

CompArch Lab 2

Griffin Tschurwald, Marie-Caroline Finke, Jessica Diller

Work Plan:

Make work plan - 1 hour - October 26th - 1 hour - October 26

Input Conditioner - 45 minutes - October 30th - 1 hour - October 30th

test bench for input conditioner - 45 minutes - October 30th - 4 hours - October 25

circuit diagram - 30 minutes - October 30th - 30 min - November 8

clock timing analysis - 30 minutes - October 30th - 15 min - November 7

create shift register - 1 hour - October 31 - 1 hour - November 1

shift register test bench - 30 minutes - October 31 - 1 hour - November 1

description of test bench strategy - 30 minutes - October 31 - 30 min - November 8

create top level module and load onto FPGA - 1 hour - November 1 - 1 hour - November 1

design, implement, and write up test sequence - 1 hour - November 1 - 1 hour - November 1

show NINJA test execution - 15 minutes - November 1st - 15 min - November 2

implement SPI memory - 5 hours - November 4th - 2 hours - November 8

choose test sequences - 30 minutes - November 6th - 30 minutes - November 8

load and test on FPGA fabric - 1 hour - November 6th - 7 hours - November 8

test with ARM processor - 1 hour - November 6th - 5 hours - November 8

write up test strategy - 30 minutes - November 6th - 1 hour - November 8

plan and execute fault injection - 1 hour - November 7th - 1 hour - November 8

write up fault injection and test pattern - 1 hour - November 7th - 1 hour - November 8

write up timing analysis - NA - NA - 30min - November 8th

write up everything - 1 hour - November 8th - 1.5 hours - November 9th

Work Plan Reflection:

For the first week of the project we stayed on schedule fairly consistently although writing the test bench for the input conditioner took considerably longer than expected and this put us behind on implementation so we were up pretty late finishing everything for the mid project check-in. We also delayed everything that wasn't explicitly due for the mid project check-in until after the check-in.

Implementing the SPI took considerably less time. We planned in extra time after implementation took so long on the previous lab. We had difficulty finding times to meet that worked for everyone in the second week of the lab and so only really started working on it

Test Bench Strategy: The test bench first tests serially pulling in 8 0's and then serially pulling in 8 1's (test cases 0 and 1). During these tests we try to put in the wrong number when the peripheral clock edge is at 0. This let's the tests not only check whether serial in works when the peripheral edge is up but also makes sure that nothing is taken in when the peripheral edge is down. The second part of the tests (test case 2) looks at whether serial out works by going

through an if statement for each serial out verifying whether the bit we are outputting is the bit we want to output.

Midpoint Deliverables:

Test Sequence Description:

a short description of what the test engineer is to do and what the state of the LEDs should be at each step

Button 0: parallel load in of hex constant A5

Button 1: 4 least significant bits

Button 2: 4 Most significant bits

Switch 0: Choose bit to serially load in

Switch 1: control clock edge

1. Press Button 0 to load in hex constant A5.
2. Check the most significant bits, they should be 1010, and the least significant bits, 0101. This verifies that parallel in is working.
3. Move switch 1 up and down
4. Check the most significant bits, they should be 0100, and the least significant bits 1010. This verifies serial in 0 is working
5. Move switch 0 up, move switch 1 up and down
6. Most significant bits: 1001, least significant bits: 0101. This verifies serial in 1 is working
7. This checks all situations. To do further testing load in all 0s by moving switch 0 to low and moving switch 1 up and down 8 times. The LEDs should all be off then. The same thing can be repeated after moving switch 0 to high at which point all LEDs should be on.

Video link:

<https://www.youtube.com/watch?v=wlxaMbX-FSI>

Building our SPI Module:

Our first steps after confirming the input conditioners and shift register was working was to build the other function within the SPI module. We built in parallel: the Finite State Machine, the address latch, and the DFF with the MISO_BUFF enable switch.

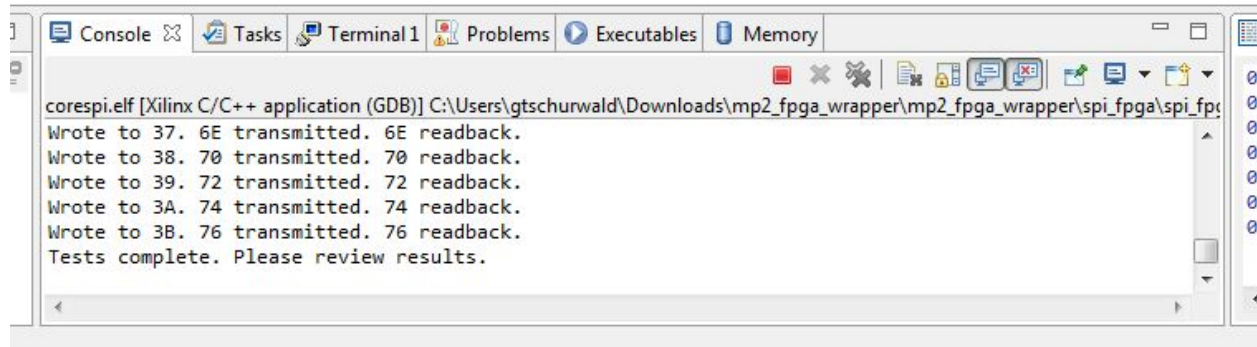
The FSM: we used case setting and the sclk positive edge cycle to change states and change outputs based on the state the FSM was in.

We then wired all the outputs and inputs together according to the block diagram given in the readme.

SPI module test strategy:

In order to test our SPI memory we decided to write a variety of values to a large number of addresses, and then read them back to see if it worked correctly. First we finished the spi_write() and spi_read() functions, then made a for loop to run all our test cases. We went

through 60 different addresses and a range of 120 different data values and all of them transmitted and read correctly. A screenshot below shows a subset of our test cases working. We ran into many Xilinx bugs during this step, but eventually worked them out by restarting our computers and vivado around 5 times.

A screenshot of a GDB console window. The window has a title bar with tabs for Console, Tasks, Terminal 1, Problems, Executables, and Memory. The main area shows the output of a program named 'corespi.elf'. The output text is as follows:

```
corespi.elf [Xilinx C/C++ application (GDB)] C:\Users\gtschurwald\Downloads\mp2_fpga_wrapper\mp2_fpga_wrapper\spi_fpga\spi_fpga.exe
Wrote to 37. 6E transmitted. 6E readback.
Wrote to 38. 70 transmitted. 70 readback.
Wrote to 39. 72 transmitted. 72 readback.
Wrote to 3A. 74 transmitted. 74 readback.
Wrote to 3B. 76 transmitted. 76 readback.
Tests complete. Please review results.
```

Fault Injection:

Fault Description:

Our injectable failure mode is that the FSM is continually reset every clk cycle regardless of what CSReset input is. This could occur if the D flip-flop that controls the reset capability was broken during manufacturing. We simulated this by adding the fault_pin being true as a condition that could start the loop within the FSM that resets the counter and state. You can test this by seeing how the FSM will stop operating and all outputs of it will remain at 0 no matter what the inputs are and no matter how long time has passed. We expected this to wholly break our SPI module, as nothing would be written to it or read from it. Unfortunately, we ran into bugs implementing the switch to inject fault testing and weren't able to see our fault in action, as we had spent many hours earlier dealing with Vivado randomly breaking.