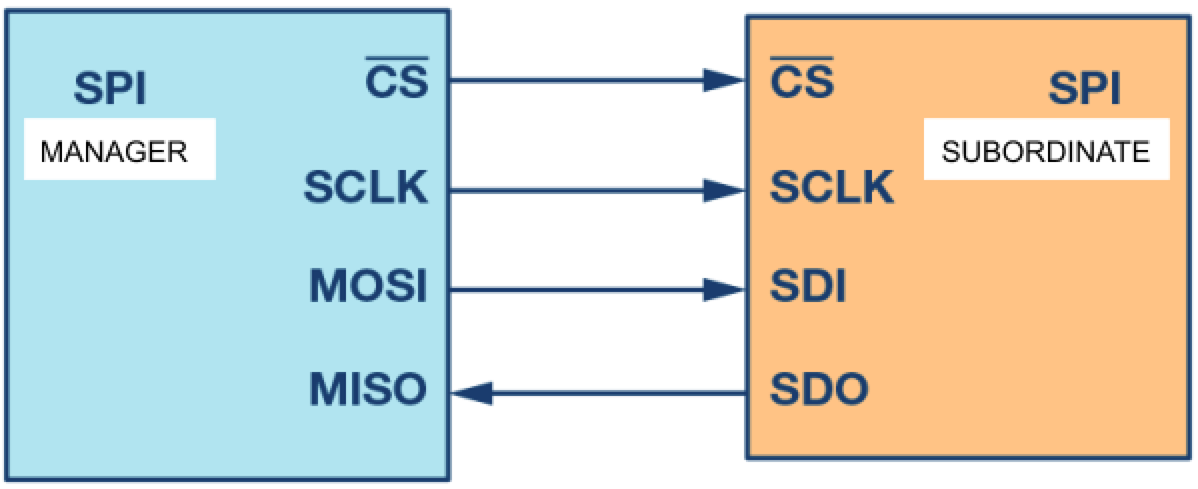
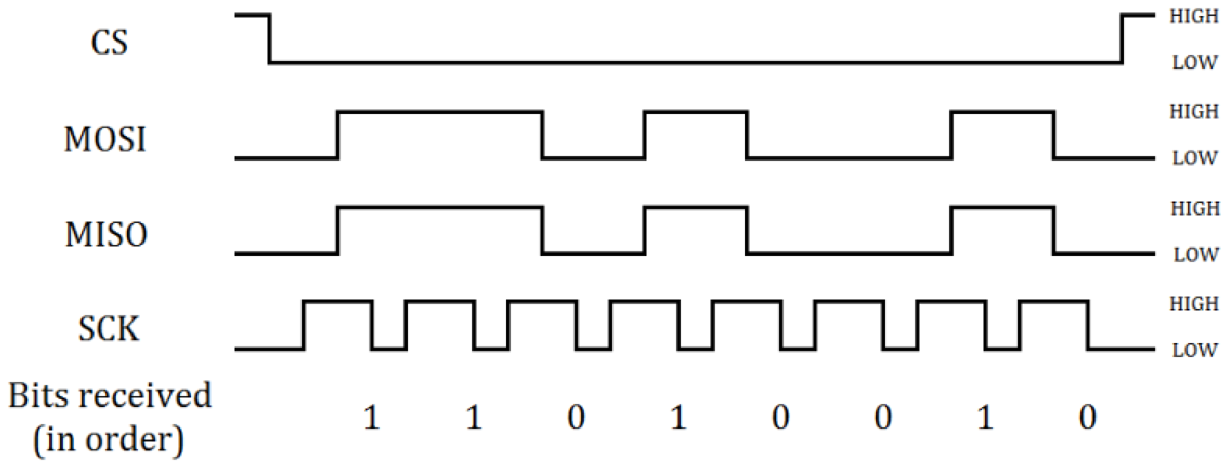
Building a "SPI Co-processor"

By: Audrey, Chris, and Jack Dec. 15, 2020

Overview





https://blog.digilentinc.com/pmod-communication-serial-peripheral-interface-in-detail/

We added serial peripheral interface (SPI) functionality to a CPU we had previously designed, which gives the CPU the ability to send and receive messages to and from external devices. Our SPI module is an extension of a simple CPU and operates by sending or receiving data to and from a separate SPI module. On a high level, the SPI module communicates through a master-slave relationship, which we will call manager-subordinate relationship for the sake of this paper. The simplest configuration of SPI is a single manager controlling a single subordinate. A manager can control multiple subordinates, but can only transfer information with one subordinate through a line called chip select (CS). Information is sent from the manager to the subordinate through the manager-out-subordinate-in (MOSI) line and is received from the suboordinate through the manager-in-subordinate-out (MISO) line. The SCLK, or the signal clock, manages the pace at which information is sent or received.

In our project, we treated our SPI module as a "co-processor", which in this case is a separate device sparingly connected to the CPU. These connections are primarily between the SPI module's register file. When data is sent from the CPU to specific locations in the SPI register file, messages can be sent or received.

SPI Is Useful

The Serial Peripheral Interface protocol, also known as SPI, is a communication interface specification used for short distance communication. The interface was first created in 1980's by Motorola and has become an industry standard since. Due to its ubiquity, we believe that it could provide useful knowledge for future careers, especially in SPI's uses for Arduino and other micro-controllers. Initially, our plan involved implementing our SPI device on an FPGA and testing it with an actual device, which proved to be rather difficult both because of our inexperience with FPGAs and the remote learning environment. We instead decided to implement our SPI module as a MIPS "co-processor" after receiving a recommendation from our professor. This proved to be a good challenge and broadened our understanding of the MIPS architecture as well as teaching us how SPI works.

SPI Implementation

At first, we tried to understand how SPI works. We wanted to get a general overview of the interactions between the manager and subordinate blocks and wanted to make sure we understood the concepts before implementing it. From there, our team implemented a SPI module in Verilog to cement our understanding of the protocol. However, after implementing a SPI module, integrating it with our CPU proved more challenging than originally expected.

Implementation of Move From and Move To Instructions

Attempt 1:

Our first attempt was to wire a SPI module directly to our CPU. We thought it was necessary to implement it as a part of our CPU and CPU register, leading us to believe we had to reserve some CPU registers for our SPI device. This however meant that our CPU would not function properly unless we heavily reworked its design.

After consulting with Jon, our Professor, we realized that the MIPS instructions Move From (MFC0) and Move To (MTC0) could be used to handle moving data between our CPU and a separate SPI module.

Attempt 2:

For our second attempt, we thought of our SPI module as a separate co-processor from our CPU, where our CPU was the manager and the SPI module was the subordinate (which only added to our confusion). This led us to think that we had to instantiate a SPI module, send data from our CPU to the SPI device, and then once it's done processing the data, send the data back to the CPU. We realized this was not the right way to approach this as it would be difficult to get our two modules to communicate. If we sent data from our SPI unit back to the CPU at the wrong time, the data could easily be lost or another instruction could get interrupted.

This led us to a separate option where we dedicated a second register for our SPI module. That way, the CPU decides when to grab data, and the SPI module can store the data indefinitely.

Success:

For our third attempt, we implemented a SPI register, but we were still unsure how the Move To and Move From commands worked. It was clear that the command "MTC0 $t1, t0" would move whatever was in CPU register $t1 would go to co-processor register $t0 and "MFC0 $t2, $t0" would go from co-processor register $t0 to CPU register $t2. However, when looking at the instruction table, we didn't closely count how many bits "rd" was and subsequently thought that "rd" was 2 bits instead of 5. This amplified our confusion as we tried to logic how we could use 2 bits as a register address. We realized our error after discussing with Jon and were able to gain insight into how counting matters.

During this attempt, we also made the mistake of putting our cases to check whether it's Move From or Move To in an always at positive edge of the clock. This was not ideal since the SPI register would essentially take longer to process the sent information and to send it back to the CPU. The transfer of data needed to be instantaneous, so our data would be sent correctly. We resolved this by changing the always to an "always@*" block which worked well when transferring our data from the CPU register to the SPI register.

Implementing SPI Module

Attempt 1:

When we started this project, it made the most sense to us to start by implementing a SPI module and figuring out what to do with it later, so we did just that. The module was based on the implementation in a video by a YouTuber called Nandland, and although heavily simplified, we were able to understand how we could implement a SPI module in Verilog. However, we failed to even consider how it might interface properly with our CPU, and as a result, this implementation of a SPI module was not revisited until our move from and move to commands were successful. It would later be completely overhauled.

Attempt 2:

Once our move from and move to commands were working, we could finally revisit our code for our SPI module. It quickly became evident that our SPI module would need to be heavily adjusted to properly integrate with our newly created SPI register. We tried a slew of different approaches and eventually managed to get some semblance of a SPI module working after getting our Move To and Move From instructions working.

Attempt 2.1:

Because we already created a Verilog SPI module before, we decided to try to wire everything in our SPI module to our SPI register file. However, this did not work the way we expected, and we had to rethink about how we were going to setup the SPI module.

Attempt 2.2:

For this attempt, we focused in on getting MISO to work while still using the SPI module we had created at the start of the project and taking full advantage of all 32 registers at our disposal. By dedicating registers to store MOSI and MISO and their corresponding control signals, it was much easier to manage when to start reading data. Sending alternating 0's and 1's whenever the clock changed, allowed us to verify that our we could in fact receive data over our MISO line. We even went as far to assume that it was working after receiving "0xaaaaaaaa" in our MISO register. What we failed to notice were the unintentional delays in receiving data as well as our control registers not updating properly. These bugs in our could led us to attempt 3.

Success!

As a result of the many different changes in our previous SPI module, we opted to completely rework and reorganize our SPI code. The code at first was more verbose and less efficient in terms of lines of code, but it made it clear where things were not performing as expected. This made it possible to think of simpler control scheme that took advantage of all the additional SPI registers that were unused.

By briefly associating MOSI or MISO with their own "start" registers, we could clearly dictate when data would begin being sent or received. If the MISO start register was pulsed, then the next 32 bits coming from the subordinate were stored on MISO data register. If the MOSI start register was pulsed, then whatever was being stored in the MOSI data register would be sent out as the next 32 bits. We were able to heavily simplify our logic as a result.

Code and Schematic

See our Github Repo

The README contains info on how to run the code, a general rundown of the code, and a block diagram showing how the SPI module interfaces with the CPU. Here is the long link: https://github.com/JackMao981/CPU_with_SPI.

Next Steps

Our code heavily relies on ideal simulation conditions to operate. To improve on this code, we would:

1. Add a clock divider to decrease SPI clock speeds to more closely match real world values.

- Right now, there is no standard specification for the speed of the SPI clock. Most bus speeds are move at around 50 MHz. To alter the frequency of our signal, we could add a clock divider to match the frequency of our choosing, since SPI operating speeds are usually slower than the FPGA clock.

2. Implement the project on an FPGA to ensure it works with hardware.
   - By modifying our code to operate on an FPGA, we can ensure that our understanding of SPI is correct. It also requires us to consider noise and other factors in real world signals.
3. Add error checking to ensure data does not contain any glitches.
   - While researching, we stumbled across an article talking about 3 forms of error detection (see this link). It talks about Cyclic Redundancy Check (CRC), SCLK count error detection, and invalid read/write address error. The CRC extends a valid SPI frame by 8 cycles. The SCLK count error detection allows the user to detect if an incorrect number of SCLK cycles are sent by the microcontroller or CPU. The invalid read/write address error detects when a nonexistent register is the target for a read/write.
4. Verify that bits are being read at their centers, not edges.
   - By verifying that bits are being read at the center, we can be more confident that the data we receive will not be at a transitioning state. Additionally, the center of a bit is often the most stable

---

Looking Back

This project has been a major time commitment for the team. It provided many challenges for us because of several faulty assumptions we had made (see SPI Implementation). Looking back, most of our time was spent debugging in Verilog. It is very different from the type of programming we're used to, and the syntax is still relatively new to us. There were so many occasions where things broke because we used "always @*" instead of "always @(posedge clk)" or vice versa. However grueling the process was for the team, our team learned not only about how SPI works, but also more about the MIPS architecture and Verilog's quirks. We also became more comfortable with delegating tasks, a big change from how we worked on labs in our Computer Architecture class.

Takeaways

Audrey: I really loved this course and I felt like I really learned a lot from this project. This final project was a whirlwind of debugging, and I was able to get a better understanding of SPI, MIPS co-processes, and Verilog. In the previous labs, we were able to implement some Verilog, but it was heavily structured. So, when my team and I started working on the SPI module, we understood how it worked and could implement the module itself, but we had no idea how to test it properly.

In this project, we had to think about many other variables (clock timing, blocking/non-blocking assignments, always@ blocks, and so much more) and I was able to learn so much about Verilog and how this could be applied to different areas.

Chris: From this project, aside from learning about SPI and MIPS co-processes, my most significant takeaway was better understanding how Verilog works. I was forced to really think about clock timing and when signals should be sent unlike in structured labs. Blocking assignments and initial blocks were especially useful. Apart from verilog syntax, this project forced me to become more comfortable building tests. They're really important to test your code as its being written.

Jack: Comparch has been my first real ECE class. In this project, I really got to learn a lot more than I thought I would liked to know about the SPI protocol. I also enhanced my block-diagram-making skills and got proficient at using draw.io. To me, making a visual representation of a big and complex system makes the task at hand a bit less daunting and more understandable. I really enjoyed the process looking back at this teaming experience and learning about verilog and as a E:C major, I feel that the course gave me a much better understanding about computers at a very low level.