Oracle eMMC Hardware Controller

Senior Design Final Report

Jack Baskin School of Engineering

University of California, Santa Cruz

Winter 2015 - Spring 2015

June 13th, 2015

John Dilger, Travis Johnson, Hunter Nichols, Sergy Pretetsky

Table of Contents

Abstract	2
Background	2
Approach	5
Methods/Implementation	6
Software/Hardware Interface	6
Controller Implementation	9
Results	21
Challenges	23
Ongoing Objectives	25
Acknowledgements	26

Abstract

In computing, bandwidth and latency to storage often bottlenecks the overall performance of the system. eMMC is a flash-based storage device designed to offer good performance while hiding the complexities of flash memory management. Interfacing with eMMC requires precise timing and clock management, requiring a controller to translate requests into the eMMC specification. Our project involved creating this controller in synthesizable verilog and targeting a Xilinx ZC706 evaluation board for testing.
Generally speaking, hardware should be minimized and simplified when possible to minimize power and die area usage. However, the eMMC device could be caught in many different error states that, if handled by hardware, would greatly increase the complexity of the controller. Furthermore, the initialization procedure would increase the complexity of the controller even more.

These factors led us to the philosophy that the eMMC controller should only perform the critical timing management and clock management that is required of it. All eMMC state management should be done by a software controller.

On the eMMC Device with which we tested, we were able to get reads of up to 20 MB/s and writes of up to 22 MB/s during sequential transfers of 1 MB.

Background

eMMC stands for embedded MultiMediaCard. It is a flash-based memory standard maintained by JEDEC. The eMMC interface between the host controller and the device is shown in Figure 1. It consists of a clock driven from the controller, a bidirectional command wire, and a bi-directional 8-bit data bus. This interface can be shared by many eMMC devices connected to the host controller.

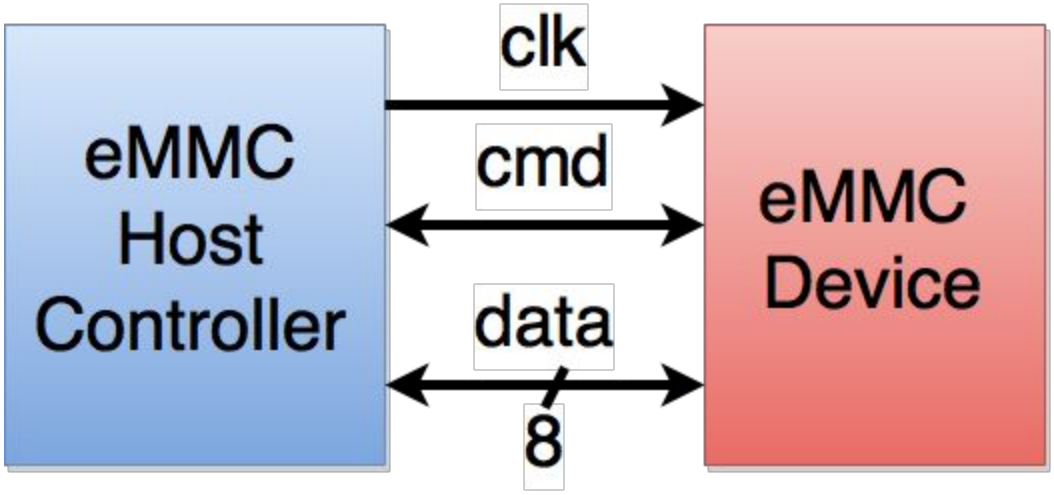


Figure 1: eMMC interface

eMMC devices act as slaves that do not transmit data on the cmd or data wires unless responding to the host controller. Most eMMC commands follow a command-response based structure. This means that the eMMC host will send a command on the cmd wire and expect a response on the cmd wire from the eMMC device.

Commands that involve data operations will trigger reads or writes to occur on the data bus. This flow can be seen in Figure 2. To ensure data integrity, both reads and writes use CRC (cyclic redundancy checks). During a read the CRC is sent back along with the data to the host controller so that the CRC can be checked against the data to ensure that no errors have occurred. For writes a CRC is sent along with the data to the eMMC device and the eMMC device responds on the data bus verifying that the data it received did not have any errors.

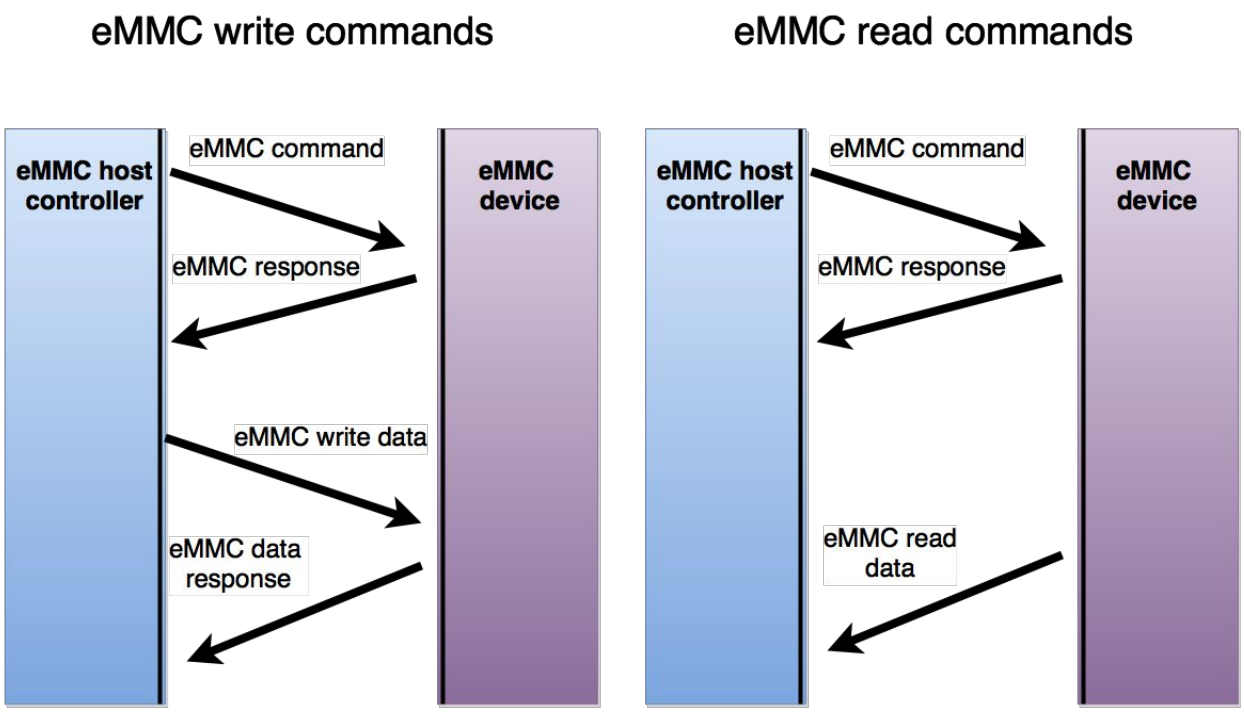


Figure 2: Command-Response based eMMC protocol

Each eMMC device can be found in a particular state at any given moment. The full state diagram for eMMC can be found in JEDEC's eMMC documentation. A simplified state diagram is shown in Figure 3. When powered on, every eMMC device begins in an idle state and must first complete an identification process before moving into transfer mode. While eMMC devices are in identification mode, they transmit on the cmd and data wires in open-drain mode. This is important because in many of the commands more than a single eMMC device may attempt to respond on the interface at the same time.

It is the host controller's job to check if all of the eMMC devices on the bus are ready. Once all of the devices are ready they must be identified individually and be assigned relative addresses by the controller.

Once an eMMC device has been assigned a relative address it switches to transfer mode. From this point on each eMMC device operates on the cmd and data wires in push-pull mode. Individual eMMC devices must be selected by their relative address before performing data operations. Only selected eMMC devices will attempt to respond. Two eMMC devices

should never attempt to communicate on the data or cmd wires at the same time once in transfer mode.

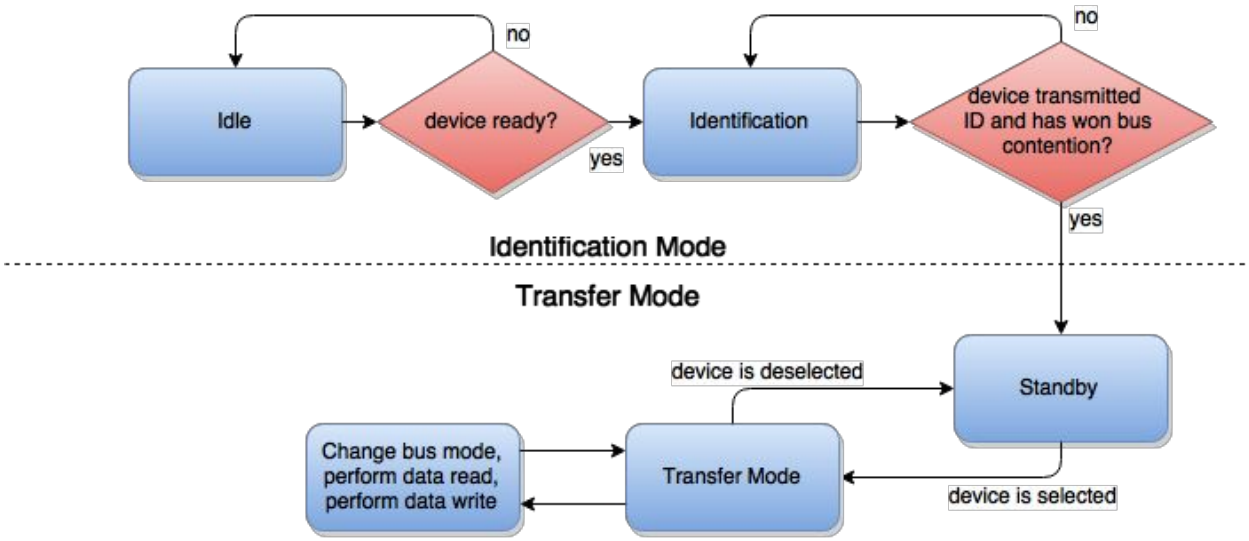


Figure 3: eMMC simplified state diagram.

Approach

Our target for this project was the Xilinx ZC706 evaluation board which uses the Zynq system on a chip. The Zynq chip includes two arm processor cores and a programmable hardware fabric allowing us to create a codesign with the hardware controller synthesized from Verilog and software driver programmed in C. The block diagram for this is shown in figure 4.

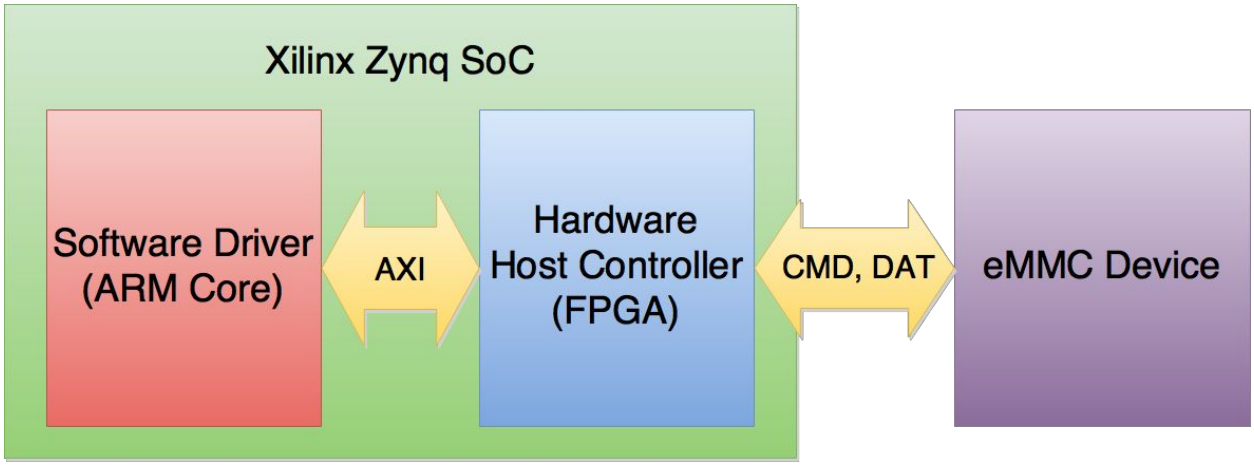


Figure 4: Block diagram of design using the Xilinx Zynq SoC

Originally, we began implementing an automatic eMMC hardware controller. The hardware controller would automatically manage all of the eMMC errors and keep track of the

state of the eMMC device. We would abstract as much of the inner workings as possible under this scheme. The eMMC hardware controller would initialize all of the eMMC devices, handle any sub-operations during reads and writes, and handle any errors during those operations.

It became clear that implementing this would not be trivial. We had two reasons for moving away from this design. First, we wouldn't have had enough time to properly implement and test an automatic hardware controller. Second, it didn't make sense to use up silicon area simply to abstract the implementation details away. eMMC initialization and error management are not performance critical tasks and do not need to be implemented in hardware. We decided it was reasonable to have the software driver perform any abstraction from the end user.

For these reasons, we created a new manual hardware controller. The job of the manual eMMC hardware controller was to enforce critical timing requirements and clock management.

Methods/Implementation

Software/Hardware Interface

We used the AXI protocol to communicate between our software driver and hardware controller. AXI stands for Advanced eXtendable Interface and is part of the AMBA standard from ARM. This method of interconnect, with the help of custom Xilinx Vivado AXI peripherals, allowed the software team to read and write directly to memory-mapped addresses to give information to the hardware controller. We did this by customizing an AXI peripheral generated by Vivado to interface with a native asynchronous FIFO. This made the FIFO AXI addressable and AXI clocked on one side and gave it a native interface on the other side with the controller clock. The diagram that shows the AXI peripheral and AXI side of the FIFO is shown in Figure 5 below.

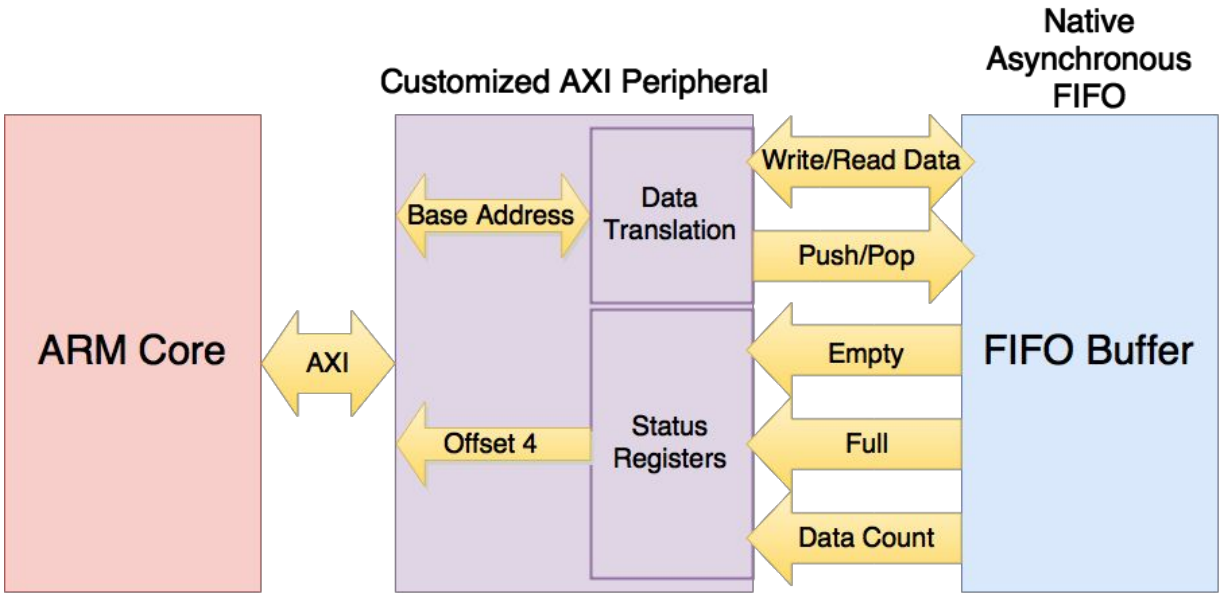


Figure 5: Diagram of AXI Peripheral

Data routing is done in the peripheral by forwarding the AXI data channel lines to the data bits of the FIFO when the controller sends data to the base address for the specific FIFO. Reading or writing also asserts the corresponding read or write enable if the FIFO is not empty or full respectively. If the FIFO for writes is full, the AXI transfer can be completed but it does not send data to the FIFO. Similarly, if the FIFO for reads is empty, it will return garbage data over AXI and will not assert the read enable. This is done because the AXI protocol can cause the software call to lock up until data is ready or the FIFO is no longer full. We implemented an additional register to enable a more robust flow control scheme. This register can be accessed from software by reading an offset of four from the base address of the FIFO. Reading this address gives a 32-bit status word. This word can be bit masked to get the desired signal. The signals are masked as following:

bit 0 - empty

bit 1 - full

bits 2-14 - data count

The data count is the amount of data currently in the FIFO. The software driver is expected to read this status register regularly to get the status information from the FIFO's that they are

interacting with. This allows complete flow control over the AXI bus at the cost of increased overhead caused by the extra AXI transactions necessary to check the status of the FIFOs.

The native FIFOs that we used were generated from the Xilinx FIFO generator. We determined that this was the easiest way to have reliable and customizable FIFOs for each desired function because we needed the FIFOs to be asynchronous and have variable depths and widths. We use seven FIFOs total: command/length, argument, response, response status, write data, read data, and data status. This interface is shown in figure 6.

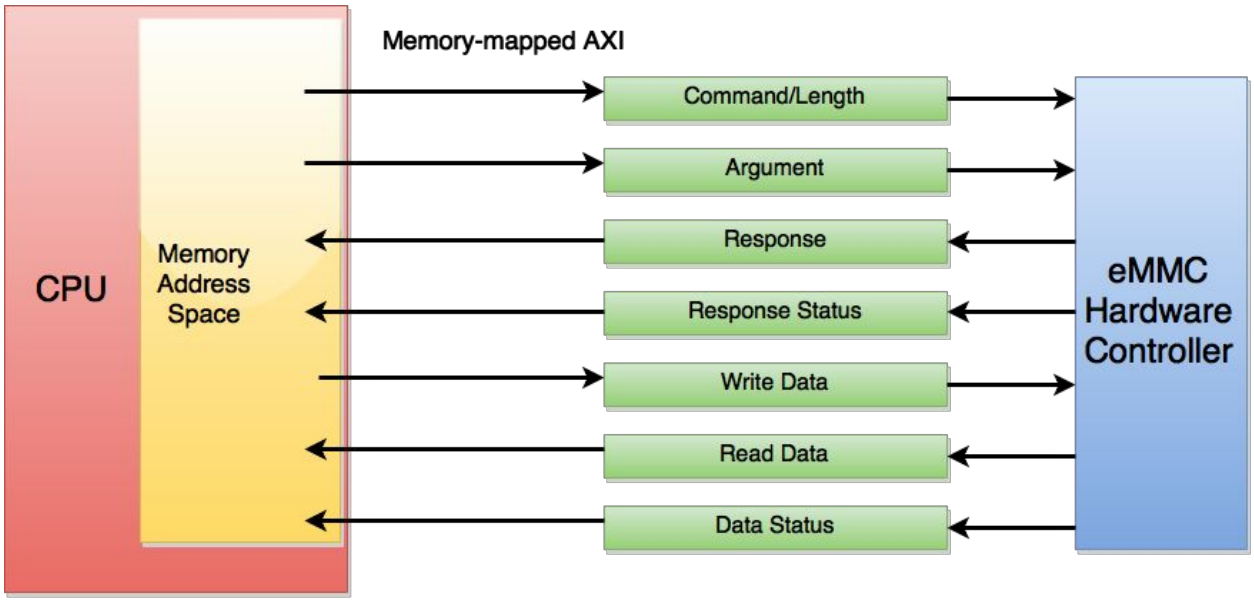


Figure 6: hardware-software interface

The command/length and argument FIFOs are used in the initial request by the software driver for the controller to issue a command to the device. They were combined into one 32 bit FIFO because the command is only 7 bits and the length is only 16 bits. The command section of this FIFO corresponds to an eMMC command as specified by JEDEC or a custom command that the driver wants to issue to the controller. The length is the length argument for read and write commands which specifies the number of blocks to read or write. This was included because it was much easier to explicitly give the length than to make the hardware more complex and extract the length from the argument. For commands that are not reads or writes, the length that is given is disregarded. The argument for the specified command is given in the argument

FIFO. When these two FIFOs are populated, and the controller is in the IDLE state, it can start interpreting the command.

The response and response status FIFOs are used to return the response that the controller receives from the device to the software driver. The response status is only 2 bits wide and corresponds to the status of the response with respect to a no-response timeout or CRC error. If there is a timeout, a 1 is returned, and if there is a CRC error a 2 is returned. Otherwise, a status of 0 signals a good response. Software must first check this register to see if the response is good, bad, or, in the case of a no-response timeout, not there.

The write data, read data, and data status FIFOs are used to transfer data. These have a large depth so that they can deal with large transfers to and from the eMMC device without stopping. The data status is used to give the status of a read or write operation. This status FIFO is used in the same way as the response status except it returns the status of the returned data. Each 512 bytes of the transfer includes a data status. This is because the block size and thus smallest transfer possible is 512 bytes and each block includes a CRC. By sending a status for each block, it allows software to realize that a block was sent incorrectly and request that specific block again.

Controller Implementation

Our hardware controller follows a manual approach which means it does not communicate with the connected eMMC device on its own, but instead it will always wait for a command from the software driver to transfer that command to the eMMC. The controller will look at each command and determine if additional functionality needs to be performed other than the command transfer. There are several defined paths our controller takes in order to properly transfer and execute an eMMC command. These paths are defined by the type of command that is to be issued:

1. Commands with No Response Expected.
2. Commands with Responses Expected.
3. Read Commands
4. Write Commands.

The path our controller takes to execute a command depends on the command's 6-bit index. Our controller extracts this 6-bit command index from the command FIFO that is part of the hardware-software interface. The 7th bit in the command index is to allow the software driver to send commands custom to the controller which set functionality within the controller. We will now explain the high level block diagram and how each block functions:

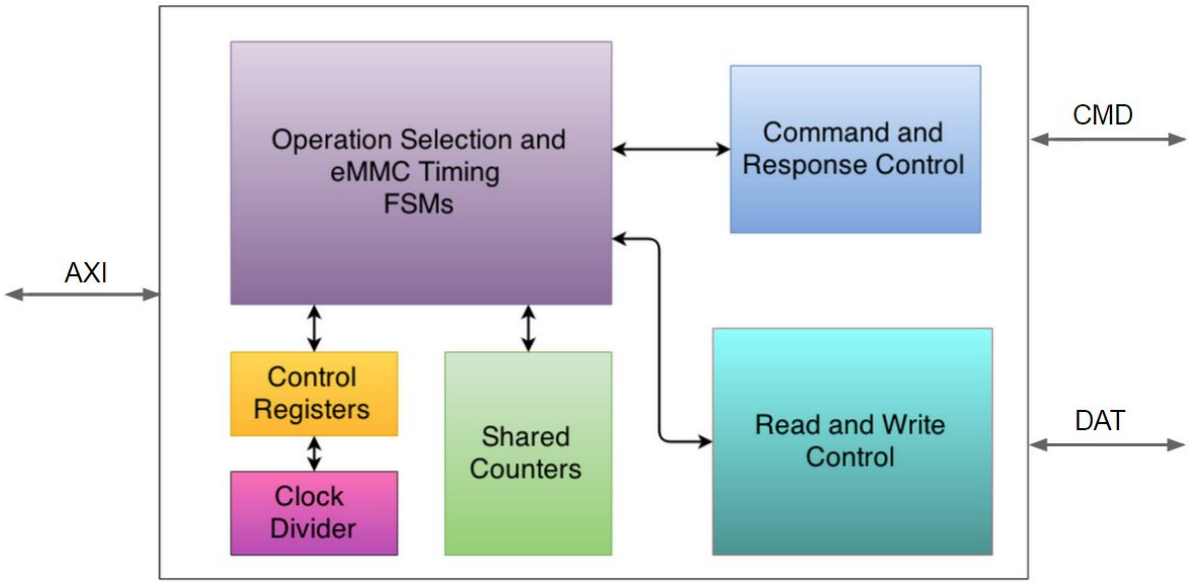


Figure 7: A high level block diagram of the hardware controller.

From Figure 7, the block Operation Selection and eMMC Timing FSMs is the central control of the controller. It will wait for an input command from the AXI Command FIFO. Once a command has been received, this block will start to interpret it to determine which path of functionality the controller will need to undergo to execute the command. After interpretation, this block will activate the other blocks in a specific order to execute the command. For example, if the input command from software is marked as a command with an expected response for the eMMC, the Operation Selection FSM will activate the Command and Response Control block. This block will then send the command to the eMMC and signal when it is done. The Operation Selection FSM will then activate the block again to look for a response from the eMMC. If a response is received, the 32-bit response is pushed to a Response AXI FIFO for the software driver. If no response is received, an error is written into a Response Status FIFO for the software driver to check. The Operation Selection and eMMC Timing block also contains FSMs to enforce timings according to the eMMC specification.

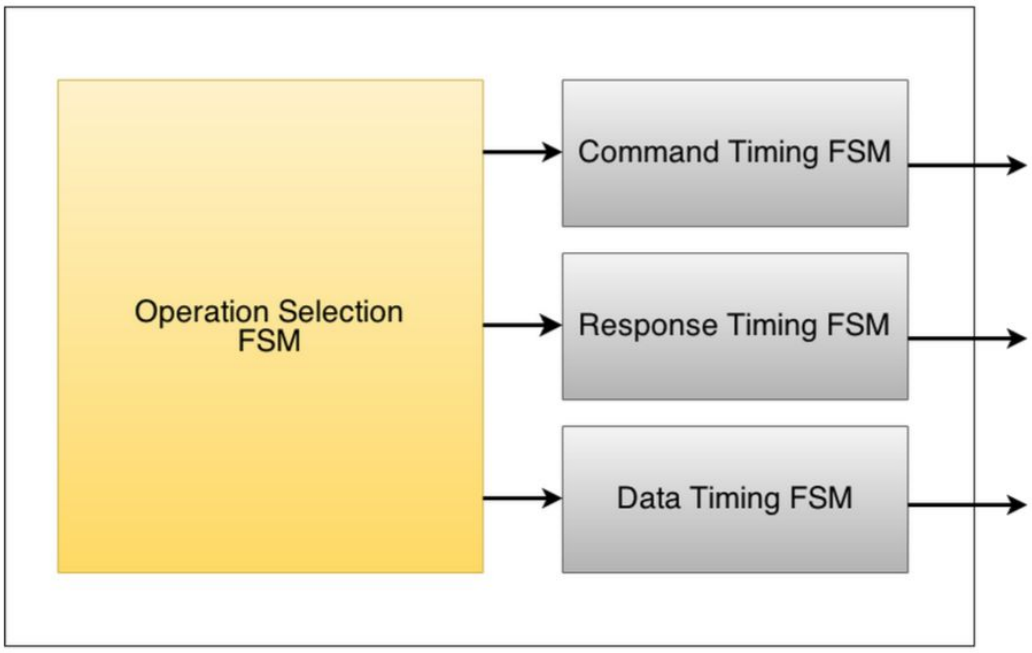


Figure 8: An inside look in the Operation selection and timing FSM block from Figure 7.

As mentioned, the Operation Selection FSM will determine the path of functionality. It does not directly activate the command, response, read, or write modules. Instead, it activates these timing FSMs which in turn activate their respective modules as seen in figure 8. The timing modules enforce eMMC timing based on the JEDEC specification:

1. The Command Timing FSM enforces that this controller is not sending commands too quickly to the eMMC. The specification requires at least 8 clock cycles between successive commands or commands after a response. This wait period is enforced by this FSM.
2. The Response Timing FSM checks for a response timeout in which the module never receives a response from the eMMC within a specified amount of time. In general, the FSM will wait 64 clock cycles before declaring that a response will never come in, and the controller should return an error to the software driver. Responses from eMMC CMD1 and CMD2 timeout after 5 clock cycles and are the only exception.
3. The Data Timing FSM is the largest and handles read and write timing. For write timing, it will ensure that our controller waits at least 2 clock cycles before data is sent to the eMMC based on the specification. This FSM will also check if multiple blocks of data are being written in this write command. The write module only writes a single block, and

this module continuously re-activate the write module until all blocks have been written based on an input length AXI FIFO.

4. The read timing in the Data Timing FSM in similar to the Response Timing FSM. After sending the read command and getting a response, this module will count a read timeout and will signal an timeout error if a block is not received from the eMMC within a specified amount of time which is dependent on each eMMC device. Additionally, this module will make sure that the controller receives the amount of blocks specified by the input length from the software driver.

Command Module:

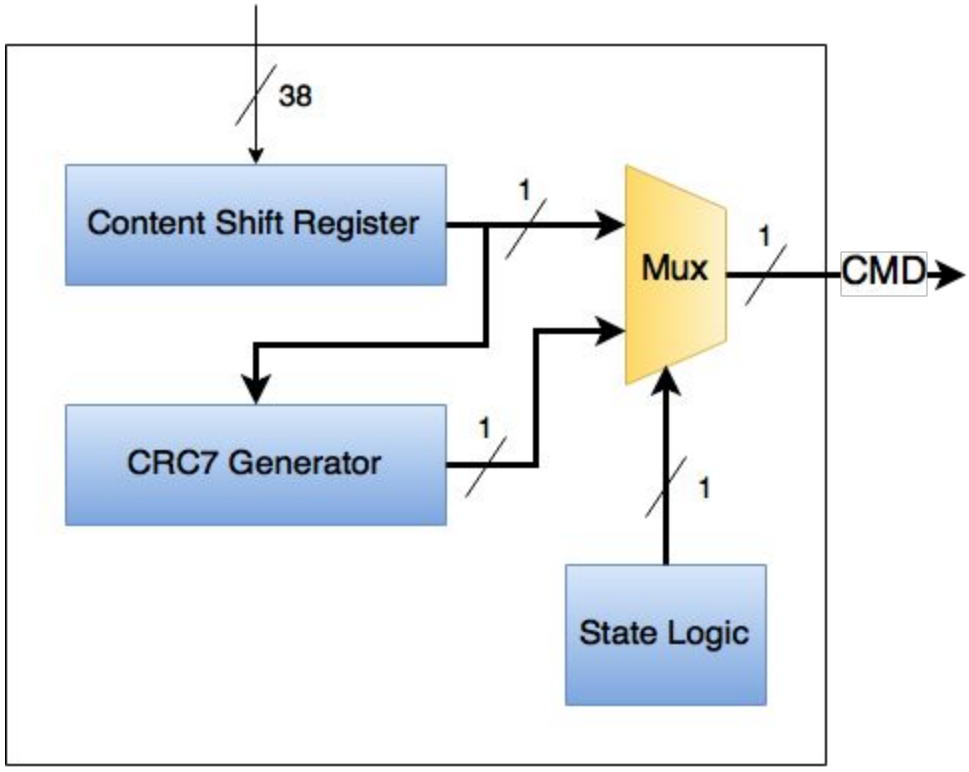


Figure 9: Implementation of the command module.

The command module takes in a 6-bit command index and 32-bit command argument to transfer to the eMMC. These are inputs are from the software driver via their respective AXI FIFOs. When the module is activated, these values are loaded into a shift register. The state logic will also make sure to send overhead bits according to the eMMC specification and then

force the shift register to start shifting data on the eMMC CMD bus. During this time, the data is also being shifted into a Cyclic Redundancy Check (CRC) Generator which is implemented similar to a linear feedback shift register. It generates an error code when all the data has been shifted into this module. After all the data has been shifted out, the state logic then chooses this error code to be sent on the CMD bus to conform to the specification.

The Response Module:

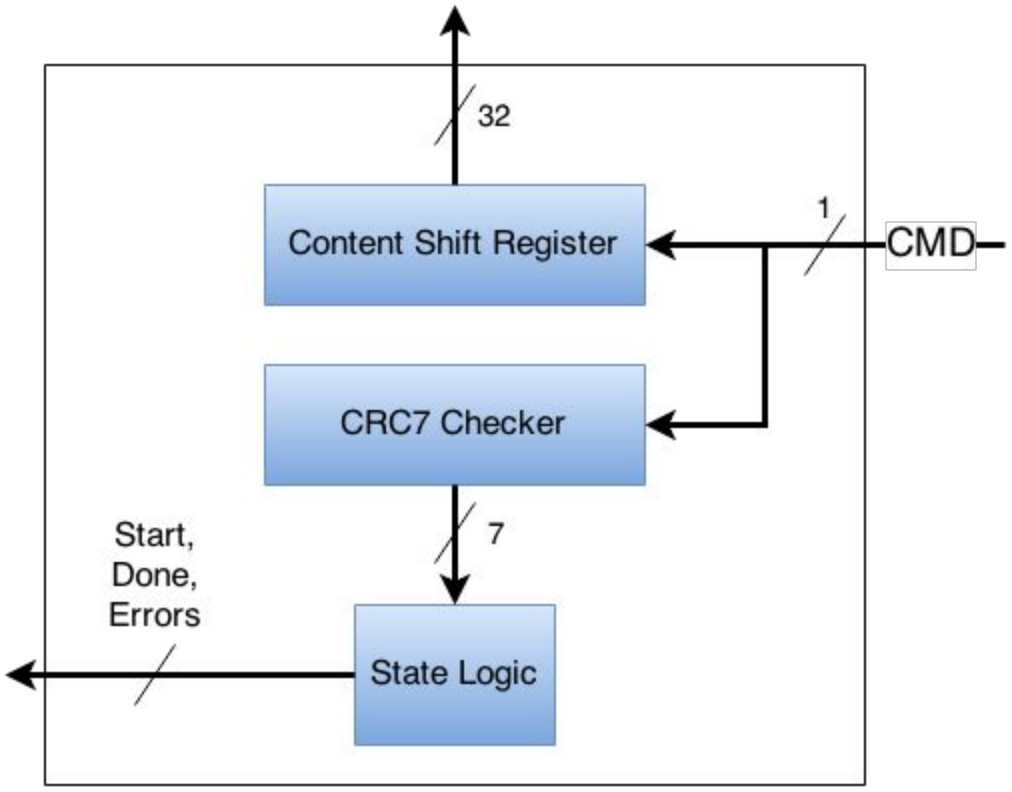


Figure 10: Implementation of response module.

The response module will shift in the data on the eMMC CMD bus into a shift register. Once activated, the state logic will signal when it receives a start bit. An eMMC response is 48 bits long, but 32 bits refer to the actual content of the response. This value will be pushed to an AXI response FIFO when it becomes ready in this module. The response is also shifted into a CRC module. If data is shifted into this module along with a CRC created from the data, we should

see the 7 bit output of the module equal to 0. Any other value would indicate that there was an error during its transmission and an error should be signaled through the response status FIFO.

The Write Module:

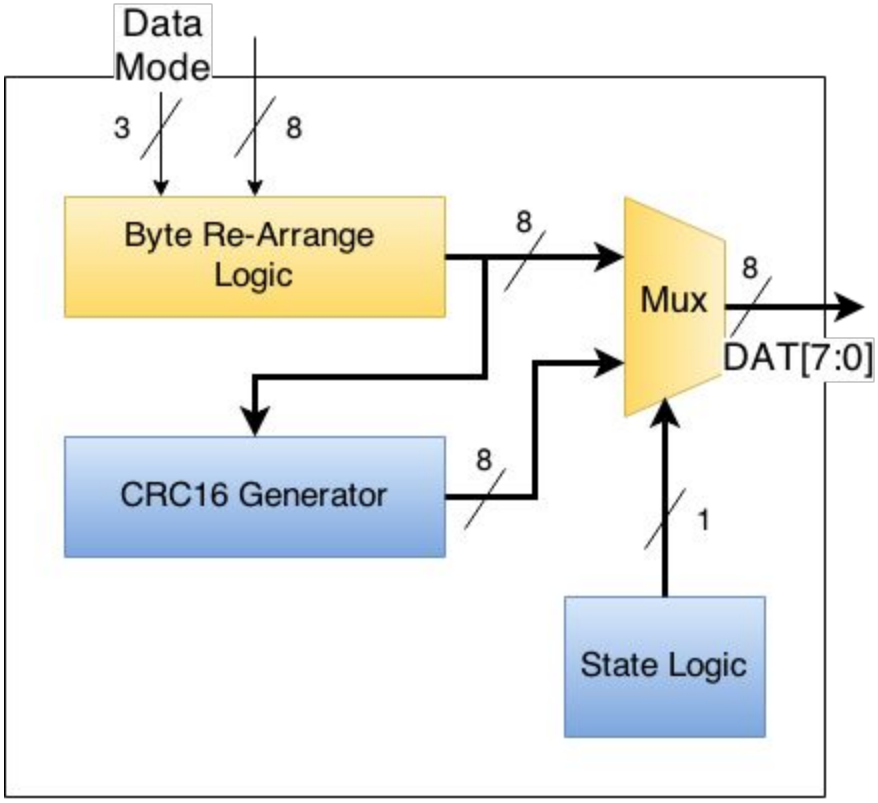


Figure 11: Implementation of the write module.

The Write Module has a similar implementation to the Command Module minus the use of a shift register. This module takes in 8 bits of data from the AXI Data FIFO and then determines how to transfer it to the eMMC based on the controller's internal data mode which is set by the software driver through a custom command. There are three primary data modes: x1, x4, and x8 bus width. This number relates to how many bits of the full 8 bit eMMC DAT bus are being used for data transfers. Therefore, x8 width bus mode would mean we are using the full bus. All of the data being sent on an eMMC DAT signal must have a 16 bit CRC transferred after its data. Eight CRC16 generators are instantiated for each signal on the eMMC DAT bus but less are used if the mode is x1 or x4. A CRC is generated by the data being shifted into it and is transferred to

the eMMC immediately after the data has been transferred. This module only sends a single block of data when activated where a block consists of 512 bytes.

This module can also handle dual data rate transfers where data is sent on both edges of the clock. However, that process is abstracted from this module, but it conforms to the Vivado selectIO wizard which performs this operation. This selectIO takes in two bytes of data and will send the first byte on the positive edge and the second byte on the negative edge. This effectively double the amount of eMMC data signals from 8 to 16. For DDR, this module now needs to instantiate 16 CRC modules and the data mode input now includes a DDR option. This module can still perform SDR operations by making the first 8 data signals equal to the second groups of 8 data signals which means that the data is the same on both edges of the clock.

The Read Module:

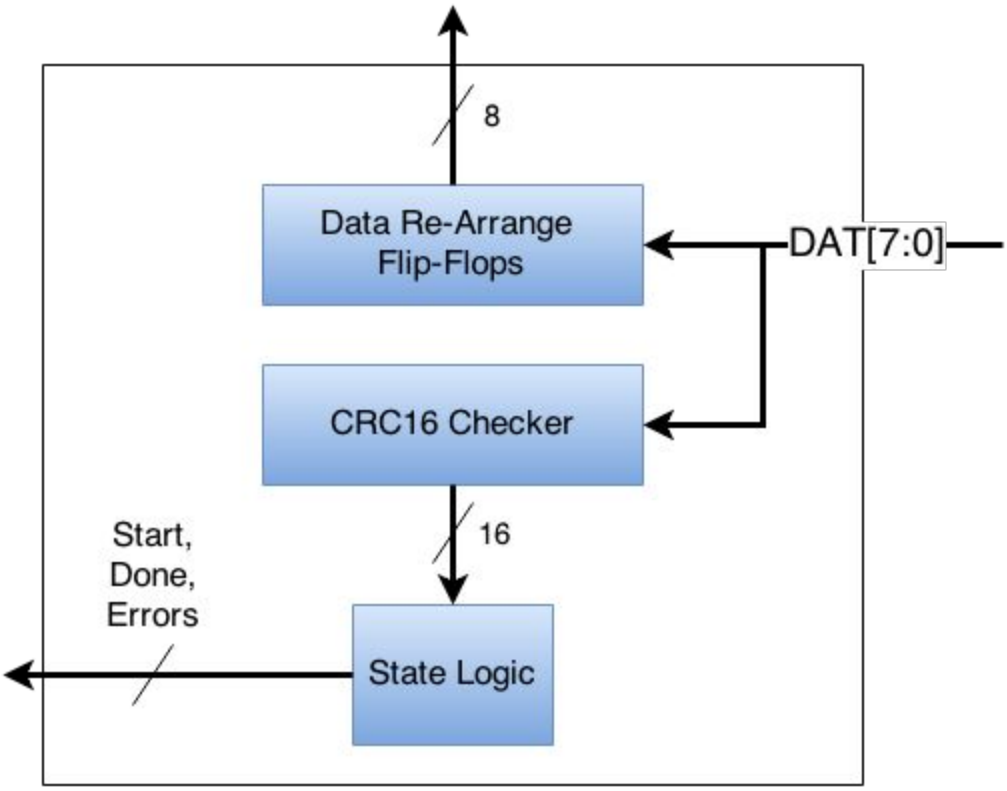


Figure 12: Implementation of the Read module.

The Read Module has a similar implementation to the Response Module minus the use of a shift register to store the response before sending it to an AXI FIFO. This module needs to

arrange the incoming data into a byte such that is can be pushed to an AXI Read FIFO for the software driver. The data only needs to arranged into a byte when the data modes are either x1 or x4. These modes imply it takes more than one clock cycle to receive a byte from the eMMC and need to be stored in flip-flops until the entire byte has been transferred. In x8 width mode, a byte is received every clock cycle and can be transferred to the AXI FIFO as soon as it is received. A CRC16 module takes in all the data and a CRC sent from the eMMC. The 16-bit output from our CRC module should be 0 if no errors occurred during transmission. A CRC16 checker is required for each data signal used in transfer. This module reads in a single block of 512 bytes.

This module can also handle eMMC dual data rate modes where data is captured on both positive and negative edges of the clock. This process is abstracted and this module is given 16 data signals as input where 8 signals presents the data on the positive edge and the other 8 signals represents data on the negative edge. This effectively double the amount of data signals. This requires doubling the amount of CRC modules as well. This module can still perform SDR reads by ignoring half of the data signals that represent the data on the negative edge of the clock.

Control Registers:

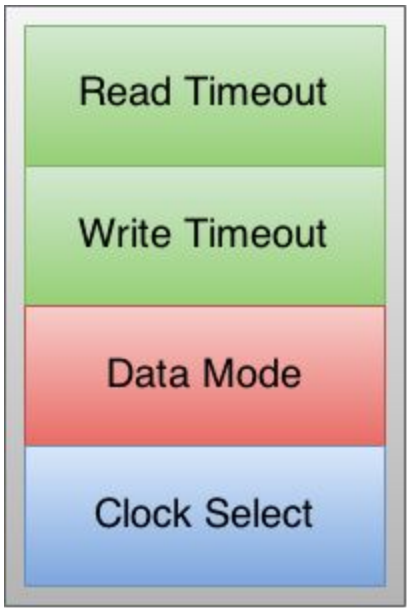


Figure 13: The control registers in the hardware controller.

There are four control registers within our hardware controller which can be set through custom commands by software driver.

1. The Read Timeout control register. This is a 32 bit register which tells the controller how long to wait for a block of data to arrive from the eMMC for a read command. A timeout occurs if a block of data does not arrive within the number of clock cycles specified by this control register. This is a control register because the calculations to determine this timeout value is beyond what the controller is capable of. Assumes a maximum value by default.
2. The Write Timeout control register. This is a 32 bit register which tells the controller how long to wait on a busy period after writing a block of data. The controller will signal a timeout to the software driver through the Data Status AXI FIFO after waiting the number of clock cycles specified in this control register. Similar to the read timeout, this value is beyond the controller's capability to calculate and needs to be set. Assumes a maximum value by default.
3. The Data Mode control register. This 3 bit control register sets the internal data width mode of the controller which defines how data will be transferred and received from the eMMC. This register content shares the same format as the DATA_WIDTH byte in the eMMC's Extended CSD control register except it is only a 3 bit value. It is set by the software driver because there would be ambiguity if multiple eMMCs were connected with different data width modes. Rather than have the controller remember all of the data widths of all connected eMMC devices, the controller relies that this is set by software.
4. The Clock Select control register. This 1 bit control register selects the clock speed in the controller and eMMC between an input clock and a divided clock. A slower clock is required during initialization of the eMMC based on the JEDEC specification. Our controller implements a clock divider to create this slower clock based off the input clock. Writing a 1 to this register sets it to the slower clock and writing a 0 sets it to the

input clock. This is done because the software driver should have control of this feature and to reduce logic in the controller.

Clock Switching:

The clock switching module was composed of a MUX with each selection running through a synchronizer. This synchronizer ensured that there is no glitch on the output of the clock if the clock select signal is switched asynchronously. Because the two clocks were related (one being divided from the other), only one stage of synchronization was needed.

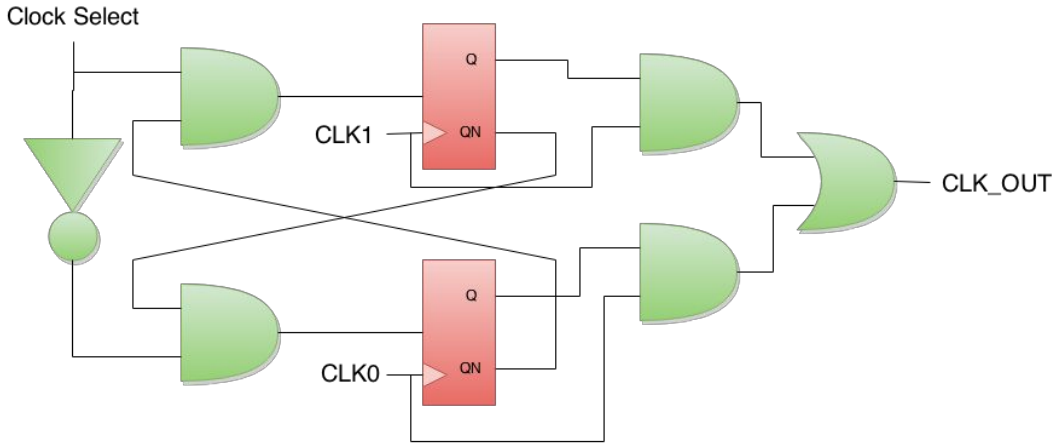


Figure 14: Clock Switching Module

Hardware Controller and eMMC Device Timing: DDR (dual data rate) and Clocking

DDR mode involves sending and receiving data on the rising and falling edge of the clock. DDR mode was implemented using Xilinx Vivado's selectIO wizard. This wizard gives various options for outputting and inputting DDR data. It inputs/outputs 16-bit SDR (single data rate) data from/to the controller and inputs/outputs 8-bit DDR data from/to the eMMC device. We configured the DDR mode in the selectIO wizard to output in "opposite edge" mode and to input in "same edge pipelined mode". A block diagram for our implementation with the selectIO block is shown in figure 15.

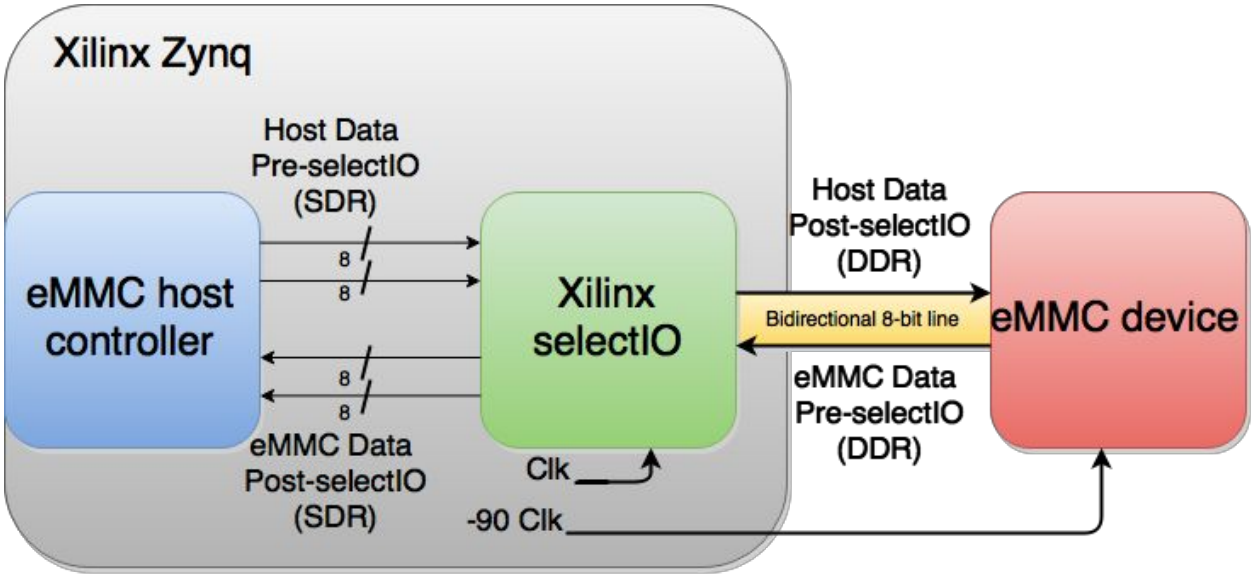


Figure 15: Block diagram of Xilinx Vivado selectIO

We also output a -90 degree phase shifted clock for the eMMC to run off of. This phase shift allowed the eMMC device to capture data accurately while values weren't changing ensuring that we don't violate setup or hold times. The timing diagram in figure 16 shows the output from our controller.

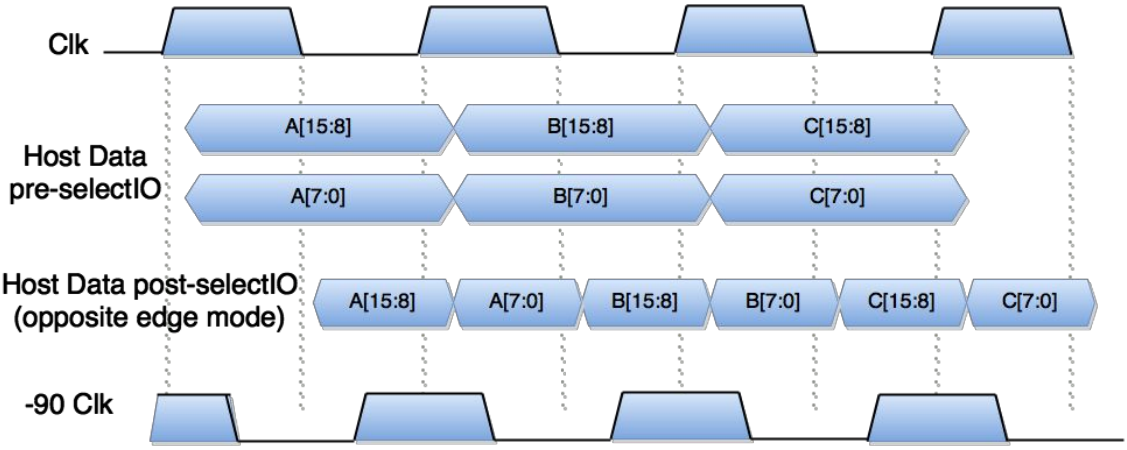


Figure 16: DDR output timing diagram

The controller sends 16 bit data shown as A, B, and C above. In the diagram each 16-bit piece of data has been broken up into 2 8-bit pieces of data. Once this data reaches the selectIO block it is converted to a DDR transmission, and a -90 degree phase shifted clock is sent out along with the DDR data to the eMMC device.

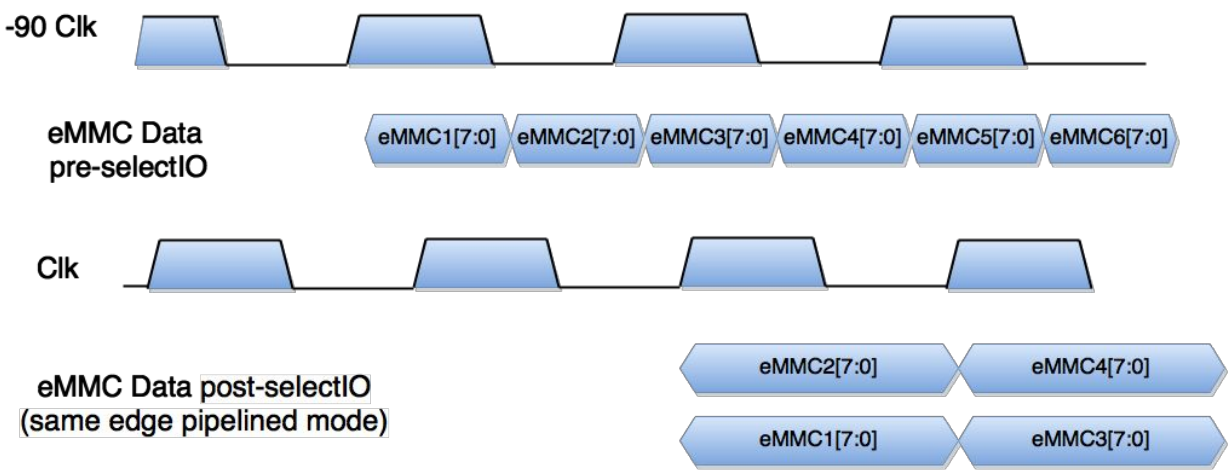


Figure 17: DDR input timing diagram

Figure 17 shows the data as it is input from the eMMC device into the hardware controller. It's clocked from the -90 degree phase shifted clock and captured by the hardware controller on the regular clock.

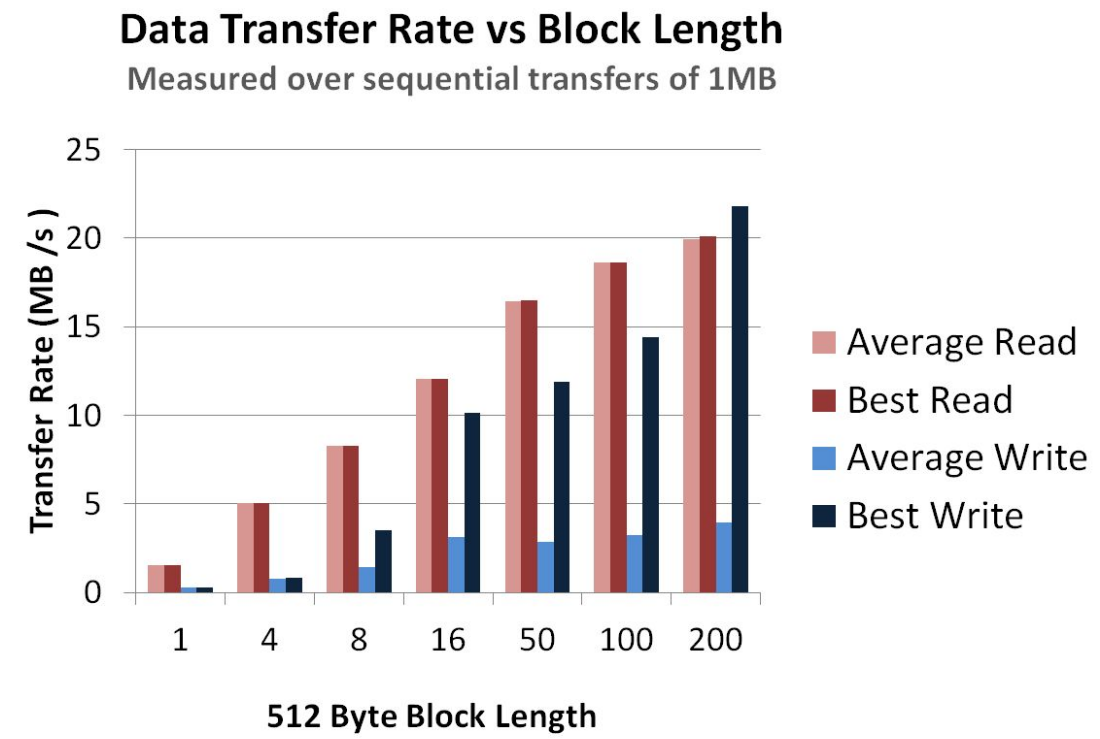
Results

After completion of the driver we conducted performance tests on the eMMC chip that we were given. We tested the speed of the chip against two variables. The first test determined the speed using variable block sized operations over sequential reads and writes of 1MB. This started with 2000 operations with block size of 1 and ended with 10 operations with block size of 200. For reads, as block size increased, speed in MB/s increased dramatically. Additionally, reads were very consistent across all the trials. Writes were a little more complex. Average writes also tended to increase as block size increased and the speed of best writes of each set of trials dramatically increased with block size. The largest difference between average and best writes were due to a caching effect that quickened the time that writes took which occurred roughly every other 1MB section. This is likely because the device caches sequential data up to a certain point and then performs the relatively long (we observed it at about 500ms) write to

flash. This effect is obvious in Figure 19 below which shows 10 trials of 1MB sequential writes in a row.

The second test showed how length of transfer effected speed. Again reads were very consistent as transfer length increased, staying around 14-16 MB's. Again we have the phenomena with the inconsistent writes. With a very short transfer length, the average writes were relatively slow but the best writes were very fast. As the length of the transfer increased, the average writes and best writes converged. This is because each of the longer writes had included the time consuming write to flash.

To get accurate results, we used hardware timers addressable from software that were triggered from logic internal to our controller state machines. We used two timers for the measurement. We started the two timers at exactly the same time. We triggered the first timer to stop based on a signal from the controller state machines that occurred at the beginning of a read or write and similarly triggered the second timer to stop at the end of the operation. Finding the difference between these two timers gave the duration of the operation. Adding these up over the number of operations in the length of the transfer gave the total time of the sequential read or write. Measuring the transfer in this way disregarded the time that the software took to transfer commands and data into the FIFO's.



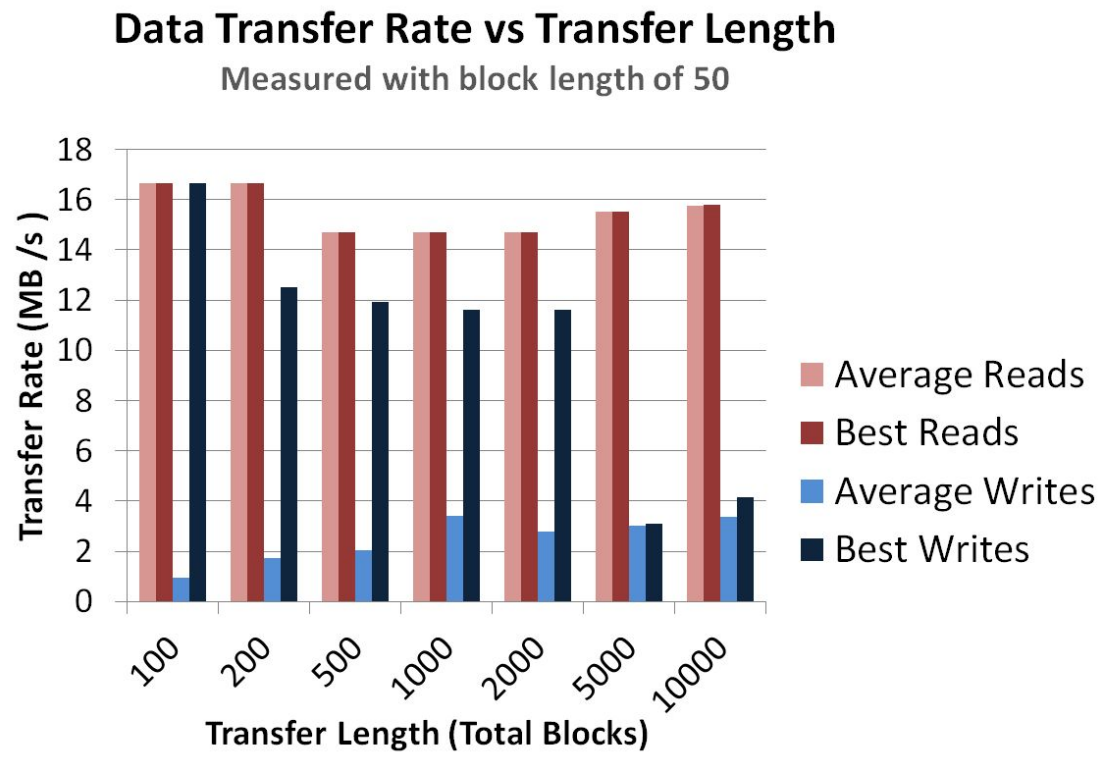


Figure 18: Performance Data

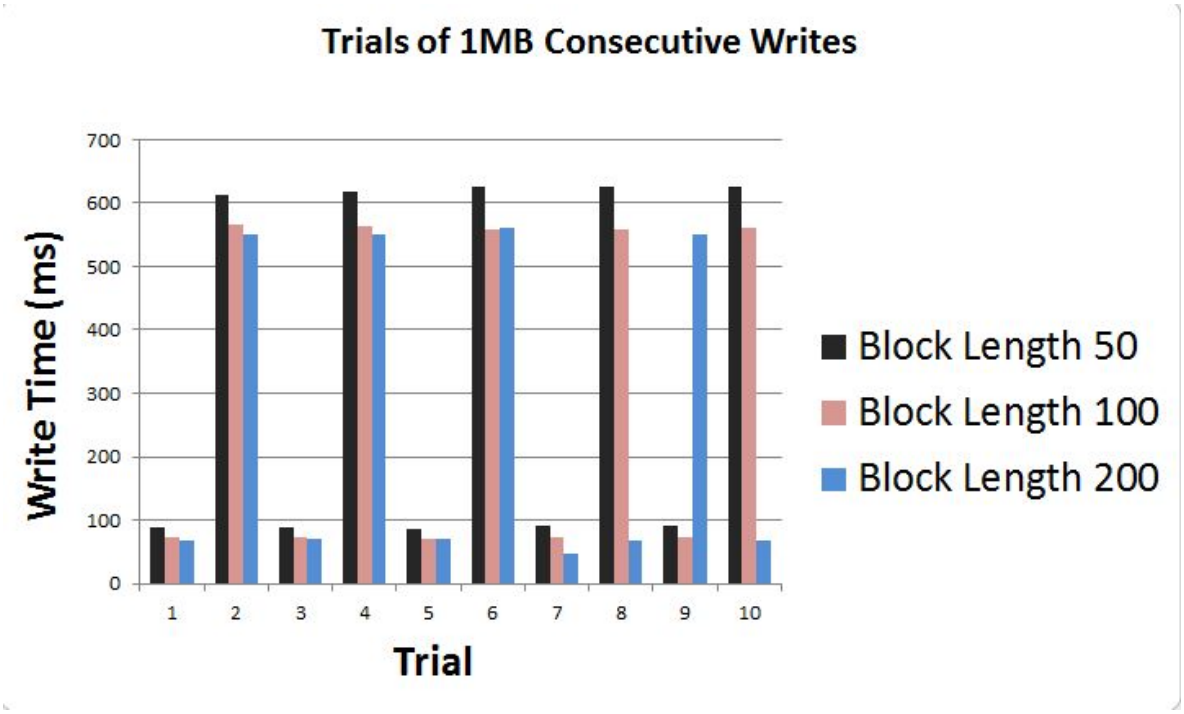


Figure 19: Write Operation Caching and Writing

Challenges

The main challenges associated with this project were correcting timing errors between the FPGA and the external eMMC chip in both single data rate (SDR) and dual data rate (DDR) modes. In our initial designs using SDR mode, we sent the clock out through a Xilinx DDR primitive module with one input tied to high and the other input tied to ground. This was the recommended way to send out a clock. However this gave us timing errors that were apparent because we kept getting inconsistent functionality and data that reported CRC errors. This problem was solved by moving the clock output directly to a pin and not through the DDR primitive. When we implemented double data rate mode we again ran into inconsistent functionality and CRC errors on the data. The problem was that the signal quality of the output at 52MHz was not good enough for the eMMC to capture at the correct edge in double data rate. We corrected this by sending a -90 degree phase-shifted clock out to the eMMC device

while outputting data on the normal clock. This gave us much wider timing margins to work with and caused the controller to function correctly and consistently.

An ongoing challenge we are having is with the DDR functionality. From the figure 17 we saw how the data is transferred from the eMMC device to the hardware controller through the selectIO block. However, the eMMC device begins transferring on the falling edge of a clock cycle. If we show this along with a standard eMMC formatted transmission we end up with the timing diagram in figure 20 where "same edge pipelined mode" is how our selectIO is configured. The standard eMMC formatted DDR data transmission can be seen in the "eMMC Data pre-selectIO" row. It begins with a start bit, followed by a "dont care", and then the data is transmitted. You can see the start bit is translated first through the selectIO and then the first data bit is translated at the same time as the "dont care". Our controller is not designed to handle the data formatted in such a way. None the less, at 52MHz DDR mode our design functions with "same edge pipelined mode".

However, slowing down the clock from 52MHz to 10Mhz causes this design to fail. A possible reason that the design is working at 52MHz is that there exists such a great delay between the edge of the -90 degree clock to the time that the data is output that it is no longer being captured on the edges shown in figure 20.

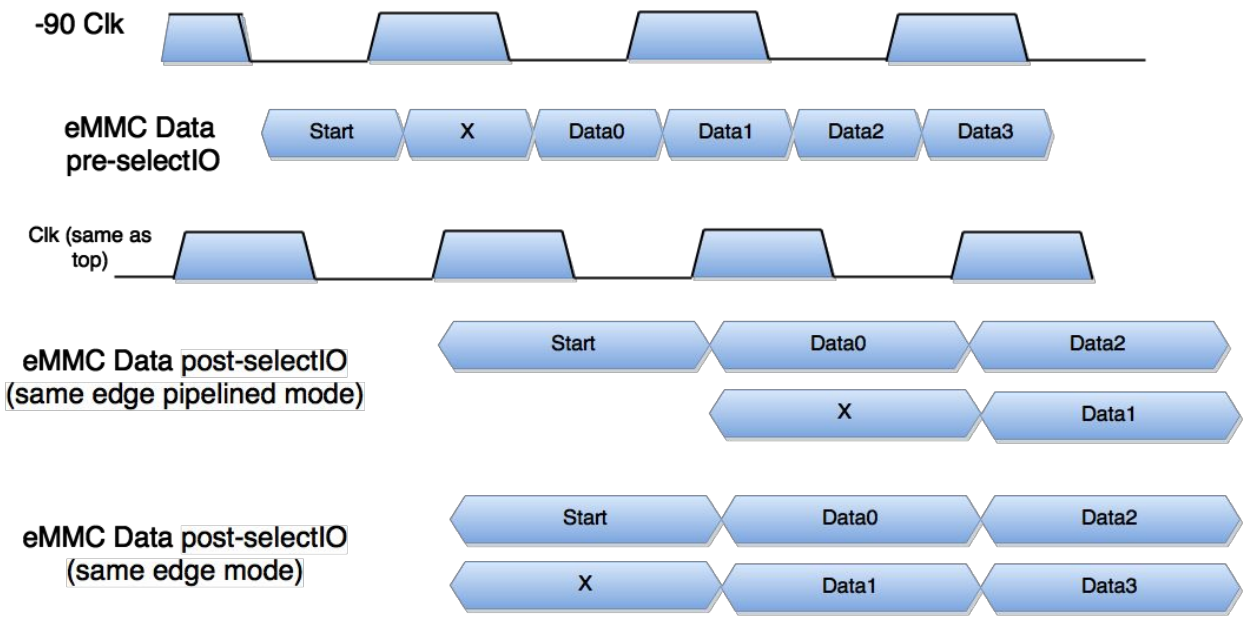


Figure 20: DDR timing diagram with example eMMC formatted transmission

Changing the selectIO DDR mode to “same edge mode” may cause the eMMC device to work at 10MHz, since this should cause the data to be formatted how our controller expects it, but more testing would need to be done.

Ongoing Objectives

The main two ongoing objectives of the hardware side of this project are the higher clock speed modes called HS200 and HS400. HS200 allows clock speeds of up to 200 MHz. However, this requires a tuning process to calibrate the timing for the data that is returned from the eMMC device. HS400 mode runs at a speed of 200 MHz but also uses dual data rate mode to transfer double the amount of data. This mode includes a data strobe signal that is accompanied by the returned data allowing the controller to sample the data on the edges of the data strobe signal.

Our plans to implement HS200 mode on the hardware side align with our approach for rest of the controller. We strive to make the hardware as simple as possible and have the software do any form of state management. In hardware we would have to implement an input delay in the data selectIO wizard. This delay could be customizable from a signal from the controller that could be changed by the software driver issuing a custom command. The controller would also have to implement CMD21 and modify our current data signal module to support special block sizes sent during the CMD21. These tools in hardware would be enough to allow software to complete the required process to be able to change the device to HS200 mode. This process involves sending a number of CMD21’s while changing the input delay to create a “window” of delay for which the data had been returned without a CRC error. The driver can then set the delay to the middle of this window and set the device and controller to HS200 mode.

HS400 mode uses the same clock rate as HS200 but it also uses double data rate. However HS400 mode also includes a data strobe that can be used to register the incoming data. The idea is that we would use this as an external clock into the data selectIO block. The

data output from the selectIO block would have to be synchronized in some way to the controller clock.

Acknowledgements

We would like to thank the Oracle team: Jon Lexau, Vincent Lee, and Hesam Fathi Moghadam. In addition we would like to thank our mentors David Munday and Ethan Papp.