ECE 440

Project 5: Block Memory - GDC - SPI

Submitted By :
Cory Holt
github.com/cholt150

Contents

1 Introduction 2
2 Design Process 2
3 Design 4
3.1 mem_ctrl 4
3.2 core_wrapper 4
3.3 spi_ctrl 5
4 Verification 5
4.1 Errors 8
5 Conclusion 9

1 Introduction

This project was a natural continuation of the GCD core project. It implements new concepts such as memory and SPI communication. The overall idea is that this module will read pairs of data and then calculate and transmit the results over an SPI bus.

2 Design Process

My overall design process began with my breaking the project into 3 main sections: a module that will read the data from memory, a module that will calculate the GCD of those two numbers, and finally a module that will transmit that result over the SPI bus. I created a simple block diagram that shows how these modules interact, and what kinds of hardware the datapath should contain.

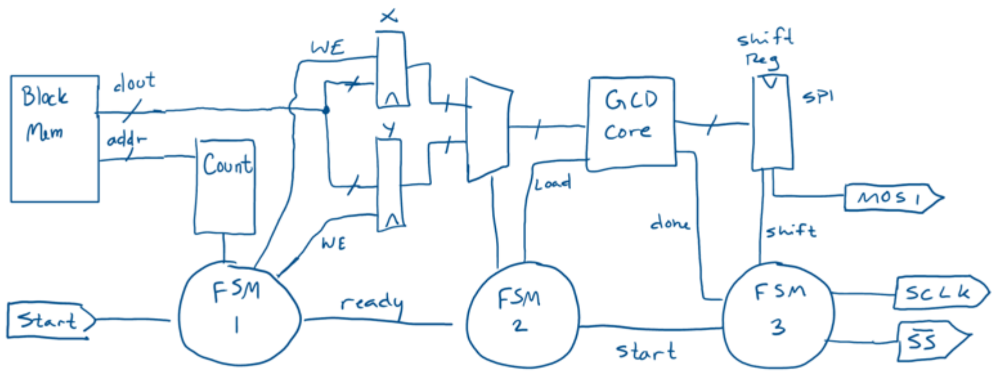


Figure 1: Block Diagram

The key pieces of hardware in the datapath are the two registers containing the X and Y values from the memory, the mux to select them and send them to the core, and the shift register that would shift out the result one bit at a time. There are also counters for the block memory which handles the address incrementing of the memory. Not shown in the block diagram are counters used by the SPI FSM which count clock cycles to derive a slower clock for the sclk signal and a counter to ensure we shift out the proper amount of bits on the MOSI wire. The following figures are my state diagrams for the 3 state machines I used.

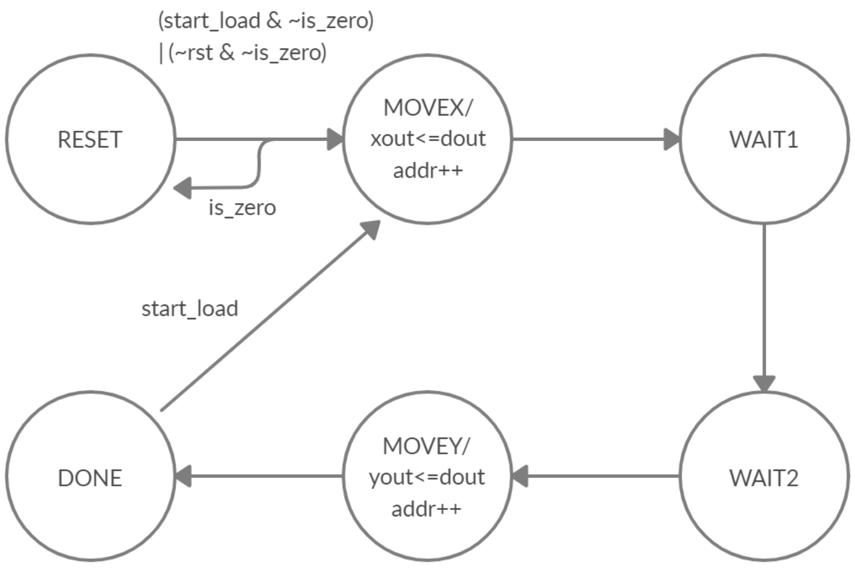


Figure 2: State Diagram 1

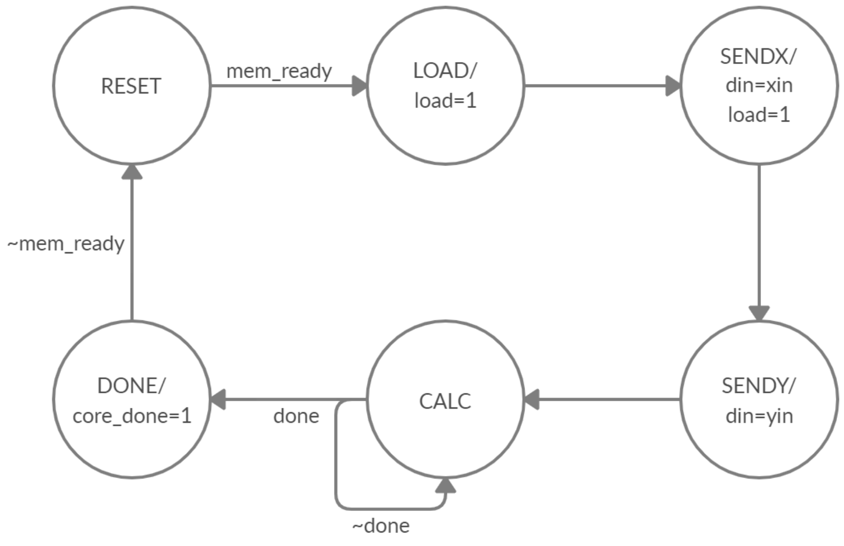


Figure 3: State Diagram 2

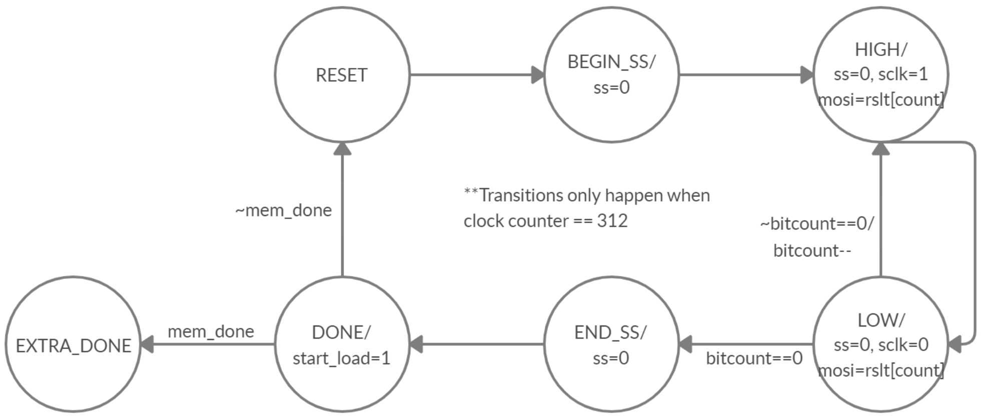


Figure 4: State Diagram 3

After this initial planning, I began to implement the design in SystemVerilog.

3 Design

I broke this project into 3 sub-modules with an overall module to connect them. Each module contains an FSM and the datapath that it controls.

3.1 mem_ctrl

The memory control module does just that, it controls the block memory access. It has a counter which increments through each address location, and moves a pair of neighboring values to registers for use by the GCD core. The controller here starts on a signal `start_load` or when the `reset` signal is de-asserted. It then works through a linear sequence of reads and transfers. The machine utilizes two wait states to ensure that the data is presented properly after a new address is given. in The data is copied to two 8-bit registers so that the core has parallel access to the data. The machine then waits until a new `start_load` signal is received from the spi module. It sets a flag to indicate to the core module that data is available.

3.2 core_wrapper

Once the `mem_ready` flag is set, this machine also steps through a linear process of loading the core and waiting for results. It asserts the load signal to the core, and then in the following clock cycles, it presents X and Y values sequentially. It then holds in a calculation wait state until the core sets its done signal. Once the core is done, it holds in a done state until the memory controller begins its next read sequence. The done state also sets a `core_done` signal to indicate to the SPI module that the result is ready to be sent.

3.3 spi_ctrl

This module was a bit more tricky as it runs at a lower frequency than the rest of the machine. The signals are sent out at 200 kHz, while the rest of the machine runs at 125 MHz. Therefore, a counter is kept, and this machine only transitions state when the counter reaches the max value of 312. Therefore, each transition of this machine has a conditional to check the comparator of the counter. The reset state is held until the `core_done` signal is asserted. Then it steps into the BEGIN_SS state. This state drives the slave select signal low, and resets a small 3-bit counter to 7. It then transitions to the clock loop. The clock loop has two states, HIGH and LOW. unsurprisingly, the HIGH state drives the sclk signal high and the LOW state drives the sclk LOW. The HIGH state unconditionally transitions to the LOW state. The LOW state checks if the bit counter is zero. If it is zero, it breaks out of the loop and transitions to the END_SS state. If it is not zero, it decrements the counter. Using the counter value, MOSI is set using a mux to select which bit of the result is presented to the MOSI wire. I had planned on using a shift register to shift out the bits one by one, but using the counter value here allows for an easier solution when creating the code for the hardware, as I can reference the counter value as an index of the register. After the clock and data sequence is complete, the END_SS state holds the SS signal low for one more 200kHz cycle to meet SPI standards. It then enters a done state which sends the `start_load` signal back around to the memory controller to signal it should start a new read sequence.

4 Verification

Initial verification for me was running simulations on the project in Vivado. Due to the differing time scales, I had to look at both a small window of time when the loading and processing was happening and a large window to see the slower SPI machine function. The first simulation here shows how my state machines handle the loading and computation process. The second simulation shows how the SPI module behaves as it sends the results on its bus.

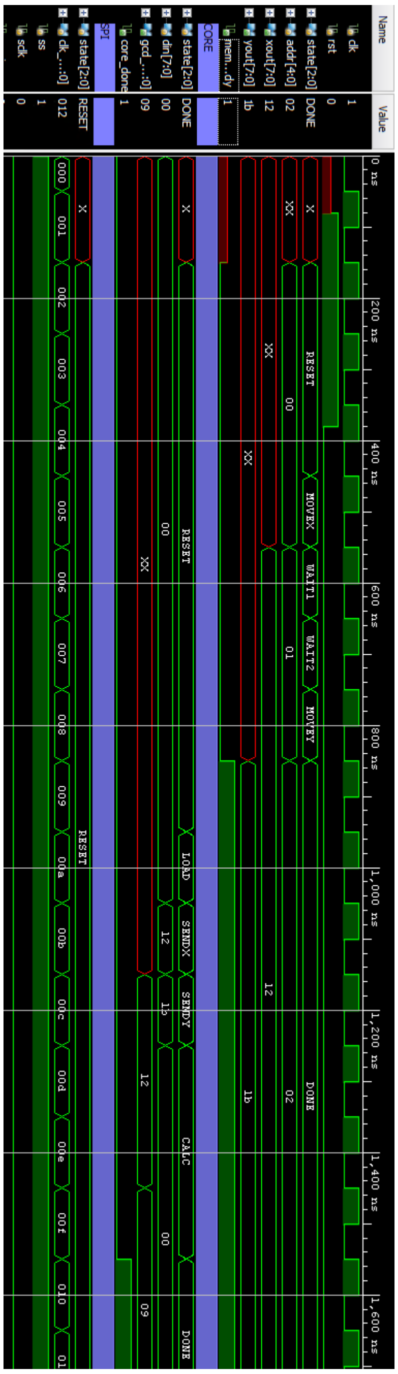


Figure 5: Behavioral Sim 1

As you can see, the memory control steps through each state and then sets a flag for the core. The core then loads the values and computes the GCD. It then sets its done flag, presents the values and waits.

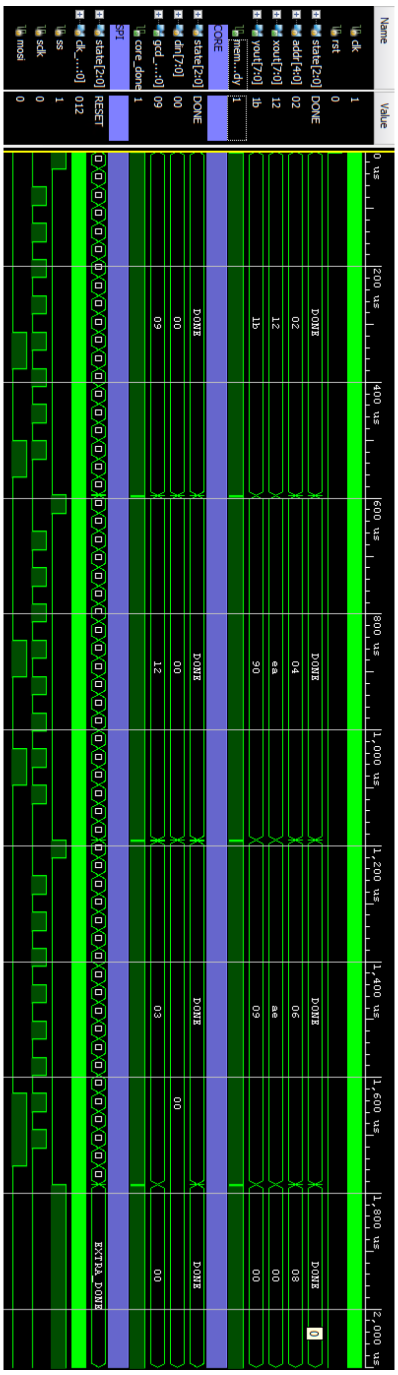
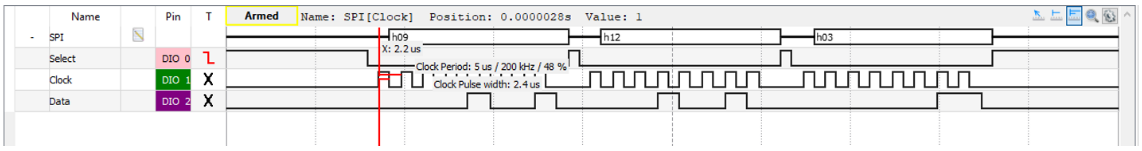
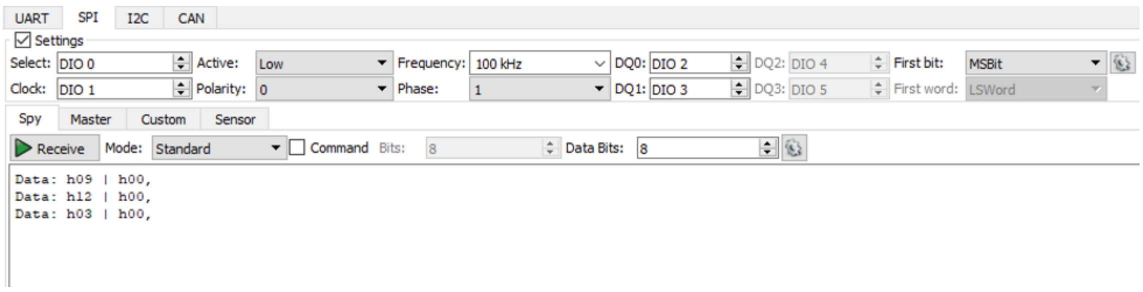


Figure 6: Behavioral Sim 2

From the first simulation we know that the result should be 9. We can see the first SPI transmission lowers the SS signal, starts the clock, and sends results. The waveform clearly shows that the binary value 9 was sent over MOSI, so I presumed that the module was functioning as intended.

Finally I verified the project by actually implementing the design on the board and verifying the behavior using the waveforms software and the Analog Discovery. The first screencap shows what the logic analyzer captured after releasing the reset button. The second screencap shows what the protocol tool showed after releasing the button.


Name
Pin
T
Armed
Name: SPI[Clock] Position: 0.0000028s Value: 1
SPI
Select
Clock
Data
DIO 0
DIO 1
DIO 2
h09
X: 2.2 us
Clock Period: 5 us / 200 kHz / 48 %
Clock Pulse width: 2.4 us
h12
h03



Settings
Select: DIO 0 Active: Low Frequency: 100 kHz DQ0: DIO 2 DQ2: DIO 4 First bit: MSBit
Clock: DIO 1 Polarity: 0 Phase: 1 DQ1: DIO 3 DQ3: DIO 5 First word: LSWord
Mode: Standard Command Bits: 8 Data Bits: 8

```
Data: h09 | h00,
Data: h12 | h00,
Data: h03 | h00,
```

As you can see, I verified that the actual results match my simulations. This also demonstrated that the clock frequency was indeed 200 kHz, and that the tool was able to identify and decode my results as an SPI signal.

4.1 Errors

I ran into a few errors while creating this project. The first issue I encountered was a read error on the memory. It appeared to be doubling the first value and neglecting to read the second address. This was caused by my reading the memory too soon after giving it a new address. I resolved this by adding an additional wait state to the FSM in between reads. Following that the memory read and behaved as expected.

I ran into no problems using the core. My machine processed the values as expected. This functionality was mostly borrowed from Project 3.

I did have a major bug with the SPI interface initially. It was dropping the last value and never sending the results. So if I gave it three pairs of values, it would only transmit two results and then say it was done. This was due to the SPI controller getting the ”zero” signal from the memory as soon as it happened, and therefore never entering the sequence to transmit the results. I fixed this by adding a final trap state ”EXTRA_DONE” and changing the order in which the done signals were checked.

5 Conclusion

I struggled with this project most in the beginning stages as I had a rough time planning out what kind of hardware I needed to make this project work. I initially tried to start with SystemVerilog code and work my way back from there, but I quickly found out that it doesn't work as well as I would like to work backwards like that. I enjoyed the process of creating a system like this, and it was really satisfying to see the results pop up on the protocol analyzer with the right results. I learned more about the design process and how I should approach problems. Overall this was a good project for me and it helped extend my knowledge of the tools and language.