ECSE318 Homework 4

Benjamin Scholar

CaseID: bbs27

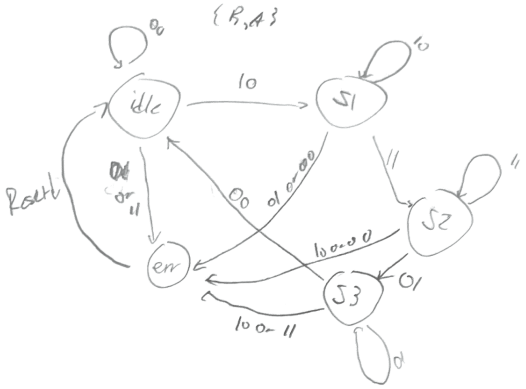
Email: scholar@case.edu

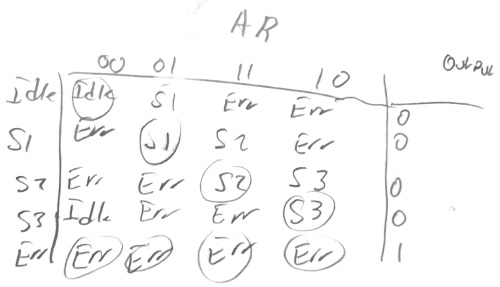
Problem 1:

Files:

- Handshake.vhd - contains state machine entity with both architectures
- Both.vhd - entity that contains both versions of the state machine and detects differences between the two state machines
- Dff.vhd - contains a D-flip-flop entity for use in the structural state machine
- Tb.v - testbench module

a) The state diagram for this state machine can be seen here:





This state machine uses 5 states, one for each input signal state plus one error state. At each state (minus the error state) the machine will remain in the current state if the input does not change and only advances to the next state if the correct state transition occurs. The correct order of input transition is as follows: R rising, A rising, R falling, A falling. If they do not occur in this exact order, the state machine goes to the error state until the RESET signal is pulled low (negative edge reset).

b) The state table can be seen here:

State \ RA	00	01	11	10	Output
Idle	Idle	S1	Err	Err	0
S1	Err	S1	S2	Err	0
S2	Err	Err	S2	S3	0
S3	Idle	Err	Err	S3	0
Err	Err	Err	Err	Err	1

And in binary representation here:

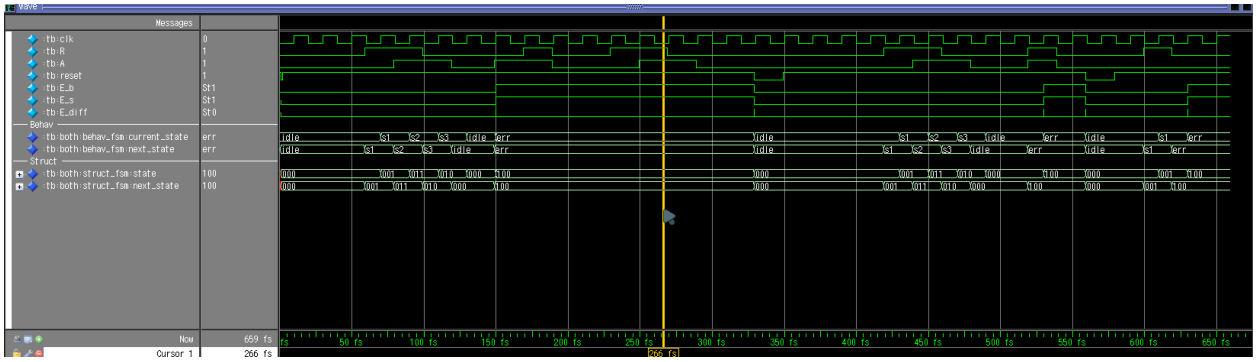
State \ RA	00	01	11	10	Output
000	000	001	100	100	0
001	100	001	011	100	0
011	100	100	011	010	0
010	000	100	100	010	0
100	100	100	100	100	1

For the structural version of this state machine, the equations for each bit of the next state were derived. These bits are then appended together to create the next state. On the negative edge of the RESET signal, the state is cleared to the idle state as specified.

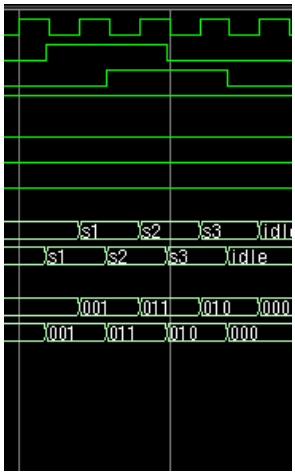
The behavioral version of the state machine is not much different, it simply implements the next state division logic as a sequence of if/else statements instead of the derived logic equations. It uses three processes, one for updating the current_state variable, one for updating the error output, and one for choosing the next state.

Both versions of the state machine are put together in the *both* entity in order to get both versions into a verilog test bench. The *both* module also generates a signal that denotes a difference between the two output signals of the handshake modules. It simply XORs the two outputs. If there is ever a difference between the two signals, this signal will go to the logic high state.

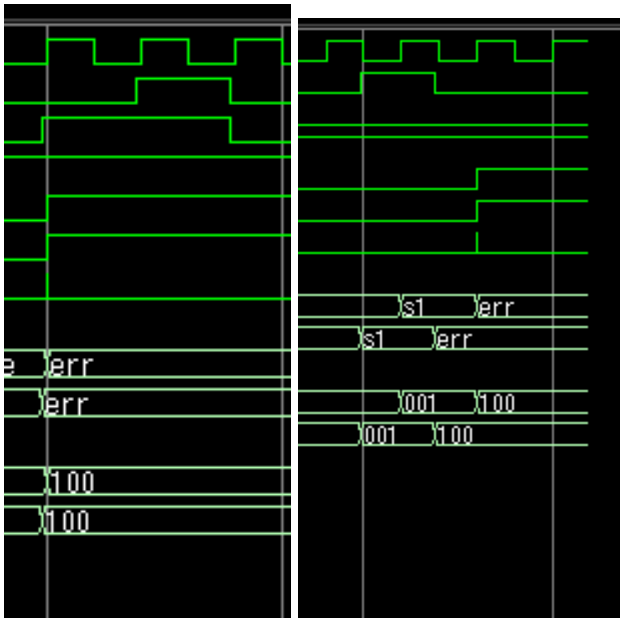
The testbench tests a number of input sequences in order to check that the state machines are correct and match with each other. The waveform output of the testbench can be seen below.



It can be seen that the output of the two versions match up exactly, thus they are functionally identical. It can also be seen that the error signals are not cleared until the reset line is pulled low.



Here we can see that the state machines function as expected when the input signals change in the correct order, as neither error output is asserted after the signals change states.



Both state machines also handle incorrect orderings correctly as well, as they immediately assert an error when one is detected. In the image on the left, the A signal goes high before the R signal, which is an error. In the picture on the right, the R signal goes high and then low again before A can be asserted.

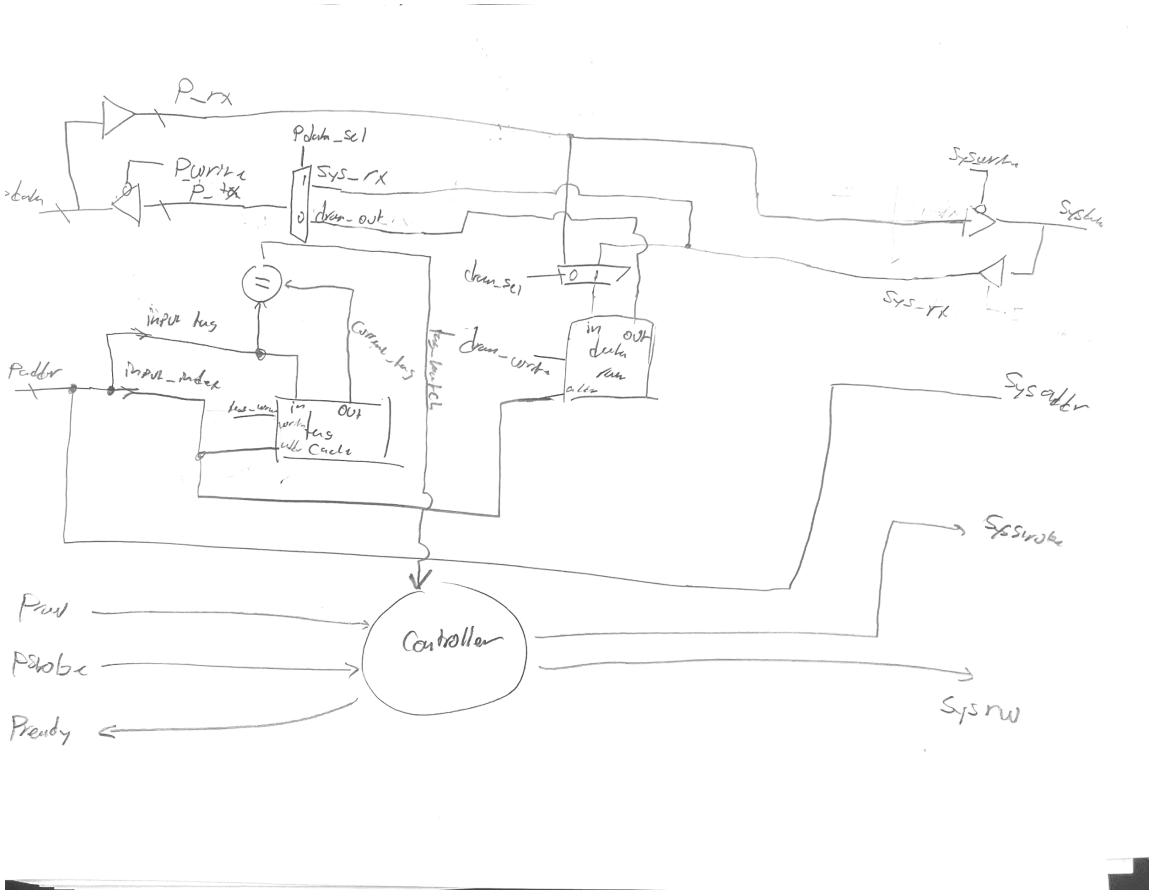
Problem 2:

Files:

- Tri_state_buffer.vhd - contains a tristate buffer with active high write enable
- Ram.vhd - generic ram entity with variable data width and address lines
- Cache_controller - contains the state machine of the cache
- Cache.vhd - puts all the elements together, contains the cache entity
- Tb.v - test bench module

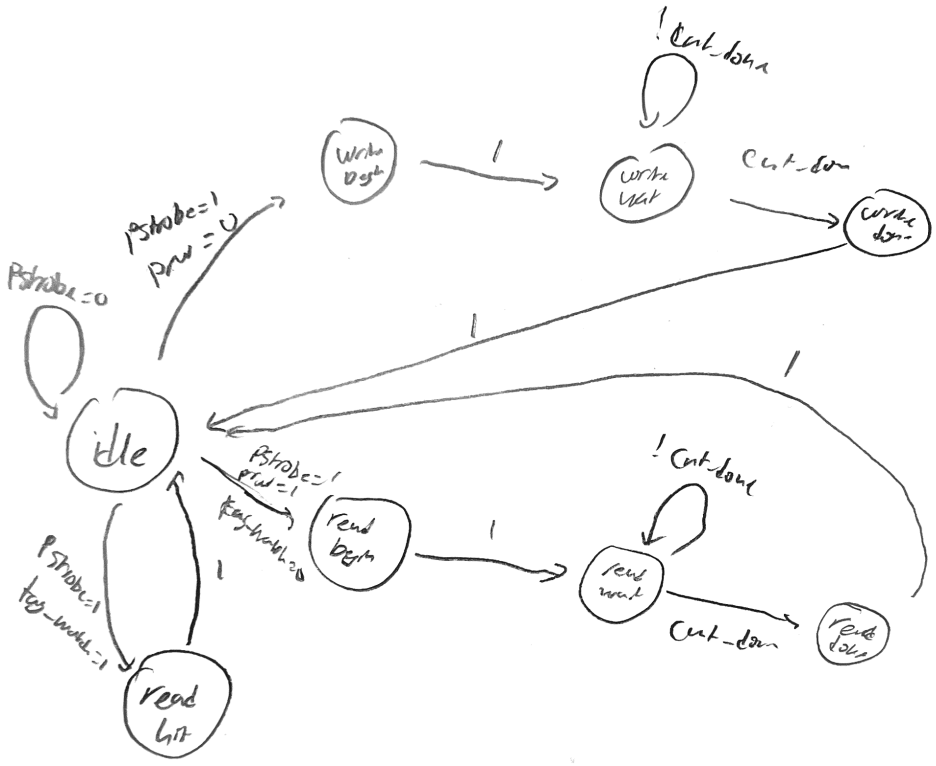
Note: If you have trouble compiling the VHDL it is because I enabled the 2008 featureset of VHDL, so that may need to be enabled for it compiles. This can be done with the "-2008" flag to the vcom compiler.

For this problem, a simple direct-mapped cache is designed in VHDL. We are instructed to create separate entities for the cache controller (state machine), the tag ram, and data ram. I chose to represent the tag and data ram entities as a generic ram entity with a variable data width and variable amount of addressing lines. The cache controller was designed as a separate entity and all resultant control signals are routed to where they need to be from the controller. For this design, I could have had the data RAM connect directly to the tri-state bus on the processor side and system side of the module, but I couldn't figure out how to "pass through" the two busses through the module, so I decided to avoid tri-state within the cache module as much as possible. A rudimentary diagram of the cache module can be found here:



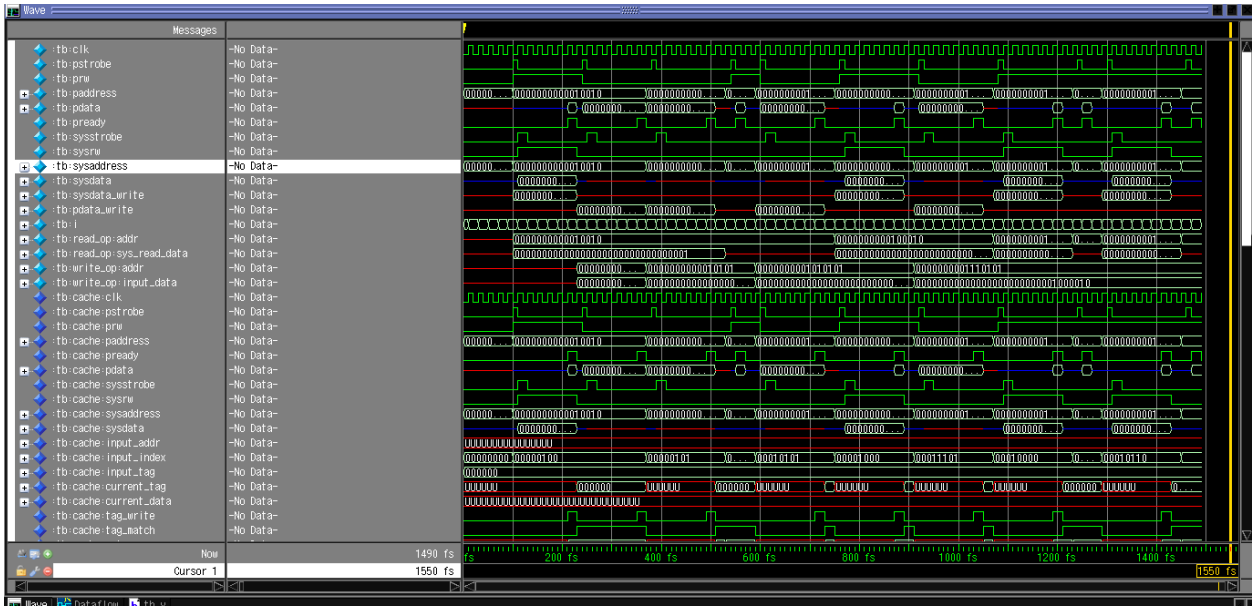
The ram module simply has an array of 2^N elements with a width of W, where N is the number of addressing lines and W is the data width. The ram module for this design is a semi-synchronous ram, so a write can occur only on a positive clock edge but a read occurs whenever the address input is changed (passively).

The cache controller was fairly straightforward and was implemented the same as any other full state machine in VHDL. I used a behavioral style to design this module in order to make it easier to generate output control signals as needed without having to compute another logic equation or K-map. A state diagram of the controller can be found here:



For the testbench of the module, I implemented two tasks, one for reading from memory and one for writing to memory in order to make it easier to generate test cases. A waveform output of the cache controller can be seen below and corresponds to the following command line output. These outputs correspond to the test case requirements from the homework assignment document. The controller appears to work for all valid inputs.

```
#    Time: 0 fs  Iteration: 1  Instance:
# data 1 read from address 0x0012
# data 69 wrote to address 0x0012
# data 42 wrote to address 0x0015
# data 69 read from address 0x0012
# data 6 wrote to address 0x0055
# data 37 read from address 0x0022
# data 66 wrote to address 0x0075
# data 1999995 read from address 0x0042
# data 66 read from address 0x0075
# data 8 read from address 0x0059
# data 37 read from address 0x0022
```

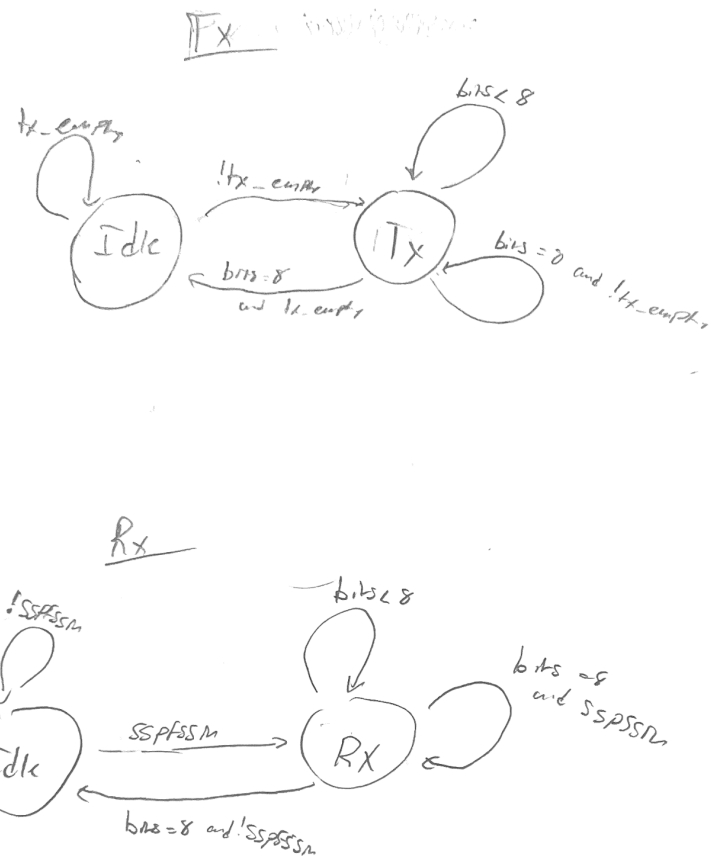

Wave
Messages
/tb/clk
/tb/pstrobe
/tb/prw
/tb/paddress
/tb/pdata
/tb/pready
/tb/sysstrobe
/tb/sysrw
/tb/sysaddress
/tb/sysdata
/tb/sysdata_write
/tb/pdata_write
/tb/i
/tb/read_op/addr
/tb/read_op/sys_read_data
/tb/write_op/addr
/tb/write_op/input_data
/tb/cache/clk
/tb/cache/pstrobe
/tb/cache/prw
/tb/cache/paddress
/tb/cache/pready
/tb/cache/pdata
/tb/cache/sysstrobe
/tb/cache/sysrw
/tb/cache/sysaddress
/tb/cache/sysdata
/tb/cache/input_addr
/tb/cache/input_index
/tb/cache/input_tag
/tb/cache/current_tag
/tb/cache/current_data
/tb/cache/tag_write
/tb/cache/tag_match
-No Data-
Now
1490 fs
Cursor 1
1550 fs
fs
200 fs
400 fs
600 fs
800 fs
1000 fs
1200 fs
1400 fs
1550 fs

Problem 3:

Files:

- Fifo.v - implements a fifo with depth and width parameters
- Rx.v - receive logic state machine
- Tx.v - transmit logic state machine
- Rxtx.v - contains both rx and tx and clock generation logic
- Sync.v - synchronizer module
- SSP.v - module containing both Fifos, sync modules, and rxtx module.
- Rx_tb.v - testbench for Rx logic
- Ssp_test.v - modified tests from course website

For this problem, we are to implement a synchronous serial port (SSP) in Verilog. I implemented the SSP with a similar structure as the diagram from the lab, with the two FIFO queues and the Tx/Rx logic being separate modules within the design. I went a step further and broke the Tx and Rx logic into separate modules in order to further break down the functionality of this circuit. Doing this allowed me to write separate test benches for both the Tx and Rx logic blocks/state machines. This made testing each of the functionalities much easier and less convoluted. Both the Tx and the Rx state machines contain 2 states: Idle and Tx/Rx respectively. The state diagrams for both Tx and Rx can be found below. Both state machines used a counter to count the number of bits sent/received per “packet.” When a state machine had sent/received all 8 bits, it would either return to the idle state or continue to send/receive the next byte/packet. A binary counter was used for this functionality.

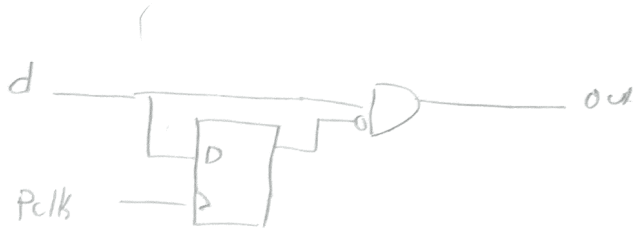


A FIFO module is used for both the Tx and Rx fifo queues. The module I designed is a synchronous FIFO queue with parameterized depth and data width. The module utilizes a push signal and a pop signal in order to determine when to add or remove data from the queue. For the Tx FIFO, the push line is controlled by the PSEL and PWRITE input, and the pop line is controlled by the Tx state machine. When a packet is done being sent, the Tx state machine pops that data from the top of the fifo. I could also potentially pop the data from the fifo before the data is sent in order to make more room in the FIFO for more data, but the I'm doing it may prevent issues if some of the data cannot be sent in full.

A simple clock generator was used in order to generate the SSPCLKOUT signal. This was done in the TxRx module, as I felt that the clock generation functionality should be kept separate from the Tx state machine itself. On every positive edge of the processor clock signal, the clock generator flips the state of the SSPCLKOUT register. This results in the desired speed

of ½ the processor clock speed. This could be done using a T flip flop if I were to design it by hand, but I'm not sure what It'll synthesize into.

One issue I encountered was caused by the fact that the processor clock was 2 times the speed of the serial in/out clocks (as specified). This was causing an issue where 2 items were being popped/pushed from/to the FIFOs instead of 1 because the pop/push line was benign held high for two processor clock cycles instead of 1. I remedied this by adding a synchronization module between the RxTx module and the push/pop lines of the FIFOs. The diagram of this circuit can be seen below. This module simply consisted of D-flip-flop and an and gate. This makes it so the push/pop lines are only asserted for 1 processor cycle instead of 1 full serial clock cycle. This solution will not work for situations where the input serial clock cycle is faster than the processor clock cycle, but a solution to this would be more complicated and may require a fully asynchronous FIFO queue.



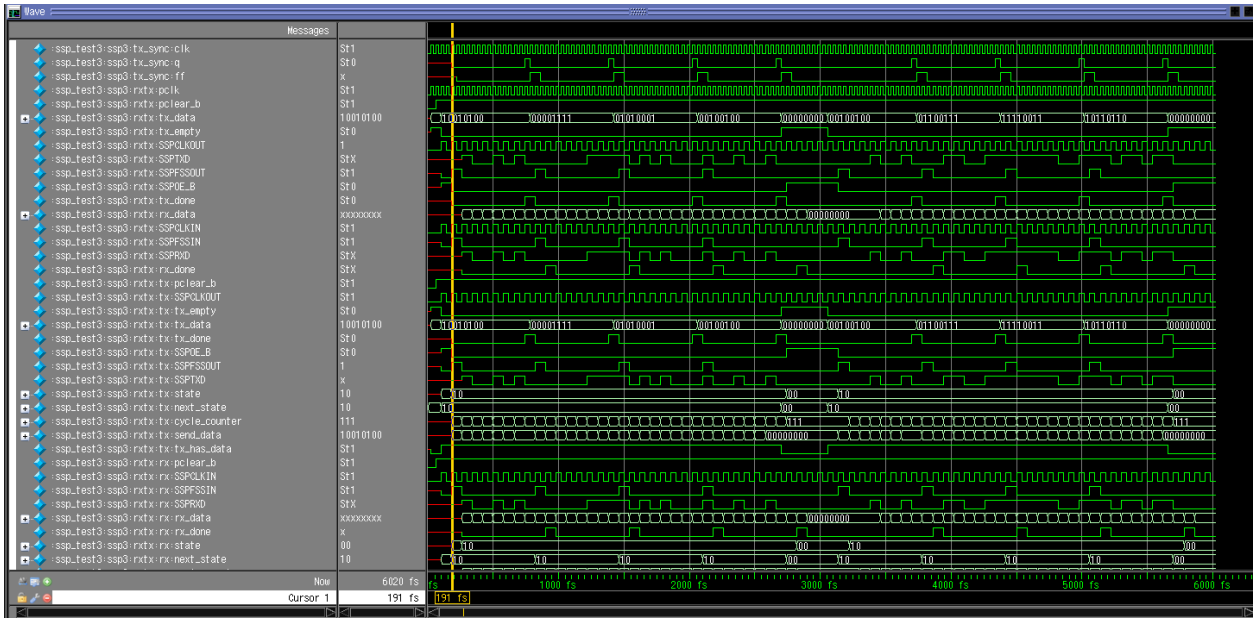
I added the following print statements to the SSP module in order to make it easier to see when data is sent/received by the Rx/Tx logic vs when the processor read/writes data to the module.

```
// For debug
always @ (posedge tx_done_sync) $display("Sent data: %08b", fifo_to_tx);
always @ (posedge rx_done_sync) $display("Recieved data: %08b", rx_to_fifo);

always @ (posedge PCLK) begin
  if(PSEL & PWRITE) $display("%08b added to TX fifo.", PWDATA);
  if(PSEL & !PWRITE) $display("%08b read from RX fifo", PRDATA);
end
```

The provided test benches were fixed using the "buf" module provided by Verilog, because nets cannot be looped back to the same module without a buffer in Verilog. I also

created my own test bench version to make the output of the module more clear. This can be found in the ssp_test.v file in the ssp_test3 module. The command line output of this module can be found here and with the waveform. All the inputs/outputs/sends/receives appear in the correct order, thus I am led to believe that the SSP module is functioning correctly.

```
VSIM 200> run -all
# 10010100 added to TX fifo.
# 00001111 added to TX fifo.
# 01010001 added to TX fifo.
# 00100100 added to TX fifo.
# Sent data: 10010100
# Recieved data: 10010100
# Sent data: 00001111
# Recieved data: 00001111
# Sent data: 01010001
# Recieved data: 01010001
# Sent data: 00100100
# Recieved data: 00100100
# 10010100 read from RX fifo
# 00001111 read from RX fifo
# 01010001 read from RX fifo
# 00100100 read from RX fifo
# 00100100 added to TX fifo.
# 01100111 added to TX fifo.
# 11110011 added to TX fifo.
# 10110110 added to TX fifo.
# Sent data: 00100100
# Recieved data: 00100100
# Sent data: 01100111
# Recieved data: 01100111
# Sent data: 11110011
# Recieved data: 11110011
# Sent data: 10110110
# Recieved data: 10110110
# 00100100 read from RX fifo
# 01100111 read from RX fifo
# 11110011 read from RX fifo
# 10110110 read from RX fifo
# Break in Module ssp_test3 at ssp_test.v line 207

VSIM 201>
```


Wave
Messages
/ssp_test3/ssp3/tx_sync/clk
/ssp_test3/ssp3/tx_sync/q
/ssp_test3/ssp3/tx_sync/ff
/ssp_test3/ssp3/rxtx/pclk
/ssp_test3/ssp3/rxtx/pclear_b
/ssp_test3/ssp3/rxtx/tx_data
/ssp_test3/ssp3/rxtx/tx_empty
/ssp_test3/ssp3/rxtx/SSPCLKOUT
/ssp_test3/ssp3/rxtx/SSPTXD
/ssp_test3/ssp3/rxtx/SSPFSSOUT
/ssp_test3/ssp3/rxtx/SSPOE_B
/ssp_test3/ssp3/rxtx/tx_done
/ssp_test3/ssp3/rxtx/rx_data
/ssp_test3/ssp3/rxtx/SSPCLKIN
/ssp_test3/ssp3/rxtx/SSPFSSIN
/ssp_test3/ssp3/rxtx/SSPRXD
/ssp_test3/ssp3/rxtx/rx_done
/ssp_test3/ssp3/rxtx/tx/pclear_b
/ssp_test3/ssp3/rxtx/tx/SSPCLKOUT
/ssp_test3/ssp3/rxtx/tx/tx_empty
/ssp_test3/ssp3/rxtx/tx/tx_data
/ssp_test3/ssp3/rxtx/tx/tx_done
/ssp_test3/ssp3/rxtx/tx/SSPOE_B
/ssp_test3/ssp3/rxtx/tx/SSPFSSOUT
/ssp_test3/ssp3/rxtx/tx/SSPTXD
/ssp_test3/ssp3/rxtx/tx/state
/ssp_test3/ssp3/rxtx/tx/next_state
/ssp_test3/ssp3/rxtx/tx/cycle_counter
/ssp_test3/ssp3/rxtx/tx/send_data
/ssp_test3/ssp3/rxtx/tx/tx_has_data
/ssp_test3/ssp3/rxtx/rx/pclear_b
/ssp_test3/ssp3/rxtx/rx/SSPCLKIN
/ssp_test3/ssp3/rxtx/rx/SSPFSSIN
/ssp_test3/ssp3/rxtx/rx/SSPRXD
/ssp_test3/ssp3/rxtx/rx/rx_data
/ssp_test3/ssp3/rxtx/rx/rx_done
/ssp_test3/ssp3/rxtx/rx/state
/ssp_test3/ssp3/rxtx/rx/next_state
Now
6020 fs
Cursor 1
191 fs
191 fs
fs
1000 fs
2000 fs
3000 fs
4000 fs
5000 fs
6000 fs