I SPI some Peripherals

Vienna, Shashank, Jordan & Alli :)

December 2019

Abstract

Have you every wanted your CPU to outsource its memory or multiplier? Well, we have just the solution for you!

This paper describes our Computer Architecture class final project which consists of understanding Serial Peripheral Interface (SPI) protocol and implementing it in Verilog. SPI protocol is a method for communication method between two computers. In SPI implementation, there is one computer which sends instructions to the other computer as to what it should do. The computer sending the instructions is called the parent, and the one receiving signals is called the child, or the SPI peripheral. SPI peripherals can do different types of functions, depending on their structure. We managed to implement two SPI peripherals. One stores incoming data and retrieves it and the other multiplies incoming signals. Finally, we implemented a test bench that acts as the parent module to communicate with the peripherals.

Motivation

All four people on this team are ECEs. Some of us have seen and worked with SPI before, but didn't know how it actually worked from a hardware standpoint. Our experiences with SPI and with computers in general have shown us that communication is a prevalent topic in computer systems. We wanted to choose a project which helped us understand what is happening "under the hood" so that we would be better prepared when we see it in the future.

We decided there's no better way to learn than to do-learn! We believed that creating a whole SPI system would allow us to understand its intricacies.

Overview of SPI Protocol

Serial Peripheral Interface (SPI) is a communication protocol for computers to send and receive data from each other using a 4-wire synchronous component [**sparkfun**]. In this protocol, one of the computers is instructing the other to preform a task. The two components are deemed the Parent and the Child, where the Parent requests that the Child preform a task, and the child cannot make a request of the Parent. A Parent can have multiple children [**sparkfun**].

The SPI protocol communicates using 4 wires: MOSI, MISO, SCLK and CS [**analog**]. We have decided to rename MOSI and MISO to POCI (parent out, child in) and PICO (parent in, child out), respectively, because we are more comfortable with this naming convention. The parent and the child components each have their own individual internal system clocks, and they both communicate using a new serial clock which is set by the parent [**analog**]. This serial clock is the SCLK wire that is input into the SPI protocol. The POCI wire takes the output of the Parent module and sends it as an input into the Child module [**analog**]. The PICO wire does the same but in the reverse direction [**analog**]. The Parent component can have multiple Child components, so the Chip Select (CS) wire is used to select which of the Child components are

requested to preform a task [**sparkfun**].

Implementation

Modules

We decided to utilize our knowledge of SPI communication to implement a multiplier peripheral and a memory peripheral. In order to create these modules we referenced Lab 2: SPI Memory from Computer Architecture 2018 [**Lab2**].

The multiplier takes two 4-bit values from the parent and returns an 8-bit result. The memory takes 16 bits from the parent: the first bit is a read/write bit, and the next 7 bits denote an address. If the first bit is a *write* bit then the parent sends an additional 8 data bits to be written to the address in memory, and if it's a *read* bit, the parent expects to receive 8 bits of data from the peripheral memory. The basic structure of both peripherals revolves around a shift register that accepts serial and parallel inputs and has serial and peripheral outputs. The shift register can either parallel load (PLOAD), hold (HOLD), shift right (RIGHT), or shift left (LEFT). The following section describes the Finite State Machine that drives the peripheral operations.

Below is the FSM for the multiplier peripheral:

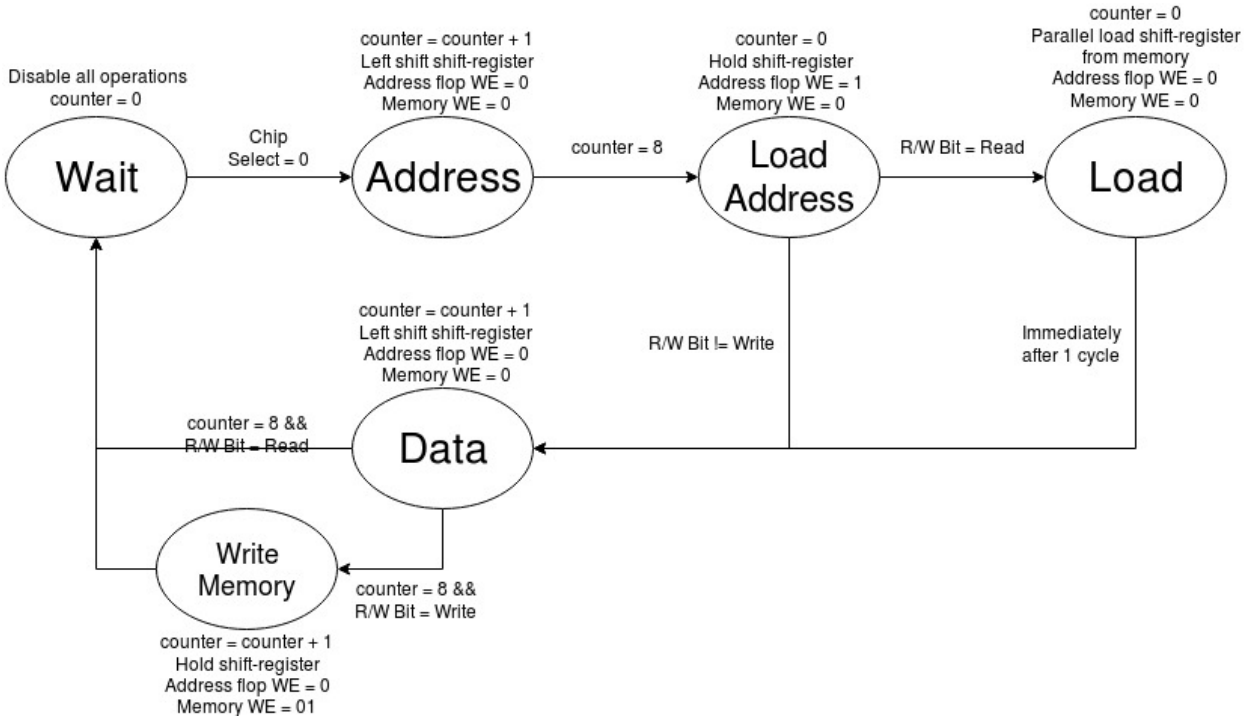

Figure 1: FSM of Multiplier Peripheral

This is the FSM of our memory module. For this module, we have six states. Initially, the SPI memory peripheral is in the WAIT state. The purpose of the WAIT state is solely to listen for input from the parent module. Once the parent sends the select signal to the child module, the child goes into the next state, which is the Address state. In this state, the address is supposed to be fully loaded into the shift register and ready to be put into the memory. Interfacing with the memory is the next step, which happens in the Load Address state. Here, the memory is just directed to an address. What happens next is dependent on the read/write command. If the parent tells the child that this is a Read operation, then the memory parallel

loads the data at the address specified into the shift register . From there, the data gets sent out the PICO line. On the other hand, if the instruction is Write, then the child loads the address and then takes in the data into the shift register (during the Data state). Afterwards, the child goes into the Write Memory state, in which it takes the data in the shift register and puts it into memory at the specified address. It does not output anything during a write operation.

Our other peripheral is the memory peripheral. Below is its FSM:

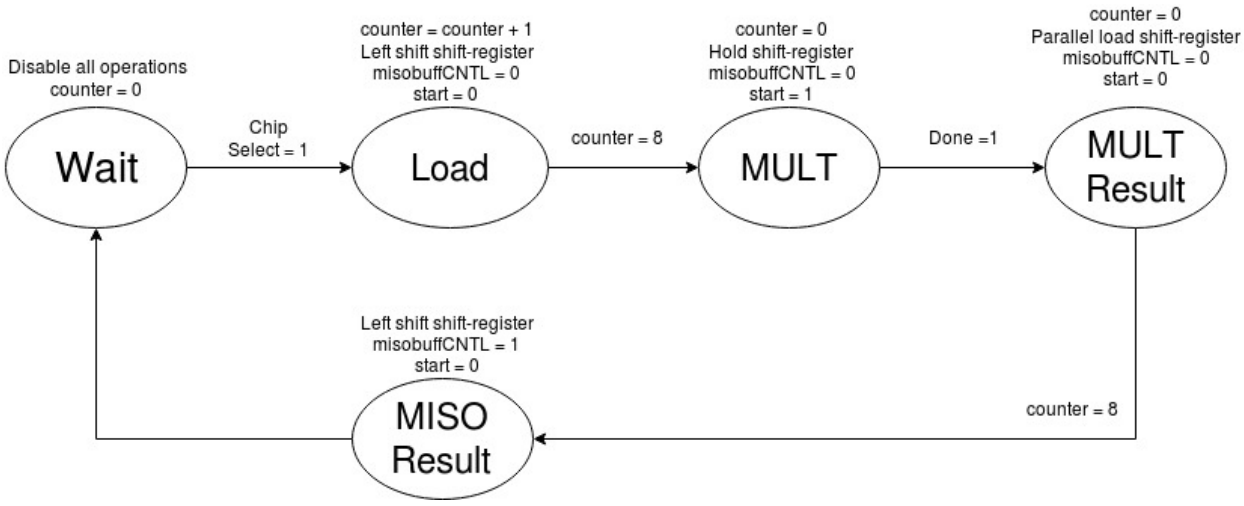

Figure 2: FSM of Memory Peripheral

Like the memory peripheral, this peripheral starts in the wait state, where it just listens for instructions. When it gets a Chip Select signal, it goes into the Load state, where it loads both 4-bit numbers that need to be multiplied into the shift register. Once that is done, the counter will reach its highest value and the child will know to go into the MULT (multiply) state. In this state, the two values get multiplied in our multiplier. When the multiplying is done, then the multiplier produces a done signal. That in turns, tells the child to go into the MULT Result state, where the output of the multiplier gets parallel loaded into the shift register and then serial output into a D-Flip Flop. Once the first value of the shift register goes into the D-Flip Flop, the child reaches its PICO result state, which turns the PICObuffCNTL signal to high. Now, the serially output signals going through the d-flip flip are allowed to continue to the PICO line. Once this is complete, the child goes back to the wait state.

Testing

In order to see these modules in action and make sure that they work, we tested the memory and multiplier peripherals in Verilog test benches. First we test the modules separately in individual test benches and then we test then together.

When we test the multiplier, we feed in the values 6 and 1 in one POCI . We wait the right number of cycles, and make sure that the PICO output shows 6*1, bit by bit.

For the memory test, we set the read/write bit to *write* and pass the peripheral an address. Then we send 8 bits of data to the peripheral. Once we've written to the memory, we know what to expect at this specific address. To check, we set the read/write bit to *read* and send the same address. In the test bench we check the MISO line for these bits and check to see if they match what we sent in the *write* test.

Our final test bench tested both of these modules together. We connected the modules by simulating a situation where both of these modules are connected to a single parent module test bench. The test first selects the multiplier child to multiply two values and return it. We check that the value is correct and store it, and the test bench then sends the result of the multiplication to the memory Child and stores it a specified address. We then send a request to the memory child to read us the value at the previously

mentioned address. If it has stored the correct value then the memory Child has fulfilled all of its necessary functionality.

Schematics

Figure 3 shows the schematic for the SPI memory peripheral. The parent sends data in through the POCI input, as well as a serial clock (SCLK) and chip select (CS). The shift register handles the serial and parallel processes, and the memory is where the data is either loaded from or stored to. The finite state machine acts, as described above, to control the mode and enable pins. When reading data, the parent received data on the PICO line.

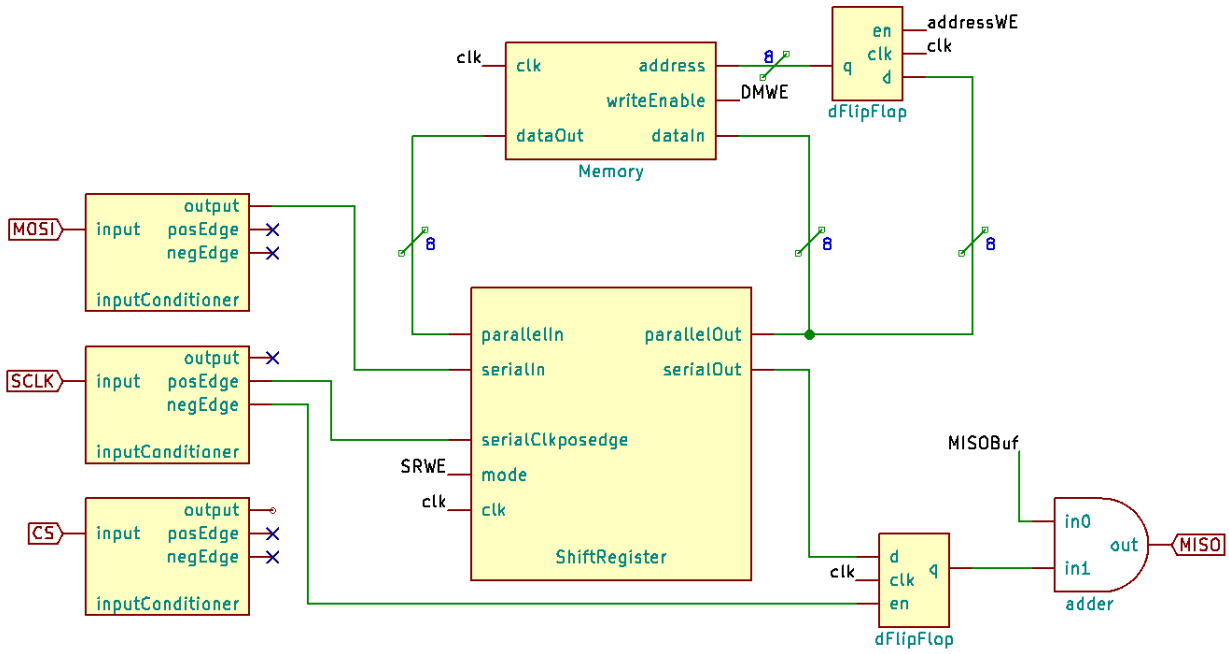

Figure 3: Schematic for the memory SPI peripheral

Figure 4 shows the schematic for the SPI multiplier peripheral. It works similarly to the memory module, but instead of reading and writing to memory it gives two 4-bit values to the multiplier and received an 8-bit result. The FSM controls the start pin and reads the done pin.

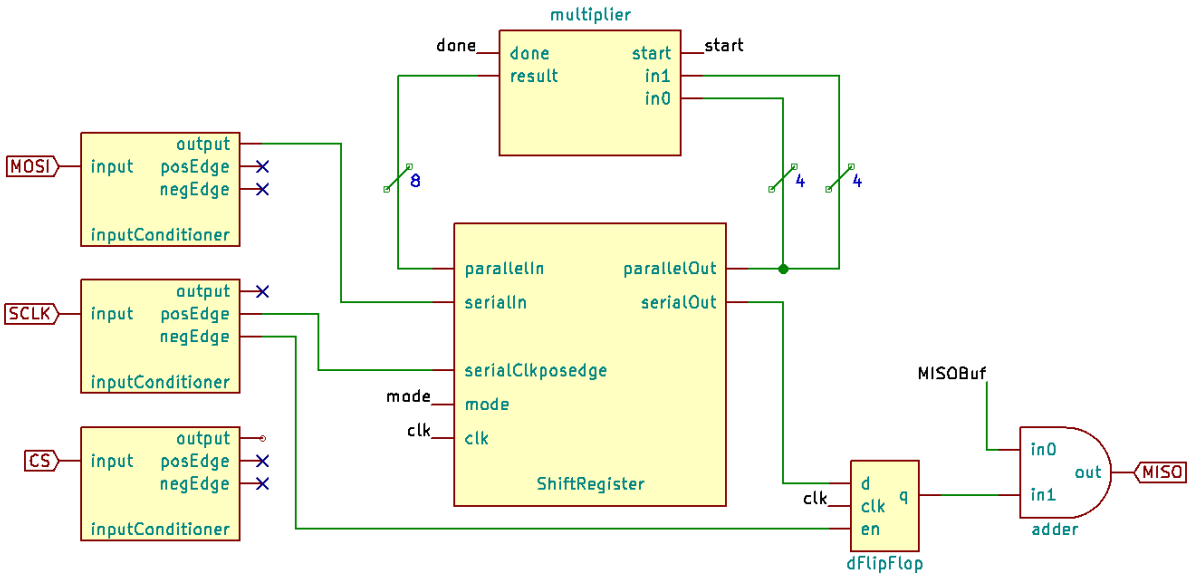

Figure 4: Schematic for the multiplier SPI peripheral

0.1 Subcomponents

0.1.1 ShiftRegister

The SPI modules required a shift register that could do both parallel and serial operations. The parallel operations were used to perform operations within the individual components and serial operations are used to communicate between the Parent and Child modules. Therefore, we created a shift register that can both take in and output values in parallel and serial. The first register takes in the serial input and the final register outputs the serial output. Each of the registers takes in a parallel input and a serial input that are decided between using a multiplexer, and each of the registers outputs a bit for the parallel output of the shift register. Our shift register also takes in a mode, which decides between different actions: Holding the current value, shifting values left, or parallel loading, which is loading a value into each of the shift registers at once. The most important aspect of this shift register is that it can act using both the clock and serial clock as an input. By using the serial clock as an enable, the shift register can update on the clock, but will only ever shift values on the serial clock, which has a speed agreed upon by both the parent and child modules.

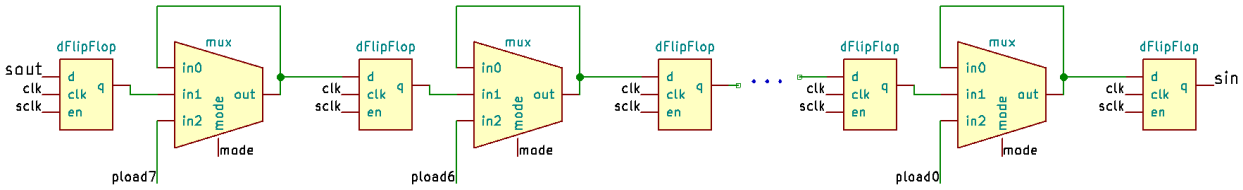

Figure 5: Schematic for the shift register

Input Conditioner

The implementation we were using was originally designed to be used on the FPGA and expected that its inputs would be very noisy. We too had originally wanted to implement SPI on the FPGA before pivoting. Thus, we still have the input conditioner. Although we do not need the input conditioner in our test bench

now, this component helped us understand better how SPI is realistically implemented. The actual purpose of this component is to synchronize, debounce and detect the positive and negative edges of incoming signals. The input conditioner synchronizes the noisy signal with the child's internal clock, by putting it through a pair of D-Flip Flops. The noisy signal is also debounced to ensure that a clear high or low signal is received, instead of noisy or a rapidly oscillating signal. The input conditioner then outputs the signal conditioned to be a clean signal synced to the clock. It also outputs a signal for when the conditioned signal goes high and when the conditioned signal goes low.

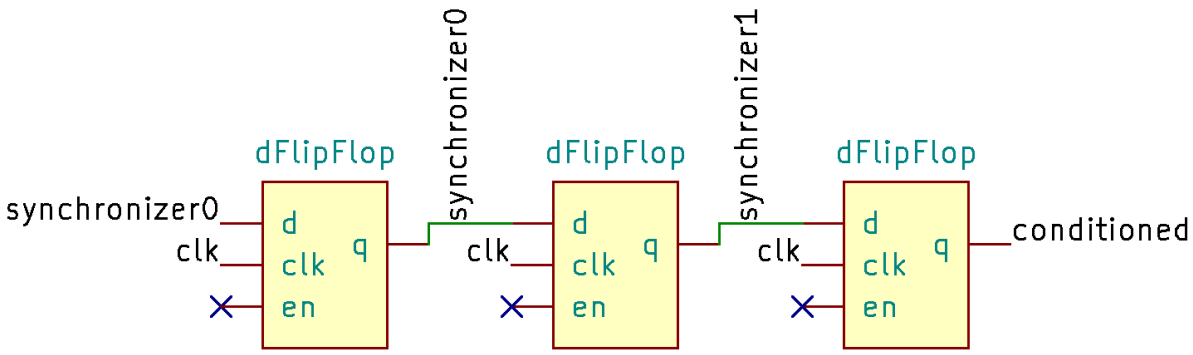

Figure 6: Schematic for the input conditioner

Issues we Faced

- General:
  - Shift register's parallel load was tied to the serial clock
  - The parameterization of the modules failed. A lot of the components we used in our circuit design were from previous projects, and we had paramterized those components to be of variable width in order to create less modules. However, upon using them in our SPI Module we found that the parameterization caused a number of errors in our code.
  -
- Multiplier:
  - Having a delay in the FSM Originally, we had a D-Flip Flop in our FSM that was holding the state. Because we couldn't set the input and output of the D-Flip Flop to the same reg wire, we named the output something else and checked that output reg's value to determine the state changes in our FSM.

    This caused a few different problems in for us. For one, we had some components of the child move to the next state while others were delayed for a cycle. This rippled into other problems, like the child getting stuck in the load state. Another problem that this caused was that the wrong values were getting input into the multiplier, because they got shifted over one too much.

    The way we solved this problem was that we took out the D-Flip Flop and just held state in a reg. This works for us, because the only way that the state can change is if it fulfills any of the conditions in the FSM- not on the clock.
  - Incorrectly conceptualizing and creating the FSM

    For the most part, we correctly conceptualized and created the FSM. However, there were some conditions and errors that we did not forsee. One of these errors is the done signal not being in sync with the sclk. This was a problem for us, because our FSM checks for the done signal on the

serial clock positive edge in order to enter the MULT result state. The way that we corrected for this error is that we created a secondary done signal, that would go high when the original done signal goes high and stay high until the child reached the next state.

A similar problem which we encountered was that the start signal was turning on multiple times during the MULT state, and even at the beginning of the MULT result state. This was happening because our condition to turn the start signal on was being fulfilled multiple times during the MULT state, including at the end. We ended up using the secondary done signal to regulate the start signal. Now, we check if the child is in the right state and the done signal isn't on before we turn on the start signal.

- Memory:
  - The FSM's counter was unable to initialize

    We initially wrote the FSM in a way that the counter was also the mode. This seemed like a good idea in theory, but in practice we ran into issues with initializing the counter properly so that it could also act as a mode.
  - Initially wrote test bench with parallel bits, when it should be handling bits 1 at a time serially

    We initially wrote a test bench that loaded bits in parallel and tried to read all 8 bits of data at once. We soon realized that we needed to write the test bench to send and receive bits serially since that is the whole point of SPI.
  - Eventually, the FSM turned out to be completely intractable.

    We kept trying to make the FSM work with the first attept code because it had some nice tricks in it (like using count for the mode, and using a shift register inside the FSM as a counter). We eventually ended up writing a new FSM based on the one we wrote for multiplier because it was a more straighforward software implementation. From this issue we faced, we learned that sometimes you have to make practical decisions about how to make progress most effectively.
  - The timing of the FSM was delayed by a clock cycle. We discovered that our FSM did not properly change state at the expected time as our memory module would always get a slightly wrong address, which would be especially problematic because we had a lot of difficulty finding the exact cause of the error. In order to fix this issue, we decided to start shifting the address into its designated register in an earlier state so that we would get the full address and be able to get the correct values.

Reflection

Overall, we feel that we followed good practices for organizing our thoughts and working efficiently. This was an engaging lab for our final project because it provided an opportunity to use technical skills we used throughout the semester while exploring a new concept. In hindsight it would have been nice to have a clearer end deliverable from the get-go, but figuring out the deliverable was part of our learning process. We could have done a better job of "killing our darlings" at some points. For example, we had an FSM for the SPI memory that we kept trying to use, and we finally ended up scrapping it and making a new one which was so much faster. It's important to know when it's worth fighting for a certain implementation style and when it's more productive to move on a try something else. In theory it seemed like a nice idea at the beginning of the project to implement SPI on a FPGA, but we realized that this would probably add a lot more debugging time that wouldn't directly contribute to our learning goals of learning SPI works.

Next Steps

This project has been an introductory way to learn about SPI and there are a lot more applications, variations and additional widgets. The reader might have noticed that we only created child modules, when SPI calls for a child and parent. We didn't create parent modules because there was no difference between creating a standalone parent module and test bench. In order to provide a technical challenge and some learning, we

would need to implement a parent module within some larger architecture such as a CPU. By implementing SPI within a CPU, we would have to create both new architecture to accommodate the necessary operations of the SPI parent module and new MIPS commands to make the SPI protocol correctly interface with the CPU.

We already have some idea of what would need to take place in order for SPI to be implemented within a CPU. Firstly, the CPU would need to be multicycle, in order to account for the fact that the SPI module requires multiple clock cycles to complete a single transmission. Also, a number of registers within the regfile will need to be specifically sectioned off from the rest to hold the values from SPI communications. All of these changes to the CPU make it compatible to interact with a SPI parent module.

In order to implement the parent module in itself, there must be an architecture very similar to the child module. The Parent module will have a shift register to both hold data from the CPU and to send data to the child module. The Parent module will also require its own finite state machine separate from the CPU or the Child module in order to correctly follow the SPI protocol. The PICO wire will connect to a buffer that will be checked by the CPU. If the CPU sees values in the PICO buffer, then it will save those values to the registers.

Although much of this planning is theoretical, these are all very actionable steps for improving upon our current SPI implementation. Taking this project further would push us closer to implementing SPI in such a way that would be used on actual circuits. If we wanted to push this implementation still further, we would use two FPGAs to simulate two different devices communicating with one another, which would almost directly mimic a practival real life SPI implmentation. There are plenty of great ways to improve upon and continue learning from this project.

References

SparkFun Page: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all

CompArch Previous Year's Lab 2 Start Git Repo: https://github.com/CompArchFA18/Lab2

Introduction to SPI by Analog Dialogue: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html