue) use the Berkeley Emulation Engine 2 (BEE2) as the host development board, designed in-house at UC Berkeley and shown in Figure 2.6a. The BEE2 has impressive communication bandwidth (on the order of 40 Gbps between the four *compute* FPGAs, and 20 Gbps off-board originating from the fifth *control* FPGA) and 4GB of local memory for each of

the five FPGAs. Each RAMP variant focuses on different applications and research areas. RAMP Red, for example, is designed to examine and develop transactional memory (a form of shared memory protection), while RAMP Blue is designed for massive multi-core research using message passing techniques such as MPI (described in Section 2.1.2). RAMP seeks to accelerate multi-core research in both academia and the corporate world. However, the system requires users to learn a custom language called "RAMP Design Language" (RDL) to connect components designed in most commonly used high-level languages such as C/C++ or HDLs like VHDL and Verilog. The BEE2 board is also a required component. This system is designed to customize hardware for a specific large-scale application using the RDL language, but currently no clear method exists for developing new applications for such an architecture (most of their technology is still in development). Seemingly, applications are to be developed using separate and established source code editors and "glued" together using RDL.

The University of Toronto has proposed a comparable platform in scope to the BEE2 for FPGA-based supercomputing [14]. The design, called the Toronto Molecular Dynamics (TMD) machine (though no longer restricted to such a specific application) consists of nine fully-connected FPGAs to be implemented on a Printed Circuit Board (PCB) in a fashion similar to the BEE2, using aggregated serial MGTs between FPGAs on a board and comparable serial connectivity between boards. Each FPGA would contain any number of embedded processors and undefined "computing engines" connected by any network topology specified by the designer of an application, restricted only by the FPGA resources. Given that soft-core processors such as the MicroBlaze can only support a limited number of connections in any of its connectivity options (described later in Chapter 3), it is unrealistic to propose the ability to apply an arbitrary network topology between computing engines within an FPGA using unaltered IP cores. A custom MPI library is proposed for communication between computing engines using message passing. A tool flow for application development is described with the intention of hiding communication details from the software designer



(a) The BEE2 board



(b) A simple RAMP example

Figure 2.6: RAMP Project block diagrams

through standard protocols. The tool flow specifics are generally defined as being manual in nature: applications are written in C/C++, parallelized explicitly by hand, and implemented in a distributed fashion among the processors. Current documentation has described a very limited proof-of-concept application running on two AP1100 FPGA development boards built by Amirix, merely indicating that programming separate processors to communicate between separate FPGA boards is possible. No other hardware has currently been produced.

Among the fabricated multi-core projects is MIT's RAW microprocessor [17, 18], a tiled multi-core processor based on a proposed "KILL" rule for multi-core (Kill If Less than Lin-

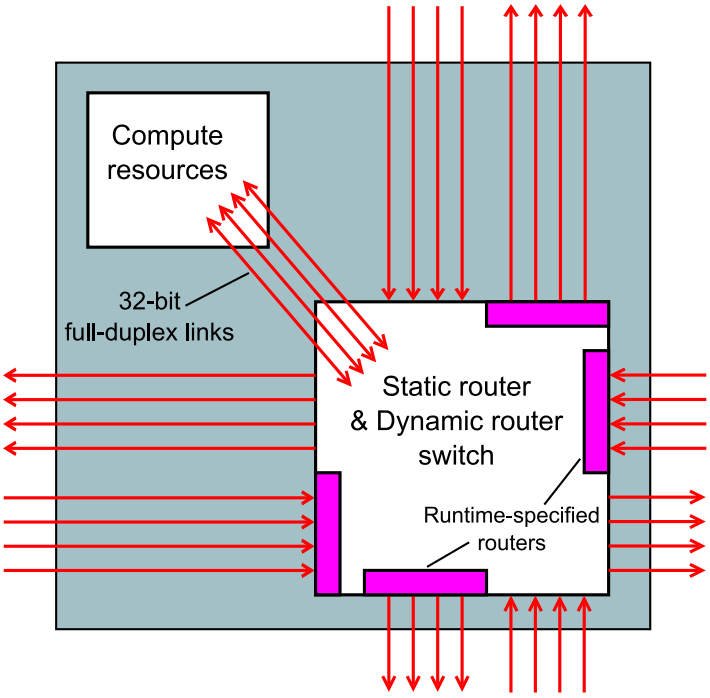


Figure 2.7: A single RAW tile

ear). This rule states that a design should be "killed," or redesigned, if the performance doesn't increase at least 1% for every 1% increase in die area used for a processor or compute core [19]. Figure 2.7 shows the basic configuration of a single RAW tile. This processor is also driven by the goal of reducing average wire length inside the processor chip to the size of one tile, equal to the length a delay would take in one clock cycle. The RAW microprocessor has been commercialized with a spin-off company called Tilera, and is currently shipping a 64-tile version of RAW. A specific cross-compiler for the processor, RAWCC, is designed to identify and separate parallelism within source code already written in C/C++ and other high-level languages, recruiting specific tiles to run parts of the application.

The University of Texas at Austin has developed a redesigned processor architecture called Explicit Data Graph Execution (EDGE) with the motivation that current RISC processors will not be able to scale properly in the future of multi-core. The redesign focuses

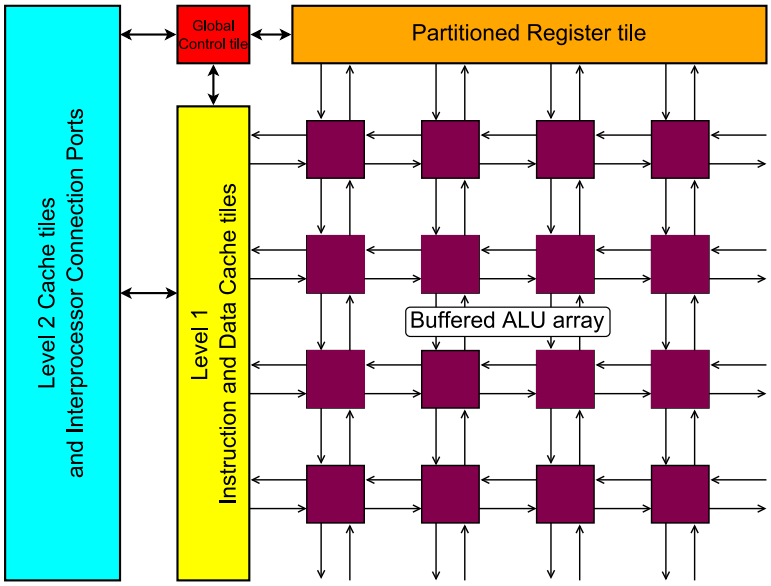


Figure 2.8: TRIPS Overview of an EDGE processor

on attacking the problems of inadequate concurrency, power efficiency on an instruction-by-instruction basis, and execution delay among distributed processors [20, 21]. Each tiled EDGE processor is further subdivided into specialized tiles. Some of the tile types include small instruction and data cache tiles, a global control tile per processor, and a 4×4 array of execution tiles consisting of buffered ALUs. Each execution tile concurrently processes streaming instructions defined by a custom compiler. Several EDGE processors are to be combined into a scalable, tiled multiprocessor called TRIPS, "the Tera-operation, Reliable, Intelligently adaptive Processing System." The custom compiler is called Scale, developed in order to efficiently utilize the new architecture and instruction set and accept common high-level languages such as C and FORTRAN. The compiler exposes parallelism by creating large blocks of TRIPS-readable code from sequential high-level source code, spatially distributing them within the ALU array of each EDGE processor in a way that reduces the amount of memory access and inter-node communication. Related instructions are placed into ALUs that are close to each other in order to reduce routing and communication delays. In addition, each processor core shares blocks of external cache tiles that act as system-wide

shared memory. While the TRIPS processor and compiler attempt to produce as much parallelism as possible out of sequential high-level algorithms, both are still in development. A two-processor fabricated prototype is currently being used for performance analysis.

In the middle of the FPGA-based and simulator-based multiprocessor research spectrum lies PROToFLEX, currently being developed at Carnegie Mellon as an FPGA-based emulation/simulation hybrid platform that intends to support large-scale multiprocessor hardware and software research [22]. Emulation and simulation is combined into an execution method called *transplanting*. The workstation provides the software simulation portion of the complete system while being connected by Ethernet to an FPGA host that provides hardware acceleration, including system-wide shared memory. Figure 2.9 shows how the workstation and FPGA interact. During system execution, processes can be "transplanted" between the FPGA and simulation workstation based on available or required functionality. Complex functionality such as I/O is provided by the workstation and common functionality such as Arithmetic Logic Units (ALUs) on the FPGA. PROToFLEX also uses a multi-core abstraction in what are referred to as *emulation engines* to simulate a multi-core environment on the FPGA. Each hardware-based emulation engine offers hardware-based instruction pipelines (emulating a compute node) that provide a simulation abstraction for a much larger contingent of processors. It is unclear from their documentation whether the emulation engines are allocated to the FPGA fabric prior to or during system operation. The motivation for PROToFLEX is in reducing the effort of building a fully-emulated multi-core environment by augmenting such hardware with software simulation. The shortcomings of simulation as compared to hardware emulation have been addressed in Section 2.1.4.

Within the realm of pure simulation, one will find Cmpware^{TM} as a simulation environment for multi-core software development [11, 12]. Cmpware offers the ability to accurately simulate many processors that might be implemented in an FPGA-based multi-core design such as the PowerPC and MicroBlaze processors. Many types of communication media can

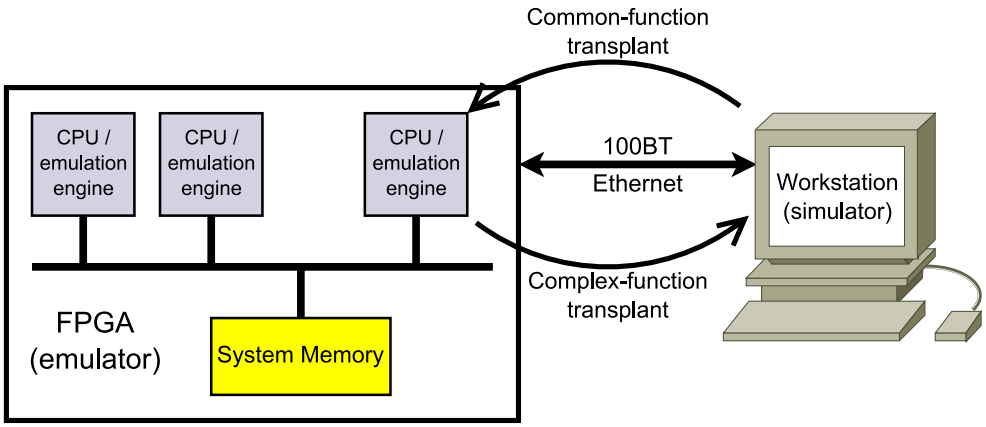


Figure 2.9: PROToFLEX Overview

be simulated in Cmpware between compute nodes. The goal of this simulation environment is to provide the ability to develop software for multi-core architectures. While having access to the hardware of the architectures beats a simulator in scalability, a proper simulation environment obviates the time-consuming process of building and placing a large design onto FPGAs.

Other research efforts are not primarily platforms upon which to develop parallel software applications, but instead focus on efficiently running past and current applications on multi-core architectures. The University of Michigan has developed a platform called Voltron that attempts to efficiently use the hybrid parallelism within single-threaded applications which would either run concurrently in lock-step among the cores, or in an asynchronous distributed fashion [23]. Stanford's ATLAS project offers hardware-based transactional memory support (a newer technology that places particular rules and controls upon shared memory access) for shared-memory multi-core architectures [24]. MIT's RAW microprocessor might be included in this list of projects, as would be any research effort that uses compilers to extract the lion's share of parallelism from otherwise sequential algorithms. Some research projects claim to provide a combined solution: platforms that run past applications without redesign, as well as providing the means to develop parallel software for the future. There are a few

older projects that developed alternative shared-memory architectures such as Stanford's Hydra CMP [25] and Smart Memories [26]. Other projects attempted to refine small-scale parallelism among sets of instructions through methods of ILP including University of California's Synchroscalar [27] and University of Washington's WaveScalar [28]. Commercial multi-core processors from vendors including Intel [29] and Sun Microsystems [30, 31] have a vested interest in providing processors that run current and past applications while expanding parallelism for future software.

From a commercial point of view, the inclusion of point-to-point connectivity found in HyperTransport™ technology in the AMD multi-core processors such as the Operton line [32] and the Cell Broadband Engine from IBM, Sony and Toshiba [8, 33] that is used in Sony's Playstation 3 gaming console are steps in the right direction regarding the future of multi-core. HyperTransport is a technology governed by the HyperTransport Consortium [34], offering an asynchronous on-chip, low-latency, high-bandwidth interconnect that is compatible with other common communication methods such as the Peripheral Connect Interface (PCI). In a general sense, it is similar to the Xilinx FSL described later in Chapter 3.2.2. HyperTransport allows more flexible communication methods beyond expanding the current uniprocessor architecture by merely connecting multiple processors to a common bus. However, though AMD's Operton processors and the Cell processor use HyperTransport, they still primarily rely on a shared system bus and shared memory that incur latency-heavy arbitration and access times.

2.3 Summary

While many efforts attempt to fill the parallelism void and keep up with Moore's Law, most architectures do not reflect what will likely be the future of multi-core. Shared memory access, bus arbitration and memory contention methods are expensive in clock cycles.

Table 2.1: Multi-core Platform Summary

Metric	Quality					
Project	RAMP	RAW	TRIPS	TMD	PROToFLEX	Cmpware
high scalability	5	4	2	5	3	4
high bi-sectional bandwidth	5	5	3	5	1	var.
high parallelism	5	4	4	5	3	5
fast prototyping	4	2	2	4	5	5
portability	1	2	1	2	4	5

Both shared-memory and common-bus architectures yield smaller bi-sectional bandwidth than meshes and highly-connected topologies. Long wires consume excessive power. Consequently, rings and bus networks are less than ideal. Software abstractions for communication shouldn't require a large amount of overhead such as the Message Passing Interface (MPI) and the software layers required for using Ethernet and the TCP/IP stack. The means of communication itself should be invisible to the software. Communication bandwidth and media should scale with the size of the network and/or the complexity of the application. While a multi-core architecture shouldn't *require* special tools or force a developer to learn a specific language to be able to run a given application, it helps to have a software development environment that facilitates inherent parallelism by design rather than resorting to an operating system or compiler that hopefully utilizes the parallel resources of the architecture. Reliance upon compilers to produce parallelism results in resistance toward improvements in software design methodologies [35]. Finally, the architecture design should be portable to other FPGA and development board types in order to facilitate the speed of prototyping such hardware designs. Table 2.1 summarizes the previous research efforts that focus on massive multi-core prototyping for software development platforms. TMD is evaluated as if the proposed architecture and PCB were currently implemented. Quality grades are evaluated in assuming each project implements around 1,000 processors. In the scale from 1 to 5 that is used, 1 represents poor quality and 5 signifies magnificent quality from the perspective of

the suggestions given in [5–9] that are currently seen to be the direction that development of multi-core platforms should take. "Var." indicates a quality that proves to be too flexible in a particular project to be given a definitive grade. It should be reiterated that there is no definitive multi-core hardware platform, nor a definitive communication method or medium. Therefore, some subjectivity is inherent to any grading scale applied to the metrics in the table.

Chapter 3

Design Overview

This chapter describes the overall design of the multi-core architecture presented in this thesis while providing insight into design limitations. Software development and debugging are briefly covered.

3.1 Fixed Network Topologies

For years, hardware designers have analyzed several types of network topologies to determine their advantages to particular applications or application types. There is no particular network that best implements all applications, so there must be compromises between the topology and the application. Some topologies include:

Linear Each processor, barring the first and last, share a connection to two other processors.

Ring A linear network with the first and last processors connected to each other. Such a connection is referred to as a *wraparound* or *torus*.

Bus A network of processors each connected to a common data path (commonly used in Ethernet-based networks).

Star Several processors each connect solely to a central processor.

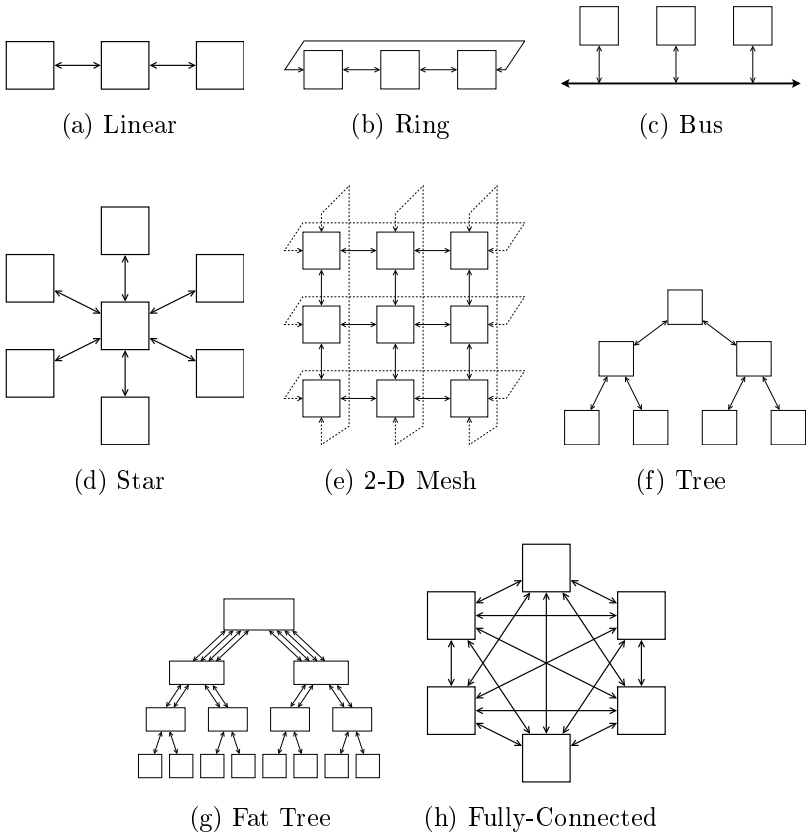


Figure 3.1: Basic topologies

2-D Mesh A two-dimensional extension of the linear or ring topologies wherein each processor connects to four other processors.

Tree Only one path exists between any two processors.

Fat Tree An extension of the tree topology: the number of communications links at each level of the tree is a function of the number of processors in the network to alleviate communication bottlenecks near the root (uppermost) processor.

Fully-Connected Each processor connects to all other processors in the network.

Different topologies work well for particular applications such as the bus network for Ethernet-based operations. A project at the University of Toronto conducted a study for

routability of particular network topologies implemented with FPGA resources [36]. Largely-connected networks provide for very flexible communication options, but require a great portion of the total supplied power to operate due to the length of the wires involved. For parallel computation, a two-dimensional mesh eliminates the need for a common bus, resulting in many more shorter wires that require less power and offer shorter delay times from processor to processor throughout the network, providing the best balance of performance and power efficiency [5]. Regarding the evaluation of network topologies, some definitions are in order:

Diameter is the maximum distance between any two processors in the network. Smaller diameter translates into fewer intermediate processors between those that wish to communicate.

Degree is the number of connections to a single processor.

Bi-section Width is the minimum number of communication links that have to be removed to partition the network into two roughly equal halves.

Channel Width refers to the number of bits that can be communicated simultaneously over a link connecting two processors equalling the number of physical wires in each communication link.

Channel Rate is the peak rate at which a single channel can deliver bits.

Channel Bandwidth is the capacity at which data can be communicated between the ends of a communication link: the product of *channel rate* and *channel width.*

Bi-section Bandwidth refers to the minimum volume of communication allowed between any two roughly equal halves of a network: the product of *bisection width* and *channel bandwidth.*

Cost can be defined in many ways: the number of communication links, the number of wires required by the network, the bisection bandwidth, etc.

Table 3.1: Fixed Topology Network metrics

Network Type	Diameter	Bi-section Width	Min Degree	Max Degree
Linear	$p-1$	1	1	2
Ring	$\lfloor \frac{p}{2} \rfloor$	2	2	2
Bus	1	1	1	1
Star	2	1	1	$p-1$
2-D Mesh (non-torus)	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	4
2-D Mesh torus	$2\lfloor \frac{\sqrt{p}}{2} \rfloor$	$2\sqrt{p}$	4	4
Binary Tree	$2\log \frac{p+1}{2}$	1	1	3
Fat Tree	$2\log \frac{p+1}{2}$	$\frac{p+1}{4}$	1	$\frac{p+1}{2}$
All-connected	1	$\frac{p^2}{4}$	$p-1$	$p-1$

Table 3.1 summarizes the Diameter, Bi-section Width and Degree of the above topologies. The value p refers to the number of processors within the network. These metrics are generally considered to be the most pertinent goodness measures of a topology. A square mesh network was chosen for this project due to its relatively high bi-sectional bandwidth as compared to the other topologies, and for its inherent scalability in two or even three dimensions. Bi-sectional bandwidth increases by merely adding more processor cores to the network, and more FPGAs could add their own array of processors to expand the mesh. The nodes in a mesh also have the advantage of having many paths from which to obtain data (a degree of 4). This reduces the possibility of a performance bottleneck, such as a situation where several nodes are transmitting data through a single channel or processor that cannot handle all of the data at once.

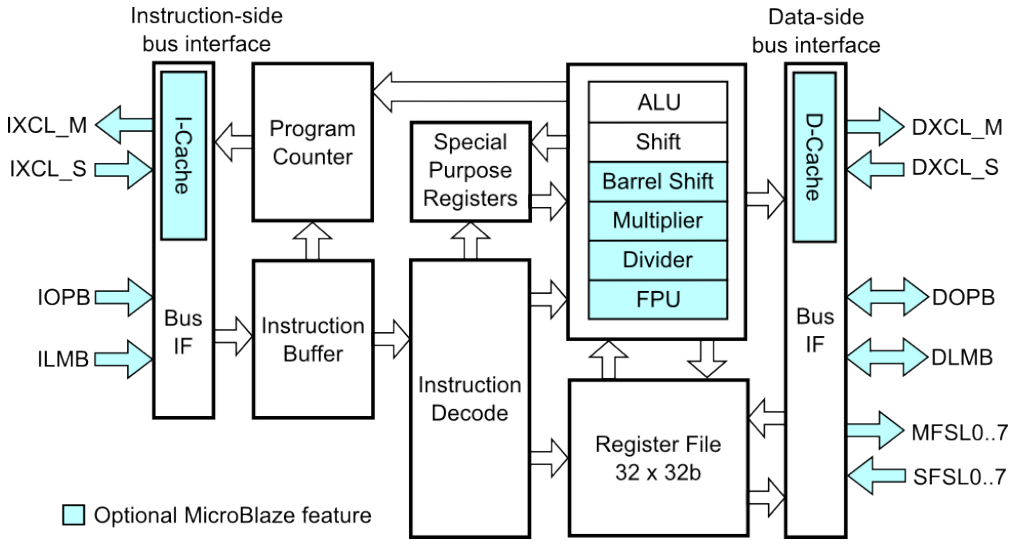


Figure 3.2: MicroBlaze Core Block Diagram

3.2 Components of the Architecture

3.2.1 Processor

The Xilinx proprietary MicroBlaze™ is a 32-bit RISC softcore processor with optimizations for Xilinx FPGAs, with many parameterizable configuration options [37]. It was chosen for its configurability, simplicity, and relatively small footprint on the FPGA fabric in terms of resource utilization. It is capable of handling several types of hardware exceptions and interrupts, can connect to the data and instruction side interfaces of 32-bit On-chip Peripheral Bus (OPB), provides up to 8 more distinct full-duplex communication ports, and can utilize a separate Local Memory Bus (LMB). Configuration options exist for adding a floating-point unit (FPU), hardware-based multipliers, and other components. The considerations regarding memory distribution for this architecture are described in Section 3.3. Processor performance is commonly determined by the multiple of *millions of instructions per second* (MIPS) that can be processed. However, the performance in MIPS based on a particular computation does not translate well across different instruction sets. A more

meaningful benchmark is the Dhrystone, developed in 1984 in software as an iterative loop and originally run on a 1 MIPS machine called the VAX 11/780. This computer produced 1,757 Dhrystones (iterations of the loop in one second), yielding a baseline metric for Dhrystone MIPS. According to Xilinx, the MicroBlaze processor is capable of 138 Dhrystone MIPS running at its maximum clock speed of 170 MHz on a Virtex-II Pro FPGA, or approximately 0.8 DMIPS/MHz.

3.2.2 On-Chip Communications

How should MicroBlaze processors communicate? Like many of the research projects described in Chapter 2.2, shared memory is an option. The Xilinx ML310 FPGA development board offers 256 megabytes (MB) of OPB-accessible external Double Data Rate (DDR) Static Random Access Memory (SRAM). DDR memory produces data on both the rising and falling edge of a clock signal. Shared memory has the advantage of keeping global data in one place for access by all components given proper data contention management. However, the OPB incurs arbitration overhead that increases as the number of OPB connections increase. Also, external shared memory access times are relatively slow compared to internal local memory. These result in significant limitations for multi-core networks of hundreds of processors. Local memory is an alternative provided by block RAM distributed through the FPGA and is connected to each processor by the LMB that provides low latency for reads and writes without arbitration. BRAM can be used as shared memory through the OPB, but has similar scalability limitations as those described for external SRAM.

Another option for inter-processor communication is a direct connection link provided by the Xilinx Fast Simplex Link (FSL). The FSL is a uni-directional, point-to-point, 32-bit (or 33-bit) FIFO-style connection between two components [38]. Each FSL contains a Master (input) and a Slave (output) interface for accessing the internal FIFO. The master and slave signals of the FSL interfaces are indicated with an "`M_`" and "`S_`" prefix, respectively. The

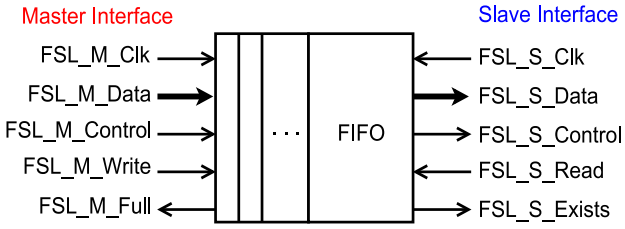


Figure 3.3: The Fast Simplex Link

Table 3.2: FSL Access Functions

Blocking Write Blocking Write w/ Control bit	MicroBlaze stalls if FSL is full
Blocking Read Blocking Read w/ Control bit	MicroBlaze stalls if FSL is empty
Non-blocking Write Non-blocking Write w/ Control bit	MicroBlaze does not stall, regardless of FSL status
Non-blocking Read Non-blocking Read w/ Control bit	

FSL is highly configurable in terms of FIFO depth, clock domain specification, implementation in BRAM or logic fabric, and other options. The FSL can operate in system-synchronous or source-synchronous mode, the latter allowing potentially different clock rates for the Master and Slave interfaces. The FSL also provides a total latency between a write performed by a sending node and a read of the same data performed by the receiving node of as little as two system-synchronous clock cycles. Xilinx provides assembly macros that allow C/C++ software to access the FSLs using simple function calls in *blocking* and *non-blocking* versions and the option to propagate a Control bit along with the data (yielding the total width of 33-bits mentioned earlier). Table 3.2 summarizes the functions. A blocking write will cause the MicroBlaze to stall if the FSL's FIFO is full, as will a blocking read performed on an empty FSL.

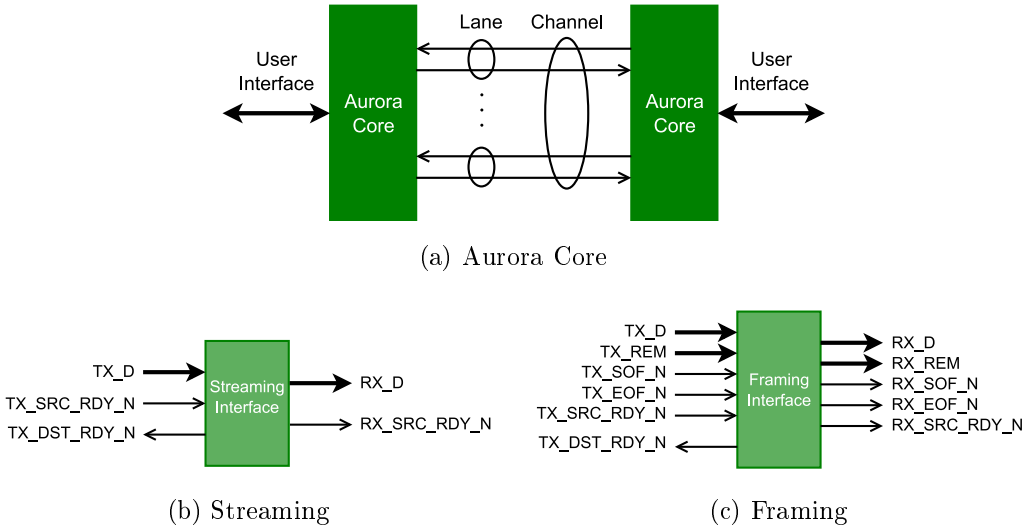
3.2.3 Off-Chip Communications

From a software perspective, off-chip communications should be transparent, easily connected to the on-chip communication medium, fast, and easy to use. As Chapter 2.1.3 described, serial I/O is preferred for its potential speed and low design complexity. These qualities were found in Xilinx's Aurora core, a proprietary gigabit-capable, self-synchronous, serial communication protocol. The core can be customized to add communication ports for other components emanating from the FPGA fabric, much like the FSLs. Using Aurora in conjunction with the FSLs, the software doesn't concern itself with off-chip communication semantics that protocols such as high-overhead Ethernet would require. The Virtex-II Pro XC2VP30 FPGA contains eight MGT cores, each capable of transceiving at a rate of 3.125 Gbps [39, 40]. Aurora directly connects to and utilizes the MGTs in two modes of operation: framing mode and streaming mode, each with either a 16-bit or 32-bit interface. Framing mode alone offers two different flow control options: Native Flow Control (NFC) and User Flow Control (UFC). More information regarding Aurora usage is given in Chapter 4.2.2. In the figures presented in this and following chapters, note that the "_N" suffix refers to an active-low signal, asserted by applying logical 0.

In terms of serial communications using MGTs, a *channel* is comprised of one or more *lanes*. Each lane is operated by and corresponds to a single MGT with both transmit (TX) and receive (RX) functionality (see Figure 3.4a). To obtain a wider datapath, each lane can be "bonded" to and operate in parallel with other lanes to form a channel. Both framing and streaming modes support channel-bonding.

3.3 ML310 and Virtex-II Pro 30 Resources

The multi-core architecture was implemented on the Xilinx ML310 Embedded Development Board containing a Virtex-II Pro XC2VP30 FPGA with more than 30,000 CLBs, 272



(a) Aurora Core

(b) Streaming

(c) Framing

Figure 3.4: The Aurora Core's Streaming and Framing Interfaces

kilobytes (KB) of BRAM, dual IBM PowerPC^{TM} 405 processors, eight RocketIO MGTs, a 256MB DDR SRAM DIMM, and common I/O peripheral attachments [41]. As described in Chapter 2.1.4, LUTs, FFs and BRAM are the primary measure of resource usage.

A mesh with as many processors as can fit on an FPGA with as much functionality and off-chip connectivity was desired. MicroBlaze processors with a small FPGA footprint and sufficient computational power, FSLs for buffered inter-processor communication, and speedy and simple Aurora to round out the off-chip communications provide a hardware designer with nearly all the tools required to create a scalable and portable multi-core platform for parallel software development. Design decisions based on combined component resource requirements in comparison to the available resources of the XC2VP30 FPGA are described in Chapter 4.

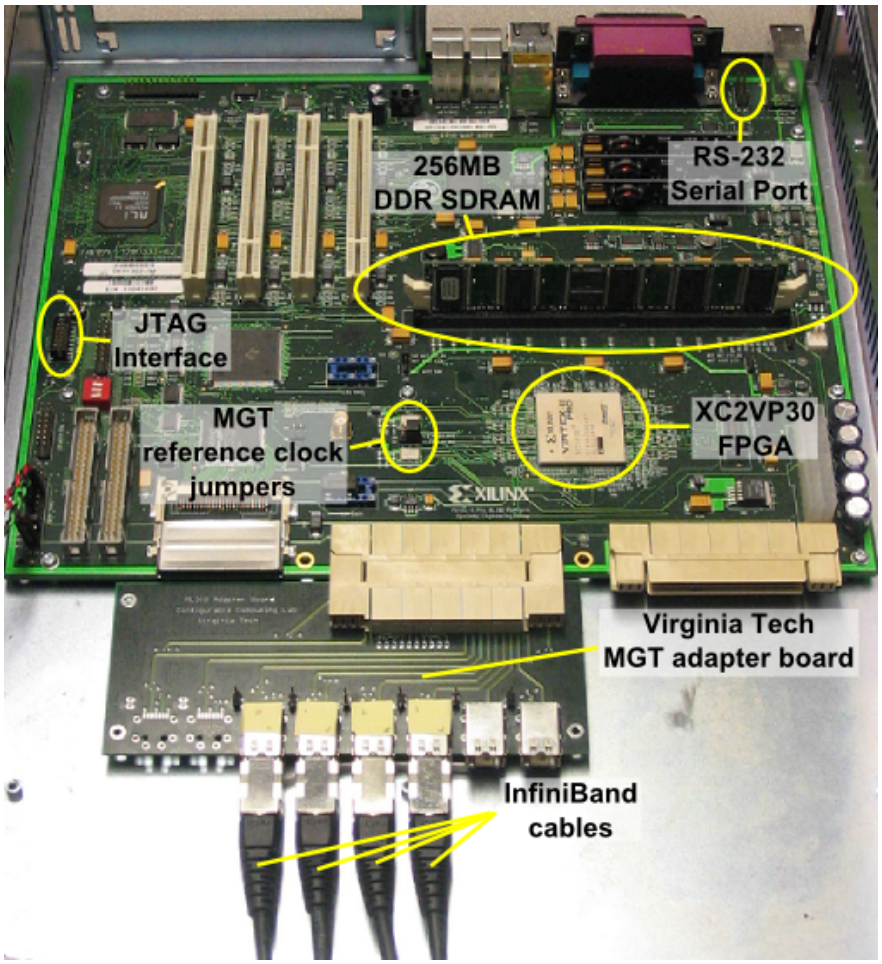


Figure 3.5: The Xilinx ML310 Embedded Development Board

3.4 Software Development

The MicroBlaze Debug Module (MDM) IP is a robust software debugging option for the MicroBlaze that connects directly to the processor's software-invisible debug port. The Xilinx Microprocessor Debugger (XMD) connects to the MDM from a separate computer through the ML310's Joint Test Action Group (JTAG) cable and interface that is also used to load a design onto the FPGA fabric. The MDM can support up to eight MicroBlazes to debug in parallel. This is a limitation for multi-core designs of more than eight processors on a single FPGA. For a small 2×2 mesh of processors, it is sufficient to use the MDM

as the primary debugging tool. However, neither the MDM nor the OPB can support the number of processors in larger square meshes. Therefore, an alternative option would be required that allows all of the MicroBlazes in a given topology to communicate some type of information to the software developer (possibly through simple print statements embedded within a program using the RS-232 serial port). Such a communication scheme would require special software commands to access the processor's internal registers, unlike the simple "read memory"-like commands available through the MDM. As mentioned in Chapter 2.1.4, software simulation is also an option for developing applications. Cmpware is an attractive choice for a distributed memory multi-core architecture.

Chapter 4

Implementation

This chapter describes the FPGA-based multi-core architecture design in more detail, covering component design, usage, and architecture debugging methods in both hardware and software.

4.0.1 Software Tools for Hardware Development

Xilinx, Inc., provides a suite of software for developing systems on their boards and corresponding FPGAs that allow access to their proprietary and third-party components. Custom components and IP can be added to hardware designs through the software tools as well. The Embedded Development Kit (EDK) provides the Xilinx Platform Studio (XPS) graphical user interface as the main software tool for embedded systems development. The multi-core architecture presented in this thesis is comprised of a combination of Xilinx proprietary components and custom IP using EDK version 9.1.02i. The terms "component" and "IP" are hereby used interchangeably.

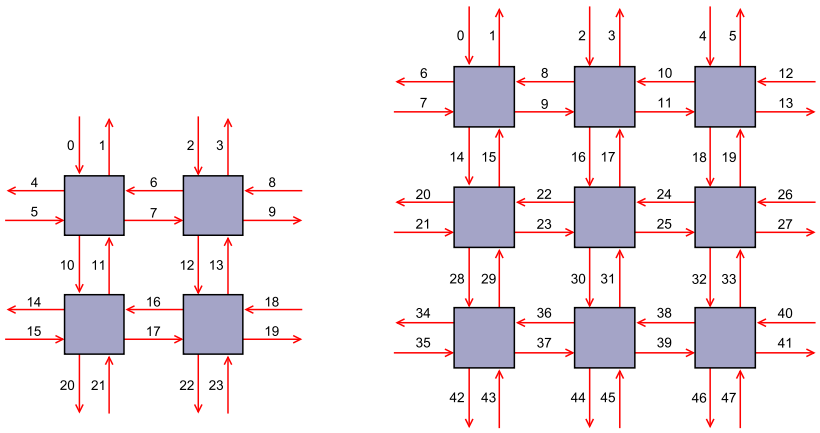
Table 4.1: Known Component Resources

Component	LUTs	FFs
Virtex-II Pro XC2VP30	27,392	27,392
MicroBlaze basic	1,570	704
MicroBlaze w/ OPB	1,647	916
MicroBlaze w/ FPU	2,620	1,302
MicroBlaze max	3,096	1,848
FSL synchronous (mesh)	42	7
FSL asynchronous (mesh)	84	12
FSL synchronous (UART)	57	12
Aurora (approx)	3,142	2,992

4.1 Design Considerations

4.1.1 Processors and Memory

XPS provides a default configuration for each component, generally yielding a standard resource usage value of Look-Up Tables, Flip-Flops and BRAM defined in Chapter 2.1.4. Table 4.1 shows resource requirements for the main mesh architecture components in various configurations. These values imply limitations that need to reconciled for any hardware design based on the resources of an FPGA. The basic MicroBlaze configuration consists of an expanded set of instructions for accessing the Machine Status Register (MSR), a 32-bit hardware multiplier, and a pattern comparator. The entry labeled "MicroBlaze max" refers to the inclusion of five FSL ports (four for the mesh interconnectivity, and one for an FSL-based RS-232 UART IP), the Floating-Point Unit (FPU), a single full OPB connection, the debug port, and the Processor Version Register used for processor identification. The necessity for the addition of the FPU is discussed at the end of this section and the other components in Section 4.2.4. Network-wide homogeneity and symmetry among processors



(a) 2×2 (b) 3×3

Figure 4.1: Basic 2×2 and 3×3 Meshes and the FSLs required

promotes simplicity and facilitates scalability. As described in the previous chapter, a square mesh provides a good balance of processing potential and power efficiency. The two smallest square mesh sizes, 2×2 and 3×3, were implemented based on the total resources available on the XC2VP30 FPGA. Figure 4.1 shows that 24 FSLs would be required for a 2×2 grid, and 48 for a 3×3 grid of processors. Given these two mesh sizes, it can be concluded from Table 4.1 that the FSL and Aurora cores collectively take up far less space on the FPGA than the combined resource utilization of the MicroBlaze processors. A reasonable claim would be that the processors alone pose the major limiting factor regarding the size of the design for a single FPGA. The number of processors in a 4×4 grid would greatly stretch, if not exceed, the resources of a single XC2VP30 FPGA. This size mesh also greatly limits the amount of local memory that would potentially be allocated to each processor. In fact, the XC2VP30 has room for 16 processors in their basic configuration, but restrictions on local memory size allocation prevents a 4×4 from being a useful implementation.

Memory connectivity and size are limited by the LMB-based BRAM controller interface. 64KB of BRAM is the largest local memory size that can possibly be allocated to a single

Table 4.2: Memory size choices

Mesh size	Memory sizes in KB	Notes
2×2	8	-
	16	-
	32	best balance
	64	BRAM 94% used
3×3	8	-
	16	most memory possible
	32	exceeds resources

processor in any configuration. In addition, memory sizes less than 64KB are only available in multiples of 8KB due to the particular manner in which the BRAM connects to a processor through the LMB[1]. Therefore, the total amount of BRAM available to processors on the FPGA is 256KB, yielding an implementation size of a maximum of four processors if each processor is allocated the full 64KB of memory. The relation

$$(\text{memory size in multiples of 8KB}) \leq 64 \text{ KB},$$

provides a guideline for the acceptable amount of allocated memory for each processor. If

$$(\text{acceptable memory size in KB}) \times (\text{number of processors}) < 256 \text{ KB},$$

then implementation is possible. Table 4.2 lists the possible memory sizes per processor available on the XC2VP30 for both mesh sizes. 16KB of BRAM is available to each processor in the 3×3 mesh with BRAM to spare, but 32KB per processor exceeds BRAM availability. Each processor in the 2×2 mesh would be allowed to use 64KB, though this would nearly exhaust the available BRAM. Considerations must be given to the resource usage of other components. Different memory sizes *cannot* be allocated on a per-processor basis. XPS and

[1]Other components do not have this restriction; BRAM usage for each non-processor component is described in Sections 4.2.1 and 4.2.2.

the Xilinx toolchain require all processors to have the same amount of memory.

Many real-world applications require floating-point calculations. The resources of this particular FPGA does not allow *all* of the processors to be configured to use the FPU in the 3×3 grid, as the resources needed exceeds availability. This results in a lack of desired functionality that the 3×3 mesh architecture might provide. However, the ability to port this design to another FPGA with more resources justifies nearly any decision made based on the limited resources of the XC2VP30. Indeed, an advantage to such a hardware design is its adaptability to other FPGA boards. Some applications might only require a few processors to utilize the FPU, in which case the 3×3 processor architecture can be rebuilt to accommodate FPUs on up to five processors on the XC2VP30 FPGA. Such a requirement is ultimately dependent on the application.

This multi-core architecture uses distributed memory in which each processor is allotted its own local non-cache memory connected by a LMB. Communication for an application must occur in the *manner* of message passing or packet forwarding, as there is no shared memory[2]. Furthermore, regardless of the intended communication format, data is represented by a stream of 32-bit words, consumed one at a time from an FSL by a processor at the software's discretion due to the nature of the FSL's FIFO-style operation. The hardware does not consider the contents or meaning of the messages; the interpretation of the data itself is left to the software. There is no separate hardware-based router, unlike those used in the RAW microprocessor[17].

[2]However, shared memory could be added to the system, as can many components, for the need of any given application, but there are restrictions to how an added component would connect to any existing component, such as the OPB handling a limited number of processors.

4.1.2 Connectivity

The FSLs can be configured to operate either synchronously, in which both the write and read clock domains would use the same clock signal, or asynchronously. Both operation modes allow BRAM-based implementations in the latest version of the FSLs, but such an implementation includes an extra clock cycle of latency if an empty FSL is read. FSL implementation was limited to FPGA fabric usage only due to the sheer number of FSLs that would be required for either size mesh interconnect. Such BRAM usage is unnecessary. Software can access the FSLs without knowledge of the implementation type. Each FSL that directly connects to two processors operates synchronously with the system clock since both processors operate on the system clock as well. FSLs connected to the Aurora core operate asynchronously in which the processor side uses the system clock and the Aurora side uses a separate MGT-based reference clock. Two types of reference clocks exist: those generated by the ML310 board accessed through static source pins within the FPGA (fastest), or by any system clock within the FPGA hardware design (slowest) that is not produced by a Digital Clock Manager (DCM). The speed of the fastest reference clock is set by a pair of jumpers on the ML310 board to either 156.25 or 125 MHz. This reference clock is used by the Aurora core and any associated user application, then increased by a factor of 20 inside the MGT to provide the serial data rate of 3.125 or 2.5 Gbps, respectively. The MGTs can be configured to use even slower clock speeds through the use of a 10x multiplier instead.

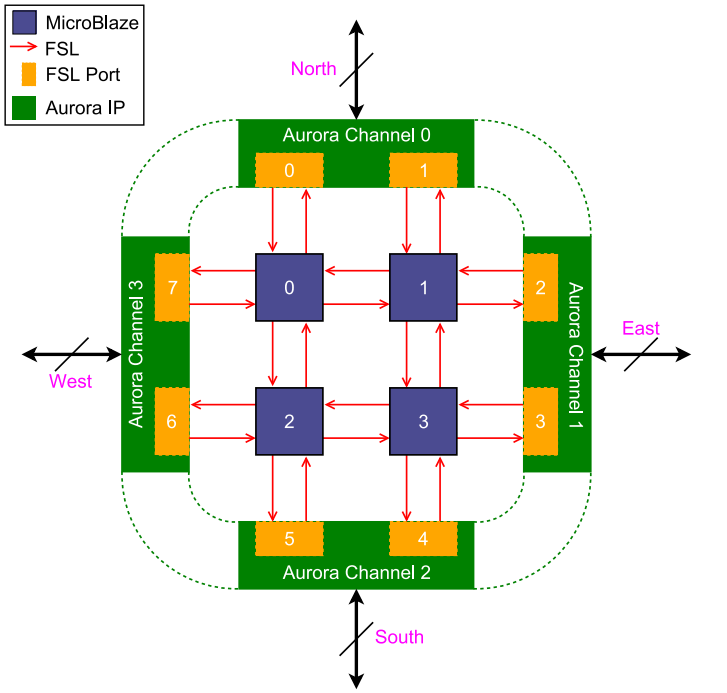
Figure 4.2 shows the connectivity of the network from the perspective of a single FPGA. The 2×2 grid contains 24 FSLs total (8 synchronous, 16 asynchronous) connecting four processors to each other in a nearest-neighbor fashion and to a relatively complex implementation of the Aurora core. The Aurora core is shown to surround the main processor network in the figure because all outgoing data must pass through this component. In the 3×3 mesh, there are a total of 48 FSLs (24 synchronous, 24 asynchronous) and nine processors with similar Aurora core connectivity. Notice that the number of MGTs and corresponding InfiniBand cables required for off-chip expansion remains the same for both mesh sizes, showing the

versatility of the network topology and Aurora IP.

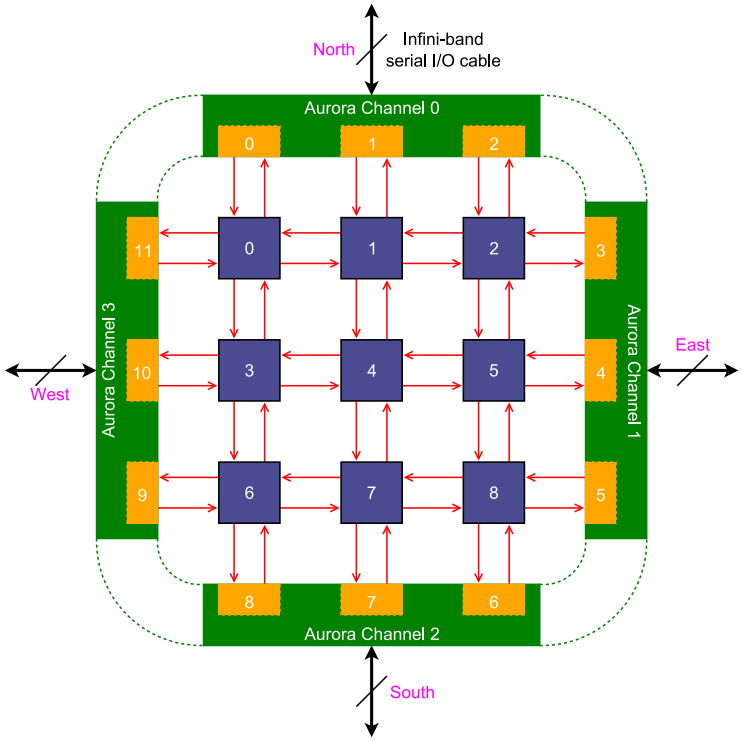
4.2 Basic Hardware Implementation

Processors in the 2×2 mesh are numbered 0 through 3 from left to right and top to bottom. The processors are numbered 0 through 8 in the same manner within the 3×3 mesh. The Aurora channels are numbered 0 through 3 in a clockwise fashion starting at the top of the grid for both meshes. Cardinal directions correspond to the Aurora channel number designations in a clockwise manner (e.g., North corresponds to channel 0, East to channel 1, etc.). A total of eight FSL ports in the 2×2 mesh and a total of twelve ports in the 3×3 mesh connect to the Aurora IP. Each Aurora channel accepts two ports and three ports per channel, respectively, per mesh. The FSL ports are numbered 0 through 7 in the 2×2 mesh and 0 through 11 in the 3×3 mesh.

Table 4.3 presents six potential versions of the final architecture implementation in regards to resources as provided by XPS. The basic architecture includes a single MicroBlaze with a full OPB connection, the rest of the MicroBlazes without an OPB connection, a particular number of synchronous and asynchronous FSLs based on the mesh size, the Aurora IP, and various other smaller components that are required by the FPGA. The next item adds FPUs to all processors as the only change. Finally, the most fully functional mesh as possible (assuming homogeneity and symmetry) is described, including all of the required FSL ports, OPB connections, and debug options. The rest of this chapter describes the design specifics of the components involved in the processor network, all of which apply to either a 2×2 or a 3×3 mesh. Both meshes are possible implementations. Note that the size of a single BRAM block is 2KB.



(a) 2×2



(b) 3×3

Figure 4.2: Full 2×2 & 3×3 Mesh Implementation

Table 4.3: Total Resources

Configuration	LUTs	FFs	BRAM (blocks)
Virtex-II Pro 30	27,392	27,392	136
Basic 3×3 mesh	20,670	10,111	72
% FPGA Utilization	75.5	36.9	52.9
3×3 mesh w/ FPUs	30,119	15,493	72
% FPGA Utilization	**110**	56.6	52.9
Full 3×3 mesh	22,464	10,859	72
% FPGA Utilization	82	39.6	52.9
Basic 2×2 mesh	11,654	6,511	64
% FPGA Utilization	42.5	23.8	47
2×2 mesh w/ FPUs	15,937	9,111	64
% FPGA Utilization	58.2	33.3	47
Full 2×2 mesh	18,283	11,018	64
% FPGA Utilization	66.7	40.2	47

4.2.1 Fast Simplex Link

As Figure 3.3 shows, each Master and Slave interface contains a data bus, a control bit, and an asynchronous clock signal. The control bit has no direct correlation to the data bus, as its context and value are dependent on its usage in software. The asynchronous clock signals provide separate clock domains for both the Master and Slave interfaces. The Master interface has `Write` and `Full` signals, while the Slave interface uses `Read` and `Exists` signals, the functions of which are described below. Signals named `FSL_Has_Data` and `FSL_Full` exist that perform the same functions as the `S_Exists` and `M_Full` signals and can be ignored if desired. However, FSL version v2.10.a allows these signals to act as interrupt sources. A `Control_IRQ` signal is also used in conjunction with these two signals for interrupt generation that can be connected directly to the MicroBlaze's interrupt capture port or through sepa-

rate interrupt controller. Allowing the FSLs to generate interrupts upon either containing valid data or when the FIFO is full is an ideal situation by allowing the processor to focus on computations and communication. The processor would handle the interrupt sources in only when the interrupt source signals are asserted. Otherwise, the processor must poll the FSL port for valid data.

Figure 4.3 shows a protocol diagram outlining the FSL's operations. Data from the Master component is applied to the FSL's input `M_Data` bus, and is accepted by the FSL on the clock cycle when its `M_Write` signal is asserted if and only if the FSL is not full (as indicated by the `M_Full` status signal). In the case where a write is performed on a full FSL, data written to the FSL is lost while the data previously residing within the FSL is preserved. The carry bit in the Machine Status Register (MSR) of the MicroBlaze processor is asserted to indicate a write error. When data is written to the FSL, the `S_Exists` flag is asserted to indicate that valid data indeed exists within the FSL. In the case where data already exists within the FSL, the `S_Data` bus retains the oldest written word until the clock cycle *after* the `S_Read` signal is asserted by the user. At that time, the value on the `S_Data` bus changes to reflect the next available word on the FSL's FIFO, or is replaced with 0 when no more words are available. However, the value of 0 is not supported to be an indicator of a lack of valid data, the honor of which goes to the deassertion of the `S_Exists` signal. In the case where a read is performed on an empty FSL, the carry bit is asserted in the MSR to indicate that any data that is taken from the output data bus is invalid [38]. A full transfer from writing a word to the Master interface to reading that same word on the Slave interface takes a total of two system clock cycles (assuming system-synchronous operation and that the FSL was previously empty). If data exists within the FSL, both a write and a read can occur on the same system-synchronous clock cycle.

These functions, as mentioned in Section 3.2.2, are provided in assembly-based C macros by Xilinx in two forms: blocking and non-blocking. Both families of blocking and non-blocking

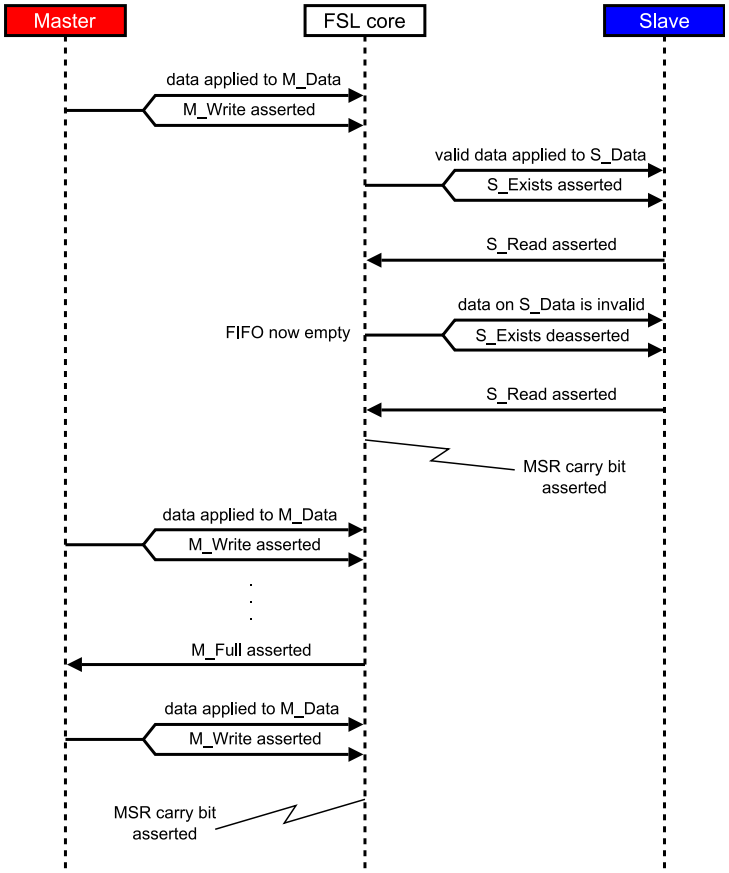


Figure 4.3: FSL Protocol Diagram

functions have versions of the functions that assert the control bit propagated with a corresponding data word through the FSL. A blocking write forces the MicroBlaze to stall until the FSL is not full, and a blocking read forces the MicroBlaze to stall until the FSL is not empty. The non-blocking functions do not force the MicroBlaze to stall and data is either destroyed on a write to a full FSL or indicated to be not valid if read from an empty FSL. Non-blocking reads are useful when interrupts cannot be used to verify valid data. Upon a non-blocking read of an empty FSL, data is taken from the `S_Data` bus. The MicroBlaze does not stall in such a situation. However, the carry bit in the MSR is asserted to indicate that the data is invalid.

To reiterate, the FSLs were configured in both synchronous and asynchronous modes based on their locations relative to the processors and Aurora. While asynchronous mode indicates two distinct clock domains, this implementation utilizes more resources than their synchronous counterparts. The FIFO depth of the FSLs is configurable and has a default value of 16 words. This default value was retained, but may be changed in the future based on the communication needs of a given application. The MicroBlaze can write to the FSLs on successive clock cycles since each write only takes one clock cycle. Blocking writes would ensure that data is not lost if the FIFO depth is too small while perhaps sacrificing overall synchronization of the entire system as the MicroBlaze would stall if the FSL was not emptied. This demonstrates the need of the FSL's FIFO buffer. Blocking writes and reads ensure the protection of the data within the FSL.

4.2.2 Aurora

Xilinx offers the Aurora core free-of-charge from their website upon registering personal information to their database. The Aurora core HDL source files are generated by the Xilinx software package Coregen^{TM} that provides an interface allowing full customization of the core: the number of lanes to be bonded into a channel (if desired), streaming or framing modes using either 16- or 32-bit interfaces (known to Aurora as a 2-byte or 4-byte interface, respectively), the designation of any of four reference clock signals, and assignments for each MGT per lane. Maximum transmission speed requires a non-routable buffered differential reference clock called `BREF_CLK` (and a second copy called `BREF_CLK2` that originates from a different set of pins and operates at the same rate as `BREF_CLK`). Slower transmission speeds can be safely handled by the `REF_CLK` and `REF_CLK2` signals. These two clock signals do not have static source pins and are allowed to be routed through the FPGA fabric to accommodate particular FPGA configurations. In contrast, the source pins for the `BREF_` clock signals cannot be changed. They offer less jitter and better error rate than the routable `REF_` clocks. Jitter is the amount of deviation in time or phase from the original clock signal as it travels through the physical wire that generally results in the misinterpretation of data. The Aurora

core and any related user application operate on these clock signals. The speed of the BREF_ reference clocks is set by a pair of jumpers on the ML310 board to 156.25 or 125 MHz. A 20x multiplier is applied to both BREF_ reference clocks for transmission. Simple tests confirmed that the 125 MHz reference clock is more stable; bit errors occurred on at least one lane on every test while using the 156.25 MHz reference clock. Therefore, the implementation of the Aurora core uses the 125 MHz reference clock that provides a maximum transmission rate of 2.5 Gbps.

With the core completely configured, it is ready for implementation. Both the 16- and 32-bit interfaces operate in the same manner, with the only difference in the width of the data ports. When either interface is not in use (i.e., data is not transmitted or received), the core generates and sends "idle characters" that provide clock synchronization as well as keep the voltage differential on the physical wires of the cable at an average of zero volts, so as to avoid DC bias and resultant bit errors. DC bias is a situation where a set of data contains long strings of 1s or 0s, shifting the overall voltage center of the signal. Good DC bias keeps the average voltage close to zero volts. Data encoding techniques that provide a balance of 1s and 0s prevent DC bias. The Aurora core always uses an 8-bit-to-10-bit (8b/10b) encoding scheme when transmitting data, where every possible combination of 8-bits of data correspond to a 10-bit encoded word that is decoded at the RX end of the communication channel.

As stated earlier, Aurora can be used in streaming mode or framing mode. Streaming mode is the simplest of the two modes of operation. Once the channel and all corresponding lanes are connected and ready for operation (also stated as being "up"), the core assumes the use of an infinite frame that needs not be indicated by any status lines. The user application simply applies data to the TX_D bus, asserts the TX_SRC_RDY_N line, and if the TX_DST_RDY_N line is also asserted by the core, data is transmitted. It should be noted that TX_DST_RDY_N is only deasserted during two scenarios: if any of the lanes or channel are not up, or when the clocks on each side of the link are being synchronized. Clock synchro-

nization occurs regularly every 3000 clock cycles [39] to ensure that the receiving end of the link is operating on the proper embedded clock within the data stream. This occurs by stopping all regular data (deasserting `TX_DST_RDY_N`) and transmitting a clock correction sequence of bits. The RX side of the streaming interface presents the received data on the `RX_D` bus and asserts `RX_SRC_RDY_N` on the same clock cycle to indicate that there is data available. If data is not consumed by the user application on the same clock cycle, the data is destroyed on the next clock cycle either by presenting another data word or an idle character.

While the streaming interface is the simplest of the two user interfaces, it does not provide the receiving side with any information regarding the data that is being transmitted (only that data is in fact being transmitted). As can be seen in Figure 4.2b, outgoing data is taken from multiple FSLs, transmitted through a single-lane channel, and incoming data is separated to corresponding FSLs at the other end of the link. A method for providing the receiving side with information regarding the meaning of the data was required, and framing mode provides this ability.

Framing mode adds a few more signals to the basic streaming interface: `TX_SOF_N`, `TX_EOF_N` and the `TX_REM` bus. Data is only valid within a frame, encapsulated by the assertion of the SOF (Start Of Frame) and EOF (End Of Frame) indicators. Figure 4.4 shows an example of a frame. Each clock cycle during which valid data within a frame is received is referred to as a *beat*. Partial words are supported in the last data beat of the frame. The number of valid bytes in that last data beat is indicated by the value on the REM bus (indicating the "REMaining" number of bytes in the final beat of the frame). The width of `TX_REM` is a function of the size of the channel given by $\lceil \log_2(n) \rceil - 1$, where n is the number of bytes per word. For example, Figure 4.4 contains a four beat frame. It uses a 32-bit interface, yielding four bytes per beat. The last beat (represented by *D3*) contains four valid bytes, indicated by the 2-bit binary-encoded value of 3 (11_{binary}).

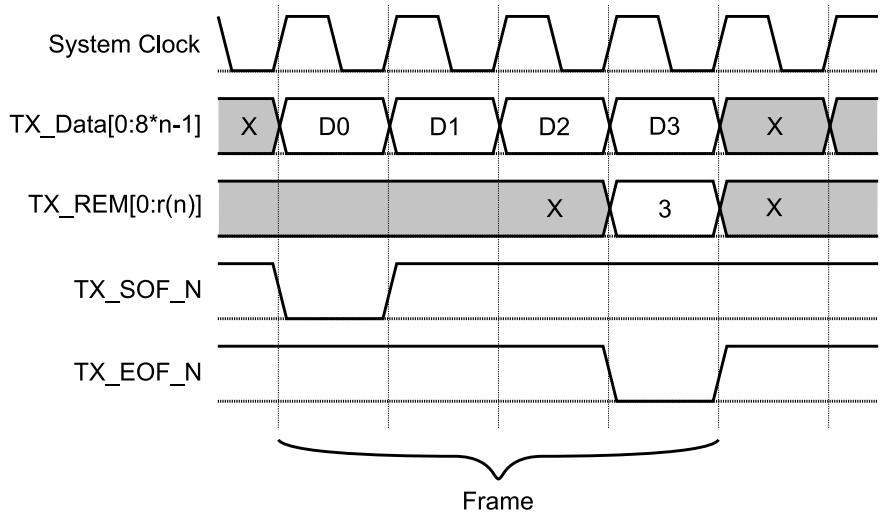


Figure 4.4: Example of an Aurora Framing mode data frame

The RX side of the link includes similar corresponding signals that provide the same functions. The difference between framing mode and streaming mode is how the core defines valid data. Framing mode only allows valid data to be transmitted within the context of the framing bookend signals (`SOF` & `EOF`); in streaming mode, all transmitted data is valid within the context of the hypothetical infinite frame that needs no literal frame encapsulation.

Framing mode offers flow control in two options: Native Flow Control (NFC) and User Flow Control (UFC). Both modes have similar ports to the interfaces for normal data and signals that provide similar functions (such as the flow control versions of the `SOF` and `EOF` signals, their own data buses, etc.). Both flow control modes can be used concurrently, but the flow control modes perform different tasks. NFC uses core-specific, unalterable message values to force the core to add a specific number of idle characters into the current frame of data. This is the only function NFC provides. UFC provides a means to send a user-defined variable-length message either within or outside a frame of regular data. As Figure 4.5 shows, UFC adds its own framing interface to the RX side of the link apart from the interface for normal data with separate `SOF`, `EOF`, `SRC_RDY_N` signals and `REM` bus.

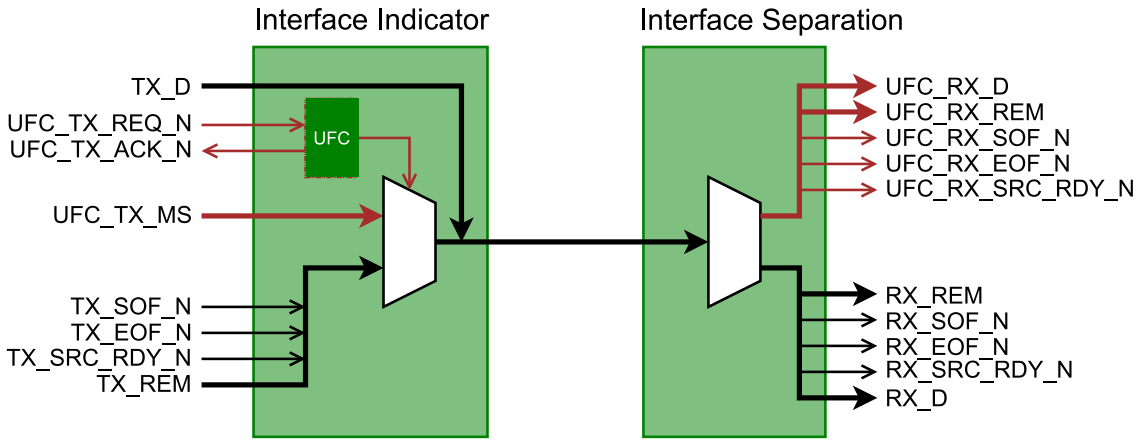


Figure 4.5: Framing and UFC interfaces

Figure 4.6 shows the protocol diagram of the UFC interface. The user requests to send a UFC message by asserting `UFC_TX_REQ_N` until the core acknowledges the request by asserting `UFC_TX_ACK_N`. Data present on the `TX_Data` bus is transmitted as the UFC message. This occurs on the clock cycle after `ACK_N` is deasserted. Occasionally, clock synchronization must occur prior to a UFC message transmission; `ACK_N` is held until clock correction is finished. The message *size* is encoded in the 3-bit-wide `UFC_TX_MS` bus and transmitted with the UFC message. The message size in bytes is determined by $(size + 1) \times 2$, where *size* can take a value from zero to seven. The message size cannot exceed 16 bytes. The RX side of the link presents the UFC message on a separate framing interface and encapsulating the UFC message with the `UFC_RX_SOF_N` and `UFC_RX_EOF_N` signals based on the value from `UFC_TX_MS`. Figure 4.5 is designed to indicate that the `TX_D` bus is always transmitted, regardless of whether a UFC message or regular data is transmitted. Its interpretation is based on whether the core has acknowledged a UFC message transfer. The clock cycle upon which `REQ_N` is asserted begins at least a 6-clock-cycle delay of the reference clock before returning to normal data transmission, during which the `TX_DST_RDY_N` signal is deasserted in order to transmit the UFC message instead of regular data.

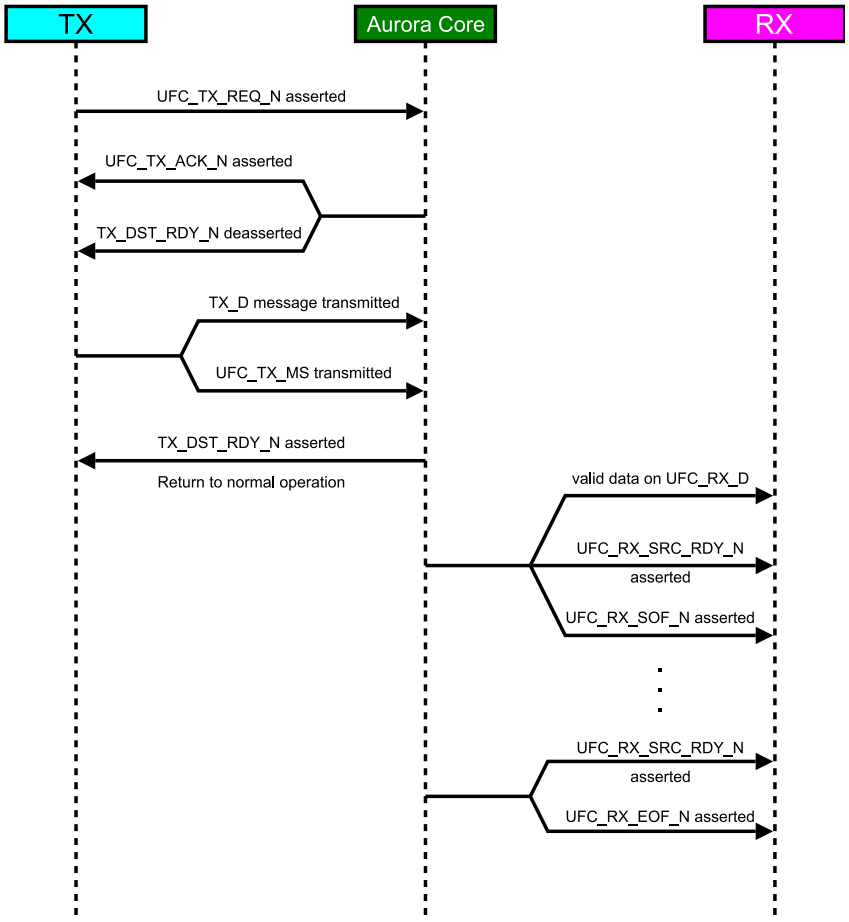
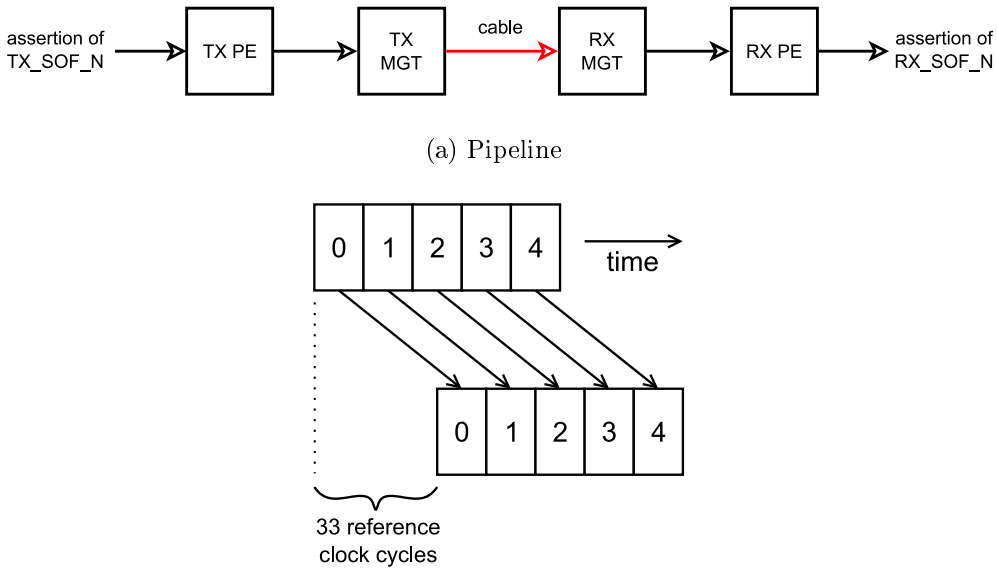


Figure 4.6: Aurora Core UFC Protocol Diagram

Transmitting a UFC message and normal data are mutually exclusive functions, where the core designates only a UFC message or normal data to be transmitted on any given clock cycle (barring the 6-cycle delay), and are then separated into their particular interfaces at the RX side. This mutual exclusion is exploited by an HDL top module called the Aurora-FSL Switch, described in Section 4.2.3.

Once a frame is started in either streaming or framing mode, it takes a total of 33 reference clock cycles to propagate from the assertion of `TX_SRC_RDY_N` within the TX MGT



(a) Pipeline

(b) Frame Latency

Figure 4.7: Latency Involved in Aurora-based Communication

to `RX_SRC_RDY_N` within the RX MGT for the same data word in the pipelined path within the Aurora core and MGT. The pipeline consists of both TX and RX versions of the Aurora PE and latency within the MGT itself (see Figure 4.7a). This results in a frame-wide latency, where once a frame begins, the latency due to the pipeline applies once to all words within the frame together, but does not accumulate. It is similar to a parent driving each of five children to soccer practice at different times. The total time driving (from home to the soccer field, and then back home to get the next child) accumulates with each child. Instead, the latency here is closer to taking all of the children to practice at once in a van. It takes time to get from home to the soccer field, but the latency applies to all children in a non-cumulative fashion. Figure 4.7b perhaps more clearly describes this overall latency.

4.2.3 Aurora-FSL Switch

In order to provide mesh scalability, off-chip communications require some method of moving data from FSLs and transmitting them by way of the Aurora protocol and InfiniBand

cables. It also requires a method for taking received data from Aurora and presenting them to the FSLs at the RX side. A data switch was developed in Verilog and embedded within the Aurora core top module that accepts data from multiple FSLs and presents them to a single Aurora channel for transmission. The Aurora core uses the 125 MHz `BREF_CLK` as its reference clock for both meshes. An application embedded within the top module of the component is synchronized with the core through the use of the reference clocks; a design using the 16-bit interface is given access to an unaltered reference clock to drive its logic, but a 32-bit interface design uses half the reference clock. The Aurora core retains the channel bandwidth by transmitting twice as much data per clock cycle as a 16-bit interface application.

Each of the external FSL ports on the edge of the mesh connect to one channel of the Aurora core IP in order to minimize the total number of MGTs that the mesh would require. The eight MGTs available on the FPGA would not satisfy the twelve FSL ports required by the 3×3 mesh if each external bidirectional FSL port was connected to a single MGT. For comparison, the 2×2 grid could fully exhaust the number of available MGTs if each FSL port would have access to its own MGT. This would obviate the need for the Aurora-FSL Switch; the FSL could pass data directly to the Aurora core. However, such a scheme limits scalability and increases physical hardware requirements. Therefore, as Figure 4.2 shows, the system was designed so that the external FSL ports on each edge of either size mesh would connect to a single Aurora channel. This method is scalable to include a greater number of processors per edge, requiring only the addition of more FSL ports in the source HDL code of the Aurora core top module. The transmit side of the channel consumes data words from a given FSL until either the FSL is empty or after 32 words are consumed. This configurable counter value is arbitrarily chosen as a balance between current FSL depth and possible frequency of FSL use by all processors. The FSLs are checked in a *round-robin* manner that depends upon the current FSL being consumed at the time of the check. If no FSL has data, and therefore no need for transmission, only idle characters are transmitted and ignored at

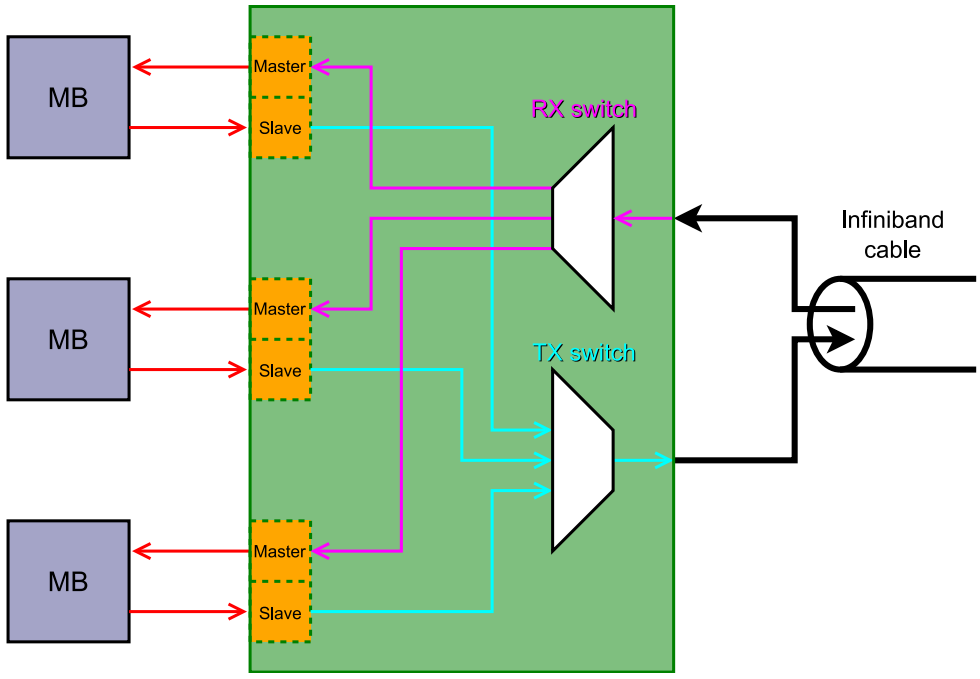
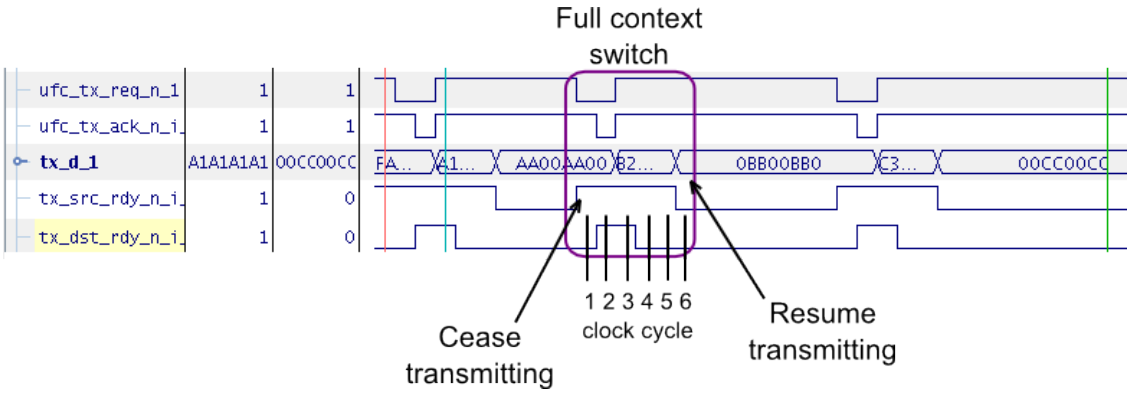


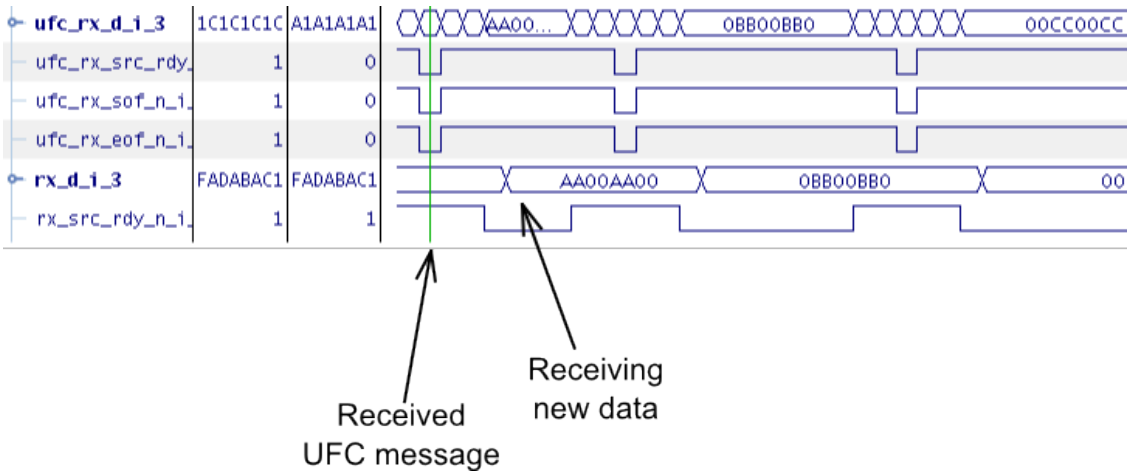
Figure 4.8: Connectivity of the Aurora-FSL Switch

the RX side (as Aurora's *modus operandi*). A fully commented transcription of the switch algorithm is provided in Appendix A. With this switching scheme, no FSL monopolizes the channel more than any other FSL, given a situation where more than one FSL has data to be transmitted.

When the Aurora-FSL Switch changes context from one FSL to another, a UFC message is first transmitted that indicates the proper destination FSL to which an indeterminate number of following data words should be routed. Each UFC message is a single 32-bit unique identifier for each FSL. The UFC message indicates the source row (for East/West FSLs) or column (for North/South FSLs) from which all following data comes. The data will be routed to the same row or column at the RX side of the Aurora channels "until further notice," retaining mesh connectivity (see Figure 4.10). This context switch incurs 6 reference clock cycles of overhead. There is a single reference clock cycle of overhead to write



(a) TX side



(b) RX side

Figure 4.9: Examples of Context Switch

the received word to an FSL. Therefore, the minimum number of reference clock cycles to transmit a single 32-bit word across an Aurora channel is

$$33 \text{ TX cycles} + 6 \text{ context switch cycles} + 1 \text{ write cycle} = 40 \text{ total ref. cycles}$$

Due to the overhead caused by the 6-clock-cycle delay between transmission of data from one FSL to the transmission of data from another FSL, overall channel efficiency is increased by transmitting data from a particular FSL on each clock cycle until the FSL is either empty or the counter expires. The most inefficient use of the channel is to only transmit a single data word from each FSL, changing FSL context after each transmission.

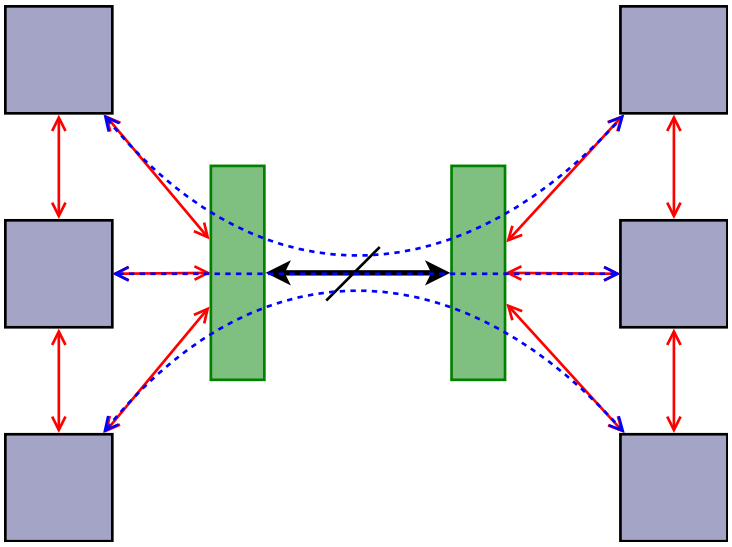


Figure 4.10: Source Implies Destination

To decrease the complexity of the Aurora-FSL Switch, the normal data-encapsulation signals `TX_SOF_N` and `TX_EOF_N` are minimally used. Upon initialization of the channels, the `SOF` signal is asserted only once. `EOF` is never asserted, thereby imitating an infinite frame. The Aurora core does not concern itself with the interpretation of frames except to support frames as an operational option. UFC messages can interrupt a regular data frame without hindering the overall frame integrity. Therefore, data is transmitted only by the assertion of `TX_SRC_RDY_N` in a manner similar to Streaming mode.

Aurora leaves overall system flow control to higher-level protocols. While MicroBlazes have the ability to stall under the two conditions of writing to a full FSL or reading from an empty FSL (using the blocking versions of the functions), the Aurora core cannot stall. The Switch allows data to be destroyed when writing to a full FSL at the RX side of the channel. Therefore, to prevent data destruction when the transmitting MicroBlaze is producing data too fast or the receiving MicroBlaze is consuming to slowly, handshaking must occur

in software between the two processors. This is left to the discretion of the software designer.

Channel-bonding was another method that was explored as an alternative to the Aurora-FSL Switch. With channel-bonding for this particular hardware design, a channel would be comprised of two lanes in parallel that contains data from an FSL in one lane and its unique identifier in the other lane. Channel efficiency would increase since this method would require no context-switching delays. The switch would route the current data word based on the current identifier word on every clock cycle. Thus, the entire bandwidth of the channel could potentially be fully utilized given sufficient utilization of the FSLs by the MicroBlazes. However, this method also unnecessarily uses an entire 32-bit lane and extra MGT to transmit the same 32-bit identifier for as many data words as are read from a particular FSL. It was found that the switch context delay was acceptable. Fewer MGTs in use is better for future designs. The channel-bonded method was abandoned.

The maximum throughput of a MicroBlaze processor can possibly be caused by an infinite loop containing non-blocking writes that force data onto the Aurora core by writing one 32-bit word (without the Control bit) to an FSL on each successive clock cycle of the 100 MHz system clock.

$$\frac{32 \text{ bits}}{8 \text{ bits per byte}} \times 100 \times 10^6 \text{ Hz} = 400 \text{ MB/s}$$

The Aurora core can transmit one 32-bit word on each successive clock cycle of any reference clock. The reference clock speeds seen by a user application, such as the Switch, for a 4-byte design are

$$\frac{125}{2} = 62.5 \text{ MHz and } \frac{156.25}{2} = 78.125 \text{ MHz}$$

Clock compensation produces three reference clock cycles of overhead every 3,000 reference clock cycles, yielding 2,997 useful reference clock cycles out of 3,000. This is accounted for in the following equations. The Aurora core limits the maximum bandwidth of a single FSL to

$$4 \text{ bytes per word} \times 62.5 \times 10^6 \text{ Hz} \times \frac{2,997}{3,000} = 249.75 \text{ MB/s}$$

$$\frac{249.75 \text{ MB/s}}{400 \text{ MB/s}} = 62.43\%$$

for the 125 MHz reference clock, and

$$4 \text{ bytes per word} \times 78.125 \times 10^6 \text{ Hz} \times \frac{2,997}{3,000} = 312.2 \text{ MB/s}$$

$$\frac{312.2 \text{ MB/s}}{400 \text{ MB/s}} = 78.05\%$$

for the 156.25 MHz reference clock. A single Aurora link cannot quite match the long-term theoretical throughput of an FSL. This further decreases with the inclusion of two FSL ports in the 2×2 mesh and three FSL ports in the 3×3 mesh as well as the number of 6-clock-cycle delays per context switch per second. Assuming successive-clock-cycle usage of the FSLs, the minimum bandwidth for a single FSL occurs when the Aurora core switches context between each FSL after transmitting only one 4-byte word.

$$T_M \times \frac{n}{n + 6 \text{ delay cycles}} \times \frac{1}{l} = T_f,$$

where T_M is the maximum throughput of an FSL through an Aurora channel (that is, 249.75 or 312.2 MB/s), n is the number of 4-byte words transmitted between context switches, l is the number of FSLs connected to a single Aurora channel, and T_f is the final resulting throughput of a single FSL. Minimum throughput is found when $n = 1$ for either reference clock and any number of FSLs per Aurora channel. Table 4.4 summarizes the maximum and minimum imposed bandwidth on a single FSL by the Aurora core assuming data is always present to be transmitted from all FSLs. The link efficiency increases as n (the number of words transmitted between context switches) increases. Recall from above that it takes 40 reference clock cycles to transmit a single 4-byte word, including the context switch and write time. This translates to

$$\frac{62.5 \text{ MHz}}{100 \text{ MHz}} \times 40 = 64 \text{ system clock cycles}$$

This is useful for understanding the results presented in Chapter 5.

Table 4.4: Theoretical Bandwidth imposed on a single FSL by the Aurora core (in MB/s)

FSLs per Channel	Ref. Clock (MHz)	Maximum	Minimum
2	125	249.75	17.83
	156.25	312.2	22.3
3	125	249.75	11.89
	156.25	312.2	14.87

4.2.4 Debugging Options

Since many, if not all, multi-core architectures require a synergy of hardware and software design to fully utilize the available parallelism, one method for debugging this particular architecture was developed using the MicroBlaze Debugging Module (MDM). Each MicroBlaze contains a debug port that directly connects to the MDM. Communication between MDM and the software designer takes place within the command-line-based Xilinx MicroBlaze Debugger (XMD) that can run underneath the GNU Debugger (GDB) graphical interface, though this is not required. XMD uses the FPGA's JTAG communication port, which is the FPGA's primarily method for accepting bitstreams from a host computer that configure the FPGA fabric. XMD also has the ability to run cycle-accurate simulations, an ability that rivals the RAMP project for functionality. This method for debugging a highly parallel program is simple, but effective. The MDM can fully debug the full mesh on a single FPGA for meshes with eight or less processors. This excludes the 3×3 mesh. However, debugging can take place on a smaller subset of processors within the larger mesh.

The alternative to a specialized debugger would be to force the processors to communicate their debug messages through the communication medium of the mesh interconnect, which would perturb the entire system in order to flush out the debug messages and resume normal computation and communication. Software-based print messages arriving at the OPB-based

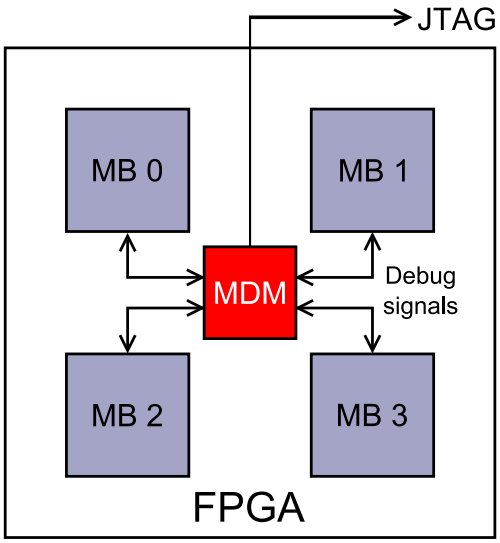


Figure 4.11: MDM Block Diagram

RS-232 UART IP from several processors in a short amount of time results in jumbled text on a host computer's terminal screen. There is no message arbitration between the OPB and the RS-232 UART component. Therefore, an FSL-based UART communication IP was developed by a colleague of the author and put into service on this project. It was modified to allow nine FSLs to access the RS-232 UART with full arbitration in order to print uninterrupted messages to a host computer. Arbitration takes place on an FSL-by-FSL basis, in a similar manner to the Aurora-FSL Switch. When any FSL contains data, it is consumed until it is empty, assuming that a print statement ends with a lack of characters within the FSL. In the event that more than one FSL contains data, only one FSL is consumed until it is empty before allowing another FSL access to the UART. Therefore, each message sent by each processor is separated and fully written to the host computer's terminal, without combining messages into a jumbled mess.

To aid this generic print functionality, the basic Processor Version Register (PVR0) within each MicroBlaze provides a means to specify a unique value to its eight most significant bits, referred to as the USR1 field of the PVR0 register. The other bits represent processor-

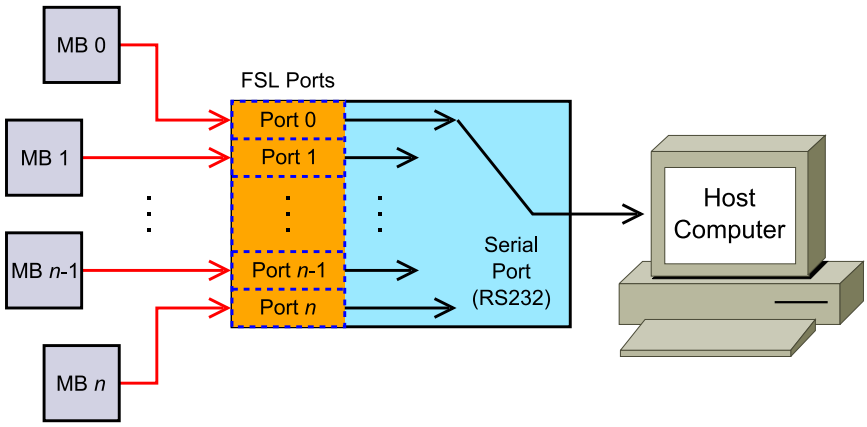


Figure 4.12: FSL2Serial IP Block Diagram

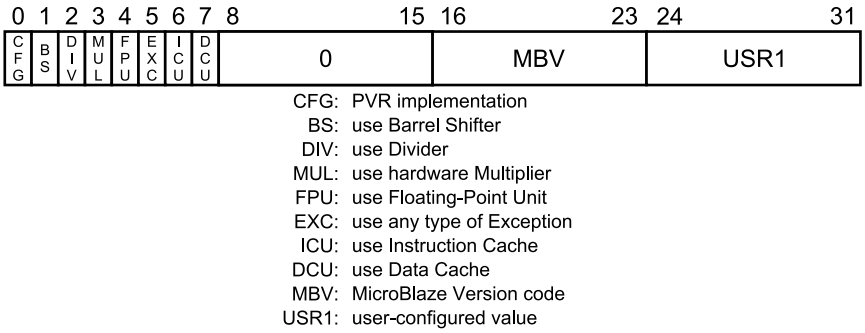


Figure 4.13: Process Version Register (PVR0)

specific configuration values. The full PVR register set includes 10 more registers with values indicating all of the possible configuration values for each MicroBlaze, including a second user-defined field of 32 bits. All of the fields of the PVR0 register can be applied to the MicroBlazes in XPS as configuration parameters. The PVR is accessible by software macros provided by Xilinx. Built *a priori* into the FPGA configuration bitstream through XPS, the USR1 field contains a unique binary value for each processor for identification purposes. C software functions were developed to print a prefix message indicating the source processor to each print statement issued to the UART. The function can be modified to suit specific output format needs by the software programmer, and currently indicates the processor number in hexadecimal.

Chapter 5

Software Applications

No hardware architecture is complete without an application to showcase its operation and, hopefully, it's advantages. The software applications were written in C, each designed to run in parallel on a number of MicroBlaze processors in either size grid. The first two are essentially proof-of-concept applications for this architecture that provide a glimpse into efficient communication between processors, network performance, and network scalability. The last application has roots within the academic community as a parallel computational goal. These applications are compared against their own relative performance in various sizes of the grid to provide network metrics and performance values. Tabulation of the results for all of the tests run on this architecture through these applications are given in at the end of each section.

The programming model for this architecture can be best described as Multiple-Instruction, Multiple-Data (MIMD) processing. Each compute entity (in this basic case consisting of only a MicroBlaze, but may include other hardware components for additional hardware acceleration) contains its own instructions and data stored in its local memory. Each processor can accept data from and transmit data to other compute nodes if required. Shared memory is not used on this architecture since communication takes place between processors directly.

However, external memory might be added to the architecture to accommodate situations such as storing data on the FPGA boards. The processor connected to this memory might be considered to be the I/O processor or control processor. Data to be processed might be transferred from memory through this processor to the rest of the network.

Depending on the processor architecture, the primary difference in processing data on a multi-processor systems and a multi-core processor (or between a conventional computer cluster and a multi-core processor) is communication latency, whether through data access from memory or data sharing between processors. If two processor cores in a multi-core processor are connected by a bus, latency is higher than if the processors share a single, dedicated and direct connection. This will affect the overall time to process data. The communication medium might require software layers for data transmission, such as the TCP/IP stack over Ethernet. This situation also increases the time taken to process data through extracting the useful data from its communicated form. For these applications, the complete communication interface consists of only the blocking and non-blocking `get()` and `put()` functions for the FSLs where data is taken directly and transmitted unaltered. Aurora is invisible to the processors, showing how Aurora aids the scalability of the mesh architecture by being encompassed on both ends by FSLs. Since all of the processors are connected to other processors by direct connections, there is no extra software layer required for communication between processors. MPI is not used for any of these applications for one fundamental reason: using MPI across FSLs would require modification of the MPI source code to use the FSL functions described above. MPI would therefore incur software-layer overhead that this architecture aims to reduce.

All three applications were developed by hand using a simple text editor for writing the C source code. The code was cross-compiled for the MicroBlazes through XPS on a host computer with the proper arguments and flags according to the user's specifications. Debugging occurred through strategically-placed print commands within the software that were

issued to the host computer through the RS-232 UART component and through the debugging capability of the MDM. Each print statement is prepended by a string that identifies the source processor through the use of the PVR0 USR1 field and a modified `xil_printf()` command (originally supplied by Xilinx). ChipScope was used to view the communication traffic across the Aurora channels. The source code for each application can be found in the Appendix.

5.1 Application 1: Data Pumping

The first application is simply called a "data-pumping" application. The processors along the diagonal of either size mesh send a series of 32-bit words in all four directions. The remaining non-diagonal processors merely poll each FSL port for data and send the data along the same path as if the processor didn't exist. For example, consider a processor that finds that the West input FSL contains data. This same data word is sent to the East output FSL. This program is called a "placeholder." The placeholder program may be run on an otherwise unused processor to enable the processor to become part of the communications network, with a 2-clock cycle delay to receive a word and to write the word to the next FSL. Figure 5.1 shows how the communications network is fully used in a concurrent fashion. The software constantly polls the FSL ports and immediately passes data along, showing how fast the processors can move data through the network at their maximum speed when all of the FSLs are being used.

5.1.1 Application 1 Results

The Data-Pumping application was written to highlight the communication through the on-chip network and the Aurora channels. Transmission times of passing a single 32-bit word across a torus-connected mesh in one direction confirm the network latency expected for such a transmission. Table 5.1 shows these round-trip times using various mesh sizes

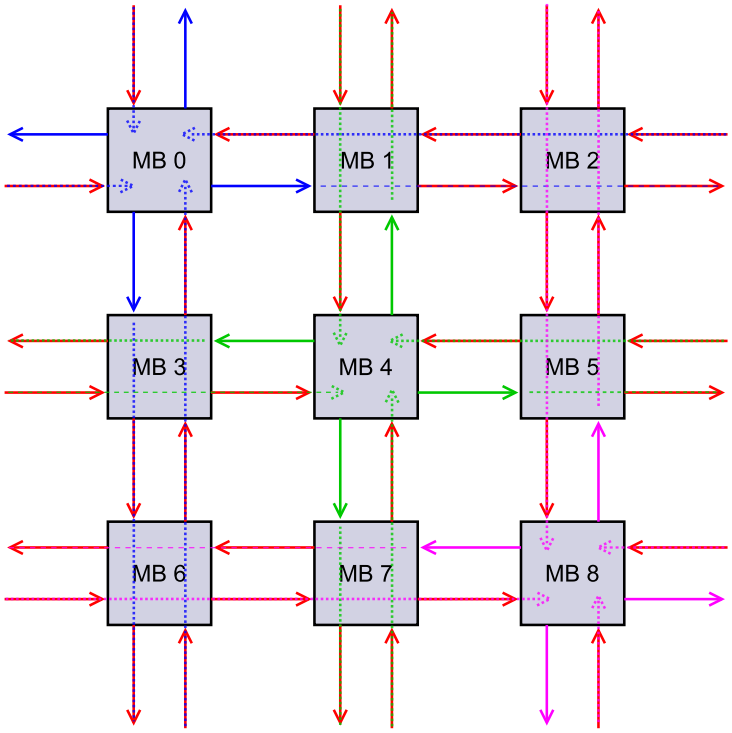
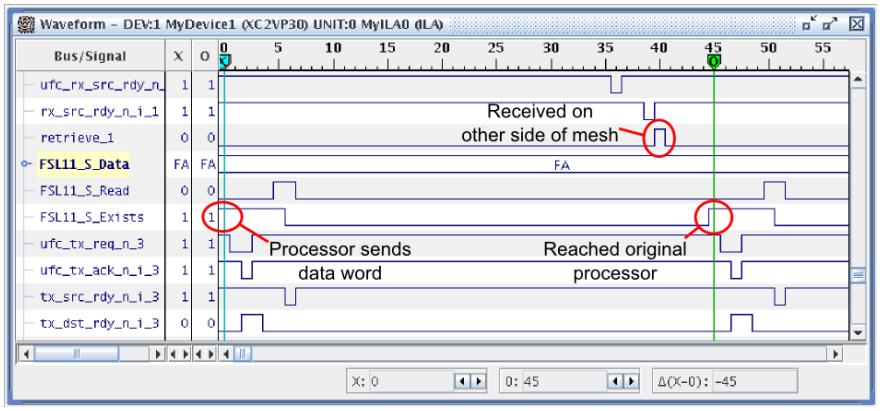


Figure 5.1: Data-pumping Flow

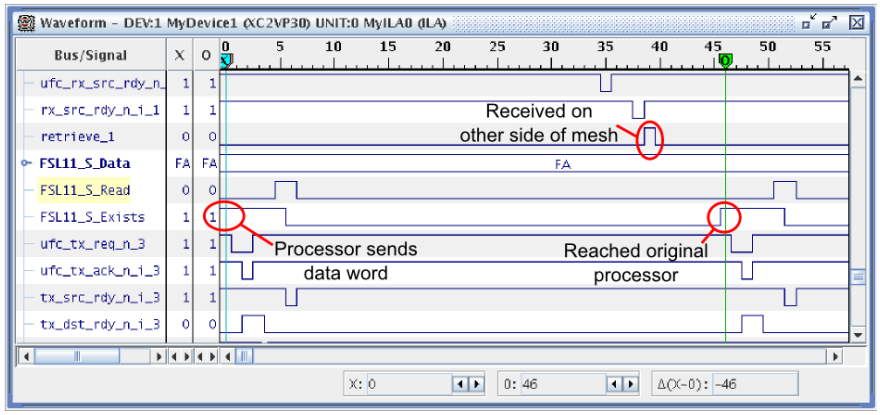
and expansions. In Figure 5.2, ChipScope shows a single transmission for the 2×2 and 3×3 torus meshes. Figure 5.2a indicates a 45 reference clock cycle (72 system clock cycle) latency between transmitting the first word from a processor directly to Aurora and that word returning to Aurora after being passed across the 2×2 mesh. Similarly, Figure 5.2b shows the latency of the same transmission across the 3×3 mesh to be 46 reference clock cycles (or 74 system clock cycles). ChipScope verifies that Aurora incurs a total latency of 40 reference clock cycles (or 64 system clock cycles) of latency and that software functions contribute to the rest of the observed latency as

$$i \times p + 40 \times 1.6 \times m + \text{extra cycles} = E$$

where i is the number of cumulative clock cycles for the program run on the processors (assuming each processor is running the same code), p is the number of processors in one dimension of the mesh, 1.6 is the ratio of system clock cycles to reference clock cycles, m is



(a) 2×2



(b) 3×3

Figure 5.2: ChipScope captures of single data word transmissions

the number of Aurora channels along the same dimension as how p is derived, and E is the expected latency. In this case, $i = 2$ for one clock cycle each of the blocking `put()` and `get()` functions that comprise the specialized placeholder source code for this test. The expected values in Table 5.1 reflects this value for i while ignoring the *extra cycles* margin of error. The generic placeholder source code can be found in Appendix B.1 that uses a verification check on non-blocking versions of these two functions. Reading and storing the OPB timer value and other possible issues such as clock-correction along the Aurora channel contribute to the extra cycles, collectively considered to be the margin of error.

Table 5.1: Data-Pumping Transmission Times (in seconds)

	Mesh Size			
Iterations	3×3	Expected	3×6	Expected
1	0.00000085	0.00000074	0.00000161	0.00000148
10	0.00000750	0.0000074.	0.00001501	0.0000148.
100	0.00007374	0.000074..	0.00014894	0.000148..
1,000	0.00073632	0.00074...	0.00148855	0.00148...
10,000	0.00753503	0.0074....	0.01490844	0.0148....
	2×2	Expected	2×4	Expected
1	0.00000084	0.00000072	0.00000145	0.00000144
10	0.00000734	0.0000072.	0.00001453	0.0000144.
100	0.00007212	0.000072..	0.00014472	0.000144..
1,000	0.00072033	0.00072...	0.00144417	0.00144...
10,000	0.00720157	0.0072....	0.01452198	0.0144....

5.2 Application 2: Packet Forwarding

There are no hardware routers in this application and the hardware does not read or interpret the data values. Each communication link is point-to-point with the source implying its destination. However, software can define how data is represented and interpreted in order to forward data to a specific processor. To show this, the second application was developed that uses a modified version of the FSL polling loop. Instead of transmitting data directly to another FSL, appropriate actions are taken based on the content of the received data. Arbitrarily, the top left processor of the entire mesh is referred to as [0,0] in [X,Y] coordinates, with increasing X values to the East and increasing Y values to the South. Processor [0,0] creates a packet with a 32-bit header, indicating itself as the source in the first 8-bit field, an arbitrary destination in the second 8-bit field, and the number of data words within the packet (excluding the header) in the last 16 bits. Each *source* and *destination* field, as indi-

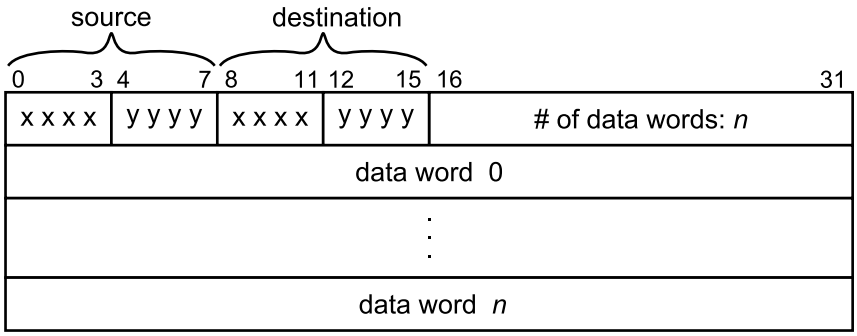


Figure 5.3: Packet, including header and data words

cated in Figure 5.3, is separated into its X and Y coordinates, four bits for each part. Macros were developed to separate each X and Y coordinate from the full source and destination fields to facilitate header parsing. Each processor was assigned its own unique identifier and stored in the PVR0 register's USR1 field, described in Chapter 4.2.4. The identifier in the USR1 field is designed to hold the [X,Y] value of each processor by reserving four bits for the X and four bits for the Y coordinate. The processor identification value is used by the software to determine if a received packet is destined for the particular processor performing the check and identifying the processor in print statements to the host computer attached to the ML310's RS-232 serial port. Each processor's software contains the same polling loop for each FSL that tests each port for valid data. If valid data is found, the polling loop parses the header word and deals with the packet accordingly: if the packet is intended for the checking processor, the processor creates a new header with the former source as the new destination and generates an arbitrary number of words to complete the packet; otherwise, the processor forwards the packet based on both the destination coordinates compared to the checking processors's identification value and the direction from which the packet was received.

For example, the following is a step-by-step example process from the viewpoint of each 3×3 processor in the data path. Recall that the coordinate system is denoted by [X,Y] with

increasing X values to the right and increasing Y downward.

1. Processor [0,0] generates a header for processor [2,2] in the bottom right corner of the mesh.
2. Packet is sent arbitrarily to the East.
3. Packet received from the West on processor [1,0].
4. Packet is not intended for processor [1,0] and the destination processor does not reside in this column; Packet is sent to East to continue from the previous direction.
5. Packet received from West on processor [2,0].
6. Packet is not intended for processor [2,0], but the destination processor is within this column; destination processor is in a greater row than processor [2,0]. Packet sent to the South.
7. Packet received from the North on processor [2,1].
8. Packet is not intended for processor [2,1] and destination processor does not reside in this row. Packet sent to the South to continue from previous direction.
9. Packet received from North on processor [2,2]. Packet intended for this processor!
10. New header is created. Former source becomes the new destination. New arbitrary data words are created. New packet sent back to the North.
11. Follows same path in reverse back to processor [0,0].
12. Process repeats *ad infinitum*.

This routing scheme was used under the condition that only a single message was active throughout the mesh at one time. All of the communication links were free to be used. However, this scheme is non-deterministic and subject to deadlock if more than a single

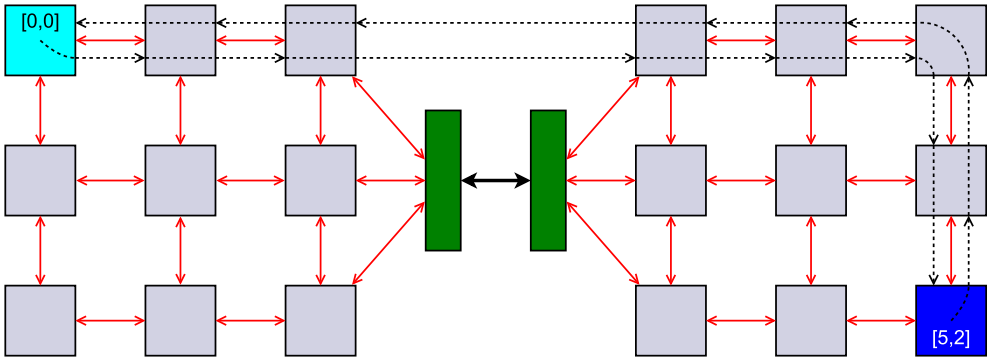


Figure 5.4: Packet Path for simple timing confirmation test

message was being transmitted through the network. In an XY-routing scheme, all messages are sent over the X-direction of the mesh and then over the Y-direction (to the EAST and then to the SOUTH, respectively, using the terminology of this architecture) [42]. This type of deterministic routing scheme is deadlock-free.

5.2.1 Application 2 Results

To expand beyond a single FPGA, each processor is given a unique identifier based on the total number of processors in the entire multi-FPGA mesh. The packets are forwarded as appropriate to the identification number of each processor. This particular application was written as an operational example of scalability and how software is solely responsible for data interpretation. However, Table 5.2 shows the measured round-trip times in clock cycles for a 3-word packet transmitted between processor [0,0] and each processor along the path shown in Figure 5.4, up to processor [5,2]. The processor identification number corresponding to the intended destination processor aided in confirming the proper route along the path.

Table 5.2: Round-trip Packet-Forwarding times (in clock cycles)

Processor	Number of Clock Cycles
1,0	132
2,0	234
3,0	438
4,0	540
5,0	642
5,1	744
5,2	778

5.3 Application 3: Particle Image Velocimetry

Particle Image Velocimetry (PIV) is a highly iterative image processing application that can benefit from parallelization. High-resolution images are taken of particles suspended in a moving fluid, much like blood cells flowing through arteries. Virginia Tech's AEThER lab is involved in processing these images to gain knowledge about cardiovascular diseases and identify ways to combat them. The desired result of processing the images is obtaining relative particle velocities between the two images (that is, the relative amount of distance and direction that particles move from one image to the next) in both the X and Y directions. The mathematical correlation between the two images can provide values of the relative velocities [43]. The computational goal is to perform the calculations in real-time as the images are taken of the moving fluid, rather than keeping a repository of images to process at a later time (or to process slowly).

The images consist of pixel intensity values that range from 0 to 255 in decimal. Processing begins by obtaining small *zones* of size Nx and Ny from the original images. One *window* is comprised of two zones, one zone from each image in the same relative position. Figure

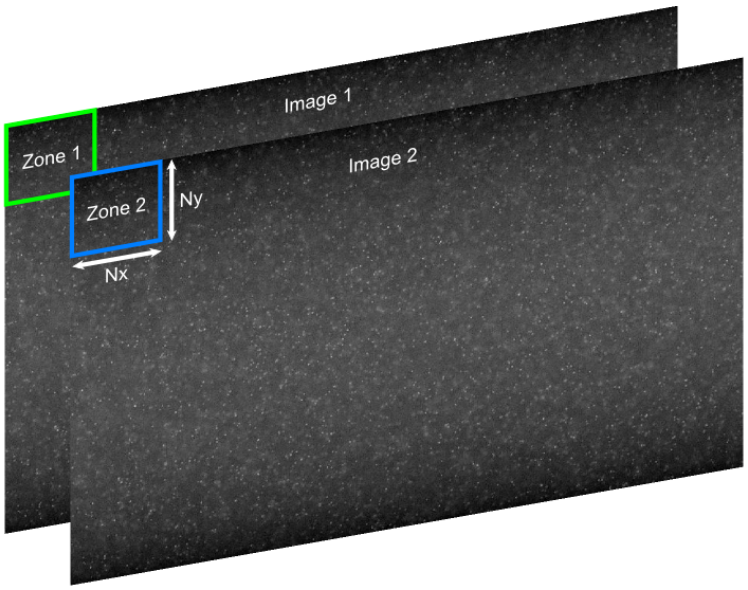


Figure 5.5: A single *window* of two *zones* taken from two high-resolution images of particles suspended in a fluid

5.5 shows how one window applies to the same area of pixels for both images. Each zone contains pixel intensity values from one image. The pixel intensity values of each zone are stored in a floating-point array, representing an all-real two-dimensional time-domain signal. Correlation of two real time-domain functions $g(t)$ and $h(t)$ is a complicated mathematical process that is simplified in the frequency domain (represented by f) as a complex conjugate multiplication:

$$Corr(g,h)(t) = G(f)H(f)^*$$

where * denotes the complex conjugate. Converting the real time-domain zone arrays into the frequency domain was performed by a custom Fast Fourier Transform (FFT) algorithm based on the algorithm found in the book *Numerical Recipes in C* [44]. The arrays become complex as a result of the FFT with both a real and imaginary part for all discrete values of the frequency spectrum. A complex conjugate array multiplication takes place that results in a single complex array. Returning these values to the time domain is done through the Inverse FFT (IFFT). However, at this point the time-domain values are very large in

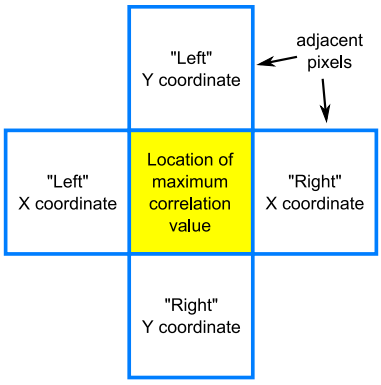


Figure 5.6: Correlation array test cases for the final velocity value

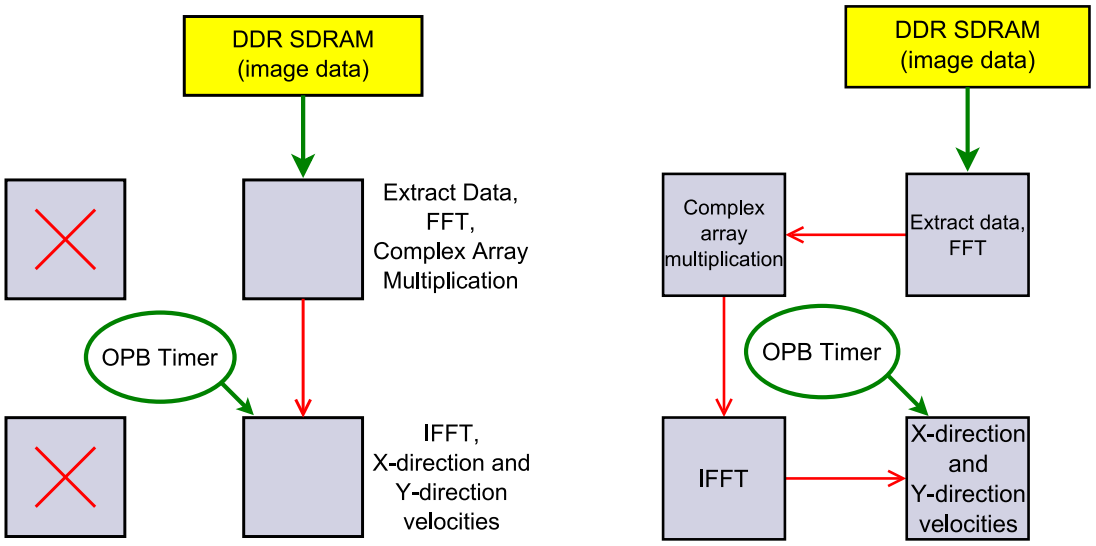
magnitude and need to be reduced in a manner other than simple relative normalization. Therefore, by testing the boundary conditions for the maximum value of each zone, the final velocity value for a zone is obtained through calculating the natural logarithm of the correlation values found in the four adjacent pixels surrounding the maximum correlation value (see Figure 5.6). The final velocity value is determined from these natural logarithm calculation results by

$$\frac{\ln(left) - \ln(right)}{2 \times \ln(center)}$$

in both the X and Y directions. In this manner, a positive X value indicates particle motion to the right and a positive Y value indicates downward particle motion. Each zone corresponds to a single signed relative velocity value for both the X and Y velocity matrices.

5.3.1 Application 3 Results

This application was transcribed from Matlab scripts into C source code in the manner of common sequential algorithms and processed on a single processor. The times taken to arrive at the final velocity solution for an increasing number of windows per run of the application served as a benchmark for performance. Parallelization of the algorithm was implemented to show some amount of speedup to validate the computational effectiveness of the processor



(a) Method 1 (b) Method 2

Figure 5.7: Correlation-based PIV parallelization by function

network. Due to the floating-point nature of the FFT values, only the 2×2 mesh was used for this application. However, the mesh was modified to allocate 64KB of local memory to each processor, since the size of the program exceeded the original design of 32KB of local BRAM. The 256MB DDR SDRAM module was also added to the mesh in order to access the two main images. An OPB timer and interrupt controller were added to obtain the calculation times. Zones of each image were extracted from memory and processed by a subset of the constituent functions of the full algorithm at each processor. The application was parallelized by function to allow data to stream between processors within the mesh. As seen in Figure 5.7, the tasks include obtaining the image data, performing the FFT, the complex array multiplication, the IFFT, and finally the calculation of the X-direction and Y-direction velocities per window. Table 5.3 shows the times taken to arrive at the final velocities for 2-processor and 4-processor versions of the algorithm. In the table, the entry for 88 windows refers to the number of windows in a single column of the main images. The final two values of 1,000 windows and the full algorithm of 12,848 windows using 32×32 pixels are derived times (shown in minutes and hours, respectively) that were calculated based on the

Table 5.3: PIV Calculation Times (in seconds) and Speedup

	Number of processors			Speedup	
Windows	1	2	4	2 proc	4 proc
1	0.6045	0.5835	0.6286	1.036	0.962
2	1.2088	0.9728	1.0429	1.243	1.159
4	2.4172	1.7511	1.8715	1.380	1.292
8	4.8339	3.3079	3.5289	1.461	1.370
20	12.0819	7.9785	8.5010	1.514	1.421
50	30.2019	19.6547	20.9303	1.536	1.443
88	53.1565	34.4447	36.6745	1.543	1.449
100	60.2129	39.1152	41.6464	1.539	1.446
1,000	10:04 min	7:09 min	7:38 min	-	-
12,848	2:09 hr	1:32 hr	1:38 hr	-	-
			Avg. speedup	1.407	1.317

average speedup shown in previous runs of the algorithm. The table also provides calculated extrapolations for further parallelization methods. Speedup S_p is calculated from the single processor time, T_1, and the p processor time, T_p, as

$$S_p = \frac{T_1}{T_p}$$

Tables 5.4 and 5.5 contain the percentage of computation time and communication time that resolve to a computation-to-communication ratio for each processor. The number of zones sent and received from each processor are included to provide context for the amount of communication required for each function or processor. In both cases, the first processor's computation time is dominated by accessing the external memory, the FFT function and the complex array multiplication. Therefore, its computation-to-communication ratio will be high. In the 2-processor version, the second processor takes less time to process its data. This ratio will be lower since it shares the same amount of communication with the only other

Table 5.4: Computation-to-Communication comparison (2 Processors)

	1	2	average
computation (%)	94.7	59.5	77.1
communication (%)	5.3	40.5	22.9
ratio	17.86	1.46	3.5
zones sent	2	0	-
zones received	0	2	-

Table 5.5: Computation-to-Communication comparison (4 Processors)

	1	2	3	4	average
computation (%)	89.4	21.1	74.2	51.6	59.1
communication (%)	10.5	78.9	25.8	48.4	40.9
ratio	8.5	0.27	2.88	1.07	1.44
zones sent	4	2	1	0	-
zones received	0	4	2	1	-

communicator in this version of the system. In the 4-processor system, the first processor sees a similar effect due to the memory access and FFT functions. The second processor has far less computational tasks to perform while communicating the most zones of any processor in both receiving and sending a total of six zones. This is the reason for such a low ratio for the second processor. The trend of decreasing ratios in the 4-processor version (barring the anomaly of the second processor) is mainly due to the algorithm's reduction of the number of arrays while maintaining a relatively constant level of computation in the first, third and fourth processors. The algorithm was separated by function and was not designed to balance the computation and communication loads.

Chapter 6

Conclusions

6.1 Summary and Contributions

This thesis presented a scalable FPGA-based multi-core platform for software development. The Data-Pumping application served to show the operation of the Aurora core (viewed through ChipScope for verification) and the tabulation of network on-chip and off-chip latencies. The expected latencies closely match those that were observed. Scalability of the mesh and software-based routing responsibilities were demonstrated by the Packet-Forwarding application where a single message was transmitted from one processor and routed through the network across boards to the destination processor using a non-deterministic routing scheme. Printed statements embedded within the Packet-Forwarding source code provided the insight of where the packet was headed and to which processor it arrived. The correlation-based Particle Image Velocimetry application showed a speed up of about 1.4 for the 2-processor system and about 1.3 for the 4-processor parallelization scheme. The fact that more processors did not provide greater gains in performance than the 2-processor implementation can likely be attributed to the computation-to-communication ratio for each function, where computationally heavy functions act as a network bottleneck. Better load balancing, parallelization in a different way or using a different application might provide higher speedup

results.

While having results from other projects against which to compare this mesh architecture might provide insight into this hardware design's supercomputing capabilities, this project is not meant to compete computationally with the projects described in Chapter 2. Multi-core hardware design is not set in stone, nor is efficient use of that hardware. This project is designed to facilitate both for the future of multi-core and parallel software.

This multi-core architecture provides scalability and high bi-sectional bandwidth due to the two-dimensional mesh of 32-bit processors, low-overhead and low-latency communication found in the FSLs and the Aurora high-speed serial communication protocol, hardware-based parallelism to fit the needs of a given application, the ability to be ported to other FPGA boards, and various methods for debugging software and viewing system communication. Rapid prototyping and lower costs compared to conventional computer clusters serve to make this design attractive for developing software on large processor networks.

6.2 Future Work

The XC2VP30 FPGA is an older and smaller part than what is currently available from Xilinx. By comparison, the newer Virtex-5 series of FPGAs include the XC5VLX110T (soon to be used on the Xilinx University Program development board) and the XC5VLX330T. Table 6.1 recaps the basic 3×3 mesh FPGA resource requirements and percentage utilization and includes the resource utilization of this mesh on both of these two newer FPGAs. Based on the resources of the smaller XC5VLX110T and assuming the same resource requirements for each component, a 4×4 mesh would be possible that uses 16 maximum-implementation MicroBlazes with 16KB of local BRAM, 80 FSLs, four total four-port Aurora channels and possibly up to the full 16 transceivers if each was dedicated to a single FSL port without the Aurora-FSL Switch. The XC5VLX330T can hold up to a 7×7 mesh that includes 49

Table 6.1: FPGA Resource Comparison for the basic 3×3 mesh

Configuration	LUTs	FFs	BRAM (blocks)	MGTs
XC2VP30	27,392	27,392	136	8
Basic 3×3 mesh	20,670	10,111	72	4
% FPGA Utilization	75.5	36.9	52.9	50
XC5VLX110T	69,120	69,120	296	16
Basic 3×3 mesh	20,670	10,111	72	4
% FPGA Utilization	30	14.6	24.3	25
XC5VLX330T	207,360	207,360	648	24
Basic 3×3 mesh	20,670	10,111	72	4
% FPGA Utilization	9.97	4.88	11.1	16.7

Table 6.2: Potential mesh sizes and implementation values for the XC5VLX110T and 330T

FPGA	Mesh size	MBs	BRAM (blocks)	FSLs	Aurora Ports	MGTs
XC5VLX110T	4×4	16	256	80	16	up to 16
XC5VLX330T	7×7	49	392	266	28	up to 24

maximum-implementation MicroBlazes with 16KB of local BRAM, 266 FSLs, a 28-FSL-port Aurora-FSL Switch divided into two channels per edge (each channel connecting either three or four FSL ports) that consumes 8 transceivers out of the possible 24. As is the case with the XC2VP30, BRAM seems to be the major limiting factor to the mesh size. Table 6.2 summarizes the resources of these mesh sizes.

A few enhancements to the mesh architecture are discussed here. For example, a global timing system for keeping accurate calculation times across boards does not exist. Currently, timing can only be done with the timer available as a component attached as a slave

to the OPB. This is not scalable to other boards. Aurora might even fill this role with the large number of high-speed transceivers available on the newer FPGAs. Transceiver synchronization between boards is not as robust as the documentation suggests. Often, a board would have to be configured first and would need to be reset while the second connecting board is being uploaded with its bitstream, assuming there is no method for powering all of the boards and loading the bitstream at the same time. Otherwise, the Aurora channels may or may not synchronize and begin transmitting. In this case, making channel status available to software could provide a simpler debugging and control method than passively viewing the channels through ChipScope. One avenue for providing this software-visible Aurora channel status is the OPB General Purpose I/O (GPIO) component. Status signals can be registered in hardware and mapped into an address space through the GPIO. Flow control from a software perspective is also fairly complicated, requiring handshaking at the two ends of an Aurora channel to ensure successful transmission. Aurora does not guarantee that data will arrive safely at the receiving end, leaving flow control to higher-level protocols. A more tightly-tuned debugging and software development environment would facilitate wider use of the mesh; however, parallel programming is not an easy task. Though effective in their scope of system visibility, the disparate debugging methods found here do not quite simplify the task a great degree. The inclusion of hardware-based routers, rather than forcing software to handle all routing responsibilities, might increase the speed of transmission between processors. Software would be free to handle data processing alone, perhaps allowing an overall decrease in processing time. Lastly, different topologies such as a three-dimensional mesh would certainly be possible with larger FPGAs, and would provide a greater degree of scalability and bi-sectional bandwidth. The connectivity of the mesh would also increase, with six avenues of passing data between processors (i.e., two FSL ports for each of the three dimensions). Other benefits of such a topology might include different parallelization options and a reduction in network degree enabling faster communication between otherwise potentially distant processors in two-dimensions.

Appendix A

Aurora-FSL Switch HDL

This is one block of the Switch that corresponds to a single MGT. Each FSL is consumed until the FSL is empty or 32 words have been transmitted (a value that can be changed). The next FSL to be checked is determined by the current FSL being consumed. This ensures no FSL dominates the Switch.

```
// ***** Aurora Channel 0 *****
// FSL Interface, 6 separate FSLs
assign FSL0_S_Clk    =    user_clk_i;    // set to 125 MHz which is reduced by half
assign FSL0_M_Clk    =    user_clk_i;    //    for 4-byte interfaces such as this
assign FSL1_S_Clk    =    user_clk_i;
assign FSL1_M_Clk    =    user_clk_i;
assign FSL2_S_Clk    =    user_clk_i;
assign FSL2_M_Clk    =    user_clk_i;
//
wire    [0:31]    FSL0_S_Data;
wire              FSL0_S_Read;
reg               fsl0_s_read_reg;
wire              FSL0_S_Exists;
reg     [0:31]    FSL0_M_Data;
reg               FSL0_M_Write;
wire              FSL0_M_Full;
//
wire    [0:31]    FSL1_S_Data;
wire              FSL1_S_Read;
reg               fsl1_s_read_reg;
```

```
wire              FSL1_S_Exists;
reg     [0:31]    FSL1_M_Data;
reg               FSL1_M_Write;
wire              FSL1_M_Full;
//
wire    [0:31]    FSL2_S_Data;
wire              FSL2_S_Read;
reg               fsl2_s_read_reg;
wire              FSL2_S_Exists;
reg     [0:31]    FSL2_M_Data;
reg               FSL2_M_Write;
wire              FSL2_M_Full;
//
// Framing TX Interface
reg     [31:0]    tx_d_0;        //
wire    [0:31]    tx_d_i_0;
wire              tx_src_rdy_n_i_0;
reg               tx_src_rdy_n_0;    //
//
wire    [0:1]     tx_rem_i_0;
wire              tx_sof_n_i_0;
reg               tx_sof_n_0;    //
wire              tx_eof_n_i_0;
//
wire              ufc_tx_req_n_i_0;
reg               ufc_tx_req_n_0;    //
wire    [0:2]     ufc_tx_ms_i_0;
wire              ufc_tx_ack_n_i_0;
//
wire              tx_dst_rdy_n_i_0;
// Framing RX Interface
wire    [0:31]    rx_d_i_0;
wire              rx_src_rdy_n_i_0;
//
wire    [0:1]     rx_rem_i_0;
wire              rx_sof_n_i_0;
wire              rx_eof_n_i_0;
//
wire    [0:31]    ufc_rx_d_i_0;
wire    [0:1]     ufc_rx_rem_i_0;
wire              ufc_rx_src_rdy_n_i_0;
wire              ufc_rx_sof_n_i_0;
wire              ufc_rx_eof_n_i_0;
```

```
// Error Detection Interface
wire              hard_error_i_0;
wire              soft_error_i_0;
// Status
wire              channel_up_i_0;
wire              lane_up_i_0;
// Clock Compensation Control Interface
wire              warn_cc_i_0;
wire              do_cc_i_0;
//
// **** Aurora Lane 0 write and read processes ****         NORTH switch
assign    tx_d_i_0            = tx_d_0;
assign    tx_src_rdy_n_i_0    = tx_src_rdy_n_0;
assign    tx_rem_i_0          = 2'b11;
assign    tx_sof_n_i_0        = tx_sof_n_0;
assign    tx_eof_n_i_0        = 1'b1;//tx_eof_n_0;
assign    ufc_tx_req_n_i_0    = ufc_tx_req_n_0;
assign    ufc_tx_ms_i_0       = 3'd1;
//
assign    FSL0_S_Read    = fsl0_s_read_reg;
assign    FSL1_S_Read    = fsl1_s_read_reg;
assign    FSL2_S_Read    = fsl2_s_read_reg;
//
// *** Begin TX Switch FSM ***
//
localparam FSL0    = 2'b00,
           FSL1    = 2'b01,
           FSL2    = 2'b10,
           NONE    = 2'b11;
//
reg     [5:0]     tx_state_0, next_tx_state_0;
reg     [1:0]     tx_last_0;
reg     [1:0]     rx_dest_0;
reg     [5:0]     tx_count_0;
wire              tx_limit_0;
reg               retrieve_0;
//
assign    tx_limit_0    = & tx_count_0;    // used if switch stays at one FSL for 32
                                           // words, then forces switch to change FSLs
                                           // (it is a ball-hogger preventative measure)

// main sequential block
always @(posedge user_clk_i) begin
```

```
    if (reset_i)
        tx_state_0 <= S4;
    else
        tx_state_0 <= next_tx_state_0;
end
//
// combinational block
always @* begin
    next_tx_state_0 = IDLE;    // default state assignment
    case(tx_state_0)
        IDLE: begin
            if (FSL0_S_Exists || FSL1_S_Exists || FSL2_S_Exists)
                next_tx_state_0 = S0;
            else
                next_tx_state_0 = IDLE;
        end
        S4: begin
            if (channel_up_i_0 && ~tx_dst_rdy_n_i_0)
                next_tx_state_0 = IDLE;
            else
                next_tx_state_0 = S4;
        end
        S0: begin
            if (~ufc_tx_ack_n_i_0) begin
                if (tx_last_0 == NONE || tx_last_0 == FSL2) begin
                    if (FSL0_S_Exists)        // here, FSL0 has priority, then FSL1,
                        next_tx_state_0 = S1; // then FSL2 and each section rotates
                    else if (FSL1_S_Exists)   // priorities so that no FSL hogs the
                        next_tx_state_0 = S2; // switch for more than 'too_long' # of
                    else if (FSL2_S_Exists)   // words
                        next_tx_state_0 = S3;
                    else
                        next_tx_state_0 = IDLE;
                end
                else if (tx_last_0 == FSL0) begin
                    if (FSL1_S_Exists)
                        next_tx_state_0 = S2;
                    else if (FSL2_S_Exists)
                        next_tx_state_0 = S3;
                    else if (FSL0_S_Exists)
                        next_tx_state_0 = S1;
                    else
                        next_tx_state_0 = IDLE;
```

```
        end
        else if (tx_last_0 == FSL1) begin
            if (FSL2_S_Exists)
                next_tx_state_0 = S3;
            else if (FSL0_S_Exists)
                next_tx_state_0 = S1;
            else if (FSL1_S_Exists)
                next_tx_state_0 = S2;
            else
                next_tx_state_0 = IDLE;
        end
    end
    else
        next_tx_state_0 = S0;
end
S1: begin    // FSL0
    if (tx_limit_0) begin                  // These three states check first
        if (FSL1_S_Exists || FSL2_S_Exists)// whether the word limit has been
            next_tx_state_0 = S0;          // reached, and then whether other
        else if (FSL0_S_Exists)            // FSLs have data. Remains in its
            next_tx_state_0 = S1;          // state or transitions to either S0
        else                               // for another FSL or IDLE if no FSL
            next_tx_state_0 = IDLE;        // has data... count_x starts over
    end                                    // automagically
    else if (~FSL0_S_Exists && (FSL1_S_Exists || FSL2_S_Exists))
        next_tx_state_0 = S0;       // go to header state if others have data
    else if (FSL0_S_Exists)                // still has data without exceeding
        next_tx_state_0 = S1;              // time, so stay here
    else                           // otherwise, no FSL has data
        next_tx_state_0 = IDLE;
end
S2: begin    // FSL1
    if (tx_limit_0) begin
        if (FSL2_S_Exists || FSL0_S_Exists)
            next_tx_state_0 = S0;
        else if (FSL1_S_Exists)
            next_tx_state_0 = S2;
        else
            next_tx_state_0 = IDLE;
    end
    else if (~FSL1_S_Exists && (FSL2_S_Exists || FSL0_S_Exists))
        next_tx_state_0 = S0;
    else if (FSL1_S_Exists)
```

```
                next_tx_state_0 = S2;
            else
                next_tx_state_0 = IDLE;
        end
        S3: begin    // FSL2
            if (tx_limit_0) begin
                if (FSL0_S_Exists || FSL1_S_Exists)
                    next_tx_state_0 = S0;
                else if (FSL2_S_Exists)
                    next_tx_state_0 = S3;
                else
                    next_tx_state_0 = IDLE;
            end
            else if (~FSL2_S_Exists && (FSL0_S_Exists || FSL1_S_Exists))
                next_tx_state_0 = S0;
            else if (FSL2_S_Exists)
                next_tx_state_0 = S3;
            else
                next_tx_state_0 = IDLE;
        end
        default: begin
            next_tx_state_0 = IDLE;
        end
    endcase
end
//
// secondary sequential block to register outputs
always @(posedge user_clk_i) begin
    if (reset_i) begin
        tx_d_0 <= 32'h0000DEAD;
        tx_src_rdy_n_0 <= 1'b1;
        tx_sof_n_0 <= 1'b1;
        ufc_tx_req_n_0 <= 1'b1;
        fsl0_s_read_reg <= 1'b0;
        fsl1_s_read_reg <= 1'b0;
        fsl2_s_read_reg <= 1'b0;
        tx_count_0 <= 6'd0;
        tx_last_0 <= NONE;
    end
    else begin    // default values
        tx_d_0 <= tx_d_0;
        tx_src_rdy_n_0 <= 1'b1;
        tx_sof_n_0 <= 1'b1;
```

```
ufc_tx_req_n_0 <= 1'b1;
fsl0_s_read_reg <= 1'b0;
fsl1_s_read_reg <= 1'b0;
fsl2_s_read_reg <= 1'b0;
tx_count_0 <= 6'd0;
tx_last_0 <= tx_last_0;
case(tx_state_0)
    IDLE: begin    // reset UFC request signal
        if (FSL0_S_Exists || FSL1_S_Exists || FSL2_S_Exists)
            ufc_tx_req_n_0 <= 1'b0;
    end
    S4: begin      // only used to start frame; meaningless otherwise
        if (channel_up_i_0) begin
            tx_d_0 <= 32'hFADABAC0;
            tx_src_rdy_n_0 <= 1'b0;
            tx_sof_n_0 <= 1'b0;
        end
    end
    S0: begin
        if (~ufc_tx_ack_n_i_0) begin    // UFC request acknowledged!
            if (tx_last_0 == NONE || tx_last_0 == FSL2) begin
                if (FSL0_S_Exists) begin
                    tx_d_0 <= Header_type_0;
                    // S1               // send the particular header of the
                end                     // FSL to be routed
                else if (FSL1_S_Exists) begin
                    tx_d_0 <= Header_type_1;
                    // S2
                end
                else if (FSL2_S_Exists) begin
                    tx_d_0 <= Header_type_2;
                    // S3
                end
            end
            else if (tx_last_0 == FSL0) begin
                if (FSL1_S_Exists) begin
                    tx_d_0 <= Header_type_1;
                    // S2
                end
                else if (FSL2_S_Exists) begin
                    tx_d_0 <= Header_type_2;
                    // S3
                end
```

```
            else if (FSL0_S_Exists) begin
                tx_d_0 <= Header_type_0;
                // S1
            end
        end
        else if (tx_last_0 == FSL1) begin
            if (FSL2_S_Exists) begin
                tx_d_0 <= Header_type_2;
                // S3
            end
            else if (FSL0_S_Exists) begin
                tx_d_0 <= Header_type_0;
                // S1
            end
            else if (FSL1_S_Exists) begin
                tx_d_0 <= Header_type_1;
                // S2
            end
        end
    end
    else    // still waiting for the core to acknowledge UFC request
        ufc_tx_req_n_0 <= 1'b0;
        // S0
end
S1: begin
    tx_last_0 <= FSL0;
    if ((FSL0_S_Exists && ~FSL0_S_Read) && ~tx_dst_rdy_n_i_0)
        fsl0_s_read_reg <= 1'b1;    // starts a read, provides protection
    if (tx_limit_0) begin           // against double reads and tx_dst_rdy_n
        if (FSL1_S_Exists || FSL2_S_Exists)
            ufc_tx_req_n_0 <= 1'b0;
            // S0, since timeout, and others have data
        else if (FSL0_S_Exists && FSL0_S_Read) begin
            if (~tx_dst_rdy_n_i_0) begin
                tx_d_0 <= FSL0_S_Data;
                tx_src_rdy_n_0 <= 1'b0;
                fsl0_s_read_reg <= 1'b1;
                tx_count_0 <= tx_count_0 + 1'b1;
                // S1, remain here, since timeout,
            end // but others have no data and FSL0 still does
        end
    end
    else if (~FSL0_S_Exists && (FSL1_S_Exists || FSL2_S_Exists)) begin
```

```
        ufc_tx_req_n_0 <= 1'b0;
        // S0, since FSL0 has no data, but others do
    end
    else if ((FSL0_S_Exists && FSL0_S_Read) && ~tx_dst_rdy_n_i_0) begin
        tx_d_0 <= FSL0_S_Data;
        tx_src_rdy_n_0 <= 1'b0;
        fsl0_s_read_reg <= 1'b1;
        tx_count_0 <= tx_count_0 + 1'b1;
        // S1 still has data, not too long, can send, so do it
    end
end
S2: begin
    tx_last_0 <= FSL1;
    if ((FSL1_S_Exists && ~FSL1_S_Read) && ~tx_dst_rdy_n_i_0)
        fsl1_s_read_reg <= 1'b1;
    if (tx_limit_0) begin
        if (FSL2_S_Exists || FSL0_S_Exists)
            ufc_tx_req_n_0 <= 1'b0;
        else if (FSL1_S_Exists && FSL1_S_Read) begin
            if (~tx_dst_rdy_n_i_0) begin
                tx_d_0 <= FSL1_S_Data;
                tx_src_rdy_n_0 <= 1'b0;
                fsl1_s_read_reg <= 1'b1;
                tx_count_0 <= tx_count_0 + 1'b1;
            end
        end
    end
    else if (~FSL1_S_Exists && (FSL2_S_Exists || FSL0_S_Exists)) begin
        ufc_tx_req_n_0 <= 1'b0;
    end
    else if ((FSL1_S_Exists && FSL1_S_Read) && ~tx_dst_rdy_n_i_0) begin
        tx_d_0 <= FSL1_S_Data;
        tx_src_rdy_n_0 <= 1'b0;
        fsl1_s_read_reg <= 1'b1;
        tx_count_0 <= tx_count_0 + 1'b1;
    end
end
S3: begin
    tx_last_0 <= FSL2;
    if ((FSL2_S_Exists && ~FSL2_S_Read) && ~tx_dst_rdy_n_i_0)
        fsl2_s_read_reg <= 1'b1;
    if (tx_limit_0) begin
        if (FSL0_S_Exists || FSL1_S_Exists)
```

```
                    ufc_tx_req_n_0 <= 1'b0;
                else if (FSL2_S_Exists && FSL2_S_Read) begin
                    if (~tx_dst_rdy_n_i_0) begin
                        tx_d_0 <= FSL2_S_Data;
                        tx_src_rdy_n_0 <= 1'b0;
                        fsl2_s_read_reg <= 1'b1;
                        tx_count_0 <= tx_count_0 + 1'b1;
                    end
                end
            end
            else if (~FSL2_S_Exists && (FSL0_S_Exists || FSL1_S_Exists)) begin
                ufc_tx_req_n_0 <= 1'b0;
            end
            else if ((FSL2_S_Exists && FSL2_S_Read) && ~tx_dst_rdy_n_i_0) begin
                tx_d_0 <= FSL2_S_Data;
                tx_src_rdy_n_0 <= 1'b0;
                fsl2_s_read_reg <= 1'b1;
                tx_count_0 <= tx_count_0 + 1'b1;
            end
        end
    endcase
  end
end
// End TX FSM
/*****************************************/
// Begin RX process (not quite a FSM)
always @(posedge user_clk_i) begin
  if (reset_i) begin
    FSL0_M_Write <= 1'b0;
    FSL1_M_Write <= 1'b0;
    FSL2_M_Write <= 1'b0;
    rx_dest_0 <= NONE;
    retrieve_0 <= 1'b0;
  end
  else if (~ufc_rx_src_rdy_n_i_0) begin
    case (ufc_rx_d_i_0)
      Header_type_0:
        rx_dest_0 <= FSL0;
      Header_type_1:
        rx_dest_0 <= FSL1;
      Header_type_2:
        rx_dest_0 <= FSL2;
    endcase
```

```
    end
    if ((~rx_src_rdy_n_i_0 && retrieve_0) || retrieve_0) begin
        FSL0_M_Write <= 1'b0;
        FSL1_M_Write <= 1'b0;
        FSL2_M_Write <= 1'b0;
        retrieve_0 <= 1'b0;
        case (rx_dest_0)
            FSL0: begin
                if (~FSL0_M_Full) begin
                    FSL0_M_Data <= rx_d_i_0;
                    FSL0_M_Write <= 1'b1;
                end
            end
            FSL1: begin
                if (~FSL1_M_Full) begin
                    FSL1_M_Data <= rx_d_i_0;
                    FSL1_M_Write <= 1'b1;
                end
            end
            FSL2: begin
                if (~FSL2_M_Full) begin
                    FSL2_M_Data <= rx_d_i_0;
                    FSL2_M_Write <= 1'b1;
                end
            end
            default: begin
                FSL0_M_Write <= 1'b0;
                FSL1_M_Write <= 1'b0;
                FSL2_M_Write <= 1'b0;
            end
        endcase
    end
    else begin
        FSL0_M_Write <= 1'b0;
        FSL1_M_Write <= 1'b0;
        FSL2_M_Write <= 1'b0;
        retrieve_0 <= 1'b0;
    end
    if (~rx_src_rdy_n_i_0)
        retrieve_0 <= 1'b1;
end
//
// *** End RX Switch Process
```

Appendix B

Application Source Code

B.1 Application 1: Data Pumping

A simple `for()` loop repeatedly applies unique data words in each of the four directions: North, East, South, and West. Then, all of the processors enter a `while()` loop to run the following code.

```
void poll_FSLs()
{
    int from_fsl = 0;
    int is_invalid = 0;

    // non-blocking get with validation check
    ngetfsl(from_fsl, NORTH);
    fsl_isinvalid(is_invalid);
    if(!is_invalid)
    {
        // allow data to continue in same direction
        putfsl(from_fsl, SOUTH);
    }

    ngetfsl(from_fsl, EAST);
    fsl_isinvalid(is_invalid);
    if(!is_invalid)
    {
```

```
        putfsl(from_fsl, WEST);
    }

    ngetfsl(from_fsl, SOUTH);
    fsl_isinvalid(is_invalid);
    if(!is_invalid)
    {
        putfsl(from_fsl, NORTH);
    }

    ngetfsl(from_fsl, WEST);
    fsl_isinvalid(is_invalid);
    if(!is_invalid)
    {
        putfsl(from_fsl, EAST);
    }
}
```

B.2 Application 2: Packet-forwarding

The original processor (usually [0,0] for tests) creates a header with itself as the source, an arbitrary destination, and the number of words following the header.

```
int make_header(int src_x, int src_y, int dst_x, int dst_y, int words)
{
    int    header = 0;

    src_x = src_x << 28;
    src_y = src_y << 24;
    dst_x = dst_x << 20;
    dst_y = dst_y << 16;

    header = (((((header | src_x) | src_y) | dst_x) | dst_y) | words);

    return header;
}
```

Each word is sent in the appropriate direction from the originating processor (to the East for XY-routing). Then, all processors run the following example of the polling loop from

above modified for parsing the header and dealing with the packet appropriately.

```
void poll_FSLs()
{
    int    from_fsl = 0;

    int    src_x = 0;
    int    src_y = 0;
    int    src = 0;
    int    dst_x = 0;
    int    dst_y = 0;
    int    dst = 0;
    int    words = 0;
    int    header = 0;

    int    is_invalid = 0;
    int    value = 0;

    int i = 0;

    //begin polling FSLs, North first
    ngetfsl(from_fsl, NORTH);
    fsl_isinvalid(is_invalid);
    // is there data on the FSL?
    if(!is_invalid)
    {   // Received valid word from the North
        // ok, then we already have the header from the check;
        // extract the # of words in the packet and the destination,
        // since this is useful for all subsequent steps
        // 0 = just header
        words = get_words(from_fsl);
        dst = get_dst(from_fsl);
        // is this packet for me?
        if (dst == me)
        {   // This packet is for me
            // extract all of the words from the FSL, since they hold no meaning here
            for(i=0; i<words; i++)
            {
                getfsl(value, NORTH);
                printf("     word %d: 0x%08x\r\n", i, value);
            }
            src_x = (me & 0xF0) >> 4;
            src_y = me & 0x0F;
```

```
    // send packet back to source
    dst_x = get_src_x(from_fsl);
    dst_y = get_src_y(from_fsl);
    header = make_header(src_x, src_y, dst_x, dst_y, words);
    // send packet: header and then data;
    // this would be where calculations would
    // be done on the data, then new data
    // sent somewhere; ascending values are the
    // new data, and the source is now the destination
    putfsl(header, NORTH);
    for (i = 0; i<words; i++)
    {
        putfsl(i, NORTH);
    }
}
// no, it's not for me, so I'll determine where it needs to go...
else if (((dst & 0x0F) == (me & 0x0F)) && ((dst & 0xF0) > (me & 0xF0)))
{    // send packet to East if dst is in this row and in a greater column
    putfsl(from_fsl, EAST);
    for (i = 0; i<words; i++)
    {
        getfsl(from_fsl, NORTH);
        putfsl(from_fsl, EAST);
    }
}
else if (((dst & 0x0F) == (me & 0x0F)) && ((dst & 0xF0) < (me & 0xF0)))
{    // send packet to West if dst is in this row and in a lesser column
    putfsl(from_fsl, WEST);
    for (i = 0; i<words; i++)
    {
        getfsl(from_fsl, NORTH);
        putfsl(from_fsl, WEST);
    }
}
else
{    // dst not in this row, sent it through
    putfsl(from_fsl, SOUTH);
    for (i = 0; i<words; i++)
    {
        getfsl(from_fsl, NORTH);
        putfsl(from_fsl, SOUTH);
    }
}
```

```
    }
    // otherwise, no data available on this FSL...

    // continue checking other FSLs...
}
```

B.3 Application 3: Particle Image Velocimetry

This application was transcribed from Matlab scripts into C code. Data is extracted from memory according to the size of the images and the size of the windows to be processed (larger windows give more general velocity values; smaller windows allow smaller memory requirements, but takes longer to process entire image with small windows). Each iteration shifts the position of the desired window by half the window length. The shift length can be altered, however. Therefore, large windows yield large shifts and *vice versa*. A single window corresponds to one zone per image, or two zones. For the FFT, an imaginary array of the same size as the image data is required and initialized to zero. Then, the complex FFT results are multiplied together to produce the frequency-domain correlation. The IFFT is applied to this data to return it to the time domain. The maximum value is determined and its natural log (or the natural of the surrounding pixels, depending on the maximum value's position in the array) is calculated producing the final velocity values. The following is a single iteration of a window.

```
// calculate windows by columns of the image
n = 0;
m = 0;
for(m=0; m<sizex; m++)    // sizex
{
    for(n=0; n<sizey; n++)    // sizey
{
xmin = X - incx + 1;
xmax = X + incx;
ymin = Y - incy + 1;
ymax = Y + incy;
```

```
// recalculate
Y+=incy;

//find the image windows for zone1 and zone2
q = max(1,ymin-1);
r = min(pix1, ymax);
//
s = max(1,xmin-1);
t = min(pix2, xmax);

// get single zone from DDR SDRAM
// zone1 from image1
i = 0;
j = 0;
for(l = s; l < t; ++l)
{
for(k = q; k < r; ++k)
{
tmp = XIo_In32(SDRAM_BASEADDR + (l*pix1+k)*4);
zone1[i][j] = (float)atoh(tmp);
i++;
}
j++;
i=0;
}


// get single zone from DDR SDRAM
// zone1 from image1
i = 0;
j = 0;
for(l = s; l < t; ++l)
{
for(k = q; k < r; ++k)
{
tmp = XIo_In32(SDRAM_BASEADDR + IM2_BASE + (l*pix1+k)*4);
zone2[i][j] = (float)atoh(tmp);
i++;
}
j++;
i=0;
}
```

```
// initialize the imaginary part of zone signal
for(i=0; i<Ny; i++)
{
for(j=0; j<Nx; j++)
{
imag1[i][j] = 0;
imag2[i][j] = 0;
}
}

// FFT on zone1
fft_r_2d(zone1, imag1, Nx, 1);

fft_r_2d(zone2, imag2, Nx, 1);


// complex conjugate of zone1
for(i=0; i<Ny; i++)
{
for(j=0; j<Nx; j++)
{
imag1[i][j] = imag1[i][j] * (-1);
}
}


// complex array multiplication between zone2 and zone1
for(i=0; i<Ny; i++)
{
for(j=0; j<Nx; j++)
{
tmpr = (zone2[i][j] * zone1[i][j]) - (imag2[i][j] * imag1[i][j]);
tmpi = (zone2[i][j] * imag1[i][j]) + (imag2[i][j] * zone1[i][j]);
zone1[i][j] = tmpr;
imag1[i][j] = tmpi;
}
}

// IFFT to obtain final correlation values
fft_r_2d(zone1, imag1, Nx, -1);


// find maximum value of the correlated zone, and store index where it is found
```

```
tmp = 0;
for(i=0; i<Ny; i++)
{
for(j=0; j<Nx; j++)
{
if(tmp < zone1[i][j])
{
tmp = zone1[i][j];
shifted_x = fftindy[j]; // loc_x fftind_ = [16-31:0-15]
shifted_y = fftindx[i]; // loc_y
reg_x = j;
reg_y = i;
}
}
}


// create proper index arrays for velocities, relative to shifted zone
for(i=0; i<Nx; i++)
{
cc_x[i] = i-incx;
cc_y[i] = i-incy;
}

// get edge pixel positions of the maximum correlation value
if(reg_x == 0)
{
edgel_x = Nx - 1;
edger_x = reg_x + 1;
}
else if (reg_x == Nx-1)
{
edgel_x = reg_x - 1;
edger_x = 0;
}
else
{
edgel_x = reg_x - 1;
edger_x = reg_x + 1;
}
if(reg_y == 0)
{
edgel_y = Ny - 1;
```

```
edger_y = reg_y + 1;
}
else if (reg_y == Ny-1)
{
edgel_y = reg_y - 1;
edger_y = 0;
}
else
{
edgel_y = reg_y - 1;
edger_y = reg_y + 1;
}

// determine x-coordinate shift_error and ln of max correlation value per zone
// that is, the relative velocity
if(shifted_x == 0) // reg_x == 16
shift_err = zone1[reg_y][edger_x] / tmp;
else if(shifted_x == Nx-1) // reg_x == incx - 1
shift_err = -(zone1[reg_y][edgel_x] / tmp);
else if(zone1[reg_y][edger_x] == 0) // poor correlation
shift_err = -(zone1[reg_y][edgel_x] / tmp);
else if(zone1[reg_y][edgel_x] == 0) // poor correlation
shift_err = zone1[reg_y][edger_x] / tmp;
else
{
left = log(zone1[reg_y][edgel_x]);
center = log(zone1[reg_y][reg_x]);
right = log(zone1[reg_y][edger_x]);
if(2*(left+right-2*center) == 0)
shift_err = 0;
else
shift_err = (left - right)/(2*(left+right-2*center));
}
//U[n][m] = cc_x[shifted_x] + shift_err;   // if U and V are matrices, use this
U = cc_x[shifted_x] + shift_err;

// find y-coordinate velocity
if(shifted_y == 0) // reg_y == 16
shift_err = -(zone1[edger_y][reg_x] / tmp);
else if(shifted_y == Ny-1) // reg_y == incy - 1
shift_err = zone1[edgel_y][reg_x] / tmp;
else if(zone1[edger_y][reg_x] == 0) // poor correlation
shift_err = zone1[edgel_y][reg_x] / tmp;
```

```
else if(zone1[reg_y][reg_x] == 0) // poor correlation
shift_err = -(zone1[edger_y][reg_x] / tmp);
else
{
left = log(zone1[edgel_y][reg_x]);
center = log(zone1[reg_y][reg_x]);
right = log(zone1[edger_y][reg_x]);
if(2*(left+right-2*center) == 0)
shift_err = 0;
else
shift_err = (left - right)/(2*(left+right-2*center));
}
//V[n][m] = -(cc_y[shifted_y] + shift_err);
V = -(cc_y[shifted_y] + shift_err);

    } // end of n (sizey)
    X+=incx;
    Y=starty;
} // end of m (sizex)
```

Bibliography

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: the End of the Road for Conventional Microarchitectures," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000, pp. 248–259.

[2] D. Geer, "Chip Makers Turn To Multicore," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[3] P. Gorder, "Multicore Processors for Science and Engineering," *Computing in Science & Engineering*, vol. 9, no. 2, pp. 3–7, March/April 2007.

[4] "Top 500 Supercomputing Sites," http://www.top500.org, October 2006.

[5] A. Agarwal, "The Why, Where and How of Multicore," in *Workshop on Edge Computing Using New Commodity Architectures (EDGE) at University of North Carolina at Chapel Hill*, May 2006. [Online]. Available: http://gamma.cs.unc.edu/EDGE/SLIDES/agarwal.pdf

[6] R. Goering, "Keynote: How multicore will reshape computing," http://www.eetimes.com/news/design/showArticle.jhtml?articleID=198700598, 2007.

[7] T. Trawick, "Multicore Communication: Today and the Future," *Embedded Computing Design*, pp. 29–31, March 2007.

[8] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, May/June 2006.

[9] M. Koning, "TIPC Working Group," Multicore Association Meeting, http://www.multicore-association.org/workgroup/tipc_status_overview.pdf, November 2005.

[10] K. Sangani, "Computing - Two good to be true - Multicore microprocessors may be great for doing lots of things at once, but what about doing one thing more quickly?" *Engineering & Technology*, vol. 2, no. 1, pp. 40–43, January 2007.

[11] S. Guccione, "Simulating A Mulitcore Supercomputer," in *Austin Conference on Integrated Circuits and Systems (ACISC)*, May 2007.

[12] ——, "Cmpware Soft Multiprocessing," http://www.cmpware.com/Docs/SoftMultiprocessing.pdf, 2006.

[13] "The Message Passing Interface (MPI) Standard," http://www-unix.mcs.anl.gov/mpi/, 2007.

[14] A. Patel, C. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow, "A Scalable FPGA-based Multiprocessor," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006, pp. 111–120.

[15] D. Patterson, "RAMP: Research Accelerator for Multiple Processors - a Community Vision for a Shared Experimental Parallel HW/SW Platform," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 19-21 March 2006, p. 1.

[16] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," in *IEEE Micro*, vol. 27, March/April 2007, pp. 46–57.

[17] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," in *IEEE Micro*, vol. 22, March/April 2002, pp. 24–35.

[18] M. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim, "Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *31st Annual International Symposium on Computer Architecture*, 19-23 June 2004, pp. 2–13.

[19] A. Agarwal and M. Levy, "The KILL Rule for Multicore," in *44th ACM/IEEE Design Automation Conference*, 4-8 June 2007, pp. 750–753.

[20] K. Sankaralingam, R. Nagarajan, R. Mcdonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. Keckler, and D. Burger, "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," in *39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006, pp. 480–491.

[21] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder, "Scaling to the end of silicon with EDGE architectures," *Computer*, vol. 37, no. 7, pp. 44–55, July 2004.

[22] E. Chung, E. Nurvitadhi, J. Hoe, B. Falsafi, and K. Mai, "PROToFLEX: FPGA-accelerated Hybrid Functional Simulator," in *IEEE International Parallel and Distributed Processing Symposium*, 26-30 March 2007, pp. 1–6.

[23] H. Zhong, S. Lieberman, and S. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," in *IEEE 13th International Symposium on High Performance Computer Architecture*, 10-14 Feb. 2007, pp. 25–36.

[24] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A Chip-Multiprocessor with Transactional Memory Support," in *Design, Automation & Test in Europe Conference & Exhibition*, 16-20 April 2007, pp. 1–6.

[25] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukolun, "The Stanford Hydra CMP," in *IEEE Micro*, vol. 20, March/April 2000.

[26] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *International Symposium on Computer Architecture*, vol. 27, June 2000.

[27] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. Jones IV, D. Franklin, V. Akella, and F. Chong, "Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor," in *In the 2004 International Symposium on Computer Architecture, Munich, Germany*, 2004.

[28] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003, pp. 291–302.

[29] R. Ramanathan, "Intelő Multi-Core Processors: Making the Move to Quad-Core and Beyond," White Paper, http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf, September 2006.

[30] A. Leon, B. Langley, and J. L. Shin, "The UltraSPARC T1 Processor: CMT Reliability," in *IEEE Custom Integrated Circuits Conference*, 10-13 Sept. 2006, pp. 555–562.

[31] A. S. Leon and D. Sheahan, "The U1traSPARC T1: A Power-Efficient High-Throughput 32-Thread SPARC Processor," in *IEEE Asian Solid-State Circuits Conference*, Nov. 2006, pp. 27–30.

[32] "AMD, Inc. Series on 64-bit Computing: It Takes an Architecture to Make a Multicore Processor," White Paper, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/MultiCoreArchitecturePaper1.pdf, 2006.

[33] J. Kahle, "The Cell Processor Architecture," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, 12-16 Nov. 2005, pp. 3–3.

[34] "HyperTransport™ Consortium - General FAQs," http://www.hypertransport.org/consortium/cons_faqs.cfm?m=2, 2007.

[35] S. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages," *IEEE Design & Test of Computers*, vol. 23, no. 5, pp. 375–386, May 2006.

[36] M. Saldana, L. Shannon, and P. Chow, "The Routability of Multiprocessor Network Topologies in FPGAs," in *SLIP '06: Proceedings of the 2006 international workshop on System-level interconnect prediction*. New York, NY, USA: ACM, 2006, pp. 49–56.

[37] *MicroBlaze Processor Reference Guide User Guide 081*, Xilinx, Inc., February 2007.

[38] *Fast Simplex Link (FSL) Bus (v2.10.a) DS499*, Xilinx, Inc., November 2006.

[39] *LogiCORE Aurora v2.6 User Guide 061*, Xilinx, Inc., March 2007.

[40] *RocketIO Transceiver User Guide 024*, Xilinx, Inc., February 2007.

[41] *ML310 User Guide: Virtex-II Pro Embedded Development Platform*, Xilinx, Inc., February 2007.

[42] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994, pp. 37–38.

[43] R. J. Adrian, "Twenty Years of Particle Image Velocimetry," *Experiments in Fluids*, vol. 39, no. 2, pp. 159–169, August 2005.

[44] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992.