Design of a Scalable Memory System for a Multi-Node, Many-Core Chip System

Sam Payne

May 5, 2014

David Wentzlaff

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering

Department of Electrical Engineering
Princeton University

I hereby declare that this independent work report represents my own work in accordance with University regulations.

Sam Payne

Design of a Scalable Memory System for a Multi-Node, Many-Core Chip System

Sam Payne

ABSTRACT

This project aims to design and test the host board for a many-core chip (the Princeton Parallel Processor) whose purpose is to experiment with a number of novel computing concepts, including a clumpy cache coherence framework and bandwidth limiting technology. The host board, implemented on an ML605 Development kit using a Virtex-6 FPGA, connects computing resources and memory resources inside a computing node. This computing node is capable of communicating with other identical nodes in a larger system to share processing and memory resources. This thesis describes the overall structure and goals of the project, including the design of the chip interface, inter-node interface, packet routing, memory controller, and I/O control. Challenges behind each of these goals, along with proposed and implemented solutions, are presented. Challenges addressed include overcoming pin limits, increasing bandwidth across limited channels, abstracting the structure of random access memories, instantiating and interfacing with Xilinx COREgen modules, designing safe mechanisms to transfer signals across clock domains, combining deadlock-free networks in a hierarchical fashion while preserving deadlock-free properties, using Xilinx synthesis flow tools to load custom logic onto FPGAs, and adjusting hardware platforms for a specific purpose.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Professor David Wentzlaff for his guidance on this thesis and his leadership throughout this project. The other members of the team—Yanqi Zhou, Tri Nguyen, Mike McKeown, Yaosheng Fu and Jonathan Balkind—were all instrumental in the design of the Princeton Parallel Processor and determining the requirements for the host board. Yaosheng and Tri put in a great effort to solidify communication between the many-core chip and host board, establishing what kinds of requests would be sent over our network as a part of the Clumpy Cache Coherence Protocol. Mike McKeown helped with and taught me the design flow for Xilinx FPGAs, Synopsis Design Compiler Tools, and Primetime PX tools. Jonathan, who will be completing this project as the chip design comes closer to completion, played an important role in organizing information for this project. I would also like to thank George Touloumes for partnering with me for the part of this project involving the study of asynchronous FIFO design and performance.

I would also like to thank Princeton's electrical engineering department and the School of Engineering and Applied Science for providing the monetary support to make this project possible. I would also like to thank Xilinx for supplying two additional FPGAs that will help this project to expand into an even larger system in the future.

Finally, I would like to thank my parents for supporting me—their encouragement and wisdom have guided me to this point and will continue to guide me for the rest of my life. I would also like to thank Erica, who has proofread this manuscript several times and kept me motivated through many a long night.

Contents

List of Figures

Section I: Introduction

This project focuses on the decisions and tradeoffs involved in designing the memory system of a multi-node computing system containing several many-core chips. In modern computing systems, single-core performance has reached a plateau, and, to maintain high performance, computers have moved to a multi-core model. An additional load is added to the memory sub-system for each core added to a computing system; this creates new bottlenecks in system performance, requiring new and creative solutions to work around new constraints. This paper aims to provide the necessary background and description of a memory system designed for the Princeton Parallel Processor. The chip is currently being designed by Professor David Wentzlaff and his team at Princeton University in partial collaboration with Professor Srini Devadas at the Massachusetts Institute of Technology. The memory system designed for this project includes an efficient interface for multiple computing nodes to communicate, an efficient interface with the many-core chip designed by Wentzlaff's team, a memory controller to abstract DDR3 memory, and a routing mechanism to direct memory requests in the multi-node network. The rest of this paper is organized as follows: The subsequent section provides motivation for the project; Section III describes the planned system as a whole including the chosen hardware platform; Section IV describes challenges associated with the chip interface and our chosen design solutions; Section V describes the interface design between computing nodes; Section VI provides background on memory systems, describes the memory controller, and details how it was integrated in the system; Section VII describes the system's network configuration and techniques for avoiding deadlock; Section VIII describes the process for loading custom logic onto the chosen platform; Section IX describes future work; and Section X Concludes.

Section II: Motivation

Though the number of transistors present on a single die has continued to increase at an exponential rate, as predicted by Moore's Law, additional transistors have provided diminishing returns on single-core performance. In the past, deep pipelining techniques, branch prediction techniques, and performance driven by Denard scaling were enough to improve performance in a pattern similar to Moore's Law. However, due to power constraints and limits in Denard Scaling, these traditional means of improving single-core performance are no longer enough to keep up with performance demand. As a result, most high-performance machines have moved to a multi-core model, boosting performance through parallel execution.

Unfortunately, scaling the number of cores on a chip to increase parallelism is not a trivial task. In particular, the memory system of a multi-core chip becomes much more complex with additional cores. At every level of the memory hierarchy, new problems are introduced that could lead to a significant loss in performance if ignored. These problems include challenges with cache coherence protocols, limits in off-chip memory bandwidth, and proper design and use of on- and off-chip networks. Many solutions exist for addressing these issues for a system containing a small number of cores (between 1 – 16), including snoopy bus based protocols (MSI, MESI) and traditional directory based protocols [1][2]. However, as the number of cores on-chip rapidly increases in the many-core era, reaching numbers closer to 100 cores per chip, these solutions no longer provide the necessary performance without adjustment[3]. It is justifiable to build larger computing systems containing hundreds of cores as applications for cloud computing and demand for powerful, shared machines has grown. However, challenges surrounding scaling memory systems have become a barrier to increasing the number of cores in today's computing systems, forcing another wave of innovation in the design of large computing systems.

Professor David Wentzlaff and his team aim to design and fabricate a many-core chip, currently referred to as the Princeton Parallel Processor, in order to study solutions to these issues. Using a novel bandwidth limiting tool, they aim to study a method of charging clients appropriately for bandwidth use, benefiting both the host and the client. Using a novel cache coherence protocol, they aim to devise a more effective cache

coherence scheme that can scale with a large number of processors. As a part of this larger project, this thesis aims to design the host-board for this chip capable of interfacing with memory, providing I/O communication, and inter-chip communication. The overall system design is laid out in the next section.

Section III: System Overview

This section describes the overall structure of the computing system which the host board designed for this project will be a part of. After an overview, we describe the notable features of the Princeton Parallel Processor closely related to the memory system. We then describe the components of the host board, and the tools used to create these components, in more elaborate detail.

3.1 Full System Description

The overall system is designed to demonstrate a number of novel memory solutions on a large-scale computing system. The system is structured to distribute cores and memory across a scalable number of nodes, each of which contains a host board, memory, and computing resources (one Princeton Parallel Processor). Each node is capable of communicating with neighboring nodes, performing I/O and servicing memory requests. This node-based architecture has been popular for many years and is similar to that of other large computing systems [4][5]. Our first prototype system will contain two such nodes. Figure 1 illustrates a breakdown of this multi-node system.

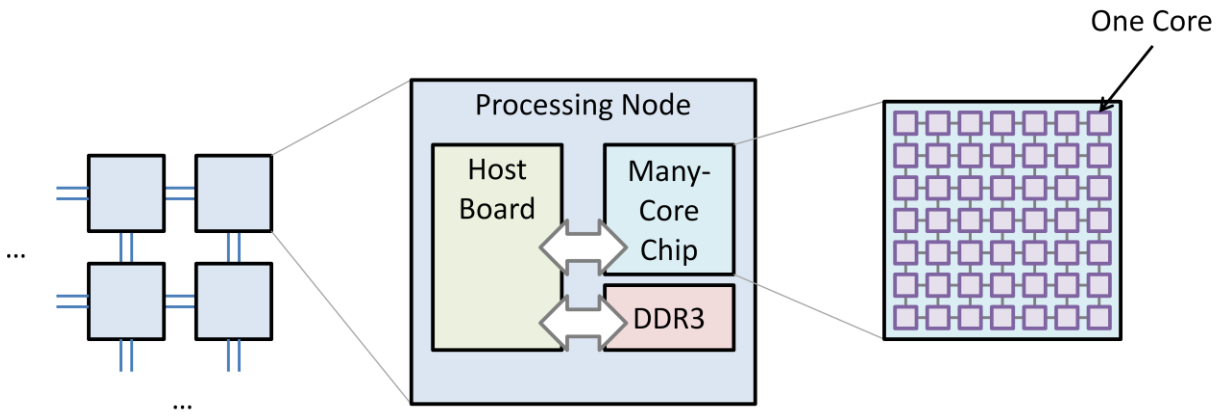


Figure 1: Diagram of the computing system. Each computing node is shown as a blue box with bi-directional links between each node. Each host board (shown in green) connects the Princeton Parallel Processor (blue-green), memory resources (red), and the resources in other nodes in the system.

Each node contains a Princeton Parallel Processor hosted by an FPGA that serves as the chipset for the many-core chip. Each processor can send messages to any other

core in the system, and each node's memory controller can service memory requests from any processor in the system. The network topology of the inter-node routers will be linear for our 2-node prototype, but it will later be changed to a topology built to support larger numbers of nodes. Candidate configurations include arranging nodes in a 2-D mesh, a 3-D grid, or a higher-dimensional structure. This design will be capable of scaling well beyond 100 nodes into multi-dimensional network topologies.

3.2 Chip Overview

The Princeton Parallel Processor built for this project aims to serve as a prototype to demonstrate new concepts in large computing systems. The chip to be built by the team aims to fit about 40 cores per chip. Each core runs on the SPARC instruction set, and each is an identical copy of a stripped-down OpenSPARC T1 core. Some features are removed from the original OpenSPARC T1 architecture in order to fit more cores on the chip. This process includes reducing the threads per core from 4 to 2 and removing the cryptographic unit (SPU). It has been shown that in this kind of system, a greater number of small cores will yield higher performance gains than a smaller number of more powerful cores [6]. This improves performance with little increase in area or power compared to traditional systems containing fewer, more powerful cores [7][8]. The proposed architecture in this system aims to study performance gains in many-core systems in several ways.

The two features closely involved with the memory hierarchy are a novel cache coherence framework coined "Clumpy Shared Memory" and a memory bandwidth controller. It should be noted that these features are important to understand the design decisions on the host board, and so information on them is provided for background, but the design of the Princeton Parallel Processor is not the subject of this paper.

3.2.1 Clumpy Shared Memory

One challenge in multi-core system design is maintaining coherence between private cache memory held by processors. Traditionally, snoopy bus protocols have been used to maintain coherence among a small number of cores; in these schemes, processors share a bus and broadcast their intent to read and write cache lines [1]. If a cache line is held by another processor and is not consistent with memory, the holding cache will intercept read commands to that line, providing the correct data. If a processor intends to

write, causing an inconsistency between caches, then other caches' copies of that data will be invalidated or replaced.

As the number of cores on the same bus increases, the amount of bandwidth supplied by the bus is not enough to service all communication between cores without a significant loss in performance. Though some systems continue to use scaled-up busses[9], bus-based protocols in larger-scale computing systems have been replaced in many cases by directory-based schemes similar to the one used in the SGI Origin system [4]. These schemes reduce unnecessary communication between cores by keeping information about individual cache lines in a directory, therefore allowing requests and communication to be routed only to the caches involved in the particular transaction (rather than broadcasting to multiple cores) [2]. However, directory-based protocols do not scale elegantly in systems with thousands of cores due to the memory overhead required to keep directory information. Additionally, directory-based protocols suffer from high transaction latency in cross-chip communications.

Clumpy Shared Memory (CSM) is a novel cache coherence framework that attempts to overcome the limits of traditional directory protocols by using a directory-based system to divide processors into cache-coherent groups. Groups are determined by process or by page and maintain coherence among cores within the group or "clump." This division is done at runtime and can be dynamically rearranged. It is important to emphasize that this is a cache coherence framework and not a substitute for a protocol; any coherence scheme can be used to maintain coherence among clumps, making clump size highly scalable. By dividing processors into clumps and maintaining coherency only within the clump, communication distances can be reduced, saving bandwidth and energy. Cores contained in the same clump are commonly arranged adjacently, making traffic to home directories local and avoiding the delays associated with cross-chip communication. Clumpy Shared Memory also maintains constant overhead by setting the number of clumps in hardware at fabrication time. Since the number of clumps in a system is independent of the number of cores in the system, significant savings in clump level directory overhead can be realized. The network in the off chip memory system must be built to accommodate the messages required by this framework in a deadlock free manner

3.2.2 Memory Bandwidth Restriction

In many shared computing systems, clients using the system require varying degrees of memory bandwidth—some clients use high amounts of bandwidth, while others limit their use to require only a small amount. In modern pricing schemes, clients demanding more bandwidth pay the same price as those requiring less. The processor in this project will include dedicated hardware to limit the bandwidth of a process. Clients can then be charged for bandwidth use by paying for higher bandwidth allowances. For example, if a client needs a high amount of bandwidth, he or she will pay to maintain a high bandwidth limit, whereas a client who does not need large amount of bandwidth would purchase a modest allowance. A client can never exceed his or her allowance, guaranteeing bandwidth to clients with greater bandwidth needs. This benefits clients by guaranteeing their bandwidth requirements are met while simultaneously saving money for clients who do not require a high bandwidth allowance. This scheme also encourages clients to optimize their workloads to reduce memory bandwidth usage since it is a precious resource. The host also benefits through the opportunity to charge greater amounts for higher bandwidth allowances.

3.3 Chipset Overview

This section details the five modules that will be included in the chipset including the chip interface, the inter-node interface, the in-node RAM interface, the global network topology and routing mechanism, and I/O communication. Creating efficient, reliable modules in these areas is the primary goal of this project. Each of these modules can be found in the block diagram presented below in Figure 2.

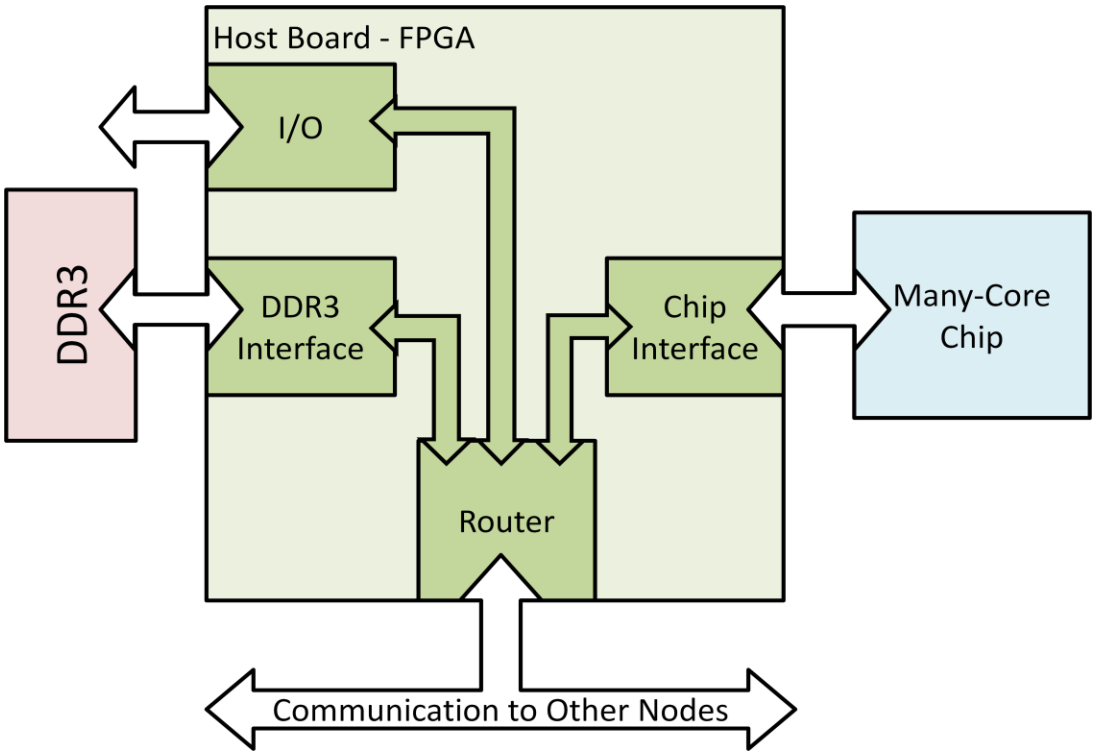


Figure 2: A block diagram of the design of a single node in the computing system. The node contains processing resources, memory resources, and the appropriate interfaces to connect all resources in the node with each other and other nodes in the network.

3.3.1 Interface with the Princeton Parallel Processor

The interface between the host board and the many-core chip requires planning with regard to both the physical interface and the chosen protocol. At the physical layer, the number of pins that can fit on the chip's package is limited. The package we expect to select will have between 218 and 256 pins, roughly 150 of which will be reserved for power and ground. Therefore, bandwidth between the chipset and the chip is limited and must be used effectively. Since all off chip requests and responses must cross this channel, including cache coherence messages, I/O requests, and requests to memory, it is important that this interface does not become a harsh bottleneck which chokes memory bandwidth. To improve bandwidth on this physical channel, the interface will likely run at a different clock frequency from the chip—this will require buffers on both the FPGA

and on the many-core chip capable of bridging clock domains. Section IV describes this interface in more detail.

3.3.2 Inter-Node Interface

The connections between nodes in the system face similar challenges to interfacing with the many-core chip. This interface is even more limited in the number of available pins and is also a critical step in the memory hierarchy. This channel will be responsible for carrying off-node requests from the local many-core chip and from other nodes in the system, including cache coherence messages, memory requests, and I/O requests. It is important that this connection be fast and reliable. Section V describes this interface in more detail.

3.3.3 Memory Controller

Each node in the system will house a 512 MB card of DDR3 RAM. In this report, the node that contains a part of the system's memory will be referred to as that memory's "base node." The base node contains a memory controller that can handle clocking, booting, and reliable use of the DDR3 memory. It must be capable of parsing the packet format in our network into memory requests and sending appropriate replies to requesters in the base node's many-core chip and in other nodes. Section VI describes a typical DRAM organization, the challenges of interfacing with that structure, and our chosen solution to these challenges.

3.3.4 System Network Topology

The base node router directs memory requests from the base node chip and other nodes' processors to local memory and local I/O. The router is also responsible for steering requests off-board to neighboring nodes. This router must be fast enough to handle more requests than are generated by the base node chip in case a load imbalance sends a large number of requests from across the system to a single node. The routing mechanism for the nodes of the system must also be carefully chosen to prevent deadlock in the system. These and other issues are discussed in more detail in Section VII.

3.3.5 I/O Control

Each node in the system will contain a set of I/O interfaces. Candidate interfaces include Serial Bus, USB, Ethernet, and PCIe. The Serial Interface will be extremely useful for debugging during development, chip bring-up, and testing. USB, Ethernet, and

PCIe are secondary goals that may or may not be mandatory in the test system. This project does not focus on the I/O interface since the requirements of this interface are not yet clear enough.

3.4 Chosen Platform

FPGAs are used to design and test the components listed above. The use of FPGAs allows for fast, flexible development with reasonable performance. This is especially useful as we begin to scale the prototype system to include different node-network topologies. It is also important to keep the system flexible as we go through the steps of bringing the chip up to speed after it arrives form fabrication; flexibility in the host board will allow us to make up for unexpected flaws in the many-core chip.

The chosen FPGA platform for this project is the Xilinx ML605 Development Board. The board contains an appropriately sized Virtex-6 FPGA (model xc6vlx240t, speed-grade 1) and necessary peripherals to accommodate this project. The FPGA has 241, 152 logic cells composed of four look up tables (LUTs), multiplexers (Muxes), and arithmetic carry logic [10]. LUTs allow implementation of boolean logic while Muxes enable combinatorial logic [11].

The board contains several I/O communication options, including PCIe, Ethernet, and USB. The ML605 board also has low and high pin count FPGA mezzanine card ports that provide high bandwidth for off-board communication that is essential for achieving inter-node communication and an efficient interface to the many-core chip [12]. In this design we use the 160 pin high-pin-count (HPC) connector to interface with our many-core chip, and the 68 pin low-pin-count (LPC) connector to connect nodes to one another. The board contains hardware that allows off-board DDR3 to interface with user's FPGA designs, and it includes a 512 MB small outline dual inline memory module. The ML605 board also provides a number of on-board peripherals to help debug, including 8 switches, 8 LEDs, an LCD display, USB, UART and JTAG ports.

Xilinx also provides development and debugging tools for the Virtex-6 FPGA included on the ML605 Board. Xilinx provides memory interface generating software compatible with the Virtex-6, allowing for flexible use with a variety of DDR RAM [14]. Xilinx's iMPACT software provides a method of creating and loading designs onto the

FPGA [14]. Figure 3 shows the ML605 board with important hardware features labeled [15].

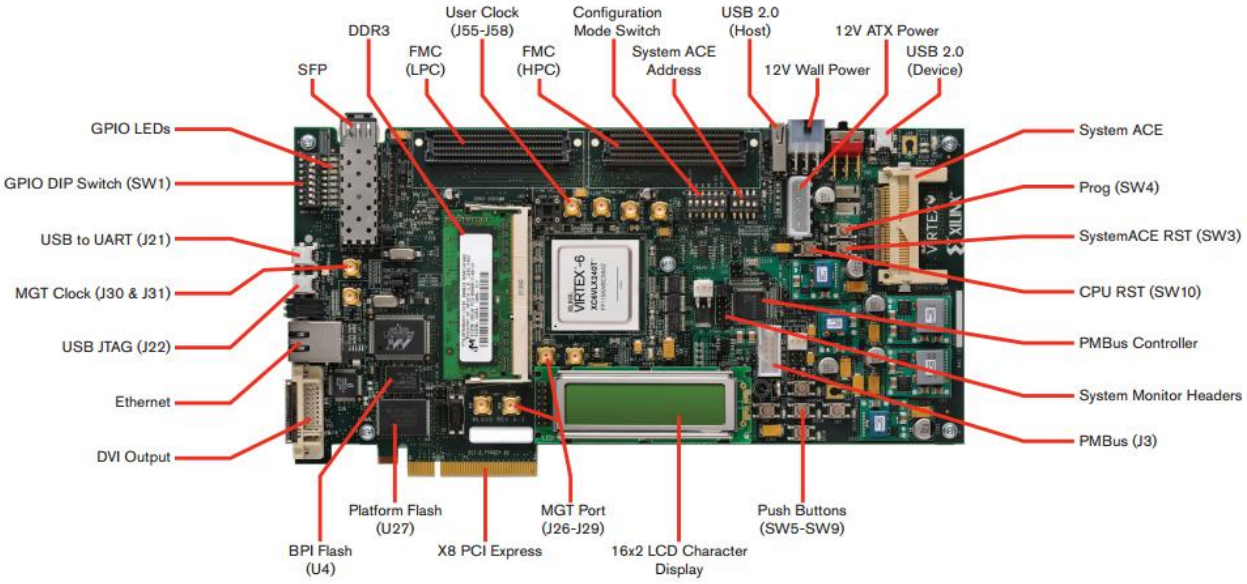


Figure 3: Xilinx ML605 board, including labels for important peripherals. Located at the center is the Virtex-6 FPGA [15].

Section IV: Chip Interface

The interface between the host board and many-core chip poses a variety of design limitations. The package design for the Princeton Parallel Processor has restrictions in terms of pin number and operating frequency. Also, the host board, the many-core chip, and the interface between the two will all run at different frequencies, prompting us to build specialized hardware on both the FPGA and the chip for crossing between clock domains. These challenges and their solutions are described in detail in this section.

4.1 Pin-Count Limits

The package we have selected for the chip will contain between 218 and 256 pins. With about 150 of these pins reserved for power and ground, this leaves fewer than 100 pins for data transfer across the chip. A number of the data pins will be reserved for booting functionality, reset, and debugging information. The rest of this channel will be responsible for carrying all off-chip requests to and from the chip, including cache coherency messages to caches in other chips, memory requests, and I/O requests to the host board and other nodes.

4.1.1 Channel Virtualization

Our system has three physical networks to prevent message dependent deadlock (for more information on message dependent deadlock, see Section VII). Each network is 64 bits wide, so we would need 192 pins to transfer all of these networks off chip (not including credit lines). We would also need 192 additional pins to transfer data from the host board back to the chip (again, not including credit lines), requiring a total of 384 pins. Figure 4 illustrates what this bridge would look like:

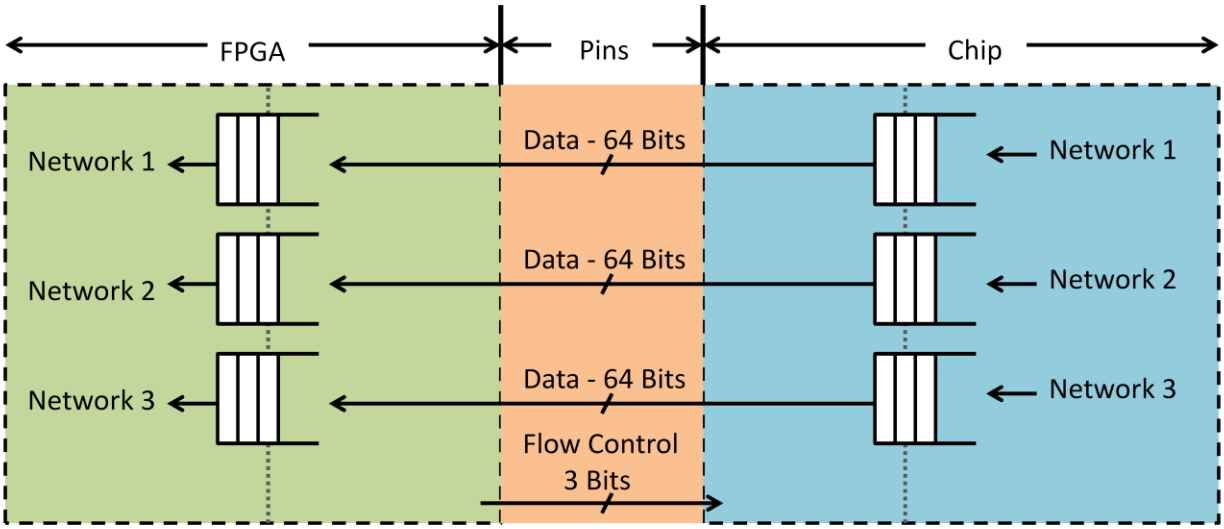


Figure 4: An abstract illustration of a possible interface between the Princeton Parallel Processor (right) and host board (left) across the package pins (center). Each on-chip network is given its own set of 64 pins to communicate off chip. Asynchronous buffers on either side of the interface guarantee safe holding and transfer of data across the channel. This half of the interface carries data from the chip to the FPGA. The other half of the interface carries data the other way, from FPGA to chip.

However, we do not have enough pins to accomplish this, either on our chip or on our 160-pin high-pin-count connector. To reduce our pin requirement, we take advantage of the fact that each network will probably not use the interface 100% of the time. To avoid idle pins, we virtualize the channel for these three networks, sharing 64 pins between the three networks by giving each network exclusive access to all 64 pins for a period of time in round-robin order (or any other fair sharing technique we choose). An illustration of the virtualized chip interface is shown in Figure 5.

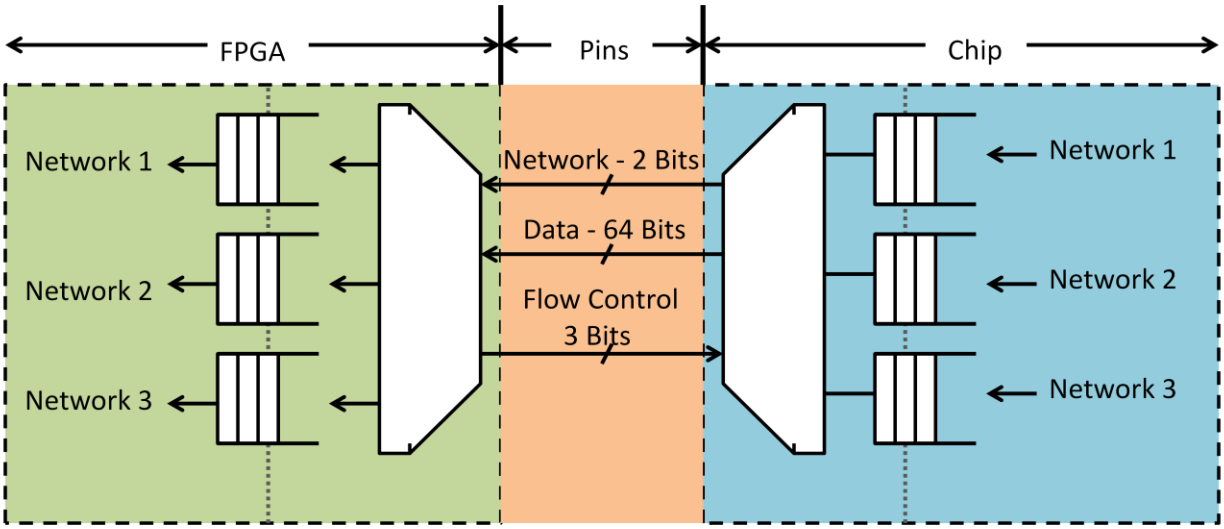


Figure 5: An illustration of a possible interface between the Princeton Parallel Processor (right) and the host board (left) across the physical package channel (center). Arbitration logic chooses which network to select for transfer across the bus.

The two larger blocks in Figure 5 are custom arbitration logic to select which network should be given access to the virtual channel. The buffers shown on the right and left are asynchronous designs that allow signals to bridge the different clock domains of the chip, the pins, and the FPGA (this will be discussed later in this section).

4.1.2 Channel Serialization

The observant reader may notice that though this design meets the 160-pin limit for the HPC connector, it does not meet our pin requirements for our chip package. This design uses 64 pins for data plus 5 pins for credit and networking overhead in each direction, totaling 138 pins, while our requirements demand that less than 100 pins are used. To overcome this, we further reduce our pin count by serializing the channel, dividing each flit into two "fits" that are each 32 bits wide. We send the flit one half at a time, maintaining flow control on the flit level and waiting for the entire flit to be sent across the channel before selecting a new network. An illustration of this system is presented in Figure 6.

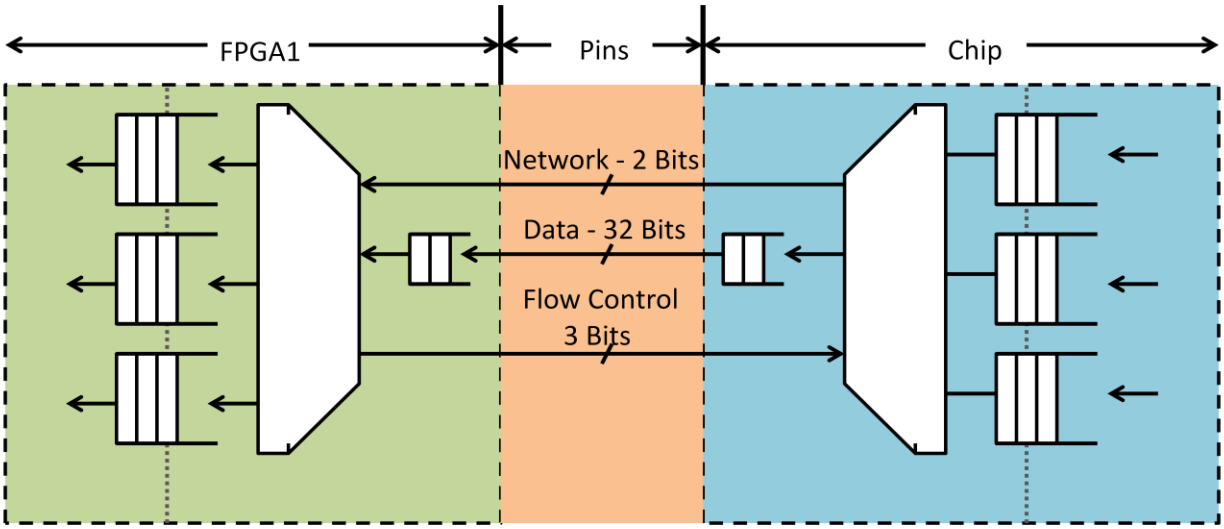


Figure 6: The final FPGA-Chip interface illustrating the serialization of the channel, reducing the pin count to 74. The center buffers represent the division of each 64-bit flit into two 32-bit fits.

This system lowers our pin requirements to 74 pins and has been tested thoroughly in simulation. Source code and tests for this design can be found in Appendices A-G.

4.2 Interface Frequency and Bandwidth

The drawback of this kind of system is that our bandwidth has been significantly reduced. By cutting the number of pins dedicated to data by 32, assuming no change in frequency, we've reduced the bandwidth across the channel by 6 times compared to the design in Figure 4 and 2 times compared to Figure 5. Though we may argue that we would not have been using all of our bandwidth in Figure 4, this kind of bandwidth loss could lead to a bottleneck at the FPGA-Chip interface.

However, since the custom logic built to virtualize the channel is far less complex than logic built on our chip or on the FPGA, the channel can run on a higher frequency clock. The FPGA allows up to a 1 GHz clock on the asynchronous FIFOs. This is 5 times faster than the 200 MHz clock the rest of our FPGA must use, and it effectively changes our bandwidth reduction to 5/6 that in Figure 4. This assumes, of course, that we would have run our channel at 200 MHz in Figure 4, which would not have been unreasonable (running the channel at any higher frequency there would have served no benefit, as that

interface is limited by the buffer speed on the FPGA). Realistically, the channel will be run around 300-500 MHz, and we may need to alter our arbitration policy to prioritize certain networks for better performance.

4.3 Crossing Clock Domains

One of the challenges of interfacing with a chip is the difference in clock speeds over which data is traveling. The chip, the host board, and the interconnect between the two all run at different clock frequencies and may not be aligned in phase. There are two difficulties here. First, if the write domain is faster than the read domain, data will pile up on the writing end. Therefore, our design calls for a buffer to hold data that the read domain is not fast enough to catch. The second challenge addresses setup and hold times. When writing a value to a register, the input to be written must hold the same stable state for a given amount of time before the clock edge in order for the register to maintain the proper value. Otherwise, the output of the register may be incorrect or meta-stable (neither a 1 or a 0). The time that the input value must be held at a 1 or a 0 before the clock edge is called the "setup time." The "hold time" is the amount of time after the clock edge when the input must be held constant before the value is actually saved in the register. If signals from one clock domain are to be stored in the other clock domain, there is no guarantee that inputs are aligned with the reading clock in any way. This can lead to repeated violations of setup and hold times.

4.3.1 Asynchronous Buffer Design

An asynchronous buffer can solve both of these design challenges. The following discussion of asynchronous buffer design is based on[16]. An asynchronous buffer rests at the crossover of two clock domains; it holds values from the write domain and passes those values on to the read domain when it can be read safely. Therefore, the write domain provides write data (wr_data), a write enable signal (wr_en), and a clock (wr_clk), and it receives a "full" signal from the FIFO (fifo_full). The full signal indicates when the FIFO is full, and the read domain must read out values before any more can be written. The read domain, on the other hand, provides a read enable signal (rd_en) and a clock signal (rd_clk), and it receives read data (rd_data) and an "empty" signal (fifo_empty) signaling that there is no valid data in the FIFO.

A naïve implementation may look like the design in Figure 7.

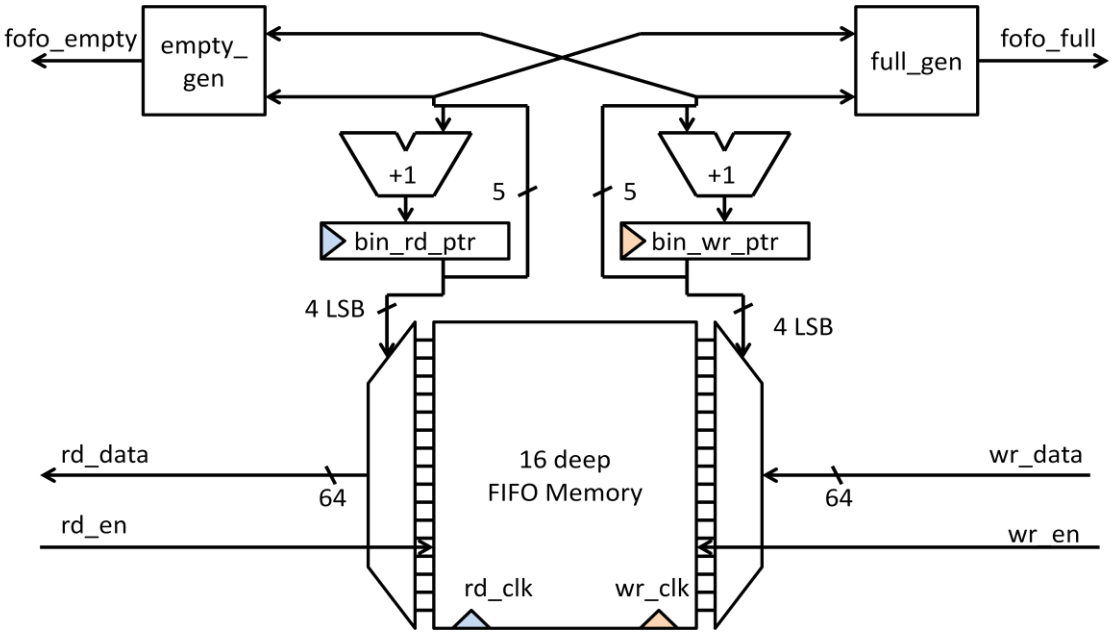


Figure 7: Naïve implementation of an asynchronous FIFO. The write client is on the right, read client is on the left, and the buffer logic is in the center. The FIFO memory is read on the read clock and written on the write clock. BEWARE: this design will not work.

As one may notice, this design will not work. Though we have a buffer in place, we have not solved the issue of violating setup and hold times—the read pointer and write pointer may not be properly captured when they cross the clock domain. There are two reasons for this. The first is that the read and write pointers may violate setup and hold times in the other clock domain. This can be quickly solved by adding synchronizing registers to the design as illustrated in Figure 8.

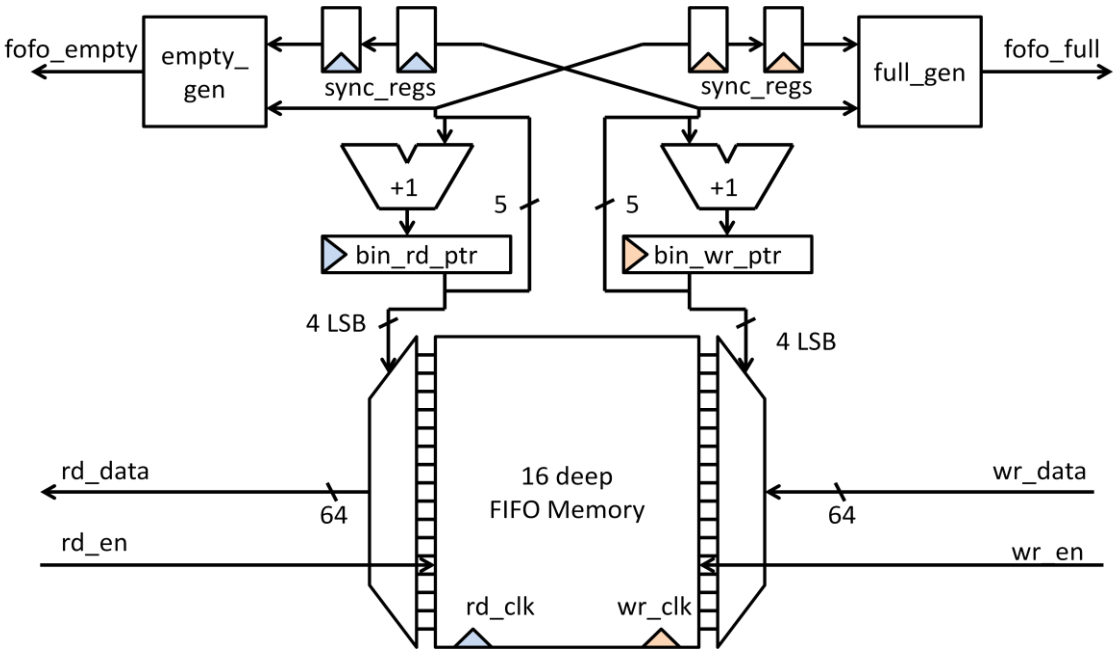


Figure 8: Second attempt at creating a safe asynchronous FIFO. This design corrects possible meta-stabilities when the read and write pointers are read by the full and empty signal generators. WARNING: This design also will not work properly.

The purpose of the synchronization registers is to prevent meta-stability. By running the read and write pointers through registers in the opposite clock domain, their values can become synchronized with the other's clock. The idea is that even if the setup or hold time of the first synchronization register is violated, causing a meta stable output, the output of the first synchronization register will settle on a high or low value before the second register reads it on the next cycle. This is a direct product of the regenerative property of CMOS inverters; the feedback loop inside of the first register will push the value to a stable state, high or low, depending on which is closer. It is highly unlikely that this process will take longer than a clock cycle, and therefore the second register will be capable of accepting a stable value in the next cycle.

This design is conservative in that the read pointer must propagate through the synchronization registers and will appear two cycles late to the logic which generates the full signal (the same goes for the write pointer and the empty signal). Therefore, the FIFO

may seem fuller than it actually is, preventing the overwriting of data but, in some cases, delaying the writing of free slots.

Gray Code

However, this design still does not work for subtle reasons. Suppose the rd_pointer is incrementing from 3 to 4: (011 → 100). If all three digits violate the setup time of the synchronization register in the write-clock domain, their values could be chosen at random (the meta-stable state can settle on either a 0 or 1). Therefore, when the write clock should read 3 (011) or 4 (100), instead it could read 0 (000), 7 (111), or any value between the two. Depending on which value appears on the first register, it is possible that the full signal will not be triggered properly, and data could be lost.

To avoid this data loss, we encode the read and write pointers in such a way that only one meta-stabile bit can result per transition. Gray Code is such an encoding; Figure 9 shows an example of a 4-bit Gray Code next to its binary counterpart. As we can see, the code only transitions one bit at a time.

	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Figure 9: Gray Code shown next to its binary counterpart. Each consecutive value differs by only one digit compared to the previous value when counting in Gray Code.*

This encoding provides the benefit that only 1 bit transitions when incrementing, so in our 3 to 4 example (now 010 → 110), if the most significant bit went meta-stable (the most significant bit being the only one that can go meta-stable), no matter which

value it settles on inside the synchronization registers, the write circuit will read a 3 or a 4—the current value or the previous value. Reading the previous value is not an issue since this would just be an over-conservative estimate of the value.

The disadvantage of Gray Code is that we cannot perform traditional arithmetic on it as we can with binary. Therefore, we must maintain two counters—one in Gray and the other in binary. Gate level diagrams of the conversion circuits are shown in Figure 10 and Figure 11.

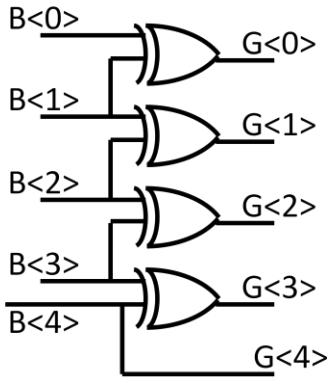


Figure 10: Binary to Gray Encoder. B<x> is the xth bit in the binary bus; G<x> is the xth bit in the Gray Code bus. The maximum gate delay of this circuit is N-1 gate delays, where N is the number of bits in the counter.

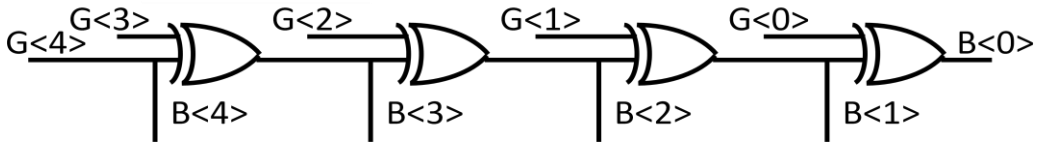


Figure 11: Gray to binary encoder. B<x> is the xth bit in the binary bus; G<x> is the xth bit in the Gray Code bus. The maximum gate delay of this circuit is N-1 gate delays, where N is the number of bits in the counter.

As we can see, the longest delay through these circuits is N-1 gate delays, where N is the number of bits in your binary and Gray Code counter. Therefore, our adjusted (and final) circuit looks like the design shown in Figure 12.

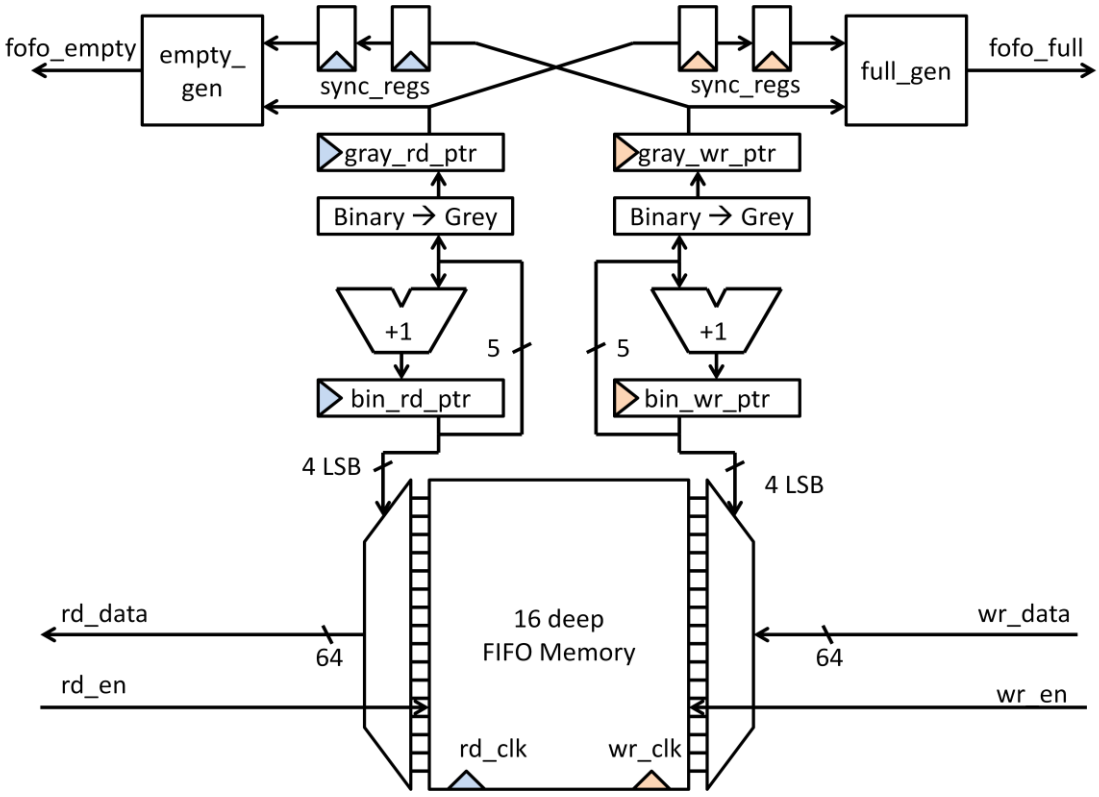


Figure 12: Asynchronous FIFO design with added conversion logic for Gray Code. This design works properly and is the final design used in this project.

Empty and Full Signal Generation

Since Gray Coding is different from binary encoding, it is important to remark on how we generate our empty and full signals form Gray counters to avoid a conversion back to binary. First, we should note that though the FIFO memory can only hold 16 values, we must keep a pointer counter that is 5 bits long. The extra bit aids in the wrap-around case to differentiate between full and empty. In binary, if all 5 bits of the two pointers are equal, then the FIFO is empty. If the most significant bits do not match and the four least significant bits point to the same location, then the FIFO is full.

However, there is a small difference in Gray Code. Due to the coding technique in Gray Coding, the N-1 least significant bits (where N is the number of bits total) cannot be compared properly because the code is symmetric. That is, once the code counts past

$(N-1)^2$-1, the most significant bit flips, and the N-1 least significant bits begin to count back down. This is illustrated in Figure 13.

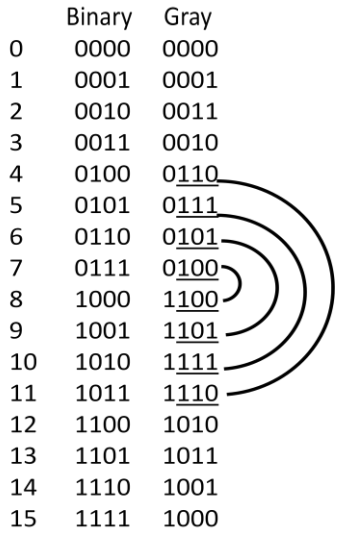


Figure 13: An illustration of symmetry of Gray Code. This logical difference compared to binary makes detecting wrap-around cases in FIFO counters more complex.

Therefore, when the FIFO is full, the first *two* most significant bits will be different, and the least significant N-2 bits will be the same. This only applies when the most significant bits are different and therefore only applies to the full case. Thus, our full and empty generation logic match the logic depicted in Figure 14 and Figure 15.

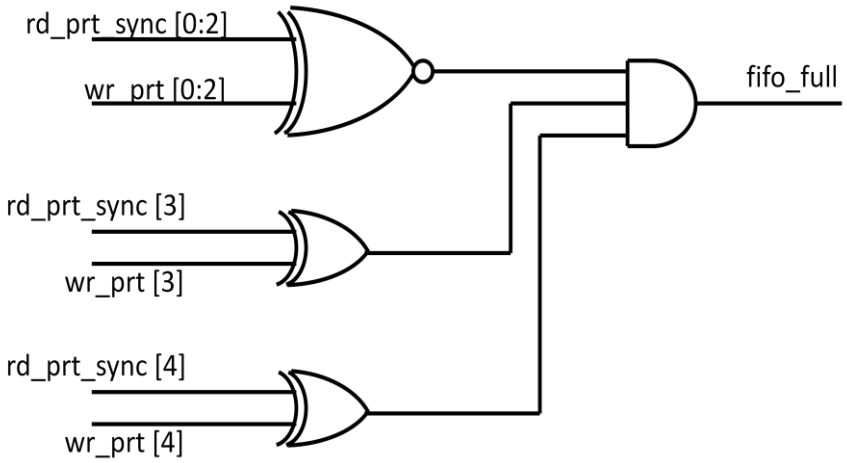


Figure 14: fifo_full generation logic – the three conditions that must be met are fed to an AND gate to determine if the FIFO is full.

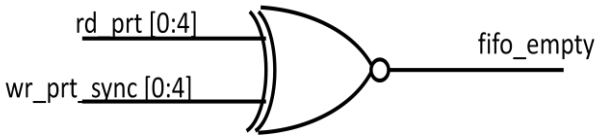


Figure 15: fifo_empty generation logic – essentially, a comparison to determine if the read and synchronized write pointer are equal.

All source code for this asynchronous design is supplied in Appendices H and I. For interested readers, thorough analysis and results co-authored with George Touloumes can be found in Appendix J.

Asynchronous buffers of this kind are needed on both the many-core chip and on the FPGA, as shown in Figure 5. This is required because the FPGA, chip, and channel are all running on different clocks. We use the above design for all asynchronous buffers required on the chip-side of our pin interface. However, this design cannot be used on an FPGA since special lookup table mappings must be used onboard the FPGA to guarantee safe signal propagation (this is a result of the fact that an FPGA uses LUTs instead of traditional logic gates to create designs). Since designing a reliable asynchronous buffer for an FPGA is a very difficult task, Xilinx provides designs for users who need them in their work. This project uses one 64-bit wide, 512-value deep asynchronous FIFO design for each network in the interface, and one 3-bit wide, 512-value deep asynchronous FIFO for credit transfer. The datasheets for these designs, and details of how they were created using Xilinx's COREgen tool, can be found in Appendix K.

As another note, the buffers in this design do not need to be large since this channel is only an intermediate step in the network. As a result, we can save significant space on the chip by keeping our buffers small. All Xilinx buffers are generated at their minimum size.

4.4 Testing Methodology

We put the design through a number of tests at a range of frequencies. First, the buffer was tested for functionality to guarantee that the device could handle different read and write rates. These rates were swept in a two-dimensional space varying read chance from 1% - 100% in steps of 10% while varying write chance from 1% - 100% in steps of 10% as well for a total of 100 tests.

Next, the design was tested at a range of read and write frequencies ranging from 5 – 500 MHz in a two-dimensional space similar to the previous test. We also tested how variations from 1%-100% write chance under constant write frequency and varying read frequency affected latency in the system. The latency, total power, power per bit, and power vs. frequency are also compared and analyzed in more detail in Appendix J. These tests and report were done in collaboration with George Touloumes. Each of the more than 500 tests run were created using automated test bench generators written for this project. An example of one test bench generated can be found in Appendix I.

Section V: FPGA Interface

5.1 Overview

Bridging signals across FPGAs is a similar issue to the one described in the previous section. The difference in this design is that we use the low-pin-count FPGA Mezzanine Connector (LPC FMC) to connect FPGAs rather than the high-pin-count (HPC) connector used for interfacing with the chip. The LPC connector has 68 pins. By dividing flits into 4 parts and serializing our channel using 16 pins rather than 32 pins, we can get all data across the channel both ways using only 42 pins. A diagram of the system is shown in Figure 16.

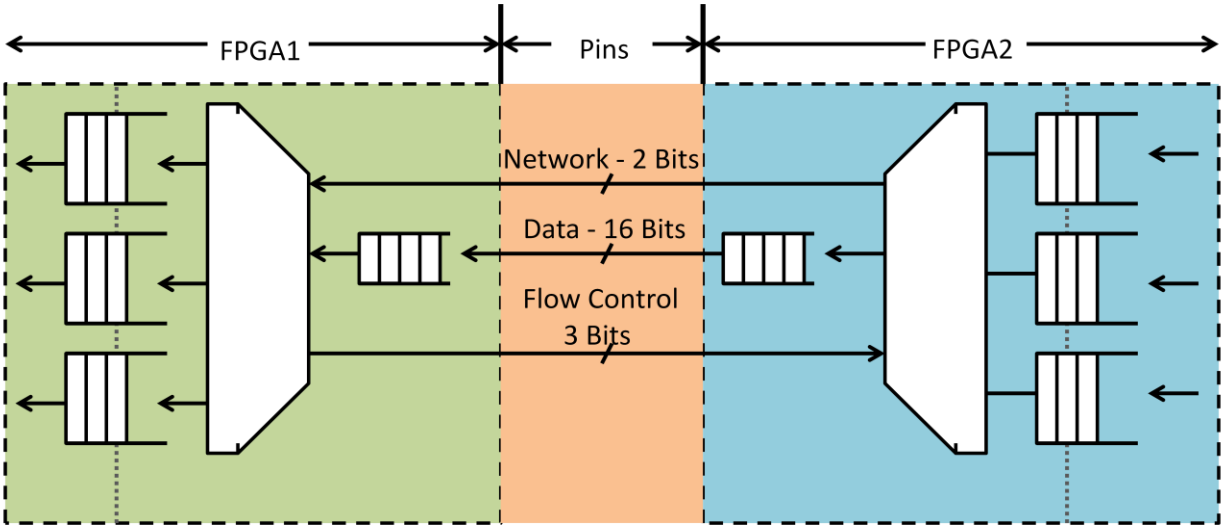


Figure 16: The FPGA-FPGA interface over the LPC connector illustrating the serialization of the channel, reducing the pin-count to 42.

This design has been thoroughly tested in simulation. Source code and tests are similar to those supplied in Appendix A-G, but they are not included in the appendix of this report due to the length and redundancy of the code. All source code for this system will be open sourced when completed.

5.2 Chosen Connectors

Typically, FPGA Mezzanine Connectors are used to connect smaller boards to FPGAs similar to the one that will hold the Princeton Parallel Processor. However, since we are using the LCP FMC to send traffic between FPGAs, we must be careful about which pins we use on the FMC cable. The cable we used was ordered from Samtech (part

number HDR-169475-02). Both ends have the same orientation and are male adapters to connect to the female ports on each board. However, these cables connect all pins, including clock and power pins, across the FMC ports, which could cause potential damage to our host boards. Figure 17 is a diagram of the connected pins on the LPC connector.

	K	J	H	G	F	E	D	C	B	A
1	NC	NC	VREF_A_M2C	GND	NC	NC	PG_C2M	GND	NC	NC
2	NC	NC	PRSNT_M2C_L	CLK1_M2C_P	NC	NC	GND	DP0_C2M_P	NC	NC
3	NC	NC	GND	CLK1_M2C_N	NC	NC	GND	DP0_C2M_N	NC	NC
4	NC	NC	CLK0_M2C_P	GND	NC	NC	GBTCLK0_M2C_P	GND	NC	NC
5	NC	NC	CLK0_M2C_N	GND	NC	NC	GBTCLK0_M2C_N	GND	NC	NC
6	NC	NC	GND	LA00_P_CC	NC	NC	GND	DP0_M2C_P	NC	NC
7	NC	NC	LA02_P	LA00_N_CC	NC	NC	GND	DP0_M2C_N	NC	NC
8	NC	NC	LA02_N	GND	NC	NC	LA01_P_CC	GND	NC	NC
9	NC	NC	GND	LA03_P	NC	NC	LA01_N_CC	GND	NC	NC
10	NC	NC	LA04_P	LA03_N	NC	NC	GND	LA06_P	NC	NC
11	NC	NC	LA04_N	GND	NC	NC	LA05_P	LA06_N	NC	NC
12	NC	NC	GND	LA08_P	NC	NC	LA05_N	GND	NC	NC
13	NC	NC	LA07_P	LA08_N	NC	NC	GND	GND	NC	NC
14	NC	NC	LA07_N	GND	NC	NC	LA09_P	LA10_P	NC	NC
15	NC	NC	GND	LA12_P	NC	NC	LA09_N	LA10_N	NC	NC
16	NC	NC	LA11_P	LA12_N	NC	NC	GND	GND	NC	NC
17	NC	NC	LA11_N	GND	NC	NC	LA13_P	GND	NC	NC
18	NC	NC	GND	LA16_P	NC	NC	LA13_N	LA14_P	NC	NC
19	NC	NC	LA15_P	LA16_N	NC	NC	GND	LA14_N	NC	NC
20	NC	NC	LA15_N	GND	NC	NC	LA17_P_CC	GND	NC	NC
21	NC	NC	GND	LA20_P	NC	NC	LA17_N_CC	GND	NC	NC
22	NC	NC	LA19_P	LA20_N	NC	NC	GND	LA18_P_CC	NC	NC
23	NC	NC	LA19_N	GND	NC	NC	LA23_P	LA18_N_CC	NC	NC
24	NC	NC	GND	LA22_P	NC	NC	LA23_N	GND	NC	NC
25	NC	NC	LA21_P	LA22_N	NC	NC	GND	GND	NC	NC
26	NC	NC	LA21_N	GND	NC	NC	LA26_P	LA27_P	NC	NC
27	NC	NC	GND	LA25_P	NC	NC	LA26_N	LA27_N	NC	NC
28	NC	NC	LA24_P	LA25_N	NC	NC	GND	GND	NC	NC
29	NC	NC	LA24_N	GND	NC	NC	TCK	GND	NC	NC
30	NC	NC	GND	LA29_P	NC	NC	TDI	SCL	NC	NC
31	NC	NC	LA28_P	LA29_N	NC	NC	TDO	SDA	NC	NC
32	NC	NC	LA28_N	GND	NC	NC	3P3VAUX	GND	NC	NC
33	NC	NC	GND	LA31_P	NC	NC	TMS	GND	NC	NC
34	NC	NC	LA30_P	LA31_N	NC	NC	TRST_L	GA0	NC	NC
35	NC	NC	LA30_N	GND	NC	NC	GA1	12P0V	NC	NC
36	NC	NC	GND	LA33_P	NC	NC	3P3V	GND	NC	NC
37	NC	NC	LA32_P	LA33_N	NC	NC	GND	12P0V	NC	NC
38	NC	NC	LA32_N	GND	NC	NC	3P3V	GND	NC	NC
39	NC	NC	GND	VADJ	NC	NC	GND	3P3V	NC	NC
40	NC	NC	VADJ	GND	NC	NC	3P3V	GND	NC	NC

Figure 17: A spreadsheet of connected pins on the LPC connector. This diagram and full descriptions of all pins can be found in the ML605 Development Kit user guide [15]. This diagram marks power pins in red and JTAG pins in orange.

The blocks highlighted in red on this diagram are power pins and had to be disconnected; otherwise, the voltage regulators on the connected ML605 boards would

try to regulate one another and damage the system. The pins marked in orange are JTAG pins that would usually be used to connect a Mezzanine card component for programming through software like Xilinx's iMPACT software. However, since FPGAs are not wired to join one another's scan chains, these pins must be disconnected to avoid conflicting clock drivers and data on these pins. If left intact, it would be impossible to program both boards at once, and it could possibly damage the development kits. The headers on these cables are custom-made, and therefore the trace mappings are available only by request of Samtec (these schematics are not included here for this reason). However, images of the alterations made to these header boards are shown in Figure 18.

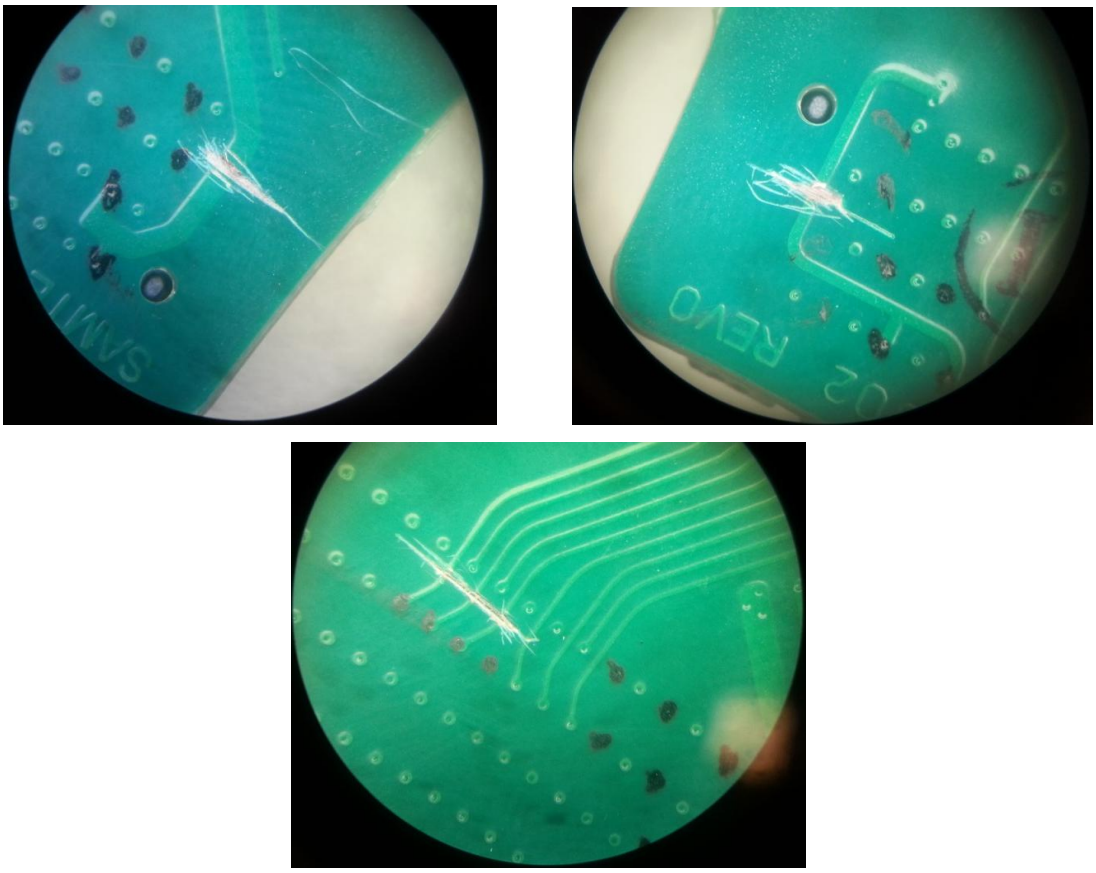


Figure 18: Modifications made to the HPC and LPC header PCB boards. These cuts and images were made using a knife and microscope.

In addition to the LPC connector cable, we also purchased an HPC connector to test our HPC FPGA interface. To save on wiring (since this cable has twice as many pins) the power signals come pre-removed, and so only JTAG connections must be severed.

The pin diagram for the HPC is much larger and can also be found in the ML605 Development Kit Hardware Guide, along with pin descriptions [15]. The position of the JTAG pins in the HPC connector is identical to the position of the JTAG pins in the LPC connector.

	K	J	H	G	F	E	D	C	B	A
1	VREF_B_M2C	GND	VREF_A_M2C	GND	PG_M2C	GND	PG_C2M	GND	RESI	GND
2	GND	CLK3_M2C_P	PRSNT_M2C_L	CLK1_M2C_P	GND	HA01_P_CC	GND	DP0_C2M_P	GND	DP1_M2C_P
3	GND	CLK3_M2C_N	GND	CLK1_M2C_N	GND	HA01_N_CC	GND	DP0_C2M_N	GND	DP1_M2C_N
4	CLK2_M2C_P	GND	CLK0_M2C_P	GND	HA00_P_CC	GND	GBTCLK0_M2C_P	GND	DP9_M2C_P	GND
5	CLK2_M2C_N	GND	CLK0_M2C_N	GND	HA00_N_CC	GND	GBTCLK0_M2C_N	GND	DP9_M2C_N	GND
6	GND	HA03_P	GND	LA00_P_CC	GND	HA05_P	GND	DP0_M2C_P	GND	DP2_M2C_P
7	HA02_P	HA03_N	LA02_P	LA00_N_CC	HA04_P	HA05_N	GND	DP0_M2C_N	GND	DP2_M2C_N
8	HA02_N	GND	LA02_N	GND	HA04_N	GND	LA01_P_CC	GND	DP8_M2C_P	GND
9	GND	HA07_P	GND	LA03_P	GND	HA09_P	LA01_N_CC	GND	DP8_M2C_N	GND
10	HA06_P	HA07_N	LA04_P	LA03_N	HA08_P	HA09_N	GND	LA06_P	GND	DP3_M2C_P
11	HA06_N	GND	LA04_N	GND	HA08_N	GND	LA05_P	LA06_N	GND	DP3_M2C_N
12	GND	HA11_P	GND	LA08_P	GND	HA13_P	LA05_N	GND	DP7_M2C_P	GND
13	HA10_P	HA11_N	LA07_P	LA08_N	HA12_P	HA13_N	GND	GND	DP7_M2C_N	GND
14	HA10_N	GND	LA07_N	GND	HA12_N	GND	LA09_P	LA10_P	GND	DP4_M2C_P
15	GND	HA14_P	GND	LA12_P	GND	HA16_P	LA09_N	LA10_N	GND	DP4_M2C_N
16	HA17_P_CC	HA14_N	LA11_P	LA12_N	HA15_P	HA16_N	GND	GND	DP6_M2C_P	GND
17	HA17_N_CC	GND	LA11_N	GND	HA15_N	GND	LA13_P	GND	DP6_M2C_N	GND
18	GND	HA18_P	GND	LA16_P	GND	HA20_P	LA13_N	LA14_P	GND	DP5_M2C_P
19	HA21_P	HA18_N	LA15_P	LA16_N	HA19_P	HA20_N	GND	LA14_N	GND	DP5_M2C_N
20	HA21_N	GND	LA15_N	GND	HA19_N	GND	LA17_P_CC	GND	GBTCLK1_M2C_P	GND
21	GND	HA22_P	GND	LA20_P	GND	HB03_P	LA17_N_CC	GND	GBTCLK1_M2C_N	GND
22	HA23_P	HA22_N	LA19_P	LA20_N	HB02_P	HB03_N	GND	LA18_P_CC	GND	DP1_C2M_P
23	HA23_N	GND	LA19_N	GND	HB02_N	GND	LA23_P	LA18_N_CC	GND	DP1_C2M_N
24	GND	HB01_P	GND	LA22_P	GND	HB05_P	LA23_N	GND	DP9_C2M_P	GND
25	HB00_P_CC	HB01_N	LA21_P	LA22_N	HB04_P	HB05_N	GND	GND	DP9_C2M_N	GND
26	HB00_N_CC	GND	LA21_N	GND	HB04_N	GND	LA26_P	LA27_P	GND	DP2_C2M_P
27	GND	HB07_P	GND	LA25_P	GND	HB09_P	LA26_N	LA27_N	GND	DP2_C2M_N
28	HB06_P_CC	HB07_N	LA24_P	LA25_N	HB08_P	HB09_N	GND	GND	DP8_C2M_P	GND
29	HB06_N_CC	GND	LA24_N	GND	HB08_N	GND	TCK	GND	DP8_C2M_N	GND
30	GND	HB11_P	GND	LA29_P	GND	HB13_P	TDI	SCL	GND	DP3_C2M_P
31	HB10_P	HB11_N	LA28_P	LA29_N	HB12_P	HB13_N	TDO	SDA	GND	DP3_C2M_N
32	HB10_N	GND	LA28_N	GND	HB12_N	GND	3P3VAUX	GND	DP7_C2M_P	GND
33	GND	HB15_P	GND	LA31_P	GND	HB19_P	TMS	GND	DP7_C2M_N	GND
34	HB14_P	HB15_N	LA30_P	LA31_N	HB16_P	HB19_N	TRST_L	GA0	GND	DP4_C2M_P
35	HB14_N	GND	LA30_N	GND	HB16_N	GND	GA1	12P0V	GND	DP4_C2M_N
36	GND	HB18_P	GND	LA33_P	GND	HB21_P	3P3V	GND	DP5_C2M_P	GND
37	HB17_P_CC	HB18_N	LA32_P	LA33_N	HB20_P	HB21_N	GND	12P0V	DP5_C2M_N	GND
38	HB17_N_CC	GND	LA32_N	GND	HB20_N	GND	3P3V	GND	GND	DP5_C2M_P
39	GND	VIO_B_M2C	GND	VADJ	GND	VADJ	GND	3P3V	GND	DP5_C2M_N
40	VIO_B_M2C	GND	VADJ	GND	VADJ	GND	3P3V	GND	RESO	GND

Figure 19: A spreadsheet of connected pins on the HPC connector. Note that the JTAG pins are in the same location as the LPC connector: D29, D30, D31, D33. This diagram and full descriptions of all pins can be found in the ML605 Development Kit user guide [15].

Unfortunately, the headers on these cables are much larger than those on the LCP connector and physically conflict with other components on the ML605 board. Therefore, alterations were made to a number of coaxial connections on one of the ML605 boards. Images of this change are shown in Figure 20. Luckily, these changes still allow for use of these connectors (with some loss of shielding), though they cannot be used at the same time as the HPC connector.

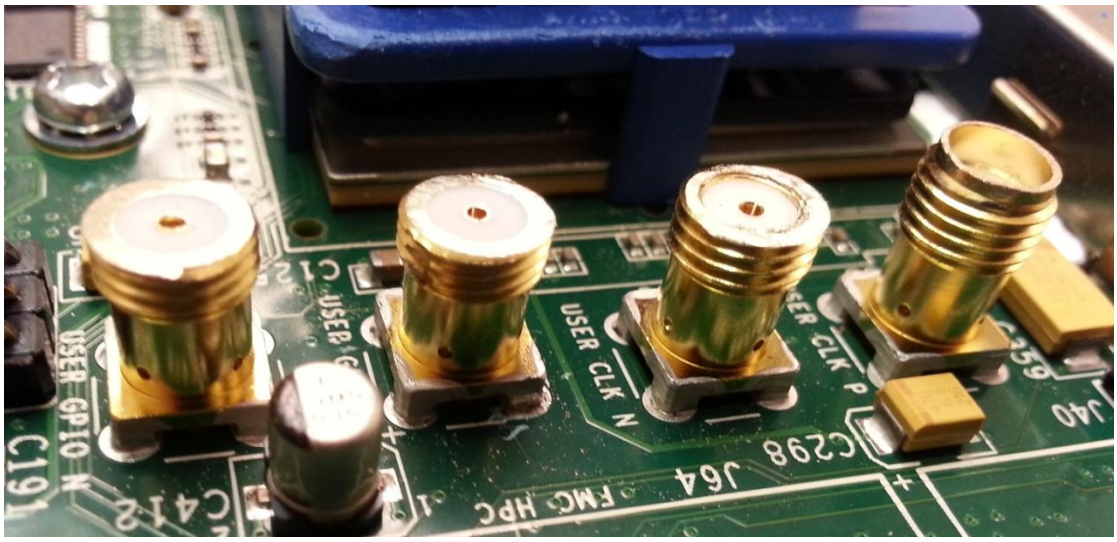


Figure 20: Modified coax headers on the ML605 board. The three on the left have been modified; the header on the right is untouched.

5.3 Testing Methodology

Both the chip-FPGA and FPGA-FPGA interfaces were tested in software using different read and write loads (swept from 10-100% in steps of 10%). Since Xilinx Asynchronous FIFOs are built to work with specific frequencies, it would be burdensome to regenerate the entire design in an effort to simulate across frequencies. The FPGA was simulated at 200 MHz and the pins at 1 GHz. All interfaces were able to successfully transfer more than 2,048 random test values across all 3 networks without any failures.

The next step in testing these interfaces includes building a synthesizable test-bench, half of which is on one FPGA and the other half on the other. The test benches send data to one another across the HPC and LPC connectors, and they are synchronized with the same values to test that the data they receive is correct. This testing structure guarantees the logic that we have built for the FPGA works. Unfortunately, we cannot test our chip-side logic outside of simulation since the many-core chip will not return from fabrication until much later.

Section VI: Memory Controller

This section provides a description of the challenges encountered when interfacing with DDR RAM. A background of RAM provides a summary of the technology, and an overview of system-specific problems will be addressed alongside our chosen solutions. We close with our solution to integrate the selected Xilinx memory controller with the networks in our system.

6.1 RAM Background

6.1.1 Random Access Memory (RAM)

RAM is the most popular type of memory used in digital systems for three reasons: It is cheap, power-efficient, and reasonably fast. RAM is cheap because the area per bit stored is very low, it is power-efficient due to recent advancements in the technology, and it is fast because the address being read or written does not affect the speed at which transactions take place (thus the term "random access"). RAM, however, is volatile, meaning it does not hold the values stored in it when power is removed, unlike a disk or solid state drives. RAM comes in two flavors: SRAM and DRAM.

6.1.2 Static Random Access Memory (SRAM)

SRAM, or Static Random Access Memory, uses four or six transistors to store each bit as shown in Figure 21.

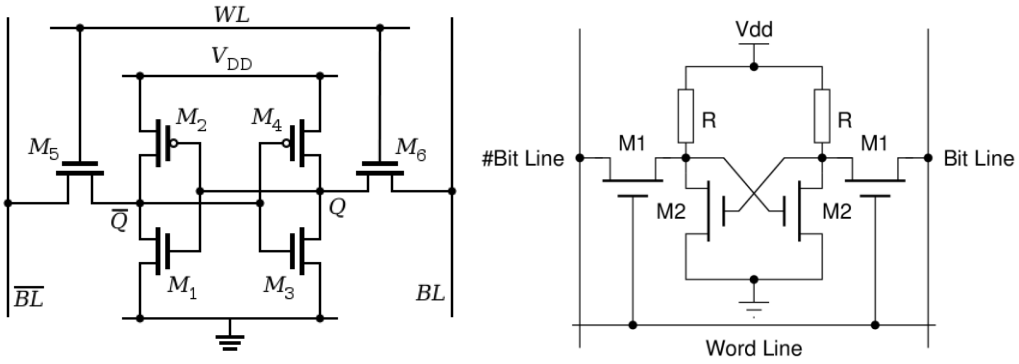


Figure 21: Standard one-bit SRAM cells. On the left, a six-transistor implementation using a flipflop and two access transistors [17]. On the right, a four-transistor implementation, replacing two transistors with resistors for slight area savings [18]. In both, a word line controls access to the cell.

Compared to DRAM, SRAM is faster since it does not use capacitors and sense amplifiers, which, though they take up less area, add to the delay of reads and writes (as

we will see). Very often, SRAM is used in cache memory in computers [19]. However, SRAM is less dense than DRAM and is therefore not the cell of choice for larger main memory chips.

6.1.3 Dynamic Random Access Memory (DRAM)

Figure 22 provides a good illustration to help clarify the following discussion. A single "stick" of DRAM is formally referred to as a dual in-line memory module (DIMM), which is a circuit board with a number of DRAM chips on board. The DRAM chips on the DIMM can function as a single *rank* or can be divided into two *ranks*. Each rank functions independently and can be regarded as a group of DRAM chips that can receive commands independently [20].

DRAM chips contain arrays of memory; the arrays are arranged by row and column. An address given to a DRAM chip will specify the row and column of the bit to be read from the array. The bit storage mechanism consists of a capacitor (which stores the value at that row/column address) gated by one or more transistors that control read and write at that address. To read out from an array, a word line must be driven to activate the transistor and read out the value charged on the capacitor. Capacitors in DRAM slowly discharge, so sense-amplifiers are necessary for readout. This is also the reason why DRAM must be refreshed periodically and is considered a volatile form of memory. The value on the capacitor is read out by a sense-amplifier that is responsible for classifying the charge held on the capacitor as a 1 or a 0. Driving word lines for one address reads out the entire row based on the row address (RAS) sent by the memory controller. The column address (CAS) specifies which element of the row to read out. Since it takes time to drive the line of an entire row, it is often convenient to read out values from the same row consecutively (also known as reading from the "open" row or "burst mode"). Closing a row and opening another takes time as well, so some schemes will close a row before servicing the next request to save time when servicing the next request (assuming it is on another row) [20, 21].

DDR3 chips contain more than one array. Arrays are read out in unison to read out full words in one row-column read. For example, in a x8 (pronounced “by eight”) DRAM chip, the chip contains at least 8 arrays. Each time a column read is executed, 1 bit is read from each of the eight arrays simultaneously, producing an 8-bit word. Groups of arrays can be arranged by *bank*. Each bank in a x8 DRAM chip contains 8 arrays. Banks function similar to ranks in that they function independently; with few exceptions banks can be read, written to, and refreshed independently of one another. To take advantage of DRAM banks, a system often interweaves memory requests by using a data bus frequency faster than a single bank can utilize. This allows higher bandwidth on the bus by parallelizing reads in the banks. A summary of these structures can be found in Figure 22.

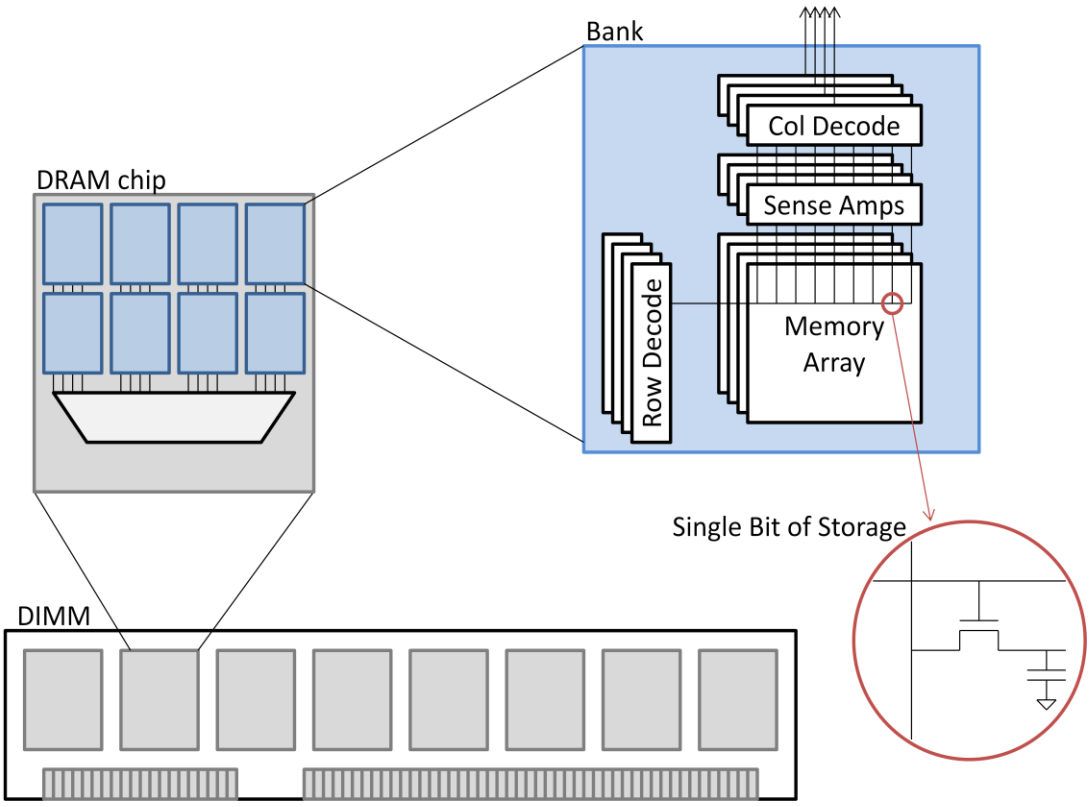


Figure 22: Basic internal organization of a x4 DRAM DIMM.

As DRAM has evolved, more modifications have been added to this model to increase throughput of these systems [20]. DDR DRAM chips have a “double data rate”

compared to older RAM chips. This is because they read out values on the rising and falling edge of the clock, doubling their bandwidth. As DRAM has become more advanced, each generation has used less power as a result of decreased supply voltage (DDR1: 2.5 V, DDR2: 1.8V, DDR3: 1.5V, 1.35V) [21]. This improvement can be attributed to advancements in low voltage signaling and detection technology.

The memory controller must abstract the complex structure of the DRAM in order to create a flat address space to which multiple processors can make requests. The controller is responsible for gathering and queuing memory requests, translating these requests into DRAM compatible command sequences, reading data back from DRAM, and returning data to requesters[20]. At the physical level, the controller must supply reset, clock, refresh, data in, and data out command signals and enable signals to the DIMM [14][22]. Given the importance of memory bandwidth in a system, the memory controller must be fast and robust.

6.2 Chosen Design Solution

Building a memory controller for an FPGA from scratch would be a task too large for one undergraduate. Luckily, Xilinx supplies tools to synthesize memory controllers on their FPGAs. Xilinx's COREgen tools create synthesizable memory controllers to user specifications. For this project, we used the Memory Interface Generator v39.2 (MIG). The tool has a wide range of capabilities for interfacing with DRAM for both Virtex-6 and Spartan-6 FPGAs. We will describe the basic structure and interface of the design created using the MIG, then describe the specific design tested for this project. Much of this description will pertain to the source code structure so that a reader would be capable of navigating the complex design files generated by the MIG.

6.2.1 Xilinx Memory Interface Generator (MIG)

The Xilinx MIG tool generates a pre-engineered controller and physical interface for Virtex-6 FPGA user designs. The design can be created in either Verilog or VHDL. It is important to note that many components of this design are hidden to the user since the controller and many Xilinx primitives are proprietary and should not be edited by the user. The proprietary core is provided through the Xilinx Platform Studio. Upon generation, the MIG creates three directories in the project directory:

- `docs/` - Contains PDF documentation.
- `example_design/` - Contains an example design attached to the memory controller to generate a test load for the controller.
- `user_design/` - Contains an exposed user interface for the user to create their own design to use the memory controller.

Both the user design and the example design use the full memory controller. The example design provides an excellent framework for testing and understanding the module, which can also be synthesized onto an FPGA to check functionality. The user design would be used to create a custom design to send traffic to the controller. Both the user and example designs use the same core elements, with the difference of the additional test module inside the example design. Both designs include:

- A memory controller located in `rtl/controller/`
- A physical interface with the FPGA memory located in `rtl/phy/`
- A top-level user interface located in `rtl/ui_top/`
- A user design constraints file for mapping proper pins and constraints onto the FPGA during synthesis located in `rtl/par/`
- Tools for the user to simulate the design using ISE located in `rtl/sim/`
- An error correcting module capable of detecting and correcting errors located in `rtl/ecc/`

The example design includes an additional `rtl/traffic_gen/` directory, which is the test module for artificially generating traffic to verify the functionality of the memory controller. All of the modules listed above are arranged in a top-level module.

When interfacing with the memory in this module, we have three points of entry. The first would be to write a design to interface with the physical interface, ignoring the memory controller module. This requires a lot of work; we would need to convert a flat address space from chip requests into RAS and CAS to be sent to the DDR3 RAM. We also would need to track and respect refresh times in the DDR3 RAM. Our second option would be to attach our design to the user interface of the MIG controller that wraps the memory controller (responsible for the flat address translation and refresh). This is the

simplest option for our project and was ultimately the chosen design route. Our third option would be to enable an AXI4 interface module to be generated by the MIG. This would create an AXI4 interface to the user design, but because other components in our system do not use this protocol, it would only create another module to translate requests through, reducing bandwidth and increasing complexity. Therefore, we did not choose this route. A block diagram of the user design top-level module can be seen in Figure 23.

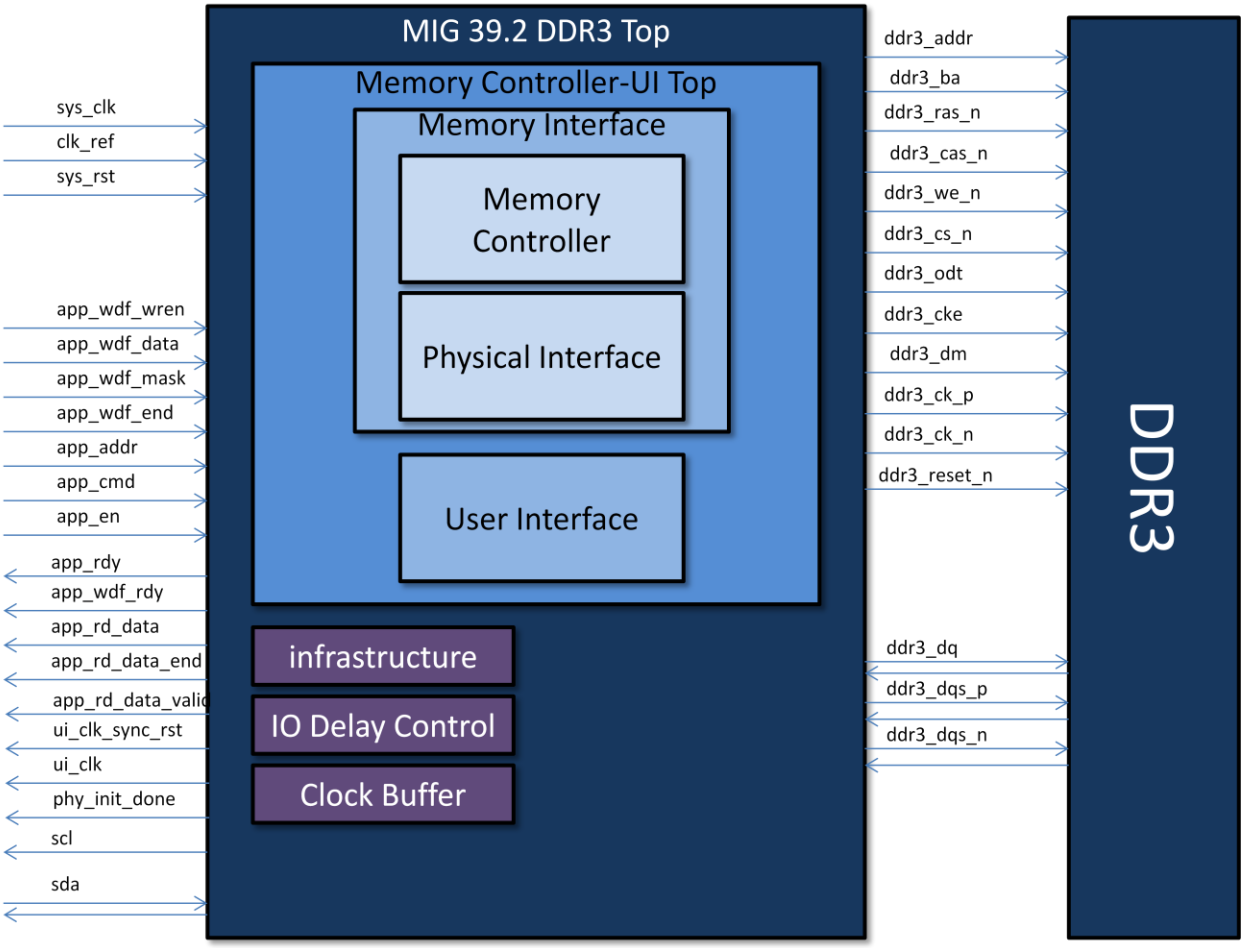


Figure 23: User design top-level module. To the left are the user provided signals for the controller to use.

The Memory Interface block wraps the memory controller and physical layer for the sake of abstraction. The Memory Controller UI Top, Example Top, and MIG 39.2 DDR3 Top do similar jobs in terms of bringing abstraction to the design. The smaller modules contained in the top level module include the following:

- Infrastructure – Helps with clock generation and distribution and reset synchronization on the FPGA
- IO Delay Control interface – Timing Simulation Library Component and FPGA control
- Clock Buffer clocks – Instantiates the input clock buffer, generating differential and single-ended clocks where needed.

These three modules are important for interfacing with the FPGA and also for simulating the design—each conquers challenges with signal integrity and clock consistency in the FPGA design. The Virtex-6 has five kinds of clock lines; the infrastructure module ensures that the clocks have low skew and can support high fan-out and low propagation delay.

The example design depicted in Figure 24 looks slightly different from the user design since the traffic generator is included in the top-level design.

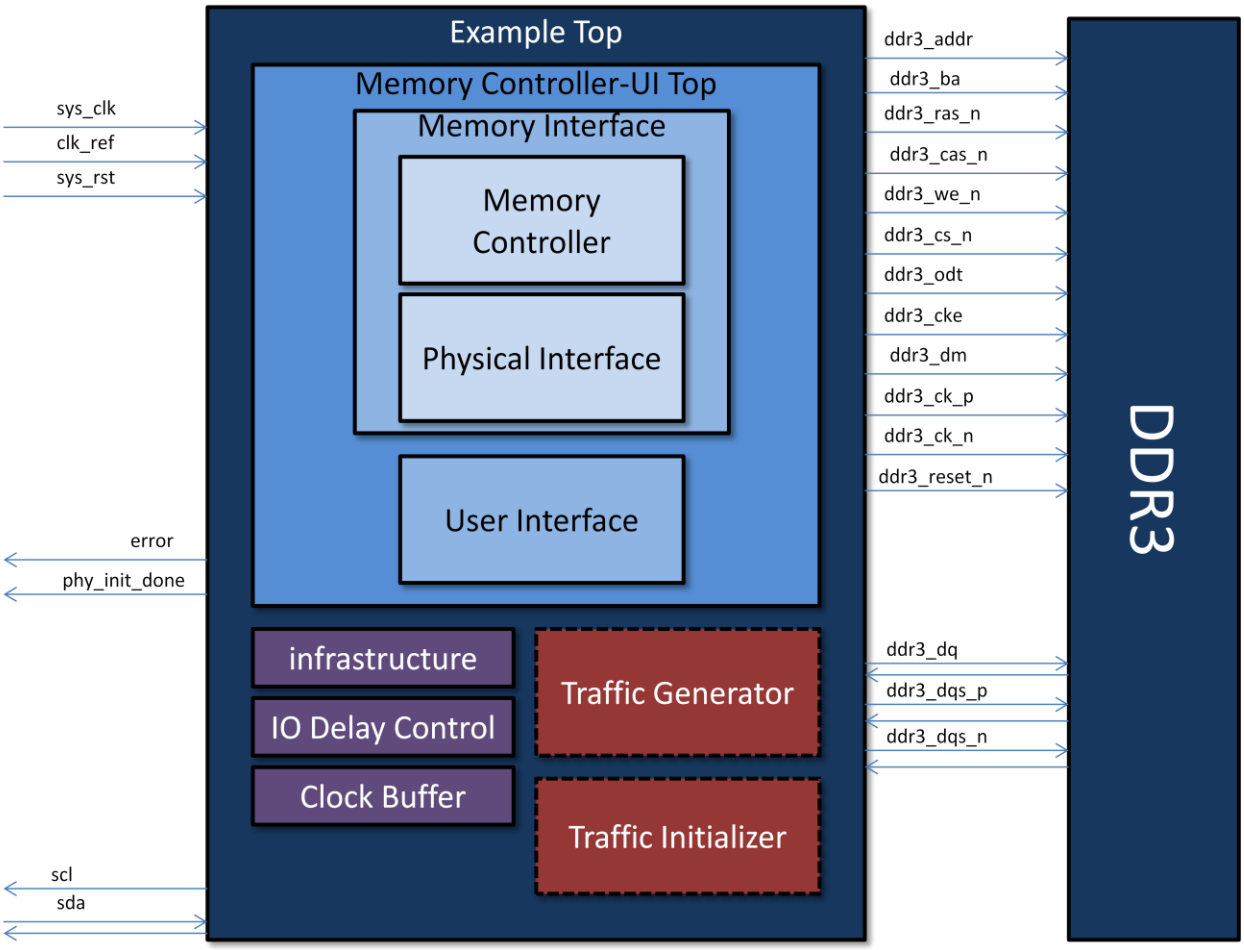


Figure 24: Example module – note that the traffic generator is now attached to the user design outputs and thus takes its place as an example user design.

The Traffic Generator is a very powerful tool for simulation when testing the integrity of the memory controller: the user can configure the design to produce repetitive test patterns, as well as random commands or a series of commands that model real-world traffic. The user can also specify an address range to test (through BEGIN_ADDRESS and END_ADDRESS parameters in the design). This design is fully synthesizable, so it can be tested in simulation and after synthesis.

MIG User Interface

Xilinx provides a reference describing each user signal to the input of its generated IP core [14]. We describe these signals in the following table to clarify which controls the user module has over the MIG user design:

<table>
<tr><td>app_addr [ADDR_WIDTH]</td><td>This input indicates the address for the request currently being submitted to the UI. The UI aggregates all the address fields of the external SDRAM and presents a flat address space to the user.</td></tr>
<tr><td>app_cmd[2:0]</td><td>This input specifies the command for the request currently being submitted to the UI:<table><tr><td>Read</td><td>001</td></tr><tr><td>Write</td><td>000</td></tr></table></td></tr>
<tr><td>app_en</td><td>This input strobes in a request. The user must apply the desired values to app_addr[], app_cmd[2:0], app_sz, and app_hi_pri, and then assert app_en to submit the request to the UI.</td></tr>
<tr><td>app_hi_pri</td><td>This input indicates that the current request is a high priority.</td></tr>
<tr><td>app_sz</td><td>The app_sz signal specifies the burst length. We use a burst length of 8, so this is set to 1.</td></tr>
<tr><td>app_wdf_data [APP_DATA_WIDTH]</td><td>This bus provides the data currently being written to the external memory.</td></tr>
<tr><td>app_wdf_end</td><td>This input indicates that the data on the app_wdf_data[] bus in the current cycle is the last data for the current request.</td></tr>
<tr><td>app_wdf_mask [APP_MASK_WIDTH]</td><td>This bus indicates which bits of app_wdf_data[] are written to the external memory and which bits remain in their current state.</td></tr>
<tr><td>app_wdf_wren</td><td>This input indicates that the data on the app_wdf_data[] bus is valid.</td></tr>
<tr><td>app_rdy</td><td>This output indicates to the user whether the request currently being submitted to the UI is accepted. If the UI does not assert this signal after app_en is asserted, the current request must be retried. The app_rdy output is not asserted if:
- PHY/Memory initialization is not yet completed
- A read is requested and the read buffer is full
- A write is requested and no write buffer pointers are available
- A periodic read is being inserted</td></tr>
<tr><td>app_rd_data [APP_DATA_WIDTH]</td><td>This output contains the data read from the external memory.</td></tr>
<tr><td>app_rd_data_end</td><td>This output indicates that the data on the app_rd_data[] bus in the current cycle is the last data for the current request.</td></tr>
<tr><td>app_rd_data_valid</td><td>This output indicates that the data on the app_rd_data[] bus is valid.</td></tr>
<tr><td>app_wdf_rdy</td><td>This output indicates that the write data FIFO is ready to receive data. Write data is accepted when both app_wdf_rdy and app_wdf_wren are asserted.</td></tr>
<tr><td>rst</td><td>This is the reset input for the UI.</td></tr>
<tr><td>clk</td><td>This is the input clock for the UI. It must be half the frequency of the clock going out to the external SDRAM.</td></tr>
<tr><td>clk_mem</td><td>This is a full-frequency clock provided from the MMCM and should only be used as an input to the OSERDES.</td></tr>
<tr><td>clk_rd_base</td><td>One copy of full-frequency clk_rd_base is routed to the individual capture clock networks for each DQS group; the phase of each of these individual DQS clocks is then adjusted during read leveling by</td></tr>
</table>

	an IODELAY to position the capture clock in the middle of the read data eye. The phase of clk_rd_base relative to clk_mem (used to drive the forwarded clock to the memory) is phase adjusted by the fine-phase shift feature of the MMCM by the phase detector logic to account for ongoing delay variations in the read capture path due to voltage and temperature changes.
`phy_init_done`	The PHY asserts phy_init_done when calibration is finished.

Table 1: Inputs and outputs available to the user design module [14].

Before we describe a typical memory transaction using this interface, we should first describe the specific design created for this project.

6.2.2 Design Implemented

As would be expected, we generated our core for the xc6v1x240t model Virtex-6 with package model ff1156 (speed grade -1). All designs were generated and written in Verilog. This design uses only one controller, though more could be added if we wished to increase the amount of memory per node (this would require additional adjustment to the in-node router). The clocks fed to the design are single-ended, and we do not use an internal reference voltage for the design since the ML605 does not require this. For our design, we chose to use the native user interface rather than the AXI4 interface.

The clock frequency of our design's RAM interface runs at 303.03 MHz. This is the only supported frequency with the higher end DDR3 controller offered by Xilinx. This design requires that the 200 MHz on-board clock be multiplied by the infrastructure module and fed to the interface module. The internals of the controller run off the 200 MHz clock. The controller supports busts of length 8, maximizing bandwidth of the DDR3 memory. The data width connecting to the RAM is 64 bits, and therefore the width of the user interface data is 256 bits wide so we can write 512 bits in a single cycle (256 bits on the rising edge, 256 bits on the falling edge). This is convenient since our system's cache line size is 512 bits. The CAS latency of this design is 5 clock cycles. Since it would be prohibitive to supply all source code for this design, and since the source code is Xilinx generated, the configuration documentation used to create the design can be found in Appendix L for reference.

6.2.3 Typical Read/Write Transactions

For every command sent to the memory controller `app_cmd`, an address must be sent to memory through `app_addr`. Before this address is accepted, `app_cmd`, `app_addr` and `app_en` must be set for one clock cycle. `app_rdy` is asserted after a delay to signify that the command as been accepted. If `app_rdy` is low, the values will not be sent to the controller and the user must wait until `app_rdy` is high. This may happen during the refresh period or if the controller is overloaded with requests. This sequence is shown in the Figure 25.

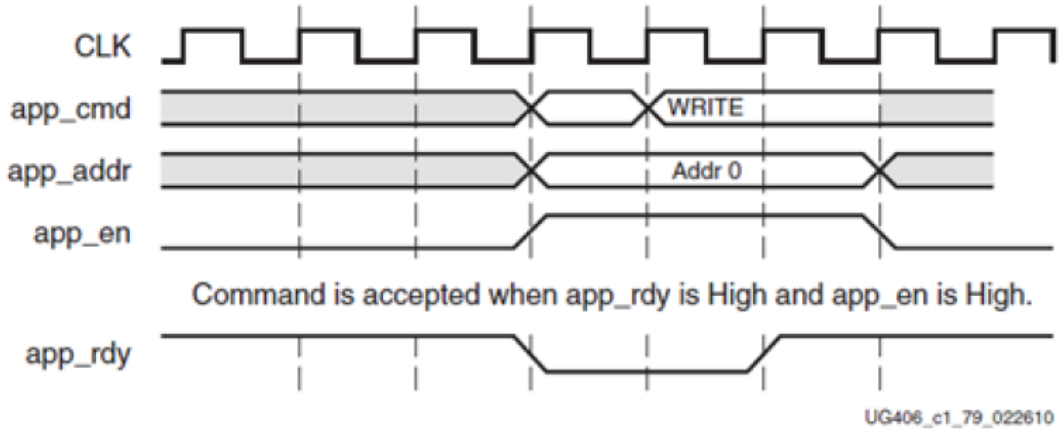


Figure 25: Sending a command and address to the memory controller[14].

In simulation, we can see a matching situation for a read request:

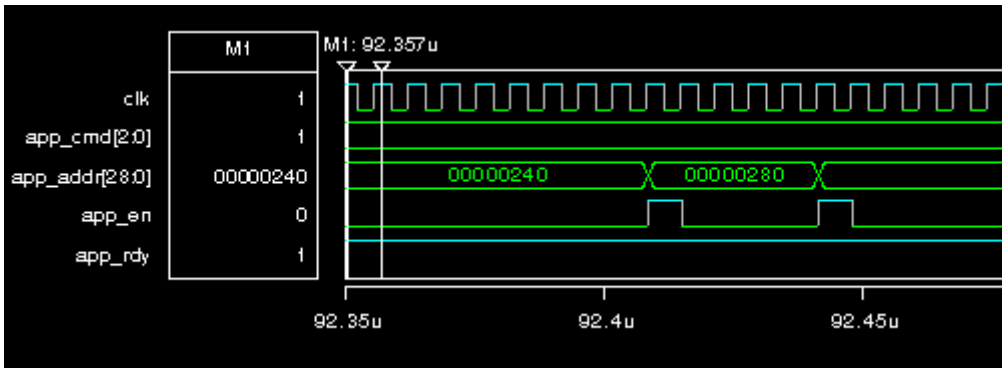


Figure 26: Sending two read requests through the user interface.

A write command takes place in two stages. First, the data to write (and the data mask) are written to a buffer. `app_wdf_wren` should be asserted when data is valid, and `app_wdf_rdy` indicates when data can be accepted by the controller.

app_wdf_end is used during burst transactions indicating that the second of two 256-wide data words has been sent (indicating all 512 bits have been sent). Finally, the app_wdf_mask should be used when indicating which data bits to ignore—each bit in the data mask can be set high to ignore 1 byte in the 512-bit data set. The write command is sent over app_cmd once the data has been accepted by the controller and is accepted in the same fashion as read commands. In Figure 27, we can see three write transactions being sent through the user interface in our simulation. Here we can see that the user design must stall when the memory controller is not ready (~91.4us). However, this stall takes place while the user module is writing data to the wdf buffer, so the length of this write transaction is not affected.

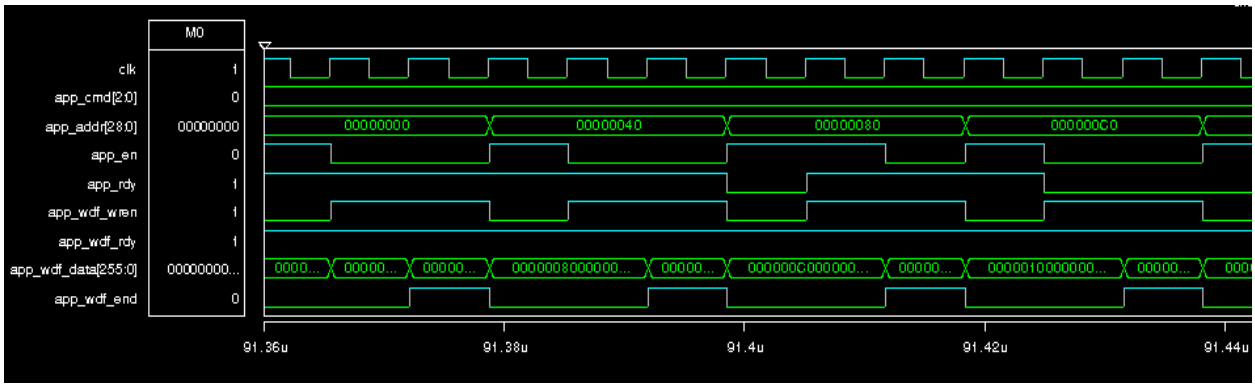


Figure 27: Four write transactions sent through user interface. Each is marked at the end with app_wdf_end going high.

When receiving data, the following protocol is used: app_rd_data_valid indicates that the data on app_rd_data is valid. app_rd_data_end indicates when the second of two 256-bit-wide words is being sent. It is important to note that outgoing read values have been reordered inside the memory controller to appear in the order in which they were requested. It is the user's responsibility to track which requests coordinate with which return data to send the data back to the requester. It is also important to note that the user must read app_rd_data when app_rd_data_valid is high since there is no rdy-val interface here. The controller will not wait for the user module to be ready and will simply drop unread data. Therefore, the controller assumes that if the user interface has requested a read, it has available buffer space to store the

data once it is returned from the controller. We can see this transaction taking place in simulation in Figure 28.

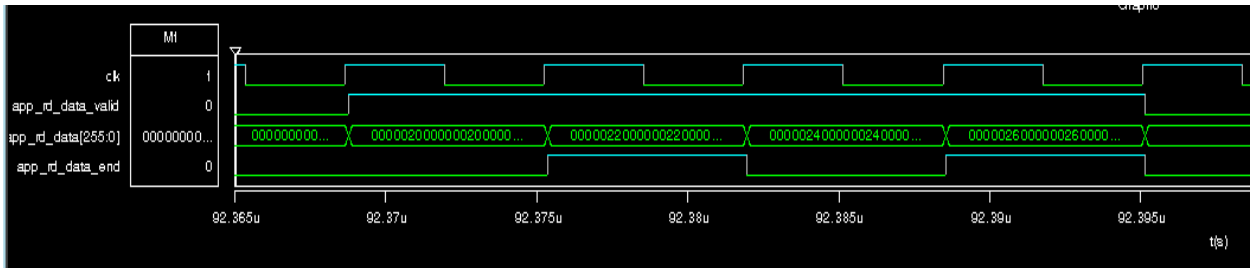


Figure 28: Read data exiting the memory controller.

6.2.4 Testing Methodology

Simulation Flow

Though Xilinx sets up all its tools to run simulations though ModelSim and Xilinx ISE simulation tools, our larger project uses vcs to compile our source code to simulate using simv. However, to make this process possible, we need to link the standard cells which Xilinx uses in its designs. This can be accomplished by adding the following flags to our vcs command:

```
-y $XILINX/verilog/src/unisims
-y $XILINX/verilog/src/xilinxcorelib
+incdir+$XILINX/verilog/src
+libext+.v $XILINX/verilog/src/glbl.v
-Mupdate -R -sverilog
```

Where `$XILINX` is the path of our Xilinx ISE library installation (typically `/opt/export/xilinx/14.2/ISE_DS/ISE` for version 14.2). This allows the linking of Xilinx intellectual property into our designs, including the Xilinx Core we obtained from the MIG. This also includes Xilinx models of DRAM. Once the designs were simulated, we could use cscope to view waveforms of the files (as seen above). This same simulation flow is required to simulate the asynchronous FIFOs used in our FPGA interface and chip interface (Sections IV and V)

6.2.5 Results and Analysis

Simulation Methodology

To check the basic functionality of our controller, we used the traffic generator provided in the example design. Though the traffic generator is powerful, it cannot

provide all the functionality we desire. An important metric we wanted to measure is the latency of the memory controller in accepting and returning data vs. the number of requests it is receiving. To test this, we built our own test bench based on a Xilinx interface module (the source code is provided in Appendix M). This test bench writes random values to random addresses and reads data from these addresses to check that the output is correct. The test bench also alters how often these requests are sent to the controller. We measure the system load % based on how often we allow the enable signal of the client to be set high. We measure latency for read commands based on the time between when the command is accepted and when the data is returned. We measure the delay on write commands based on the time between when the 512-bit request data has been fully accepted to the controller's buffer and the time when the command is executed by the controller. For each test, 1,024 values were written and read from RAM to determine the average latency. To simulate worst-case command patterns, many read and write instructions were issued one after the other.

System Bandwidth

The theoretical max bandwidth for the system would occur in the case that the controller is accepting 256 bits every clock cycle. Since the memory controller interface runs on the 100 MHz FPGA clock, this implies that the controller can accept write data or return read data at a rate of 25.6 Gbit/s (3.2 GB/s). However, a barrier to this is whether the data buffer in the memory controller fills up before write commands can be issued—the memory controller only holds enough data for 16 instructions to be in flight at the same time and will stall if this limit is reached. We can see in our results below when this happens.

Latency vs. Throughput

In our experiments, we swept the system load from 10% up to 100% to determine what kind of latency guarantees the system could provide. The results of the sweep are shown in Figure 29:

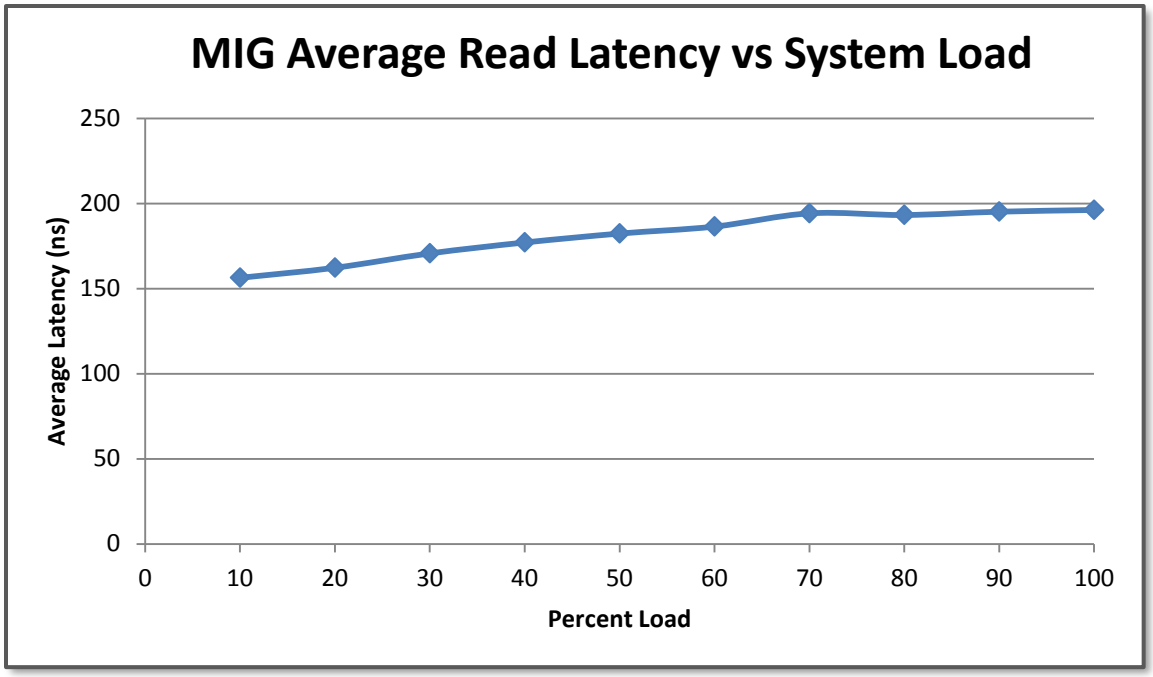


Figure 29: Average read latency vs. system load using a custom test bench attached to the user interface.

As seen above, the average read latency maxes out at ~196 nanoseconds (with an interface clock period of 10 ns). The highest observed latency was 630 nanoseconds (63 clock cycles), while the lowest was under 10% load at 80 ns (8 clock cycles).

The average write latency shows a very different pattern. Again, this latency was measured as the time between when the 512 bits to be written were accepted by the controller and the time when the command was accepted to the controller. The results are shown below in Figure 30:

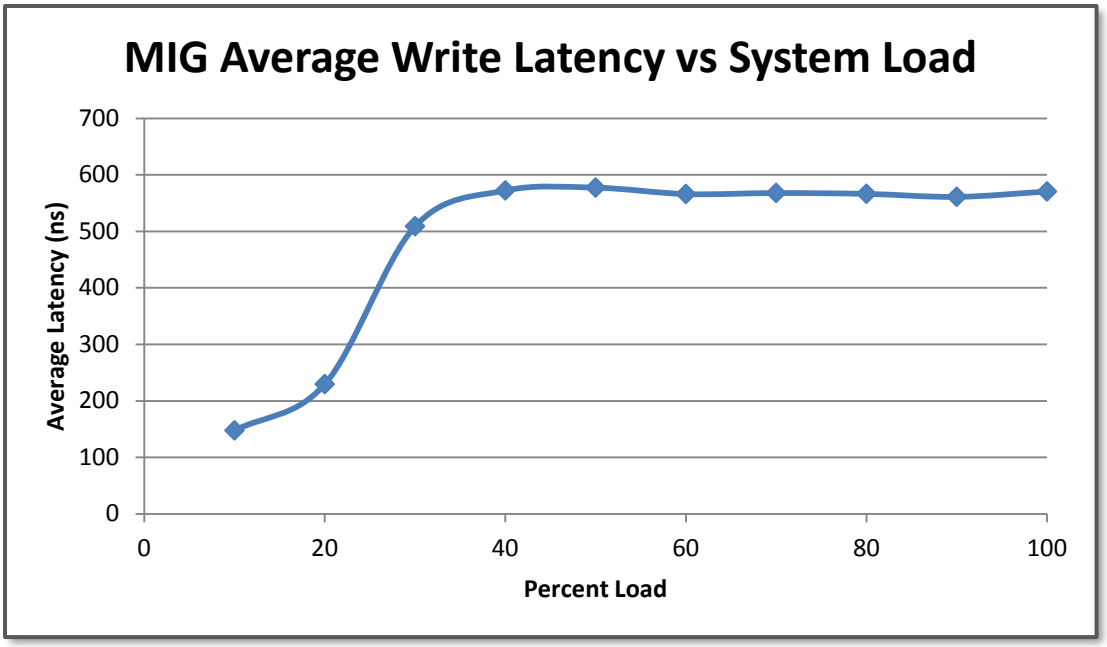


Figure 30: Average write latency vs. system load using a custom test bench attached to the user interface.

The average latency evens out at about 570 ns. The maximum latency observed was 10,190ns while the minimum observed was 10ns (the command was accepted the cycle after data was present). As we can see, there is a jump in latency at around 35% system load. Understanding the results of these tests also provides valuable insight to understanding the internals of the memory controller. This jump occurs because write data is accepted to the data FIFO faster than commands are accepted to the controller, causing write data to wait in queue until the command interface is ready. In our system, this buffer does not become a bottleneck since packets cannot be switched through the network fast enough to fill this buffer.

6.3 Packet Translation

The packets used to transmit and request data used in this system will not match the format that the memory controller will accept. As a result, packets must be translated for the memory controller. The following section discusses the packet format our chip will use to communicate with main memory, followed by a discussion of the module used to translate these packets for use by the memory controller.

6.3.1 Request/Response Format

Request packets in our system use a 3-flit header, which is shown in Figure 31. The reader should be reminded that this packet format is used globally for memory requests as well as I/O and cache coherence—therefore, some fields are not required when making memory requests.

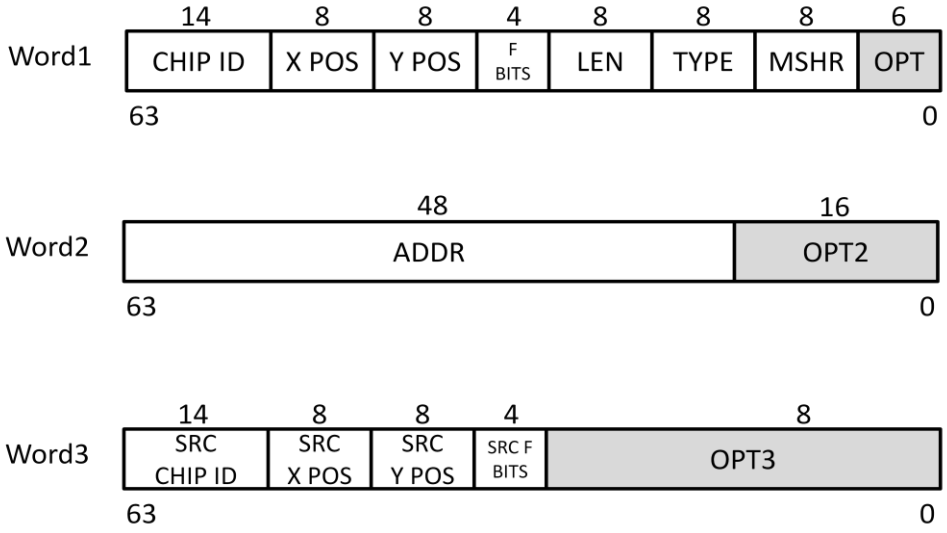


Figure 31: Three-flit packet header for requests to main memory – all OPT fields are reserved for future adjustment to the system.

The Chip ID and X and Y position are all used for on-chip routing. The F bits are option bits that change depending on the command. The Len (length) field specifies how many flits follow the first, including data. The maximum size for a packet sent to memory is 10 flits long; this would occur during a write request with a full cache line in tow. The "type" field denotes what kind of memory request is sent to the controller—the type can include store, read, non-cacheable store, and non-cacheable read. However, non-cacheable commands can only be sent to I/O in our system, so the memory controller does not have to handle these commands. The miss status handling register (MSHR) bits hold the MSHR information for the requester to use when processing acknowledgments from main memory. Bits 45-23 of the second packet are used for addressing into the 512 MB in-node memory. The bottom 6 bits of the address are used for byte addressing, primarily used in I/O communication. The top 19 bits of the address are used by the in-node routers to determine which node can complete the memory request.

An observant reader may see that the address space allows for more than 512 MB to be mapped per memory controller and more than one memory controller to be mapped per node for the address space to be filled. This gives the system flexibility to change node configurations to be more powerful in future designs.

Finally, the third word contains information about the requester, which will be used in responses from the memory controller. In our system, both reads and writes require acknowledgments to be sent back to requesters. Both share the following 1-flit header:

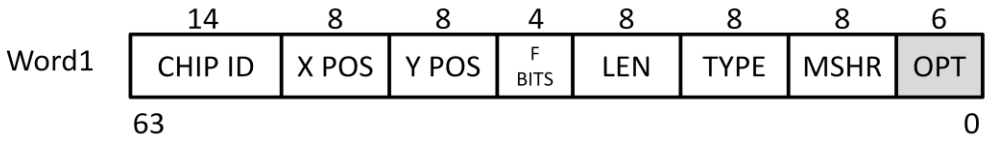


Figure 32: Acknowledge message header.

The length of a read response will be 8 (0 additional header flits, 8 data flits), while the length of a write response will be 0. Requests to the memory controller arrive on a different network to prevent network deadlock, which is discussed in greater depth in Section VII.

6.3.2 Packet Translation Module

This section describes the design of the module used to gather memory requests, translate them for servicing, and send responses to requesters. A block diagram of this module is depicted in Figure 33.

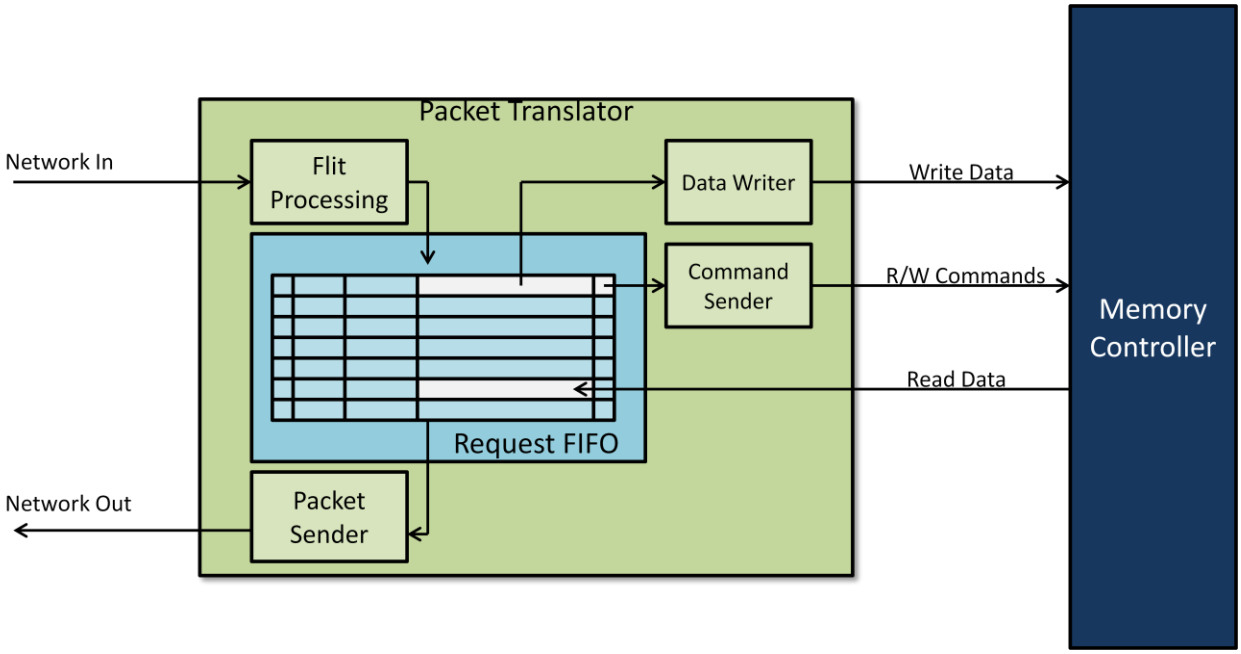


Figure 33: Top level diagram of the packet translation module. Central blue module is a FIFO with adjustable size. The four surrounding modules read and write to the FIFO to service requests to the memory controller.

As we can see in Figure 33, the largest component in the module is a FIFO memory designed to buffer request packets while their commands are in flight in the memory controller. Five modules read and write to the buffer in the five different stages of a request:

Filling State: In this stage, the flit-processing module accepts flits and fills in a request FIFO entry. The head is parsed to determine the command, source, and destination of the packet. The module then reads in the subsequent flits to fill in the address of the packet, and if the command is a write command, the data section of the FIFO buffer entry is filled.

Waiting_WDF State: This state is only activated for write commands. The Data Writer module sends data to the data buffer inside the memory controller. Once this has been done, the buffer entry can advance to the Waiting_CMD state. This must be done because the write command cannot be issued before the data has been provided to the memory controller.

Waiting_CMD State: In this state, the Command Sender module sends the packet's command and the address at which that command is to be performed to the memory controller. At this point, write commands advance to the Ready state since their data will be written by the controller and acknowledge signals can be sent back. Read commands advance to the Waiting_DATA State as they wait for the controller to return the read data requested.

Waiting_DATA State: In this stage, read command packets wait for the Data Accepter to return requested data from the memory controller. Once the data has arrived, they advance to the Ready State as they can be sent back to their requester.

Ready State: When in this state, the Packet Sender can create a set of flits out of the request FIFO entry and send them over the network. Once this step is finished, buffer entries enter the Invalid State. The packet sending module also performs adjustments to the head of a packet to route back to the requester.

Invalid State: A buffer entry in this state is not in use and can be filled by the Flit Processing unit if needed.

The source code for this module is provided in Appendix N. The module is fully parameterizable, allowing us to change buffer sizes or change how the module functions depending on how our packet format may change in the future.

Section VII: Network Topology

The first prototype we will build will contain only two nodes. Routing on-chip is determined by the X and Y positions or the chip-ID in the packet header. In-node routers, located on the host board, determine which node to route packets to based on the Chip ID field, or the high-order bits of the memory address. The in-node routers have to read the first flit of the header to determine where to route the message; the on-chip router is responsible for determining whether to send requests to the in-node memory controller, chip, I/O, or off chip. It is important to note that because the system is scalable and reconfigurable the host boards will have to be aware of their location in the system; during the boot sequence, host boards will discover other nodes in the system to determine routing paths and their identity in the system—delivering this information to the many-core chip to determine how Chip IDs are routed. It is important that however the routers are connected in the system that our routing policy is such that we can avoid network deadlock and message-dependent deadlock.

7.1 Deadlock Avoidance

7.1.1 Message-Dependent Deadlock

A message-dependent deadlock could arise when a set of network resources is used by one message that is waiting for another set of network resources held by a second message that is waiting on the first message [23]. This type of scenario is particularly common in request-response systems or systems where one message can generate more messages.

As an example, suppose Cache A needs to invalidate a line in Cache B, but both can only handle a limited number of requests at a time. If Cache A has reached the limit on outstanding requests it can have at one time, it may stop accepting requests that would require it to generate more requests. While these unaccepted requests line up outside of Cache A, the acknowledge messages from Cache B (Message K) will return and take its place at the back of the line. This causes a message dependency; the first unaccepted request (Message U) is waiting for Message K to arrive at Cache A so that Cache A can accept Message U. In the meantime, Message K is waiting for Message U to enter Cache A so Message K can reach the front of the line.

This kind of deadlock is message dependent because it is independent of the network topology or routing mechanism and could still occur on a deadlock-free network. To strictly avoid situations like these, virtual channels or additional on-chip networks can be added to the system. To solve the above situation, we can add a network (and ports on the caches) for acknowledgments to eliminate any message dependencies. Since Message K no longer waits in the same line as Message U (it does not use the same network resources), the dependency is eliminated.

The Princeton Parallel Processor uses three such networks to fully prevent message dependent deadlock, since the longest possible message chain within our message dependency graph is three messages long[23]. To prevent message dependent deadlock, any message that produces new traffic that could cause a cyclic dependency instead creates new traffic on a different network, guaranteeing that the new message uses a disjoint set of network resources. This is why our memory controller accepts requests on one network and sends responses and acknowledges on another network. However, we can only guarantee that the messages in our network will not deadlock if the routing algorithms used to direct messages are also deadlock-free.

7.1.2 Network Deadlock

To illustrate a deadlock-prone network, imagine a set of routers connected in a 2-dimensional mesh topology. Messages can be routed in the positive or negative X and Y direction by any router until they meet their destination. Each router has a bi-directional link in each direction, so messages traveling in X to and from routers do not conflict. We also require each message to travel the shortest required distance to reach its destination (though, as one can see, there are multiple shortest paths). Finally, each router in this network uses a wormhole routing technique: that is, the first flit can be passed on to the next router before the last flit in the packet has arrived. This saves us buffer space and time since the routers do not need to store the entire message before routing it. Figure 34 illustrates the network structure for such a network.

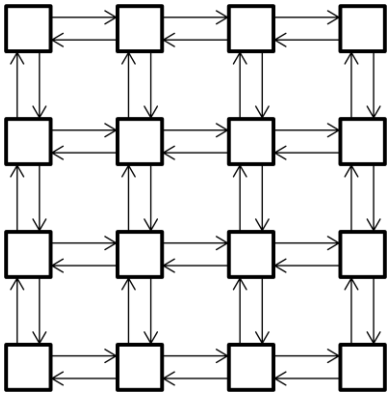


Figure 34: 2-D mesh network topology. Each square is a node in the network capable of worm-hole routing. Each arrow represents a directional link in the mesh.

However, we can show very quickly that there could be a deadlock in this network with the example in Figure 35 that contains four messages, each of which take the shortest route to their destination.

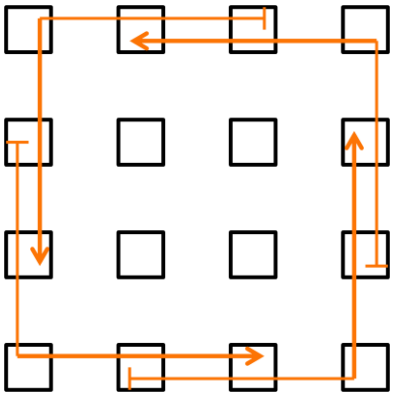


Figure 35: A simple illustration of a deadlock in a 2-D mesh network using shortest-path, worm-hole routing.

If each of these four messages is sent at the same time and if the routers cannot buffer the entire message packet, flits will be left at points in the network waiting to be accepted. However, these flits occupy network resources that other messages need to make progress toward their destination. This example shows how this kind of resource holding could lead to a cyclic dependency where none of the four messages can make progress since they are waiting on resources held by another message. Note that this kind of dependency is independent of message type—it is a fault in the routing policy. Since

it is sometimes hard to create an example of deadlock, we can view this problem from another perspective by looking at the dependency graph of the network, like the one shown in Figure 36.

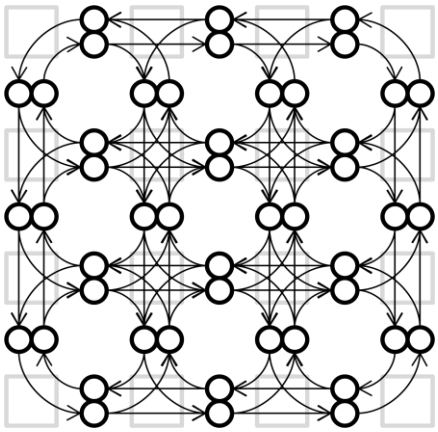


Figure 36: An illustration of the link-dependency graph in a 2-D mesh network using shortest-path, worm-hole routing. Each circle in the picture is a link between two routers, each represented by the squares in the figure.

Here, each circle represents a link in the network, and the arrows show which routing paths can be taken by messages to reach their destinations in the graph. We can see that there are many cycles present in this dependency graph, proving that network deadlock could occur.

Since it is not always convenient to draw a dependency graph, especially for complex networks in greater than three dimensions, we can also consider this problem from the perspective of the "Turn Model" used in adaptive routing in k-ary n-cubes [24]. The model defines a "turn" as when a message switches from routing in one direction to another. The network we have shown as an example allows eight different turns, pictured on the left of Figure 37 (ignoring 180 degree turns, which are impossible in a network routing by shortest path). In combination, these turns can lead to a number of cyclic dependencies illustrated on the right of Figure 37.

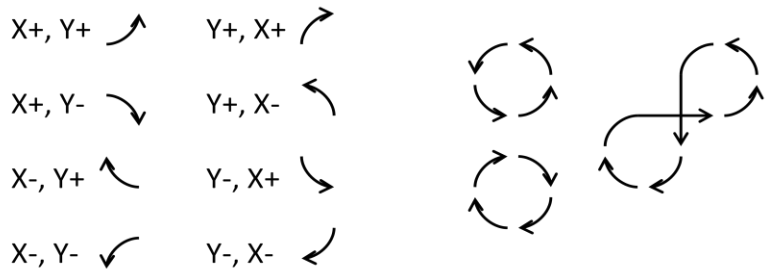


Figure 37: An illustration of which turns were allowed in the network proposed in Figure 34 and Figure 36.

The turn model shows us that by eliminating enough turns, we can prevent deadlock, while still allowing messages to reach any destination in the network. For example, if we impose a Dimension Order Routing restriction on our network, that is, we can only route in X then Y, we eliminate all possible cycles [25]. Figure 38 illustrates the turns allowed under this routing policy.

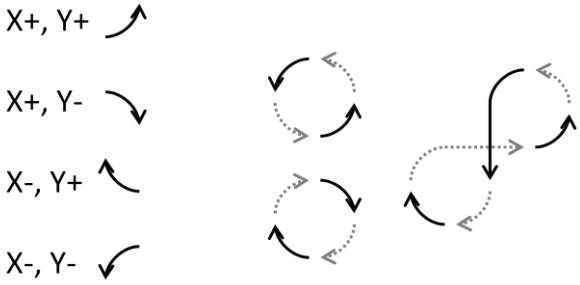


Figure 38: An illustration of which turns were allowed in a network that uses X >> Y dimension-order routing.

The cyclic dependency elimination can also be observed in our updated dependency graph shown in Figure 39.

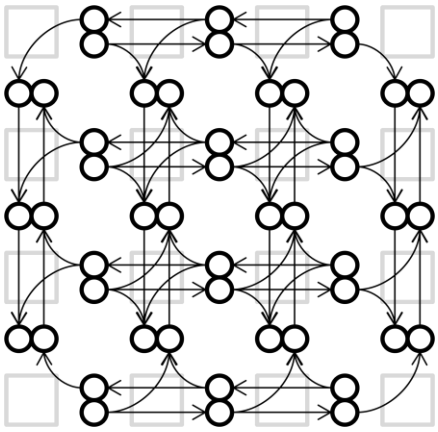


Figure 39: An illustration of the link-dependency graph in a 2-D mesh network using dimension-order, worm-hole routing. Each circle in the picture is a link between two routing nodes represented by the squares in the figure.

This same Dimension Order Routing algorithm can be applied to any number of dimensions and is provably deadlock free [25]. We use X >> Y (X before Y) dimension ordered routing on all of our chips to prevent network deadlock. However, when connecting these networks, there are complications.

7.2 Challenges of Hierarchical Networks

When connecting the deadlock-free networks on our chip into a hierarchical network, we are not guaranteed to create a globally deadlock-free network [26]. In our system, we route messages off chip through a single node to a network connecting a linear array of host boards containing one chip each. This means that our Z-axis does not complicate our network the same way that a Z dimension in a traditional network would. That is, the Z axis in our network must be routed to and away from using the X and Y dimensions. Figure 40 illustrates an example of inter-chip messages passing in a hierarchical network similar to this.

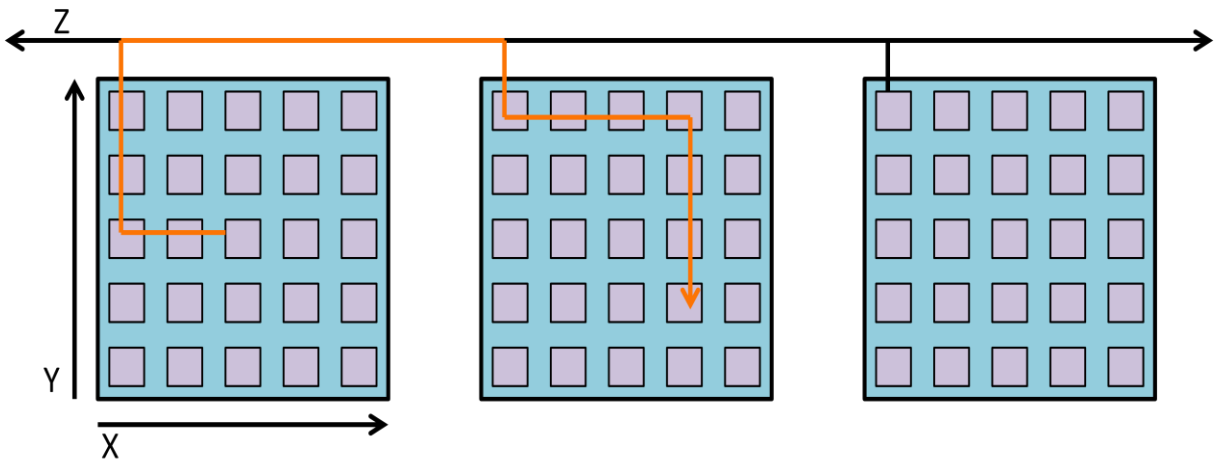


Figure 40: A set of 2-D meshes using dimension-order routing connected in a hierarchical network. The orange arrow represents the path of a message through this network that violates absolute X >> Y >> Z ordering.

This routing pattern violates dimension ordered routing in that messages can route in X, then Y, then Z, then X and then Y again. To overcome this, we could impose a provably deadlock-free "direction ordered algorithm"—that is, force not an absolute dimension order (X >> Y) but an absolute direction order (X+ >> Y+ >> X- >> Y-)[25]. In this case, since the boundary node is in the top left corner, we could impose the following order: X- >> Y+ >> Z+ >> Z- >> X+ >> Y-. All packets could still reach all points in the network following this ordering, which by direction-ordered routing is provably deadlock-free. In the turn model we can prove that we still avoid cyclic dependencies on our chips.

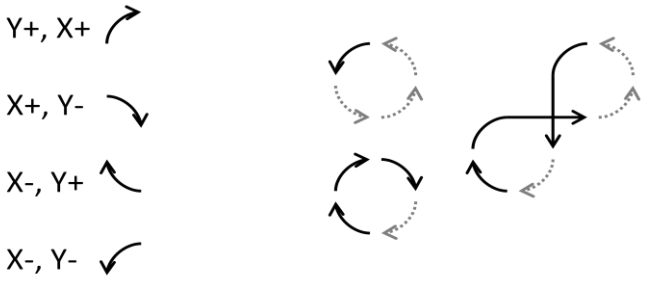


Figure 41: An illustration of which turns were allowed in a network that uses X+ >> Y+ >> X- >> Y- direction-order routing.

However, adopting this routing algorithm would require significant adjustment to the OpenSPARC routers in our chip. Happily, we can show that this is not necessary and that the routing algorithm used in Figure 40 is actually sufficient for preventing deadlock. Although the network violates the absolute ordering of X >> Y >> Z, we can still show that the turns we have added to our routing algorithm do not create cyclic dependencies in our network.

In our routing ordering: X → Y → Z → X → Y, the constraints placed on the Xs and Ys are special. The first Y and second X are always on the boundary of the mesh and are always X+ and Y+. Therefore, in the cases where these directions appear, the first X and second Y must be X- and Y-. These routing restrictions prohibit the following turns in our network:

Y-, X+	Y+, X-
Y-, X-	Z+, X-
Y-, Z+	Z+, Y+
Y-, Z-	Z-, X-
X+, Z+	Z-, Y-
X+, Z-	

Figure 42: The set of turns not allowed in the hierarchical network shown in Figure 40. This can be expanded to higher dimensions and also applied to chips with exit points in other corners of the 2-D meshes.

As guided by the Turn Model, we have eliminated cycles in every cross-dimensional plane (X-Y, Y-Z, X-Z). We have prevented 9 turns, provably disallowing deadlock. Alternatively, a quick analysis of the dependency graph of our network shown in Figure 43 reveals that there are no cyclic dependencies in this network (with emphasis added on the Z dimension):

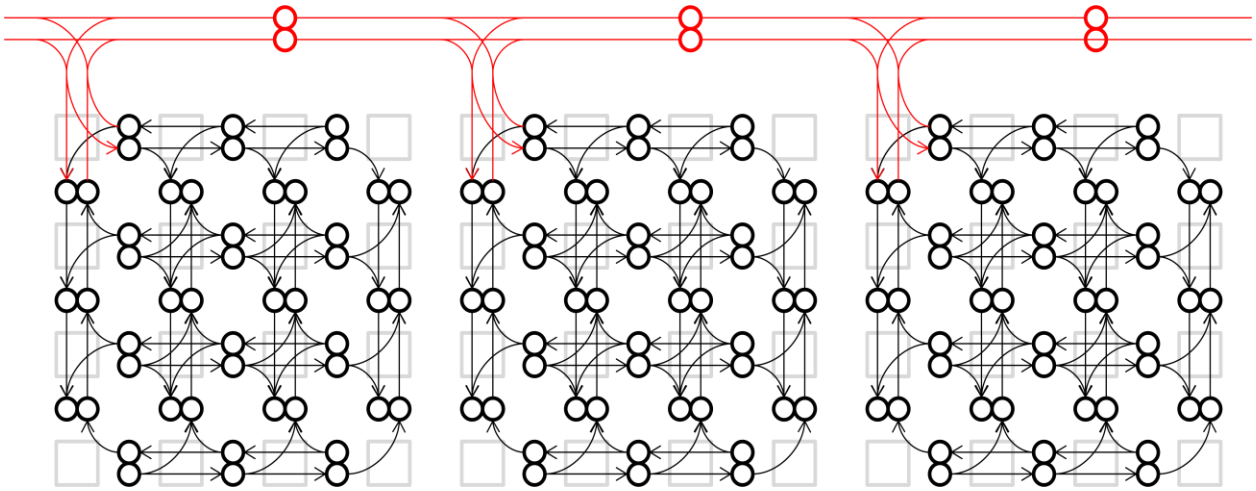


Figure 43: An illustration the dependency graph of the network shown in Figure 40. The red nodes and connections represent the Z axis connecting the 2-D meshes to one another.

Using GraphViz's graph visualization tools, we can arrange both the 2-D dimension-ordered dependency graph shown in Figure 39 and the hierarchical dependency graph shown in Figure 43 as directed acyclic graphs. These are very large graphs; therefore, they and the code used to generate them are included in Appendix O, P and Q for the interested reader.

This evidence shows that the network topology chosen for this project is indeed a deadlock free network and will be capable of supporting requests from the many-core chip.

Section VIII: Synthesis Flow

8.1 Xflow

The Xilinx development environment includes a set of command line tools that can be used for performing development environment functions. One desired function is the ability to program an FPGA based on a set of RTL (Verilog or VHDL) designs. Xflow is a Xilinx-provided tool that scripts together a set of Xilinx tools to create a full design flow for a variety of purposes. Here we describe how Xflow can be used to create the necessary files to program an FPGA from a set of RTL files. There are many other uses for Xflow that we do not include here but can be found in the Xilinx command line tools user guide [27].

Figure 44 illustrates the inputs and possible outputs to the Xflow tool. The tool takes in design files, such as Verilog or VHDL, and FLW (flow) files that specify which programs to run in the flow given a command line option. The OPT (option) files specify the command line options for those files and where to place outputs and logs. Trigger files are other files used by programs Xflow calls.

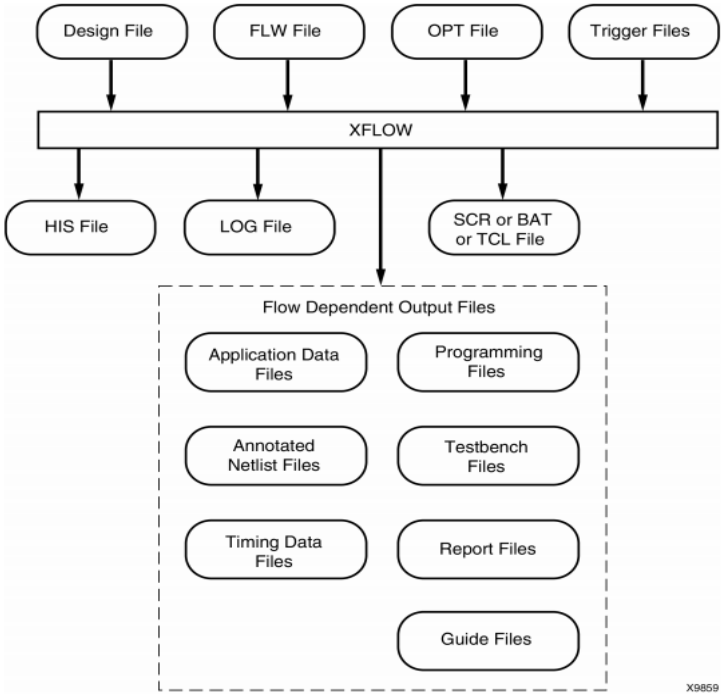


Figure 44: Flowchart illustrating the possible inputs and outputs of the Xilinx Xflow command line tool[27].

Each flow file is associated with a "flow" command line argument and contains a set of programs that are used to complete that flow. The three flows we use are "Synthesis," "Implementation," and "Configure." These three phases and the programs that make up their flow are contained in Figure 45. This flow can be quite complicated; be sure to use the figure as an anchor for understanding.

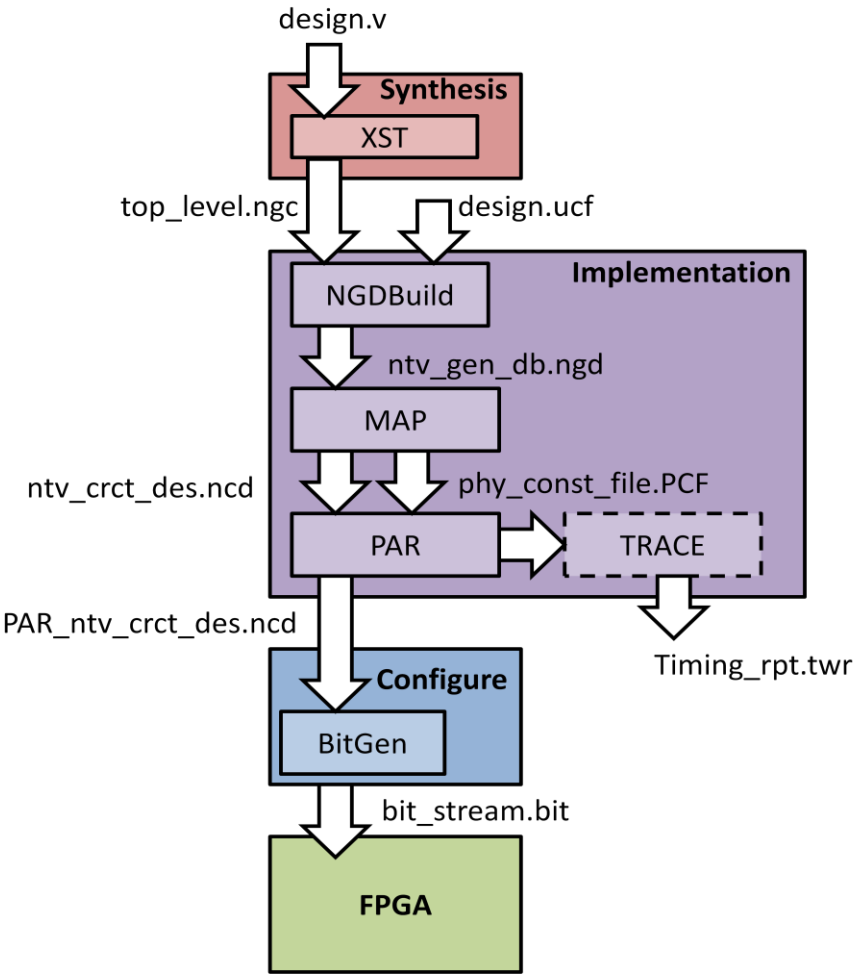


Figure 45: Synthesis flow containing the Synthesis, Implementation, and Configure flows. In each is shown the programs making up that flow and the files accepted and passed by each program.

For each of the following phases, we must specify the device we are using to ensure correct synthesis for the Virtex-6 FPGA we are targeting (xc6vlx240t).

Synthesis takes in RTL design files (in our case, Verilog) and uses Xilinx Synthesis Technology Software (XST) to create a Native Generic Database (NGC) binary file that can be used as a top-level design file. XST performs the step of combining our RTL design with Xilinx's proprietary cores, such as the one obtained from MIG or the asynchronous FIFO generating tool, and creates a single format that can be read by other Xilinx tools.

The implementation phase runs four programs: NGDBuild, MAP, PAR, and TRACE. First, the NGC file is fed to NGDBuild inside the implementation phase, along with the user design constraints (UCF). The user design constraints include which pins to map I/O to and what standard to use on those connections. Each signal mapping is structured in the following way: the keyword "NET" declares a signal mapping, followed by the name of the design signal (input or output) that will be mapped onto the FPGA. The LOC variable declares which pin of the FPGA the signal will be mapped to; this is specific to each FPGA/development board pair and should be looked up. Finally, the I/O Standard variable provides the voltage standard to be used to ensure the safe and correct transfer of signals off the FPGA (buffers may need to be put in place by the Xflow tool to guarantee that these standards are met).

```
NET "sw_in<4>"   LOC = "D22" | IOSTANDARD = LVTTL;
```

Figure 46: An example user design constraint for a dip-switch on the ML605 board—a full example set of code can be found in Appendix R of a simple mapping connecting LEDs and switches and HPC pins on the development board.

In addition to pin assignments, the ucf file also declares other constraints such as operating frequency.

NGDbuild outputs a Native Generic Database (NGD), which is a binary file containing the logical description of the design in terms of original components and the primitives to which the design has been reduced.

The NGD file is fed to MAP, which will map the design onto the target FPGA. The MAP program uses a Macro library file (NMC) containing the definitions of physical macros the NGD file contains. The MAP program then outputs a Native Circuit

Description (NCD) file, which is a physical description of the design in terms of the components on the target device (in our case, the Virtex-6). MAP also outputs a Physical Constraints File (PCF) used for placement and routing. In the process, the MAP program performs a Design Rule Check (DRC) on the input design and the physical output design.

The Place and Route program (PAR) accepts the mapped NCD file as an input and (as the name implies) places and routes the design (i.e., places components in appropriate logic and routes signals using appropriate connections). The placements are made based on timing constraints. PAR outputs another NCD file, which is a placed and routed NCD file capable of being read by BitGen.

The final phase of implementation runs the Timing Reporter and Circuit Evaluator (TRACE), which provides static timing analysis of the FPGA design based in input timing constraints from the ucf file. IT accepts the NCD file generated by PAR and the PCF file generated by MAP to determine if the design fits timing constraints. This step outputs a default timing report (TWR). This step is optional, but recommended.

Finally, the configure phase uses BitGen to accept the NCD file and generate a bit stream to program the FPGA. The NCD file contains information about the internal logic and interconnections of the FPGA that are used by the given design. This bit stream is then sent to the FPGA, which implements the design using its resources and can then be run. More information about all of these tools and the syntax to use them can be found in the Xilinx Command Line Tools User Guide[27].

8.2 System Ace

An additional step of our design flow allows an FPGA to be loaded using a compact flash device rather than loading the FPGA with a bit file after every boot. The first step of this process involves creating a System Ace file structure using Xilinx's iMPACT tools. This creates a bootable version of the bit file for the FPGA to load and use. The System Ace file structure allows for up to 8 designs to be loaded onto a compact flash device. Once the designs are loaded, the Ace System on the FPGA can be set with three address switches pictured in Figure 47.

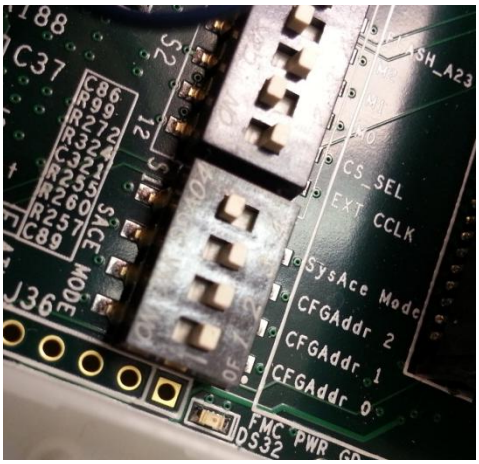


Figure 47: The three address switches and activation switch used to select a design on the compact flash to boot onto the FPGA.

These switches can choose between the appropriate design we want to use to boot, allowing for multiple boot options and quick iteration checks. These features may be useful in future development.

Section IX: Future Work

The goals for this two-semester project have been completed; however, there are further steps that must be taken before this system can be used to host the Princeton Parallel Processor. Below, we will describe a test module that can be used to test all components detailed in this project together. We then describe the steps and challenges to creating that test module and using it on an FPGA. Finally, we conclude with a number of further steps that could be done to expand or make the system more robust.

9.1 Final Testing Setup

Given the two ML605 boards currently in our possession, we will be capable of creating a 2-node system with two 40-core chips, and two memory controllers. This setup would be capable of testing all cache coherence protocols we want to experiment with on a small scale. A diagram of this test system is shown in Figure 48.

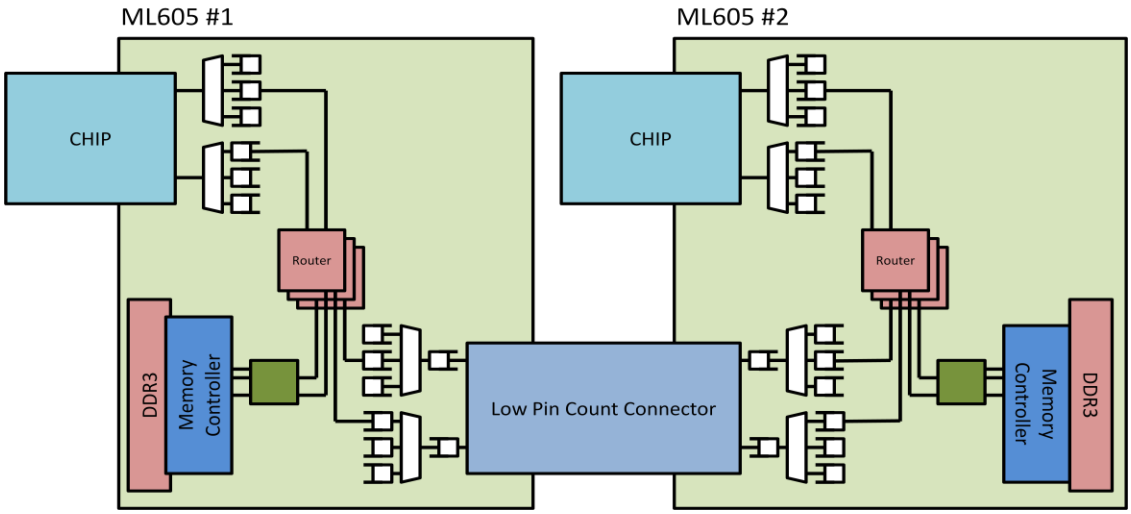


Figure 48: An illustration of the final 2-node test system. Two chips are connected to the FPGA over the interface described in Section IV. Routers determine which memory controller to send requests to as detailed in Section VII. An LPC connector joins the two ML605 boards as described in Section V, and each node contains one memory controller integrated to the network by a translation module as described in Section VI.

Later, with the addition of two more FPGAs on the way, we will be able to expand this system to a four node system. However, since the many-core chips will not return from fabrication for several months, we cannot yet test the system in the way

shown above. Instead, we can instantiate the traffic generator described in Section VI to simulate a chip sending real traffic to the system. This setup will test all components detailed in this report and is shown in Figure 49.

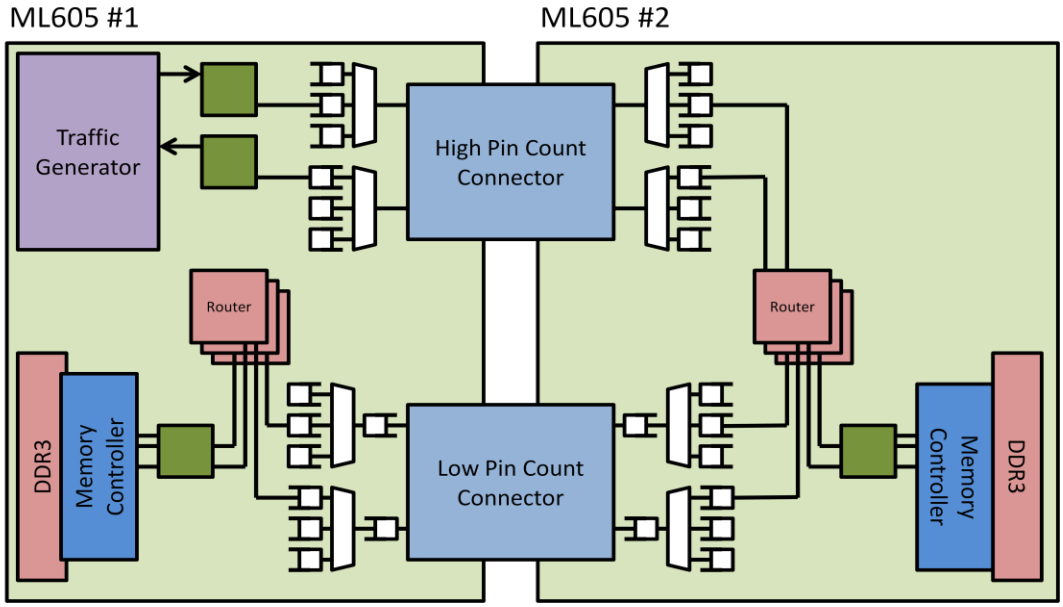


Figure 49: An illustration of the final 2-node test system with the chips replaced with a Xilinx Traffic Generator to simulate traffic being sent to the memory controller by a test system.

The only modules that must be written to create this system are the router (a fairly simple module that we only lack system specifications to build) and the two translation blocks (shown in green) to interface the Xilinx traffic generator with our network.

The next step in testing will be synthesizing these tests onto the ML605 host boards. Though components of this system were each tested in simulation, the design must still be tested on the ML605 development kits. Testing in simulation and testing on an FPGA hold very different challenges and debugging flows. The synthesizable test mentioned above will be setup on 2 node system pictured in Figure 50.

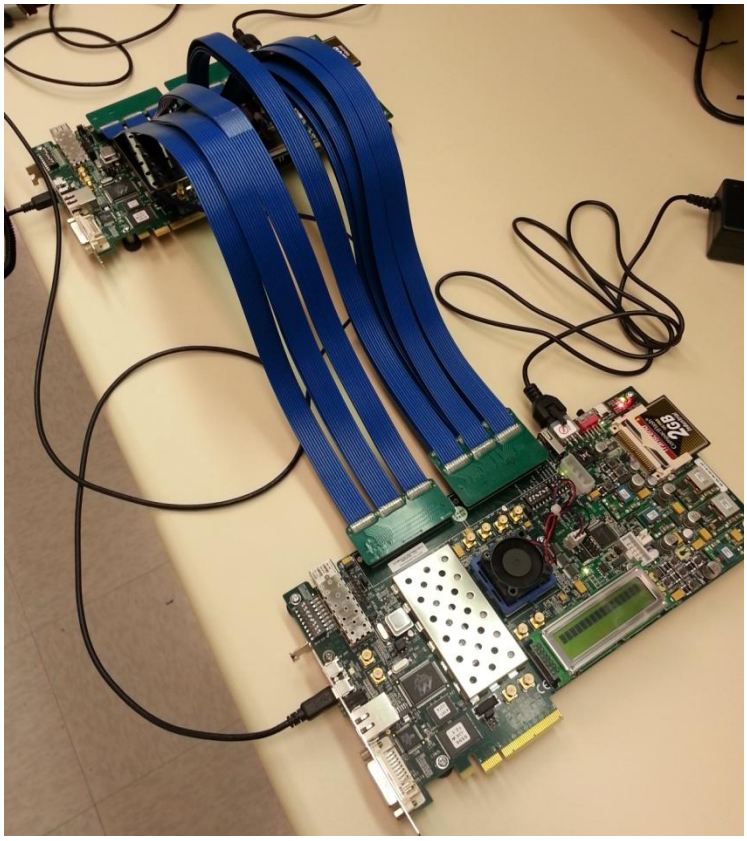


Figure 50: A picture of the final testing setup. The two ML605 boards are connected to one another via LPC (left) and HPC (right) FMC cables.

However, the interfacing modules may need to be adjusted before they are loaded onto the FPGA to account of possible clock skew.

When sending data over the low- and high-pin-count-connector, the sending FPGA will supply a clock signal to the receiving module. However, given variable resistance in the paths of the data signals, our data values may be arriving at different times; we must make sure that the clock edge is positioned to read all data values correctly. The Virtex-6 comes equipped with two mixed mode clock managers (MMCMs) that can help multiply, divide, and adjust the phase of clocks for this purpose [28]. Challenges with clock skew will likely limit the clock frequency at which we can run our channels.

As details about the system continue to develop, the boot sequence of the host board will also solidify. Determining how the host board will discover its position in the

system, how it will boot the many-core chip and how to debug this difficult process are all hurdles for the future.

9.2 Additional Optimizations

In addition to testing the system, there is plenty of optimization which could be done to boost the performance of the system. Optimizing the clock frequency of the channels to guarantee fast robust data transfer will probably be a future goal of this project. Studying methods for compressing data before sending it across the HPC and LPC connectors may also be worthwhile investigation.

Additionally, the size of the buffer for the translation module which integrates the memory controller with the rest of the system may be an interesting study. Given the number of in-flight commands the memory controller can handle and the number of requests arriving from all nodes in the network, the buffer size of the translation module may be an interesting point of optimization. We would like to reduce network congestion by storing as many requests as possible, but there are limits to the size of this buffer in our system.

Finally, building the host board to provide information about the many-core system will be very useful. Given its flexible structure, the FPGA is the perfect place to house modules to track errors and statistics about bandwidth usage in the system. The FPGA will become a critical component in developing, expanding and testing this system in the future.

Section X: Conclusion

This project has made significant progress in building the host board for the many-core Princeton Parallel Processor designed by David Wentzlaff and his team. Design for the memory controller, chip interface, inter-node interface and routing mechanisms are complete. However, as the many-core chip comes closer to completion, there will be more changes to make to the system. As this research will be passed to the members of Wentzlaff's team, this thesis will hopefully serve as an introduction and guide on past decisions made in the design of the host board in order to make future development easier.

Over the course of this project, I have learned a substantial amount about building real systems. During the course of my studies under David Wentzlaff (ELE 475 and ELE 580A), I gained a solid knowledge about modern CPU architecture and the features of massively parallel computing systems. Working on this project helped to solidify my knowledge in these areas and my understanding of how to design and build this kind of system as part of a team. I learned about DRAM architecture, became comfortable analyzing networks, gained real experience writing test benches, grasped the challenges of developing complex logic for FPGAs, and learned solid techniques for coordinating a large project distributed among a group. Hopefully this document can relay some of the lessons I have learned to those who will complete this project in the future.

Works Cited

[1] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th annual international symposium on Computer Architecture,* pp. 124-131, June 13-17, 1983.

[2] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *Proceedings of the 17th annual international symposium on Computer Architecture,* pp. 148-159, 28-31 May 1990.

[3] D. Chaiken, J. Kubiatowicz and A. Agarwal, "LimitLESS directories: A scalable cache coherence scheme," *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems,* pp. 224-234, 8-11 April 1991.

[4] D. Lenoski and J. Laudon, "The SGI Origin: A ccNUMA Highly Scalable Server," *In Proceedings of the 24th Annual International Symposium on omputer Architecture,* June 1997.

[5] N. Satoshi, T. Satoru, N. Norihito, W. Takayuki and S. Akihiro, "Hardware Technology of the SX-9 - Main System".

[6] L. Hammond, B. Nayfeh and K. Olukotun, "A single-chip multiprocessor," *IEEE,* vol. 30, no. 9, pp. 79-85, September 1997.

[7] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar and S. Borkar, "An 80-Tile Sub-100-W TeraFLOPS Processsor in 65nm," *CMOS. IEEE Journal of Solid State Circuits,* 2008.

[8] J. Kim, "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks," *ISLPED,* pp. 424-427, 2003.

[9] C. Worth, "StarFire: Extending the SMP envelope," *IEEE,* 1998.

[10] "Xilinx Virtex-6 Product Selection Guide," [Online]. Available: http://www.xilinx.com/publications/matrix/Product_Selection_Guide.pdf. [Accessed 27 April 2014].

[11] A. Esa, "Design of an Arbiter for DDR3 Memory," Worchester Polytechnic Institute, 2013.

[12] "ML605 Product Guide," [Online]. Available: http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf. [Accessed 27 April 2014].

[13] D. A. Patterson and J. L. Hennessy, Computer Architecture and Quantitative Approach, Waltham Elsevier, 2012.

[14] "Xilinx Vertex-6 FPGA Memory Interface Solutions," [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf. [Accessed 27 April 2014].

[15] "ML605 Hardware User Guide," [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf. [Accessed 27 April 2014].

[16] C. E. Cummings, “Simulation and synthesis techniques for asynchronous FIFO design,” *SNUG (Synopsys Users Group Conference)*, 2002.

[17] “Wikimedia Image Library”.

[18] “Wikimedia Image Library”.

[19] M. Superbasaux and C. de, “DRAM Technology,” 1997.

[20] B. Jacob, S. W. Ng and D. T. Wang, “Memory Systems – Cache, DRAM, Disk,” Elsevier, 2008.

[21] Hewlett-Packard Development Company, LP., “Memory Technology Evolution: An Overview of System Memory Technologies,” December 2010.

[22] “Micron DDR3 Datasheet,” [Online]. Available: http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf. [Accessed 27 April 2014].

[23] Y. H. Song and T. M. Pinkston, “A Progressive Approach to Handling Message-Dependent Deadlock in Parallel Computer Systems,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, no. 3, pp. 259-275, 2003.

[24] Ni, C. J. Glass and L. M., “The Turn Model for Adaptive Routing,” East Lansing, MI, Michigan State University.

[25] W. J. Dally and C. L. Seitz, “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks,” Vols. C-36, no. 5, May 1987.

[26] M. M. H. Rahman, A. Shah and Y. Inoguchi, “A Deadlock-Free Dimension Order Routing for Hierarchical 3D-Mesh Network,” *International Conference of Computer and Information Science (ICCIS),* 2012.

[27] “Xilinx Command Line Tools Guide,” [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/devref.pdf. [Accessed 27 April 2014].

[28] "Virtex-6 FPGA clocking resources user guide," [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug362.pdf. [Accessed 27 April 2014].

[29] Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis and N. Jouppi, “Rethinking DRAM Design and Organization for Energy-Contrained Multi-Cores,” *ACM SIGARCH Computer Architecture News,* vol. 38, no. 3, 2010.

[30] “Xilinx AXI Reference Guide,” [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. [Accessed 27 April 2014].

[31] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu and Patt, “Parallel application memory scheduling,” *In Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture,* 2011.

[32] Ahsan, Bushra and M. Zahran, “Cache performance, system performance, and off-chip bandwidth... pick any two,” *Proceedings of INA-OCMC,* 2009.

[33] Han, HoSuk and K. S. Stevens, “Clocked and asynchronous FIFO characterization and comparison,” *Very Large Scale Integration (VLSI-SoC),* vol. 17, 2009.

[34] T. M. Pinkston and S. Warnakulasuriya, “Characterization of Deadlocks in k-ary n-Cube Networks,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 10, no. 9, pp. 904-921, September 1999.

[35] R. Holsmark, S. Kumar, M. Palesi and A. Mejia, “HiRA: A Methodology for Deadlock Free Routing in Hierarchical Networks on Chip,” *IEEE,* 2009.

Appendix A – chip_bridge_send_32.v

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Module encapsulates the off chip interface for networks 1, 2 and 3
 * channel is 32 bits wide, splitting flits into two parts before send
 * relies on arbitration logic in chip_net_chooser.v
 *
 *****************************************************************/
`timescale 1ns/1ps

module chip_bridge_out (
    rst,
    wr_clk,
    rd_clk,
    bin_data_1,
    bin_val_1,
    bin_rdy_1,
    bin_data_2,
    bin_val_2,
    bin_rdy_2,
    bin_data_3,
    bin_val_3,
    bin_rdy_3,
    data_to_fpga,
    data_channel,
    credit_from_fpga
);

input rst;
input wr_clk;
input rd_clk;

output          bin_rdy_1;
output          bin_rdy_2;
output          bin_rdy_3;
output [31:0]   data_to_fpga;
output [ 1:0]    data_channel;

input [63:0]    bin_data_1;
input           bin_val_1;
input [63:0]    bin_data_2;
input           bin_val_2;
input [63:0]    bin_data_3;
input           bin_val_3;
input [2:0]     credit_from_fpga;

wire network_rdy_1;
wire network_rdy_2;
wire network_rdy_3;

wire fifo1_full;
wire fifo2_full;
wire fifo3_full;

wire [63:0] network_data_1;
wire [63:0] network_data_2;
wire [63:0] network_data_3;

wire network_val_1;
wire network_val_2;
wire network_val_3;

wire [63:0] data_to_serial_buffer;
reg  [31:0] serial_buffer_data[1:0];
reg   [0:0] serial_buffer_data_counter;
wire  [1:0] channel_to_serial_buffer;
reg   [1:0] serial_buffer_channel;
```

```
assign bin_rdy_1 = ~fifo1_full;
assign bin_rdy_2 = ~fifo2_full;
assign bin_rdy_3 = ~fifo3_full;

assign data_to_fpga = serial_buffer_data[serial_buffer_data_counter];
assign data_channel = serial_buffer_channel;

bridge_network_chooser separator(
    .rst    (rst),
    .clk    (rd_clk),
    .data_out(data_to_serial_buffer),
    .data_channel(channel_to_serial_buffer),
    .din_1  (network_data_1),
    .rdy_1  (network_rdy_1),
    .val_1  (~network_val_1),
    .din_2  (network_data_2),
    .rdy_2  (network_rdy_2),
    .val_2  (~network_val_2),
    .din_3  (network_data_3),
    .rdy_3  (network_rdy_3),
    .val_3  (~network_val_3),
    .credit_from_fpga(credit_from_fpga)
);
//input-output fixed to 64 bits
async_fifo #(
.DSIZE(64),
.ASIZE(3),
.MEMSIZE(8) )
fifo_1(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(network_rdy_1),
    .wval(bin_val_1),
    .wdata(bin_data_1),
    .rdata(network_data_1),
    .wfull(fifo1_full),
    .rempty(network_val_1)
);

async_fifo #(
.DSIZE(64),
.ASIZE(3),
.MEMSIZE(8) )
fifo_2(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(network_rdy_2),
    .wval(bin_val_2),
    .wdata(bin_data_2),
    .rdata(network_data_2),
    .wfull(fifo2_full),
    .rempty(network_val_2)
);

async_fifo #(
.DSIZE(64),
.ASIZE(3),
.MEMSIZE(8) )
fifo_3(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(network_rdy_3),
    .wval(bin_val_3),
```

```
    .wdata(bin_data_3),
    .rdata(network_data_3),
    .wfull(fifo3_full),
    .rempty(network_val_3)
);

always @(posedge rd_clk) begin
    if(rst) begin
        serial_buffer_data [0] <= 32'd0;
        serial_buffer_data [1] <= 32'd0;
        serial_buffer_channel <= 2'd0;
        serial_buffer_data_counter = 1'b1;
    end
    else begin
        if( channel_to_serial_buffer != 0 && serial_buffer_data_counter == 1) begin

            serial_buffer_data[0] <= data_to_serial_buffer[31:0];
            serial_buffer_data[1] <= data_to_serial_buffer[63:32];
            serial_buffer_channel <= channel_to_serial_buffer;

            serial_buffer_data_counter <= serial_buffer_data_counter + 1'b1;
        end
        else if( data_channel != 0 && serial_buffer_data_counter != 1'b1) begin
            serial_buffer_data_counter <= serial_buffer_data_counter + 1'b1;
        end
        else begin
            serial_buffer_data_counter <= 1'b1;
            serial_buffer_channel <= channel_to_serial_buffer;
        end
    end
end

endmodule
```

Appendix B – chip_net_chooser_32.v

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Module encapsulates logic used to choose between 3 on chip networks
 * based upon round robin ordering factoring in which networks are
 * empty and which have been given priority previously
 * ***************************************************************/
`timescale 1ns/1ps

module bridge_network_chooser(
    rst,
    clk,
    data_out,
    data_channel,
    din_1,
    rdy_1,
    val_1,
    din_2,
    rdy_2,
    val_2,
    din_3,
    rdy_3,
    val_3,
    credit_from_fpga
);

input rst;
input clk;

input [63:0] din_1;
input [63:0] din_2;
input [63:0] din_3;
input        val_1;
input        val_2;
input        val_3;
input [ 2:0] credit_from_fpga;

output [63:0] data_out;
output [ 1:0] data_channel;
output        rdy_1;
output        rdy_2;
output        rdy_3;

reg [8:0] credit_1; //holds the number of in flight instructions
reg [8:0] credit_2;
reg [8:0] credit_3;

wire [1:0] select;
reg  [0:0] select_counter;

reg sel_23;
reg sel_13;
reg sel_12;
reg [1:0] sel_123;


/*******************************************************
//Combinational Logic
*******************************************************/
assign data_out =   rdy_1 ? din_1 :
                    rdy_2 ? din_2 :
                    rdy_3 ? din_3 : 64'd0;

assign data_channel = select;

assign rdy_1 = (select == 2'b01 && select_counter == 0) ? 1 : 0;
assign rdy_2 = (select == 2'b10 && select_counter == 0) ? 1 : 0;
```

```
assign rdy_3 = (select == 2'b11 && select_counter == 0) ? 1 : 0;

assign select = ( (select_counter != 0          ) )   ? select :
                ( (credit_1 == 9'd255 || ~val_1) &&            //3 networks full
                  (credit_2 == 9'd255 || ~val_2) &&
                  (credit_3 == 9'd255 || ~val_3) )   ? 2'b00  :
                ( (credit_2 == 9'd255 || ~val_2) &&            //2 networks full
                  (credit_3 == 9'd255 || ~val_3) )   ? 2'b01  :
                ( (credit_1 == 9'd255 || ~val_1) &&
                  (credit_3 == 9'd255 || ~val_3) )   ? 2'b10  :
                ( (credit_1 == 9'd255 || ~val_1) &&
                  (credit_2 == 9'd255 || ~val_2) )   ? 2'b11  :
                ( (credit_1 == 9'd255 || ~val_1) )    ? (sel_23 ? 2'b11 : 2'b10) : //1
network full
                ( (credit_2 == 9'd255 || ~val_2) )   ? (sel_13 ? 2'b11 : 2'b01) :
                ( (credit_3 == 9'd255 || ~val_3) )   ? (sel_12 ? 2'b10 : 2'b01) :
                                                       sel_123; //0 networks full
/*************************************************************
//Sequential Logic
*************************************************************/
always @(posedge clk) begin
    if(rst) begin
        credit_1 <= 9'd0;
        credit_2 <= 9'd0;
        credit_3 <= 9'd0;
        sel_23 <= 0;
        sel_13 <= 0;
        sel_12 <= 0;
        sel_123 <= 0;

        select_counter <= 0;
    end
    else begin
        //update select hold
        if(select == 0) begin
            select_counter <= 0;
        end
        else begin
            select_counter <= select_counter + 2'b01;
        end

        //update incoming credits
        if(credit_from_fpga[0] & ~(rdy_1 & val_1)) begin
            credit_1 <= credit_1 - 9'd1;
        end
        if(credit_from_fpga[1] & ~(rdy_2 & val_2)) begin
            credit_2 <= credit_2 - 9'd1;
        end
        if(credit_from_fpga[2] & ~(rdy_3 & val_3)) begin
            credit_3 <= credit_3 - 9'd1;
        end

        if((credit_1 < 9'd255) &&
           (credit_2 < 9'd255) &&
           (credit_3 < 9'd255) &&
           (sel_123 == 0)          )
            sel_123 <= 2'b01;

        //update outgoing credits
        if(rdy_1 & val_1) begin
            sel_13 <= 1;
            sel_12 <= 1;
            if (sel_123 == 2'b01) begin
                sel_123 <= 2'b10;
            end
            if(~credit_from_fpga[0]) begin
                credit_1 <= credit_1 + 9'd1;
            end
        end
        if(rdy_2 & val_2) begin
```

```
            sel_23 <= 1;
            sel_12 <= 0;
            if (sel_123 == 2'b10) begin
                sel_123 <= 2'b11;
            end
            if( ~credit_from_fpga[1]) begin
                credit_2 <= credit_2 + 9'd1;
            end
        end
        if(rdy_3 & val_3) begin
            sel_23 <= 0;
            sel_13 <= 0;
            if (sel_123 == 2'b11) begin
                sel_123 <= 2'b01;
            end
            if (~credit_from_fpga[2]) begin
                credit_3 <= credit_3 + 9'd1;
            end
        end
    end
end //end always

endmodule
```

Appendix C – chip_bridge_rcv_32.v

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Module encapsulates the off chip receive interface for networks
 * 1, 2 and 3. packets received are 64 bits wide, received in 2
 * 32-bit payloads.
 *
 *****************************************************************/
`timescale 1ns/1ps

module chip_bridge_in (
    rst,
    wr_clk,
    rd_clk,
    bout_data_1,
    bout_val_1,
    bout_rdy_1,
    bout_data_2,
    bout_val_2,
    bout_rdy_2,
    bout_data_3,
    bout_val_3,
    bout_rdy_3,
    data_from_fpga,
    data_channel,
    credit_to_fpga
);

input rst;
input wr_clk;
input rd_clk;
input bout_rdy_1;
input bout_rdy_2;
input bout_rdy_3;
input [31:0] data_from_fpga;
input [ 1:0] data_channel;

output [63:0]   bout_data_1;
output          bout_val_1;
output [63:0]   bout_data_2;
output          bout_val_2;
output [63:0]   bout_data_3;
output          bout_val_3;
output [2:0]    credit_to_fpga;

wire sort_rdy_1;
wire sort_rdy_2;
wire sort_rdy_3;

wire [63:0] sort_data_1;
wire [63:0] sort_data_2;
wire [63:0] sort_data_3;

wire sort_val_1;
wire sort_val_2;
wire sort_val_3;

wire fifo1_empty;
wire fifo2_empty;
wire fifo3_empty;

wire fifo1_full;
wire fifo2_full;
wire fifo3_full;

assign bout_val_1 = ~fifo1_empty;
assign bout_val_2 = ~fifo2_empty;
```

```
assign bout_val_3 = ~fifo3_empty;


reg [31:0] channel_buffer [1:0];
reg [0:0]  channel_buffer_count;
wire [63:0] buffered_data;
reg [1:0] buffered_channel;

/*********************************************************
//credit FIFO boot system
*********************************************************/
assign  sort_data_1  =  (buffered_channel  ==  2'b01  &&  channel_buffer_count  ==  0)  ?
buffered_data : 64'd0;
assign  sort_data_2  =  (buffered_channel  ==  2'b10  &&  channel_buffer_count  ==  0)  ?
buffered_data : 64'd0;
assign  sort_data_3  =  (buffered_channel  ==  2'b11  &&  channel_buffer_count  ==  0)  ?
buffered_data : 64'd0;

assign sort_val_1   = (buffered_channel == 2'b01 && channel_buffer_count == 0) ? 1'b1 :
1'b0;
assign sort_val_2   = (buffered_channel == 2'b10 && channel_buffer_count == 0) ? 1'b1 :
1'b0;
assign sort_val_3   = (buffered_channel == 2'b11 && channel_buffer_count == 0) ? 1'b1 :
1'b0;

/*********************************************************
//buffer broken up flits
*********************************************************/
assign   buffered_data   =   {data_from_fpga,   channel_buffer[2],   channel_buffer[1],
channel_buffer[0]};

always @(posedge wr_clk) begin
    if(rst) begin
        channel_buffer[0] = 32'd0;
        channel_buffer[1] = 32'd0;
        channel_buffer_count = 0;
    end
    else begin
        if(data_channel != 0) begin //do not store data if no channel is assigned
            channel_buffer[channel_buffer_count] <= data_from_fpga;
            buffered_channel <= data_channel;
            channel_buffer_count <= channel_buffer_count + 1'b1;
        end
        else begin
            buffered_channel <= data_channel;
            channel_buffer_count <= 2'b00;
        end
    end
end

//input-output fixed to 64 bits
/*********************************************************
//data FIFOs
*********************************************************/
async_fifo #(
.DSIZE(64),
.ASIZE(4),
.MEMSIZE(16) )
fifo_1(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(bout_rdy_1),
    .wval(sort_val_1),
    .wdata(sort_data_1),
    .rdata(bout_data_1),
    .wfull(fifo1_full), //credit system should prevent this from ever being full
    .rempty(fifo1_empty)
);
```

```
async_fifo #(
.DSIZE(64),
.ASIZE(4),
.MEMSIZE(16) )
fifo_2(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(bout_rdy_2),
    .wval(sort_val_2),
    .wdata(sort_data_2),
    .rdata(bout_data_2),
    .wfull(fifo2_full), //credit system should prevent this from ever being full
    .rempty(fifo2_empty)
);

async_fifo #(
.DSIZE(64),
.ASIZE(4),
.MEMSIZE(16) )
fifo_3(
    .rreset(rst),
    .wreset(rst),
    .wclk(wr_clk),
    .rclk(rd_clk),
    .ren(bout_rdy_3),
    .wval(sort_val_3),
    .wdata(sort_data_3),
    .rdata(bout_data_3),
    .wfull(fifo3_full), //credit system should prevent this from ever being full
    .rempty(fifo3_empty)
);

/*********************************************************
//credit FIFO
*********************************************************/
wire [2:0] credit_gather;
wire credit_empty;
wire [2:0] credit_fifo_out;

reg  [2:0] credit_to_fpga_r;

//FIFO used for bridging credits between clock domains
async_fifo #(
.DSIZE(3),
.ASIZE(4),
.MEMSIZE(16) )
credit_fifo(
    .rreset(rst),
    .wreset(rst),
    .wclk(rd_clk),
    .rclk(wr_clk),
    .ren(~rst),
    .wval(~(rst)),
    .wdata(credit_gather),
    .rdata(credit_fifo_out),
    .wfull(),               //credit fifo should never be full
    .rempty(credit_empty)
);

assign credit_to_fpga = credit_to_fpga_r;

assign credit_gather[0] = bout_rdy_1 & bout_val_1;
assign credit_gather[1] = bout_rdy_2 & bout_val_2;
assign credit_gather[2] = bout_rdy_3 & bout_val_3;

/*************************************************
 channel side sequential logic
```

```
 ************************************************/
always@(posedge wr_clk) begin
   if(rst) begin
        credit_to_fpga_r <= 3'b000;
   end
   else begin
       if(~credit_empty) begin
           credit_to_fpga_r <= credit_fifo_out;
       end
       else
           credit_to_fpga_r <= 3'b000;
   end
end

endmodule
```

Appendix D – fpga_bridge_send_32.v

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Module encapsulates the off chip interface for networks 1, 2 and 3
 * channel is 32 bits wide, splitting flits into four parts before send
 * relies on arbitration logic in fpga_net_chooser_32.v
 *
 *****************************************************************/
`timescale 1ns/1ps

module fpga_bridge_send_32 (
    rst,
    wr_clk,
    rd_clk,
    bin_data_1,
    bin_val_1,
    bin_rdy_1,
    bin_data_2,
    bin_val_2,
    bin_rdy_2,
    bin_data_3,
    bin_val_3,
    bin_rdy_3,
    data_to_chip,
    data_channel,
    credit_from_chip
);

input rst;
input wr_clk;
input rd_clk;

output          bin_rdy_1;
output          bin_rdy_2;
output          bin_rdy_3;
output [31:0]   data_to_chip;
output [ 1:0]    data_channel;

input [63:0]    bin_data_1;
input           bin_val_1;
input [63:0]    bin_data_2;
input           bin_val_2;
input [63:0]    bin_data_3;
input           bin_val_3;
input [2:0]     credit_from_chip;

wire network_rdy_1;
wire network_rdy_2;
wire network_rdy_3;

wire fifo1_full;
wire fifo2_full;
wire fifo3_full;

wire [63:0] network_data_1;
wire [63:0] network_data_2;
wire [63:0] network_data_3;

wire network_val_1;
wire network_val_2;
wire network_val_3;

wire network_empty_1;
wire network_empty_2;
wire network_empty_3;

wire [63:0] data_to_serial_buffer;
```

```
reg  [31:0] serial_buffer_data[1:0];
reg   [0:0] serial_buffer_data_counter;
wire  [1:0] channel_to_serial_buffer;
reg   [1:0] serial_buffer_channel;

assign bin_rdy_1 = ~fifo1_full;
assign bin_rdy_2 = ~fifo2_full;
assign bin_rdy_3 = ~fifo3_full;

assign data_to_chip = serial_buffer_data[serial_buffer_data_counter];
assign data_channel = serial_buffer_channel;

bridge_network_chooser_32 separator(
    .rst    (rst),
    .clk    (rd_clk),
    .data_out(data_to_serial_buffer),
    .data_channel(channel_to_serial_buffer),
    .val_1  (network_val_1),
    .val_2  (network_val_2),
    .val_3  (network_val_3),
    .din_1  (network_data_1),
    .din_2  (network_data_2),
    .din_3  (network_data_3),
    .rdy_1  (network_rdy_1),
    .rdy_2  (network_rdy_2),
    .rdy_3  (network_rdy_3),
    .empty_1(network_empty_1),
    .empty_2(network_empty_2),
    .empty_3(network_empty_3),
    .credit_from_chip(credit_from_chip)
);
//input-output fixed to 64 bits

a_fifo_w64 fifo_1(
    .rst(rst),
    .wr_clk(wr_clk),
    .rd_clk(rd_clk),
    .din(bin_data_1),
    .wr_en(bin_val_1),
    .rd_en(network_rdy_1),
    .dout(network_data_1),
    .full(fifo1_full),
    .valid(network_val_1),
    .empty(network_empty_1)
);

a_fifo_w64 fifo_2(
    .rst(rst),
    .wr_clk(wr_clk),
    .rd_clk(rd_clk),
    .din(bin_data_2),
    .wr_en(bin_val_2),
    .rd_en(network_rdy_2),
    .dout(network_data_2),
    .full(fifo2_full),
    .valid(network_val_2),
    .empty(network_empty_2)
);

a_fifo_w64 fifo_3(
    .rst(rst),
    .wr_clk(wr_clk),
    .rd_clk(rd_clk),
    .din(bin_data_3),
    .wr_en(bin_val_3),
    .rd_en(network_rdy_3),
    .dout(network_data_3),
    .full(fifo3_full),
    .valid(network_val_3),
    .empty(network_empty_3)
```

```
);

always @(posedge rd_clk) begin
    if(rst) begin
        serial_buffer_data [0] <= 32'd0;
        serial_buffer_data [1] <= 32'd0;
        serial_buffer_channel <= 2'd0;
        serial_buffer_data_counter = 1'b1;
    end
    else begin
        if( channel_to_serial_buffer != 0 && serial_buffer_data_counter == 1) begin

            serial_buffer_data[0] <= data_to_serial_buffer[31:0];
            serial_buffer_data[1] <= data_to_serial_buffer[63:32];
            serial_buffer_channel <= channel_to_serial_buffer;

            serial_buffer_data_counter <= serial_buffer_data_counter + 1'b1;
        end
        else if( data_channel != 0 && serial_buffer_data_counter != 1'b1) begin
            serial_buffer_data_counter <= serial_buffer_data_counter + 1'b1;
        end
        else begin
            serial_buffer_data_counter <= 1'b1;
            serial_buffer_channel <= channel_to_serial_buffer;
        end
    end
end

endmodule
```

Appendix E – fpga_net_chooser_32.v

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Module encapsulates logic used to choose between 3 on chip networks
 * based upon round robin ordering factoring in which networks are
 * empty and which have been given priority previously
 * ***************************************************************/
`timescale 1ns/1ps

module bridge_network_chooser_32 (
    rst,
    clk,
    data_out,
    data_channel,
    din_1,
    rdy_1,
    val_1,
    din_2,
    rdy_2,
    val_2,
    din_3,
    rdy_3,
    val_3,
    empty_1,
    empty_2,
    empty_3,
    credit_from_chip
);

input rst;
input clk;

input [63:0] din_1;
input [63:0] din_2;
input [63:0] din_3;
input        val_1;
input        val_2;
input        val_3;
input [ 2:0] credit_from_chip;
input empty_1;
input empty_2;
input empty_3;

output [63:0] data_out;
output [ 1:0] data_channel;
output        rdy_1;
output        rdy_2;
output        rdy_3;

reg [8:0] credit_1; //holds the number of in flight instructions
reg [8:0] credit_2;
reg [8:0] credit_3;

wire [1:0] select;
reg  [1:0] select_counter;

reg sel_23;
reg sel_13;
reg sel_12;
reg [1:0] sel_123;

/*******************************************************
//Combinational Logic
*******************************************************/
assign data_out =   val_1 ? din_1 :
                    val_2 ? din_2 :
                    val_3 ? din_3 : 64'd0;
```

```
assign data_channel =   val_1 ? 2'b01 :
                        val_2 ? 2'b10 :
                        val_3 ? 2'b11 : 2'b00;

assign rdy_1 = (select == 2'b01 && select_counter == 0) ? 1 : 0;
assign rdy_2 = (select == 2'b10 && select_counter == 0) ? 1 : 0;
assign rdy_3 = (select == 2'b11 && select_counter == 0) ? 1 : 0;

assign select = ( (select_counter != 0        ) )   ? select :
                ( (credit_1 == 9'd7 || empty_1) &&            //3 networks full
                  (credit_2 == 9'd7 || empty_2) &&
                  (credit_3 == 9'd7 || empty_3) )   ? 2'b00  :
                ( (credit_2 == 9'd7 || empty_2) &&            //2 networks full
                  (credit_3 == 9'd7 || empty_3) )   ? 2'b01  :
                ( (credit_1 == 9'd7 || empty_1) &&
                  (credit_3 == 9'd7 || empty_3) )   ? 2'b10  :
                ( (credit_1 == 9'd7 || empty_1) &&
                  (credit_2 == 9'd7 || empty_2) )   ? 2'b11  :
                ( (credit_1 == 9'd7 || empty_1) )    ? (sel_23 ? 2'b11 : 2'b10) : //1
network full
                ( (credit_2 == 9'd7 || empty_2) )   ? (sel_13 ? 2'b11 : 2'b01) :
                ( (credit_3 == 9'd7 || empty_3) )   ? (sel_12 ? 2'b10 : 2'b01) :
                                                    sel_123; //0 networks full

/*****************************************************
//Sequential Logic
*****************************************************/
always @(posedge clk) begin
    if(rst) begin
        credit_1 <= 9'd0;
        credit_2 <= 9'd0;
        credit_3 <= 9'd0;
        sel_23 <= 0;
        sel_13 <= 0;
        sel_12 <= 0;
        sel_123 <= 0;

        select_counter <= 0;
    end
    else begin
        //update select hold
        if(select == 0) begin
            select_counter <= 0;
        end
        else begin
            select_counter <= select_counter + 2'b01;
        end

        //update incoming credits
        if(credit_from_chip[0] & ~(val_1)) begin
            credit_1 <= credit_1 - 9'd1;
        end
        if(credit_from_chip[1] & ~(val_2)) begin
            credit_2 <= credit_2 - 9'd1;
        end
        if(credit_from_chip[2] & ~(val_3)) begin
            credit_3 <= credit_3 - 9'd1;
        end

        if((credit_1 < 9'd7) &&
           (credit_2 < 9'd7) &&
           (credit_3 < 9'd7) &&
           (sel_123 == 0)          )
            sel_123 <= 2'b01;

        //update outgoing credits
        if(val_1) begin
            sel_13 <= 1;
            sel_12 <= 1;
```

```
            if (sel_123 == 2'b01) begin
                sel_123 <= 2'b10;
            end
            if(~credit_from_chip[0]) begin
                credit_1 <= credit_1 + 9'd1;
            end
        end
        if(val_2) begin
            sel_23 <= 1;
            sel_12 <= 0;
            if (sel_123 == 2'b10) begin
                sel_123 <= 2'b11;
            end
            if( ~credit_from_chip[1]) begin
                credit_2 <= credit_2 + 9'd1;
            end
        end
        if(val_3) begin
            sel_23 <= 0;
            sel_13 <= 0;
            if (sel_123 == 2'b11) begin
                sel_123 <= 2'b01;
            end
            if (~credit_from_chip[2]) begin
                credit_3 <= credit_3 + 9'd1;
            end
        end
    end
end //end always
endmodule
```

Appendix F – chip_bridge_send_top.v

This is the top level module for conducting tests involving the chip sending data to the FPGA – other modules exist for FPGA to chip communication, and FPGA to FPGA communication but are not included. Each of these additional designs is very similar to the one shown here.

```
`timescale 1ns/1ps
module chip_bridge_out_top(
    rst,
    chip_clk,
    fpga_clk,
    intcnct_clk,
    network_in_1,
    network_in_2,
    network_in_3,
    data_in_val_1,
    data_in_val_2,
    data_in_val_3,
    data_in_rdy_1,
    data_in_rdy_2,
    data_in_rdy_3,
    network_out_1,
    network_out_2,
    network_out_3,
    data_out_val_1,
    data_out_val_2,
    data_out_val_3,
    data_out_rdy_1,
    data_out_rdy_2,
    data_out_rdy_3,
    dbg_interconnect_data,
    dbg_interconnect_channel
);

input    rst;
input    chip_clk;
input    fpga_clk;
input    intcnct_clk;
input [63:0]   network_in_1;
input [63:0]   network_in_2;
input [63:0]   network_in_3;
input    data_in_val_1;
input    data_in_val_2;
input    data_in_val_3;
input    data_out_rdy_1;
input    data_out_rdy_2;
input    data_out_rdy_3;

output    data_in_rdy_1;
output    data_in_rdy_2;
output    data_in_rdy_3;
output [63:0]   network_out_1;
output [63:0]   network_out_2;
output [63:0]   network_out_3;
output    data_out_val_1;
output    data_out_val_2;
output    data_out_val_3;

output [31:0] dbg_interconnect_data;
output [1:0] dbg_interconnect_channel;

wire [31:0] intcnct_data;
wire [1:0]  intcnct_channel;
wire [2:0]  intcnct_credit_back;

assign dbg_interconnect_data = intcnct_data;
assign dbg_interconnect_channel = intcnct_channel;
```

```
chip_bridge_out chip_side(
    .rst(rst),
    .wr_clk(chip_clk),
    .rd_clk(intcnct_clk),
    .bin_data_1(network_in_1),
    .bin_val_1(data_in_val_1),
    .bin_rdy_1(data_in_rdy_1),
    .bin_data_2(network_in_2),
    .bin_val_2(data_in_val_2),
    .bin_rdy_2(data_in_rdy_2),
    .bin_data_3(network_in_3),
    .bin_val_3(data_in_val_3),
    .bin_rdy_3(data_in_rdy_3),
    .data_to_fpga(intcnct_data),
    .data_channel(intcnct_channel),
    .credit_from_fpga(intcnct_credit_back)
);

fpga_bridge_rcv_32 fpga_side (
    .rst(rst),
    .wr_clk(intcnct_clk),
    .rd_clk(fpga_clk),
    .bout_data_1(network_out_1),
    .bout_val_1(data_out_val_1),
    .bout_rdy_1(data_out_rdy_1),
    .bout_data_2(network_out_2),
    .bout_val_2(data_out_val_2),
    .bout_rdy_2(data_out_rdy_2),
    .bout_data_3(network_out_3),
    .bout_val_3(data_out_val_3),
    .bout_rdy_3(data_out_rdy_3),
    .data_from_chip(intcnct_data),
    .data_channel(intcnct_channel),
    .credit_to_chip(intcnct_credit_back)
);
endmodule
```

Appendix G – chip_bridge_send_top.t.v

This is a non-synthesizable test bench to simulate data being sent across the chip-FPGA interface.

```
//Testbench for the chip interface -- data coming off of chip
`define VERBOSITY 1
`define CHIP_PERIOD 2
`define FPGA_PERIOD 3
`define CHANNEL_PERIOD 1

//randomly generate input delays
`define WRITE1_CHANCE 10
`define WRITE2_CHANCE 10
`define WRITE3_CHANCE 10

//randomly generate output delays
`define READ1_CHANCE 10
`define READ2_CHANCE 10
`define READ3_CHANCE 10

`timescale 1ns/1ps

module bridge_out_tb;

parameter DATASIZE = 255;

reg             boot_done;
reg             rst;
reg             chip_clk;
reg             fpga_clk;
reg             intcnct_clk;
wire   [63:0]   network_in_1;
wire   [63:0]   network_in_2;
wire   [63:0]   network_in_3;
reg             data_in_val_1;
reg             data_in_val_2;
reg             data_in_val_3;
wire            data_in_rdy_1;
wire            data_in_rdy_2;
wire            data_in_rdy_3;
wire   [63:0]   network_out_1;
wire   [63:0]   network_out_2;
wire   [63:0]   network_out_3;
wire            data_out_val_1;
wire            data_out_val_2;
wire            data_out_val_3;
reg             data_out_rdy_1;
reg             data_out_rdy_2;
reg             data_out_rdy_3;

chip_bridge_out_top dut(
    .rst(rst),
    .chip_clk(chip_clk),
    .fpga_clk(fpga_clk),
    .intcnct_clk(intcnct_clk),
    .network_in_1(network_in_1),
    .network_in_2(network_in_2),
    .network_in_3(network_in_3),
    .data_in_val_1(data_in_val_1),
    .data_in_val_2(data_in_val_2),
    .data_in_val_3(data_in_val_3),
    .data_in_rdy_1(data_in_rdy_1),
    .data_in_rdy_2(data_in_rdy_2),
    .data_in_rdy_3(data_in_rdy_3),
    .network_out_1(network_out_1),
    .network_out_2(network_out_2),
    .network_out_3(network_out_3),
    .data_out_val_1(data_out_val_1),
    .data_out_val_2(data_out_val_2),
    .data_out_val_3(data_out_val_3),
```

```
    .data_out_rdy_1(data_out_rdy_1),
    .data_out_rdy_2(data_out_rdy_2),
    .data_out_rdy_3(data_out_rdy_3),
    .dbg_interconnect_data(),
    .dbg_interconnect_channel()
);


integer i1, i2, i3;
integer j1, j2, j3;
integer seed1, seed2, seed3;
integer dbg1, dbg2, dbg3;
integer all_passed = 1;

reg [63:0]  write_mem_1[DATASIZE:0];
reg [63:0]  write_mem_2[DATASIZE:0];
reg [63:0]  write_mem_3[DATASIZE:0];

assign network_in_1 = write_mem_1[i1];
assign network_in_2 = write_mem_2[i2];
assign network_in_3 = write_mem_3[i3];

event check_output;

initial begin
    //initialize clocks
    chip_clk = 0;
    intcnct_clk = 0;
    fpga_clk = 0;

    #1  seed1 = $time;
    #1  seed2 = $time;
    #1  seed3 = $time;
    for( i1 = 0; i1 < DATASIZE; i1 = i1 + 1)begin
        write_mem_1[i1] = {$random(seed1), $random(seed2)};
        write_mem_2[i1] = {$random(seed1), $random(seed2)};
        write_mem_3[i1] = {$random(seed2), $random(seed2)};
    end
    i1 = 0;
    i2 = 0;
    i3 = 0;
    j1 = 0;
    j2 = 0;
    j3 = 0;

    $dumpfile("dump.vcd");
    $dumpvars(0, dut);
    all_passed = 1;
    $dumpon;
    boot_done = 0;
    rst = 0;
    #70
    rst = 1;
    #70
    rst = 0;
    #70
    boot_done = 1;

    #50000 //timeout
    $display( "    !TEST TIMEOUT!     ");
    $display( " [ TEST BENCH FAILED ] ");
    $finish;
end
/*****************************************************************
APPLY INPUTS
*****************************************************************/
//CHIP CLOCK
always
    #`CHIP_PERIOD chip_clk <= ~chip_clk;
```

```
//INTERCONNECT CLOCK
always
    #`CHANNEL_PERIOD intcnct_clk <= ~intcnct_clk;

//WRITING TESTBENCH (CHIP SIDE)
always@(posedge chip_clk) begin

    //network 1
    if (data_in_rdy_1 && data_in_val_1)
        i1 <= i1 + 1;

    if(({$random(seed1)%50+50} < `WRITE1_CHANCE) &&
       (boot_done) &&
       (i1 <= DATASIZE-1))begin
        data_in_val_1 <= 1'b1;
    end
    else begin
        data_in_val_1 <= 1'b0;
    end

    //network 2
    if (data_in_rdy_2 && data_in_val_2)
        i2 <= i2 + 1;

    if(({$random(seed1)%50 +50} < `WRITE2_CHANCE) &&
       (boot_done) &&
       (i2 <= DATASIZE-1))begin
        data_in_val_2 <= 1'b1;
    end
    else begin
        data_in_val_2 <= 1'b0;
    end

    //network 3
    if (data_in_rdy_3 && data_in_val_3)
        i3 <= i3 + 1;

    if(({$random(seed1)%50+50} < `WRITE3_CHANCE) &&
       (boot_done) &&
       (i3 <= DATASIZE-1))begin
        data_in_val_3 <= 1'b1;
    end
    else begin
        data_in_val_3 <= 1'b0;
    end
end
//FPGA CLOCK
always
    #`FPGA_PERIOD fpga_clk <= ~fpga_clk;

//READING TESTBENCH (FPGA SIDE)
always @(posedge fpga_clk) begin
    //network 1
    if(  ({$random(seed1)%50 + 50} < `READ1_CHANCE) &&
         ( boot_done ) &&
         ( j1 <= DATASIZE-1 )
       ) begin
        data_out_rdy_1 <= 1'b1;
    end
    else begin
        data_out_rdy_1 <= 1'b0;
    end

    //network 2
    if(  ({$random(seed1)%50+50} < `READ1_CHANCE) &&
         ( boot_done ) &&
         ( j2 <= DATASIZE-1 )
       ) begin
        data_out_rdy_2 <= 1'b1;
    end
```

```
    else begin
        data_out_rdy_2 <= 1'b0;
    end

    //network 3
    if(  ({$random(seed1)%50+50} < `READ1_CHANCE) &&
         ( boot_done ) &&
         ( j3 <= DATASIZE-1 )
       ) begin
        data_out_rdy_3 <= 1'b1;
    end
    else begin
        data_out_rdy_3 <= 1'b0;
    end

    if(data_out_val_1 || data_out_val_2 || data_out_val_3)
        -> check_output;
end

/*****************************************************************
EVENTS
*****************************************************************/

always @( check_output ) begin

    //check channel 1
    if(data_out_val_1 && j1 < DATASIZE - 1) begin
        if( network_out_1 != write_mem_1[j1] ) begin
            all_passed = 0;
            $display(  "  [  FAILED  ]  @%d  channel=1,  rdata=%h,  expected=%h;  ",  j1,
network_out_1, write_mem_1[j1][63:0]);
        end
        else if ( `VERBOSITY == 1 ) begin
            $display(  "  [  PASSED  ]  @%d  channel=1,  rdata=%h,  expected=%h;  ",  j1,
network_out_1, write_mem_1[j1][63:0]);
        end
        j1 = j1+1;
    end

    //check channel 2
    if(data_out_val_2 && j2 < DATASIZE - 1) begin
        if( network_out_2 != write_mem_2[j2] ) begin
            all_passed = 0;
            $display(  "  [  FAILED  ]  @%d  channel=2,  rdata=%h,  expected=%h;  ",  j2,
network_out_2, write_mem_2[j2][63:0]);
        end
        else if ( `VERBOSITY == 1 ) begin
            $display(  "  [  PASSED  ]  @%d  channel=2,  rdata=%h,  expected=%h;  ",  j2,
network_out_2, write_mem_2[j2][63:0]);
        end
        j2 = j2+1;
    end

    //check channel 1
    if(data_out_val_3 && j3 < DATASIZE - 1) begin
        if( network_out_3 != write_mem_3[j3] ) begin
            all_passed = 0;
            $display(  "  [  FAILED  ]  @%d  channel=3,  rdata=%h,  expected=%h;  ",  j3,
network_out_3, write_mem_3[j3][63:0]);
        end
        else if ( `VERBOSITY == 1 ) begin
            $display(  "  [  PASSED  ]  @%d  channel=3,  rdata=%h,  expected=%h;  ",  j3,
network_out_3, write_mem_3[j3][63:0]);
        end
    j3 = j3+1;
    end

    if( (j1 >= DATASIZE-1 ) &&
        (j2 >= DATASIZE-1 ) &&
        (j3 >= DATASIZE-1 ) ) begin
```

```
        if( all_passed )
            $display( " [ TEST BENCH PASSED ] ");
        else
            $display( " [ TEST BENCH FAILED ] ");
        $finish;
    end
end
endmodule
```

Appendix H – Asynchronous FIFO Source Code

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Asynchronous FIFO design, capable of functioning at a variety of
 * frequencies. This design has been tested in simulation for
 * functionality across read/write frequencies, latency and power use.
 * ***************************************************************/


module async_fifo
#(
  parameter DSIZE = 64,
  parameter ASIZE = 5,
  parameter MEMSIZE = 16 // should be 2 ^ (ASIZE-1)
)
(
  rdata,
  rempty,
  rclk,
  ren,
  wdata,
  wfull,
  wclk,
  wval,
  wreset,
  rreset
  );

//Inputs and Outputs
output  [DSIZE-1:0]   rdata;
output      rempty;
output      wfull;
input   [DSIZE-1:0]   wdata;
input       wval;
input       ren;
input       rclk;
input       wclk;
input       wreset;
input       rreset;

//Internal Registers
reg [ASIZE-1:0] g_wptr;
reg [ASIZE-1:0] g_rptr;

reg [ASIZE-1:0] g_rsync1, g_rsync2;
reg [ASIZE-1:0] g_wsync1, g_wsync2;

//Memory
reg [DSIZE-1:0]   fifo[MEMSIZE-1:0];

wire [ASIZE-1:0] b_wptr;
wire [ASIZE-1:0] b_wptr_next;
wire [ASIZE-1:0] g_wptr_next;
wire [ASIZE-1:0] b_rptr;
wire [ASIZE-1:0] b_rptr_next;
wire [ASIZE-1:0] g_rptr_next;

/*****************************************************************
COMBINATIONAL LOGIC
*****************************************************************/

//convert gray to binary
assign b_wptr[ASIZE-1:0] = ({1'b0, b_wptr[ASIZE-1:1]} ^ g_wptr[ASIZE-1:0]);
assign b_rptr[ASIZE-1:0] = ({1'b0, b_rptr[ASIZE-1:1]} ^ g_rptr[ASIZE-1:0]);

//increment
assign b_wptr_next = b_wptr + 1;
assign b_rptr_next = b_rptr + 1;
```

```
//convert binary to gray
assign g_wptr_next[ASIZE-1:0] = {1'b0, b_wptr_next[ASIZE-1:1]} ^ b_wptr_next[ASIZE-1:0];
assign g_rptr_next[ASIZE-1:0] = {1'b0, b_rptr_next[ASIZE-1:1]} ^ b_rptr_next[ASIZE-1:0];

//full and empty signals
assign wfull =  (g_wptr[ASIZE-1]   != g_rsync2[ASIZE-1]  ) &&
    (g_wptr[ASIZE-2]   != g_rsync2[ASIZE-2]  ) &&
    (g_wptr[ASIZE-3:0] == g_rsync2[ASIZE-3:0]) ||
    (wreset || rreset);

assign rempty =  (g_wsync2[ASIZE-1:0] == g_rptr[ASIZE-1:0]) ||
           (wreset || rreset);

//output values
assign rdata = fifo[b_rptr[ASIZE-2:0]];

/*****************************************************************
SEQUENTIAL LOGIC
*****************************************************************/

//transfer register values
always @(posedge rclk) begin
  if (rreset) begin
    g_rptr <= 0;
  end
  else if (ren && !rempty) begin
    g_rptr <= g_rptr_next;
  end

  g_wsync1 <= g_wptr;
  g_wsync2 <= g_wsync1;
end

always @(posedge wclk) begin
  if (wreset) begin
    g_wptr <= 0;
  end
  else if (wval && !wfull) begin
    fifo[b_wptr[ASIZE-2:0]] <= wdata;
    g_wptr <= g_wptr_next;
  end

  g_rsync1 <= g_rptr;
  g_rsync2 <= g_rsync1;
end
endmodule
```

Appendix I – An Example Asynchronous FIFO Test Bench

This test bench is capable of testing a range of frequencies and the latencies of transactions. The output waveforms of this test can also be used for power analysis using Primetime PX.

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Test Bench for async fifo
 * ***************************************************************/

`define VERBOSITY 1             //report all read values
`define WPERIOD 1
`define RPERIOD 1               //period of read clock
`define WDELAY  0               //phase offset of write clock
`define RDELAY  0
`define TB_ID   R1--W1
`define FIFO_DESIGN async_fifo
`define CHECK_PERIOD 0
`define LOAD_PERCENT 100

//no need to parameterize these in scripts
`define OUT_PREFIX /scratch/gpfs/spayne/script_test_updateTB/vcd/dump_async_fifo_tb
`define OUT_SUFFIX vcd
`define STRINGIFY(x, y, z) `"x.y.z`"


`timescale 1ns/1ps              //do not change

//parameters
parameter DATASIZE = 512;       //number of values to be written and tested
parameter TB_ASIZE = 5;          //address size of test bench (remember 1 bit reserved for
wrap around)
parameter TB_DSIZE = 64;        //size of data fed into test bench
parameter TB_MEMSIZE = 16;      //depth of fifo should be 2 ^ (ASIZE-1)

module async_fifo_tb;

/*****************************************************************
DECLARATIONS
*****************************************************************/
        wire    [TB_DSIZE-1:0]  rdata;
        wire                    rempty;
        reg                     rclk;
        reg                     ren;
        reg     [TB_DSIZE-1:0]  wdata;
        wire                    wfull;
        reg                     wclk;
        reg                     wval;
        reg                     wreset;
        reg                     rreset;
        //fifo module under test
        `FIFO_DESIGN #( .DSIZE(TB_DSIZE),
                        .ASIZE(TB_ASIZE),
                        .MEMSIZE(TB_MEMSIZE) ) async_fifo1
        (
                .rdata  (rdata),
                .rempty (rempty),
                .rclk   (rclk),
                .ren    (ren),
                .wdata  (wdata),
                .wfull  (wfull),
                .wclk   (wclk),
                .wval   (wval),
                .wreset (wreset),
                .rreset (rreset)
        );

        //test memories
```

```
        reg     [TB_DSIZE-1:0]  check_val;
        reg     [TB_DSIZE-1:0]  write_mem[DATASIZE];
        reg     [TB_DSIZE-1:0]  check_mem[DATASIZE];
        integer                 latency[DATASIZE];
        integer                 avg_latency;
/*****************************************************************
INITIALIZATIONS
*****************************************************************/

        //loop variable
        integer i;

        //random number generator seed
        integer seed1, seed2, seed3;

        //initialize test memories
        initial begin
                seed1 = $time;
                seed2 = $time;
                seed3 = $time;
                for( i = 0; i < DATASIZE; i=i+1) begin
                        write_mem[i] = {$random(seed1), $random(seed2)};
                        check_mem[i] = write_mem[i];
                end

        end
        //read loop variable
        integer j;

        //write loop variable
        integer k;

        //event to trigger checking of output
        event check_output;

        //integer for determining if all tests passed
        integer all_passed;

        //setup dump file
        initial begin
                $dumpfile(`STRINGIFY(`OUT_PREFIX, `TB_ID, `OUT_SUFFIX));           //this
should be different for each test/test bench
                $dumpvars(0, async_fifo1);               //module's variables recorded in
dump file
                all_passed = 1;
                $dumpon;
        end
/*****************************************************************
APPLY INPUTS
*****************************************************************/
        //READ CLIENT
        integer checking;
        initial begin
                k = 0;          //initialize counter variable
                checking = 0;
                #`RDELAY
                rclk    = 0;    //read clock starts low
                ren     = 1;    //read enable starts high
                rreset  = 1;    //reset starts high - keep in mind we do not record power
while reset is high
                forever begin
                        //READ CLIENT
                        check_val <= rdata;
                        if(checking) begin
                        #`CHECK_PERIOD rclk = ~rclk; //check_period accounts for checking
time
                        checking = 0;
                        end
                        else begin
```

```
                        #`RPERIOD rclk = ~rclk; //toggle the read clock on every clock
period
                        end


                        if(rclk == 1 && rempty == 0 && ren == 1) begin
                        #1 -> check_output;
                        checking = 1;
                        //avg_latency = avg_latency+latency[k];
                        //$display( " latency = %d ", latency[k]);

                        end
                        if(k >= DATASIZE) begin
                                $dumpoff;
                                avg_latency = avg_latency/DATASIZE;
                                $display( " AVERAGE LATENCY = %d ", avg_latency);
                                if( all_passed )
                                        $display( " [ TEST BENCH PASSED ] ");
                                else
                                        $display( " [ TEST BENCH FAILED ] ");
                                $finish;
                        end
                end
        end

        //WRITE CLIENT
        initial begin
                wval = 0;
                avg_latency = 0;
                j = 0;          //initialize counter variable
                wdata   = 0;    //start write data at 0
                #`WDELAY        //determines the offset in phase of the write client
                wclk    = 0;    //start write clock low
                wreset   = 1;     //start reset high - remember we do not record power
during this period
                forever begin
                        //WRITE CLIENT
                        #`WPERIOD wclk = ~wclk; //on write period, toggle write clock

                        if($random(seed3)%100 < `LOAD_PERCENT)
                                wval = 1;
                        else begin
                                //$display( "DEBUG: no write");
                                wval = 0;
                        end

                        wdata   = write_mem[j];
                        if(wclk == 1 && wfull == 0 && wval == 1 && j <= DATASIZE-1) begin
                                latency[j]=$time;
                                j=j+1;
                        end
                end
        end

        initial begin
                #600
                wreset = 0;
                rreset = 0; //at time 20, release reset
        end

/*****************************************************************
EVENTS
*****************************************************************/

        //output event
        always @ ( check_output ) begin
                latency[k] = $time-latency[k];
                avg_latency = avg_latency+latency[k];
                $display( " latency = %d @%d ", latency[k], k);
                if( check_val != check_mem[k] ) begin
```

```
                        all_passed = 0;
                        $display(  "  [  FAILED  ]  @%d:  rdata=%d,  expected=%d;  ",  k,
check_val, check_mem[k][TB_DSIZE-1:0]);
                end
                else if ( `VERBOSITY == 1 ) begin
                        $display(  "  [  PASSED  ]  @%d:  rdata=%d,  expected=%d;  ",  k,
check_val, check_mem[k][TB_DSIZE-1:0]);
                end
                k = k+1;
        end
endmodule
```

Appendix J – Analysis of Asynchronous FIFO

This report was created as part of a project for Princeton Course ELE 462 – Very Large Scale Integrated Circuit Design.

Performance Analysis of an Asynchronous Buffer for Crossing Clock Domains in 32nm CMOS Technology

Sam Payne, George Touloumes

Introduction

The number of cores in high-performance computing systems has continued to grow in an effort to increase computing power. However, each extra core places more stress on system networks, causing memory and inter-chip communications to become a critical bottleneck in achieving higher performance [1]. Given that the number of pins on a processor's packaging has increased more slowly than the speed of processor performance, off-chip bandwidth has become a critical obstacle to achieving high-performance in large systems. A component involved in bridging clock domains is the asynchronous buffer. For this project, we have designed an asynchronous buffer to study how a component accepting two different clocks performs and consumes power [Fig. 1]. The primary goals of this project were to create a fully functional, synthesizable Verilog design of an asynchronous FIFO and simulate it to determine functionality and power consumption in state-of-the-art 32nm CMOS component libraries. Previous work has studied the original FIFO design functionality [2], and other researchers have compared different implementations of synchronous and asynchronous FIFOs [3]. Our results provide specific analysis of how dynamic power, static power, throughput, and latency change under different read frequencies, write frequencies, and system loads.

Design

The design contains a dual port memory driven by a read and write clock. The buffer delivers messages from a write client to a read client in FIFO order. The write client provides a data signal and data valid signal, and, in return, the FIFO sends the write client a "full" signal to prevent overwriting unread values. The read client sends an enable signal, receiving a data signal and an empty signal to prevent junk values from being read. The buffer tracks the read and write locations to generate the full and empty signals. The synchronization registers are in place to mitigate the violation of setup and hold times. Since the clocks are not in sync with each other (in frequency or phase), any signal crossing the clock domain must be synchronized. The first register may go meta-stable, but the output will settle before the next clock cycle due to regenerative feedback inside the register. Thus, the second register sees a stable value (either 0 or 1). Additionally, we must convert our read and write counters to Gray Code before crossing clock domains to ensure that only one bit goes meta-stable. Otherwise, values other than the previous count or current count might be stored. Gray Coding ensures that when incrementing a number, only one bit changes at a time, guaranteeing that only one bit can be meta-stable. The drawback of using synchronization registers is that there is a two-cycle delay when transferring counters between clock domains, making the full and empty signals conservative by two cycles.

Methodology

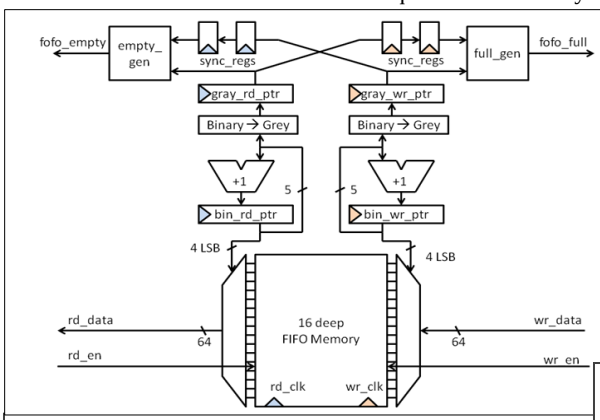
To test the buffer, we wrote a test bench to stimulate the device with different traffic densities (10-100% load) using read and write clients functioning at a variety of read and write frequencies (5 – 500 MHz). Each test wrote 255 randomly generated 64 bit values into a buffer of depth 16. Using vcs, we can check the functionality of the device under each of these conditions though an event-driven simulation. Design compiler tools can use 32nm library cells to implement this design. We used Primetime PX to analyze the power consumption of the design based on the events in the simulation and the 32nm design library. The results are discussed below.

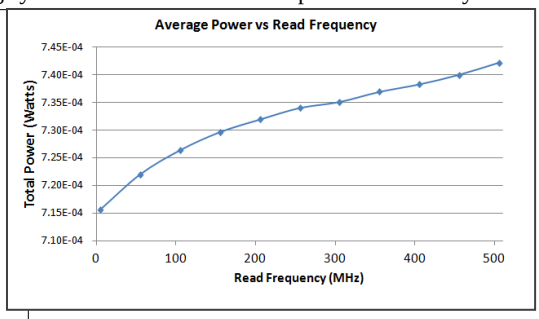
Results and Discussion

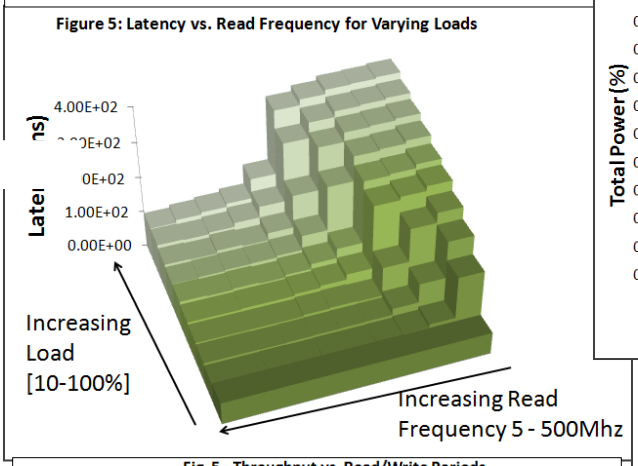
In Figure 2, the average power consumed increases in a linear trend as the read frequency increases, as we could expect from CMOS technology. (the same is true of write frequency, though not shown). Figure 4 demonstrates that as the write frequency increases, static power as a percentage of total power increases rapidly due to lower throughput and longer simulation time (the longer simulation time leads to a longer time for leakage to occur between switches, which occur less often at lower frequencies). In contrast, dynamic power increases as expected. Figure 3 is an important study for this kind of device, demonstrating how varying loads affect the latency in the FIFO. In this test, write frequency is held at a constant 50 MHz. The figure clearly illustrates which loads will fill the FIFO at different read periods, as indicated by the jump in latency (occurring at different points for each load). If possible, a designer should design a system with read frequency and average load in mind to reduce latency through the FIFO. Figure 5 illustrates how throughput changes for varying read/write frequencies. At a point, throughput becomes limited by the low write frequency, as seen by the tail at the right end of the graph. On the other hand, the differing plateaus at the left of the graph show the point at which the read frequency is limiting performance. Ideally, the read and write frequency would be close for the sake of achieving high throughput without wasting power through a needlessly high frequency on the other clock. (Many systems of course do not allow for this matching of frequency – lest we defeat the purpose of having different clock domains). Figure 6 demonstrates how the energy per bit changes for different read and write frequencies; we can see that maximum efficiency is achieved when both the read and write clocks are fast. We can also note that if one clock is slow by comparison, the efficiency suffers (as demonstrated by the rising energy sections at the right and left quadrant side of the graph).

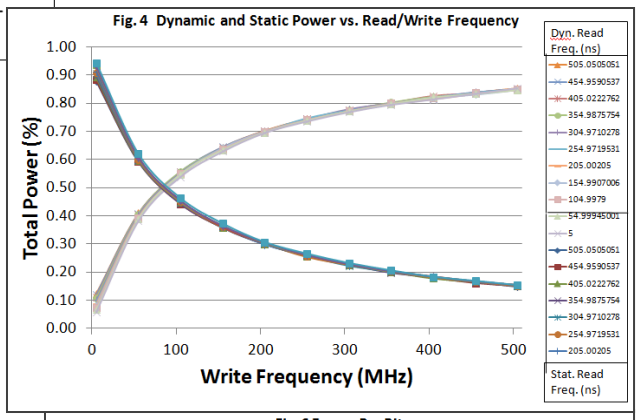
Summary

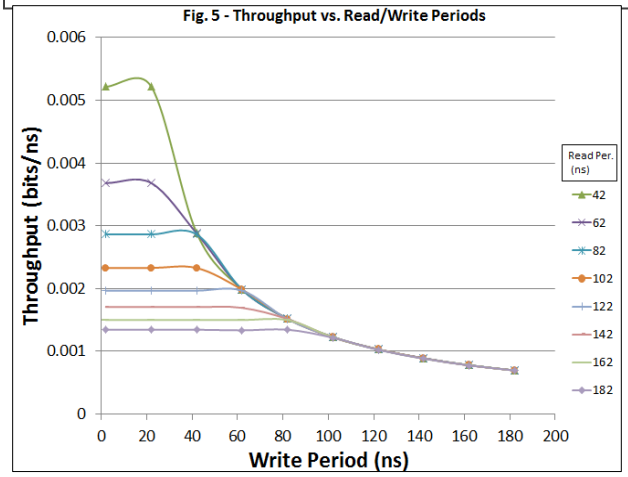
We have built and simulated a 32nm asynchronous FIFO across varying read and write clock frequencies. Overall, we characterized the trade-offs of these frequencies under varying system loads in terms of both power and latency.

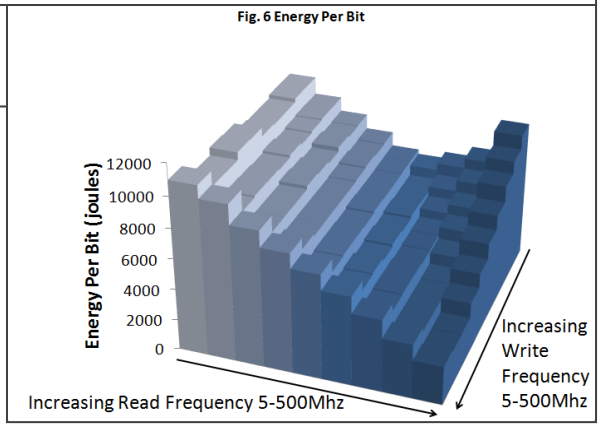












References

[1] Ahsan, Bushra, and Mohamed Zahran. "Cache performance, system performance, and off-chip bandwidth... pick any two." *Proceedings of INA-OCMC* (2009).

[2] Cummings, Clifford E. "Simulation and synthesis techniques for asynchronous FIFO design." *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*. 2002.

[3] Han, HoSuk, and Kenneth S. Stevens. "Clocked and

Appendix K – 64 Bit Wide Asynchronous FIFO Datasheet from Xilinx COREgen

```
##############################################################
#
# Xilinx Core Generator version 14.2
# Date: Thu Apr  3 00:09:02 2014
#
##############################################################
#
#  This file contains the customisation parameters for a
#  Xilinx CORE Generator IP GUI. It is strongly recommended
#  that you do not manually alter this file as it may cause
#  unexpected and unsupported behavior.
#
##############################################################
#
#  Generated from component: xilinx.com:ip:fifo_generator:9.2
#
##############################################################
#
# BEGIN Project Options
SET addpads = false
SET asysymbol = true
SET busformat = BusFormatAngleBracketNotRipped
SET createndf = false
SET designentry = Verilog
SET device = xc6vlx240t
SET devicefamily = virtex6
SET flowvendor = Other
SET formalverification = false
SET foundationsym = false
SET implementationfiletype = Ngc
SET package = ff1156
SET removerpms = false
SET simulationfiles = Structural
SET speedgrade = -2
SET verilogsim = true
SET vhdlsim = false
# END Project Options
# BEGIN Select
SELECT Fifo_Generator xilinx.com:ip:fifo_generator:9.2
# END Select
# BEGIN Parameters
CSET add_ngc_constraint_axi=false
CSET almost_empty_flag=false
CSET almost_full_flag=false
CSET aruser_width=1
CSET awuser_width=1
CSET axi_address_width=32
CSET axi_data_width=64
CSET axi_type=AXI4_Stream
CSET axis_type=FIFO
CSET buser_width=1
CSET clock_enable_type=Slave_Interface_Clock_Enable
CSET clock_type_axi=Common_Clock
CSET component_name=a_fifo_w64
CSET data_count=false
CSET data_count_width=10
CSET disable_timing_violations=false
CSET disable_timing_violations_axi=false
CSET dout_reset_value=0
CSET empty_threshold_assert_value=5
CSET empty_threshold_assert_value_axis=1022
CSET empty_threshold_assert_value_rach=1022
CSET empty_threshold_assert_value_rdch=1022
CSET empty_threshold_assert_value_wach=1022
CSET empty_threshold_assert_value_wdch=1022
CSET empty_threshold_assert_value_wrch=1022
CSET empty_threshold_negate_value=6
```

```
CSET enable_aruser=false
CSET enable_awuser=false
CSET enable_buser=false
CSET enable_common_overflow=false
CSET enable_common_underflow=false
CSET enable_data_counts_axis=false
CSET enable_data_counts_rach=false
CSET enable_data_counts_rdch=false
CSET enable_data_counts_wach=false
CSET enable_data_counts_wdch=false
CSET enable_data_counts_wrch=false
CSET enable_ecc=false
CSET enable_ecc_axis=false
CSET enable_ecc_rach=false
CSET enable_ecc_rdch=false
CSET enable_ecc_wach=false
CSET enable_ecc_wdch=false
CSET enable_ecc_wrch=false
CSET enable_read_channel=false
CSET enable_read_pointer_increment_by2=false
CSET enable_reset_synchronization=true
CSET enable_ruser=false
CSET enable_tdata=false
CSET enable_tdest=false
CSET enable_tid=false
CSET enable_tkeep=false
CSET enable_tlast=false
CSET enable_tready=true
CSET enable_tstrobe=false
CSET enable_tuser=false
CSET enable_write_channel=false
CSET enable_wuser=false
CSET fifo_application_type_axis=Data_FIFO
CSET fifo_application_type_rach=Data_FIFO
CSET fifo_application_type_rdch=Data_FIFO
CSET fifo_application_type_wach=Data_FIFO
CSET fifo_application_type_wdch=Data_FIFO
CSET fifo_application_type_wrch=Data_FIFO
CSET fifo_implementation=Independent_Clocks_Builtin_FIFO
CSET fifo_implementation_axis=Common_Clock_Block_RAM
CSET fifo_implementation_rach=Common_Clock_Block_RAM
CSET fifo_implementation_rdch=Common_Clock_Block_RAM
CSET fifo_implementation_wach=Common_Clock_Block_RAM
CSET fifo_implementation_wdch=Common_Clock_Block_RAM
CSET fifo_implementation_wrch=Common_Clock_Block_RAM
CSET full_flags_reset_value=0
CSET full_threshold_assert_value=1019
CSET full_threshold_assert_value_axis=1023
CSET full_threshold_assert_value_rach=1023
CSET full_threshold_assert_value_rdch=1023
CSET full_threshold_assert_value_wach=1023
CSET full_threshold_assert_value_wdch=1023
CSET full_threshold_assert_value_wrch=1023
CSET full_threshold_negate_value=1018
CSET id_width=4
CSET inject_dbit_error=false
CSET inject_dbit_error_axis=false
CSET inject_dbit_error_rach=false
CSET inject_dbit_error_rdch=false
CSET inject_dbit_error_wach=false
CSET inject_dbit_error_wdch=false
CSET inject_dbit_error_wrch=false
CSET inject_sbit_error=false
CSET inject_sbit_error_axis=false
CSET inject_sbit_error_rach=false
CSET inject_sbit_error_rdch=false
CSET inject_sbit_error_wach=false
CSET inject_sbit_error_wdch=false
CSET inject_sbit_error_wrch=false
CSET input_data_width=64
```

```
CSET input_depth=1024
CSET input_depth_axis=1024
CSET input_depth_rach=16
CSET input_depth_rdch=1024
CSET input_depth_wach=16
CSET input_depth_wdch=1024
CSET input_depth_wrch=16
CSET interface_type=Native
CSET output_data_width=64
CSET output_depth=1024
CSET overflow_flag=false
CSET overflow_flag_axi=false
CSET overflow_sense=Active_High
CSET overflow_sense_axi=Active_High
CSET performance_options=Standard_FIFO
CSET programmable_empty_type=No_Programmable_Empty_Threshold
CSET programmable_empty_type_axis=No_Programmable_Empty_Threshold
CSET programmable_empty_type_rach=No_Programmable_Empty_Threshold
CSET programmable_empty_type_rdch=No_Programmable_Empty_Threshold
CSET programmable_empty_type_wach=No_Programmable_Empty_Threshold
CSET programmable_empty_type_wdch=No_Programmable_Empty_Threshold
CSET programmable_empty_type_wrch=No_Programmable_Empty_Threshold
CSET programmable_full_type=No_Programmable_Full_Threshold
CSET programmable_full_type_axis=No_Programmable_Full_Threshold
CSET programmable_full_type_rach=No_Programmable_Full_Threshold
CSET programmable_full_type_rdch=No_Programmable_Full_Threshold
CSET programmable_full_type_wach=No_Programmable_Full_Threshold
CSET programmable_full_type_wdch=No_Programmable_Full_Threshold
CSET programmable_full_type_wrch=No_Programmable_Full_Threshold
CSET rach_type=FIFO
CSET rdch_type=FIFO
CSET read_clock_frequency=200
CSET read_data_count=false
CSET read_data_count_width=10
CSET register_slice_mode_axis=Fully_Registered
CSET register_slice_mode_rach=Fully_Registered
CSET register_slice_mode_rdch=Fully_Registered
CSET register_slice_mode_wach=Fully_Registered
CSET register_slice_mode_wdch=Fully_Registered
CSET register_slice_mode_wrch=Fully_Registered
CSET reset_pin=true
CSET reset_type=Asynchronous_Reset
CSET ruser_width=1
CSET synchronization_stages=2
CSET synchronization_stages_axi=2
CSET tdata_width=64
CSET tdest_width=4
CSET tid_width=8
CSET tkeep_width=4
CSET tstrb_width=4
CSET tuser_width=4
CSET underflow_flag=false
CSET underflow_flag_axi=false
CSET underflow_sense=Active_High
CSET underflow_sense_axi=Active_High
CSET use_clock_enable=false
CSET use_dout_reset=false
CSET use_embedded_registers=false
CSET use_extra_logic=false
CSET valid_flag=true
CSET valid_sense=Active_High
CSET wach_type=FIFO
CSET wdch_type=FIFO
CSET wrch_type=FIFO
CSET write_acknowledge_flag=false
CSET write_acknowledge_sense=Active_High
CSET write_clock_frequency=1000
CSET write_data_count=false
CSET write_data_count_width=10
CSET wuser_width=1
```

```
# END Parameters
# BEGIN Extra information
MISC pkg_timestamp=2012-06-23T13:35:37Z
# END Extra information
GENERATE
# CRC: 35a0dae0
```

Appendix L – Memory Controller Design Datasheet

```
CORE Generator Options:
   Target Device                  : xc6vlx240t-ff1156
   Speed Grade                    : -2
   HDL                            : verilog
   Synthesis Tool                 : Foundation_ISE

MIG Output Options:
   Module Name                    : mig_39_2_DDR3
   No of Controllers              : 1
   Selected Compatible Device(s)  : --
   Hardware Test Bench            : enabled

FPGA Options:
   Clock Type                     : Single-Ended
   Debug Port                     : OFF
   Internal Vref                  : disabled

Extended FPGA Options:
   DCI for DQ, DQS/DQS#           : enabled
   DCI for Address/Control        : disabled

/******************************************************/
/*                  Controller 0                      */
/******************************************************/
Controller Options :
   Memory                  : DDR3_SDRAM
   Interface               : NATIVE
   Design Clock Frequency  : 3300 ps (303.03 MHz)
   Memory Type             : SODIMMs
   Memory Part             : MT16JSF51264HZ-1G4
   Equivalent Part(s)      : --
   Data Width              : 64
   ECC                     : Disabled
   Data Mask               : enabled
   ORDERING                : Normal

Memory Options:
   Burst Length (MR0[1:0])          : 8 - Fixed
   Read Burst Type (MR0[3])         : Sequential
   CAS Latency (MR0[6:4])           : 5
   Output Drive Strength (MR1[5,1]) : RZQ/7
   Rtt_NOM - ODT (MR1[9,6,2])       : RZQ/4
   Rtt_WR - Dynamic ODT (MR2[10:9]) : Dynamic ODT off

Selected Banks and Pins usage :
       Data          :bank 26(40) -> Number of pins used : 35
                       bank 35(40) -> Number of pins used : 24
                       bank 36(40) -> Number of pins used : 35

       Address/Control:bank 25(40) -> Number of pins used : 33

       System Clock  :bank 34(40) -> Number of pins used : 7


       VRN/VRP       :bank 26(40) -> Number of pins used : 2
                       bank 35(40) -> Number of pins used : 2
                       bank 36(40) -> Number of pins used : 2

       VREF          :bank 25(40) -> Number of pins used : 2
                       bank 26(40) -> Number of pins used : 2
                       bank 35(40) -> Number of pins used : 2
                       bank 36(40) -> Number of pins used : 2

       BUFR          :bank 25(40) -> Number of pins used : 1
                       bank 35(40) -> Number of pins used : 1

       BUFIO         :bank 26(40) -> Number of pins used : 3
                       bank 35(40) -> Number of pins used : 2
                       bank 36(40) -> Number of pins used : 3
```

```
       Total IOs used :    150
Notes:

   1) IODELAY Power Versus Performance option (FPGA Options page) is removed
      from selection and is always set to HIGH internally from 3.8 release.
      IODELAY_HP_MODE parameter value will be always "ON" in
      MIG generated RTL.

   2) RTT (nominal) - On Die Termination (ODT) selection value of "Disabled" is
      removed from selection and one of the RTT values (RZQ/4, RZQ/2 and RZQ/6)
      should be selected. In Verify UCF and Update Design flow, MIG will set
      RTT_NOM parameter as 60 (i.e., RZQ/4) in MIG generated RTL for
      RTT value of "Disabled".
```

Appendix M – Modified Memory Controller Test Bench

Note: the majority of this testbench was generated by the Xilinx COREgen tool. Therefore, the first 870 lines have been excluded from this reference.

```
  //***************************************************************************
  // COMMAND GENERATION
  //***************************************************************************
  reg [APP_DATA_WIDTH-1:0] write_memory[TEST_SIZE*2];
  reg [ADDR_WIDTH-1:0]     write_addr[TEST_SIZE];
  integer                  latency_R[TEST_SIZE];
  integer                  latency_W[TEST_SIZE];

  //initialize traffic generators
  integer seed, icount;
  integer error;

  initial begin
    error = 0;
    seed = $time;
    for( icount = 0; icount < TEST_SIZE; icount = icount + 1) begin
      write_memory[icount*2] = {$random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed)};  //must  be  256  bits  wide,  each  random
generates 32 bits
      write_memory[icount*2+1] = {$random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed),
                                $random(seed)};  //must  be  256  bits  wide,  each  random
generates 32 bits
      write_addr[icount]   =  ($random(seed) + BEGIN_ADDRESS) % END_ADDRESS;
      latency_R[icount]      =  0;
      latency_W[icount]      =  0;
    end
  end

  //REQUEST CLIENT
  /*This client generates read and write requests:
    WRITE_NUM requests are generated at an address
    then READ_NUM requests are generated at an address followed by more write requests
    commands are sent COMMAND_LOAD percent of the time, else stalled
    The write requests at a particular address must occur before the desired read address
    therefore READ_NUM <= WRITE_NUM
   */

  integer rc_wcycle_track;
  integer rc_rcycle_track;
  integer rc_cmd_mode;
  integer rc_wcount;
  integer rc_rcount;
  integer rc_waddr_count;
  integer rc_raddr_count;
  integer rc_wdf_values;
  integer rc_wdf_stalled;
  integer rc_cmd_stalled;
  integer latency_W_count;

  reg rc_app_wdf_wren;
  reg rc_app_wdf_end;
  reg [ADDR_WIDTH-1:0] rc_app_addr;
  reg [2:0] rc_app_cmd;
```

```
reg rc_app_en;

assign app_en       = rc_app_en;
assign app_cmd      = rc_app_cmd;
assign app_addr     = rc_app_addr;
assign app_wdf_wren = rc_app_wdf_wren;
assign app_wdf_data = write_memory[rc_wcount];
assign app_wdf_end  = rc_app_wdf_end;

initial begin
  rc_app_en = 0;
  rc_app_wdf_wren = 0;
  rc_app_wdf_end = 0;
  rc_wcycle_track = 0;
  rc_rcycle_track = 0;
  rc_cmd_mode = 0; //WRITE=0, READ=1. Start writing
  rc_wcount = 0;
  rc_rcount = 0;
  rc_waddr_count = 0;
  rc_raddr_count = 0;
  rc_wdf_stalled = 0;
  rc_cmd_stalled = 0;
  rc_wdf_values = 0;
  latency_W_count = 0;
end

always@(posedge clk) begin
  if(rst || !phy_init_done) begin
    rc_wcycle_track = 0;
    rc_rcycle_track = 0;
    rc_cmd_mode = 0; //WRITE=0, READ=1. Start writing
    rc_wcount = 0;
    rc_rcount = 0;
    rc_waddr_count = 0;
    rc_raddr_count = 0;
    rc_wdf_stalled = 0;
    rc_cmd_stalled = 0;
    rc_wdf_values = 0;
  end
  else begin
    //manage switching of modes
    if(rc_wcycle_track >= WRITE_NUM && rc_cmd_mode == 0 ) begin
      rc_wcycle_track = 0;
      rc_cmd_mode = 1;
    end else if(rc_rcycle_track >= READ_NUM && rc_cmd_mode == 1) begin
      rc_rcycle_track = 0;
      rc_cmd_mode = 0;
    end

    //write data to FIFO

    if(app_wdf_rdy && rc_app_wdf_wren) begin
      if(rc_wdf_values%2 == 0) begin
        rc_wcount = rc_wcount + 1;
        rc_wdf_values = rc_wdf_values+1;
        rc_app_wdf_end = 1'b1;
      end
      else if(rc_wdf_values%2 == 1) begin
        latency_W[rc_wcount/2] = $time;
        rc_wcount = rc_wcount + 1;
        rc_wdf_values = rc_wdf_values+1;
        rc_app_wdf_end = 1'b0;
      end
    end

    //issue write commands
    if(rc_cmd_mode == 0) begin
      rc_app_cmd = 3'd0;
      if(rc_wdf_values >= 2 && !rc_cmd_stalled) begin //data can be issued a command on
        rc_app_addr = write_addr[rc_waddr_count] << 6;
```

```
        if({$random(seed)}%100 <= COMMAND_LOAD) begin
          rc_app_en = 1'b1;
          rc_cmd_stalled = 1;
        end
      end
      else if(app_rdy && rc_app_en ) begin
        latency_W[latency_W_count] = $time-latency_W[latency_W_count];
        latency_W_count=latency_W_count+1;
        rc_wdf_values = rc_wdf_values-2;
        rc_waddr_count = rc_waddr_count+1;
        rc_wcycle_track = rc_wcycle_track+1;
        rc_cmd_stalled = 0;
        rc_app_en = 1'b0;
      end
      else if (rc_cmd_stalled) begin
        if({$random(seed)}%100 <= COMMAND_LOAD) begin
          rc_app_en = 1'b1;
        end
        else begin
          rc_app_en = 1'b0;
        end
      end
    end


    //issue read commands
    if(rc_cmd_mode == 1) begin
      rc_app_cmd = 3'd1;
      if(!rc_cmd_stalled) begin
        rc_app_addr = write_addr[rc_raddr_count] << 6;
        if({$random(seed)}%100 <= COMMAND_LOAD) begin
          rc_cmd_stalled = 1;
          rc_app_en = 1'b1;
        end
      end
      else if(app_rdy && rc_cmd_stalled && rc_app_en) begin
        latency_R[rc_raddr_count] = $time;
        rc_raddr_count = rc_raddr_count+1;
        rc_rcycle_track = rc_rcycle_track+1;
        rc_cmd_stalled = 0;
        rc_app_en = 1'b0;
      end
    end

    if({$random(seed)}%100 <= WRITE_LOAD && rc_wcount < TEST_SIZE*2) begin
        rc_app_wdf_wren = 1'b1;
    end
    else begin
        rc_app_wdf_wren = 1'b0;
    end


  end
end //always



//READ CLIENT
integer read_count;
integer latency_R_count;
reg [APP_DATA_WIDTH-1:0] read_value;


initial begin
read_count = 0;
latency_R_count = 0;
read_value = 0;
end

always @(posedge clk) begin
```

```
    if(rst || !phy_init_done) begin
      read_count = 0;
      latency_R_count = 0;
      read_value = 0;
    end
    else begin
      if(app_rd_data_valid) begin
        read_value = app_rd_data;
        if(read_value != write_memory[read_count])
          error=1;
        read_count = read_count+1;
      end
      if(app_rd_data_end) begin
        latency_R[latency_R_count] = $time-latency_R[latency_R_count];
        latency_R_count = latency_R_count+1;
      end

    end
  end

  assign app_wdf_mask         = {APP_MASK_WIDTH{1'b0}};

  //***************************************************************************
  // Reporting the test case status
  //***************************************************************************
//integer average_latency;
  initial
  begin : Logging
     fork
        begin : calibration_done
           wait (phy_init_done);
           $display("Calibration Done");
           wait (latency_R_count == TEST_SIZE - 1);
           latency_R[latency_R_count] = $time-latency_R[latency_R_count];
           for( icount = 0; icount < TEST_SIZE; icount=icount+1) begin
             $display(    "    LATENCY_R:    %d    LATENCY_W:    %d",    latency_R[icount],
latency_W[icount]);
           end
           if (!error) begin
              $display("TEST PASSED");
           end
           else begin
              $display("TEST FAILED: DATA ERROR");
           end
           disable calib_not_done;
            $dumpoff;
            $finish;
        end

        begin : calib_not_done
           #10000000000;
           if (!phy_init_done) begin
              $display("TEST FAILED: INITIALIZATION DID NOT COMPLETE");
           end
           disable calibration_done;
             $dumpoff;
             $finish;
        end
     join
  end
endmodule
```

Appendix N – Packet Translation Module

This module is described in the comment below:

```
/*****************************************************************
 * Author: Sam Payne
 *
 * Packet translation module capable of translating network requests
 * into commands for the memory controller. The translator also
 * creates outgoing packets at acknowledgments to requesters.
 *****************************************************************/


module pkt_trans_dp
# (
  parameter MAX_PKT_LEN         = 11, //measured in flits
  parameter MAX_PKT_LEN_LOG     = 4,
  parameter ADDR_WIDTH          = 29, //size of addr to MC
  parameter MEMORY_WIDTH        = 8,
  parameter DATA_WIDTH          = 64,
  parameter LOC_ADDR_HI         = 45,
  parameter LOC_ADDR_LO         = 22,
  parameter LOC_ADDR_SIZE       = 23, //512 MB, addressed by 64 byte cache lines

  parameter FLIT_SIZE           = 64, //Size of one flit

  parameter FLT_ADDR_HI         = 63,
  parameter FLT_ADDR_LO         = 16,
  parameter FLT_ADDR_SIZE       = 48,

  parameter FLT_CHIPID_HI       = 63,
  parameter FLT_CHIPID_LO       = 50,
  parameter FLT_CHIPID_SIZE     = 14,

  parameter FLT_XPOS_HI         =49,
  parameter FLT_XPOS_LO         =42,
  parameter FLT_XPOS_SIZE       =8,

  parameter FLT_YPOS_HI         =41,
  parameter FLT_YPOS_LO         =34,
  parameter FLT_YPOS_SIZE       =8,

  parameter FLT_FBITS_HI        =33,
  parameter FLT_FBITS_LO        =30,
  parameter FLT_FBITS_SIZE      =4,

  parameter FLT_PAYLOADL_HI     =29,
  parameter FLT_PAYLOADL_LO     =22,
  parameter FLT_PAYLOADL_SIZE   =8,

  parameter FLT_MSG_TYPE_HI     =21,
  parameter FLT_MSG_TYPE_LO     =14,
  parameter FLT_MSG_TYPE_SIZE   =8,

  parameter FLT_TAG_HI          =13,
  parameter FLT_TAG_LO          =6,
  parameter FLT_TAG_SIZE        =8,

  parameter FLT_OPT_HI          =15,
  parameter FLT_OPT_LO          =0,
  parameter FLT_OPT_SIZE        =16,

  parameter FLT_SRC_CHIPID_HI   =63,
  parameter FLT_SRC_CHIPID_LO   =50,
  parameter FLT_SRC_CHIPID_SIZE =14,

  parameter FLT_SRC_XPOS_HI        =49,
  parameter FLT_SRC_XPOS_LO        =42,
  parameter FLT_SRC_XPOS_SIZE      =8,

  parameter FLT_SRC_YPOS_HI        =41,
  parameter FLT_SRC_YPOS_LO        =34,
```

```
  parameter FLT_SRC_YPOS_SIZE      =8,

  parameter FLT_SRC_FBITS_HI    =33,
  parameter FLT_SRC_FBITS_LO    =30,
  parameter FLT_SRC_FBITS_SIZE  =4,

//message types
  parameter MSG_TYPE_NC_LOAD_REQ = 8'd14,
  parameter MSG_TYPE_NC_STORE_REQ = 8'd15,
  parameter MSG_TYPE_LOAD_MEM = 8'd19,
  parameter MSG_TYPE_STORE_MEM = 8'd20,

  parameter MSG_TYPE_NC_LOAD_MEM_ACK = 8'd26,
  parameter MSG_TYPE_NC_STORE_MEM_ACK = 8'd27,
  parameter MSG_TYPE_LOAD_MEM_ACK = 8'd24,
  parameter MSG_TYPE_STORE_MEM_ACK = 8'd25,

  parameter IN_FLIGHT_LIMIT   = 16, //number of commands the MC can have in flight
  parameter BUFFER_ADDR_SIZE  = 4 //(log_2(IN_FLIGHT_LIMIT)+1)
)(
clk,
rst,
flit_in,
flit_in_val,
flit_in_rdy,
flit_out,
flit_out_val,
flit_out_rdy,
app_rdy,
app_wdf_rdy,
app_rd_data,
app_rd_data_end,
app_rd_data_valid,
phy_init_done,
app_wdf_wren,
app_wdf_data,
app_wdf_mask,
app_wdf_end,
app_addr,
app_en,
app_cmd
);

`timescale 1ps/100fs

//packet parser states
`define ACCEPT_W1   0
`define ACCEPT_W2   1
`define ACCEPT_W3   2
`define ACCEPT_DATA 3
`define STALLED     4

//buffer item states
`define FILLING      0
`define WAITING_WDF  1
`define WAITING_CMD  2
`define WAITING_DATA 3
`define READY        4
`define INACTIVE     5

`define FIRST       0
`define SECOND      1

`define ADDR_ONE {{BUFFER_ADDR_SIZE-1{1'b0}}, {1'b1}}

localparam APP_DATA_WIDTH     = DATA_WIDTH * 4; //one cache line, one write request
localparam APP_MASK_WIDTH     = APP_DATA_WIDTH / 8;

input clk;
input rst;
```

```
//System Network Interface
input  [FLIT_SIZE-1:0] flit_in;
input  flit_in_val;
output flit_in_rdy;

output [FLIT_SIZE] flit_out;
output flit_out_val;
input  flit_out_rdy;

//MC User Interface
input                          app_rdy;
input                          app_wdf_rdy;
input [APP_DATA_WIDTH-1:0]     app_rd_data;
input                          app_rd_data_end;
input                          app_rd_data_valid;
input                          phy_init_done;

output                         app_wdf_wren;
output [APP_DATA_WIDTH-1:0]    app_wdf_data;
output [APP_MASK_WIDTH-1:0]    app_wdf_mask;
output                         app_wdf_end;
output [ADDR_WIDTH-1:0]        app_addr;
output                         app_en;
output [2:0]                   app_cmd;

// global buffers
reg [FLIT_SIZE-1:0]            pkt_w1        [IN_FLIGHT_LIMIT-1:0];
reg [FLIT_SIZE-1:0]            pkt_w2        [IN_FLIGHT_LIMIT-1:0];
reg [FLIT_SIZE-1:0]            pkt_w3        [IN_FLIGHT_LIMIT-1:0];
reg [(APP_DATA_WIDTH*2)-1:0]   pkt_data_buf  [IN_FLIGHT_LIMIT-1:0];
reg [2:0]                      pkt_state_buf [IN_FLIGHT_LIMIT-1:0];
reg [FLT_MSG_TYPE_SIZE-1:0]    pkt_cmd_buf   [IN_FLIGHT_LIMIT-1:0];

//*******************************************************
// ACCEPT PACKETS
//*******************************************************
reg [FLIT_SIZE-1:0]          in_data_buf[MAX_PKT_LEN-4:0]; //buffer for incomming packets
reg [BUFFER_ADDR_SIZE-1:0]   buf_current_in;  //address of buffer slot being filled
reg [BUFFER_ADDR_SIZE-1:0]   buf_current_out; //address of buffer slot being sent
reg [MAX_PKT_LEN_LOG-1:0]    remaining_flits; //flits remaining in current packet
reg [2:0]                    acc_state;

//debug signals
wire [2:0] dbg_state_in = pkt_state_buf[buf_current_in];
wire [2:0] dbg_state_0 = pkt_state_buf[0];
wire [2:0] dbg_state_1 = pkt_state_buf[1];
wire [2:0] dbg_state_2 = pkt_state_buf[2];
wire [2:0] dbg_state_3 = pkt_state_buf[3];
wire [2:0] dbg_state_4 = pkt_state_buf[4];
wire [2:0] dbg_state_11 = pkt_state_buf[11];
wire [2:0] dbg_state_12 = pkt_state_buf[12];
wire [2:0] dbg_state_13 = pkt_state_buf[13];
wire [2:0] dbg_state_14 = pkt_state_buf[14];
wire [2:0] dbg_state_15 = pkt_state_buf[15];

wire [BUFFER_ADDR_SIZE-1:0] dbg_in_next = buf_current_in+`ADDR_ONE;

//assignments
assign flit_in_rdy  = (acc_state != `STALLED && !rst && phy_init_done);

always @(posedge clk) begin
    if(rst || !phy_init_done) begin
        //initialize buffer pointers
        buf_current_in <= 0;
        remaining_flits <= 0;
        acc_state = `ACCEPT_W1;
        //initialize all buffers to zero
        for(int i=0; i < IN_FLIGHT_LIMIT; i=i+1) begin
            pkt_w1        [i] <= 0;
```

```
            pkt_w2        [i] <= 0;
            pkt_w3        [i] <= 0;
            pkt_data_buf  [i] <= 0;
            pkt_cmd_buf   [i] <= 0;
            pkt_state_buf [i] <= `INACTIVE; //initialize state to inactive
        end
        for(int i=0; i < MAX_PKT_LEN-3; i++) begin
            in_data_buf[i] <= 0;
        end
    end
    else begin
        if(flit_in_val) begin
            case (acc_state) //determine which word of the packet we are accepting
            //############### Accepting Head Flit ###################
            `ACCEPT_W1: begin
                pkt_state_buf [buf_current_in] <= `FILLING;
                pkt_w1        [buf_current_in] <= flit_in;
                pkt_cmd_buf             [buf_current_in]    <=    flit_in[FLT_MSG_TYPE_HI:
FLT_MSG_TYPE_LO];
                remaining_flits <= flit_in[FLT_PAYLOADL_HI:FLT_PAYLOADL_LO]-1;
                acc_state       <= `ACCEPT_W2;
            end

            //############### Accepting Addr Flit ###################
            `ACCEPT_W2: begin
                pkt_w2        [buf_current_in] <= flit_in;
                remaining_flits <= remaining_flits-1;
                acc_state <= `ACCEPT_W3;
            end
            //############### Accepting Src Flits ###################
            `ACCEPT_W3: begin
                pkt_w3        [buf_current_in] <= flit_in;

                if(remaining_flits == 0) begin //read command
                    //check if we can start accepting the next command
                    if(buf_current_in+`ADDR_ONE == buf_current_out) begin
                        buf_current_in <= buf_current_in + 1;
                        pkt_state_buf[buf_current_in] <= `WAITING_CMD;
                        acc_state <= `STALLED;
                    end
                    else begin
                        buf_current_in <= buf_current_in + 1;
                        pkt_state_buf[buf_current_in] <= `WAITING_CMD;
                        acc_state <= `ACCEPT_W1;
                    end
                end
                else begin //write command
                    remaining_flits <= remaining_flits-1;
                    acc_state <= `ACCEPT_DATA;
                end
            end
            //############### Accepting Data Flits ###################
            `ACCEPT_DATA: begin
                in_data_buf[remaining_flits] <= flit_in;
                remaining_flits <= remaining_flits - 1;
                //check if we are done accepting flits
                if((remaining_flits == 0)) begin
                    buf_current_in <= buf_current_in+1;
                    pkt_state_buf[buf_current_in] <= `WAITING_WDF;

                    pkt_data_buf[buf_current_in] <= {in_data_buf[0],
                                                     in_data_buf[1],
                                                     in_data_buf[2],
                                                     in_data_buf[3],
                                                     in_data_buf[4],
                                                     in_data_buf[5],
                                                     in_data_buf[6],
                                                     in_data_buf[7]};
                    //we are ready for the next command
                    if (buf_current_in+`ADDR_ONE != buf_current_out) begin
```

```
                        acc_state <= `ACCEPT_W1;
                    end
                    else begin
                        acc_state <= `STALLED;
                    end
                end
            end
            //#################### Stalled ######################
            `STALLED: begin //waiting for buffer space to be freed
                if(buf_current_in != buf_current_out) begin
                    pkt_state_buf[buf_current_in] <= `FILLING;
                    acc_state <= `ACCEPT_W1;
                end
            end
        endcase
    end
  end
end

//*****************************************************
// MC DATA SEND
//*****************************************************
reg [BUFFER_ADDR_SIZE-1:0] buf_current_wdf; //tracks the current data sender to MC
reg buf_wdf_data_half;  //which half of the data we are writing;
reg r_app_wdf_wren;     //write_enable

wire [2:0] dbg_state_wdf = pkt_state_buf[buf_current_wdf];
wire [APP_DATA_WIDTH*2-1:0] dbg_data_wdf = pkt_data_buf[buf_current_wdf];

assign app_wdf_wren = (pkt_state_buf[buf_current_wdf] == `WAITING_WDF) ? 1 : 0;

assign app_wdf_data = (buf_wdf_data_half == `FIRST)                    ?
                      pkt_data_buf[buf_current_wdf][APP_DATA_WIDTH*2-1:APP_DATA_WIDTH]  :
                      pkt_data_buf[buf_current_wdf][APP_DATA_WIDTH-1:0];
assign app_wdf_mask = 0;
assign app_wdf_end  = (buf_wdf_data_half == `SECOND);

always@(posedge clk) begin
  if(rst || !phy_init_done) begin
    buf_current_wdf <= 0;
    buf_wdf_data_half <= `FIRST;
    r_app_wdf_wren <= 0;
  end
  else begin
    if( (pkt_cmd_buf[buf_current_wdf] == MSG_TYPE_NC_LOAD_REQ ||
         pkt_cmd_buf[buf_current_wdf] == MSG_TYPE_LOAD_MEM) &&
        (pkt_state_buf[buf_current_wdf] != `INACTIVE) )
    begin  //no need to write data for read commands
      buf_current_wdf <= buf_current_wdf + 1;
    end
    r_app_wdf_wren <= (pkt_state_buf[buf_current_wdf] == `WAITING_WDF) ? 1 : 0;
    if(app_wdf_wren && app_wdf_rdy) begin
      buf_wdf_data_half <= ~buf_wdf_data_half;              //present first or second 32
bytes
      if(buf_wdf_data_half == `SECOND) begin
        pkt_state_buf[buf_current_wdf] <= `WAITING_CMD;
        buf_current_wdf <= buf_current_wdf + 1;
      end
    end
  end
end

//*****************************************************
// MC CMD SEND
//*****************************************************
reg [BUFFER_ADDR_SIZE-1:0] buf_current_cmd; //tracks current command sender to MC
reg r_app_en;                               //command enable

assign app_en = r_app_en;
assign app_addr = pkt_w2[buf_current_cmd][LOC_ADDR_HI: LOC_ADDR_LO];
```

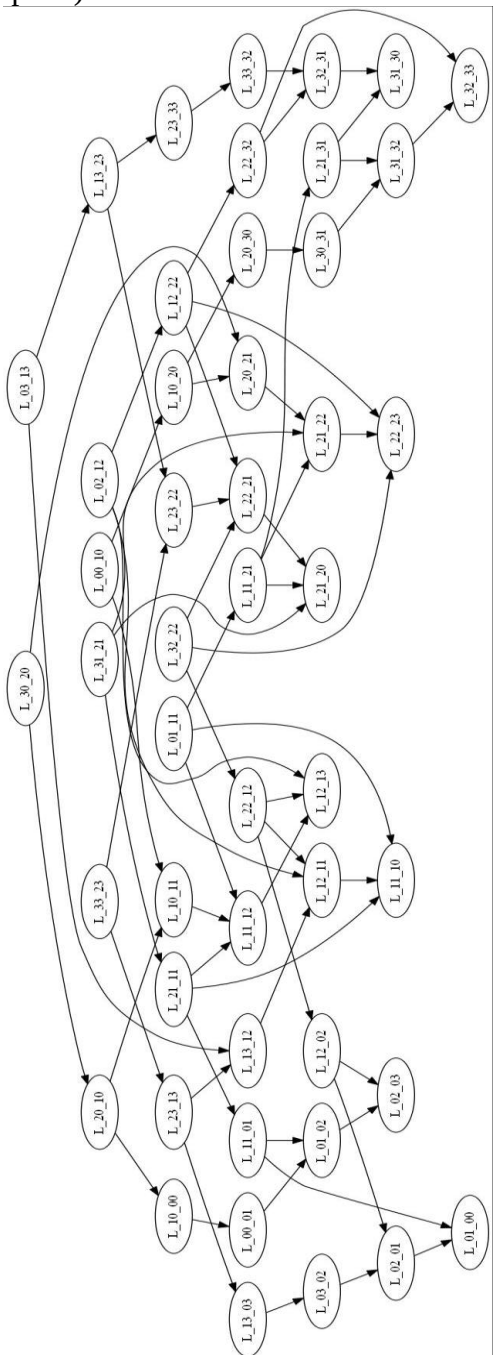
```
assign app_cmd = (pkt_cmd_buf[buf_current_cmd] == MSG_TYPE_NC_STORE_REQ ||
                  pkt_cmd_buf[buf_current_cmd] == MSG_TYPE_STORE_MEM) ? 3'b000 : 3'b001;

always @(posedge clk) begin
  if(rst || !phy_init_done) begin
    buf_current_cmd <= 0;
    r_app_en <= 0;
  end
  else begin
    if(pkt_state_buf[buf_current_cmd] == `WAITING_CMD) begin
      r_app_en <= 1;
    end

    if (app_en && app_rdy) begin
      r_app_en <= 0;
      if(pkt_cmd_buf[buf_current_cmd] == MSG_TYPE_NC_STORE_REQ ||
          pkt_cmd_buf[buf_current_cmd] == MSG_TYPE_STORE_MEM) begin
        pkt_state_buf[buf_current_cmd] <= `READY;
      end
      else begin
        pkt_state_buf[buf_current_cmd] <= `WAITING_DATA;
      end
      buf_current_cmd <= buf_current_cmd+1;
    end

  end

end

//*********************************************************
// MC DATA RCV
//*********************************************************
reg [BUFFER_ADDR_SIZE-1:0] buf_current_data_rcv;
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_read_data_cmd = pkt_cmd_buf[buf_current_data_rcv];

//debug commands
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_cmd_0 = pkt_cmd_buf[0];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_cmd_1 = pkt_cmd_buf[1];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_cmd_2 = pkt_cmd_buf[2];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_cmd_3 = pkt_cmd_buf[3];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_cmd_4 = pkt_cmd_buf[4];

always @(posedge clk) begin
    if(rst || !phy_init_done) begin
        buf_current_data_rcv <= 0;
    end
  else begin
    if( (pkt_cmd_buf[buf_current_data_rcv] == MSG_TYPE_NC_STORE_REQ ||
          pkt_cmd_buf[buf_current_data_rcv] == MSG_TYPE_STORE_MEM) &&
         (pkt_state_buf[buf_current_data_rcv][2:0] != `INACTIVE))
    begin //no need to receive data for write commands
      buf_current_data_rcv <= buf_current_data_rcv+1;
    end

    if(app_rd_data_valid) begin
      if(!app_rd_data_end) begin
        pkt_data_buf[buf_current_data_rcv][APP_DATA_WIDTH-1:0] <= app_rd_data;
      end
      else if (app_rd_data_end) begin
        pkt_data_buf[buf_current_data_rcv][APP_DATA_WIDTH*2-1:APP_DATA_WIDTH]           <=
app_rd_data;
        pkt_state_buf[buf_current_data_rcv] <= `READY;
        buf_current_data_rcv <= buf_current_data_rcv+1;
      end
    end
  end
end

//*********************************************************
// SEND PACKETS
```

```
//******************************************************
reg [MAX_PKT_LEN_LOG-1:0]    remaining_flt_out;
reg [FLIT_SIZE-1:0]          flit_out_buffer[MAX_PKT_LEN];

assign flit_out = flit_out_buffer[remaining_flt_out];
assign flit_out_val = pkt_state_buf[buf_current_out] == `READY;

wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_0 = pkt_cmd_buf[0];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_1 = pkt_cmd_buf[1];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_2 = pkt_cmd_buf[2];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_3 = pkt_cmd_buf[3];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_4 = pkt_cmd_buf[4];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_5 = pkt_cmd_buf[5];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_6 = pkt_cmd_buf[6];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_7 = pkt_cmd_buf[7];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_8 = pkt_cmd_buf[8];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_9 = pkt_cmd_buf[9];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_10 = pkt_cmd_buf[10];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_11 = pkt_cmd_buf[11];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_12 = pkt_cmd_buf[12];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_13 = pkt_cmd_buf[13];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_14 = pkt_cmd_buf[14];
wire [FLT_MSG_TYPE_SIZE-1:0] dbg_current_cmd_15 = pkt_cmd_buf[15];

always @(posedge clk) begin
  if(rst || !phy_init_done) begin
    buf_current_out <= IN_FLIGHT_LIMIT-1; //initialize current out
    remaining_flt_out <= 0;
    for(int i=0; i < MAX_PKT_LEN; i=i+1) begin
        flit_out_buffer[i] <= 64'h0;
    end
  end
  else begin
    if(pkt_state_buf[buf_current_out] == `READY ||
       pkt_state_buf[buf_current_out] == `INACTIVE ) begin
      if (remaining_flt_out == 0) begin
        //deactivate current buffer entry
        pkt_state_buf[buf_current_out] <= `INACTIVE;
        //load next buffer entry if it is ready
        if(pkt_state_buf[buf_current_out+`ADDR_ONE] == `READY ) begin
            buf_current_out <= buf_current_out+`ADDR_ONE;

            //load response - data is attached
            if( pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_NC_LOAD_REQ ||
                pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_LOAD_MEM
                ) begin
                remaining_flt_out <= MAX_PKT_LEN-2;

                //initialize flit_out header
                flit_out_buffer[MAX_PKT_LEN-2][FLT_CHIPID_HI:FLT_CHIPID_LO]             <=
pkt_w1[buf_current_out+`ADDR_ONE][FLT_SRC_CHIPID_HI:FLT_SRC_CHIPID_LO];
                flit_out_buffer[MAX_PKT_LEN-2][FLT_XPOS_HI       :FLT_XPOS_LO       ]    <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_XPOS_HI  :FLT_SRC_XPOS_LO  ];
                flit_out_buffer[MAX_PKT_LEN-2][FLT_YPOS_HI       :FLT_YPOS_LO       ]    <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_YPOS_HI  :FLT_SRC_YPOS_LO  ];
                flit_out_buffer[MAX_PKT_LEN-2][FLT_FBITS_HI      :FLT_FBITS_LO      ]    <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_FBITS_HI :FLT_SRC_FBITS_LO ];
                flit_out_buffer[MAX_PKT_LEN-2][FLT_TAG_HI        :FLT_TAG_LO        ]    <=
pkt_w1[buf_current_out+`ADDR_ONE][FLT_TAG_HI       :FLT_TAG_LO       ];
                flit_out_buffer[MAX_PKT_LEN-2][FLT_PAYLOADL_HI:FLT_PAYLOADL_LO]          =
MAX_PKT_LEN-3;

                //determine return message type
                if(pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_NC_LOAD_REQ) begin
                    flit_out_buffer[MAX_PKT_LEN-2][FLT_MSG_TYPE_HI:FLT_MSG_TYPE_LO]
=MSG_TYPE_NC_LOAD_MEM_ACK;
                end
                if(pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_NC_STORE_REQ) begin
                    flit_out_buffer[MAX_PKT_LEN-2][FLT_MSG_TYPE_HI:FLT_MSG_TYPE_LO]
=MSG_TYPE_NC_STORE_MEM_ACK;
```

```
                end

                //initialize return data
                flit_out_buffer[MAX_PKT_LEN-3]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((8)*FLIT_SIZE)-1:(7)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-4]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((7)*FLIT_SIZE)-1:(6)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-5]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((6)*FLIT_SIZE)-1:(5)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-6]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((5)*FLIT_SIZE)-1:(4)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-7]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((4)*FLIT_SIZE)-1:(3)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-8]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((3)*FLIT_SIZE)-1:(2)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-9]                                          <=
pkt_data_buf[buf_current_out+`ADDR_ONE][((2)*FLIT_SIZE)-1:(1)*FLIT_SIZE];
                flit_out_buffer[MAX_PKT_LEN-10]<=
pkt_data_buf[buf_current_out+`ADDR_ONE][((1)*FLIT_SIZE)-1:(0)*FLIT_SIZE];
            end
            else begin //store command
                flit_out_buffer[remaining_flt_out+1][FLT_PAYLOADL_HI:FLT_PAYLOADL_LO]    =
0; //no data to return
                remaining_flt_out <= 0;

                flit_out_buffer[0][FLT_CHIPID_HI:FLT_CHIPID_LO]                         <=
pkt_w1[buf_current_out+`ADDR_ONE][FLT_SRC_CHIPID_HI:FLT_SRC_CHIPID_LO];
                flit_out_buffer[0][FLT_XPOS_HI            :FLT_XPOS_LO            ]     <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_XPOS_HI  :FLT_SRC_XPOS_LO  ];
                flit_out_buffer[0][FLT_YPOS_HI            :FLT_YPOS_LO            ]     <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_YPOS_HI  :FLT_SRC_YPOS_LO  ];
                flit_out_buffer[0][FLT_FBITS_HI         :FLT_FBITS_LO         ]         <=
pkt_w3[buf_current_out+`ADDR_ONE][FLT_SRC_FBITS_HI :FLT_SRC_FBITS_LO ];
                flit_out_buffer[0][FLT_TAG_HI             :FLT_TAG_LO             ]   <=
pkt_w1[buf_current_out+`ADDR_ONE][FLT_TAG_HI       :FLT_TAG_LO       ];

                if(pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_LOAD_MEM) begin
                    flit_out_buffer[0][FLT_MSG_TYPE_HI:FLT_MSG_TYPE_LO]
=MSG_TYPE_LOAD_MEM_ACK;
                end
                if(pkt_cmd_buf[buf_current_out+`ADDR_ONE] == MSG_TYPE_STORE_MEM) begin
                    flit_out_buffer[0][FLT_MSG_TYPE_HI:FLT_MSG_TYPE_LO]
=MSG_TYPE_STORE_MEM_ACK;
                end
            end
          end
      end
      else begin
        if(flit_out_rdy && flit_out_val) begin
          remaining_flt_out <= remaining_flt_out-1;
        end
      end
    end
  end
end
endmodule
```

Appendix O – DAG illustration of the 2-D Mesh Chip Network Structure

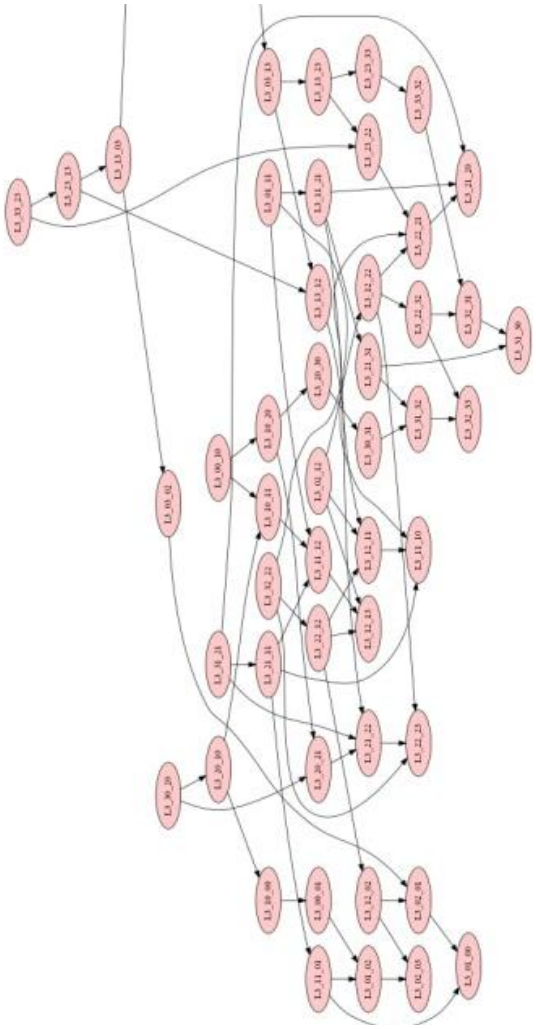
The following DAG illustrates that a 2-Dimensional mesh using Dimension Order routing is deadlock free. The mesh is a 4x4 structure and each link denotes the coordinates of the nodes which it connects in order in the following manner: L_XY_XY, where the first X, Y pair is the coordinate of the origin node and the second X, Y pair is the destination node. All possible paths through the system can be traced from top to bottom and all routing nodes (XY pairs) are reachable from all other nodes.

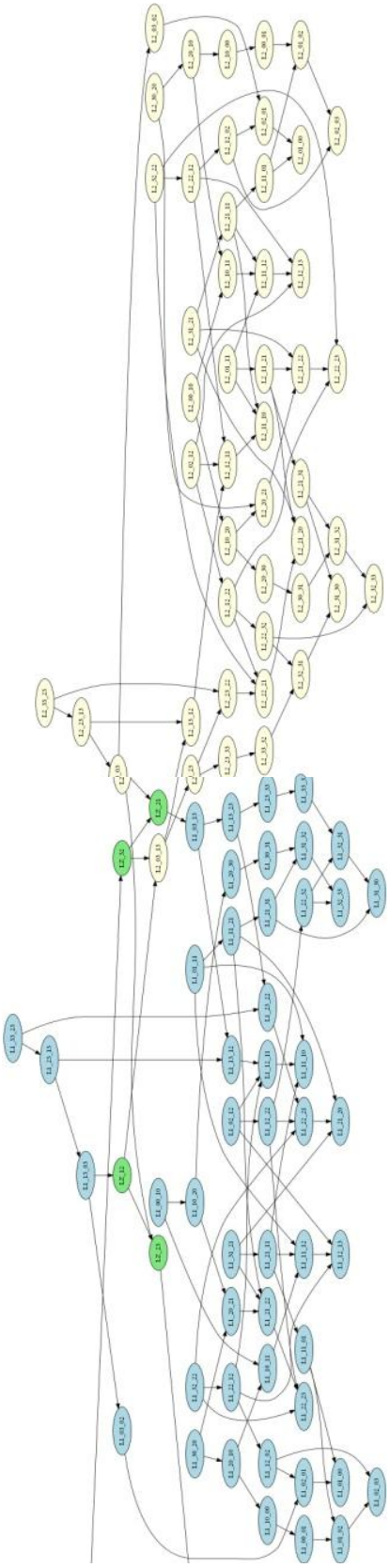


Appendix P – Link Dependency Digraph for a Hierarchical Network Containing Three 2-D Meshes

The following DAG illustrates that a 2-Dimensional mesh using Dimension Order routing connected in a linear array is deadlock free. Each mesh is a 4x4 structure and each link denotes the coordinates of the nodes which it connects in order in the following manner: L_XY_XY, where the first X, Y pair is the coordinate of the origin node and the second X, Y pair is the destination node. The three meshes' links are painted red, blue and yellow and links joining the networks are colored in green. All possible paths through the system can be traced from top to bottom and all routing nodes (XY pairs) are reachable from all other nodes.

This graph is very long and so it is divided into two halves. The second half is on the following page.





Appendix Q – Dot Graph Used to Generate DAGs

The dependency graph in dot format is supplied here for the reader to understand the structure of this digraph and to create their own visual representation of necessary. It is presented here in two column format for the sake of saving space.

```
digraph globaL1_mesh {

    //style
    LZ_12, LZ_21, LZ_23, LZ_32
             [style = filled, fillcolor =
"#80E680"];

    //interconnect links
    L2_13_03 -> LZ_21 -> L1_03_13
    L1_13_03 -> LZ_12 -> L2_03_13
    L3_13_03 -> LZ_32 -> L2_03_13
    L2_13_03 -> LZ_23 -> L3_03_13
    LZ_12 -> LZ_23
    LZ_32 -> LZ_21;

/**********************MESH 1
***********************/
       //style
    L1_00_01, L1_01_02, L1_02_03,
    L1_10_11, L1_11_12, L1_12_13,
    L1_20_21, L1_21_22, L1_22_23,
    L1_30_31, L1_31_32, L1_32_33,
    L1_01_00, L1_02_01, L1_03_02,
    L1_11_10, L1_12_11, L1_13_12,
    L1_21_20, L1_22_21, L1_23_22,
    L1_31_30, L1_32_31, L1_33_32,
    L1_00_10, L1_01_11, L1_02_12,
L1_03_13,
    L1_10_20, L1_11_21, L1_12_22,
L1_13_23,
    L1_20_30, L1_21_31, L1_22_32,
L1_23_33,
    L1_10_00, L1_11_01, L1_12_02,
L1_13_03,
    L1_20_10, L1_21_11, L1_22_12,
L1_23_13,
    L1_30_20, L1_31_21, L1_32_22,
L1_33_23
         [style = filled, fillcolor =
lightblue];

    //vertical lines
       L1_00_01 -> L1_01_02 -> L1_02_03
       L1_10_11 -> L1_11_12 -> L1_12_13
       L1_20_21 -> L1_21_22 -> L1_22_23
       L1_30_31 -> L1_31_32 -> L1_32_33
    L1_03_02 -> L1_02_01 -> L1_01_00
    L1_13_12 -> L1_12_11 -> L1_11_10
    L1_23_22 -> L1_22_21 -> L1_21_20
    L1_33_32 -> L1_32_31 -> L1_31_30

    //horizontal lines
    L1_03_13 -> L1_13_23 -> L1_23_33
    L1_02_12 -> L1_12_22 -> L1_22_32
    L1_01_11 -> L1_11_21 -> L1_21_31
    L1_00_10 -> L1_10_20 -> L1_20_30
    L1_33_23 -> L1_23_13 -> L1_13_03
    L1_32_22 -> L1_22_12 -> L1_12_02
    L1_31_21 -> L1_21_11 -> L1_11_01
    L1_30_20 -> L1_20_10 -> L1_10_00

    //X+ Y+ turns
    L1_00_10 -> L1_10_11
    L1_10_20 -> L1_20_21
    L1_20_30 -> L1_30_31
    L1_01_11 -> L1_11_12
    L1_11_21 -> L1_21_22
    L1_21_31 -> L1_31_32
    L1_02_12 -> L1_12_13
    L1_12_22 -> L1_22_23
    L1_22_32 -> L1_32_33

    //X+ Y- turns
    L1_03_13 -> L1_13_12
    L1_02_12 -> L1_12_11
    L1_01_11 -> L1_11_10

    L1_13_23 -> L1_23_22
    L1_12_22 -> L1_22_21
    L1_11_21 -> L1_21_20
    L1_23_33 -> L1_33_32
    L1_22_32 -> L1_32_31
    L1_21_31 -> L1_31_30

    //X- Y+ turns
    L1_10_00 -> L1_00_01
    L1_11_01 -> L1_01_02
    L1_12_02 -> L1_02_03
    L1_20_10 -> L1_10_11
    L1_21_11 -> L1_11_12
    L1_22_12 -> L1_12_13
    L1_30_20 -> L1_20_21
    L1_31_21 -> L1_21_22
    L1_32_22 -> L1_22_23

    //X- Y- turns
    L1_33_23 -> L1_23_22
    L1_23_13 -> L1_13_12
    L1_13_03 -> L1_03_02
    L1_32_22 -> L1_22_21
    L1_22_12 -> L1_12_11
    L1_12_02 -> L1_02_01
    L1_31_21 -> L1_21_20
    L1_21_11 -> L1_11_10
    L1_11_01 -> L1_01_00

/**********************MESH 2
***********************/
    //style
    L2_00_01, L2_01_02, L2_02_03,
    L2_10_11, L2_11_12, L2_12_13,
    L2_20_21, L2_21_22, L2_22_23,
    L2_30_31, L2_31_32, L2_32_33,
    L2_01_00, L2_02_01, L2_03_02,
    L2_11_10, L2_12_11, L2_13_12,
    L2_21_20, L2_22_21, L2_23_22,
    L2_31_30, L2_32_31, L2_33_32,
    L2_00_10, L2_01_11, L2_02_12,
L2_03_13,
    L2_10_20, L2_11_21, L2_12_22,
L2_13_23,
    L2_20_30, L2_21_31, L2_22_32,
L2_23_33,
```

```
    L2_10_00, L2_11_01, L2_12_02,
L2_13_03,
    L2_20_10, L2_21_11, L2_22_12,
L2_23_13,
    L2_30_20, L2_31_21, L2_32_22,
L2_33_23
        [style = filled, fillcolor =
lightyellow];

    //vertical lines
    L2_00_01 -> L2_01_02 -> L2_02_03
    L2_10_11 -> L2_11_12 -> L2_12_13
    L2_20_21 -> L2_21_22 -> L2_22_23
    L2_30_31 -> L2_31_32 -> L2_32_33
    L2_03_02 -> L2_02_01 -> L2_01_00
    L2_13_12 -> L2_12_11 -> L2_11_10
    L2_23_22 -> L2_22_21 -> L2_21_20
    L2_33_32 -> L2_32_31 -> L2_31_30

    //horizontal lines
    L2_03_13 -> L2_13_23 -> L2_23_33
    L2_02_12 -> L2_12_22 -> L2_22_32
    L2_01_11 -> L2_11_21 -> L2_21_31
    L2_00_10 -> L2_10_20 -> L2_20_30
    L2_33_23 -> L2_23_13 -> L2_13_03
    L2_32_22 -> L2_22_12 -> L2_12_02
    L2_31_21 -> L2_21_11 -> L2_11_01
    L2_30_20 -> L2_20_10 -> L2_10_00

    //X+ Y+ turns
    L2_00_10 -> L2_10_11
    L2_10_20 -> L2_20_21
    L2_20_30 -> L2_30_31
    L2_01_11 -> L2_11_12
    L2_11_21 -> L2_21_22
    L2_21_31 -> L2_31_32
    L2_02_12 -> L2_12_13
    L2_12_22 -> L2_22_23
    L2_22_32 -> L2_32_33

    //X+ Y- turns
    L2_03_13 -> L2_13_12
    L2_02_12 -> L2_12_11
    L2_01_11 -> L2_11_10
    L2_13_23 -> L2_23_22
    L2_12_22 -> L2_22_21
    L2_11_21 -> L2_21_20
    L2_23_33 -> L2_33_32
    L2_22_32 -> L2_32_31
    L2_21_31 -> L2_31_30

    //X- Y+ turns
    L2_10_00 -> L2_00_01
    L2_11_01 -> L2_01_02
    L2_12_02 -> L2_02_03
    L2_20_10 -> L2_10_11
    L2_21_11 -> L2_11_12
    L2_22_12 -> L2_12_13
    L2_30_20 -> L2_20_21
    L2_31_21 -> L2_21_22
    L2_32_22 -> L2_22_23

    //X- Y- turns
    L2_33_23 -> L2_23_22
    L2_23_13 -> L2_13_12
    L2_13_03 -> L2_03_02
    L2_32_22 -> L2_22_21
    L2_22_12 -> L2_12_11
    L2_12_02 -> L2_02_01
    L2_31_21 -> L2_21_20
    L2_21_11 -> L2_11_10
    L2_11_01 -> L2_01_00

/**********************MESH 3
**********************/
    //style
    L3_00_01, L3_01_02, L3_02_03,
    L3_10_11, L3_11_12, L3_12_13,
    L3_20_21, L3_21_22, L3_22_23,
    L3_30_31, L3_31_32, L3_32_33,
    L3_01_00, L3_02_01, L3_03_02,
    L3_11_10, L3_12_11, L3_13_12,
    L3_21_20, L3_22_21, L3_23_22,
    L3_31_30, L3_32_31, L3_33_32,
    L3_00_10, L3_01_11, L3_02_12,
L3_03_13,
    L3_10_20, L3_11_21, L3_12_22,
L3_13_23,
    L3_20_30, L3_21_31, L3_22_32,
L3_23_33,
    L3_10_00, L3_11_01, L3_12_02,
L3_13_03,
    L3_20_10, L3_21_11, L3_22_12,
L3_23_13,
    L3_30_20, L3_31_21, L3_32_22,
L3_33_23
        [style = filled, fillcolor =
"#FFCACA"];

    //vertical lines
    L3_00_01 -> L3_01_02 -> L3_02_03
    L3_10_11 -> L3_11_12 -> L3_12_13
    L3_20_21 -> L3_21_22 -> L3_22_23
    L3_30_31 -> L3_31_32 -> L3_32_33
    L3_03_02 -> L3_02_01 -> L3_01_00
    L3_13_12 -> L3_12_11 -> L3_11_10
    L3_23_22 -> L3_22_21 -> L3_21_20
    L3_33_32 -> L3_32_31 -> L3_31_30

    //horizontal lines
    L3_03_13 -> L3_13_23 -> L3_23_33
    L3_02_12 -> L3_12_22 -> L3_22_32
    L3_01_11 -> L3_11_21 -> L3_21_31
    L3_00_10 -> L3_10_20 -> L3_20_30
    L3_33_23 -> L3_23_13 -> L3_13_03
    L3_32_22 -> L3_22_12 -> L3_12_02
    L3_31_21 -> L3_21_11 -> L3_11_01
    L3_30_20 -> L3_20_10 -> L3_10_00

    //X+ Y+ turns
    L3_00_10 -> L3_10_11
    L3_10_20 -> L3_20_21
    L3_20_30 -> L3_30_31

    L3_01_11 -> L3_11_12
    L3_11_21 -> L3_21_22
    L3_21_31 -> L3_31_32
    L3_02_12 -> L3_12_13
    L3_12_22 -> L3_22_23
    L3_22_32 -> L3_32_33

    //X+ Y- turns
    L3_03_13 -> L3_13_12
    L3_02_12 -> L3_12_11
    L3_01_11 -> L3_11_10
    L3_13_23 -> L3_23_22
    L3_12_22 -> L3_22_21
    L3_11_21 -> L3_21_20
    L3_23_33 -> L3_33_32
    L3_22_32 -> L3_32_31
    L3_21_31 -> L3_31_30

    //X- Y+ turns
    L3_10_00 -> L3_00_01
    L3_11_01 -> L3_01_02
    L3_12_02 -> L3_02_03
    L3_20_10 -> L3_10_11
```

```
    L3_21_11 -> L3_11_12
    L3_22_12 -> L3_12_13
    L3_30_20 -> L3_20_21
    L3_31_21 -> L3_21_22
    L3_32_22 -> L3_22_23

    //X- Y- turns
    L3_33_23 -> L3_23_22
    L3_23_13 -> L3_13_12
    L3_13_03 -> L3_03_02
    L3_32_22 -> L3_22_21
    L3_22_12 -> L3_12_11
    L3_12_02 -> L3_02_01
    L3_31_21 -> L3_21_20
    L3_21_11 -> L3_11_10
    L3_11_01 -> L3_01_00
}
```

Appendix R – Example User Design Constraints mapping

Note: this constraints file does not indicate any clock limits

```
## DIP SWITCH PINS ##################################################
NET "switches<0>"              LOC = "D22"; ## 1 on SW1 DIP switch (active-High)
NET "switches<1>"              LOC = "C22"; ## 2 on SW1 DIP switch (active-High)
NET "switches<2>"              LOC = "L21"; ## 3 on SW1 DIP switch (active-High)
NET "switches<3>"              LOC = "L20"; ## 4 on SW1 DIP switch (active-High)
NET "switches<4>"              LOC = "C18"; ## 5 on SW1 DIP switch (active-High)
NET "switches<5>"              LOC = "B18"; ## 6 on SW1 DIP switch (active-High)
NET "switches<6>"              LOC = "K22"; ## 7 on SW1 DIP switch (active-High)
NET "switches<7>"              LOC = "K21"; ## 8 on SW1 DIP switch (active-High)

## LED PINS #########################################################
NET "leds<0>"              LOC = "AC22"; ## 2 on LED DS12, 1 on J62
NET "leds<1>"              LOC = "AC24"; ## 2 on LED DS11, 2 on J62
NET "leds<2>"              LOC = "AE22"; ## 2 on LED DS9, 3 on J62
NET "leds<3>"              LOC = "AE23"; ## 2 on LED DS10, 4 on J62
NET "leds<4>"              LOC = "AB23"; ## 2 on LED DS15, 5 on J62
NET "leds<5>"              LOC = "AG23"; ## 2 on LED DS14, 6 on J62
NET "leds<6>"              LOC = "AE24"; ## 2 on LED DS22, 7 on J62
NET "leds<7>"              LOC = "AD24"; ## 2 on LED DS21, 8 on J62

## HPC PINS #########################################################
NET "hpc_send<0>"              LOC = "AP33"; ## F23 on J64
NET "hpc_send<1>"              LOC = "AP32"; ## F22 on J64
NET "hpc_send<2>"              LOC = "AM31"; ## E22 on J64
NET "hpc_send<3>"              LOC = "AL30"; ## E21 on J64
NET "hpc_send<4>"              LOC = "AL33"; ## F26 on J64
NET "hpc_send<5>"              LOC = "AM33"; ## F25 on J64
NET "hpc_send<6>"              LOC = "AN34"; ## E25 on J64
NET "hpc_send<7>"              LOC = "AN33"; ## E24 on J64
```