Final Project
Hardware Design - ECE 311

January 12, 2021

Partners: Jinhan Zhang
Kevin Jiang
Junpeng Lu
Ravindra Bisram
Tianshu Ren
Instructor: Professor Shlayan

Abstract

The goal of this project is to implement the FPGA as a microprocessor that can connect to peripheral devices and sensors on a self-driving mobility scooter and detect any errors when the scooter is making a 90 degree turn. The sensors to be used are an absolute encoder, two incremental encoders, and one IMU. An SPI protocol was implemented on the FPGA in order to recieve data from the sensors. A non-holonomic car model was used to generate ideal sensor numbers for a 90 degree turn

Introduction

Objective

Consider a test where the scooter is to turn 90 degrees in one direction. This should correspond to changes in the gyroscope and accelerometer in the IMU, steering motion encoded in the absolute encoder and wheel motion encoded in the incremental encoder.

Given an ideal 90 degree turn, one can consider what the sensor data should look like.

1 Implement an SPI protocol to read data from the absolute encoder.

2 Implement an SPI protocol to read gyroscope and accelerometer data from the IMU.

3 Implement the incremental encoder's protocol to read data from each incremental encoder.

4 Determine the ideal sensor behavior during a 90 degree turn in either direction. This will be parameterized by the speed, pitch of the turn, etc. and may be complicated.

5 Compare your sensor data to the ideal behavior. Present deviations for each sensor in an error signal or set of error signals.

6 If something is notably very wrong in turning, determine which device presents this error. Light a corresponding error light.

Required Resources

- Absolute Encoder, model EMS2550A from Bourns
  - Datasheet: https://www.bourns.com/docs/Product-Datasheets/EMS22A.pdf
- Inertial Measurement Unit (IMU)
  - Datasheet: https://www.st.com/resource/en/datasheet/lsm6ds3.pdf
- Incremental Encoder, from CALT
  - Datasheet: shorturl.at/mxyD4
- FPGA board, Zedboard

Implementation Outline

- To generate numbers for ideal sensor behavior during a 90 degree turn in either direction, we will use the nonholonomic car model, shown here: http://planning.cs.uiuc.edu/node658.html.
- Using this model, we can generate numbers for ideal steering angle (corresponding to the absolute encoder), ideal speed (corresponding to incremental encoder), and ideal angle between the orientation of the car and the Euclidean x and y axes (corresponding to orientation from the IMU gyroscope).
- Next we implement SPI protocol using Verilog that will allow the FPGA to read data from the absolute encoder, incremental encoder, and IMU. This will require reading the datasheets of each sensor and allocating pins on the FPGA for each sensor.
- We also need to program the FPGA to compare the sensor data to the ideal numbers. Whenever there is a deviation of the data from the ideal number by a certain amount, we can indicate which part of the car is failing with a light.

Background

SPI Research

Serial Peripheral Interface is a synchronous serial communication interface bus commonly used to transfer information between a micro-controller and various types of peripherals. SPI devices communicate in full-duplex mode using a master-slave architecture, meaning that data can be transferred simultaneously between a primary information hub and one of several secondary devices.

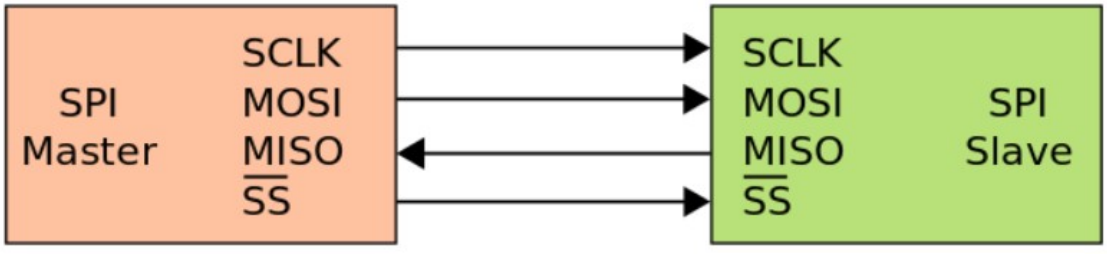

Figure 1: Basic SPI model with single slave

The device that generates the clock signal (usually called SPI CLK or SCLK for serial clock) is called the master. The master can also configure both the clock polarity (CPOL) and phase (CPHA) in accordance to the data being transferred. CPOL can either be 0, in which the clock idles at 0 and pulses 1, or 1, where the clock idles at 1 and toggles low. CPHA determines on which clock edge (trailing or leading) the MOSI (Master out - Slave in) and MISO (Master in - Slave out) lines transmit their information.

Data transmitted between the master and the slave is synchronized to the clock generated by the master. During each SPI clock cycle the master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended.

Transmissions normally involve two shift registers of some given word-size, such as eight bits (for our implementation, we did 16 bits at a time), one in the master and one in the slave; they are connected in a virtual ring topology. Data is usually shifted out with the most significant bit first. On the clock edge, both master and slave shift out a bit and output it on the transmission line to the counterpart. On the next clock edge, at each receiver the bit is sampled from the transmission line and set as a new least-significant bit of the shift register. After the register bits have been shifted out and in, the master and slave have exchanged register values. If more data needs to be exchanged, the shift registers are reloaded and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.

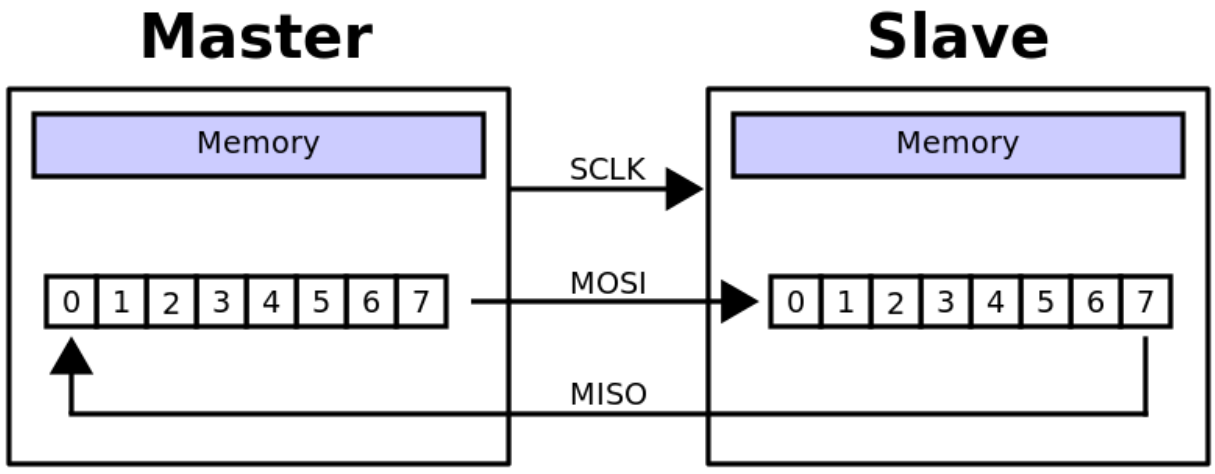

Figure 2: Transfer of data

The master is able to communicate with multiple slaves by using chip select (CS), but can only use one at a time.

Procedure

Preliminary Math Calculation For Ideal Number

Using the nonholonomic car model, with given turning radius limitation in real life we would like to generate ideal number for ideal steering angle (corresponding to the absolute encoder), ideal speed (corresponding to incremental encoder), and ideal angle between the orientation of the car and the Euclidean x and y axes (corresponding to orientation from the IMU gyroscope).

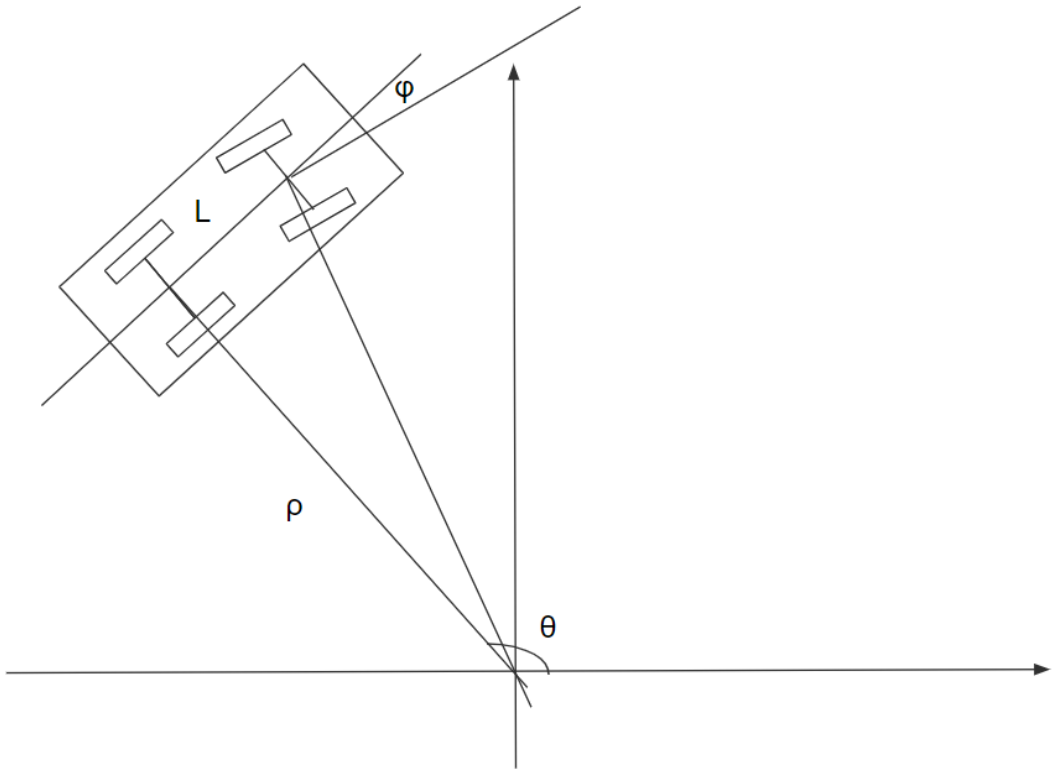

Figure 3: Car Motion Model

As seen in the Figure 3, let L denote the distance between the front and rear axles center and let the steering angle of the scooter be ϕ. L cen be measured with the actual car model. If for an ideal angle ϕ, the center of the circular motion of the car can be found by intersecting the two vertical lines of the steering direction and the car body placement direction. The intersection of these two vertical lines can be viewed as the origin of the Cartesian coordinate system and the center of the circular motion, in which we denote the radius of the circle by ρ and the tangential speed of the scooter as s. We denote the angle between the positive x-axis and the line which connects the center of the circular path and the midpoint of the rear axle of the scooter as θ. Note that so far we could represent the radius ρ as $\rho = L/tan(\phi)$, and every point (x, y) in the coordinate could be represented as $(x, y) = (\rho * cos(\theta), \rho * sin(\theta))$. Let ω be the derivative of θ over time, which is the angular frequency of the scooter, and we know that it equals the speed s over the radius ρ as represented in the equation: $\theta' = \omega = s/\rho$. Therefore, we obtain the derivative of x and y over time in terms of radius ρ and angular frequency ω as $x' = -\omega * \rho * sin(\omega * t)$ and $y' = \omega * \rho * cos(\omega * t)$. Thus, give an ideal number for the steering angle, the geometry of the circular motion is then determined. However, since in real life, we usually have hard limitation on the turning radius instead of the steering angle, we can then change the system to let the ideal number for the turning radius decide the ideal geometry of the motion. Once the the track of the circular motion is settled, a ideal value of the tangential speed of the car can be chosen based on the specs of the car.
In summary:
Give ρ, s, L:
$\phi = arctan(L/\rho)$
$(x, y) = (\rho * cos(\theta), \rho * sin(\theta))$
$\theta' = \omega = s/\rho$
$x' = -\omega * \rho * sin(\omega * t)$
$y' = \omega * \rho * cos(\omega * t)$

Programming SPI

We only need to implement a master for the SPI protocol since the slaves are the absolute encoder, the incremental encoder, and the IMU. The SPI master protocol is implemented with reference to the state machine shown in Figure 4. Our SPI master module supports the use of chip-select which is need for the slaves. In the FSM, there are 5 main states in total. By default, the state that the master is at is the idle state. In this state, the master is ready to receive a start signal that allows tells the master to start sending data to the slave through the MOSI line and receive data from the slave through the MISO line. After the start signal arrives, the master transistions into the load state where the MOSI data and other state variables are initialized to prepare for transaction. As long as there are bits left to be transimitted and received, the master will remain in the transact state where the master and slave exchange information through the MOSI and MISO lines. The master will store the MISO data in a certain register. Once there are no bits left to transmit, the master will go to the unload state which clears the state variables and prepares the data received from MISO to the output. This completes a single cycle in the SPI protocol. During any states within this process, it is possible to reset protocol through the use of a reset flag which will transition the master to the reset state where the state variables are reset and the master will return to the idle state.

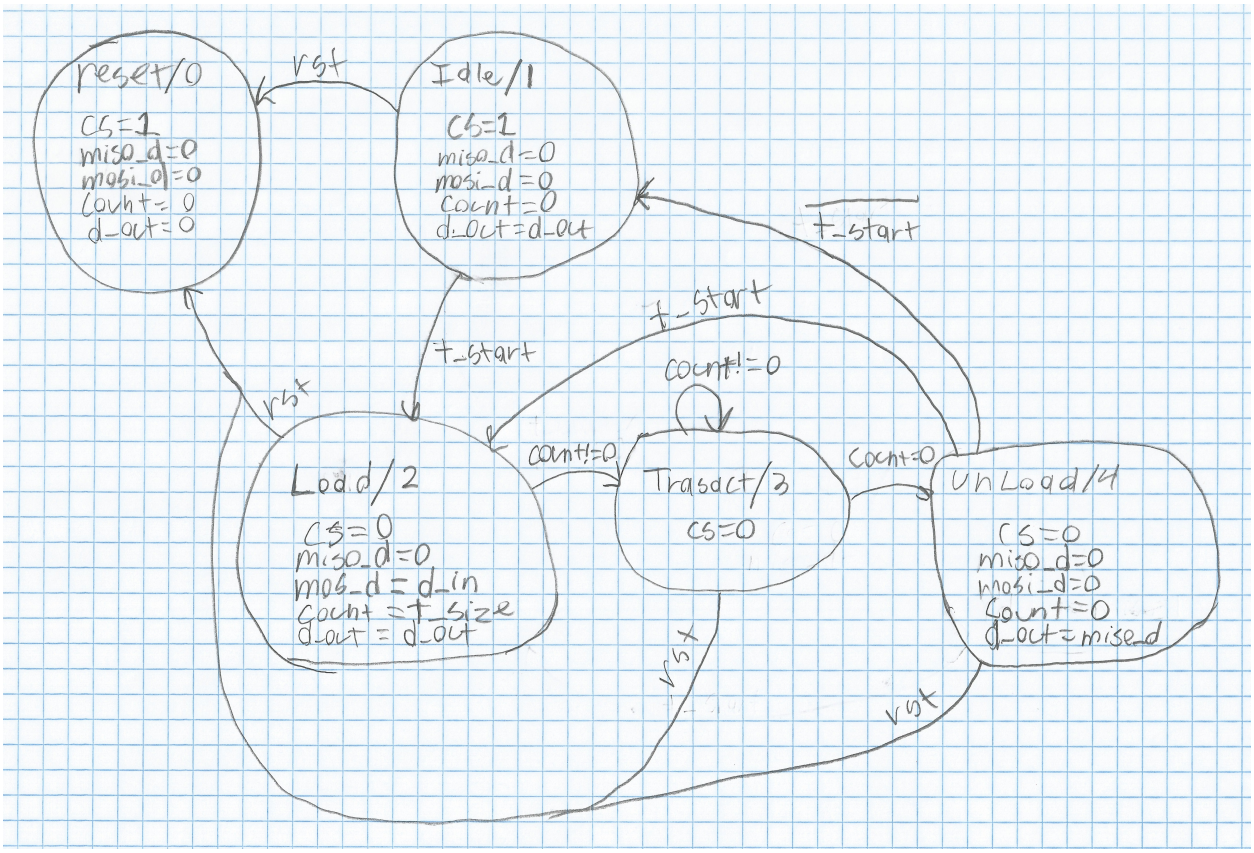

Figure 4: FSM for the SPI Master

The actual verilog code follows closely with the FSM explained above. Our SPI master module will take in the following inputs: the system clock, a reset flag, MOSI bytes to send to the slave, the start transmission signal, and the MISO data received from the slave. It will output the following: a ready signal to indicate that the master is ready for the next transaction, a data valid signal to indicate that the MISO data collected from the slave is ready to be read, the actual bytes received from MISO, the SPI clock to sychronize the master and slave, and the MOSI data to be sent to the slave. The first section in the code deals with the clocking of the entire SPI protocol. Since the only clock that we can use is the built in FPGA

clock which is set at an unchanged frequency, our module must use that clock to produce a SPI clock with suitable frequency for the slaves. Everything done during transaction is clocked based on the SPI clock not the built-in system clock. As a result, this SPI clock is only activated when a start transmission signal is received which turns on the SPI clock and allows the master and slave to start transmitting data. At the end of a transaction cycle, a ready signal is outputted from here to indicate that the protocol is ready to start another transaction. The SPI clock is outputted to the slave from the master to ensure synchronization, this is done at the end of the SPI master module.

The next few sections of the code deal with the actual transaction process of the protocol. First, the inputted MOSI data is stored in a register. Then, on each SPI clock rising edge, the most significant bit of the data is sent to the slave until all the bits have been transmitted. Similarly, on the MISO side, data is read in from the slave bit by bit from the slave. Each received bit is store in a register from the most significant to the least significant bit. Once all the bits are received, a data valid signal is outputted to indicate that the MISO data is ready to be read and processed.

In order to implement chip select, an instance of the SPI master module described earlier is created in a higher module. This higher module acts as a simple wrapper that controls when the chip select should be high or low depending on the current state of the master. As transaction happen, the chip select should be switch to low to enable the slave.

Lastly, a driver module is wrapped on top of the SPI master with chip select module which controls the inputs to the SPI master and processes the outputs. All the inputs are defined in the constraints and fed into the SPI master module accordingly. The output of the SPI are processed as follows. Whenever the ready signal is high which indicates that the protocol is ready to start a transaction, the driver module will pulse a start transmission signal which starts a single cycle of the transaction process. Whenever the data valid signal is received from the SPI master, the MISO data output is read and store in a register in the driver to be processed later. Everything done is the driver is sychronized with the built in system clock. Once the data are collected, they are to be decoded properly depending on the encoder or IMU at hand. The decoded data is then used to compute the variables during the motion of the scooter and compared with the calculated ideal values.

Hardware Implementation

To implement the design in hardware, we first initialized the system clock in the constraint. The reset is simply assigned to a switch. The connection between the FPGA and the encoders are connected through the JA pins on the side of the board where VCC and ground are provided as well. For details see the constraint file.

Results

Simulation

In our simulation, an instance of the SPI master with chip select is created. A custom clock of frequency corresponding to the frequency of the encoder is used. To simulate the SPI protocol, we transmit 2 bytes of data through the MOSI line and then attempt to received back the same data we sent out through MISO. As shown in the simulation results in Figure 5, every 2 bytes sent out is received back and all the signals such as start transmission, ready, and data valid behave properly. The simulation verilog code is shown in Figure 19 and 20 in the Appendix.

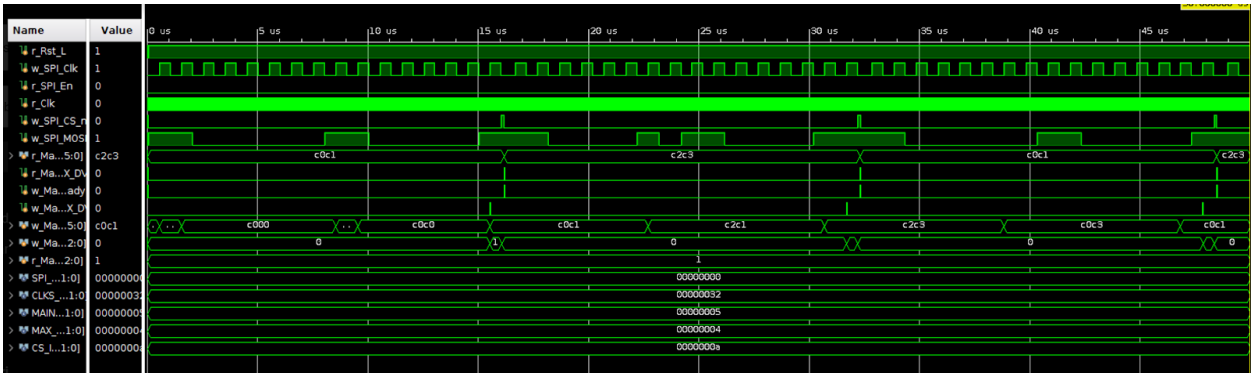

Figure 5: Simulation Results for the SPI Protocol

Synthesis and Implementation Results

The synthesis result diagram is shown in Figure 6. The project summary is shown in Figure 7. It can be seen that the IO utilization is the highest at 23.5

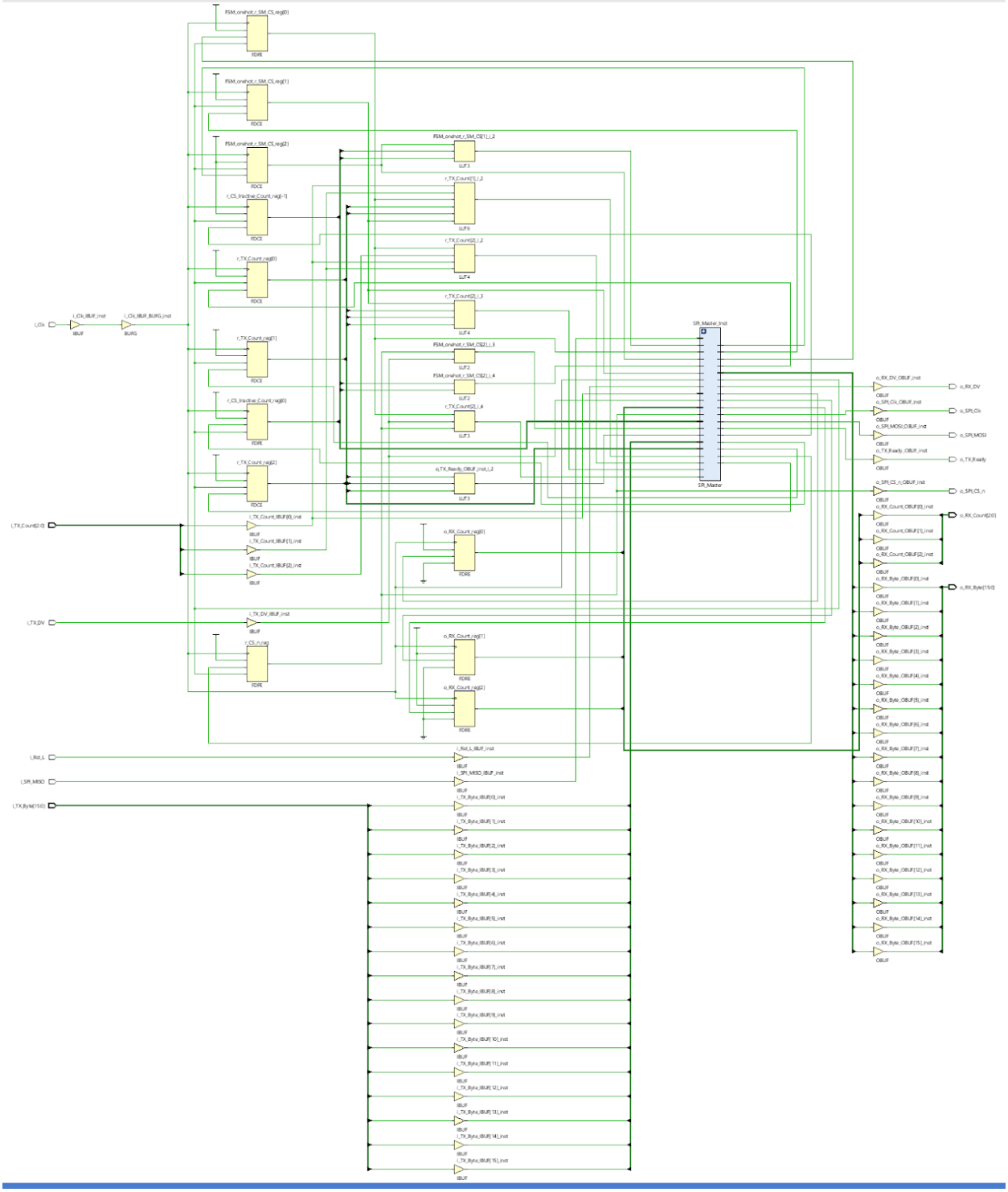

Figure 6: Synthesis Results

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	59	53200	0.11
FF	69	106400	0.06
IO	47	200	23.50
BUFG	1	32	3.13

Figure 7: Project Summary

Challenges and Further Steps

Despite our simulation working, we were not able to actually collect data from the absolute encoder. This testbench in particular assumed that the slave outputs whatever the master inputs. According to the datasheet, our absolute encoder, which is the slave, requires no master input and simply outputs 16 bits of data over 16 clock cycles after every pulse from the chip select terminal of the SPI interface. Although our clock was functional on th FPGA board, the board would not send out a chip select pulse even after clearing the logical conditions needed for the pulse. A multimeter was used to measure the voltage at each terminal of the PMOD connections on the board, and the wrong values were found. However, due to time contraints, we were unable to find the bug in our SPI interface. Our Verilog modules and constraint files can be found in the appendix.

For future students taking this classes, I believe continuing where we stopped can be a good project. So far we have completed module design, simulation, and the appropriate math models. The remaining steps are implementing the SPI for each sensor, which requires good understanding of hardware constraints and being skilled at reading datasheets.

References

For understanding SPI:
1. https://hackaday.io/project/119133-rops/log/144622-starting-with-verilog-and-spi
2. https://github.com/nandland/spi-master/blob/master/Verilog/source/SPI_Master_With_Single_CS.v
3. https://github.com/nandland/spi-master/blob/master/Verilog/source/SPI_Master.v
4. https://www.youtube.com/watch?v=TR0Pw89EuGk

Appendix

SPI Master module

```
33
34  module SPI_Master
35    #(parameter SPI_MODE = 0,
36      parameter CLKS_PER_HALF_BIT = 2)
37    (
38     // Control/Data Signals,
39     input        i_Rst_L,     // FPGA Reset
40     input        i_Clk,       // FPGA Clock
41
42     // TX (MOSI) Signals
43     input [15:0]  i_TX_Byte,        // Byte to transmit on MOSI
44     input        i_TX_DV,          // Data Valid Pulse with i_TX_Byte
45     output reg   o_TX_Ready,       // Transmit Ready for next byte
46
47     // RX (MISO) Signals
48     output reg       o_RX_DV,     // Data Valid pulse (1 clock cycle)
49     output reg [15:0] o_RX_Byte,   // Byte received on MISO
50
51     // SPI Interface
52     output reg o_SPI_Clk,
53     input      i_SPI_MISO,
54     output reg o_SPI_MOSI
55     );
56
57    // SPI Interface (All Runs at SPI Clock Domain)
58    wire w_CPOL;     // Clock polarity
59    wire w_CPHA;     // Clock phase
60
61    reg [$clog2(CLKS_PER_HALF_BIT*2)-1:0] r_SPI_Clk_Count;
62    reg r_SPI_Clk;
63    reg [5:0] r_SPI_Clk_Edges;
64    reg r_Leading_Edge;
65    reg r_Trailing_Edge;
66    reg       r_TX_DV;
67    reg [15:0] r_TX_Byte;
68
69    reg [3:0] r_RX_Bit_Count;
70    reg [3:0] r_TX_Bit_Count;
71
72    // CPOL: Clock Polarity
73    // CPOL=0 means clock idles at 0, leading edge is rising edge.
74    // CPOL=1 means clock idles at 1, leading edge is falling edge.
75    assign w_CPOL  = (SPI_MODE == 2) | (SPI_MODE == 3);
76
```

Figure 8: SPI Master module

```
77    // CPHA: Clock Phase
78    // CPHA=0 means the "out" side changes the data on trailing edge of clock
79    //              the "in" side captures data on leading edge of clock
80    // CPHA=1 means the "out" side changes the data on leading edge of clock
81    //              the "in" side captures data on the trailing edge of clock
82    assign w_CPHA  = (SPI_MODE == 1) | (SPI_MODE == 3);
83
84
85
86    // Purpose: Generate SPI Clock correct number of times when DV pulse comes
87    always @(posedge i_Clk or negedge i_Rst_L)
88    begin
89      if (~i_Rst_L)
90      begin
91        o_TX_Ready      <= 1'b0;
92        r_SPI_Clk_Edges <= 0;
93        r_Leading_Edge  <= 1'b0;
94        r_Trailing_Edge <= 1'b0;
95        r_SPI_Clk       <= w_CPOL; // assign default state to idle state
96        r_SPI_Clk_Count <= 0;
97      end
98      else
99      begin
100
101       // Default assignments
102       r_Leading_Edge  <= 1'b0;
103       r_Trailing_Edge <= 1'b0;
104
105       if (i_TX_DV)
106       begin
107         o_TX_Ready      <= 1'b0;
108         r_SPI_Clk_Edges <= 32;  // Total # edges in one byte ALWAYS 16
109       end
110       else if (r_SPI_Clk_Edges > 0)
111       begin
112         o_TX_Ready <= 1'b0;
113
114         if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT*2-1)
115         begin
116           r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
117           r_Trailing_Edge <= 1'b1;
118           r_SPI_Clk_Count <= 0;
119           r_SPI_Clk       <= ~r_SPI_Clk;
120         end
121         else if (r_SPI_Clk_Count == CLKS_PER_HALF_BIT-1)
122         begin
```

Figure 9: SPI Master module

SPI Master with Chip Select module

Testbench

```
123             r_SPI_Clk_Edges <= r_SPI_Clk_Edges - 1;
124             r_Leading_Edge  <= 1'b1;
125             r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
126             r_SPI_Clk       <= ~r_SPI_Clk;
127           end
128           else
129           begin
130             r_SPI_Clk_Count <= r_SPI_Clk_Count + 1;
131           end
132         end
133         else
134         begin
135           o_TX_Ready <= 1'b1;
136         end
137
138
139       end // else: !if(~i_Rst_L)
140     end // always @ (posedge i_Clk or negedge i_Rst_L)
141
142
143     // Purpose: Register i_TX_Byte when Data Valid is pulsed.
144     // Keeps local storage of byte in case higher level module changes the data
145     always @(posedge i_Clk or negedge i_Rst_L)
146     begin
147       if (~i_Rst_L)
148       begin
149         r_TX_Byte <= 16'h0000;
150         r_TX_DV   <= 1'b0;
151       end
152       else
153         begin
154           r_TX_DV <= i_TX_DV; // 1 clock cycle delay
155           if (i_TX_DV)
156           begin
157             r_TX_Byte <= i_TX_Byte;
158           end
159         end // else: !if(~i_Rst_L)
160     end // always @ (posedge i_Clk or negedge i_Rst_L)
161
162
163     // Purpose: Generate MOSI data
164     // Works with both CPHA=0 and CPHA=1
165     always @(posedge i_Clk or negedge i_Rst_L)
166     begin
```

Figure 10: SPI Master module

```
166   begin
167     if (~i_Rst_L)
168     begin
169       o_SPI_MOSI     <= 1'b0;
170       r_TX_Bit_Count <= 4'b1111; // send MSb first
171     end
172     else
173     begin
174       // If ready is high, reset bit counts to default
175       if (o_TX_Ready)
176       begin
177         r_TX_Bit_Count <= 4'b1111;
178       end
179       // Catch the case where we start transaction and CPHA = 0
180       else if (r_TX_DV & ~w_CPHA)
181       begin
182         o_SPI_MOSI     <= r_TX_Byte[4'b1111];
183         r_TX_Bit_Count <= 4'b1110;
184       end
185       else if ((r_Leading_Edge & w_CPHA) | (r_Trailing_Edge & ~w_CPHA))
186       begin
187         r_TX_Bit_Count <= r_TX_Bit_Count - 1;
188         o_SPI_MOSI     <= r_TX_Byte[r_TX_Bit_Count];
189       end
190     end
191   end
192
193
194   // Purpose: Read in MISO data.
195   always @(posedge i_Clk or negedge i_Rst_L)
196   begin
197     if (~i_Rst_L)
198     begin
199       o_RX_Byte      <= 16'h0000;
200       o_RX_DV        <= 1'b0;
201       r_RX_Bit_Count <= 4'b1111;
202     end
203     else
204     begin
205
206       // Default Assignments
207       o_RX_DV   <= 1'b0;
208
209       if (o_TX_Ready) // Check if ready is high, if so reset bit count to default
210       begin
211         r_RX_Bit_Count <= 4'b1111;
```

Figure 11: SPI Master module

```
204     begin
205
206       // Default Assignments
207       o_RX_DV   <= 1'b0;
208
209       if (o_TX_Ready) // Check if ready is high, if so reset bit count to default
210       begin
211         r_RX_Bit_Count <= 4'b1111;
212       end
213       else if ((r_Leading_Edge & ~w_CPHA) | (r_Trailing_Edge & w_CPHA))
214       begin
215         o_RX_Byte[r_RX_Bit_Count] <= i_SPI_MISO;  // Sample data
216         r_RX_Bit_Count            <= r_RX_Bit_Count - 1;
217         if (r_RX_Bit_Count == 4'b0000)
218         begin
219           o_RX_DV   <= 1'b1;   // Byte done, pulse Data Valid
220         end
221       end
222     end
223   end
224
225
226   // Purpose: Add clock delay to signals for alignment.
227   always @(posedge i_Clk or negedge i_Rst_L)
228   begin
229     if (~i_Rst_L)
230     begin
231       o_SPI_Clk  <= w_CPOL;
232     end
233     else
234       begin
235         o_SPI_Clk <= r_SPI_Clk;
236       end // else: !if(~i_Rst_L)
237   end // always @ (posedge i_Clk or negedge i_Rst_L)
238
239
240 endmodule // SPI_Master
241
```

Figure 12: SPI Master module

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Description: SPI (Serial Peripheral Interface) Master
//              With single chip-select (AKA Slave Select) capability
//
//              Supports arbitrary length byte transfers.
//
//              Instantiates a SPI Master and adds single CS.
//              If multiple CS signals are needed, will need to use different
//              module, OR multiplex the CS from this at a higher level.
//
// Note:        i_Clk must be at least 2x faster than i_SPI_Clk
//
// Parameters:  SPI_MODE, can be 0, 1, 2, or 3.  See above.
//              Can be configured in one of 4 modes:
//              Mode | Clock Polarity (CPOL/CKP) | Clock Phase (CPHA)
//               0   |             0             |        0
//               1   |             0             |        1
//               2   |             1             |        0
//               3   |             1             |        1
//
//              CLKS_PER_HALF_BIT - Sets frequency of o_SPI_Clk.  o_SPI_Clk is
//              derived from i_Clk.  Set to integer number of clocks for each
//              half-bit of SPI data.  E.g. 100 MHz i_Clk, CLKS_PER_HALF_BIT = 2
//              would create o_SPI_CLK of 25 MHz.  Must be >= 2
//
//              MAX_BYTES_PER_CS - Set to the maximum number of bytes that
//              will be sent during a single CS-low pulse.
//
//              CS_INACTIVE_CLKS - Sets the amount of time in clock cycles to
//              hold the state of Chip-Selct high (inactive) before next
//              command is allowed on the line.  Useful if chip requires some
//              time when CS is high between trasnfers.
////////////////////////////////////////////////////////////////////////////////
```

Figure 13: SPI Master with Chip Select module

```
36 module SPI_Master_With_Single_CS
37   #(parameter SPI_MODE = 0,
38     parameter CLKS_PER_HALF_BIT = 2,
39     parameter MAX_BYTES_PER_CS = 4,
40     parameter CS_INACTIVE_CLKS = 1)
41   (
42    // Control/Data Signals,
43    input        i_Rst_L,     // FPGA Reset
44    input        i_Clk,       // FPGA Clock
45
46    // TX (MOSI) Signals
47    input [$clog2(MAX_BYTES_PER_CS+1)-1:0] i_TX_Count,  // # bytes per CS low
48    input [15:0]  i_TX_Byte,       // Byte to transmit on MOSI
49    input        i_TX_DV,         // Data Valid Pulse with i_TX_Byte
50    output       o_TX_Ready,      // Transmit Ready for next byte
51
52    // RX (MISO) Signals
53    output reg [$clog2(MAX_BYTES_PER_CS+1)-1:0] o_RX_Count,  // Index RX byte
54    output       o_RX_DV,     // Data Valid pulse (1 clock cycle)
55    output [15:0] o_RX_Byte,   // Byte received on MISO
56
57    // SPI Interface
58    output o_SPI_Clk,
59    input  i_SPI_MISO,
60    output o_SPI_MOSI,
61    output o_SPI_CS_n
62    );
63
64   localparam IDLE        = 2'b00;
65   localparam TRANSFER    = 2'b01;
66   localparam CS_INACTIVE = 2'b10;
67
```

Figure 14: SPI Master with Chip Select module

```
68  reg [1:0] r_SM_CS;
69  reg r_CS_n;
70  reg [$clog2(CS_INACTIVE_CLKS)-1:0] r_CS_Inactive_Count;
71  reg [$clog2(MAX_BYTES_PER_CS+1)-1:0] r_TX_Count;
72  wire w_Master_Ready;
73
74  // Instantiate Master
75  SPI_Master
76    #(.SPI_MODE(SPI_MODE),
77      .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT)
78      ) SPI_Master_Inst
79   (
80   // Control/Data Signals,
81   .i_Rst_L(i_Rst_L),     // FPGA Reset
82   .i_Clk(i_Clk),         // FPGA Clock
83
84   // TX (MOSI) Signals
85   .i_TX_Byte(i_TX_Byte),         // Byte to transmit
86   .i_TX_DV(i_TX_DV),             // Data Valid Pulse
87   .o_TX_Ready(w_Master_Ready),   // Transmit Ready for Byte
88
89   // RX (MISO) Signals
90   .o_RX_DV(o_RX_DV),       // Data Valid pulse (1 clock cycle)
91   .o_RX_Byte(o_RX_Byte),   // Byte received on MISO
92
93   // SPI Interface
94   .o_SPI_Clk(o_SPI_Clk),
95   .i_SPI_MISO(i_SPI_MISO),
96   .o_SPI_MOSI(o_SPI_MOSI)
97   );
98
99
```

Figure 15: SPI Master with Chip Select module

```
100   // Purpose: Control CS line using State Machine
101   always @(posedge i_Clk or negedge i_Rst_L)
102   begin
103     if (~i_Rst_L)
104     begin
105       r_SM_CS <= IDLE;
106       r_CS_n  <= 1'b1;   // Resets to high
107       r_TX_Count <= 0;
108       r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
109     end
110     else
111     begin
112
113       case (r_SM_CS)
114       IDLE:
115         begin
116           if (r_CS_n & i_TX_DV) // Start of transmission
117           begin
118             r_TX_Count <= i_TX_Count - 1; // Register TX Count
119             r_CS_n     <= 1'b0;       // Drive CS low
120             r_SM_CS    <= TRANSFER;   // Transfer bytes
121           end
122         end
123
124       TRANSFER:
125         begin
126           // Wait until SPI is done transferring do next thing
127           if (w_Master_Ready)
128           begin
129             if (r_TX_Count > 0)
130             begin
131               if (i_TX_DV)
132               begin
133                 r_TX_Count <= r_TX_Count - 1;
134               end
135             end
136             else
```

Figure 16: SPI Master with Chip Select module

```
137             begin
138               r_CS_n  <= 1'b1; // we done, so set CS high
139               r_CS_Inactive_Count <= CS_INACTIVE_CLKS;
140               r_SM_CS             <= CS_INACTIVE;
141             end // else: !if(r_TX_Count > 0)
142           end // if (w_Master_Ready)
143         end // case: TRANSFER
144
145       CS_INACTIVE:
146         begin
147           if (r_CS_Inactive_Count > 0)
148           begin
149             r_CS_Inactive_Count <= r_CS_Inactive_Count - 1'b1;
150           end
151           else
152           begin
153             r_SM_CS <= IDLE;
154           end
155         end
156
157       default:
158         begin
159           r_CS_n  <= 1'b1; // we done, so set CS high
160           r_SM_CS <= IDLE;
161         end
162       endcase // case (r_SM_CS)
163     end
164   end // always @ (posedge i_Clk or negedge i_Rst_L)
165
166
167   // Purpose: Keep track of RX_Count
168   always @(posedge i_Clk)
169   begin
170     begin
171       if (r_CS_n)
172       begin
173         o_RX_Count <= 0;
174       end
175       else if (o_RX_DV)
176       begin
177         o_RX_Count <= o_RX_Count + 1'b1;
178       end
179     end
180   end
```

Figure 17: SPI Master with Chip Select module

```
  end

  assign o_SPI_CS_n = r_CS_n;

  assign o_TX_Ready  = ((r_SM_CS == IDLE) | (r_SM_CS == TRANSFER && w_Master_Ready == 1'b1 && r_TX_Count > 0)) & ~i_TX_DV;

endmodule // SPI_Master_With_Single_CS
```

Figure 18: SPI Master with Chip Select module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Description:       Simple test bench for SPI Master with CS module
//////////////////////////////////////////////////////////////////////////////////


module SPI_Master_With_Single_CS_TB ();

  parameter SPI_MODE = 0;           // CPOL = 1, CPHA = 1
  parameter CLKS_PER_HALF_BIT = 25;  // 6.25 MHz
  parameter MAIN_CLK_DELAY = 5;     // 25 MHz
  parameter MAX_BYTES_PER_CS = 4;   // 2 bytes per chip select
  parameter CS_INACTIVE_CLKS = 10;  // Adds delay between bytes


  logic r_Rst_L     = 1'b0;
  logic w_SPI_Clk;
  logic r_SPI_En    = 1'b0;
  logic r_Clk       = 1'b0;
  logic w_SPI_CS_n;
  logic w_SPI_MOSI;

  // Master Specific
  logic [15:0] r_Master_TX_Byte = 0;
  logic r_Master_TX_DV = 1'b0;
  logic w_Master_TX_Ready;
  logic w_Master_RX_DV;
  logic [15:0] w_Master_RX_Byte;
  logic [$clog2(MAX_BYTES_PER_CS+1)-1:0] w_Master_RX_Count, r_Master_TX_Count = 2'b10;

  // Clock Generators:
  always #(MAIN_CLK_DELAY) r_Clk = ~r_Clk;

  // Instantiate UUT
  SPI_Master_With_Single_CS
  #(.SPI_MODE(SPI_MODE),
    .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT),
    .MAX_BYTES_PER_CS(MAX_BYTES_PER_CS),
    .CS_INACTIVE_CLKS(CS_INACTIVE_CLKS)
    ) UUT
  (
   // Control/Data Signals,
   .i_Rst_L(r_Rst_L),     // FPGA Reset
   .i_Clk(r_Clk),         // FPGA Clock

   // TX (MOSI) Signals
   .i_TX_Count(r_Master_TX_Count),   // Number of bytes per CS
   .i_TX_Byte(r_Master_TX_Byte),     // Byte to transmit on MOSI
   .i_TX_DV(r_Master_TX_DV),         // Data Valid Pulse with i_TX_Byte
   .o_TX_Ready(w_Master_TX_Ready),   // Transmit Ready for Byte

   // RX (MISO) Signals
```

Figure 19: SPI Testbench

```
52      // RX (MISO) Signals
53      .o_RX_Count(w_Master_RX_Count), // Index of RX'd byte
54      .o_RX_DV(w_Master_RX_DV),       // Data Valid pulse (1 clock cycle)
55      .o_RX_Byte(w_Master_RX_Byte),   // Byte received on MISO
56
57      // SPI Interface
58      .o_SPI_Clk(w_SPI_Clk),
59      .i_SPI_MISO(w_SPI_MOSI),
60      .o_SPI_MOSI(w_SPI_MOSI),
61      .o_SPI_CS_n(w_SPI_CS_n)
62      );
63
64
65    // Sends a single byte from master.  Will drive CS on its own.
66    task SendSingleByte(input [15:0] data);
67      @(posedge r_Clk);
68      r_Master_TX_Byte <= data;
69      r_Master_TX_DV   <= 1'b1;
70      @(posedge r_Clk);
71      r_Master_TX_DV <= 1'b0;
72      @(posedge r_Clk);
73      @(posedge w_Master_TX_Ready);
74    endtask // SendSingleByte
75
76
77    initial
78      begin
79        // Required for EDA Playground
80        $dumpfile("dump.vcd");
81        $dumpvars;
82
83        repeat(1) @(posedge r_Clk);
84        r_Rst_L  = 1'b0;
85        repeat(1) @(posedge r_Clk);
86        r_Rst_L          = 1'b1;
87
88        // Test sending 2 bytes
89        SendSingleByte(16'hC0C1);
90        $display("Sent out 0xC0C1, Received 0x%X", w_Master_RX_Byte);
91        SendSingleByte(16'hC2C3);
92        $display("Sent out 0xC2C3, Received 0x%X", w_Master_RX_Byte);
93
94        repeat(200) @(posedge r_Clk);
95        $finish();
96      end // initial begin
97
98  endmodule // SPI_Master_With_Single_CS_TB
99
```

Figure 20: SPI Testbench