FPGA Design for DDR3 Memory

Sponsored by Teradyne, North Reading, MA

A Major Qualifying Project proposal to be submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science

By:

_____
Laura Fischer

_____
Yura Pyatnychko

Submitted On: 12 March, 2012

_____
Professor Xinming Huang, Advisor, Electrical & Computer Engineering

_____
Professor John A. Mcneill, Advisor, Electrical & Computer Engineering

Acknowledgements

We would like to thank:

Teradyne of North Reading, MA, including our contacts, for arranging the project and providing us with the hardware to develop our design.

Professor Xinming Huang for his guidance and advice as an advisor throughout the project.

Professor John A. McNeill for organizing the project and attending to the project needs as an advisor.

Abstract

The project presents a memory arbiter system capable of allowing two systems to communicate to the same DDR3 SDRAM memory. The arbiter was designed using Verilog, implemented using Xilinx Integrated Software Environment (ISE) and validated using iSim and ChipScope. The final design is implemented on a Virtex 6 FPGA chip. The arbiter can achieve a maximum performance of around 50 Gb/s, with the two systems reaching transfer rates of 25 Gb/s.

Executive Summary

In the last 30 years, computer memory has evolved rapidly, seeing improvements in both capacity and speed. However, the logic for controlling the memory has also become increasingly more complex and difficult to interface with. The third generation of double data rate synchronous dynamic random access memory (DDR3 SDRAM) is the newest and fastest volatile memory currently available. DDR3 DRAM is one of the many types of random access memory used to temporarily hold data that the system (e.g. computer) needs to have quick access to.

DRAM memory is designed for communication with a single system. For two systems to share the same memory, an arbiter must be used. The memory arbiter serves as an interface to both systems, and grants each system access to the memory one at a time to avoid collisions. In this project, Teradyne assigned the MQP team the task of developing a memory arbiter for DDR3 SDRAM for implementation on a Virtex 6 FPGA. The DDR3 SDRAM would be used in Teradyne's testers as data buffers between test-benches and the units under tests (UUT). The memory would enable very fast data transfer rates and significantly reduce the time the testers take to transfer data. The development of the arbiter was done on ML605 evaluation kit using Xilinx tools. Xilinx Integrated Software Environment (ISE) served as the primary work-bench for the development of the arbiter. It enabled the team to write Verilog code that describes the design of the arbiter and to insert specialized Xilinx cores, such as a DDR3 memory interface and first in first out (FIFO) blocks.

The arbiter was designed to follow a see-saw like data flow. In a two system configuration, one system is given permission to communicate with the arbiter. The permission periodically switches from one system to the other (i.e. switches alternately) regardless of whether either of systems need to send commands. When a system is given permission to the arbiter they are allowed to send read or write commands; these commands are buffered inside FIFOs within the arbiter. When system two is given permission to the arbiter, the arbiter executes the buffered commands of system one by sending them

to memory. This buffering scheme enables un-synchronized communication between the systems and the arbiter and enables heavy data throughput. Ultimately, the developed arbiter allows the systems to communicate at speeds of up to 25 Gb/s. The effective transfer rate of the two systems is dependent on both their own transfer rate and the transfer rate of the other system that is connected to the arbiter. This relationship is very similar to a model used for evaluating the equivalent resistance of two resisters connected in parallel.

Throughout the development process, the MQP team extensively tested the logic in simulation. The DDR3 SDRAM was modeled using Xilinx DDR3 memory model as well as a stimulus module developed by the team that emulated the expected behavior of the two systems. A Verilog test-bench was written for utilizing the memory model and triggering the stimulus modules. The simulation was run in Xilinx's iSim Environment. The iSim tool was very useful because its GUI easily enables the display of any signal waveform in the design. Since simulation has a fast turn-around time, the MQP team iteratively validated the design in iSim to ensure any bugs in the code were caught early. Although done less frequently, the FPGA implementation of the design was periodically validated using ChipScope. ChipScope is used to probe signals in hardware, and it provided the most accurate depiction of design behavior. However, this process required a lot more effort and was more time consuming then validating through RTL simulation.

Although the arbiter has been fully developed and is fully functional, it still needs to be properly tested with greater coverage. Functionality checkers should be developed for this design, which would test the validity of the arbiter in a run-time environment as of in practical use.

Contents

Table of Figures

1. Introduction

In many applications, it is useful for more than one system to interact with memory. In testing specifically, when the unit under test (UUT) is communicating with memory, there needs to be a second system regulating what the UUT is writing to memory. Teradyne has assigned us the task of developing an arbiter for the latest double data rate memory (DDR3).

Xilinx has released a memory controller for DDR3 which handles everything for communication between one system and memory. If two systems try to read or write to DDR3 memory at the same time, however, there is a great risk the data read or written will not be accurate. The goal of this project is to design a traffic controller, or arbiter, that will delegate which system's turn it is to send requests while blocking requests from the other system. An arbiter will prevent read/write collisions and maintain request order to ensure the memory is holding accurate data.

There are several challenges to designing an arbiter that interacts with a DDR3 memory controller. First is that the arbiter must work around the memory's refresh rate. During a refresh, there can be no communication with the memory. In addition, the arbiter must keep all read and write requests from the systems in order. If one system sends a read before the other sends a write, the read must be executed first. Finally, the arbiter cannot allow one system to use the arbiter for too long. By keeping all these factors in mind, the developed an arbiter should be successful in a testing environment.

The approach to developing the arbiter in this project is to first create a block diagram show how the two systems will interact with memory. Once the block diagram is finished, the task is to develop an arbiter protocol which will keep requests in order and balance access time between the two systems. The first step in implementing the design is to get one system to communicate with the DDR3 memory controller. This involves developing a state machine to fulfill the memory interface protocol specified by the controller. The state machine needs to be able to pause during refresh cycles and pick-

up where it left off when the refresh cycle is through. The refresh cycle is difficult to detect at the correct time in order to avoid losing or repeating a request. Once one system works completely, the next step is getting two systems to work using the arbiter protocol.

2. Background

There are two different types of random access memory: synchronous and dynamic. Synchronous random access memory (SRAM) is used for high-speed, low power applications while dynamic random access memory (DRAM) is used for its low cost and high density. Designers have been working to make DRAM faster and more energy efficient. The following sections will discuss the differences between these two types of RAM, as well as present the progression of DRAM towards a faster, more energy efficient design.

2.1. Random Access Memory (RAM)

Today, the most common type of memory used in digital systems is random access memory (RAM). The time it takes to access RAM is not affected by the data's location in memory. RAM is volatile, meaning if power is removed, then the stored data is lost. As a result, RAM cannot be used for permanent storage. However, RAM is used during runtime to quickly store and retrieve data that is being operated on by a computer. In contrast, nonvolatile memory, such as hard disks, can be used for storing data even when not powered on. Unfortunately, it takes much longer for the computer to store and access data from this memory. There are two types of RAM: static and dynamic. In the following sections the differences between the two types and the evolution of DRAM will be discussed. [8]

2.2. Static Random Access Memory (SRAM)

Static random access memory (SRAM) stores data as long as power is being supplied to the chip [8]. Each memory cell of SRAM stores one bit of data using six transistors: a flip flop and two access transistors (i.e. four transistors) [5;9]. SRAM is the faster of the two types of RAM because it does not involve capacitors, which involve sense amplification of a small charge. For this reason, it is used in cache memory of computers .[4] Additionally, SRAM requires a very small amount of power to maintain its data in standby mode [9]. Although SRAM is fast and energy efficient it is also expensive due to the

amount of silicon needed for its large cell size [5;10]. This presented the need for a denser memory cell, which brought about DRAM.

2.3. Dynamic Random Access Memory (DRAM)

According to Wakerly, “In order to build RAMs with higher density (more bits per chip), chip designers invented memory cells that use as little as one transistor per bit”[8]. Each DRAM cell consists of one transistor and a capacitor [4]. Since capacitors “leak” or lose charge over time, DRAM must have a refresh cycle to prevent data loss [11].

According to a high-performance DRAM study on earlier versions of DRAM, DRAM’s refresh cycle is one reason DRAM is slower than SRAM [2]. The cells of DRAM use sense amplifiers to transmit data to the output buffer in the case of a read and transmit data back to the memory cell in the case of a refresh [4]. During a refresh cycle, the sense amplifier reads the degraded value on a capacitor into a D-Latch and writes back the same value to the capacitor so it is charged correctly for 1 or 0 [8]. Since all rows of memory must be refreshed and the sense amplifier must determine the value of a, already small, degenerated capacitance, refresh takes a significant amount of time[4;2]. The refresh cycle typically occurs about every 64 milliseconds [8].The refresh rate of the latest DRAM (DDR3) is about 1 microsecond.

Although refresh increases memory access time, according to a high-performance DRAM study on earlier versions of DRAM, the greatest amount of time is lost during row addressing, more specifically, “[extracting] the required data from the sense amps/row caches” [2]. During addressing, the memory controller first strobes the row address (RAS) onto the address bus. Once the RAS is sent, a sense amplifier (one for each cell in the row) determines if a charge indicating a 1 or 0 is loaded into each capacitor. This step is long because “the sense amplifier has to read a very weak charge” and “the row is formed by the gates of memory cells.” [4] The controller then chooses a cell in the row from which to read from by strobing the column address (CAS) onto the address bus. A write requires the

enable signal to be asserted at the same time as the CAS, while a read requires the enable signal to be de-asserted. The time it takes the data to move onto the bus after the CAS is called the CAS latency [2].

Although recent generations of DRAM are still slower than SRAM, DRAM is used when a larger amount of memory is required since it is less expensive. For example, in embedded systems, a small block of SRAM is used for the critical data path, and a large block of DRAM is used to satisfy all other needs [6]. The following section will discuss the development of DRAM into a faster, more energy efficient memory.

2.4. Development of DRAM

Many factors are considered in the development of high performance RAM. Ideally, the developer would always like memory to transfer more data and respond in less time; memory would have higher bandwidth and lower latency. However, improving upon one factor often involves sacrificing the other.[2]

Bandwidth is the amount of data transferred per second. It depends on the width of the data bus and the frequency at which data is being transferred. Latency is the time between when the address strobe is sent to memory and when the data is placed on the data bus. DRAM is slower than SRAM because it periodically disables the refresh cycle and because it takes a much longer time to extract data onto the memory bus. Advancements have been, however, to several different aspects of DRAM to increase bandwidth and decrease latency. [2]

Over time, DRAM has evolved to become faster and more energy efficient by decreasing in cell size and increasing in capacity[2]. In the following section, we will look at different types of DRAM and how DDR3 memory has come to be.

2.4.1. DRAM

One of the reasons the original DRAM was very slow is because of extensive addressing overhead. In the original DRAM, an address was required for every 64-bit access to memory. Each access took six clock

cycles. For a four 64-bit access to consecutive addresses in memory, the notation for timing was 6-6-6-6. Dashes separate memory accesses and the numbers indicate how long the accesses take. This DRAM timing example took 24 cycles to access the memory four times. [2]

In contrast, more recent DRAM implements burst technology which can send many 64-bit words to consecutive addresses. While the first access still takes six clock cycles due memory accessing, the next three adjacent addresses can be performed in as little as one clock cycle since the addressing does not need to be repeated. During burst mode, the timing would be 6-1-1-1, a total of nine clock cycles.[2]

The original DRAM is also slower than its descendants because it is asynchronous. This means there is no memory bus clock to synchronize the input and output signals of the memory chip. The timing specifications are not based on a clock edge, but rather on maximum and minimum timing values (in seconds). [2] The user would need to worry about designing a state machine with idle states, which may be inconsistent when running the memory at different frequencies.

2.4.2. Synchronous DRAM

In order to decrease latency, SDRAM utilizes a memory bus clock to synchronize signals to and from the system and memory. Synchronization ensures that the memory controller does not need to follow strict timing; it simplifies the implemented logic and reduces memory access latency. With a synchronous bus, data is available at each clock cycle. [2]

SDRAM divides memory into two to four banks for concurrent access to different parts of memory. Simultaneous access allows continuous data flow by ensuring there will always be a memory bank read for access [2]. The addition of banks adds another segment to the addressing, resulting in a bank, row, and column address. The memory controller determines if an access addresses the same bank and row as the previous access, so only a column address strobe must be sent. This allows the access to occur much more quickly and can decrease overall latency. [9]

2.4.3. DDR1 SDRAM

DDR1 SDRAM (i.e. first generation of SDRAM) doubles the data rate (hence the term DDR) of SDRAM without changing clock speed or frequency. DDR transfers data on both the rising and falling edge of the clock, has a pre-fetch buffer and low voltage signaling, which makes it more energy efficient than previous designs.[2]

Unlike SDRAM, which transfers 1 bit per clock cycle from the memory array to the data queue, DDR1 transfers 2 bits to the queue in two separate pipelines. The bits are released in order on the same output line. This is called a 2n-prefetch architecture. In addition, DDR1 utilizes double transition clocking by triggering on both the rising and falling edge of the clock to transfer data. As a result, the bandwidth of DDR1 is doubled without an increase in the clock frequency. [2]

In addition to doubling the bandwidth, DDR1 made advances is energy efficiency. DDR1 can operate at 2.5 V instead of the 3.3V operating point of SDRAM thanks to low voltage signaling technology.[2]

2.4.4. DDR2 SDRAM

Data rates of DDR2 SDRAM are up to eight times more than original SDRAM. At an operation voltage of 1.8V, it achieves lower power consumption than DDR1. DDR2 SDRAM has a 4-bit prefetch buffer, an improvement from the DDR1 2-bit prefetch. This means that 4 bits are transferred per clock cycle from the memory array to the data bus, which increases bandwidth. [2]

2.4.5. DDR3 SDRAM

DDR3 provides two burst modes for both reading and writing: burst chop (BC4) and burst length eight (BL8)[3]. BC4 allows bursts of four by treating data as though half of it is masked. This creates smooth transitioning if switching from DDR2 to DDR3 memory. [1] However, burst mode BL8 is the primary burst mode. BL8 allows the most data to be transferred in the least amount of time; it transfers the greatest number of 64-bit data packets (eight) to or from consecutive addresses in memory, which means

addressing occurs once for every eight data packets sent. In order to support a burst length of eight data packets, DDR3 SDRAM has an 8-bit prefetch buffer.

DDR3, like its predecessors, not only improves upon bandwidth, but also energy conservation. Power consumption of DDR3 can be up to 30 percent less than DDR2. The DDR3 operating voltage is the lowest yet, at 1.5 V, and low voltage versions are supported at voltages of 1.35 V.

2.5. Timeline

Ideally, memory performance would improve at the same rate as central processing unit (CPU) performance. However, memory latency has only improved about five percent each year [9]. The longest latency (RAS latency) of the newest release of DRAM for each year is shown in the plot in Figure 2.5-1.

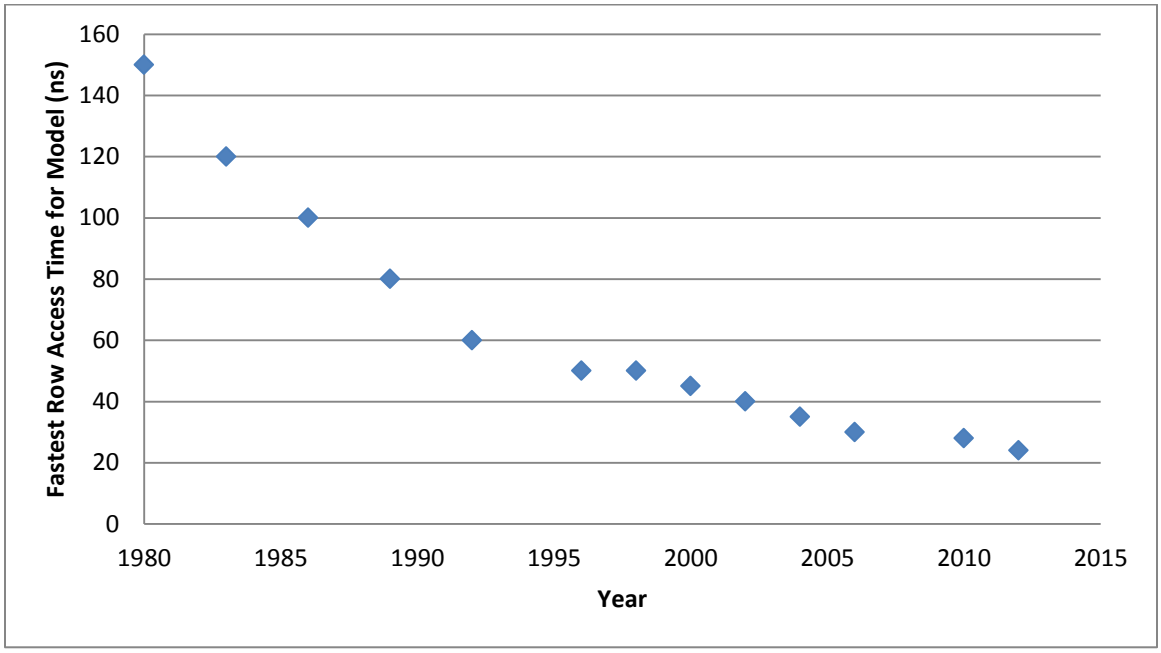

Figure 2-1: DRAM Row Access Latency vs. Year [9]

As seen in Figure 2.5-1, the row access latency decreases linearly with every new release of DRAM until 1996. Once SDRAM is released in 1996, the difference in latency from year to year is much smaller. With recent memory releases it is much more difficult to reduce RAS latency. This can be seen

especially for DDR2 and DDR3 memory releases 2006 to 2012. CAS latency, unlike RAS latency, consistently decreases (bandwidth increases) with every memory release, and in the new DDR3 memory, is very close to 0 ns. Figure 2.5-2 shows the column access latency.

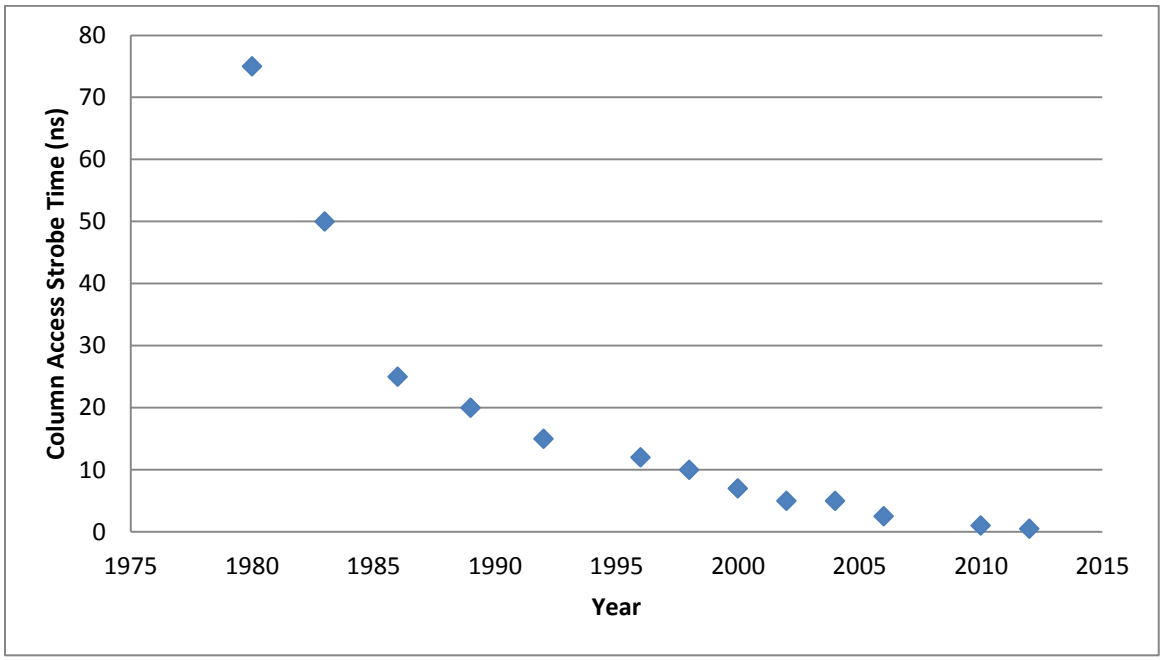

Figure 2-2: DRAM Column Address Time vs. Year [9]

Looking at some prominent areas of the CAS graph, it can be seen in Figure 2.5-2 that bandwidth greatly increased (CAS decreased) from 1983 to 1986. This is due to the switch from NMOS DRAMs to CMOS DRAMs [9]. In 1996 the first SDRAM was released. The CAS latency decreased (bandwidth increased) due to synchronization and banking. In later years, the CAS latency does not decrease by much, but this is expected since the latency is already much smaller.

Comparing Figure 2.5-2 to Figure2.5-1, CAS time decreases much more drastically than RAS time. This means the bandwidth greatly improves, while latency improves much more slowly. In 2010, when DDR2 was released, it can be seen that latency was sacrificed (Figure 2.5-1) for an increase in bandwidth (Figure 2.5-2). [9]

3. Methodology

In this section the ML605 and Virtex-6 board hardware is described as well as the tools utilized for design and validation. The Xilinx Integrated Software Environment (ISE) was used for design and iSim and ChipScope were used for validation in simulation and in hardware.

3.1. Hardware

3.1.1. Virtex-6 FPGA

The Virtex-6 FPGA (XC6VLX240T) is used to implement the arbiter. This FPGA has 241, 152 logic cells and is organized into banks (40 pins per bank) [15;14]. These logic cells, or slices, are composed of four look-up tables (LUTs), multiplexers, and arithmetic carry logic. LUTs implement boolean functions, and multiplexers enable combinatorial logic. Two slices form a configurable logic block (CLB). [16].

In order to distribute a clock signal to all these logic blocks, the FPGA has five types of clock lines: BUFG, BUFR, BUFIO, BUFH, and high-performance clock. These lines satisfy “requirements of high fanout, short propagation delay, and extremely low skew”[15]. The clock lines are also split into categories depending on the sections of the FPGA and components they drive. The three categories are: global, regional, and I/O lines. Global clock lines drive all flip-flops, clock enables, and many logic inputs. Regional clock lines drive all clock destinations in their region and two bordering regions. There are six to eighteen regions in an FPGA. Finally, I/O clock lines are very fast and only drive I/O logic and serializer/deserializer circuits. [15]

3.1.2. ML605 Board

The Virtex-6 FPGA is included on the ML605 Development Board. In addition to the FPGA, the development board includes a 512 MB DDR3 small outline dual inline memory module (SODIMM), which our design arbitrates access to. A SODIMM is the type of board the memory is manufactured on [9]. The FPGA also includes 32 MB of linear BPI Flash and 8 Kb of IIC EEPROM.

Communication mechanisms provided on the board include Ethernet, SFP transceiver connector, GTX port, USB to UART bridge, USB host and peripheral port, and PCI Express. [13] The only connection used during this project was the USB JTAG connector. It was used to program and debug the FPGA from the host computer.

There are three clock sources on the board: a 200 MHz differential oscillator, 66 MHz single-ended oscillator and SMA connectors for an external clock.[13] This project utilizes the 200MHz oscillator.

Peripherals on the ML605 board were useful for debugging purposes. The push buttons were used to trigger sections of code execution in ChipScope such as reading and writing from memory. Dip switches acted as configuration inputs to our code. For example, they acted as a safety to ensure the buttons on the board were not automatically set to active when the code was downloaded to the board. In addition, the value on the switches indicated which system would begin writing first for debugging purposes. LEDs were used to check functionality of sections of code as well, and for additional validation, they can be used to indicate if an error as occurred. Although we did not use it, the ML605 board provides an LCD.

3.2. Tools

Now that the hardware where the design is placed is described, the software used to manipulate the design can be described. The tools for design include those provided within Xilinx Integrated Software Environment, and the tools used for validation include iSim and ChipScope. This looks at the turn-around time for both validation tools and what it means for the design process.

3.2.1. Xilinx Integrated Software Environment (ISE)

We designed the arbiter using Verilog hardware description language in Xilinx Integrated Software Environment (ISE). ISE is an environment in which the user can “take [their] design from design entry through Xilinx device programming”[14]. The main workbench for ISE is ISE Project Navigator. The

Project Navigator tool allows the user to effectively manage their design and call upon development processes. In Figure 3.3-1, a screen shot of ISE Project Navigator can be seen.

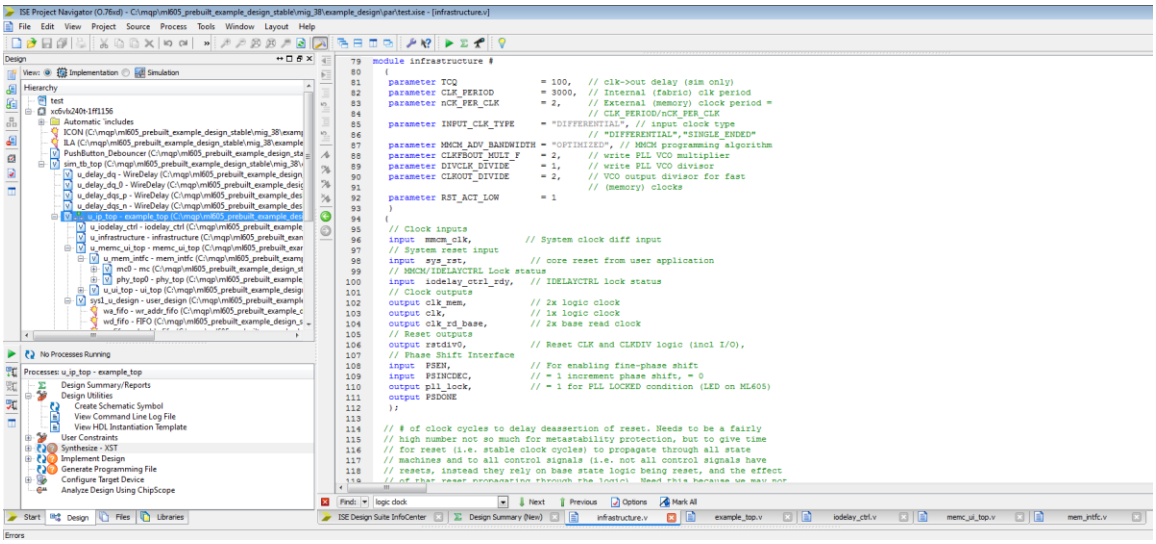

Figure 3-1: Screen Shot of ISE Project Navigator

Figure 3.3-1 shows some main windows in ISE Project Navigator. On the right hand side is the window for code entry. The hierarchal view of modules in the design appears on the left, and when implementation is selected from the top, the design implementation progress is shown in the bottom window. If simulation were selected instead of implementation there would be an option to run the design for simulation.

The main processes called upon by ISE are synthesis, implementation, and bit stream generation. During synthesis, Xilinx Synthesis Technology (XST) is called upon. XST synthesizes Verilog, VHDL or mixed language designs and creates netlist files. Netlist files, or NGC files, contain the design logic and constraints. They are saved for use in the implementation process. During synthesis, the XST checks for synthesis errors (parsing) and infers macros from the code. When the XST infers macros it recognizes parts of the code that can be replaced with components in its library such as MUXes, RAMs,

adder and subtracters. In addition to inferring macros, the XST recognizes finite state machines and encodes them in a way that would be best for reduced area and/or increased speed. [14]

Implementation is the longest process to perform on the design. The first step of implementation is to combine the netlists and constraints into a design/NGD file. The NGD file is the design file reduced to Xilinx primitives. This process is called translation. During the second step, mapping, the design is fitted into the target device. This involves turning logic into FPGA elements such as configurable logic blocks. Mapping produces a native circuit description (NCD) file. The third step, place and route, uses the mapped NCD file to place the design and route timing constraints. Finally, the program file is generated and, at the finish of this step, a bitstream is ready to be downloaded to the board. [14]

3.2.2. Synthesis and Simulation

Once the design has been synthesized, simulation of the design is possible. Simulating a design enables verification of logic functionality and timing [14]. We used simulation tool in ISE (isim) to view timing and signal values. In order to utilize isim, we created a test bench to provide the design with stimulus. Since simulation only requires design synthesis, it is a relatively fast process. The short turn-around time of simulation means we were able to iteratively test small changes to the design and, therefore, debug our code efficiently.

3.2.3. Implementation and Hardware Validation

Once the design was working in simulation, we still needed to test the design's functionality in hardware. Testing the design in hardware is the most reliable validation method. In order to download the design to the board, it first needs to be implemented in ISE. Implementation has a much longer turn-around time than synthesis, so while functionality in hardware ensures the design is working, simulation is the practical choice for iterative verification.

In order to test our design in hardware, we utilized ChipScope Pro Analyzer, a GUI which allows the user to “configure [their] device, choose triggers, setup the console, and view results of the capture on the fly” [12]. In order to use ChipsScope Pro, you may either insert ChipScope Pro Cores into the design using the Core Generator, a tool that can be accessed in ISE Project Navigator, or utilize the PlanAhead or Core Inserter tool, which automatically inserts cores into the design netlist for you. [12]

One method of inserting ChipScope cores into the design is by utilizing PlanAhead software. The PlanAhead tool enables the creation of floorplans. Floorplans provide an initial view of “the design’s interconnect flow and logic module sizes” [17,59]. This helps the designer to “avoid timing, utilization, and routing congestion issues” [17, 112]. PlanAhead also allows the designer to create and configure I/O ports and analyze implementation results, which aids in the discovery of bottlenecks in the design[17]. For our project, however, we utilized PlanAhead only for its ability to automatically insert ChipScope cores. PlanAhead proved to be inefficient for our purposes since many times, when a change was made in the design, the whole netlist would need to be selected again. In addition, there were bugs in the software that greatly affected the turn-around time of debugging, and it crashed several times. If PlanAhead were used for floor planning and other design tools, then it might have proved to be much for useful.

In replace of PlanAhead, we utilized the Core Generator within ISE. The ChipScope cores provided by Xilinx include ICON, ILA, VIO, ATC2, and IBERT. The designer can choose which cores to insert by using the Core Generator in ISE. The ICON core provides communication between the different cores and the computer running ChipScope. [14] It can connect up to fifteen ILA, VIO, and ATC2 cores [12]. The ILA core is used to synchronously monitor internal signals. It contains logic to trigger inputs and outputs and capture data. ILA cores allow up to sixteen trigger ports, which can be 1 to 256 bits wide. The VIO core can monitor signals like ILA, but also drive internal FPGA signals real-time. The ATC2 core is similar to the ILA core, but was created for Agilent FPGA dynamic probe technology. Finally, the IBERT

core contains “all the logic to control, monitor, and change transceiver parameters and perform bit error ratio tests” [12,19].

The only ChipScope cores we were concerned with in this project were the ICON and ILA cores. We inserted one ChipScope ILA and ICON cores using the ISE Core Generator within ISE Project Navigator. The ILA core allowed us to monitor internal signals in the FPGA. Instead of inserting a VIO core, which allows inputs to and outputs from ChipScope, we used buttons to trigger the execution of write and read logic.

3.2.4. Analysis of Turn-Around Times

As introduced in sections 3.3.2 and 3.3.3, implementation takes much longer than synthesis. Therefore, when it comes down to turn-around time, simulation is much more effective for iterative debugging. In Figure 3.3-2, the phases for simulation and hardware validation can be seen as well as the time it takes to complete each phase.

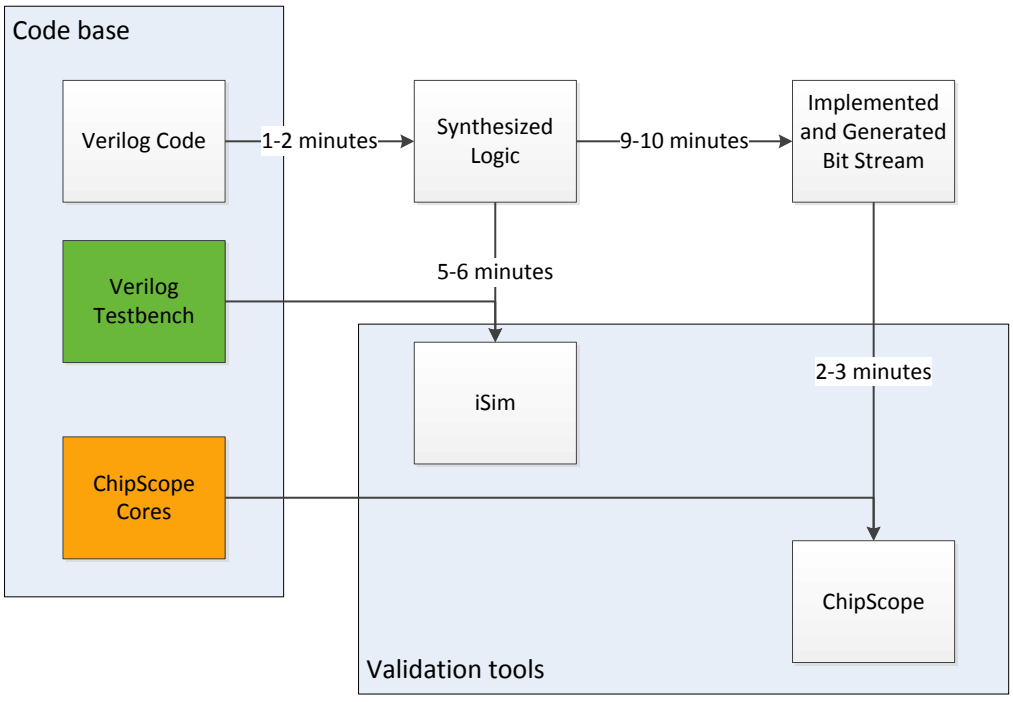

Figure 3-2: Flow Chart and Timing for Simulation and Hardware Validation

For simulation, the process starts at Verilog code, becomes synthesized logic, and using a testbench, is run in iSim for viewing. This process takes about eight minutes total. A system's simulation run-time is much longer than if it were running on hardware, but simulation is still faster than hardware validation because it does not have to undergo implementation. The bottleneck in our simulation process is the set up time for the DDR3 memory model which accounts for most of the simulation time.

Hardware validation starts at Verilog code, is synthesized, implemented, and imported into ChipScope. This whole process takes about fifteen minutes. Most of the time spent for hardware validation is on implementation of the design. In addition, hardware validation requires more of the user's attention. It is more difficult and takes more time to set up a ChipScope core than it does to create a test bench for simulation. While a test bench (green) involves writing some simple code, a ChipScope core (orange) involves setting up all the signals to be probed. Not only is simulation faster, but the iSim tool is easier to use than ChipScope. Figure 3.3-3 shows a screen shot of iSim.

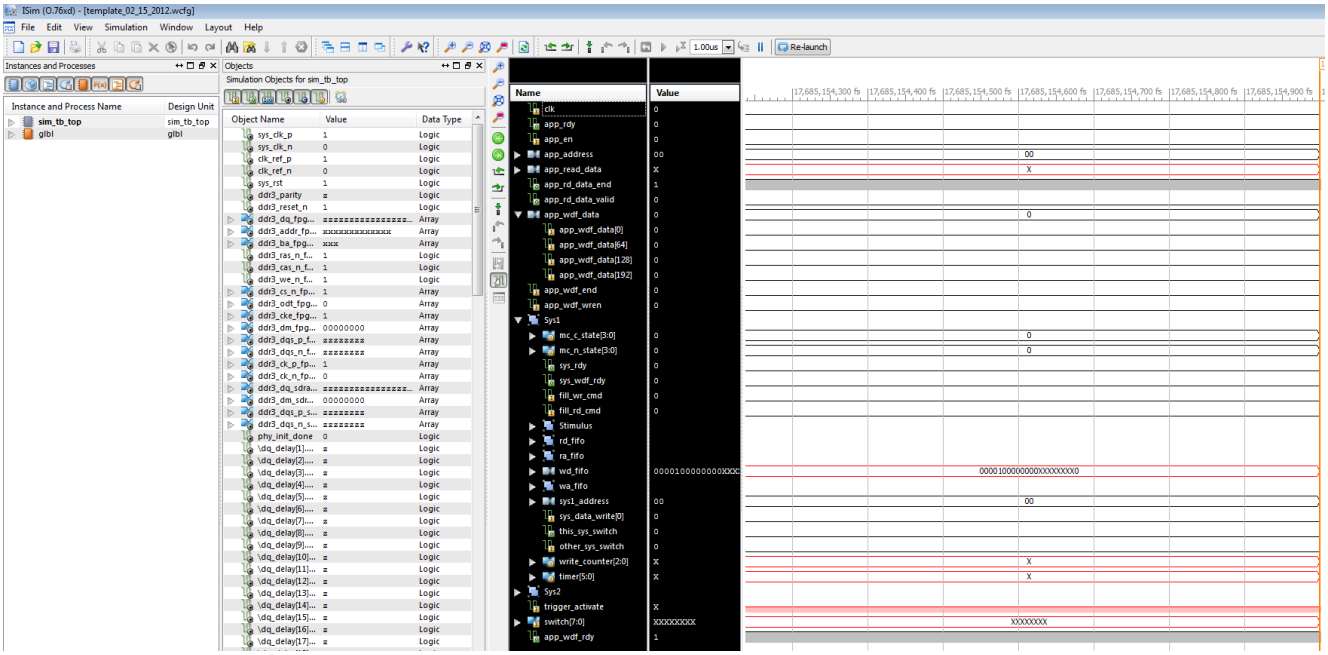

Figure 3-3: iSim Screen Shot

The screen shot of iSim shows the instance names in the first column, all the signals to choose from in the second, and the signals and their waveforms in the third and fourth columns. The user can view any signal without having to port it out of the design and re-implement like when using ChipScope. When adding an additional signal in iSim, only simulation needs to be restarted. The iSim interface makes debugging much easier with collapsible signal viewing, grouping abilities, and a large window for viewing many signals at once. A screen shot of ChipScope is shown in Figure 3.3-4.

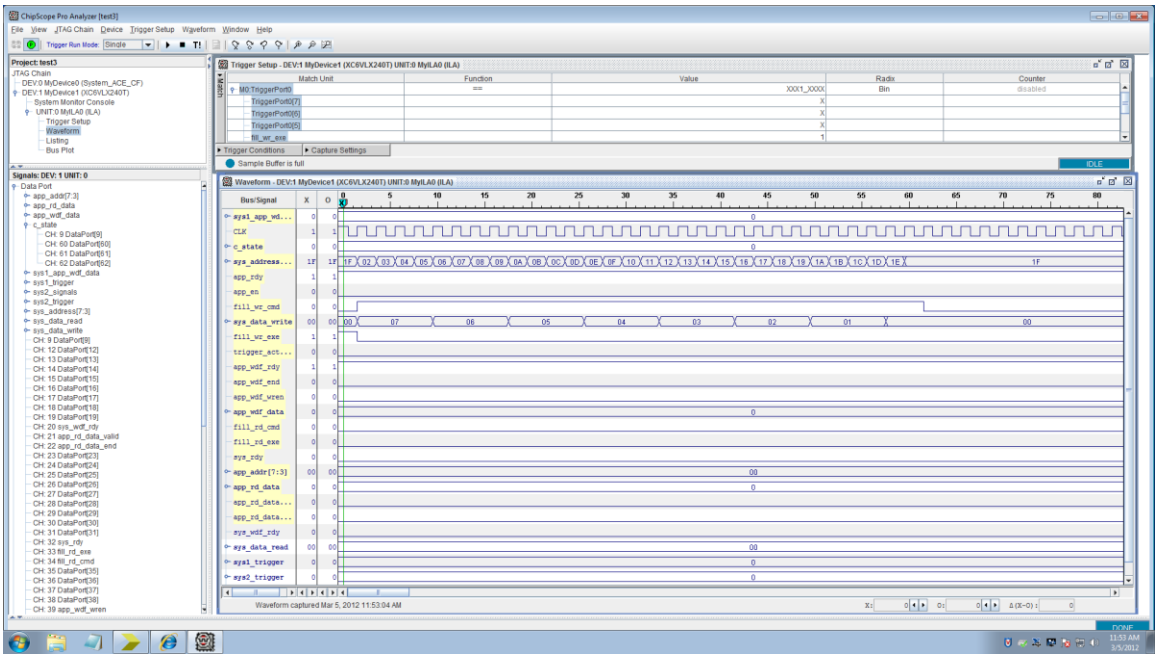

Figure 3-4: ChipScope Screen Shot

In ChipScope, you can view the devices, signals, triggers, and waveforms window. The time ChipScope is able to capture is much less than iSim. For this reason, triggers are required to execute different parts of code; this is where buttons were utilized. If a signal could not fit into the allowable number of signal inputs or was forgotten, it would need to be added to the design and implemented all over again.

In conclusion, hardware validation is the most accurate validation possible. However, it has a much longer turn-around time than simulation. Therefore, simulation is used for iterative debugging and functionality testing, while hardware validation is the next step to ensure design accuracy.

3.2.5. Xilinx Core Generator

One tool in ISE that was very important to our project was the CORE Generator. The core generator provided us with not only the ChipScope cores, but the memory controller, and FIFOs as well. The core generator can be accessed within ISE Project Navigator. It provides many additional functions for the designer. The options provided for creating FIFOs, for example, include common or independent clocks, first-word fall-through; a variety of flags to indicated the amount of data in the FIFO and write width, read width and depth. The different width capabilities allowed us to create asynchronous FIFOs. The memory controller was created using the Xilinx memory interface generator (MIG). There were options to use an AXI4, native, or user interface, which is discussed in a following section on interfacing with the Xilinx MIG.

4. Implementation and Design

4.1. Xilinx Memory Interface Generator

The Xilinx Memory Interface Generator is located within the CORE generator of XPS [3]. The memory interface, or memory controller, is composed of many modules which allow both communication with and testing of the memory. The memory controller handles communicating with the high-speed interface of the DDR3 so that we don't have to. Therefore, our user design only needs to account for interfacing with the memory controller.

4.1.1. Memory Controller Hierarchy

Figure 4.1-1 provides a general modular view of the memory controller. For a more detailed structural map of the signals between modules refer to Figure 6.1 in the appendix.

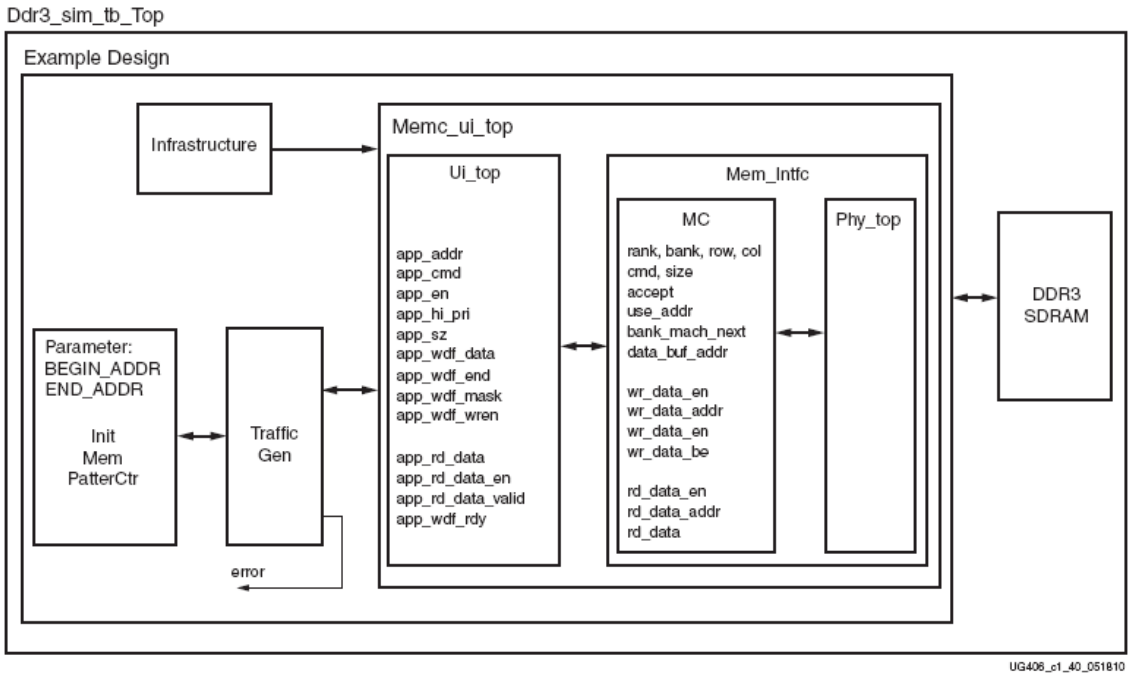

Figure 4-1: Example Design Block Diagram [3]

The memory controller is contained within the wrapper, example_top, shown in Figure 4.1-1 as example design. Within example_top, there are four overarching modules: memc_ui_top, infrastructure, m_traffic_gen, and init_mem0 (init_mem_PatterCtr).

memc_ui_top: This is the memory controller. It decodes addresses into rows, columns, and banks and converts command signals into actual data transfers to and from memory.

Infrastructure: Distributes the clock signal to all the modules.

m_traffic_gen: The traffic generator modules of the MIG core generate a variety of command and data patterns in order to test the memory. The read data is then compared to the expected (generated) patterns in order to detect any faults.

init_mem0: This module seeds m_traffic_gen's fifos with commands, addresses, and data.

Vio_sync_out_32 u_cs4: provides an interface for the user to manually control the traffic generator within ChipScope.

Iodelay_ctrl: Sets and maintains a 200 MHz clock signal for the system.

4.1.2. Infrastructure and Clock Frequencies

System clocks and reset are generated within Infrastructure. The infrastructure module distributes two clock signals to the other modules: clk_mem and clk. The clk signal is 200Mhz and is used within the arbiter to interface with the memory controller. The clk_mem signal is 400Mhz and is used for communication between the memory controller and the DDR3 memory. Infrastructure also enables reset by enabling the user to control the sys_rst input into the module.

4.1.3. Validation Capabilities

The memory controller enables memory testing and validation. It uses the traffic_gen module to generate sequences for addresses, data, instructions and burst length. In order to utilize the traffic generator test patterns and comparator, the following ports may be used: instr_mode_i, addr_mode_i, bl_mode_i, and data_mode_i. The different modes they allow are:

Address Modes:

Fixed

Pseudo-random

Sequential

Burst Length Modes:

Fixed

Pseudo-random

Data Modes:

Reserved

Fixed: Data stays the same.

Address : address is used as data pattern

Hammer: all 1s are on DQ pins on the rising edge of DQS and 0s on falling edge of DQS

Neighbor: all 1s on DQ pins on rising edge of DQS except for one pin.

Walking 1s: walking 1s on DQ pins.

Walking 0s: walking 0s on DQ pins.

Pseudo-random: 32-stage LFSR generates random data and is seeded by the starting address.

4.2. Interfacing with the Xilinx MIG

There are a three ways to interface with the Xilinx Memory Controller. When deciding which method to use, we considered both the capabilities each provided and ease of use.

AXI4 Slave Interface: AXI4 provides a standard so developers only need to learn one protocol for IP. It provides different options such as bursts of up to 256 data transfer cycles with a single address, a small amount of logic, and even the removal of an “address phase” which allows unlimited data burst size.[18] With the AXI interface, there can be several masters and slaves. Data can move in both directions and the AXI interconnect even provides the logic for arbitration.

Native: The native interface does not contain buffers in order to return data as soon as possible. This means the data might be out of order. It would be the designer’s job to reorder the received data [3].

UI: The user interface is a simpler version of the Native interface. It buffers read/write data, reorders return data to match order of requests, and translates addressing to that required by SDRAM [3].

4.2.1. User Interface

Since it was our assigned task to create our own arbiter, we eliminated the option of using the AXI4 interface. The Native interface would require us reorder data coming from memory, which would make arbitrating between two systems much more complicated. The user interface does the reordering for us and enables the creation of a simpler design; we implemented the user interface for interfacing with memory. Figure 4.2-1 shows the interface between the user design (our arbiter) and memory (DDR3 SDRAM). The user interface (UI) provides a more intuitive way to communicate with memory. The signals between the user design and the UI are the only ones in Figure 4.2-1 the arbiter design needs to account for.

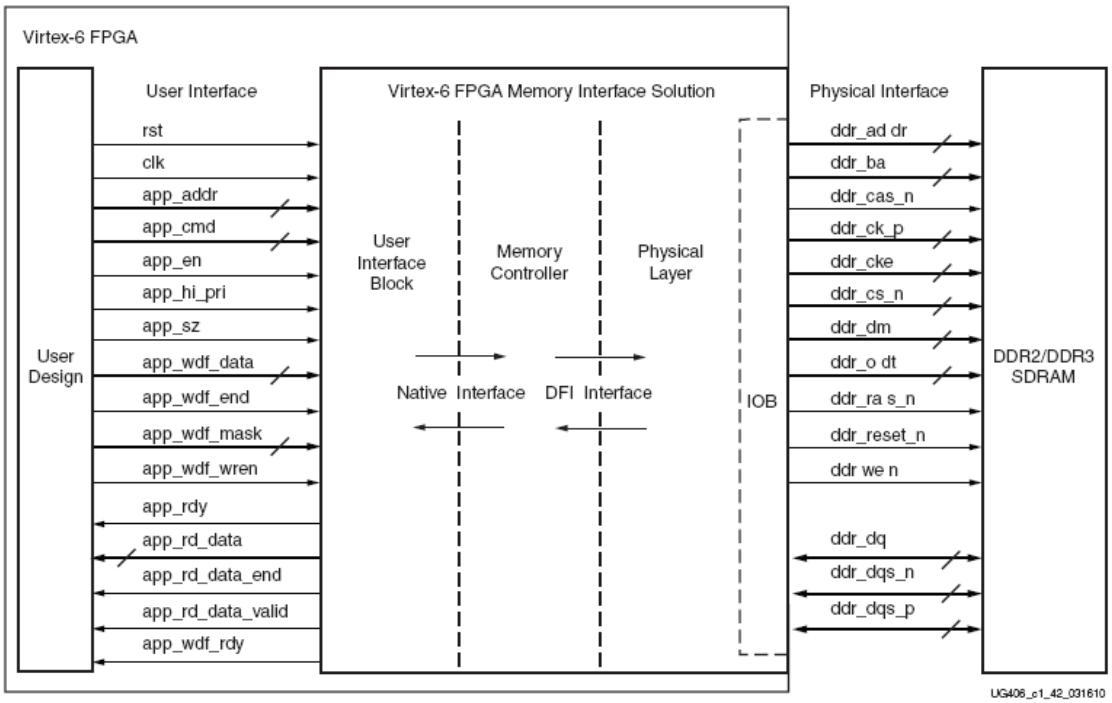

Figure 4-2: User Interface between User Design and Memory [3]

4.2.1.1. Bursts

We chose to implement DDR3 utilizing the maximum allowed burst length of 8. This ensures maximum data throughput. The actual memory has a 64-bit interface. A burst of eight (BL8) requires eight 64-bit packets of data to be sent to memory in a row. Since the Xilinx user interface is 256 bits wide, the arbiter sends 256-bit packets of data at a time to the memory controller user interface. Two 256-bit packets, or 512 bits total, is a burst of 8. Only one address and command is sent for every 512 bits sent.

4.2.1.2. Sending a Command and Address

For every data packet of 512 bits sent, an address needs to be sent to memory. To send an address app_cmd, app_addr, and app_en must be set for one clock cycle. Only when app_en is asserted, are the values on app_cmd and app_addr sent. If the app_rdy signal is de-asserted, the values will not be sent to memory and must be held until app_rdy is high again. Refer to Figure 4.2-2 which shows app_rdy low avoidance.

app_cmd: Indicates current request (read/write).

app_addr: Address for current request.

app_en: Asserted in order to send command and address. (101)

app_rdy: Indicates whether or not the UI is ready to accept commands. (75)

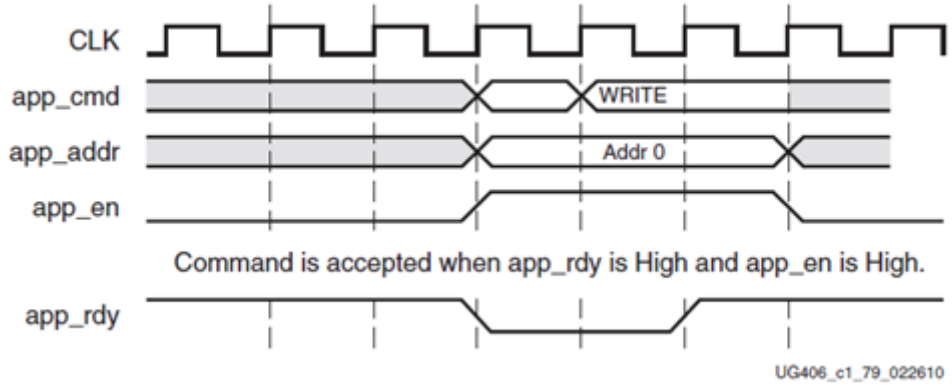

Figure 4-3: Sending a Command and Address to the Memory Controller [3]

4.2.1.3. Writing

The user interface has two modes of writing with a burst type of BL8. If the user is sending one packet of data at a time, it must be within two clock cycles of when the address was sent to memory. However, if the user is writing back-to-back, one packet of data right after another, then there is no limit for how much earlier or later it is sent to memory compared to the corresponding address. [3] Our design sends data back-to-back, yet always sends the address with the data since there is no obvious benefit to sending them independent of each other. In this way, we can always be sure the correct data is being sent to the correct address.

Communication between the arbiter and the Xilinx memory interface/controller utilizes the **app** signals provided by the memory controller. These signals can be seen in table in Figure 4.2-3. In Figure 4.2-4 a simulation example for writing is shown.

App_wdf_data	Holds the data being sent from the arbiter to the memory controller. Two 256 bit packets must be sent back-to-back.
App_wdf_wren	Asserted to indicate when data is being sent.
App_wdf_end	Indicates the end of the 512 bit data burst. Asserted on the second 256 bit packet.
App_address	Address set for entire 512 bit packets in our design. Only sent once when app_en is asserted.
App_cmd	Command set to 0 for writing. Sent at same time as app_address.
App_en	The command and address are not sent until app_en is asserted. It is asserted for only one clock cycle unless app_rdy is low.
App_rdy	Addresses and Commands cannot be sent when app_rdy is low.
App_wdf_data	Data cannot be sent when app_wdf_data is low.

Figure 4-4: Signals Used for Writing [3]

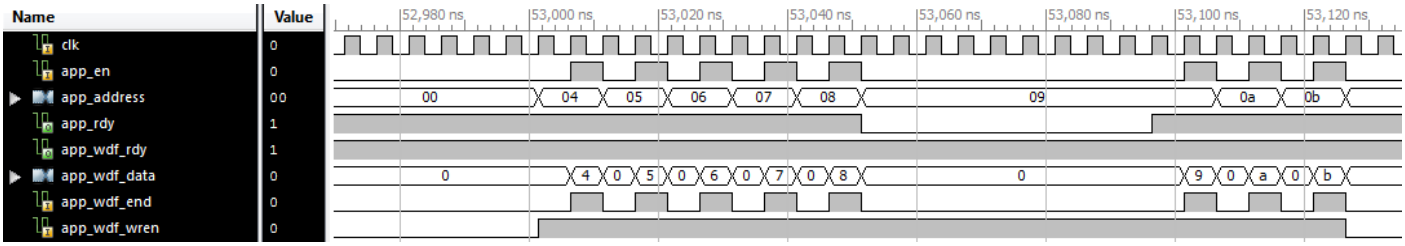

Figure 4-5: Simulation Screen Shot of Writing

As seen in Figure4.2-4, app_wdf_end is asserted for every other 512 bit packet since data is sent back-to-back. When the data 0,5 is sent, app_wdf_end is asserted at the same time 5 is sent. Address and data stop being sent for a period of time after data 0,8 is sent. This due to the design's app_rdy low avoidance. When app_rdy is de-asserted, the address and command values are held until app_rdy is asserted again. App_rdy goes low whenever a refresh occurs or whenever the memory controller cannot accept commands.

4.2.1.4. Reading

In reading, the only signals the user controls are the command, address, and app_en signals. The other signals are the read data, read valid and app_rdy signals. Like in write, there is app_rdy avoidance. The table of signals is shown in Figure 4.2-5, and a simulation waveform example is shown in Figure 4.2-6.

App_cmd	Set to 1 for reading. Transmitted to memory controller when app_en is asserted.
App_addr	Set to address. Transmitted when app_en is asserted.
App_en	Sends app_cmd and app_addr.
App_read_data	With some latency, the data is returned on this bus.
App_rd_data_valid	The data on app_read_data bus is valid when app_rd_data_valid is asserted.
App_rd_data_end	Asserted on every second 256 bit packets to indicate the end of the 512 bit packet.
App_rdy	The command and address will not be sent when app_rdy is low.

Figure 4-6: Signal Used for Reading [3]

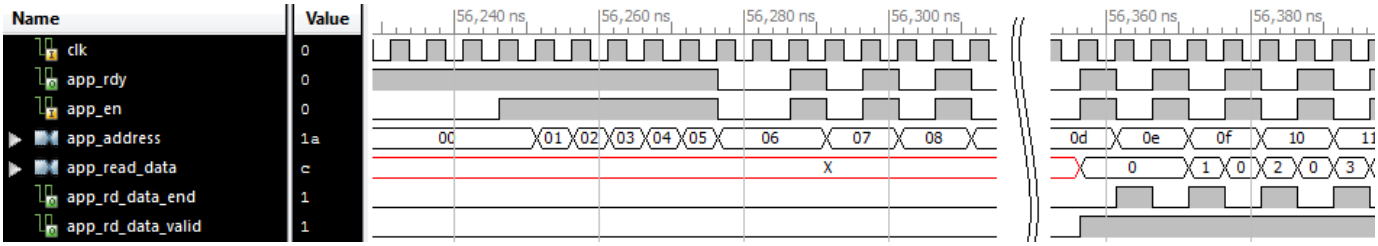

Figure 4-7: Simulation Screen Shot of Reading

The read addresses are sent to the memory controller back-to-back as seen in Figure 4.2-5. Although not shown, the app_cmd signal is set to zero at the same time app_addr is set. Above the app_address signal, app_en is shown to stay high when appropriate. When address 6 is being sent, however, app_rdy goes low and app_en goes low as well to avoid skipping the address and moving on to the next. Once app_rdy is high again, app_en goes high for one clock cycle to send address 6. After the break in the waveform, the start of returning data can be seen. The data returns back-to-back on app_rd_data, app_rd_data_end is asserted for the second part of every 512 bit packet, and when app_rd_data_valid is asserted, the state machine uses the data.

4.3. Arbiter Design

The development process for the arbiter logic involved establishing design specifications, developing its architecture and flow, and creating mathematical models for evaluating its performance. The architecture of the arbiter was developed to meet specifications defined by Teradyne, as well as from a careful evaluation of the tools and technology available to the team.

4.3.1. Product Design Specifications

This section describes the main specifications of the arbiter. The MQP team was asked to develop an FPGA module that allows two unsynchronized systems to communicate within the same DDR3 memory space. Additionally, the design needs to enable the two systems to communicate to the DDR3 memory at through-put speeds exceeding 8Gb/s.

4.3.1.1. Two systems to one DDR3 memory arbitration

The first and most important specification of the design states that the module should allow two unique systems to communicate with a single memory. The memory controller core that is available from Xilinx only allows for a single system to communicate with the DDR3 memory. Additionally, in using the Xilinx core, the system has to send read and write requests exclusively from one another. The Xilinx memory controller core provides the backbone for the project, and was used for communication between the arbiter design and DDR3 memory.

4.3.1.2. Synchronizing I/O for differently clocked systems

It is important to allow the arbiter design to connect to multiple systems which may operate at different clock frequencies from the memory controller and from one another. FIFO (i.e. first in, first out) blocks were used heavily in the design for the purpose of synchronizing the data flow between the memory controller and the systems.

4.3.1.3. 8Gb/s Bandwidth

The factor which distinguishes the performance of the design from its functionality is the specification indicating that the arbiter should be able to achieve a through-put data rate of more than 8Gb/s. This specification led the team to carefully evaluate the memory controller core to understand the most efficient way to communicate with it. It was determined that the controller operates at its peak performance when the system sends read and write commands in chains of BL8 bursts. Each BL8 burst takes two clock cycles to complete and consists of two 256-bit data packets. With the memory controller's maximum operating frequency at 200MHz, the data-flow should be able to reach speeds of up to around 51.2Gb/s.

$$Maximum\ Speed \cong \frac{256\ bits}{cycle} * f_{maximum}$$

$$f_{maximum} = 200MHz = \frac{200\ million\ cycles}{seconds}$$

$$Maximum\ Speed \cong \frac{256\ bits}{cycle} * \frac{200\ million\ cycles}{second} = \frac{51.2\ Gigabits}{second}$$

This speed is confirmed by the ML605 Hardware User Guide which states that the evaluation board's DDR3 SDRAM has been tested to 800MT/s (UG534, 17).

Having the memory controller be split between two systems means that each system would be able to achieve a data flow of half that rate (i.e. 25.6 Gb/s). However, the bandwidth of one system is greatly affected by both the speed of the second system as well as by the speed of the memory controller. The relationship of how the speed of one system affects the speed of the other is explained in the later part of this chapter.

4.3.1.4. Implicit Specifications

It is important to consider implicit requirements which may have been overlooked in the problem statement. Meta-stability is often an issue that must be addressed when designing any system that acts as a client to two servers. Meta-stability is a phenomenon that is encountered when two systems send a command to an arbitration block at the same instance, and the arbiter is unable to determine which system should gain control of the data-path. This issue was addressed by the design's state-machine's see-saw data flow which periodically switches control to the arbiter from one system to the other, regardless of whether the systems require it. As a result, the arbiter is pro-active in giving out control to the systems, rather than reacting to systems requesting control. This design scheme fixes meta-stability issues at the expense of giving up performance. For example, if one system is very active in utilizing the memory and the other system is completely idle, the memory controller will not be fully utilized. The idle system will still be given control to arbiter even as it doesn't need it, when the control could have been given to the active system. This is not major issue considering that even if each system is given control to the memory for only half of the time, it may still achieve a bandwidth of around 25.6 Gb/s.

Another implicit specification that has strongly influenced the design of the arbiter is preserving memory coherency between the two systems. To meet all of the specifications, the arbiter design needed to incorporate FIFOs for buffering data as it flows between a system and the memory controller, and vice versa. Buffering commands and data needs to be done very diligently to avoid rearranging orders between commands within the state machine of the arbiter. For example, if one system sends write commands during its turn, and then the second system sends read requests to that same memory space, the read-back must consist of the data that was updated by the first system.

4.3.2. Design Topology

The block diagram in Figure 4.3-1 displays the design topology of the arbiter configured for two systems sharing one memory. It shows how the two systems are connected to a single DDR3 memory, through their associated arbiter blocks, and a shared memory controller. This block diagram excludes the infrastructure module which is used to generate the clock for the DDR3 memory and the memory controller module.

4.3.2.1. example_top wrapper

The example_top module is a wrapper which contains all the other modules in the design. Its ports are mapped to the physical pins on the FPGA and are used to connect to the memory and the two systems. In addition to providing physical mappings to the FPGA chip's pins, example top is also used for wiring up all the containing modules. This module was inherited from an example design created by Xilinx memory interface generator. Originally, it contained traffic generator modules which could be used to test the DDR3 memory on the ML605 evaluation boards. During the development of the arbiter, these test modules were replaced with arbiter_block modules which were developed by the MQP team.

4.3.2.2. memory_controller

The memory_controller module came with the original Xilinx example design project which our design was leveraged on. It is used to simplify communication to the memory by taking care of the refresh cycle, and enabling the system to use a simple BL8 interface. In BL8 mode every address is associated with 512 bit data packets, which creates a need for the arbiter design to have some sort of serial-to-parallel converter so that it would could map to a reasonable number of physical pins.

4.3.2.3. arbiter_block

The systems are connected to arbiter_block modules which consist of all necessary FIFOs for buffering read and write commands and data before they are transferred between the system and the memory

controller. The FIFOs that are encapsulated in the arbiter_block, include two address FIFOs used for buffering the addresses for either write or read commands, a 64-to-512 write data FIFO used for buffering write data that is to be sent from system to the memory, and 512-to-64 read data FIFO used for buffering read-back data between the memory controller and the system. Write data (wd_fifo), write address (wa_fifo) and read address (ra_fifo) FIFOs are filled at system clock's frequency, and emptied at memory controller clock's frequency. Read-back data FIFO is filled at memory controller clock's frequency and emptied at system clock's frequency. The reading and writing to these FIFOs is controlled in mostly controller by the state-machine in the arbiter block that is shown in Figure 4.3-2.

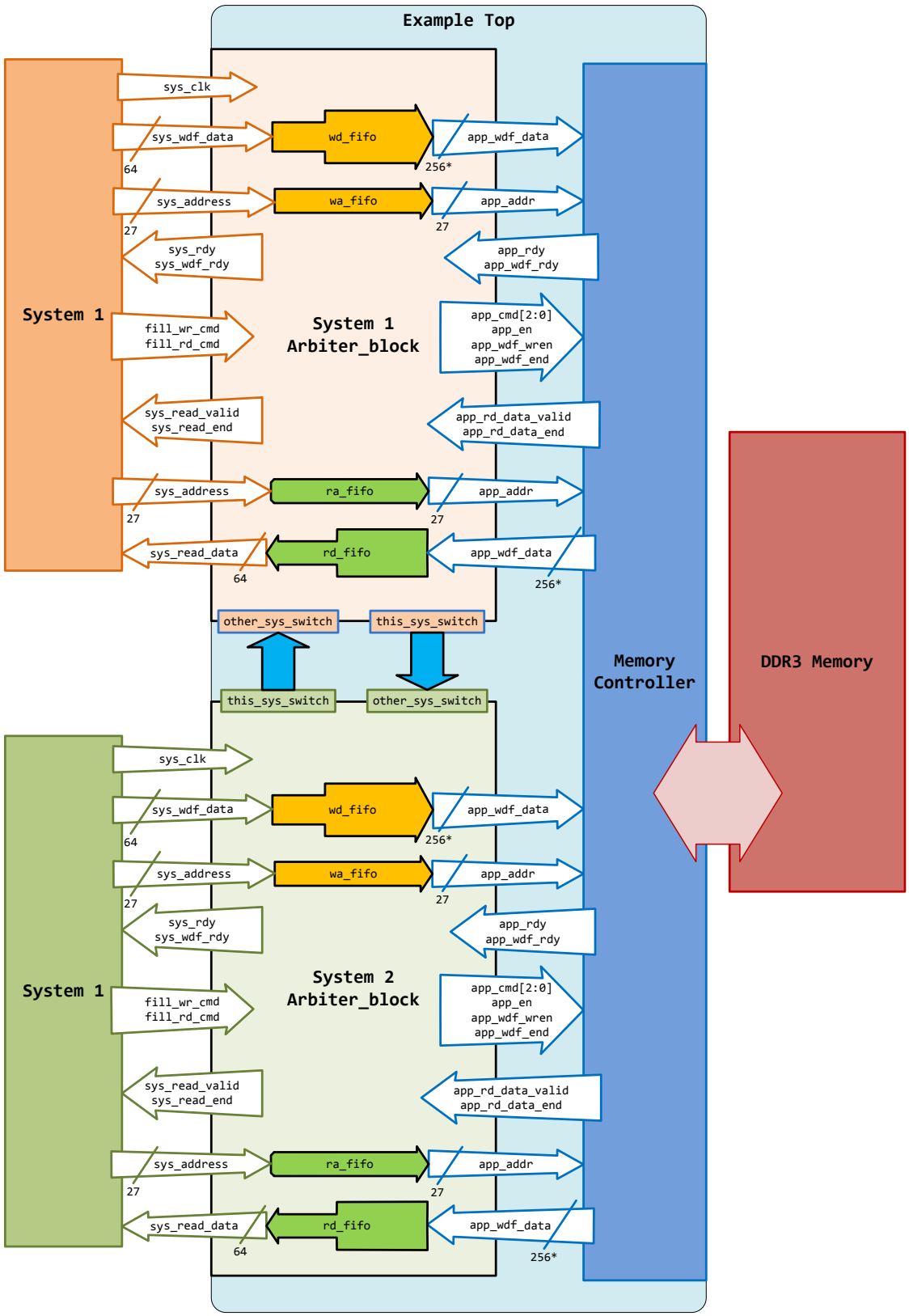

Figure 4-8: Two arbiter_blocks set-up to connect the DDR3 memory to two different systems

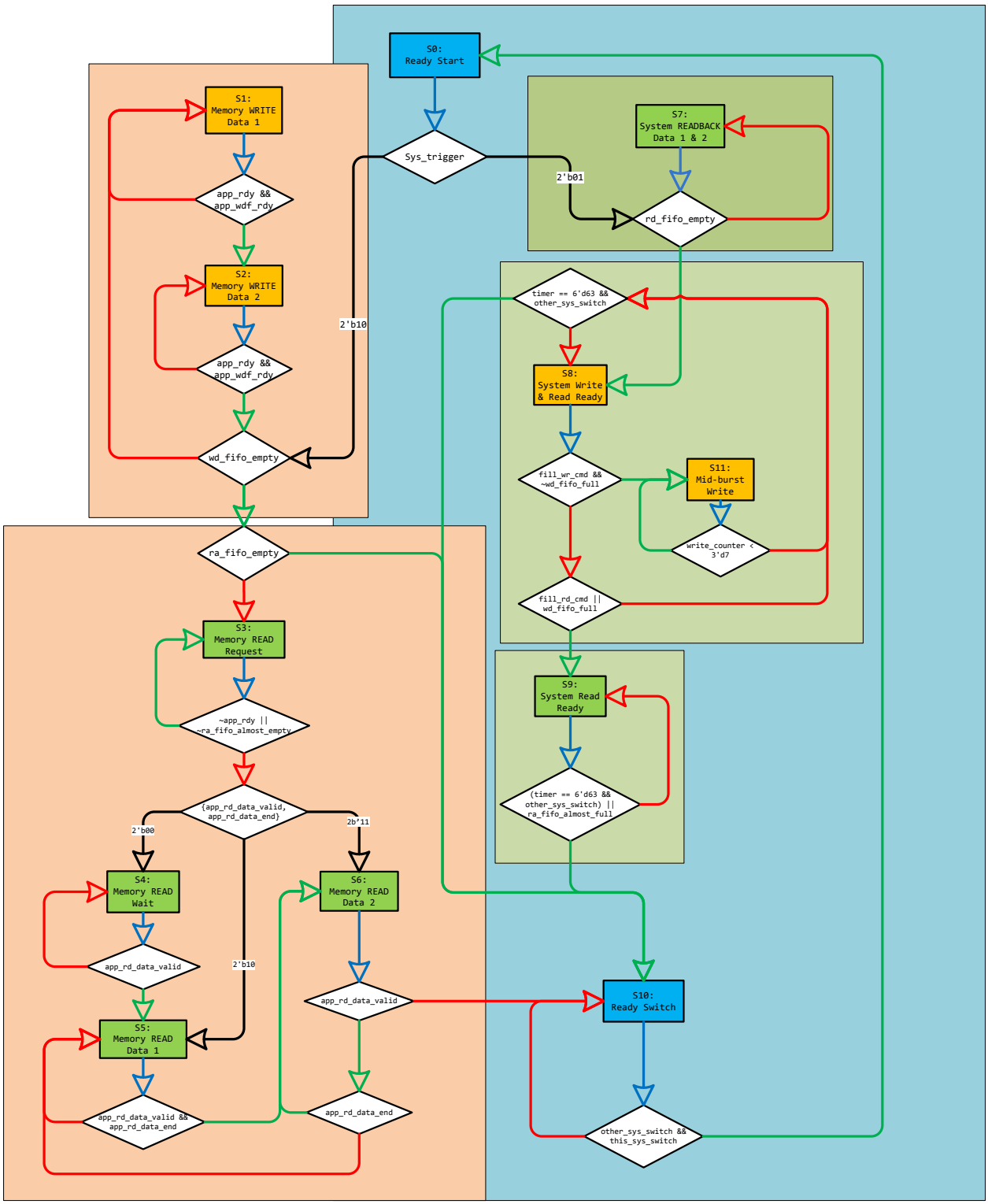

Figure 4-9: Arbiter State Machine Full

4.3.3. Arbiter Flow

The flow of the arbiter is described in the state-machine illustrated in Figure 4.3-2 above. This state-machine is broken down into two primary branches, which characterize the arbiter's see-saw like flow. The green and orange blocks shown in the state-machine distinguish these two separate branches. The green blocks identify the states when the system interacts with the arbiter block, while the orange blocks identify states where the arbiter block interacts with the memory controller. The arrows between all the smaller blocks indicate the flow of the state machine. Rectangular state-blocks indicate states (Designated by a label S_), and the rhombus shapes indicate conditions which are used for determining the flow between the states. Blue arrows show unconditional flow from states to their first conditions, while green and red arrows demonstrate the paths representing Boolean value which result from the evaluation of the condition blocks from which they exit. Green arrows indicate logical true paths, and red arrows indicate logical false paths. Black arrows show paths from conditions which cannot be evaluated to binary values.

In a two system configuration, the two arbiter blocks are always present in opposite branches. For example while system 1 is in the orange branch, System 2 has to be in the green branch. Once both arbiter blocks reach the end of their branches, they return to state S0 and switch to opposite branches. This state-machine is continuously flowing, so there is no ready or idle state. During the green branch, the system interacts with the arbiter block and is either sent back the read-back data for its previous read requests, or is allowed to send new write and read commands. During the orange branch, the arbiter_block executes any commands that may have been buffered in its FIFOs. To explain the state machine in greater granularity, the flow of the state-machine will be followed starting from a system being granted control of the arbiter at the block shown in Figure 4.3-3.

4.3.3.1. System-to-Arbiter Write

The portion of the state machine shown in Figure 4.3-3, displays the flow where a system is granted permission to either send read or write commands. During state S8, the arbiter_block asserts both its sys_rdy and sys_wdf_rdy signals. This notifies the system that it has permission to communicate with the memory (through the arbiter). When the system first enters S8 the system can either send write commands or read requests to the arbiter.

To send a write request, the system must assert the fill_wr_cmd, after having set starting address and the first 64-bits of data on their respective busses. Then, for the following seven clock cycles, the arbiter enters state S11 where the system must send the remaining 448 bits of data, 64 bits at a time. During these 7 clock cycles the arbiter will pick-up whatever is on the data bus and use it to fill the wd_fifo. During state S11, the system no longer has permission to send any new commands, and this will be evident by the de-asserted sys_rdy signal. Once the system finishes writing a full 512 bit data burst, the arbiter's state machine returns to state S8, where it awaits for a new command from the system. To exit out of the S8-S11 loop, the system must be at S8 and one of the following events must happen: the systems sends a read request or arbiter_block timer expires.

- Arbiter Timer

The arbiter_block module contains a timer register which is used to keep track of how many clock cycles have elapsed since the system has been given permission to communicate with the arbiter. This is used to limit the number of clock cycles the system is given every time it has a turn.

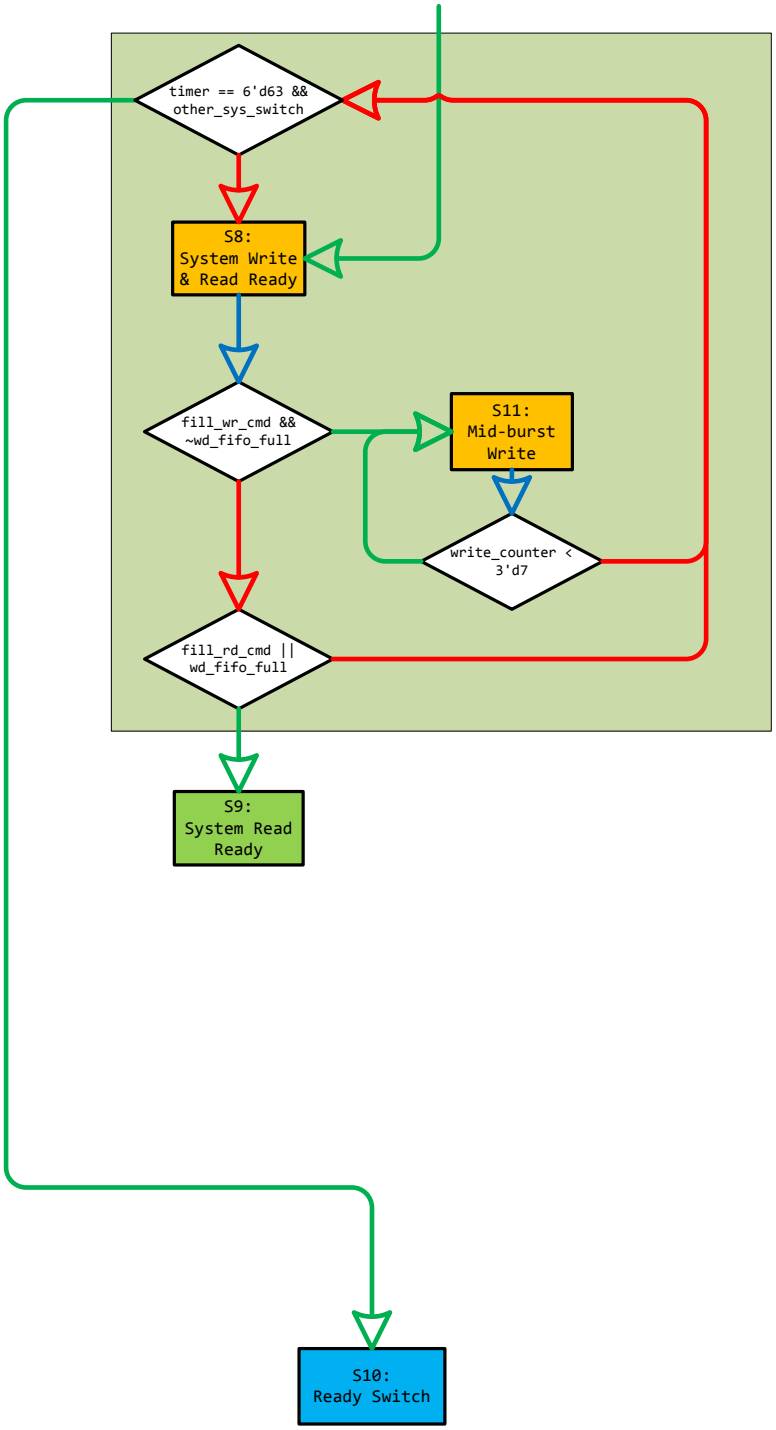

Figure 4-10: System-to-Arbiter Write

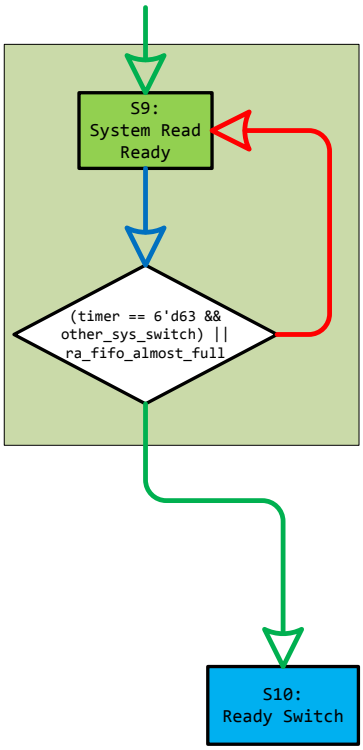

Figure 4-11: System-to-Arbiter Read

4.3.3.2. System-to-Arbiter Read

To send a read command the system needs to assert fill_rd_cmd, after having set the address from which it would like to retrieve 512 bits of data from. If the system sends a read request from state S8, the state-machine transitions to state S9, where the arbiter keeps the sys_rdy signal asserted but de-asserts the sys_wdf_rdy signal. During this state the system is no longer allowed to send write commands, but it may continue to send read requests for as long as the arbiter_block timer register has not expired, and the read address FIFO has not filled up. Once all the read requests are sent, the system will need to wait for the read-back data which is returned when the system's arbiter re-enters this branch after passing through arbiter-to-memory execution branch.

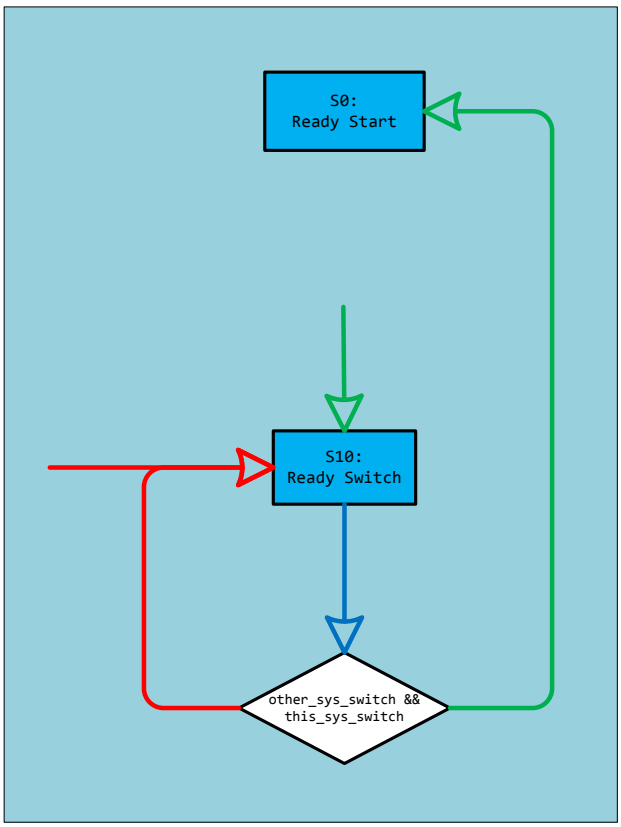

Figure 4-12: Switching Branches

4.3.3.3. Arbiter Branch Switching

Once the arbiter block completes its full branch, it is halted at state S10. At this branch the arbiter_block will assert its sys_switch signal which will notify any other arbiter block that it has completed its branch. Once all arbiter_blocks have their sys_switch signals asserted, they will enter state S0, where they will switch branches and resume operation. In the flow described in this chapter, the arbiter_block leaves the system-to-arbiter branch where it buffered commands coming from the system, and enters arbiter-to-memory branch where the buffered commands will be executed to the memory.

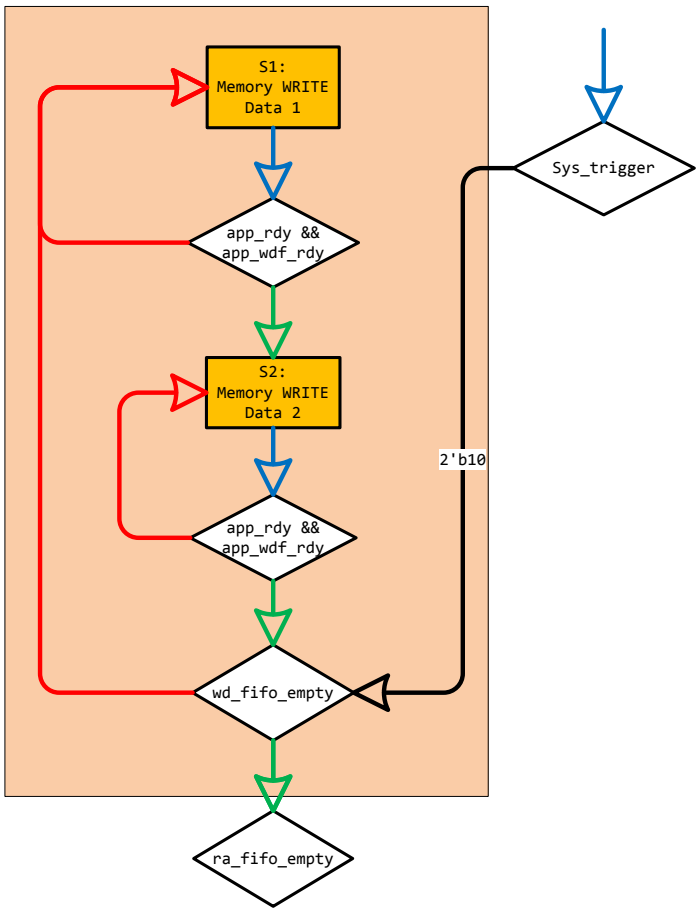

Figure 4-13: Arbiter-to-Memory Write

4.3.3.4. Arbiter-to-Memory Write

When the arbiter_block enters its Arbiter-to-Memory execution branch, it first executes all buffered write commands. The write addresses and write data are read from the wa_fifos and wd_fifos, at same rate. The data is than written to its corresponding addresses on the SDRAM using the XIlinx memory controller core.

Writing a single 512 bit data packet takes two clock cycles because the data is split into two 256-bit packets. The first packet is sent in state S1, while the second part of the data is sent during state S2. This flow takes into account memory controllers 's app_rdy and app_wdf_rdy which are used for telling the arbiter that the controller is not ready to receive new data or commands. This most often happens when the memory controller refreshes the SDRAM. The arbiter_block remains in S1-S2 loop for as long

as wd_fifo still has data buffered inside it. Once it is empty, the arbiter_block exits this loop, and attempts to execute buffered read commands stored in ra_fifo.

4.3.3.5. Arbiter-to-Memory Read

After the arbiter executes any buffered write commands, the state-machine enters its arbiter-to-memory read section shown in Figure 4.3-7. In state S3, the arbiter_block empties out any buffered read request from the system and passes them into the memory controller. The state-machine remains in state S3, for as long as ra_fifo contains buffered read addresses.

After the arbiter sends all the read requests, it will expect to receive read-back data from the memory controller. The read-back data can arrive in any of the states shown in Figure 4.3-7. If the ra_fifo has a long list of buffered read commands, it is possible for the first read-back data to arrive before arbiter block has finished emptying out ra_fifo. If the read back data arrives after all the read requests are sent, then it will arrive during states S5 and S6. The memory controller communicates to the arbiter indicating that there is read-back data on the data bus by asserting its app_read_valid signal. This signal is used by the arbiter logic for capturing the data and so it may be pushed inside the read data FIFO, rd_fifo. Once the last of the read-back data is sent back from the memory controller and filled in the rd_fifo, the state-machine enters state S10, and waits for the other arbiter to finish its branch. The data which is buffered in rd_fifo returns to the system at the start of the next branch.

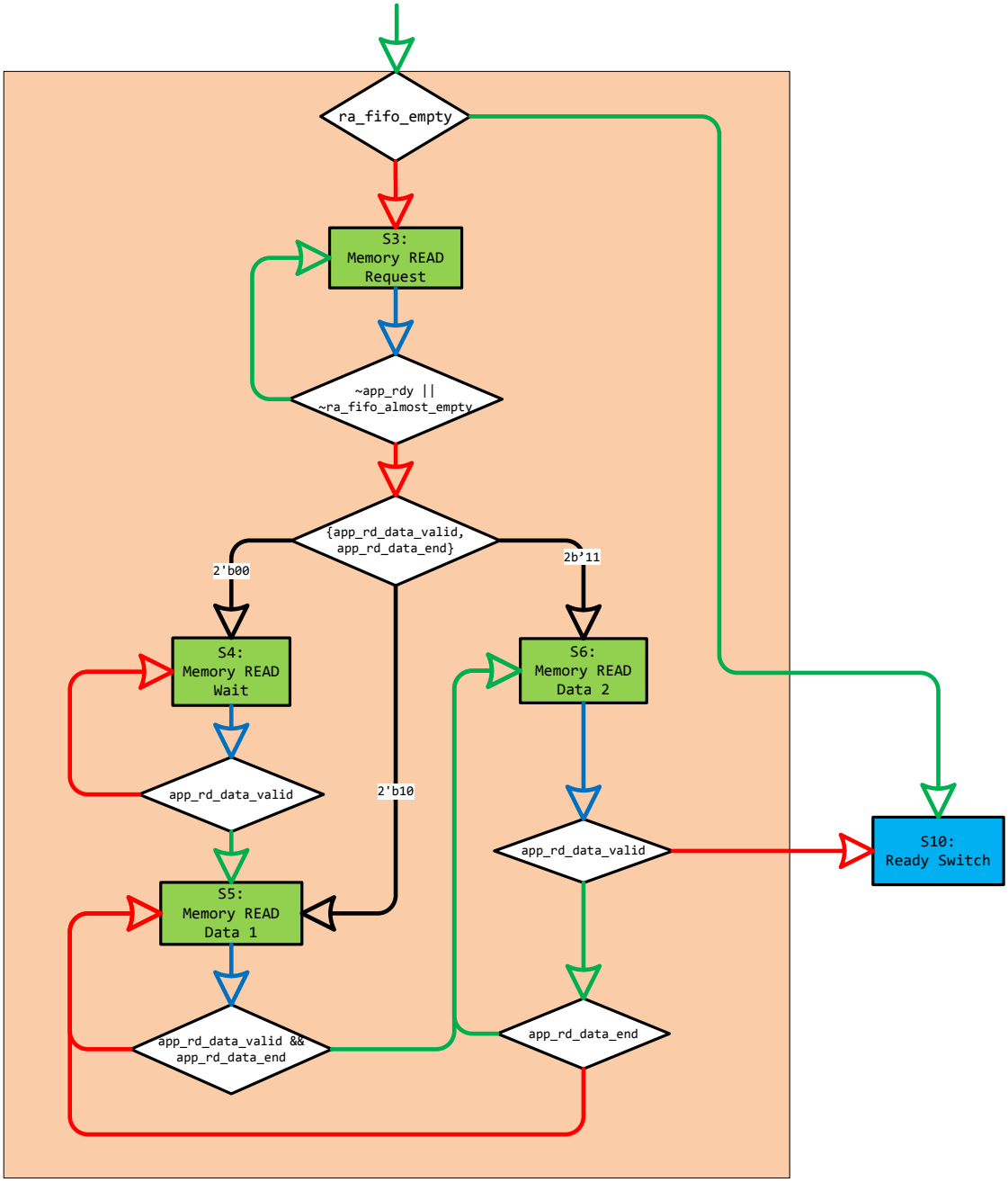

Figure 4-14: Arbiter-to-Memory Read

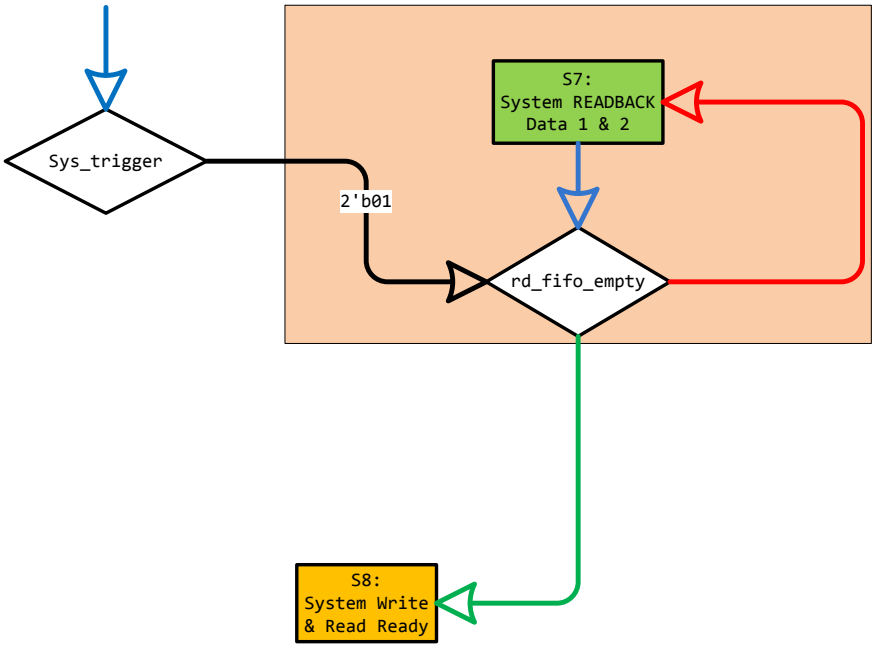

Figure 4-15: Arbiter-to-System Read-back

4.3.3.6. Arbiter-to-System Readback

The arbiter_block enters state S7, when it first enters system-to-arbiter branch and its rd_fifo contains buffered read-back data. During this state, the read-back data is emptied out of the rd_fifo and sent back to the system, 512 bits at a time. However each 512 bit data is broken down to a chain of eight 64 bit packets which are returned during a series of eight clock cycles. The system is made aware that the read-back data is present on its data bus by sys_read_valid and sys_read_end signals. An asserted sys_read_valid indicates that there is valid read-back data on the bus, while an asserted sys_read_end signal indicates that the 64-bit data packet on the bus is the last data packet of the 512-bit data stream.

4.3.4. Determining Arbiter's Performance

The arbiter's maximum performance is ultimately capped by the frequency of the memory controller. Typically this frequency runs between 100MHz and 200Mhz, and the memory controller is able to send and receive 256 bits every clock cycle. Because systems are allowed to send 64 bits at every clock cycle, they would operate at highest efficiency at clock speeds that are four times faster than the memory controller. If a system operates at a frequency lower than that, the arbiter will be under-utilized by the system. If the system runs at a clock that is faster than four times the memory controller frequency, than the arbiter will become overwhelmed and will dynamically cut-down the system's effective bandwidth.

While the memory controller clock sets the cap on the arbiter's performance, it is important to note that the frequencies of the systems also have a limiting effect on each other's effective bandwidth. It is possible to derive a mathematical model which can be used to estimate the performance of the arbiter. First the effective frequency for a particular system is derived by finding the ratio of how many cycles a system is given control by the arbiter per second.

$$f_{effective1} = \frac{n_1}{\left(\frac{n_1}{f_1} + \frac{n_2}{f_2}\right)} = \frac{n_1 * f_1 * f_2}{n_1 * f_2 + n_2 * f_1}$$

n_1 and n_2 represent the number of cycles system 1 and system 2 stay in control for each time they are given permission to send commands. $f_1$and $f_2$are the frequencies of the two systems. In the above derivation the total time for an arbiter to complete a full cycle in its state machine is expressed as $\left(\frac{n_1}{f_1} + \frac{n_2}{f_2}\right)$. The effective frequency of one of those systems may be evaluated by taking the ratio of the number of cycles that system has control for by the total time. The data flow may then be evaluated by factoring the effective frequency by 64, which is the

number of bits that are sent and received by each clock cycle(data bus width). The model for the data rate would than equal:

$$BW_{effective1} = \frac{64 * n_1 * f_1 * f_2}{n_1 * f_2 + n_2 * f_1}$$

By default, the arbiter is balanced so that both systems are given the same number of clock cycle for sending their commands. This simplifies the model further into:

$$BW_{effective1} = \frac{64 * f_1 * f_2}{f_2 + f_1}$$

First thing that can be noted from this model is that both systems are balanced to have the same effective bandwidth even when operating at different frequencies. This is very useful characteristic as it enables for two systems, regardless of their individual speeds to have the same speed for communicating with the memory.

Additionally it is interesting that the bandwidth equation for the arbiter seems to closely follow the equation for calculating the equivalent resistance of two resistors connected up in parallel with one another.

$$R_{equivelent} = \frac{R_1 * R_2}{R_2 + R_1}$$

To interpolate this analogy even further it is possible to compare our arbiter to an analog-relm resistor load model where the analog properties such as voltage, current and resistance equate to the digital properties of our arbiter shown in the following table.

Arbiter	**Simple Analog Resistor Network**
Frequency (frequency)	Resistance(Ohm)
Data bus width (bits/cycle)	Current(Amps)
Bandwidth (bits/second)	Voltage(Volts)

$$BW_{effective1} = width_{databus} * \frac{f_1 * f_2}{f_2 + f_1} \ \ll\gg \ V = I * \frac{R_1 * R_2}{R_2 + R_1}$$

5. Arbiter Validation

As mentioned in the methodology chapter, there are two approaches to validating the design; quicker validation of the simulated logic using iSim waveform viewer and Verilog test-benches, and more precise but slower validation of the design implemented on FGPA using Chipscope. Validation must also be done at two different stages of the development process; one in parallel with the development of the design, and another one once the design is finished. Development stage validation focused on observing the behavior of waveforms in the design and making sure that their behavior follows functionality and timing which the MQP team intended. Once the design is fully developed, it is also very important to create functionality checkers, which can be used to more reliably validate that the final arbiter design meets all of its specs.

5.1. Emulating Systems

In order to validate the arbiter, it is necessary to provide it with stimulus that would model the behavior that can be expected from the systems. A stimulus module was created to act as instances of systems communicating with the arbiter. These fake systems could be triggered to send chains of write or read commands by an assertion of a signal. The stimulus module sends its command chains in accordance to the arbiter's protocol. It has a trigger for enabling a write chain of 32 data packets with values 0-31 to addresses 0-31, as well as trigger for enabling a read chain for addresses 0-31. In a typical validation flow the write chain is triggered first, and the generated waveforms are observed to assure that the logic is behaving correctly. Once the write chain finishes, the read chain is triggered and similar observations are made on the generated waveforms. Because the read chains reads from the same address which the write chain had previously written to, the resulting read-back data is expected to match the write data. For the simulation flow, the arbiter design is connected to three logical models. A model of the DDR3 memory provided by Xilinx, as well as two models, one of each system, sys1, and sys2 which were

created by the MQP team. The models for the systems are instances of the stimulus module and their sole purpose is to send read and write commands to the arbiter. The following section describes typical development stage validation process, as well as provides more details on how the arbiter operates at signal level.

5.2. Development Stage Validation

During the development of the arbiter design, the logic was periodically validated. Making incremental steps in the development best describes the strategy with which the MQP team tackled this project. By validating the design every time a change was made, the team was able to effectively catch bugs before they could accumulate. This strategy provides a lot of control, and made the development process easily manageable. Simulating the design and observing its behavior using iSim, was the most frequently used technique in development stage validation. It is a lot more effective to run simulations of synthesized logic during the development stage, because of its relatively fast turn-around time. Hardware validation using Chipscope requires the synthesized logic to pass an additional implementation phase which can take up to 12 more minutes.

5.2.1. Switching branches

Figure 5.2-1 shows one of the systems switching into the system-to-arbiter branch where system, sys1, is given permission to either send read or write commands to the arbiter. Once sys1 enters the system-to-arbiter branch it enters state 8. The register keeping track of how many clock cycles the system has had access to arbiter for, is incremented and the arbiter notifies sys1 that it is ready to receive new read or write commands by asserting both its sys_rdy and sys_wdf_rdy signals. It is interesting to note that other_sys_switch signals is being set to high immediately after switching branches. This is because the other system, Sys2, does not have any buffered write or read commands and is just waiting in its state 10, until Sys1 finishes communicating with the arbiter.

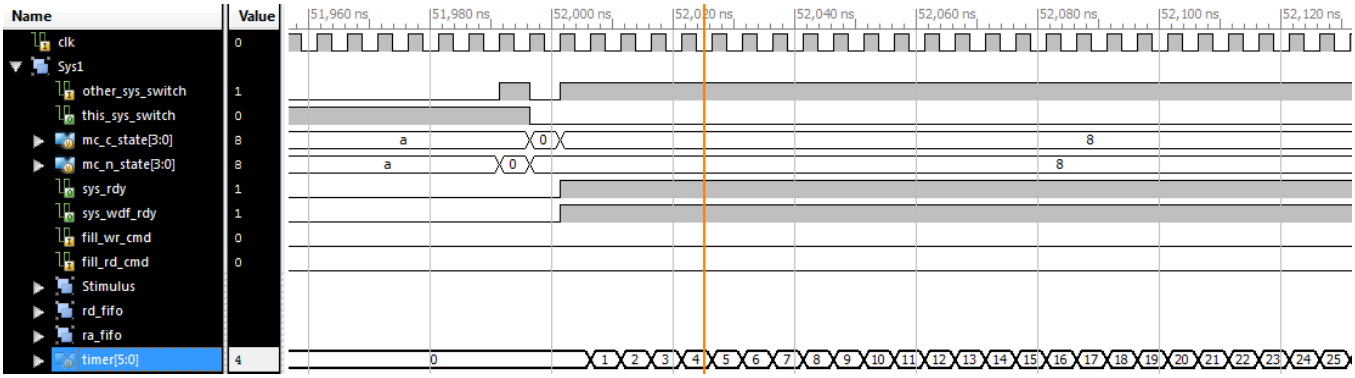

Figure 5-1: Switching Branches

5.2.2. System-to-Arbiter Write

Figure 5.2-2 below, shows how at the 34th cycle in system-to-arbiter branch, Sys1 begins sending write commands. Fill_wr_cmd signal is asserted four times, once at the beginning of each new write command. The four writes are sent to addresses zero to three, as can be noted from looking at the sys1_address waveform. The data is sent 64 bits at a time, where every 0th bit in the 64 bits makes up 8 bit value corresponding to the address. This bit is apparent on the sys_wdf_data[0] signal which starts out equaling to a binary value of 00 at address zero , and binary 11 at address three. The stimulus module which sends these commands, attempts to send write data to addresses for address 0-31, but it is halted because the timer register reaches a value of 63, so the arbiter block must exist the branch, so Sys2's arbiter block may enter it.

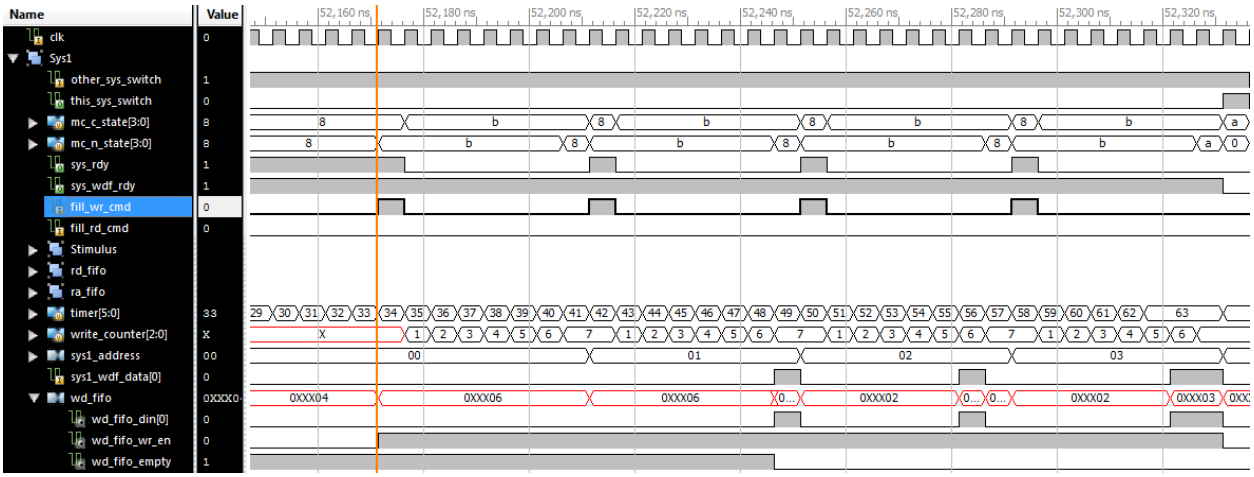

Figure 5-2: System-to-Arbiter Write

5.2.3. Arbiter executing buffered Writes

Once the arbiter switches branches, the write commands that were buffered in the previously get executed through the memory controller. As can be observed in the iSim waveform viewer in Figure 5.2-3, it takes four times less clock cycles to execute write commands then it does to buffer them. This means that the systems would perform optimally with the memory at transfer frequencies which are four times faster than the arbiter frequency. The arbiter sends write commands to the memory controller between state 1 and state 2. During state 1 the arbiter sends the first half of each 512 bit data packet and during state 2 it sends the second half of the packet. The address is sent during state 2, with the second part of the data packet.

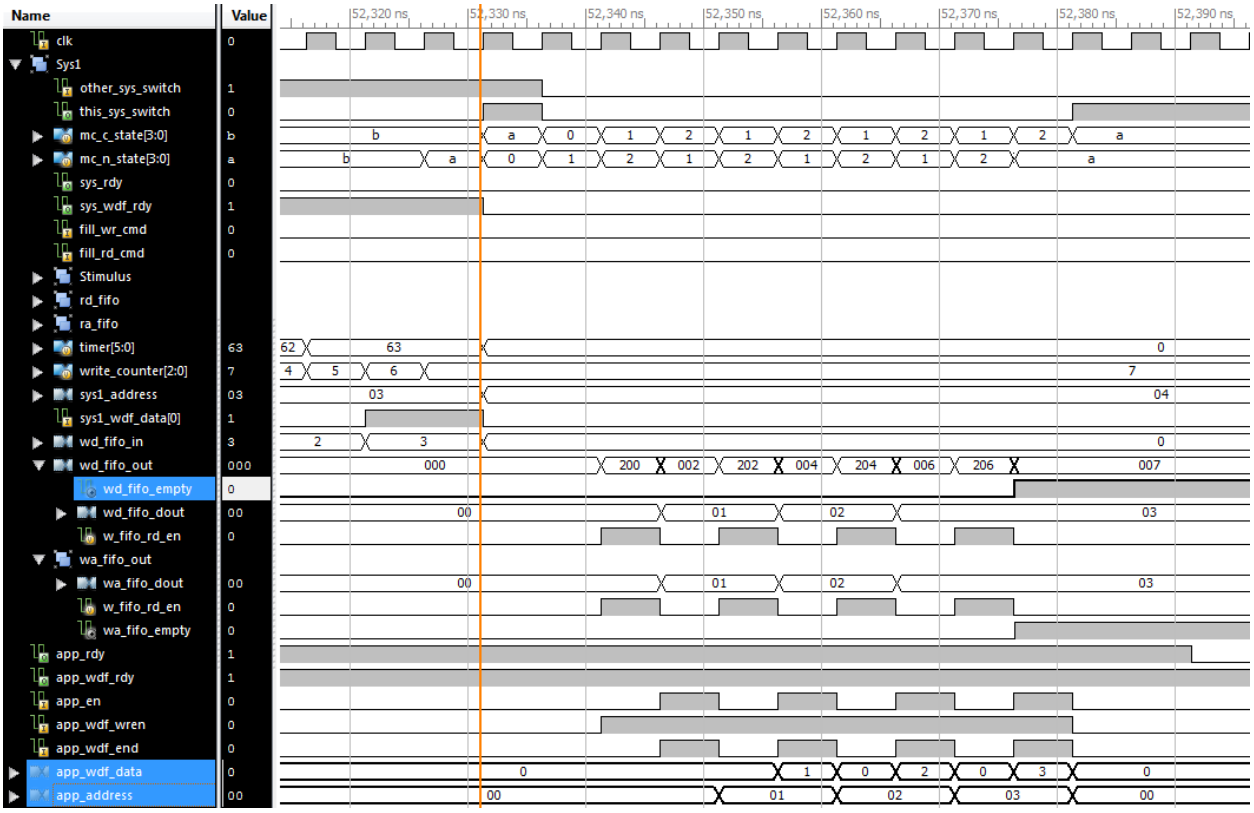

Figure 5-3: Arbiter-to-System Write

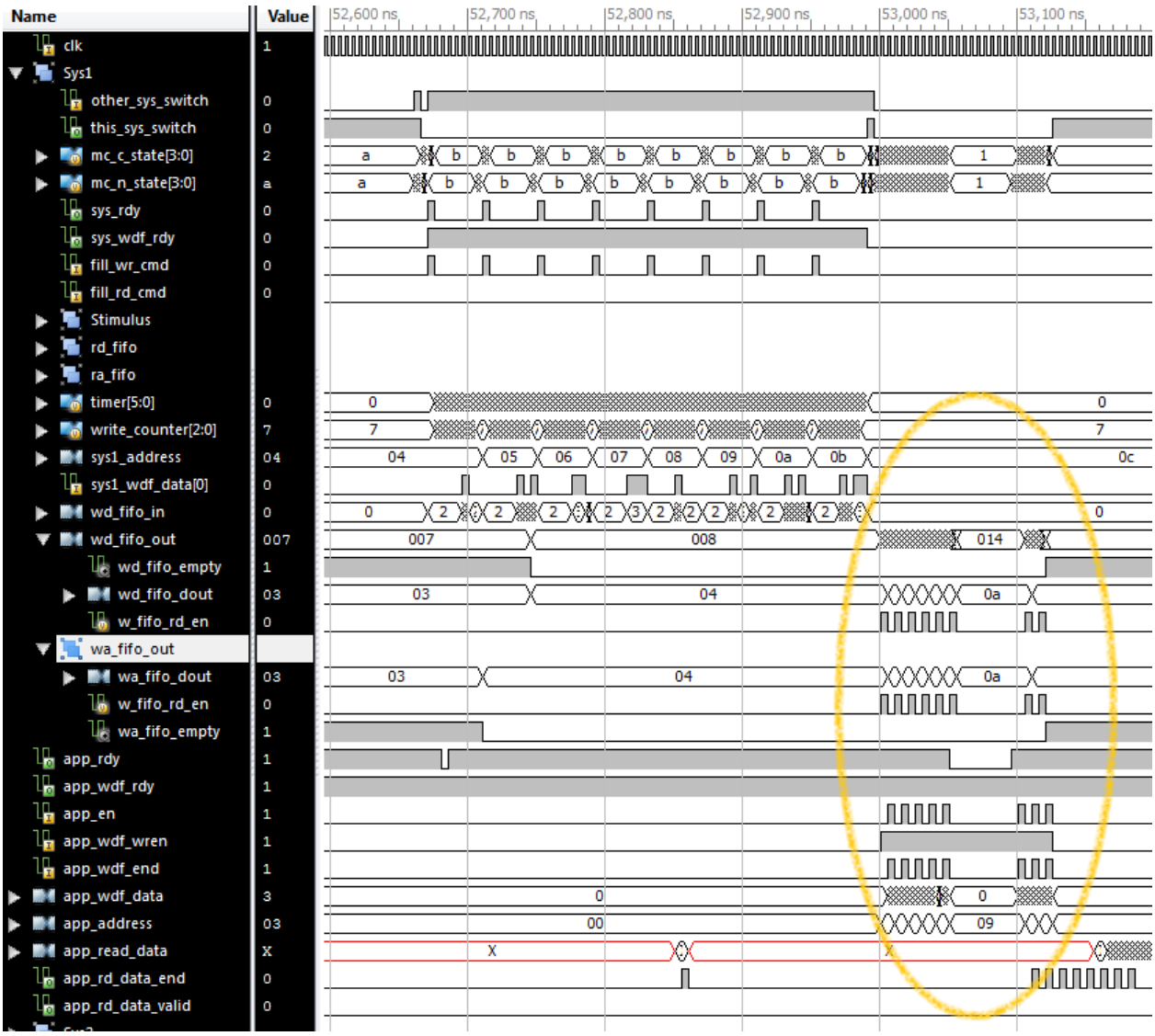

Figure 5-4: Arbiter successfuly avoiding app_rdy low

5.2.4. Validating all possible cases

Throughout the validation process it is very important to obtain as much coverage of the logic functionality as possible. Coverage means having the logic be stimulated in every way possible to reassure that no bugs will emerge in all the different possible scenarios. For example in Figure 5.2-4 above, the waveforms demonstrate additional coverage of the case where the memory controller might de-assert its app_rdy signal in the middle of the arbiter attempting to write. It is important for arbiter to be able to avoid this hazard, and as can be seen by the waveforms it does so by extending its state 1

until app_rdy is high again. It is very important to reassure that the arbiter design is able to avoid such uncommon circumstances early in the development process as to minimize the amount of bugs we would get in the final design.

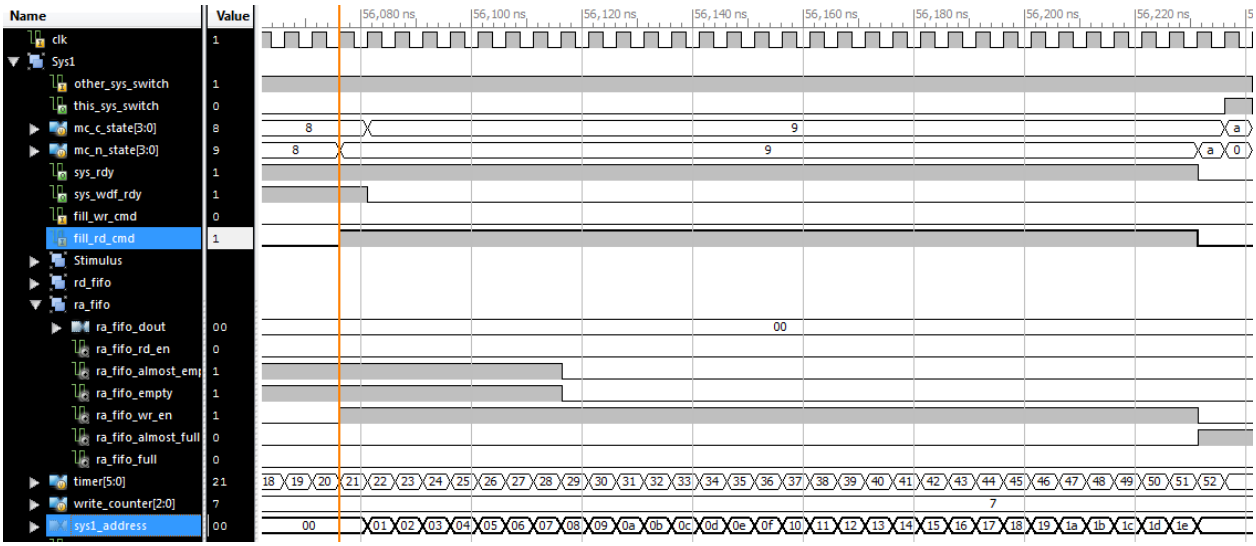

Figure 5-5: System-to-Arbiter read requests

5.2.5. System-to-Arbiter Read commands

After the write commands are sent to addresses 0-31, the stimulus module is then also triggered to send read commands to the same addresses. Figure 5.2-5 above displays the waveforms that are generated when the system attempts to read. After its first read command when the timer is at 21, the arbiter's state changes from state 8 to state 9. At state 9 the system continues to send read commands, as it is permitted to do so because of the asserted sys_rdy signal. However, sys_wdf_rdy goes down in state 9, which means that the system is not allowed to send any write commands. During this branch, the system sends read commands to addresses 0 to 18(1e), which are all buffered in the read address fifo, ra_fifo. The arbiter ends the branch when the ra_fifo fills up as can be observed at timer 52 where the ra_fifo_full signal is asserted. In its next branch, the arbiter will empty out the ra_fifo and send the buffered read commands through the memory controller.

5.2.6. Arbiter-to-System Read execute

Figure 5.2-6 shows the behavior of the arbiter block during its arbiter-to-memory branch where it empties out its ra_address fifo and sends read commands to the memory controller. The read commands are chained together and are sent at a rate as fast as the memory controller allows. At the very start the memory controller allows for the arbiter to send a new read request every clock cycle, but then the memory controller starts to periodically de-assert its app_rdy signal, and as a result the arbiter only sends new read commands every two clock cycles.

The read-back data starts to arrive from the system, before all the read commands even finish getting sent. It can be observed that the memory controller returns the correct values from the addresses zero to three.

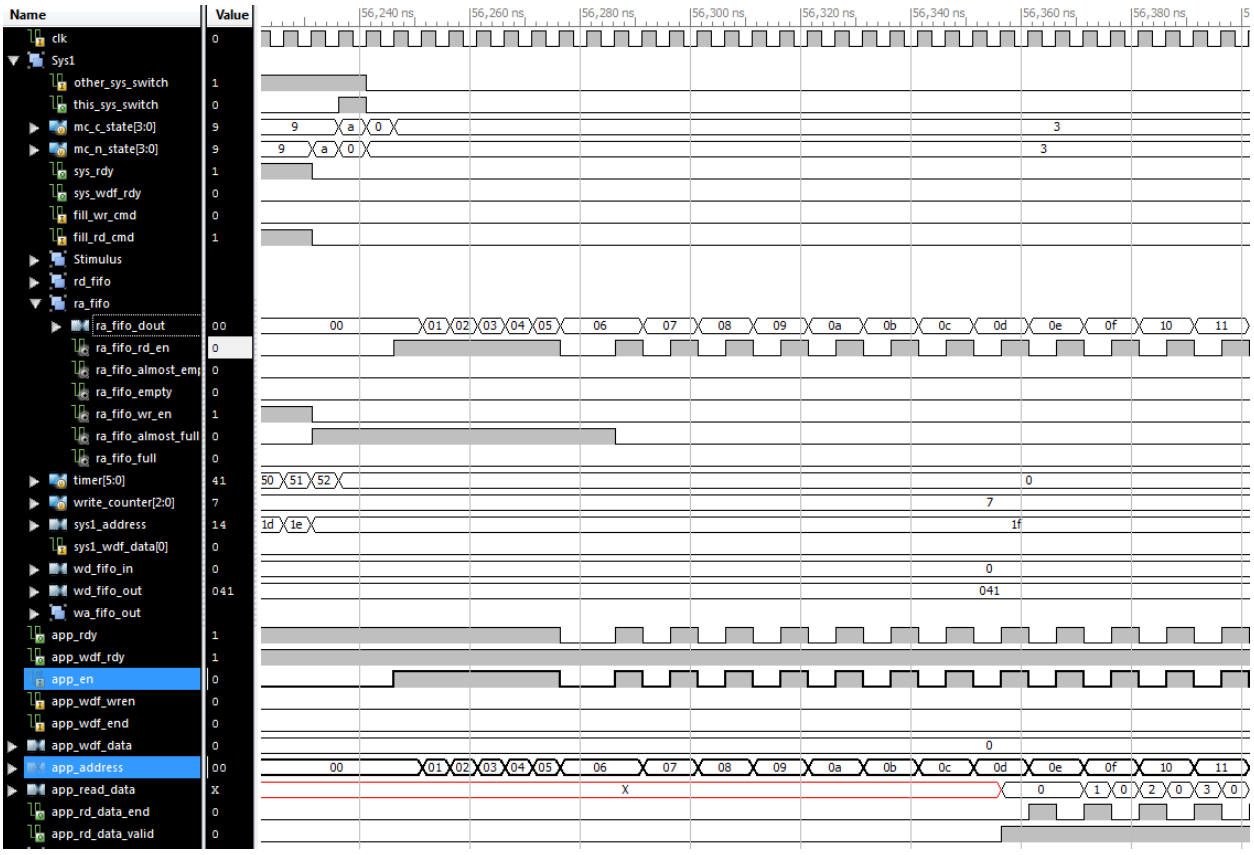

Figure 5-6: Arbiter-to-Memory read execution

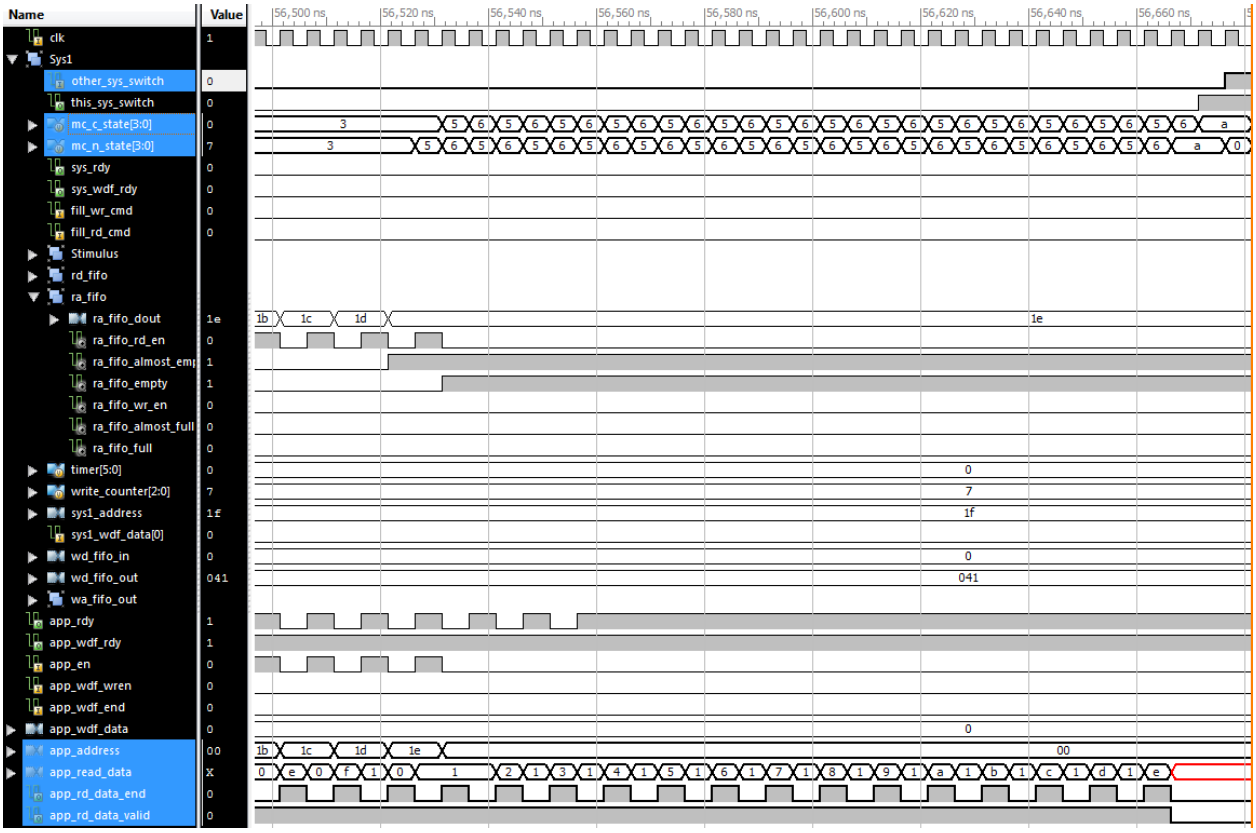

Figure 5-7: Arbiter-to-Memory read-back

Once the arbiter_block empties out all the read commands, its state-machine switches from state 3 to states 5 and 6. During states 5 and 6, the arbiter receives read-back data from the memory controller, without sending any new read commands. During states 3, 5 and 6, the read-back data is buffered in the read data fifo, rd_fifo. The buffered read-back data is not returned back to the system, until its next branch where rd_fifo is emptied.

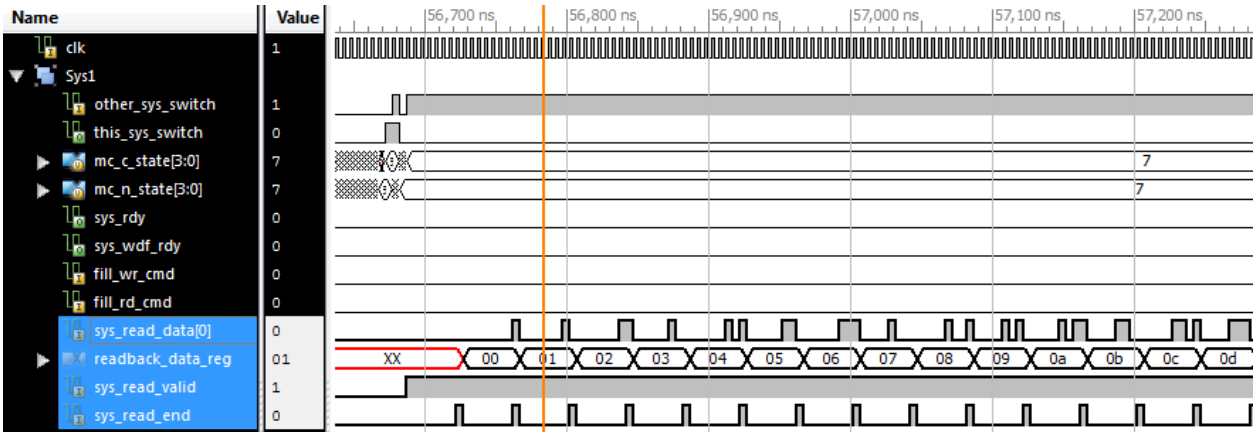

Figure 5-8: System-to-Arbiter Read-back

5.2.7. System-to-Arbiter Read-back

When the arbiter block first re-enters its System-to-Arbiter branch and finds that it has buffered data in its rd_fifo, it enters state 7. During this state, 512 bit data packets are released from rd_fifo every 8 clock cycles. These 512 bit data packets are split up into a series of eight 64 bit packets which are then sent back to the system. Sys_read_data[0] shows the 0^{th} bit of each one of these 64 bit data packets, while readback_data_reg shows eight bit representation of the 512 bit data packets as they are being released from the rd_fifo. These eight bits consist of every 64^{th} bit in the 512 bit data packet. The read-back data that the system receives appears to match the write data the system sent, validating that the arbiter design works properly. However this is very limited validation of the design and is not enough to assure that it will work correctly when operating over millions of transactions.

5.3. Functionality Checkers

Once the arbiter design has been fully developed, it is important to create functionality checkers for it which will run tests on the arbiter over millions of transactions and validate that it does not contains any bugs. Validating millions of transactions means that the team would no longer be observing waveforms manually but instead the team would focus on creating automated tests which would make the observations for them. Creating functionality checkers may require as much effort as it does to develop the design, and it may take months of work to fully validate the arbiter design developed by the MQP group.

There are two possible testing schemes that could be implemented for these checkers. One would be having two emulated system write and read data that can be expressed as a function of the address it is being written to. Then when the data is read back all the values can be compared to the values evaluated from their address. The other scheme would involve implementing an SRAM that would keep track of all the data flow as it flows in and out of the DDR3 memory. Then when the data is read back from the DDR3 memory, it would be compared with the values that were stored in SRAM.

Address dependent data

5.3.1. Data as a function of its address

One testing scheme that may be implemented is having the systems write data to the memory whose value is derived from the address that it is being written to. For example, one system could write data which is equal to 2x the address that it being written to, while the second system could write data that is 3x the address that it is being written to. In this scheme it is important to keep track of the address that were written to, and which system wrote to them, so that when the data is read back from an address, the checker knows whether the data should match to be 2x or 3x the address. To do this, the stimulus modules should be set-up to own exclusive addressing spaces on the memory. This means that the two systems would write to their own exclusive address space, and one system would never write to any

address which the second system writes to. To implement this testing scheme, the team would need to tinker with the already existing stimulus module so that it writes to thousands of addresses instead of 32.

5.3.2. SRAM tracking

Functionality checkers may also be implemented without having the data that is written be dependent of the address it is written to or the system which writes this data. This testing scheme would provide more test coverage but would also be much more difficult to implement. An SRAM memory would be connected in parallel to DRAM, and as data gets written to the DRAM a carbon copy of it is stored in SRAM. The SRAM memory already exists on the FPGA so it would not be too difficult to implement. The greatest difficulty comes from figuring out how to compress the data that is written to DRAM so that it may fit in the much smaller SRAM. One way to do this is by limiting the address range of the DRAM which the arbiter writes to. Another way to this, would be by compressing the data packets which are sent to DRAM, so that they may fit in SRAM.

6. Conclusion

The development of an arbiter provides an in depth look into the logic design process. It forced us to decipher many pages of documentation and decide which methods of implementation were the best. Debugging in parallel with the development of the design meant iteratively viewing signal waveforms in iSim. At first debugging only in ChipScope, we soon learned it was the least effective way of doing things. ChipScope became a second measure of validation when iSim did not solve our problems or when a final measure of accuracy was required. Designing the arbiter was challenging because the design specifications constantly changed as we became more familiar with the technology, and tools that were available to us. Specifications were affected when we found better ways to do thing, when we noticed flaws in the design, or when the design simply wasn't working. Overall, learning how to communicate with the memory controller and designing an arbiter fostered knowledge about memory, debugging, performance, and many logic design concepts.

References

[1] Elpida Memory, Inc., "New Features of DDR3 SDRAM," March 2009. [Online]. Available: http://www.elpida.com/pdfs/E1503E10.pdf. [Accessed 15 December 2011].

[2] Hewlett-Packard Development Company, LP., "Memory technology evolution: an overview of system memory technologies," December 2010. [Online]. Available: http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00256987/c00256987.pdf. [Accessed 12 December 2011].

[3] "Virtex-6 FPGA Memory Interface Solutions User Guide," 1 March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf. [Accessed 20 September 2011].

[4] B. Matas and C. de Suberbasaux, "DRAM Technology," 1997. [Online]. Available: http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC07.PDF. [Accessed 5 January 2012].

[5] B. Matas and C. de Suberbasaux, "SRAM Technology," 1997. [Online]. Available: http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC08.PDF. [Accessed 10 January 2012].

[6] M. Barr, "Embedded Systems Memory Types," Netrino, May 2001. [Online]. Available: http://www.netrino.com/Embedded-Systems/How-To/Memory-Types-RAM-ROM-Flash. [Accessed 11 February 2012].

[7] V. Cuppu, B. Davis, B. Jacob and T. Mudge, "High-Performance DRAMs in Workstation Environments," *IEEE Transactions on Computers,* vol. 50, no. 11, pp. 1133-1153, 2001.

[8] J. F. Wakerly, "Memory, CPLDs, and FPGAs," in *Digital Design Principles and Practices*, Upper Saddle River, Pearson Prentice Hall, 2006, pp. 822-840.

[9] J. L. Hennessy and D. A. Patterson, in *Computer Architecture A Quantitative Approach*, Waltham, Elsevier, 2012, pp. 97-101.

[10] J. H. Davies, in *MSP430 Microcontroller Basics*, Burlington, Elsevier, 2008, p. 12.

[11] P. Singer, "Dynamic random access memory (DRAM)," *Semiconductor International,* vol. 26, no. 2, p. 84, 2003.

[12] Xilinx, "ChipScope Pro Software and Cores," 6 July 2011. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/chipscope_pro_sw_cores_ug029.pdf. [Accessed 1 March 2012].

[13] Xilinx, "ML605 Hardware User Guide," 18 July 2011. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf. [Accessed 3 March 2012].

[14] Xilinx, "ISE Help," 2008. [Online]. Available: http://www.xilinx.com/itp/xilinx10/isehelp/isehelp_start.htm. [Accessed 28 February 2012].

[15] Xilinx, "Virtex-6 Family Overview," 9 January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. [Accessed 1 March 2012].

[16] Core Technologies, "FPGA Logic Cells Comparison," Core Technologies, 2009. [Online]. Available: http://www.1-core.com/library/digital/fpga-logic-cells/. [Accessed 6 March 2012].

[17] Xilinx, "PlanAhead User Guide," 18 January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/PlanAhead_UserGuide.pdf. [Accessed 7 March 2012].

[18] Xilinx, "AXI Reference Guide," 18 January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_a

xi_reference_guide.pdf . [Accessed 10 October 2011].

Appendix A: MIG Core Architecture

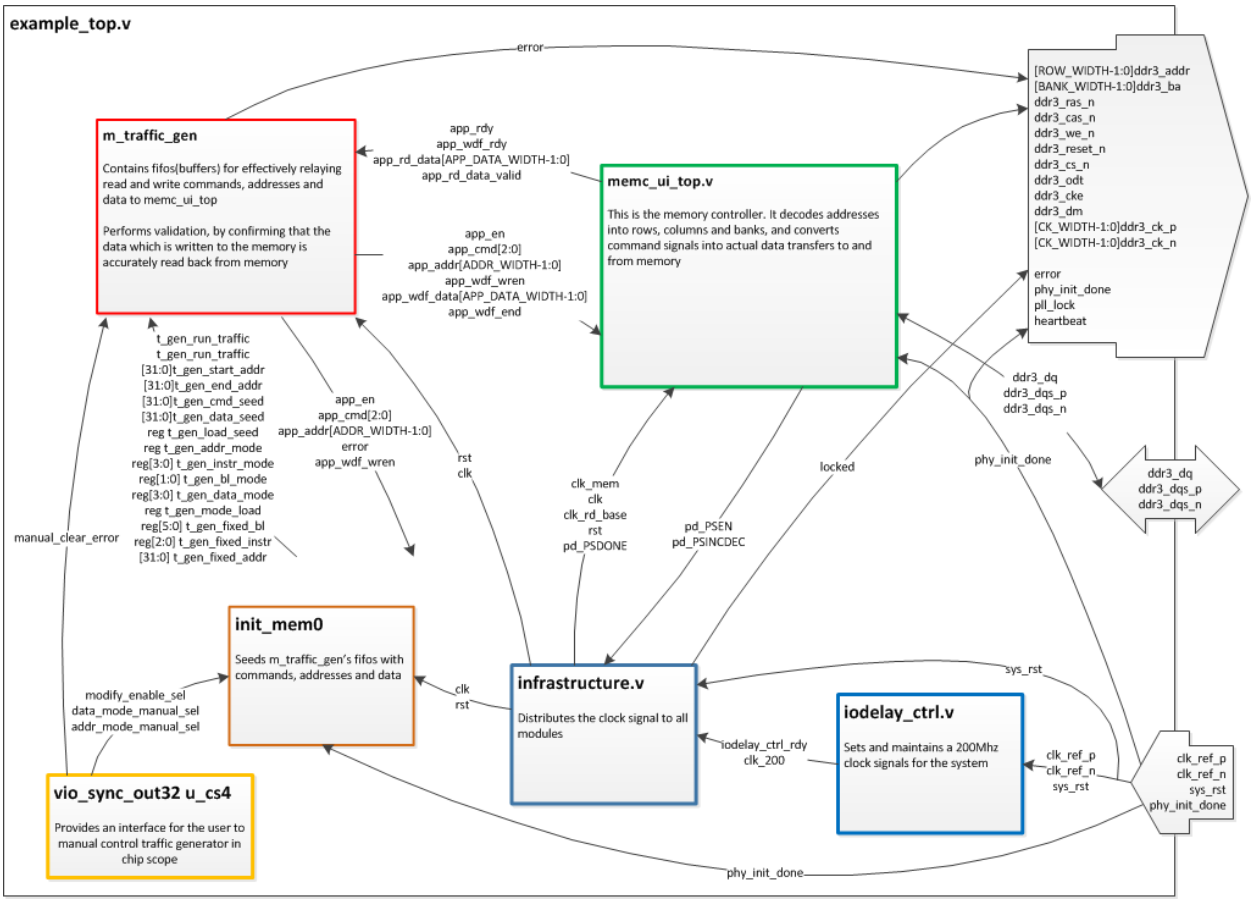

Figure 6.1 General Block Diagram of Memory Controller

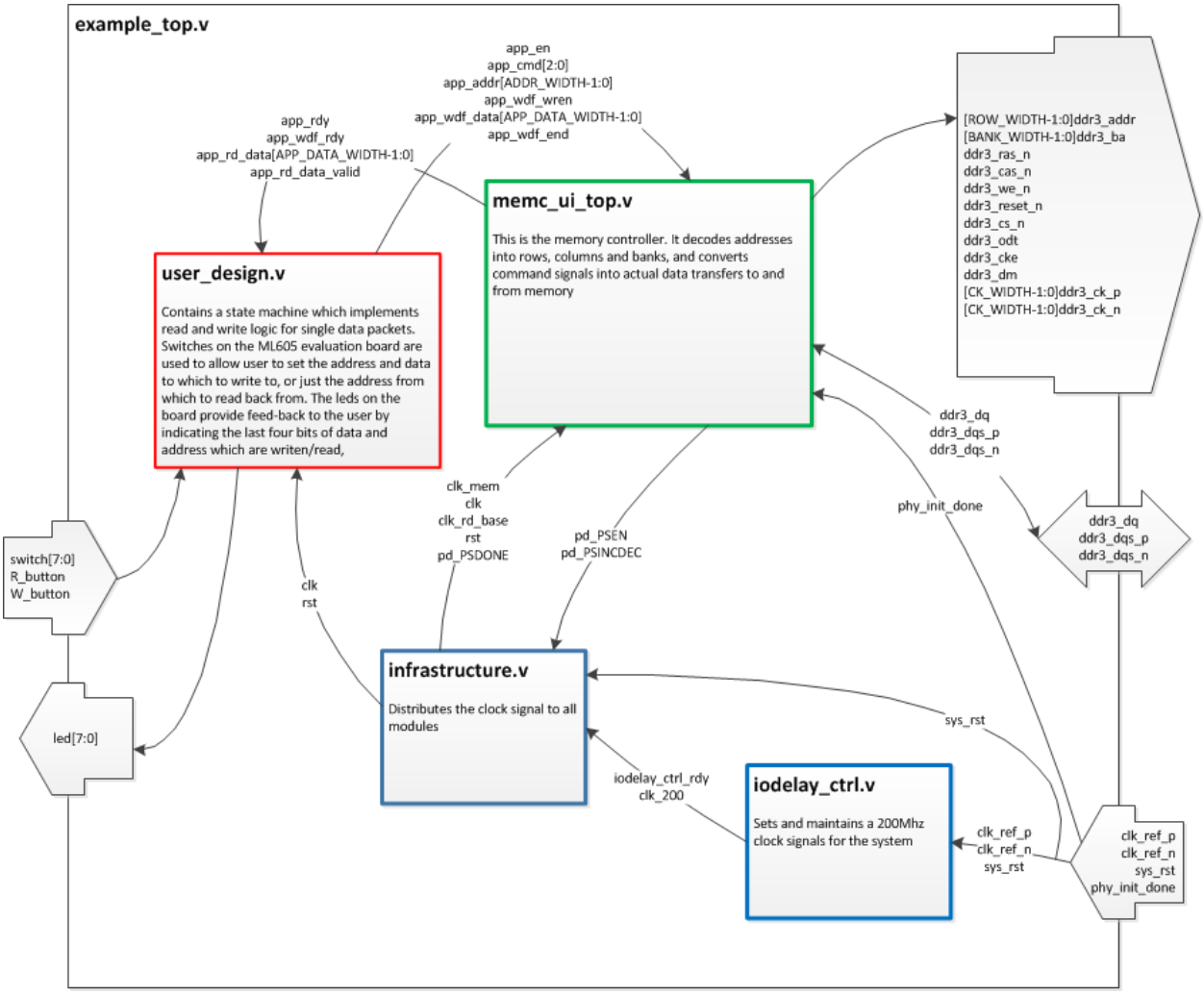

Figure 6.2 Arbiter (user_design) Module Connected to Memory Controller Blocks

Appendix B: Code

Example Top

```
`timescale 1ps/1ps

module example_top #
  (
   parameter REFCLK_FREQ             = 200,
                                       // # = 200 when design frequency <=
533 MHz,
                                       //   = 300 when design frequency > 533
MHz.
   parameter IODELAY_GRP             = "IODELAY_MIG",
                                       // It is associated to a set of
IODELAYs with
                                       // an IDELAYCTRL that have same
IODELAY CONTROLLER
                                       // clock frequency.
   parameter MMCM_ADV_BANDWIDTH      = "OPTIMIZED",
                                       // MMCM programming algorithm
   parameter CLKFBOUT_MULT_F         = 6,
                                       // write PLL VCO multiplier.
   parameter DIVCLK_DIVIDE           = 1,
                     // ML605 200MHz input clock (VCO = 1200MHz)use "2" for
          400MHz SMA,
                                       // write PLL VCO divisor.
   parameter CLKOUT_DIVIDE           = 3,  //400MHz clock
                                       // VCO output divisor for fast
(memory) clocks.
   parameter nCK_PER_CLK             = 2,
                                       // # of memory CKs per fabric clock.
                                       // # = 2, 1.
   parameter tCK                     = 2500,
                                       // memory tCK paramter.
                                       // # = Clock Period.
   parameter DEBUG_PORT              = "ON",
                                       // # = "ON" Enable debug
signals/controls.
                                       //   = "OFF" Disable debug
signals/controls.
   parameter SIM_BYPASS_INIT_CAL     = "OFF",
                                       // # = "OFF" -  Complete memory init &
                                       //              calibration sequence
                                       // # = "SKIP" - Skip memory init &
                                       //              calibration sequence
                                       // # = "FAST" - Skip memory init & use
                                       //              abbreviated calib
sequence
```

```
parameter nCS_PER_RAND           = 1,
                                   // # of unique CS outputs per Rank for
                                   // phy.
parameter DQS_CNT_WIDTH          = 3,
                                   // # = ceil(log2(DQS_WIDTH)).
parameter RANK_WIDTH             = 1,
                                   // # = ceil(log2(RANKS)).
parameter BANK_WIDTH             = 3,
                                   // # of memory Bank Address bits.
parameter CK_WIDTH               = 1,
                                   // # of CK/CK# outputs to memory.
parameter CKE_WIDTH              = 1,
                                   // # of CKE outputs to memory.
parameter COL_WIDTH              = 10,
                                   // # of memory Column Address bits.
parameter CS_WIDTH               = 1,
                                   // # of unique CS outputs to memory.
parameter DM_WIDTH               = 8,
                                   // # of Data Mask bits.
parameter DQ_WIDTH               = 64,
                                   // # of Data (DQ) bits.
parameter DQS_WIDTH              = 8,
                                   // # of DQS/DQS# bits.
parameter ROW_WIDTH              = 13,
                                   // # of memory Row Address bits.
parameter BURST_MODE             = "8",
                                   // Burst Length (Mode Register 0).
                                   // # = "8", "4", "OTF".
parameter BM_CNT_WIDTH           = 2,
                                   // # = ceil(log2(nBANK_MACHS)).
parameter ADDR_CMD_MODE          = "1T" ,
                                   // # = "2T", "1T".
parameter ORDERING               = "STRICT",
                                   // # = "NORM", "STRICT".
parameter WRLVL                  = "ON",
                                   // # = "ON" - DDR3 SDRAM
                                   //   = "OFF" - DDR2 SDRAM.
parameter PHASE_DETECT           = "ON",
                                   // # = "ON", "OFF".
parameter RTT_NOM                = "60",
                                   // RTT_NOM (ODT) (Mode Register 1).
                                   // # = "DISABLED" - RTT_NOM disabled,
                                   //   = "120" - RZQ/2,
                                   //   = "60"  - RZQ/4,
                                   //   = "40"  - RZQ/6.
parameter RTT_WR                 = "OFF",
                                   // RTT_WR (ODT) (Mode Register 2).
                                   // # = "OFF" - Dynamic ODT off,
                                   //   = "120" - RZQ/2,
                                   //   = "60"  - RZQ/4,
```

```
   parameter OUTPUT_DRV             = "HIGH",
                                      // Output Driver Impedance Control
(Mode Register 1).
                                      // # = "HIGH" - RZQ/7,
                                      //   = "LOW" - RZQ/6.
   parameter REG_CTRL               = "OFF",
                                      // # = "ON" - RDIMMs,
                                      //   = "OFF" - Components, SODIMMs,
UDIMMs.
   parameter nDQS_COL0              = 3,
                                      // Number of DQS groups in I/O column
#1.
   parameter nDQS_COL1              = 5,
                                      // Number of DQS groups in I/O column
#2.
   parameter nDQS_COL2              = 0,
                                      // Number of DQS groups in I/O column
#3.
   parameter nDQS_COL3              = 0,
                                      // Number of DQS groups in I/O column
#4.
   parameter DQS_LOC_COL0           = 24'h020100,
                                      // DQS groups in column #1.
   parameter DQS_LOC_COL1           = 40'h0706050403,
                                      // DQS groups in column #2.
   parameter DQS_LOC_COL2           = 0,
                                      // DQS groups in column #3.
   parameter DQS_LOC_COL3           = 0,
                                      // DQS groups in column #4.
   parameter tPRDI                  = 1_000_000,
                                      // memory tPRDI paramter.
   parameter tREFI                  = 7800000,
                                      // memory tREFI paramter.
   parameter tZQI                   = 128_000_000,
                                      // memory tZQI paramter.
   parameter ADDR_WIDTH             = 27,
                                      // # = RANK_WIDTH + BANK_WIDTH
                                      //     + ROW_WIDTH + COL_WIDTH;
   parameter ECC                    = "OFF",
   parameter ECC_TEST               = "OFF",
   parameter TCQ                    = 100,
   // Traffic Gen related parameters
//   parameter EYE_TEST               = "FALSE",
                                      // set EYE_TEST = "TRUE" to probe
memory
                                      // signals. Traffic Generator will
only
                                      // write to one single location and no
                                      // read transactions will be
generated.
```

```
//   parameter DATA_PATTERN            = "DGEN_ALL",
                                        // "DGEN_HAMMER", "DGEN_WALKING1",
                                        // "DGEN_WALKING0","DGEN_ADDR","
                                        //
"DGEN_NEIGHBOR","DGEN_PRBS","DGEN_ALL"
//   parameter CMD_PATTERN             = "CGEN_ALL",
                                        //
"CGEN_PRBS","CGEN_FIXED","CGEN_BRAM",
                                        // "CGEN_SEQUENTIAL", "CGEN_ALL"

//   parameter BEGIN_ADDRESS           = 32'h00000000,
//   parameter PRBS_SADDR_MASK_POS     = 32'h00000000,
//   parameter END_ADDRESS             = 32'h00ffffff,
//   parameter PRBS_EADDR_MASK_POS     = 32'hff000000,
//   parameter SEL_VICTIM_LINE         = 11,
   parameter RST_ACT_LOW             = 0,              // ML605 reset active
high
                                        // =1 for active low reset,
                                        // =0 for active high.
   parameter INPUT_CLK_TYPE          = "DIFFERENTIAL"
                                        // input clock type DIFFERENTIAL or
SINGLE_ENDED
                     //   parameter STARVE_LIMIT           = 2
                                        // # = 2,3,4.
   )
  (

//  input                               sys_clk_p,
//differential system clocks
//  input                               sys_clk_n,
  input                              clk_ref_p,     //differential iodelayctrl
clk
  input                              clk_ref_n,
  input                                sys_rst,   // System reset
  output [CK_WIDTH-1:0]                ddr3_ck_p,
  output [CK_WIDTH-1:0]                ddr3_ck_n,

   output [ROW_WIDTH-1:0]               ddr3_addr,
   output [BANK_WIDTH-1:0]              ddr3_ba,
   output                               ddr3_ras_n,
   output                               ddr3_cas_n,

   output                               ddr3_we_n,
   output                               ddr3_reset_n,
   output [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_cs_n,
   output [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_odt,
   output [CKE_WIDTH-1:0]               ddr3_cke,
   output [DM_WIDTH-1:0]                ddr3_dm,
      inout  [DQ_WIDTH-1:0]               ddr3_dq,
   inout  [DQS_WIDTH-1:0]               ddr3_dqs_p,
```

```
  inout  [DQS_WIDTH-1:0]                ddr3_dqs_n,
//  inout                                 sda,      // ML605
//  output                                scl,      // ML605
//  output                                error,
  //output                              pll_lock,   // ML605 GPIO LED
  //output                              heartbeat,  // ML605 GPIO LED

     //input [26:0]
     sys_address,
     //inout     [511:0]
     sys_data,

     //input[7:0]      switch,
     //input      n_button,
     //input           s_button,
     //input           w_button,
     //input           e_button,
     //input           c_button,
     output            phy_init_done,

     input sys1_fill_wr_exe2,
     input sys1_fill_rd_exe2,

     input sys2_fill_wr_exe2,
     input sys2_fill_rd_exe2,

     input trigger_activate,

     input [7:0] switch

  );

     //wire      n_button;
     //wire      s_button;
     //wire      w_button;
     //wire      e_button;
     //wire      c_button;
     wire [7:0]  switch;

     wire  trigger_activate;

     //wire      sys1_fill_rd_cmd;
     wire  sys1_fill_rd_exe;
     //wire      sys1_fill_wr_cmd;
     wire  sys1_fill_wr_exe;

     wire  sys2_fill_rd_exe;
     wire  sys2_fill_wr_exe;

     wire [3:0]  c_state;
```

```
 function integer STR_TO_INT;
    input [7:0] in;
    begin
      if(in == "8")
        STR_TO_INT = 8;
      else if(in == "4")
        STR_TO_INT = 4;
      else
        STR_TO_INT = 0;
    end
  endfunction

  localparam SYSCLK_PERIOD          = tCK * nCK_PER_CLK;

  localparam DATA_WIDTH          = 64;
  localparam PAYLOAD_WIDTH       = (ECC_TEST == "OFF") ? DATA_WIDTH :
DQ_WIDTH;
  //localparam BURST_LENGTH        = STR_TO_INT(BURST_MODE);
  localparam APP_DATA_WIDTH      = PAYLOAD_WIDTH * 4;
  localparam APP_MASK_WIDTH      = APP_DATA_WIDTH / 8;

  wire                                clk_ref;
  wire                                sys_clk;
  wire                                mmcm_clk;
  wire                                iodelay_ctrl_rdy;
  //wire                                phy_init_done;
  //  (* KEEP = "TRUE" *) wire          sda_i;
  //  (* KEEP = "TRUE" *) wire          scl_i;
  wire                                rst;
  wire                                clk;
  wire                                clk_mem;
  wire                                clk_rd_base;
  wire                                pd_PSDONE;
  wire                                pd_PSEN;
  wire                                pd_PSINCDEC;
  wire  [(BM_CNT_WIDTH)-1:0]          bank_mach_next;
  wire                                ddr3_parity;
  wire                                app_hi_pri;
  wire [APP_MASK_WIDTH-1:0]           app_wdf_mask;
  wire [3:0]                          app_ecc_multiple_err_i;
//  wire [47:0]                         traffic_wr_data_counts;
//  wire [47:0]                         traffic_rd_data_counts;
  wire [ADDR_WIDTH-1:0]               app_addr;
  wire [2:0]                          app_cmd;
  wire                                app_en;
  wire                                app_sz;
  //wire                                app_rdy;
  wire [APP_DATA_WIDTH-1:0]           app_rd_data;
  wire                                app_rd_data_valid;
```

```
  wire [APP_DATA_WIDTH-1:0]          app_wdf_data;
  wire                               app_wdf_end;
  wire                               app_wdf_rdy;
  wire                               app_wdf_wren;


  wire [35:0]
      ila_CONTROL;
  wire [511:0]
      sys_data_write;
  wire [26:0]
      sys_address;
  wire [511:0]
      sys_data_read;
  wire
      sys_rdy;
  wire
      sys_wdf_rdy;

//  wire                               modify_enable_sel;
//  wire [2:0]                         data_mode_manual_sel;
//  wire [2:0]                         addr_mode_manual_sel;
//  wire                               t_gen_run_traffic;
//  wire  [3:0]                        t_gen_instr_mode;
//  wire [31:0]                        t_gen_start_addr;
//  wire [31:0]                        t_gen_end_addr;
//  wire [31:0]                        t_gen_cmd_seed;
//  wire [31:0]                        t_gen_data_seed;
//  wire                               t_gen_load_seed;
//  wire [2:0]                         t_gen_addr_mode;
//  wire [1:0]                         t_gen_bl_mode;
//  wire [3:0]                         t_gen_data_mode;
//  wire                               t_gen_mode_load;
//  wire [5:0]                         t_gen_fixed_bl;
//  wire [2:0]                         t_gen_fixed_instr;
//  wire [31:0]                        t_gen_fixed_addr;
//  wire                               manual_clear_error;
//  wire [6:0]                         tg_wr_fifo_counts;
//  wire [6:0]                         tg_rd_fifo_counts;


  wire [5*DQS_WIDTH-1:0]             dbg_cpt_first_edge_cnt;
  wire [5*DQS_WIDTH-1:0]             dbg_cpt_second_edge_cnt;
  wire [5*DQS_WIDTH-1:0]             dbg_cpt_tap_cnt;
  wire                               dbg_dec_cpt;
  wire                               dbg_dec_rd_dqs;
  wire                               dbg_dec_rd_fps;
  wire [5*DQS_WIDTH-1:0]             dbg_dq_tap_cnt;
  wire [5*DQS_WIDTH-1:0]             dbg_dqs_tap_cnt;
  wire                               dbg_inc_cpt;
```

```
wire [DQS_CNT_WIDTH-1:0]          dbg_inc_dec_sel;
wire                              dbg_inc_rd_dqs;
wire                              dbg_inc_rd_fps;
wire                              dbg_ocb_mon_off;
wire                              dbg_pd_off;
wire                              dbg_pd_maintain_off;
wire                              dbg_pd_maintain_0_only;
wire [4:0]                        dbg_rd_active_dly;
wire [3*DQS_WIDTH-1:0]            dbg_rd_bitslip_cnt;
wire [2*DQS_WIDTH-1:0]            dbg_rd_clkdly_cnt;
wire [4*DQ_WIDTH-1:0]             dbg_rddata;
wire [1:0]                        dbg_rdlvl_done;
wire [1:0]                        dbg_rdlvl_err;
wire [1:0]                        dbg_rdlvl_start;
wire [DQS_WIDTH-1:0]              dbg_wl_dqs_inverted;
wire [5*DQS_WIDTH-1:0]            dbg_wl_odelay_dq_tap_cnt;
wire [5*DQS_WIDTH-1:0]            dbg_wl_odelay_dqs_tap_cnt;
wire [2*DQS_WIDTH-1:0]            dbg_wr_calib_clk_delay;
wire [5*DQS_WIDTH-1:0]            dbg_wr_dq_tap_set;
wire [5*DQS_WIDTH-1:0]            dbg_wr_dqs_tap_set;
wire                              dbg_wr_tap_set_en;
wire                              dbg_idel_up_all;
wire                              dbg_idel_down_all;
wire                              dbg_idel_up_cpt;
wire                              dbg_idel_down_cpt;
wire                              dbg_idel_up_rsync;
wire                              dbg_idel_down_rsync;
wire                              dbg_sel_all_idel_cpt;
wire                              dbg_sel_all_idel_rsync;
wire                              dbg_pd_inc_cpt;
wire                              dbg_pd_dec_cpt;
wire                              dbg_pd_inc_dqs;
wire                              dbg_pd_dec_dqs;
wire                              dbg_pd_disab_hyst;
wire                              dbg_pd_disab_hyst_0;
wire                              dbg_wrlvl_done;
wire                              dbg_wrlvl_err;
wire                              dbg_wrlvl_start;
wire [4:0]                        dbg_tap_cnt_during_wrlvl;
wire [19:0]                       dbg_rsync_tap_cnt;
wire [255:0]                      dbg_phy_pd;
wire [255:0]                      dbg_phy_read;
wire [255:0]                      dbg_phy_rdlvl;
wire [255:0]                      dbg_phy_top;
wire [3:0]                        dbg_pd_msb_sel;
wire [DQS_WIDTH-1:0]              dbg_rd_data_edge_detect;
wire [DQS_CNT_WIDTH-1:0]          dbg_sel_idel_cpt;
wire [DQS_CNT_WIDTH-1:0]          dbg_sel_idel_rsync;
wire [DQS_CNT_WIDTH-1:0]          dbg_pd_byte_sel;
```

```
  wire                                ddr3_cs0_clk;
  wire [35:0]                         ddr3_cs0_control;
  wire [383:0]                        ddr3_cs0_data;
  wire [7:0]                          ddr3_cs0_trig;
  wire [255:0]                        ddr3_cs1_async_in;
  wire [35:0]                         ddr3_cs1_control;
  wire [255:0]                        ddr3_cs2_async_in;
  wire [35:0]                         ddr3_cs2_control;
  wire [255:0]                        ddr3_cs3_async_in;
  wire [35:0]                         ddr3_cs3_control;
  wire                                ddr3_cs4_clk;
  wire [35:0]                         ddr3_cs4_control;
  wire [31:0]                         ddr3_cs4_sync_out;


//***************************************************************************

 // assign error = 1'b0;
  assign app_hi_pri = 1'b0;
  assign app_wdf_mask = {APP_MASK_WIDTH{1'b0}};

// ML605 comment out the following signals to enable traffic generator
control from VIO console:
// assign manual_clear_error     = 1'b0;
// assign modify_enable_sel      = 1'b1;
// assign data_mode_manual_sel   = 3'b010; // ADDR_DATA
// assign addr_mode_manual_sel   = 3'b011; //SEQUENTIAL_ADDR

  wire  locked; // ML605
  //assign pll_lock = locked;  // ML605

/*  MUXCY scl_inst
    (
     .O  (scl),
     .CI (scl_i),
     .DI (1'b0),
     .S  (1'b1)
     );

  MUXCY sda_inst
    (
     .O  (sda),
     .CI (sda_i),
     .DI (1'b0),
     .S  (1'b1)
     );
*/
  assign clk_ref = 1'b0;
//  assign sys_clk = 1'b0;
// ML605 200MHz clock sourced from BUFG within "idelay_ctrl" module.
```

```
  wire clk_200;


  iodelay_ctrl #
    (
     .TCQ             (TCQ),
     .IODELAY_GRP     (IODELAY_GRP),
     .INPUT_CLK_TYPE (INPUT_CLK_TYPE),
     .RST_ACT_LOW     (RST_ACT_LOW)
     )
    u_iodelay_ctrl
      (
       .clk_ref_p        (clk_ref_p),  // ML605 200MHz EPSON oscillator
       .clk_ref_n        (clk_ref_n),
       .clk_ref          (clk_ref),
       .sys_rst          (sys_rst),
       .clk_200          (clk_200),    // ML605 200MHz clock from BUFG to
MMCM CLKIN1
       .iodelay_ctrl_rdy (iodelay_ctrl_rdy)
       );
/* ML605 comment out "clk_ibuf" module. MIG default requires 2 inputs clocks.

  clk_ibuf #
    (
     .INPUT_CLK_TYPE (INPUT_CLK_TYPE)
     )
    u_clk_ibuf
      (
       .sys_clk_p         (sys_clk_p),
       .sys_clk_n         (sys_clk_n),
       .sys_clk           (sys_clk),
       .mmcm_clk          (mmcm_clk)
       );
*/


  infrastructure #
    (
     .TCQ                (TCQ),
     .CLK_PERIOD         (SYSCLK_PERIOD),
     .nCK_PER_CLK        (nCK_PER_CLK),
     .MMCM_ADV_BANDWIDTH (MMCM_ADV_BANDWIDTH),
     .CLKFBOUT_MULT_F    (CLKFBOUT_MULT_F),
     .DIVCLK_DIVIDE      (DIVCLK_DIVIDE),
     .CLKOUT_DIVIDE      (CLKOUT_DIVIDE),
     .RST_ACT_LOW        (RST_ACT_LOW)
     )
    u_infrastructure
      (
       .clk_mem          (clk_mem),
```

```
       .clk              (clk),
       .clk_rd_base      (clk_rd_base),
       .pll_lock         (locked),     // ML605 GPIO LED output port
       .rstdiv0          (rst),
       .mmcm_clk         (clk_200),    // ML605 single input clock 200MHz
from "iodelay_ctrl"
       .sys_rst          (sys_rst),
       .iodelay_ctrl_rdy (iodelay_ctrl_rdy),
       .PSDONE           (pd_PSDONE),
       .PSEN             (pd_PSEN),
       .PSINCDEC         (pd_PSINCDEC)
       );


  memc_ui_top #
  (
   .ADDR_CMD_MODE        (ADDR_CMD_MODE),
   .BANK_WIDTH           (BANK_WIDTH),
   .CK_WIDTH             (CK_WIDTH),
   .CKE_WIDTH            (CKE_WIDTH),
   .nCK_PER_CLK          (nCK_PER_CLK),
   .COL_WIDTH            (COL_WIDTH),
   .CS_WIDTH             (CS_WIDTH),
   .DM_WIDTH             (DM_WIDTH),
   .nCS_PER_RANK         (nCS_PER_RANK),
   .DEBUG_PORT           (DEBUG_PORT),
   .IODELAY_GRP          (IODELAY_GRP),
   .DQ_WIDTH             (DQ_WIDTH),
   .DQS_WIDTH            (DQS_WIDTH),
   .DQS_CNT_WIDTH        (DQS_CNT_WIDTH),
   .ORDERING             (ORDERING),
   .OUTPUT_DRV           (OUTPUT_DRV),
   .PHASE_DETECT         (PHASE_DETECT),
   .RANK_WIDTH           (RANK_WIDTH),
   .REFCLK_FREQ          (REFCLK_FREQ),
   .REG_CTRL             (REG_CTRL),
   .ROW_WIDTH            (ROW_WIDTH),
   .RTT_NOM              (RTT_NOM),
   .RTT_WR               (RTT_WR),
   .SIM_BYPASS_INIT_CAL  (SIM_BYPASS_INIT_CAL),
   .WRLVL                (WRLVL),
   .nDQS_COL0            (nDQS_COL0),
   .nDQS_COL1            (nDQS_COL1),
   .nDQS_COL2            (nDQS_COL2),
   .nDQS_COL3            (nDQS_COL3),
   .DQS_LOC_COL0         (DQS_LOC_COL0),
   .DQS_LOC_COL1         (DQS_LOC_COL1),
   .DQS_LOC_COL2         (DQS_LOC_COL2),
   .DQS_LOC_COL3         (DQS_LOC_COL3),
   .tPRDI                (tPRDI),
```

```
 .tREFI                (tREFI),
 .tZQI                 (tZQI),
 .BURST_MODE           (BURST_MODE),
 .BM_CNT_WIDTH         (BM_CNT_WIDTH),
 .tCK                  (tCK),
 .ADDR_WIDTH           (ADDR_WIDTH),
 .TCQ                  (TCQ),
 .ECC                  (ECC),
 .ECC_TEST             (ECC_TEST),
 .PAYLOAD_WIDTH        (PAYLOAD_WIDTH),
 .APP_DATA_WIDTH       (APP_DATA_WIDTH),
 .APP_MASK_WIDTH       (APP_MASK_WIDTH)
 )
u_memc_ui_top
(
 .clk                              (clk),
 .clk_mem                          (clk_mem),
 .clk_rd_base                      (clk_rd_base),
 .rst                              (rst),
 .ddr_addr                         (ddr3_addr),
 .ddr_ba                           (ddr3_ba),
 .ddr_cas_n                        (ddr3_cas_n),
 .ddr_ck_n                         (ddr3_ck_n),
 .ddr_ck                           (ddr3_ck_p),
 .ddr_cke                          (ddr3_cke),
 .ddr_cs_n                         (ddr3_cs_n),
 .ddr_dm                           (ddr3_dm),
 .ddr_odt                          (ddr3_odt),
 .ddr_ras_n                        (ddr3_ras_n),
 .ddr_reset_n                      (ddr3_reset_n),
 .ddr_parity                       (ddr3_parity),
 .ddr_we_n                         (ddr3_we_n),
 .ddr_dq                           (ddr3_dq),
 .ddr_dqs_n                        (ddr3_dqs_n),
 .ddr_dqs                          (ddr3_dqs_p),
 .pd_PSEN                          (pd_PSEN),
 .pd_PSINCDEC                      (pd_PSINCDEC),
 .pd_PSDONE                        (pd_PSDONE),
 .phy_init_done                    (phy_init_done),
 .bank_mach_next                   (bank_mach_next),
 .app_ecc_multiple_err             (app_ecc_multiple_err_i),
 .app_rd_data                      (app_rd_data),
 .app_rd_data_end                  (app_rd_data_end),
 .app_rd_data_valid                (app_rd_data_valid),
 .app_rdy                          (app_rdy),
 .app_wdf_rdy                      (app_wdf_rdy),
 .app_addr                         (app_addr),
 .app_cmd                          (app_cmd),
 .app_en                           (app_en),
 .app_hi_pri                       (app_hi_pri),
```

```
 .app_sz                          (1'b1),
 .app_wdf_data                    (app_wdf_data),
 .app_wdf_end                     (app_wdf_end),
 .app_wdf_mask                    (app_wdf_mask),
 .app_wdf_wren                    (app_wdf_wren),
 .app_correct_en                  (1'b1),
 .dbg_wr_dqs_tap_set              (dbg_wr_dqs_tap_set),
 .dbg_wr_dq_tap_set               (dbg_wr_dq_tap_set),
 .dbg_wr_tap_set_en               (dbg_wr_tap_set_en),
 .dbg_wrlvl_start                 (dbg_wrlvl_start),
 .dbg_wrlvl_done                  (dbg_wrlvl_done),
 .dbg_wrlvl_err                   (dbg_wrlvl_err),
 .dbg_wl_dqs_inverted             (dbg_wl_dqs_inverted),
 .dbg_wr_calib_clk_delay          (dbg_wr_calib_clk_delay),
 .dbg_wl_odelay_dqs_tap_cnt       (dbg_wl_odelay_dqs_tap_cnt),
 .dbg_wl_odelay_dq_tap_cnt        (dbg_wl_odelay_dq_tap_cnt),
 .dbg_rdlvl_start                 (dbg_rdlvl_start),
 .dbg_rdlvl_done                  (dbg_rdlvl_done),
 .dbg_rdlvl_err                   (dbg_rdlvl_err),
 .dbg_cpt_tap_cnt                 (dbg_cpt_tap_cnt),
 .dbg_cpt_first_edge_cnt          (dbg_cpt_first_edge_cnt),
 .dbg_cpt_second_edge_cnt         (dbg_cpt_second_edge_cnt),
 .dbg_rd_bitslip_cnt              (dbg_rd_bitslip_cnt),
 .dbg_rd_clkdly_cnt               (dbg_rd_clkdly_cnt),
 .dbg_rd_active_dly               (dbg_rd_active_dly),
 .dbg_pd_off                      (dbg_pd_off),
 .dbg_pd_maintain_off             (dbg_pd_maintain_off),
 .dbg_pd_maintain_0_only          (dbg_pd_maintain_0_only),
 .dbg_inc_cpt                     (dbg_inc_cpt),
 .dbg_dec_cpt                     (dbg_dec_cpt),
 .dbg_inc_rd_dqs                  (dbg_inc_rd_dqs),
 .dbg_dec_rd_dqs                  (dbg_dec_rd_dqs),
 .dbg_inc_dec_sel                 (dbg_inc_dec_sel),
 .dbg_inc_rd_fps                  (dbg_inc_rd_fps),
 .dbg_dec_rd_fps                  (dbg_dec_rd_fps),
 .dbg_dqs_tap_cnt                 (dbg_dqs_tap_cnt),
 .dbg_dq_tap_cnt                  (dbg_dq_tap_cnt),
 .dbg_rddata                      (dbg_rddata)
 );


// Traffic Gen Modules
/*init_mem_pattern_ctr #
  (
   .FAMILY        ("VIRTEX6"),
   .MEM_BURST_LEN (BURST_LENGTH),
   .BEGIN_ADDRESS (BEGIN_ADDRESS),
   .END_ADDRESS   (END_ADDRESS),
   .DWIDTH        (APP_DATA_WIDTH),
   .ADDR_WIDTH    (ADDR_WIDTH),
```

```
   .EYE_TEST      (EYE_TEST)
   )
  init_mem0
    (
     .clk_i                (clk),
     .rst_i                (rst),
     .mcb_cmd_en_i         (app_en),
     .mcb_cmd_instr_i      (app_cmd[2:0]),
     .mcb_cmd_addr_i       (app_addr),
     .mcb_cmd_bl_i         (6'b001000),
     .mcb_init_done_i      (phy_init_done),
     .cmp_error            (error),
     .run_traffic_o        (t_gen_run_traffic),
     .start_addr_o         (t_gen_start_addr),
     .end_addr_o           (t_gen_end_addr),
     .cmd_seed_o           (t_gen_cmd_seed),
     .data_seed_o          (t_gen_data_seed),
     .load_seed_o          (t_gen_load_seed),
     .addr_mode_o          (t_gen_addr_mode),
     .instr_mode_o         (t_gen_instr_mode),
     .bl_mode_o            (t_gen_bl_mode),
     .data_mode_o          (t_gen_data_mode),
     .mode_load_o          (t_gen_mode_load),
     .fixed_bl_o           (t_gen_fixed_bl),
     .fixed_instr_o        (t_gen_fixed_instr),
     .fixed_addr_o         (t_gen_fixed_addr),
     .mcb_wr_en_i          (app_wdf_wren),
     .vio_modify_enable    (modify_enable_sel),
     .vio_data_mode_value  (data_mode_manual_sel),
     .vio_addr_mode_value  (addr_mode_manual_sel),
     .vio_bl_mode_value    (2'b01),
     .vio_fixed_bl_value   (6'b000010)
     );

mcb_traffic_gen #
  (
   .FAMILY              ("VIRTEX6"),
   .MEM_BURST_LEN       (BURST_LENGTH),
   .PORT_MODE           ("BI_MODE"),
   .DATA_PATTERN        (DATA_PATTERN),
   .CMD_PATTERN         (CMD_PATTERN),
   .ADDR_WIDTH          (ADDR_WIDTH),
   .MEM_COL_WIDTH       (COL_WIDTH),
   .NUM_DQ_PINS         (PAYLOAD_WIDTH),
   .SEL_VICTIM_LINE     (SEL_VICTIM_LINE),
   .DWIDTH              (APP_DATA_WIDTH),
   .DQ_ERROR_WIDTH      (PAYLOAD_WIDTH/8),
   .PRBS_SADDR_MASK_POS (PRBS_SADDR_MASK_POS),
   .PRBS_EADDR_MASK_POS (PRBS_EADDR_MASK_POS),
   .PRBS_SADDR          (BEGIN_ADDRESS),
```

```
 .PRBS_EADDR          (END_ADDRESS),
 .EYE_TEST            (EYE_TEST)
 )
m_traffic_gen
  (
   .clk_i              (clk),
   .rst_i              (rst),
   .run_traffic_i      (t_gen_run_traffic),
   .manual_clear_error (manual_clear_error),
   .start_addr_i       (t_gen_start_addr),
   .end_addr_i         (t_gen_end_addr),
   .cmd_seed_i         (t_gen_cmd_seed),
   .data_seed_i        (t_gen_data_seed),
   .load_seed_i        (t_gen_load_seed),
   .addr_mode_i        (t_gen_addr_mode),
   .instr_mode_i       (t_gen_instr_mode),
   .bl_mode_i          (t_gen_bl_mode),
   .data_mode_i        (t_gen_data_mode),
   .mode_load_i        (t_gen_mode_load),
   .fixed_bl_i         (t_gen_fixed_bl),
   .fixed_instr_i      (t_gen_fixed_instr),
   .fixed_addr_i       (t_gen_fixed_addr),
   .bram_cmd_i         (39'b0),
   .bram_valid_i       (1'b0),
   .bram_rdy_o         (),
   .mcb_cmd_en_o       (app_en),
   .mcb_cmd_instr_o    (app_cmd[2:0]),
   .mcb_cmd_addr_o     (app_addr),
   .mcb_cmd_bl_o       (),
   .mcb_cmd_full_i     (~app_rdy),
   .mcb_wr_en_o        (app_wdf_wren),
   .mcb_wr_data_o      (app_wdf_data[APP_DATA_WIDTH-1:0]),
   .mcb_wr_full_i      (~app_wdf_rdy),
   .mcb_wr_data_end_o  (app_wdf_end),
   .mcb_wr_fifo_counts (tg_wr_fifo_counts),
   .mcb_wr_mask_o      (),
   .mcb_rd_en_o        (tg_rd_en),
   .mcb_rd_data_i      (app_rd_data[APP_DATA_WIDTH-1:0]),
   .mcb_rd_empty_i     (~app_rd_data_valid),
   .mcb_rd_fifo_counts (tg_rd_fifo_counts),
   .counts_rst         (rst),
   .wr_data_counts     (),
   .rd_data_counts     (),
   .cmp_data           (),
   .cmp_error          (),
   .cmp_data_valid     (),
   .error              (error),
   .error_status       (),
   .mem_rd_data        (),
   .fixed_data_i       ({APP_DATA_WIDTH{1'b0}}),
```

```
      .dq_error_bytelane_cmp(),
      .cumlative_dq_lane_error()
      );*/


// If debug port is not enabled, then make certain control input
// to Debug Port are disabled
generate
  if (DEBUG_PORT == "OFF") begin: gen_dbg_tie_off
    assign dbg_wr_dqs_tap_set     = 'b0;
    assign dbg_wr_dq_tap_set      = 'b0;
    assign dbg_wr_tap_set_en      = 1'b0;
    assign dbg_pd_off             = 1'b0;
    assign dbg_pd_maintain_off    = 1'b0;
    assign dbg_pd_maintain_0_only = 1'b0;
    assign dbg_ocb_mon_off        = 1'b0;
    assign dbg_inc_cpt            = 1'b0;
    assign dbg_dec_cpt            = 1'b0;
    assign dbg_inc_rd_dqs         = 1'b0;
    assign dbg_dec_rd_dqs         = 1'b0;
    assign dbg_inc_dec_sel        = 'b0;
    assign dbg_inc_rd_fps         = 1'b0;
    assign dbg_pd_msb_sel         = 'b0 ;
    assign dbg_sel_idel_cpt       = 'b0 ;
    assign dbg_sel_idel_rsync     = 'b0 ;
    assign dbg_pd_byte_sel        = 'b0 ;
    assign dbg_dec_rd_fps         = 1'b0;
  end
endgenerate

generate
  if (DEBUG_PORT == "ON") begin: gen_dbg_enable

    // Connect these to VIO if changing output (write)
    // IODELAY taps desired
    assign dbg_wr_dqs_tap_set     = 'b0;
    assign dbg_wr_dq_tap_set      = 'b0;
    assign dbg_wr_tap_set_en      = 1'b0;

    // Connect these to VIO if changing read base clock
    // phase required
    assign dbg_inc_rd_fps         = 1'b0;
    assign dbg_dec_rd_fps         = 1'b0;

    //*****************************************************
    // CS0 - ILA for monitoring PHY status, testbench error,
    //       and synchronized read data
    //*****************************************************
```

```
      // Assignments for ILA monitoring general PHY
      // status and synchronized read data
      assign ddr3_cs0_clk             = clk;
      assign ddr3_cs0_trig[1:0]       = dbg_rdlvl_done;
      assign ddr3_cs0_trig[3:2]       = dbg_rdlvl_err;
      assign ddr3_cs0_trig[4]         = phy_init_done;
      assign ddr3_cs0_trig[5]         = 1'b0;
      assign ddr3_cs0_trig[7:5]       = 'b0;   // ML605

      // Support for only up to 72-bits of data
      if (DQ_WIDTH <= 72) begin: gen_dq_le_72
        assign ddr3_cs0_data[4*DQ_WIDTH-1:0] = dbg_rddata;
      end else begin: gen_dq_gt_72
        assign ddr3_cs0_data[287:0] = dbg_rddata[287:0];
      end

      assign ddr3_cs0_data[289:288]   = dbg_rdlvl_done;
      assign ddr3_cs0_data[291:290]   = dbg_rdlvl_err;
      assign ddr3_cs0_data[292]       = phy_init_done;
      assign ddr3_cs0_data[293]       = 1'b0; // ML605 connect to ERROR from
TrafficGen
      //assign ddr3_cs0_data[294]       = app_rd_data_valid; // ML605 read
data valid
      //assign ddr3_cs0_data[295]       = pll_lock; // ML605 PLL_LOCK status
indicator
      //assign ddr3_cs0_data[383:296]   = 'b0;
            assign ddr3_cs0_data[383:294]   = 'b0;


      //*****************************************************
      // CS1 - Input VIO for monitoring PHY status and
      //       write leveling/calibration delays
      //*****************************************************

      // Support for only up to 18 DQS groups
      if (DQS_WIDTH <= 18) begin: gen_dqs_le_18_cs1
        assign ddr3_cs1_async_in[5*DQS_WIDTH-1:0]     =
dbg_wl_odelay_dq_tap_cnt;
        assign ddr3_cs1_async_in[5*DQS_WIDTH+89:90]   =
dbg_wl_odelay_dqs_tap_cnt;
        assign ddr3_cs1_async_in[DQS_WIDTH+179:180]   = dbg_wl_dqs_inverted;
        assign ddr3_cs1_async_in[2*DQS_WIDTH+197:198] =
dbg_wr_calib_clk_delay;
      end else begin: gen_dqs_gt_18_cs1
        assign ddr3_cs1_async_in[89:0]    = dbg_wl_odelay_dq_tap_cnt[89:0];
        assign ddr3_cs1_async_in[179:90]  = dbg_wl_odelay_dqs_tap_cnt[89:0];
        assign ddr3_cs1_async_in[197:180] = dbg_wl_dqs_inverted[17:0];
        assign ddr3_cs1_async_in[233:198] = dbg_wr_calib_clk_delay[35:0];
      end
```

```
      assign ddr3_cs1_async_in[235:234] = dbg_rdlvl_done[1:0];
      assign ddr3_cs1_async_in[237:236] = dbg_rdlvl_err[1:0];
      assign ddr3_cs1_async_in[238]     = phy_init_done;
      assign ddr3_cs1_async_in[239]     = 1'b0; // Pre-MIG 3.4: Used for
rst_pll_ck_fb
      assign ddr3_cs1_async_in[240]     = 1'b0; // ML605 ERROR from
TrafficGen
      assign ddr3_cs1_async_in[255:241] = 'b0;

      //*****************************************************
      // CS2 - Input VIO for monitoring Read Calibration
      //       results.
      //*****************************************************

      // Support for only up to 18 DQS groups
      if (DQS_WIDTH <= 18) begin: gen_dqs_le_18_cs2
        assign ddr3_cs2_async_in[5*DQS_WIDTH-1:0]     = dbg_cpt_tap_cnt;
        // Reserved for future monitoring of DQ tap counts from read leveling
        assign ddr3_cs2_async_in[5*DQS_WIDTH+89:90]   = 'b0;
        assign ddr3_cs2_async_in[3*DQS_WIDTH+179:180] = dbg_rd_bitslip_cnt;
      end else begin: gen_dqs_gt_18_cs2
        assign ddr3_cs2_async_in[89:0]    = dbg_cpt_tap_cnt[89:0];
        // Reserved for future monitoring of DQ tap counts from read leveling
        assign ddr3_cs2_async_in[179:90]  = 'b0;
        assign ddr3_cs2_async_in[233:180] = dbg_rd_bitslip_cnt[53:0];
      end

      assign ddr3_cs2_async_in[238:234] = dbg_rd_active_dly;
      assign ddr3_cs2_async_in[255:239] = 'b0;

      //*****************************************************
      // CS3 - Input VIO for monitoring more Read Calibration
      //       results.
      //*****************************************************

      // Support for only up to 18 DQS groups
      if (DQS_WIDTH <= 18) begin: gen_dqs_le_18_cs3
        assign ddr3_cs3_async_in[5*DQS_WIDTH-1:0]     =
dbg_cpt_first_edge_cnt;
        assign ddr3_cs3_async_in[5*DQS_WIDTH+89:90]   =
dbg_cpt_second_edge_cnt;
        assign ddr3_cs3_async_in[2*DQS_WIDTH+179:180] = dbg_rd_clkdly_cnt;
      end else begin: gen_dqs_gt_18_cs3
        assign ddr3_cs3_async_in[89:0]    = dbg_cpt_first_edge_cnt[89:0];
        assign ddr3_cs3_async_in[179:90]  = dbg_cpt_second_edge_cnt[89:0];
        assign ddr3_cs3_async_in[215:180] = dbg_rd_clkdly_cnt[35:0];
      end

      assign ddr3_cs3_async_in[255:216] = 'b0;
```

```
      //*****************************************************
      // CS4 - Output VIO for disabling OCB monitor, Read Phase
      //       Detector, and dynamically changing various
      //       IODELAY values used for adjust read data capture
      //       timing
      //*****************************************************

      assign ddr3_cs4_clk                = clk;
      assign dbg_pd_off             = ddr3_cs4_sync_out[0];
      assign dbg_pd_maintain_off    = ddr3_cs4_sync_out[1];
      assign dbg_pd_maintain_0_only = ddr3_cs4_sync_out[2];
      assign dbg_ocb_mon_off        = ddr3_cs4_sync_out[3];
      assign dbg_inc_cpt            = ddr3_cs4_sync_out[4];
      assign dbg_dec_cpt            = ddr3_cs4_sync_out[5];
      assign dbg_inc_rd_dqs         = ddr3_cs4_sync_out[6];
      assign dbg_dec_rd_dqs         = ddr3_cs4_sync_out[7];
      assign dbg_inc_dec_sel        = ddr3_cs4_sync_out[DQS_CNT_WIDTH+7:8];

// ML605 add assignments to control traffic generator function from VIO
console:

//      assign manual_clear_error     = ddr3_cs4_sync_out[24];     // ML605
debug
//      assign modify_enable_sel      = ddr3_cs4_sync_out[25];     // ML605
debug
//      assign addr_mode_manual_sel   = ddr3_cs4_sync_out[28:26];  // ML605
debug
//      assign data_mode_manual_sel   = ddr3_cs4_sync_out[31:29];  // ML605
debug
/*
      icon5 u_icon
        (
         .CONTROL0 (ddr3_cs0_control),
         .CONTROL1 (ddr3_cs1_control),
         .CONTROL2 (ddr3_cs2_control),
         .CONTROL3 (ddr3_cs3_control),
         .CONTROL4 (ddr3_cs4_control)
         );

      ila384_8 u_cs0
        (
         .CLK     (ddr3_cs0_clk),
         .DATA    (ddr3_cs0_data),
         .TRIG0   (ddr3_cs0_trig),
         .CONTROL (ddr3_cs0_control)
         );

      vio_async_in256 u_cs1
        (
         .ASYNC_IN (ddr3_cs1_async_in),
```

```
         .CONTROL  (ddr3_cs1_control)
         );

      vio_async_in256 u_cs2
        (
         .ASYNC_IN (ddr3_cs2_async_in),
         .CONTROL  (ddr3_cs2_control)
         );

      vio_async_in256 u_cs3
        (
         .ASYNC_IN (ddr3_cs3_async_in),
         .CONTROL  (ddr3_cs3_control)
         );

      vio_sync_out32 u_cs4
        (
         .SYNC_OUT (ddr3_cs4_sync_out),
         .CLK      (ddr3_cs4_clk),
         .CONTROL  (ddr3_cs4_control)
         );*/
    end
  endgenerate


// Add ML605 heartbeat counter and LED assignments
/*  reg   [28:0] led_counter;

always @( posedge clk )
  begin
    if ( rst )
      led_counter <= 0;
    else
      led_counter <= led_counter + 1;
  end

assign heartbeat = led_counter[27];*/

//System 1
wire [1:0] sys1_trigger;

wire sys1_app_wdf_wren;
wire sys1_app_wdf_end;
wire [26:0] sys1_app_addr;
wire [255:0] sys1_app_wdf_data;
wire [2:0] sys1_app_cmd;
wire sys1_app_en;

wire sys1_rdy;
wire sys1_wdf_rdy;
```

```
wire [63:0] sys1_data_write;
wire [26:0] sys1_address;
wire sys1_fill_wr_cmd;

wire [63:0] sys1_data_read;
wire sys1_fill_rd_cmd;
wire sys1_read_valid;
wire sys1_read_end;

wire[3:0] sys1_c_state;
wire sys1_switch;

wire [511:0] sys1_wd_fifo_dout;

//System 2
wire [1:0] sys2_trigger;

wire sys2_app_wdf_wren;
wire sys2_app_wdf_end;
wire [26:0] sys2_app_addr;
wire [255:0] sys2_app_wdf_data;
wire [2:0] sys2_app_cmd;
wire sys2_app_en;

wire sys2_rdy;
wire sys2_wdf_rdy;

wire [63:0] sys2_data_write;
wire [26:0] sys2_address;
wire sys2_fill_wr_cmd;

wire [63:0] sys2_data_read;
wire sys2_fill_rd_cmd;
wire sys2_read_valid;
wire sys2_read_end;

wire[3:0] sys2_c_state;
wire sys2_switch;

assign app_wdf_wren  =  (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_wdf_wren :
                    (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
sys2_app_wdf_wren :
                    0;

assign app_wdf_end =    (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_wdf_end :
                   (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
sys2_app_wdf_end :
```

```
                  0;

assign app_wdf_data = (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_wdf_data :
                   (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
      sys2_app_wdf_data :
                  256'd0;

assign app_addr  =      (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_addr :
                  (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
sys2_app_addr :
                  27'd0;

assign app_cmd  = (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_cmd :
                  (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
sys2_app_cmd :
                  3'd0;


assign app_en  =  (sys1_trigger == 2'b10 && sys2_trigger == 2'b01) ?
sys1_app_en :
                  (sys1_trigger == 2'b01 && sys2_trigger == 2'b10) ?
sys2_app_en :
                  0;
wire[5:0] sys1_timer;

//assign ila_DATA[2:0] = {sys2_app_en, sys2_app_wdf_wren, sys2_app_wdf_end &&
|sys2_app_cmd //&& |sys2_app_addr && |sys2_app_wdf_data};

user_design sys1_u_design
(
   .app_wdf_wren        (sys1_app_wdf_wren),
   .app_wdf_data        (sys1_app_wdf_data),
   .app_wdf_end         (sys1_app_wdf_end),
   .app_addr            (sys1_app_addr),
   .app_cmd             (sys1_app_cmd),
   .app_en              (sys1_app_en),
   .app_rdy             (app_rdy),
   .app_wdf_rdy         (app_wdf_rdy),
   .app_rd_data         (app_rd_data),
   .app_rd_data_end     (app_rd_data_end),
   .app_rd_data_valid   (app_rd_data_valid),
   .ui_clk_sync_rst     (ui_clk_sync_rst),
   .ui_clk              (clk),

      .switch(~switch),
      .trigger_activate(trigger_activate),
```

```
      .sys_rdy(sys1_rdy),
      .sys_wdf_rdy(sys1_wdf_rdy),

      .sys_address(sys1_address),
      .sys_data_write(sys1_data_write),
      .fill_wr_cmd(sys1_fill_wr_cmd),

      .sys_data_read(sys1_data_read),
      .fill_rd_cmd(sys1_fill_rd_cmd),
      .sys_read_valid(sys1_read_valid),
      .sys_read_end(sys1_read_end),

      .c_state(sys1_c_state),
      .trigger(sys1_trigger),

      .this_sys_switch(sys1_switch),
      .other_sys_switch(sys2_switch),

      .timer_out(sys1_timer),
      .wd_fifo_dout(sys1_wd_fifo_dout)
);

user_design sys2_u_design
(
   .app_wdf_wren        (sys2_app_wdf_wren),
   .app_wdf_data        (sys2_app_wdf_data),
   .app_wdf_end         (sys2_app_wdf_end),
   .app_addr            (sys2_app_addr),
   .app_cmd             (sys2_app_cmd),
   .app_en              (sys2_app_en),
   .app_rdy             (app_rdy),
   .app_wdf_rdy         (app_wdf_rdy),
   .app_rd_data         (app_rd_data),
   .app_rd_data_end     (app_rd_data_end),
   .app_rd_data_valid   (app_rd_data_valid),
   .ui_clk_sync_rst     (ui_clk_sync_rst),
   .ui_clk              (clk),


      .switch(switch),
      .trigger_activate(trigger_activate),

      .sys_rdy(sys2_rdy),
      .sys_wdf_rdy(sys2_wdf_rdy),

      .sys_address(sys2_address),
      .sys_data_write(sys2_data_write),
      .fill_wr_cmd(sys2_fill_wr_cmd),

      .sys_data_read(sys2_data_read),
```

```
      .fill_rd_cmd(sys2_fill_rd_cmd),
      .sys_read_valid(sys2_read_valid),
      .sys_read_end(sys2_read_end),

      .c_state(sys2_c_state),
      .trigger(sys2_trigger),

      .this_sys_switch(sys2_switch),
      .other_sys_switch(sys1_switch),

      .timer_out(),
      .wd_fifo_dout()
);


//wire sys1_fill_wr_exe2;
//wire sys1_fill_rd_exe2;

//assign sys1_fill_wr_exe2 = (switch == 8'b1000_0001 && sys1_fill_wr_exe) ?
1: 0;
//assign sys1_fill_rd_exe2 = (switch == 8'b1000_0001 && sys1_fill_rd_exe) ?
1: 0;

//wire sys2_fill_wr_exe2;
//wire sys2_fill_rd_exe2;

//assign sys2_fill_wr_exe2 = (switch == 8'b1000_0001 && sys2_fill_wr_exe) ?
1: 0;
//assign sys2_fill_rd_exe2 = (switch == 8'b1000_0001 && sys2_fill_rd_exe) ?
1: 0;

chipScope_stimulus SYS1_Stimulus (
      .ui_clk(clk),
      .sys_rdy(sys1_rdy),
      .sys_wdf_rdy(sys1_wdf_rdy),

      .sys_read_valid(sys1_read_valid),
      .sys_read_end(sys1_read_end),
      .sys_read_data(sys1_data_read),

      .fill_wr_exe(sys1_fill_wr_exe2),
      .fill_rd_exe(sys1_fill_rd_exe2),

      .fill_wr_cmd(sys1_fill_wr_cmd),
      .fill_rd_cmd(sys1_fill_rd_cmd),

      .wr_data(sys1_data_write),
      .address(sys1_address)
    );
```

```
chipScope_stimulus SYS2_Stimulus (
      .ui_clk(clk),
      .sys_rdy(sys2_rdy),
      .sys_wdf_rdy(sys2_wdf_rdy),

      .sys_read_valid(sys2_read_valid),
      .sys_read_end(sys2_read_end),
      .sys_read_data(sys2_data_read),

      .fill_wr_exe(sys2_fill_wr_exe2),
      .fill_rd_exe(sys2_fill_rd_exe2),


      .fill_wr_cmd(sys2_fill_wr_cmd),
      .fill_rd_cmd(sys2_fill_rd_cmd),

      .wr_data(sys2_data_write),
      .address(sys2_address)
    );


/*
wire[63:0] ila_DATA;
wire [7:0] ila_TRIG0;

ICON debug_icon (
    .CONTROL0(ila_CONTROL) // INOUT BUS [35:0]
);

ILA debug_ila (
    .CONTROL(ila_CONTROL), // INOUT BUS [35:0]
    .CLK(clk_mem), // IN
    .DATA(ila_DATA), // IN BUS [63:0]
    .TRIG0(ila_TRIG0) // IN BUS [7:0]
);


assign ila_DATA[63] = clk;
assign ila_DATA[62:60] = sys1_c_state[3:1];
assign ila_DATA[9] = sys1_c_state[0];
assign ila_DATA[59:55] = sys1_address[7:3];//address
assign ila_DATA[54] = app_rdy;
assign ila_DATA[53] = app_en;

//fill write
assign ila_DATA[52] = sys1_data_write[0];
assign ila_DATA[51] = sys1_rdy;
assign ila_DATA[50] = sys1_wdf_rdy;

assign ila_DATA[49] = sys1_data_read[0];
```

```
assign ila_DATA[48] = sys1_read_end;
assign ila_DATA[47] = sys1_read_valid;

assign ila_DATA[46:45] = 2'd0;


assign ila_DATA[44] = sys1_fill_wr_cmd;
assign ila_DATA[43] = sys1_fill_wr_exe;

//exe write
assign ila_DATA[42] = trigger_activate;
assign ila_DATA[41] = app_wdf_rdy;
assign ila_DATA[40] = app_wdf_end;
assign ila_DATA[39] = app_wdf_wren;
assign ila_DATA[38:35] = {app_wdf_data[192], app_wdf_data[128],
app_wdf_data[64], app_wdf_data[0]};

//fill read
assign ila_DATA[34] = sys1_fill_rd_cmd;
assign ila_DATA[33] = sys1_fill_rd_exe;

//exe read
assign ila_DATA[32] = sys1_rdy;
assign ila_DATA[31:27] = app_addr[7:3];
assign ila_DATA[26:23] = {app_rd_data[192], app_rd_data[128],
app_rd_data[64], app_rd_data[0]};
assign ila_DATA[22] =    app_rd_data_end;
assign ila_DATA[21] =   app_rd_data_valid;

//read back
assign ila_DATA[20] = sys1_wdf_rdy;

assign ila_DATA[19:12] = {sys1_wd_fifo_dout[448], sys1_wd_fifo_dout[384],
sys1_wd_fifo_dout[320], sys1_wd_fifo_dout[256], sys1_wd_fifo_dout[192],
sys1_wd_fifo_dout[128], sys1_wd_fifo_dout[64], sys1_wd_fifo_dout[0]};

assign ila_DATA[11:10] = sys1_trigger;
assign ila_DATA[8:7] = sys2_trigger;

//SPARE pins
assign ila_DATA[6:3] = {sys1_app_wdf_data[192], sys1_app_wdf_data[128],
sys1_app_wdf_data[64], sys1_app_wdf_data[0]};
//assign ila_DATA[2:0] = {sys2_app_en, sys2_app_wdf_wren, sys2_app_wdf_end};

assign ila_TRIG0 = {    n_button,
                                    s_button,
                                    c_button,
                                    sys1_fill_wr_exe,
                                    trigger_activate,
                                    sys1_fill_rd_exe,
```

```
                                        sys2_fill_rd_exe,
                                      sys2_fill_wr_exe}; //all the outputs
from button debouncers

PushButton_Debouncer n_debouncer (
.clk(clk),
.PB(n_button),
.PB_state(),
.PB_down(sys1_fill_wr_exe),
.PB_up()
);

PushButton_Debouncer s_debouncer (  //pass through
.clk(clk),
.PB(s_button),
.PB_state(),
.PB_down(sys2_fill_wr_exe),
.PB_up()
);

PushButton_Debouncer w_debouncer (
.clk(clk),
.PB(w_button),
.PB_state(),
.PB_down(sys1_fill_rd_exe),
.PB_up()
);

PushButton_Debouncer e_debouncer (  //pass through
.clk(clk),
.PB(e_button),
.PB_state(),
.PB_down(sys2_fill_rd_exe),
.PB_up()
);
PushButton_Debouncer c_debouncer (  //pass through
.clk(clk),
.PB(c_button),
.PB_state(),
.PB_down(trigger_activate),
.PB_up()
);*/

endmodule
```

User_design(Arbiter_block)

```
`timescale 1ns / 1ps

module user_design(
   output app_wdf_wren,
   output [255:0] app_wdf_data,
   output app_wdf_end,
   output [27-1:0] app_addr,  //addr for accessing the memory
   output [2:0] app_cmd,
   output app_en,
   input app_rdy,      //signal indicating whether the memory is able to
accept new commands
   input app_wdf_rdy,    //write mig fifo ready to receive data
      input [255:0] app_rd_data,
   input app_rd_data_end,   //current clock cycle is last of output read data
   input app_rd_data_valid,  //wait for it to = 1 to process read data
   input ui_clk_sync_rst,
   input ui_clk,

      input [7:0] switch,
      input trigger_activate,

      output sys_rdy,
      output sys_wdf_rdy,

      input[26:0] sys_address,
      input[63:0] sys_data_write,
      input fill_wr_cmd,

      output[63:0] sys_data_read,
      input fill_rd_cmd,
      output sys_read_valid,
      output sys_read_end,

      output [3:0] c_state,
      output [1:0] trigger,

      output this_sys_switch,
      input other_sys_switch,

      output [5:0] timer_out,
      output [511:0] wd_fifo_dout
);
      reg sys_read_valid;
      reg sys_read_end;

      wire [5:0] timer_out;
      assign timer_out = timer;
```

```
	wire this_sys_switch;
	assign this_sys_switch = (mc_c_state == 4'd10) ? 1: 0;

	wire [1:0] trigger;
	assign trigger = sys_trigger;

	wire sys_rdy;  //system can send any commands
	assign sys_rdy = ((mc_c_state == 4'd8 && ~wd_fifo_almost_full) ||
(mc_c_state == 4'd9 && ~ra_fifo_almost_full)) ? 1: 0;

	wire sys_wdf_rdy;  //system can only send write commands
	assign sys_wdf_rdy = ((mc_c_state == 4'd8 || mc_c_state == 4'd11) &&
~wd_fifo_almost_full) ? 1: 0;

	reg [1:0] sys_trigger;
	wire [3:0] c_state;
	assign c_state = mc_c_state;

	wire [3:0] mem_wdf_data;
	wire [3:0] mem_read_data;

	/****FIFO registers, wires, instantiation****/
	reg w_fifo_rd_en;
	wire wa_fifo_wr_en;
	assign wa_fifo_wr_en = ~wa_fifo_full && fill_wr_cmd && mc_c_state ==
4'd8;

	wire[26:0] wa_fifo_dout;
	reg [26:0] wa_fifo_dout_reg;
	reg wa_fifo_rd_en;


wr_addr_fifo wa_fifo (
	.wr_clk(ui_clk), // input wr_clk
	.rd_clk(ui_clk), // input rd_clk
	.din(sys_address), // input [26 : 0] din
	.wr_en(wa_fifo_wr_en), // input wr_en
	.rd_en(w_fifo_rd_en), // input rd_en
	.dout(wa_fifo_dout), // output [26 : 0] dout
	.full(wa_fifo_full), // output full
	.almost_full(wa_fifo_almost_full), // output almost_full
	.empty(wa_fifo_empty), // output empty
	.almost_empty(wa_fifo_almost_empty), // output almost_empty
	.rd_data_count(), // output [5 : 0] rd_data_count
	.wr_data_count() // output [5 : 0] wr_data_count
);


	//registers/wires used for reading from the fifo
```

```
	wire [5:0] wd_fifo_rd_data_count;   //test bench
	reg [5:0] wd_fifo_rd_data_count_reg;

	wire [5:0] wd_fifo_wr_data_count;
	assign S_LED = |wd_fifo_wr_data_count;

	//511:0 data, 538:512 address
	wire [511:0] wd_fifo_dout;
	reg [511:0] wd_fifo_dout_reg;

	wire wd_fifo_empty;
	wire wd_fifo_full;
	wire wd_fifo_almost_full;

	//registers/wires used for writing to the fifo
	wire wd_fifo_wr_en;
	assign wd_fifo_wr_en = (~wd_fifo_full && fill_wr_cmd && mc_c_state ==
4'd8) || mc_c_state == 4'd11;

	//511:0 data, 538:512 address
	wire [63:0] wd_fifo_din;
	assign wd_fifo_din = sys_data_write;

	FIFO wd_fifo (
		.wr_clk(ui_clk), // input wr_clk
		.rd_clk(ui_clk), // input rd_clk
		.din(wd_fifo_din), // input [63 : 0] din
		.wr_en(wd_fifo_wr_en), // input wr_en
		.rd_en(w_fifo_rd_en), // input rd_en
		.dout(wd_fifo_dout), // output [511 : 0] dout
		.full(wd_fifo_full), // output full
		.almost_full(wd_fifo_almost_full), // output almost_full
		.empty(wd_fifo_empty), // output empty
		.almost_empty(almost_empty), // output almost_empty
		.rd_data_count(wd_fifo_rd_data_count), // output [4 : 0]
rd_data_count
		.wr_data_count(wd_fifo_wr_data_count) // output [5 : 0]
wr_data_count
	);

	/*************** Read address fifo ***************/

	wire  ra_fifo_rd_en;

	wire [5:0] ra_fifo_rd_data_count;   //test bench
	wire [5:0] ra_fifo_wr_data_count;

	//511:0 data, 538:512 address
	wire [26:0] ra_fifo_dout;
	reg [26:0] ra_fifo_dout_reg;
```

```
	wire ra_fifo_empty;
	wire ra_fifo_almost_empty;
	wire ra_fifo_full;
	wire ra_fifo_almost_full;

	wire ra_fifo_wr_en;
	assign ra_fifo_wr_en    = ~ra_fifo_full && fill_rd_cmd;


	rd_addr_fifo ra_fifo
	(
		.wr_clk(ui_clk), // input wr_clk
		.rd_clk(ui_clk), // input rd_clk
		.din(sys_address), // input [26 : 0] din
		.wr_en(ra_fifo_wr_en), // input wr_en
		.rd_en(ra_fifo_rd_en), // input rd_en
		.dout(ra_fifo_dout), // output [26 : 0] dout
		.full(), // output full
		.almost_full(ra_fifo_full), // output almost_full
		.empty(ra_fifo_empty), // output empty
		.almost_empty(ra_fifo_almost_empty), // output almost_empty
		.rd_data_count(ra_fifo_rd_data_count), // output [5 : 0]
rd_data_count
		.wr_data_count(ra_fifo_wr_data_count), // output [5 : 0]
wr_data_count
		.prog_full(ra_fifo_almost_full) // output prog_full
	);

	/*************** Read data fifo ***************/
	reg   [255:0] rd_fifo_first_half;
	wire  [511:0] rd_fifo_din;

	wire  rd_fifo_wr_en;
	wire  rd_fifo_full;

	wire  rd_fifo_rd_en;
	wire  rd_fifo_empty;
	wire  rd_fifo_almost_empty;

	wire  [8:0] rd_fifo_rd_data_count;
	wire  [4:0] rd_fifo_wr_data_count;

	wire  [63:0] rd_fifo_dout;
	reg   [2:0] read_back_counter;

	//reg [511:0] rd_fifo_dout_reg;

	assign rd_fifo_din =  ((mc_c_state == 4'd3 || mc_c_state == 4'd5) &&
app_rd_data_end) ? {rd_fifo_first_half, app_rd_data}:
```

```
                                                      512'd0;

      assign rd_fifo_wr_en =  ((mc_c_state == 4'd3 || mc_c_state == 4'd5) &&
(app_rd_data_end && app_rd_data_valid) && ~rd_fifo_full) ? 1:
                                                      0;

      assign sys_data_read = rd_fifo_dout;

      rd_data_fifo rd_fifo(
            .wr_clk(ui_clk), //mem_clk
            .rd_clk(ui_clk), //sys_clk
            .din(rd_fifo_din), // input [511 : 0] din
            .wr_en(rd_fifo_wr_en), // input wr_en
            .rd_en(rd_fifo_rd_en), // input rd_en
            .dout(rd_fifo_dout), // output [63 : 0] dout
            .full(rd_fifo_full), // output full
            .empty(rd_fifo_empty), // output empty
            .almost_empty(rd_fifo_almost_empty), // output almost_empty
            .rd_data_count(rd_fifo_rd_data_count), // output [8 : 0]
rd_data_count
            .wr_data_count(rd_fifo_wr_data_count) // output [4 : 0]
wr_data_count
      );


      /*************** State Machine ****************/
      reg app_wdf_wren_reg;      //active high strobe for app_wdf_data
      reg app_wdf_end_reg;       //current clock cycle is last cycle of input
data
      reg [2:0] app_cmd_reg;     //read or write
      reg app_en_reg;               //strobe for app_addr, app_cmd, app_sz,
and app_hi_pri

      assign app_wdf_wren = app_wdf_wren_reg;
      assign app_wdf_end = app_wdf_end_reg;
      assign app_cmd = app_cmd_reg;
      //assign app_en = app_en_reg;
      assign app_en = (mc_c_state == 4'd2 || mc_c_state == 4'd3) && app_rdy ?
1: 0;

      reg [3:0] mc_c_state;  //current state
      reg [3:0] mc_n_state;  //next state

      assign mem_read_data = {app_rd_data[192], app_rd_data[128],
app_rd_data[64], app_rd_data[0]};
      assign mem_wdf_data = {app_wdf_data[192], app_wdf_data[128],
app_wdf_data[64], app_wdf_data[0]};

//*****Initial_Conditions******//
initial
```

```
begin
      read_back_counter <= 3'd0;
   sys_trigger <= 2'd0;
      mc_c_state <= 4'd0;
      mc_n_state <= 4'd0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 1'b0;
      app_wdf_end_reg <= 1'b0;
end

//**** MC_N_STATE_STATE MACHINES*****/

//update fifo_read_data_reg when n_state == 1 and c_state is 0 or 2
always @(posedge ui_clk)
begin
      if(mc_c_state == 4'd0 && mc_n_state == 4'd1)
      begin
            wd_fifo_dout_reg <= wd_fifo_dout;
            wa_fifo_dout_reg <= wa_fifo_dout;
      end
      else if(mc_c_state == 4'd2 && mc_n_state == 4'd1)
      begin
            wd_fifo_dout_reg <= wd_fifo_dout;
            wa_fifo_dout_reg <= wa_fifo_dout;
      end
      else if(mc_c_state == 4'd2 && mc_n_state == 4'd0)
      begin
            wd_fifo_dout_reg <= 512'd0;
            wa_fifo_dout_reg <= 27'd0;
      end
      else
      begin
            wd_fifo_dout_reg <= wd_fifo_dout_reg;
            wa_fifo_dout_reg <= wa_fifo_dout_reg;
      end
end

//set app_wdf_data to the appropriate portions of the whole
fifo_read_data_reg
//depending on the current state
assign app_wdf_data = (mc_c_state == 4'd1) ? wd_fifo_dout_reg[511:256]:
                                             (mc_c_state == 4'd2) ?
wd_fifo_dout_reg[255:0]:
                                             256'd0;


assign app_addr = (mc_c_state == 4'd1 || mc_c_state == 4'd2)      ?
wa_fifo_dout_reg:
                                       (mc_c_state == 4'd3)
                        ? ra_fifo_dout:
```

```
                                    27'd0;

assign ra_fifo_rd_en = (mc_c_state == 4'd3 && app_rdy == 1) ? 1:
                                                      0;

assign rd_fifo_rd_en = (mc_c_state ==4'd7) ? 1: 0;


//State 8/9 Timer
reg [5:0] timer;

always @(posedge ui_clk)
begin
      if(mc_c_state != 4'd8 && mc_c_state != 4'd11 && mc_n_state == 4'd8)
            timer <= 6'd0;
      else if((mc_n_state == 4'd8 || mc_n_state == 4'd9 || mc_n_state ==
4'd11) && timer < 6'd63)
            timer <= timer + 1;
      else if(mc_n_state == 4'd10)
            timer <= 6'd0;
      else
            timer <= timer;
end

always @(posedge ui_clk)
begin
      if(mc_c_state == 4'd10 && mc_n_state == 4'd0)
      begin
            sys_trigger <= ~sys_trigger;
      end
      else if(trigger_activate && switch == 8'b00001111)
            sys_trigger <= 2'b01;
      else if(trigger_activate && switch == 8'b11110000)
            sys_trigger <= 2'b10;
      else
            sys_trigger <= sys_trigger;
end

reg [2:0] write_counter;

//current state is set to next state, and all registers are updated
accordingly
always @(posedge ui_clk)
begin
      mc_c_state <= mc_n_state;

      case(mc_n_state)
      4'd0:
      begin
            //w_fifo_rd_data_count_reg <= w_fifo_rd_data_count;
```

```
      //set initial amount of data in fifo

//register will be controlled (decremented) by state machine

//because the fifo has to much delay from when data is pushed out

//and rd_data_count is updated


      app_wdf_wren_reg <= 1'b0;
      app_wdf_end_reg <= 1'b0;
      app_cmd_reg <= 3'b111;

//default state of app_cmd and app_en

      app_en_reg <= 0;
      w_fifo_rd_en <= 0;
end

4'd1:
Begin
      //if all readies are high and there is data in fifo
      if(~wa_fifo_empty && mc_c_state != 4'd1)
      begin
            //increment address, decrement number in fifo, and read
      from fifo
            w_fifo_rd_en <= 1;
      end
      else
            w_fifo_rd_en <= 0;


      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_cmd_reg <= 3'b000;
      //write command
      app_wdf_wren_reg <= 1'b1;
      app_wdf_end_reg <= 1'b0;
end


4'd2:
Begin
      //if app_en was being sent, but app_rdy low, keep app_en high
      if(app_en_reg && ~app_rdy)
            app_en_reg <= 1;

      //app_en was not high, set app_en high for sending 512 bit packet
      else if(~app_en_reg && app_rdy)
```

```
            app_en_reg <= 1;
      else
            app_en_reg <= 0;

      if(app_wdf_wren_reg && app_wdf_end_reg && ~app_wdf_rdy)
      begin
            app_wdf_wren_reg <= 1;
            app_wdf_end_reg <= 1;
      end
      else if(app_wdf_wren_reg && ~app_wdf_end_reg && app_wdf_rdy)
      begin
            app_wdf_wren_reg <= 1;
            app_wdf_end_reg <= 1;
      end
      else
      begin
            app_wdf_wren_reg <= 0;
            app_wdf_end_reg <= 0;
      end

      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_cmd_reg <= 3'b000;
      w_fifo_rd_en <= 0;
end

4'd3:
begin
      if(app_rdy)
            app_en_reg <= 1;
      else
            app_en_reg <= 0;

      if(~app_rd_data_end && app_rd_data_valid)
            rd_fifo_first_half <= app_rd_data;

      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_cmd_reg <= 3'b001;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
4'd4:
begin
      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
```

```
4'd5:
begin
      rd_fifo_first_half <= app_rd_data;

      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
4'd6:
begin
      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
4'd7:
begin
      if(mc_c_state != 3'd7)
            read_back_counter <= 3'd0;
      else
            read_back_counter <= read_back_counter + 1;

      if(read_back_counter == 3'd6)
            sys_read_end <= 1;
      else
            sys_read_end <= 0;

      sys_read_valid <= 1;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
4'd8:
begin
      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
end
4'd9:
begin
      sys_read_end <= 0;
      sys_read_valid <= 0;
      app_en_reg <= 0;
      app_wdf_wren_reg <= 0;
      app_wdf_end_reg <= 0;
```

```
	end
	4'd10:
	begin
		sys_read_end <= 0;
		sys_read_valid <= 0;
		app_en_reg <= 0;
		app_wdf_wren_reg <= 0;
		app_wdf_end_reg <= 0;
	end
	4'd11:
	begin
		if(mc_c_state == 4'd8)
			write_counter <= 1;
		else
			write_counter <= write_counter + 1;

		sys_read_end <= 0;
		sys_read_valid <= 0;
		app_en_reg <= 0;
		app_wdf_wren_reg <= 0;
		app_wdf_end_reg <= 0;
	end

	endcase
end



//next state, mc_n_state is determined
always @(mc_c_state, app_rdy, app_wdf_rdy, ra_fifo_empty,
ra_fifo_almost_empty, app_rd_data_valid, app_rd_data_end,
rd_fifo_empty, sys_trigger, timer, fill_rd_cmd, fill_wr_cmd,
other_sys_switch, this_sys_switch, wa_fifo_empty, wd_fifo_full,
write_counter)
begin
	case(mc_c_state)
	4'd0: //Ready state
	begin
		if(sys_trigger == 2'b01)
		begin
			if(rd_fifo_empty)
				mc_n_state <= 4'd8;
			else
				mc_n_state <= 4'd7;
		end
		else if(sys_trigger == 2'b10)
		begin
			if(~wa_fifo_empty)
				mc_n_state <= 4'd1;
			else if(~ra_fifo_empty)
```

```
                begin
                        mc_n_state <= 4'd3;
                end
                else
                        mc_n_state <= 4'd10;
        end

end
4'd1: //Send first data packet
begin
        if(app_wdf_rdy && app_rdy)
                mc_n_state <= 4'd2;
        else
                mc_n_state <= 4'd1;
end
4'd2: //Send second data packet
begin
        if(app_rdy && app_wdf_rdy)
        begin
                if(wa_fifo_empty)
                begin
                        if(ra_fifo_empty)
                                mc_n_state <= 4'd10;
                        else
                                mc_n_state <= 4'd3;
                end
                else
                        mc_n_state <= 4'd1;
        end
        else
                mc_n_state <= 4'd2;

end //end write chain

4'd3: //check if read is valid, set app_en to send address for
retrieval
begin
   if(~app_rdy || ~ra_fifo_almost_empty)
   begin
      mc_n_state <= 4'd3;
   end

      else if(app_rd_data_valid && app_rd_data_end)
      begin
            mc_n_state <= 4'd6;
      end
      else if(app_rd_data_valid && ~app_rd_data_end)
      begin
            mc_n_state <= 4'd5;
      end
```

```
            else if(~app_rd_data_valid && ~app_rd_data_end)
            begin
                  mc_n_state <= 4'd4;
            end
      end

      4'd4: //wait for read valid state
      Begin
      //wait for app_rd_data_valid to be asserted, set app_en = 0 (can only
be high for one clk cycle)
            if(app_rd_data_valid)
            begin
                  mc_n_state <= 4'd5;
            end
            else
                  mc_n_state <= 4'd4;
      end
      4'd5:
      begin
            if(app_rd_data_valid && app_rd_data_end)
            begin
                  mc_n_state <= 4'd6;
            end
            else
                  mc_n_state <= 4'd5;
      end

      4'd6:
      begin
            if(app_rd_data_valid)
            begin
                  if(app_rd_data_end)
                        mc_n_state <= 4'd6;
                  else
                        mc_n_state <= 4'd5;
            end
            else
                  mc_n_state <= 4'd10;
      end

      4'd7: //rd_fifo readback
      begin
            if(rd_fifo_empty)
                  mc_n_state <= 4'd8;
            else
                  mc_n_state <= 4'd7;
      end

      4'd8: //w_fifo and ra_fifo ready. start timer
      begin
```

```
            if(fill_wr_cmd && ~wd_fifo_full)
                  mc_n_state <= 4'd11;
            else if(fill_rd_cmd || wd_fifo_full)
                  mc_n_state <= 4'd9;
            else if(timer == 6'd63 && other_sys_switch)
                  mc_n_state <= 4'd10;
            else
                  mc_n_state <= 4'd8;

      end

      4'd9: //ra_fifo ready
      begin
            if((timer == 6'd63 && other_sys_switch) || ra_fifo_almost_full)
                  mc_n_state <= 4'd10;
            else
                  mc_n_state <= 4'd9;
      end

      4'd10:
      begin
            if(~other_sys_switch || ~this_sys_switch)
                  mc_n_state <= 4'd10;
            else
                  mc_n_state <= 4'd0;
      end

      4'd11:
      begin
            if(write_counter < 3'd7)
                  mc_n_state <= 4'd11;
            else if(timer == 6'd63 && other_sys_switch)
                  mc_n_state <= 4'd10;
            else
                  mc_n_state <= 4'd8;
      end

      endcase
end

endmodule
```

Stimulus

```
`timescale 1ns / 1ps
module chipScope_stimulus(
      input ui_clk,

      input sys_rdy,
      input sys_wdf_rdy,
```

```
      input[63:0] sys_read_data,
      input sys_read_valid,
      input sys_read_end,

      input fill_wr_exe,
      input fill_rd_exe,

      output fill_wr_cmd,
      output fill_rd_cmd,
      output[63:0] wr_data,
      output[26:0] address
      );

      wire [63:0] sys_read_data;

      reg sys_read_valid_reg;

      reg rd_mode;
      reg wr_mode;

      reg[7:0] counter1;
      reg[7:0] counter2;
      reg[7:0] counter3;

      assign fill_wr_cmd = wr_mode && sys_rdy && sys_wdf_rdy && write_counter
== 3'd0;
      assign fill_rd_cmd = rd_mode && sys_rdy;

      assign wr_data =  (write_counter == 3'd7) ? {63'd0, counter1[0]}:
                                          (write_counter == 3'd6) ? {63'd0,
counter1[1]}:
                                          (write_counter == 3'd5) ? {63'd0,
counter1[2]}:
                                          (write_counter == 3'd4) ? {63'd0,
counter1[3]}:
                                          (write_counter == 3'd3) ? {63'd0,
counter1[4]}:
                                          (write_counter == 3'd2) ? {63'd0,
counter1[5]}:
                                          (write_counter == 3'd1) ? {63'd0,
counter1[6]}:
                                          (write_counter == 3'd0) ? {63'd0,
counter1[7]}:
                                          64'd0;

      assign address =  (wr_mode) ? {19'd0, counter1, 3'd0}:
                                          (rd_mode) ? {19'd0, counter3,
3'd0}:
                                          27'd0;
```

```
	reg [511:0] readback_data;
	reg [511:0] readback_data_temp;

	reg [2:0] readback_counter;
	reg [2:0] write_counter;

	initial
	begin
		counter1 = 8'd0;
		counter2 = 8'd0;
		counter3 = 8'd0;
		wr_mode <= 0;
	end

	always@(negedge ui_clk)
	begin
		sys_read_valid_reg <= sys_read_valid;

		if(sys_read_valid && ~sys_read_valid_reg)
			readback_counter <= 3'd0;

		else if(sys_read_valid)
			readback_counter <= readback_counter + 1;
	end


	always@(posedge ui_clk)
	begin
		if(sys_read_valid && readback_counter == 3'd0)
			readback_data_temp[63:0] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd1)
			readback_data_temp[127:64] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd2)
			readback_data_temp[191:128] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd3)
			readback_data_temp[255:192] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd4)
			readback_data_temp[319:256] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd5)
			readback_data_temp[383:320] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd6)
			readback_data_temp[447:384] <= sys_read_data;
		else if(sys_read_valid && readback_counter == 3'd7)
			readback_data <= {sys_read_data,
readback_data_temp[447:0]};

	end
```

```
always@(posedge ui_clk)
begin
      if(wr_mode && ~sys_rdy && write_counter == 3'd0)
            write_counter <= write_counter;
      else if(wr_mode)
            write_counter <= write_counter + 1;

      if(fill_wr_exe && ~rd_mode && ~wr_mode)
      begin
            wr_mode <= 1;
            write_counter <= 3'd0;
            counter1 <= 8'd0;
            counter2 <= 8'd0;
      end
      else if(wr_mode && counter1 < 8'd32)
      begin
            if(write_counter == 3'd7)
            begin
                  counter1 <= counter1 + 1;
                  counter2 <= counter2 + 1;
            end
            else
            begin
                  counter1 <= counter1;
                  counter2 <= counter2;
            end
      end
      else if(write_counter == 3'd7)
            wr_mode <= 0;
      else
            wr_mode <= wr_mode;

      if(fill_rd_exe && ~wr_mode && ~rd_mode)
      begin
            rd_mode <= 1;
            counter3 <= 8'd0;
      end
      else if(rd_mode && counter3 < 8'd32)
      begin
            if(sys_rdy)
                  counter3 <= counter3 + 1;
            else
                  counter3 <= counter3;
      end
      else
            rd_mode <= 0;

end
```

```
endmodule
```

Verilog Testbench

```
`timescale 1ps/100fs

module sim_tb_top;

  parameter REFCLK_FREQ           = 200;
                                    // # = 200 when design frequency < 533
MHz,
                                    //   = 300 when design frequency >= 533
MHz.
  parameter SIM_BYPASS_INIT_CAL   = "FAST";
                                    // # = "OFF" -  Complete memory init &
                                    //              calibration sequence
                                    // # = "SKIP" - Skip memory init &
                                    //              calibration sequence
                                    // # = "FAST" - Skip memory init & use
                                    //              abbreviated calib
sequence
  parameter RST_ACT_LOW           = 1;
                                    // =1 for active low reset,
                                    // =0 for active high.
  parameter IODELAY_GRP           = "IODELAY_MIG";
                                    //to phy_top
  parameter nCK_PER_CLK           = 2;
                                    // # of memory CKs per fabric clock.
                                    // # = 2, 1.
  parameter nCS_PER_RANK          = 1;
                                    // # of unique CS outputs per Rank for
                                    // phy.
  parameter DQS_CNT_WIDTH         = 3;
                                    // # = ceil(log2(DQS_WIDTH)).
  parameter RANK_WIDTH            = 1;
                                    // # = ceil(log2(RANKS)).
  parameter BANK_WIDTH            = 3;
                                    // # of memory Bank Address bits.
  parameter CK_WIDTH              = 1;
                                    // # of CK/CK# outputs to memory.
  parameter CKE_WIDTH             = 1;
                                    // # of CKE outputs to memory.
  parameter COL_WIDTH             = 10;
                                    // # of memory Column Address bits.
  parameter CS_WIDTH              = 1;
                                    // # of unique CS outputs to memory.
  parameter DM_WIDTH              = 8;
                                    // # of Data Mask bits.
  parameter DQ_WIDTH              = 64;
```

```
                                  // # of Data (DQ) bits.
  parameter DQS_WIDTH             = 8;
                                  // # of DQS/DQS# bits.
  parameter ROW_WIDTH             = 13;
                                  // # of memory Row Address bits.
  parameter BURST_MODE            = "8";
                                  // Burst Length (Mode Register 0).
                                  // # = "8", "4", "OTF".
  parameter INPUT_CLK_TYPE        = "DIFFERENTIAL";
                                  // input clock type DIFFERENTIAL or
SINGLE_ENDED
  parameter BM_CNT_WIDTH          = 2;
                                  // # = ceil(log2(nBANK_MACHS)).
  parameter ADDR_CMD_MODE         = "1T" ;
                                  // # = "2T", "1T".
  parameter ORDERING              = "STRICT";
                                  // # = "NORM", "STRICT".
  parameter RTT_NOM               = "60";
                                  // RTT_NOM (ODT) (Mode Register 1).
                                  // # = "DISABLED" - RTT_NOM disabled,
                                  //   = "120" - RZQ/2,
                                  //   = "60" - RZQ/4,
                                  //   = "40" - RZQ/6.
   parameter RTT_WR               = "OFF";
                                     // RTT_WR (ODT) (Mode Register 2).
                                     // # = "OFF" - Dynamic ODT off,
                                     //   = "120" - RZQ/2,
                                     //   = "60" - RZQ/4,
  parameter OUTPUT_DRV            = "HIGH";
                                  // Output Driver Impedance Control (Mode
Register 1).
                                  // # = "HIGH" - RZQ/7,
                                  //   = "LOW" - RZQ/6.
  parameter REG_CTRL              = "OFF";
                                  // # = "ON" - RDIMMs,
                                  //   = "OFF" - Components, SODIMMs,
UDIMMs.
  parameter CLKFBOUT_MULT_F       = 6;
                                  // write PLL VCO multiplier.
  parameter DIVCLK_DIVIDE         = 2;
                                  // write PLL VCO divisor.
  parameter CLKOUT_DIVIDE         = 3;
                                  // VCO output divisor for fast (memory)
clocks.
  parameter tCK                   = 2500;
                                  // memory tCK paramter.
                                  // # = Clock Period.
  parameter DEBUG_PORT            = "OFF";
                                  // # = "ON" Enable debug
signals/controls.
```

```
                                //   = "OFF" Disable debug
signals/controls.
  parameter tPRDI                = 1_000_000;
                                // memory tPRDI paramter.
  parameter tREFI                = 7800000;
                                // memory tREFI paramter.
  parameter tZQI                 = 128_000_000;
                                // memory tZQI paramter.
  parameter ADDR_WIDTH           = 27;
                                // # = RANK_WIDTH + BANK_WIDTH
                                //     + ROW_WIDTH + COL_WIDTH;
  parameter STARVE_LIMIT         = 2;
                                // # = 2,3,4.
  parameter TCQ                  = 100;
  parameter ECC                  = "OFF";
  parameter ECC_TEST             = "OFF";


  //***************************************************************************//
  // Traffic Gen related parameters
  //***************************************************************************//
  parameter EYE_TEST             = "FALSE";
                                   // set EYE_TEST = "TRUE" to probe
memory
                                   // signals. Traffic Generator will only
                                   // write to one single location and no
                                   // read transactions will be generated.
  parameter DATA_PATTERN         = "DGEN_ALL";
                                    // "DGEN_HAMMER", "DGEN_WALKING1",
                                    // "DGEN_WALKING0","DGEN_ADDR","
                                    //
"DGEN_NEIGHBOR","DGEN_PRBS","DGEN_ALL"
  parameter CMD_PATTERN          = "CGEN_ALL";
                                    //
"CGEN_RPBS","CGEN_FIXED","CGEN_BRAM",
                                    // "CGEN_SEQUENTIAL", "CGEN_ALL"

  parameter BEGIN_ADDRESS        = 32'h00000000;
  parameter PRBS_SADDR_MASK_POS  = 32'h00000000;
  parameter END_ADDRESS          = 32'h000003ff;
  parameter PRBS_EADDR_MASK_POS  = 32'hfffffc00;
  parameter SEL_VICTIM_LINE      = 11;


//***************************************************************************/
/
  // Local parameters Declarations

//***************************************************************************/
/
```

```
  localparam real TPROP_DQS          = 0.00;  // Delay for DQS signal during
Write Operation
  localparam real TPROP_DQS_RD       = 0.00;  // Delay for DQS signal during
Read Operation
  localparam real TPROP_PCB_CTRL     = 0.00;  // Delay for Address and Ctrl
signals
  localparam real TPROP_PCB_DATA     = 0.00;  // Delay for data signal during
Write operation
  localparam real TPROP_PCB_DATA_RD  = 0.00;  // Delay for data signal during
Read operation

  localparam MEMORY_WIDTH = 16;
  localparam NUM_COMP = DQ_WIDTH/MEMORY_WIDTH;
  localparam real CLK_PERIOD = tCK;
  localparam real REFCLK_PERIOD = (1000000.0/(2*REFCLK_FREQ));
  localparam DRAM_DEVICE = "SODIMM";
                         // DRAM_TYPE: "UDIMM", "RDIMM", "COMPS"

   // VT delay change options/settings
  localparam VT_ENABLE                  = "OFF";
                                        // Enable VT delay var's
  localparam VT_RATE                    = CLK_PERIOD/500;
                                        // Size of each VT step
  localparam VT_UPDATE_INTERVAL         = CLK_PERIOD*50;
                                        // Update interval
  localparam VT_MAX                     = CLK_PERIOD/40;
                                        // Maximum VT shift



//*****************************************************************************/
/
  // Wire Declarations

//*****************************************************************************/
/
  reg sys_clk;
  reg clk_ref;
  reg sys_rst_n;

  wire sys_clk_p;
  wire sys_clk_n;
  wire clk_ref_p;
  wire clk_ref_n;


  reg [DM_WIDTH-1:0]                ddr3_dm_sdram_tmp;

  wire sys_rst;
```

```
  //wire                                error;
//  wire                                phy_init_done;
  wire                               ddr3_parity;
  wire                               ddr3_reset_n;
//  wire                                sda;
//  wire                                scl;

  wire [DQ_WIDTH-1:0]                ddr3_dq_fpga;
  wire [ROW_WIDTH-1:0]               ddr3_addr_fpga;
  wire [BANK_WIDTH-1:0]              ddr3_ba_fpga;
  wire                               ddr3_ras_n_fpga;
  wire                               ddr3_cas_n_fpga;
  wire                               ddr3_we_n_fpga;
  wire [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_cs_n_fpga;
  wire [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_odt_fpga;
  wire [CKE_WIDTH-1:0]               ddr3_cke_fpga;
  wire [DM_WIDTH-1:0]                ddr3_dm_fpga;
  wire [DQS_WIDTH-1:0]               ddr3_dqs_p_fpga;
  wire [DQS_WIDTH-1:0]               ddr3_dqs_n_fpga;
  wire [CK_WIDTH-1:0]                ddr3_ck_p_fpga;
  wire [CK_WIDTH-1:0]                ddr3_ck_n_fpga;

  wire [DQ_WIDTH-1:0]                ddr3_dq_sdram;
  reg [ROW_WIDTH-1:0]                ddr3_addr_sdram;
  reg [BANK_WIDTH-1:0]               ddr3_ba_sdram;
  reg                                ddr3_ras_n_sdram;
  reg                                ddr3_cas_n_sdram;
  reg                                ddr3_we_n_sdram;
  reg [(CS_WIDTH*nCS_PER_RANK)-1:0]  ddr3_cs_n_sdram;
  reg [(CS_WIDTH*nCS_PER_RANK)-1:0]  ddr3_odt_sdram;
  reg [CKE_WIDTH-1:0]                ddr3_cke_sdram;
  wire [DM_WIDTH-1:0]                ddr3_dm_sdram;
  wire [DQS_WIDTH-1:0]               ddr3_dqs_p_sdram;
  wire [DQS_WIDTH-1:0]               ddr3_dqs_n_sdram;
  reg [CK_WIDTH-1:0]                 ddr3_ck_p_sdram;
  reg [CK_WIDTH-1:0]                 ddr3_ck_n_sdram;

  reg [ROW_WIDTH-1:0]               ddr3_addr_r;
  reg [BANK_WIDTH-1:0]              ddr3_ba_r;
  reg                               ddr3_ras_n_r;
  reg                               ddr3_cas_n_r;
  reg                               ddr3_we_n_r;
  reg [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_cs_n_r;
  reg [(CS_WIDTH*nCS_PER_RANK)-1:0] ddr3_odt_r;
  reg [CKE_WIDTH-1:0]               ddr3_cke_r;


    reg[7:0] switch;
    reg trigger_activate;
```

```
    reg sys1_fill_wr_exe2;
    reg sys1_fill_rd_exe2;

    reg sys2_fill_wr_exe2;
    reg sys2_fill_rd_exe2;

    initial begin
          sys_clk   = 1'b0;
          clk_ref   = 1'b1;
          sys_rst_n = 1'b1;



    end

reg[7:0] s;
reg[7:0] s2;

    initial begin
          sys1_fill_rd_exe2 = 0;
          sys1_fill_wr_exe2 = 0;

          sys2_fill_rd_exe2 = 0;
          sys2_fill_wr_exe2 = 0;


    wait(phy_init_done)
          #(3560)

                switch <= 8'b00001111;
                trigger_activate <= 1;
          #(5000*1)
                trigger_activate<=0;

          #(5000*100)
          sys1_fill_wr_exe2 <= 1;
          #(5000*1)
          sys1_fill_wr_exe2 <= 0;

          #(5000*130*6)
                sys1_fill_rd_exe2 <= 1;
          #(5000*1)
                sys1_fill_rd_exe2 <= 0;

end

assign sys_rst = RST_ACT_LOW ? sys_rst_n : ~sys_rst_n;

// Generate system clock = twice rate of CLK
```

```
  always
    sys_clk = #(CLK_PERIOD/2.0) ~sys_clk;

  // Generate IDELAYCTRL reference clock (200MHz)
  always
    clk_ref = #REFCLK_PERIOD ~clk_ref;

  assign sys_clk_p = sys_clk;
  assign sys_clk_n = ~sys_clk;

  assign clk_ref_p = clk_ref;
  assign clk_ref_n = ~clk_ref;



//*****************************************************************************/
/

  always @( * ) begin
    ddr3_ck_p_sdram   <=  #(TPROP_PCB_CTRL) ddr3_ck_p_fpga;
    ddr3_ck_n_sdram   <=  #(TPROP_PCB_CTRL) ddr3_ck_n_fpga;
    ddr3_addr_sdram   <=  #(TPROP_PCB_CTRL) ddr3_addr_fpga;
    ddr3_ba_sdram     <=  #(TPROP_PCB_CTRL) ddr3_ba_fpga;
    ddr3_ras_n_sdram  <=  #(TPROP_PCB_CTRL) ddr3_ras_n_fpga;
    ddr3_cas_n_sdram  <=  #(TPROP_PCB_CTRL) ddr3_cas_n_fpga;
    ddr3_we_n_sdram   <=  #(TPROP_PCB_CTRL) ddr3_we_n_fpga;
    ddr3_cs_n_sdram   <=  #(TPROP_PCB_CTRL) ddr3_cs_n_fpga;
    ddr3_cke_sdram    <=  #(TPROP_PCB_CTRL) ddr3_cke_fpga;
    ddr3_odt_sdram    <=  #(TPROP_PCB_CTRL) ddr3_odt_fpga;
    ddr3_dm_sdram_tmp <=  #(TPROP_PCB_DATA) ddr3_dm_fpga;//DM signal
generation
  end

  assign ddr3_dm_sdram = ddr3_dm_sdram_tmp;

// Controlling the bi-directional BUS
  genvar dqwd;
  generate
    for (dqwd = 1;dqwd < DQ_WIDTH;dqwd = dqwd+1) begin : dq_delay
      WireDelay #
       (
        .Delay_g  (TPROP_PCB_DATA),
        .Delay_rd (TPROP_PCB_DATA_RD),
      .ERR_INSERT ("OFF")
       )
      u_delay_dq
       (
        .A     (ddr3_dq_fpga[dqwd]),
        .B     (ddr3_dq_sdram[dqwd]),
        .reset (sys_rst_n),
```

```
    .phy_init_done (phy_init_done)
     );
   end
    WireDelay #
     (
      .Delay_g  (TPROP_PCB_DATA),
      .Delay_rd (TPROP_PCB_DATA_RD),
    .ERR_INSERT (ECC)
     )
    u_delay_dq_0
     (
      .A     (ddr3_dq_fpga[0]),
      .B     (ddr3_dq_sdram[0]),
      .reset (sys_rst_n),
    .phy_init_done (phy_init_done)
     );

endgenerate

genvar dqswd;
generate
  for (dqswd = 0;dqswd < DQS_WIDTH;dqswd = dqswd+1) begin : dqs_delay
    WireDelay #
     (
      .Delay_g  (TPROP_DQS),
      .Delay_rd (TPROP_DQS_RD),
    .ERR_INSERT ("OFF")
     )
    u_delay_dqs_p
     (
      .A     (ddr3_dqs_p_fpga[dqswd]),
      .B     (ddr3_dqs_p_sdram[dqswd]),
      .reset (sys_rst_n),
    .phy_init_done (phy_init_done)
     );

    WireDelay #
     (
      .Delay_g  (TPROP_DQS),
      .Delay_rd (TPROP_DQS_RD),
    .ERR_INSERT ("OFF")
     )
    u_delay_dqs_n
     (
      .A     (ddr3_dqs_n_fpga[dqswd]),
      .B     (ddr3_dqs_n_sdram[dqswd]),
      .reset (sys_rst_n),
    .phy_init_done (phy_init_done)
     );
  end
```

```
  endgenerate
//  assign sda = 1'b1;
//  assign scl = 1'b1;

  example_top #
    (
     .nCK_PER_CLK              (nCK_PER_CLK),
     .tCK                      (tCK),
     .RST_ACT_LOW              (RST_ACT_LOW),
     .REFCLK_FREQ              (REFCLK_FREQ),
     .IODELAY_GRP              (IODELAY_GRP),
     .INPUT_CLK_TYPE           (INPUT_CLK_TYPE),
     .BANK_WIDTH               (BANK_WIDTH),
     .CK_WIDTH                 (CK_WIDTH),
     .CKE_WIDTH                (CKE_WIDTH),
     .COL_WIDTH                (COL_WIDTH),
     .nCS_PER_RANK             (nCS_PER_RANK),
     .DQ_WIDTH                 (DQ_WIDTH),
     .DM_WIDTH                 (DM_WIDTH),
     .DQS_CNT_WIDTH            (DQS_CNT_WIDTH),
     .DQS_WIDTH                (DQS_WIDTH),
     .ROW_WIDTH                (ROW_WIDTH),
     .RANK_WIDTH               (RANK_WIDTH),
     .CS_WIDTH                 (CS_WIDTH),
     .BURST_MODE               (BURST_MODE),
     .BM_CNT_WIDTH             (BM_CNT_WIDTH),
     .CLKFBOUT_MULT_F          (CLKFBOUT_MULT_F),
     .DIVCLK_DIVIDE            (DIVCLK_DIVIDE),
     .CLKOUT_DIVIDE            (CLKOUT_DIVIDE),
     .OUTPUT_DRV               (OUTPUT_DRV),
     .REG_CTRL                 (REG_CTRL),
     .RTT_NOM                  (RTT_NOM),
     .RTT_WR                   (RTT_WR),
     .SIM_BYPASS_INIT_CAL      (SIM_BYPASS_INIT_CAL),
     .DEBUG_PORT               (DEBUG_PORT),
     .tPRDI                    (tPRDI),
     .tREFI                    (tREFI),
     .tZQI                     (tZQI),
     .ADDR_CMD_MODE            (ADDR_CMD_MODE),
     .ORDERING                 (ORDERING),
     .STARVE_LIMIT             (STARVE_LIMIT),
     .ADDR_WIDTH               (ADDR_WIDTH),
     .ECC                      (ECC),
     .ECC_TEST                 (ECC_TEST),
     .TCQ                      (TCQ),
     .EYE_TEST                 (EYE_TEST),
     .DATA_PATTERN             (DATA_PATTERN),
     .CMD_PATTERN              (CMD_PATTERN),
     .BEGIN_ADDRESS            (BEGIN_ADDRESS),
     .END_ADDRESS              (END_ADDRESS),
```

```
    .PRBS_EADDR_MASK_POS       (PRBS_EADDR_MASK_POS),
    .PRBS_SADDR_MASK_POS       (PRBS_SADDR_MASK_POS),
    .SEL_VICTIM_LINE           (SEL_VICTIM_LINE)
    )
   u_ip_top
     (
                 .clk_ref_p            (sys_clk_p),
                 .clk_ref_n            (sys_clk_n),
                 .sys_rst              (sys_rst),
                 .ddr3_ck_p            (ddr3_ck_p_fpga),
                 .ddr3_ck_n            (ddr3_ck_n_fpga),

                 .ddr3_addr            (ddr3_addr_fpga),
                 .ddr3_ba              (ddr3_ba_fpga),
                 .ddr3_ras_n           (ddr3_ras_n_fpga),
                 .ddr3_cas_n           (ddr3_cas_n_fpga),

                 .ddr3_we_n            (ddr3_we_n_fpga),
                 .ddr3_cs_n            (ddr3_cs_n_fpga),
                 .ddr3_cke             (ddr3_cke_fpga),
                 .ddr3_odt             (ddr3_odt_fpga),
                 .ddr3_reset_n         (ddr3_reset_n),
                 .ddr3_dm              (ddr3_dm_fpga),
                 .ddr3_dq              (ddr3_dq_fpga),
                 .ddr3_dqs_p           (ddr3_dqs_p_fpga),
                 .ddr3_dqs_n           (ddr3_dqs_n_fpga),


                 .phy_init_done        (phy_init_done),

                 .sys1_fill_wr_exe2(sys1_fill_wr_exe2),
                 .sys1_fill_rd_exe2(sys1_fill_rd_exe2),

                 .sys2_fill_wr_exe2(sys2_fill_wr_exe2),
                 .sys2_fill_rd_exe2(sys2_fill_rd_exe2),

                 .trigger_activate       (trigger_activate),
                 .switch                              (switch)

      );

   // Extra one clock pipelining for RDIMM address and
   // control signals is implemented here (Implemented external to memory
model)
   always @( posedge ddr3_ck_p_sdram[0] ) begin
     if ( ddr3_reset_n == 1'b0 ) begin
       ddr3_ras_n_r <= 1'b1;
       ddr3_cas_n_r <= 1'b1;
       ddr3_we_n_r  <= 1'b1;
       ddr3_cs_n_r  <= {(CS_WIDTH*nCS_PER_RANK){1'b1}};
```

```
      ddr3_odt_r   <= 1'b0;
    end
    else begin
      ddr3_addr_r  <= #(CLK_PERIOD/2) ddr3_addr_sdram;
      ddr3_ba_r    <= #(CLK_PERIOD/2) ddr3_ba_sdram;
      ddr3_ras_n_r <= #(CLK_PERIOD/2) ddr3_ras_n_sdram;
      ddr3_cas_n_r <= #(CLK_PERIOD/2) ddr3_cas_n_sdram;
      ddr3_we_n_r  <= #(CLK_PERIOD/2) ddr3_we_n_sdram;
      ddr3_cs_n_r  <= #(CLK_PERIOD/2) ddr3_cs_n_sdram;
      ddr3_odt_r   <= #(CLK_PERIOD/2) ddr3_odt_sdram;
    end
  end

  // to avoid tIS violations on CKE when reset is deasserted
  always @( posedge ddr3_ck_n_sdram[0] )
    if ( ddr3_reset_n == 1'b0 )
      ddr3_cke_r <= 1'b0;
    else
      ddr3_cke_r <= #(CLK_PERIOD) ddr3_cke_sdram;



//***************************************************************************
  // Instantiate memories

//***************************************************************************

  genvar r,i,dqs_x;
  generate
    if(DRAM_DEVICE == "COMP") begin : comp_inst
      for (r = 0; r < CS_WIDTH; r = r+1) begin: mem_rnk
        if(MEMORY_WIDTH == 16) begin: mem_16
          if(DQ_WIDTH/16) begin: gen_mem
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
                 .ck      (ddr3_ck_p_sdram),
                 .ck_n    (ddr3_ck_n_sdram),
                 .cke     (ddr3_cke_sdram[r]),
                 .cs_n    (ddr3_cs_n_sdram[r]),
                 .ras_n   (ddr3_ras_n_sdram),
                 .cas_n   (ddr3_cas_n_sdram),
                 .we_n    (ddr3_we_n_sdram),
                 .dm_tdqs (ddr3_dm_sdram[(2*(i+1)-1):(2*i)]),
                 .ba      (ddr3_ba_sdram),
                 .addr    (ddr3_addr_sdram),
                 .dq      (ddr3_dq_sdram[16*(i+1)-1:16*(i)]),
                 .dqs     (ddr3_dqs_p_sdram[(2*(i+1)-1):(2*i)]),
```

```
                .dqs_n   (ddr3_dqs_n_sdram[(2*(i+1)-1):(2*i)]),
                .tdqs_n  (),
                .odt     (ddr3_odt_sdram[r])
                );
            end
          end
          if (DQ_WIDTH%16) begin: gen_mem_extrabits
            ddr3_model u_comp_ddr3
              (
               .rst_n   (ddr3_reset_n),
               .ck      (ddr3_ck_p_sdram),
               .ck_n    (ddr3_ck_n_sdram),
               .cke     (ddr3_cke_sdram[r]),
               .cs_n    (ddr3_cs_n_sdram[r]),
               .ras_n   (ddr3_ras_n_sdram),
               .cas_n   (ddr3_cas_n_sdram),
               .we_n    (ddr3_we_n_sdram),
               .dm_tdqs ({ddr3_dm_sdram[DM_WIDTH-1],ddr3_dm_sdram[DM_WIDTH-
1]}),
               .ba      (ddr3_ba_sdram),
               .addr    (ddr3_addr_sdram),
               .dq      ({ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)],
                          ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)]}),
               .dqs     ({ddr3_dqs_p_sdram[DQS_WIDTH-1],
                          ddr3_dqs_p_sdram[DQS_WIDTH-1]}),
               .dqs_n   ({ddr3_dqs_n_sdram[DQS_WIDTH-1],
                          ddr3_dqs_n_sdram[DQS_WIDTH-1]}),
               .tdqs_n  (),
               .odt     (ddr3_odt_sdram[r])
               );
            end
          end
          else if((MEMORY_WIDTH == 8) || (MEMORY_WIDTH == 4)) begin: mem_8_4
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
                 .ck      (ddr3_ck_p_sdram),
                 .ck_n    (ddr3_ck_n_sdram),
                 .cke     (ddr3_cke_sdram[r]),
                 .cs_n    (ddr3_cs_n_sdram[r]),
                 .ras_n   (ddr3_ras_n_sdram),
                 .cas_n   (ddr3_cas_n_sdram),
                 .we_n    (ddr3_we_n_sdram),
                 .dm_tdqs (ddr3_dm_sdram[i]),
                 .ba      (ddr3_ba_sdram),
                 .addr    (ddr3_addr_sdram),
                 .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
                 .dqs     (ddr3_dqs_p_sdram[i]),
```

```
                .dqs_n   (ddr3_dqs_n_sdram[i]),
                .tdqs_n  (),
                .odt     (ddr3_odt_sdram[r])
                );
          end
        end
      end
    end
    else if(DRAM_DEVICE == "RDIMM") begin: rdimm_inst
      for (r = 0; r < CS_WIDTH; r = r+1) begin: mem_rnk
        if((MEMORY_WIDTH == 8) || (MEMORY_WIDTH == 4)) begin: mem_8_4
          for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
            ddr3_model u_comp_ddr3
              (
                .rst_n   (ddr3_reset_n),
                .ck      (ddr3_ck_p_sdram[(i*MEMORY_WIDTH)/72]),
                .ck_n    (ddr3_ck_n_sdram[(i*MEMORY_WIDTH)/72]),
                .cke     (ddr3_cke_r[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                .cs_n
(ddr3_cs_n_r[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                .ras_n   (ddr3_ras_n_r),
                .cas_n   (ddr3_cas_n_r),
                .we_n    (ddr3_we_n_r),
                .dm_tdqs (ddr3_dm_sdram[i]),
                .ba      (ddr3_ba_r),
                .addr    (ddr3_addr_r),
                .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
                .dqs     (ddr3_dqs_p_sdram[i]),
                .dqs_n   (ddr3_dqs_n_sdram[i]),
                .tdqs_n  (),
                .odt     (ddr3_odt_r[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)])
                );
          end
        end
      end
    end
    else if(DRAM_DEVICE == "UDIMM") begin: udimm_inst
      for (r = 0; r < CS_WIDTH; r = r+1) begin: mem_rnk
        if(MEMORY_WIDTH == 16) begin: mem_16
          if(DQ_WIDTH/16) begin: gen_mem
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
                 .ck      (ddr3_ck_p_sdram[(i*MEMORY_WIDTH)/72]),
                 .ck_n    (ddr3_ck_n_sdram[(i*MEMORY_WIDTH)/72]),
                 .cke
(ddr3_cke_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
```

```
                .cs_n
(ddr3_cs_n_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                .ras_n   (ddr3_ras_n_sdram),
                .cas_n   (ddr3_cas_n_sdram),
                .we_n    (ddr3_we_n_sdram),
                .dm_tdqs (ddr3_dm_sdram[(2*(i+1)-1):(2*i)]),
                .ba      (ddr3_ba_sdram),
                .addr    (ddr3_addr_sdram),
                .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
                .dqs     (ddr3_dqs_p_sdram[(2*(i+1)-1):(2*i)]),
                .dqs_n   (ddr3_dqs_n_sdram[(2*(i+1)-1):(2*i)]),
                .tdqs_n  (),
                .odt
(ddr3_odt_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)])
                );
            end
          end
          if (DQ_WIDTH%16) begin: gen_mem_extrabits
            ddr3_model u_comp_ddr3
              (
               .rst_n   (ddr3_reset_n),
               .ck      (ddr3_ck_p_sdram[(DQ_WIDTH-1)/72]),
               .ck_n    (ddr3_ck_n_sdram[(DQ_WIDTH-1)/72]),
               .cke     (ddr3_cke_sdram[((DQ_WIDTH-1)/72)+(nCS_PER_RANK*r)]),
               .cs_n    (ddr3_cs_n_sdram[((DQ_WIDTH-
1)/72)+(nCS_PER_RANK*r)]),
               .ras_n   (ddr3_ras_n_sdram),
               .cas_n   (ddr3_cas_n_sdram),
               .we_n    (ddr3_we_n_sdram),
               .dm_tdqs ({ddr3_dm_sdram[DM_WIDTH-1],ddr3_dm_sdram[DM_WIDTH-
1]}),
               .ba      (ddr3_ba_sdram),
               .addr    (ddr3_addr_sdram),
               .dq      ({ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)],
                          ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)]}),
               .dqs     ({ddr3_dqs_p_sdram[DQS_WIDTH-1],
                          ddr3_dqs_p_sdram[DQS_WIDTH-1]}),
               .dqs_n   ({ddr3_dqs_n_sdram[DQS_WIDTH-1],
                          ddr3_dqs_n_sdram[DQS_WIDTH-1]}),
               .tdqs_n  (),
               .odt     (ddr3_odt_sdram[((DQ_WIDTH-1)/72)+(nCS_PER_RANK*r)])
               );
            end
          end
          else if((MEMORY_WIDTH == 8) || (MEMORY_WIDTH == 4)) begin: mem_8_4
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
```

```
              .ck      (ddr3_ck_p_sdram[(i*MEMORY_WIDTH)/72]),
              .ck_n    (ddr3_ck_n_sdram[(i*MEMORY_WIDTH)/72]),
              .cke
(ddr3_cke_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
              .cs_n
(ddr3_cs_n_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
              .ras_n   (ddr3_ras_n_sdram),
              .cas_n   (ddr3_cas_n_sdram),
              .we_n    (ddr3_we_n_sdram),
              .dm_tdqs (ddr3_dm_sdram[i]),
              .ba      (ddr3_ba_sdram),
              .addr    (ddr3_addr_sdram),
              .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
              .dqs     (ddr3_dqs_p_sdram[i]),
              .dqs_n   (ddr3_dqs_n_sdram[i]),
              .tdqs_n  (),
              .odt
(ddr3_odt_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)])
              );
          end
        end
      end
    end
    else if(DRAM_DEVICE == "SODIMM") begin: sodimm_inst
      for (r = 0; r < CS_WIDTH; r = r+1) begin: mem_rnk
        if(MEMORY_WIDTH == 16) begin: mem_16
          if(DQ_WIDTH/16) begin: gen_mem
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
                 .ck      (ddr3_ck_p_sdram[(i*MEMORY_WIDTH)/72]),
                 .ck_n    (ddr3_ck_n_sdram[(i*MEMORY_WIDTH)/72]),
                 .cke
(ddr3_cke_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                 .cs_n
(ddr3_cs_n_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                 .ras_n   (ddr3_ras_n_sdram),
                 .cas_n   (ddr3_cas_n_sdram),
                 .we_n    (ddr3_we_n_sdram),
                 .dm_tdqs (ddr3_dm_sdram[(2*(i+1)-1):(2*i)]),
                 .ba      (ddr3_ba_sdram),
                 .addr    (ddr3_addr_sdram),
                 .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
                 .dqs     (ddr3_dqs_p_sdram[(2*(i+1)-1):(2*i)]),
                 .dqs_n   (ddr3_dqs_n_sdram[(2*(i+1)-1):(2*i)]),
                 .tdqs_n  (),
```

```
                .odt
(ddr3_odt_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)])
                );
            end
          end
          if (DQ_WIDTH%16) begin: gen_mem_extrabits
            ddr3_model u_comp_ddr3
              (
               .rst_n   (ddr3_reset_n),
               .ck      (ddr3_ck_p_sdram[(DQ_WIDTH-1)/72]),
               .ck_n    (ddr3_ck_n_sdram[(DQ_WIDTH-1)/72]),
               .cke     (ddr3_cke_sdram[((DQ_WIDTH-1)/72)+(nCS_PER_RANK*r)]),
               .cs_n    (ddr3_cs_n_sdram[((DQ_WIDTH-
1)/72)+(nCS_PER_RANK*r)]),
               .ras_n   (ddr3_ras_n_sdram),
               .cas_n   (ddr3_cas_n_sdram),
               .we_n    (ddr3_we_n_sdram),
               .dm_tdqs ({ddr3_dm_sdram[DM_WIDTH-1],ddr3_dm_sdram[DM_WIDTH-
1]}),
               .ba      (ddr3_ba_sdram),
               .addr    (ddr3_addr_sdram),
               .dq      ({ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)],
                          ddr3_dq_sdram[DQ_WIDTH-1:(DQ_WIDTH-8)]}),
               .dqs     ({ddr3_dqs_p_sdram[DQS_WIDTH-1],
                          ddr3_dqs_p_sdram[DQS_WIDTH-1]}),
               .dqs_n   ({ddr3_dqs_n_sdram[DQS_WIDTH-1],
                          ddr3_dqs_n_sdram[DQS_WIDTH-1]}),
               .tdqs_n  (),
               .odt     (ddr3_odt_sdram[((DQ_WIDTH-1)/72)+(nCS_PER_RANK*r)])
               );
            end
          end
          if((MEMORY_WIDTH == 8) || (MEMORY_WIDTH == 4)) begin: mem_8_4
            for (i = 0; i < NUM_COMP; i = i + 1) begin: gen_mem
              ddr3_model u_comp_ddr3
                (
                 .rst_n   (ddr3_reset_n),
                 .ck      (ddr3_ck_p_sdram[(i*MEMORY_WIDTH)/72]),
                 .ck_n    (ddr3_ck_n_sdram[(i*MEMORY_WIDTH)/72]),
                 .cke
(ddr3_cke_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                 .cs_n
(ddr3_cs_n_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)]),
                 .ras_n   (ddr3_ras_n_sdram),
                 .cas_n   (ddr3_cas_n_sdram),
                 .we_n    (ddr3_we_n_sdram),
                 .dm_tdqs (ddr3_dm_sdram[i]),
                 .ba      (ddr3_ba_sdram),
                 .addr    (ddr3_addr_sdram),
```

```
                .dq      (ddr3_dq_sdram[MEMORY_WIDTH*(i+1)-
1:MEMORY_WIDTH*(i)]),
                .dqs     (ddr3_dqs_p_sdram[i]),
                .dqs_n   (ddr3_dqs_n_sdram[i]),
                .tdqs_n  (),
                .odt
(ddr3_odt_sdram[((i*MEMORY_WIDTH)/72)+(nCS_PER_RANK*r)])
                );
           end
         end
       end
     end
  endgenerate



//***************************************************************************
  // Reporting the test case status

//***************************************************************************
  initial
  begin : Logging
  fork
    begin : calibration_done
      wait (phy_init_done);
      $display("Calibration Done");
      #50000000;
      //if (!error) begin
      //  $display("TEST PASSED");
      //end
      //else begin
      //  $display("TEST FAILED: DATA ERROR");
      //end
      disable calib_not_done;
       $finish;
    end

    begin : calib_not_done
      #1000000000;
      if (!phy_init_done) begin
        $display("TEST FAILED: INITIALIZATION DID NOT COMPLETE");
      end
      disable calibration_done;
       $finish;
    end
  join
 end
```

```
endmodule
```