FPGA Servo Robot Arm

ECE 526: Digital Design with Verilog and SystemVerilog

Final Project

Written By: Aaron Joseph Nanas

Professor Orod Haghighiara

May 11, 2021

Introduction:

This project explores the basics of robot arm motion, and it shows how a simple 3-DoF (Degrees of Freedom) robot arm can be implemented with an FPGA. Several topics are applied: Pulse Width Modulation, Serial Protocol Interface, and 3D Printing. An FPGA/SoC board—the Zedboard—is used to implement the design in Verilog, and Xilinx Vivado software is used to program the board's FPGA. Two Pmods are used: one Pmod CON3 to interface the servos and to provide power from an external power source, and one PMOD JSTK2 to control the servo rotation. In addition, the board's push buttons are used to select which motor should run.

For the arm's structure, the materials are 3D printed using an Original Prusa MINI+ Printer with PLA filament. The 3D files are open-source and obtained from Arduino's Project Hub. Three 180° MG995 Servos are used, and they are rated at 4.8V - 6.0V.

Interfacing the Joystick Pmod:

1) The sample files from the Pmod Joystick reference manual are used for this design to easily interface the device. Initially, the design incremented the amount of degrees with the board's DIP switches. However, that original method proved to be inefficient when controlling the arm, as it required constantly setting the switches high or low. The better solution was then to implement a joystick for better control.
2) The joystick communicates with the Zedboard via SPI protocol with SPI Mode 0. The CS is active low. Each data byte is organized with the most significant bit first. In addition, for this design, the MOSI line has been removed, and the joystick's trigger button is not used.
3) Raw measured data from the joystick is stored at a frequency rate of 100 Hz. The design will continuously read input from the joystick when it has started. The X and Y values will range from 0 to 1023. A value of 0 or 1023 to the x-axis will correspond to the joystick being moved left or right, respectively. In the same manner, a value of 0 or 1023 to the y-axis indicates that the joystick is being moved down or up, respectively.
4) When the CS signal is set to be active (low), 15 uS (66.67kHz) must first pass before sending the first data byte. A clock divider for 66.67 kHz is then needed for the interface.

Procedure and Design Flow:

1) After the bitstream is generated, open Hardware Manager and program the device. It is important to note that the power supply must either be turned off or not in output mode before the FPGA is programmed. When the Zedboard's DONE light is active (i.e. it is blue), then the external power supply can safely output to the Pmod CON3 servo interface.
2) The Zedboard's constraints file was used to interface the Pmods used. This required looking at the board's datasheet to determine the pins for each Pmod port.
3) PWM is explored in this project. The 100 Hz clock divider module will increment COUNT by 1 until it reaches the necessary final value (which is 1000000 in this case) to create a refresh rate of 10 ms for the servos. This frequency is needed to toggle between low and high. On the other hand, the duty cycle will determine the angle at which the servo will rotate.
4) The joystick_decoder module will continuously read the joystick data input. Based on the current x and y values, the output angle value will be updated. The push buttons correspond to which servo motor should be run. All push buttons can be pressed to run all motors at once, or one at a time.
5) The formula for the angle_decoder module is taken from an Instructable, which is cited in the References section. This module will take an input angle value, and then convert it to a constant value in order to obtain the appropriate duty cycle.
6) The PWM_comparator module takes in the count value, and constant value from the angle decoder. If the count value is less than the constant than the output is 1, otherwise the output is zero. This creates the PWM signal.
7) The PWM_comparator module will take in inputs A and B. Input A will be connected to the COUNT signal of the 100 Hz clock divider. On the other hand, input B will connect to the constant value outputted by the angle_decoder module. When the COUNT is less than the constant value (A < B), the PWM will be 1; otherwise, it is set to be 0.
8) The Servo_Controller module will then make the necessary connections between the lower-level modules. This can then be instantiated multiple times in the primary controller module to add additional servos as needed.

Hierarchy of Design:

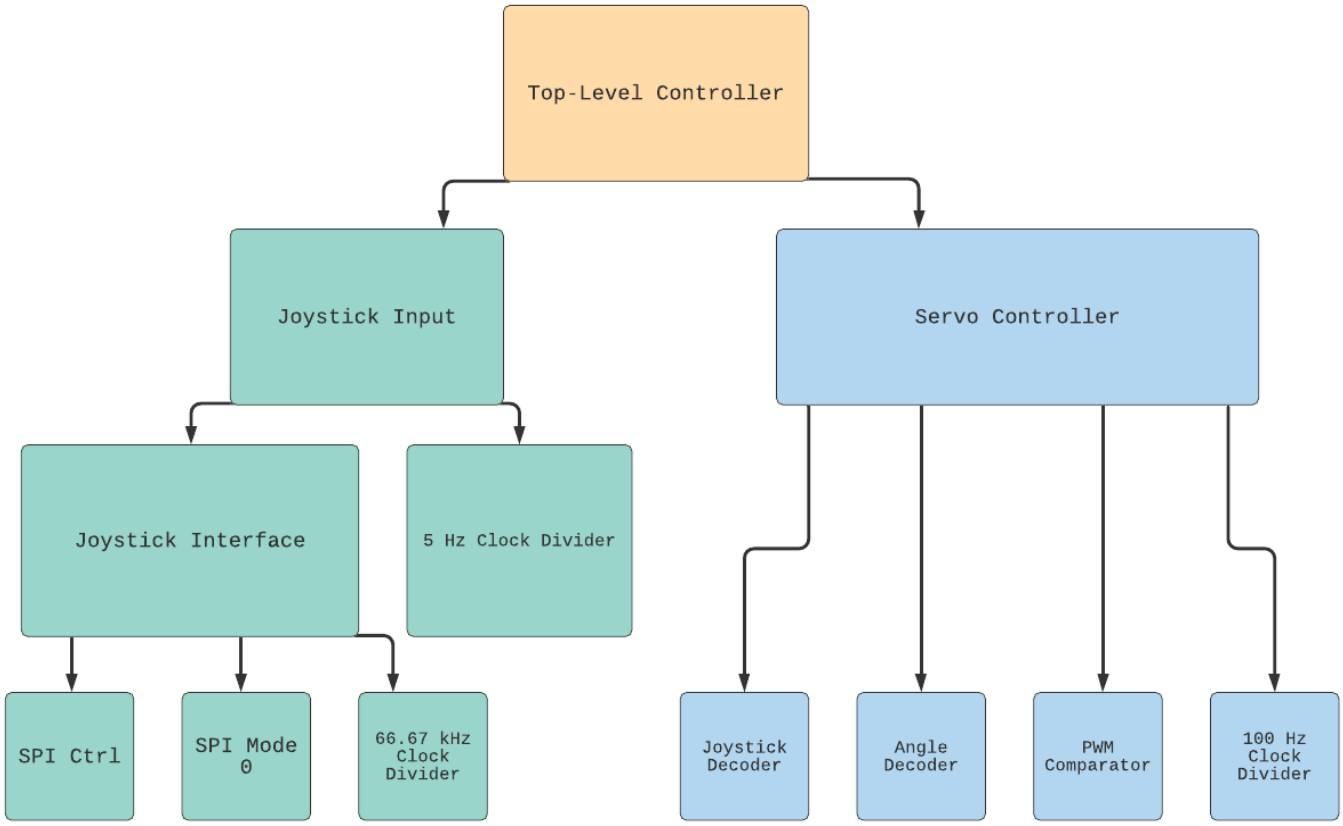

Fig. 1: Block Diagram of Module Hierarchy

SPI Ctrl FSM:

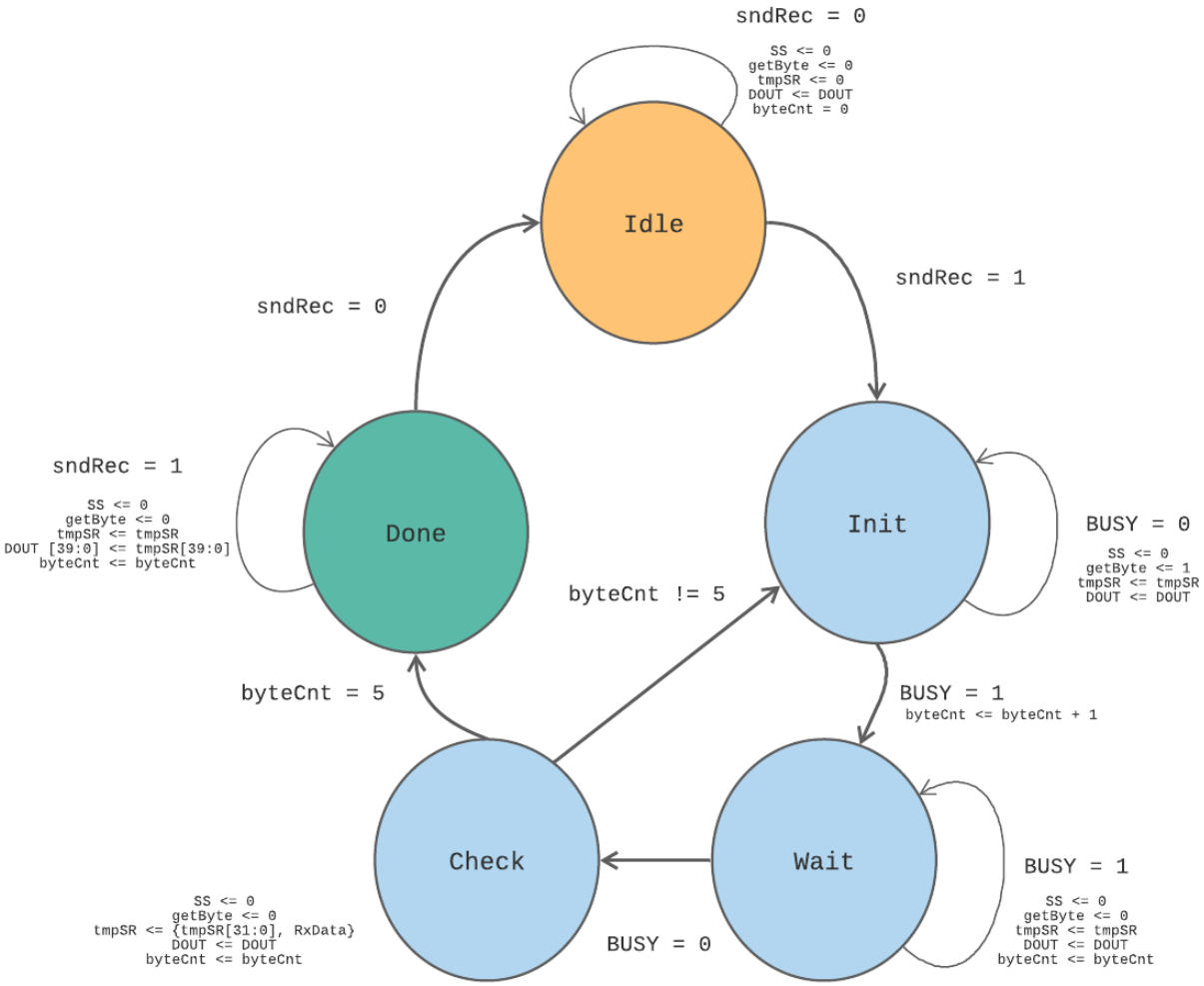

Fig. 2: SPI Ctrl FSM Diagram

SPI Mode 0 FSM:

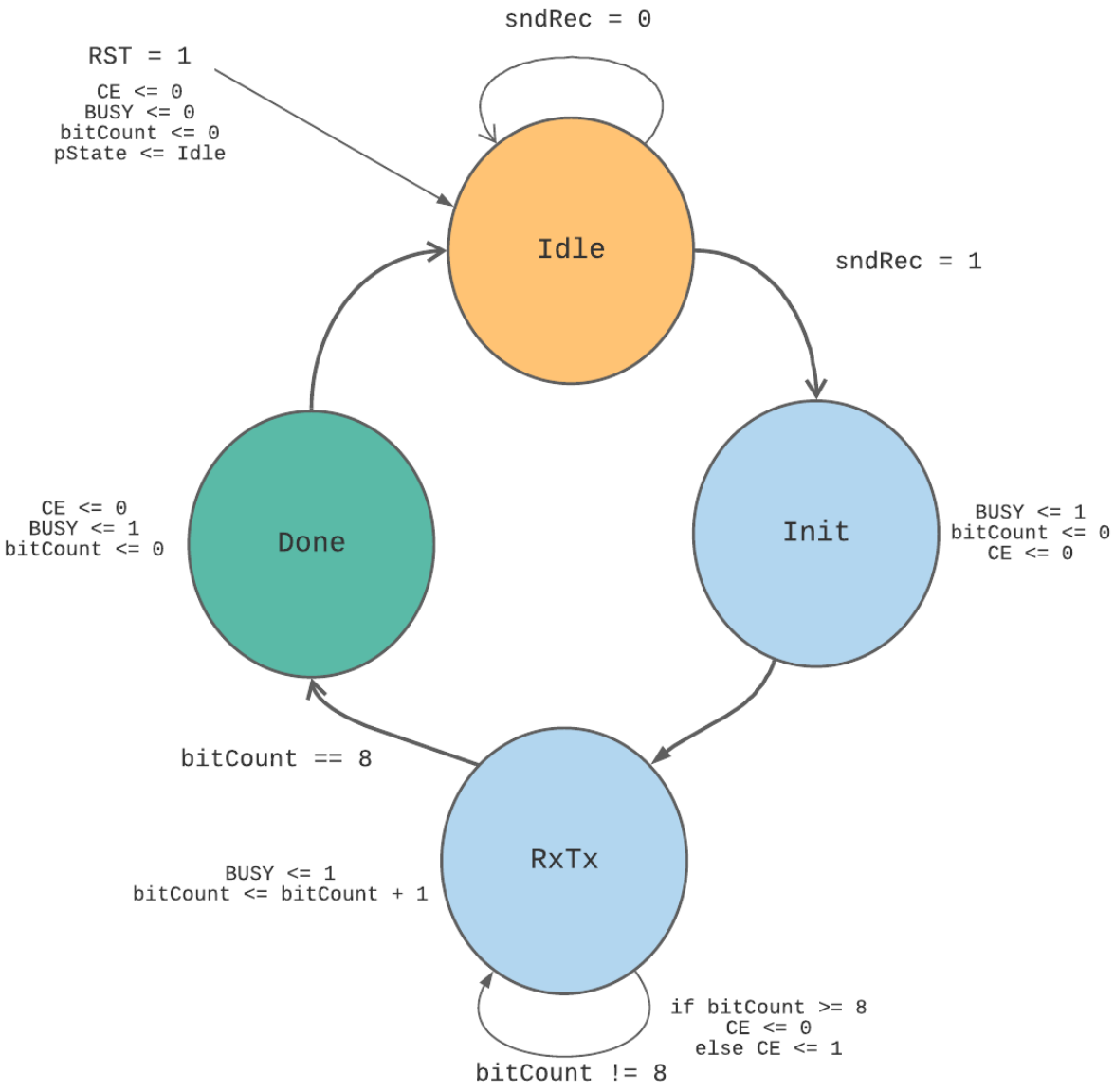

Fig. 3: SPI Mode 0 FSM Diagram

Testing and Verification:

To verify that the design can output a PWM signal, a simple testbench is made. The waveform in Figure 4 illustrates the PWM output when either one of the push buttons are pressed. Another way to verify the PWM signals is to use an oscilloscope. Figure 5 presents the waveform results on an oscilloscope of the servos. Note that in Figure 5, the frequency shown is 100 Hz.

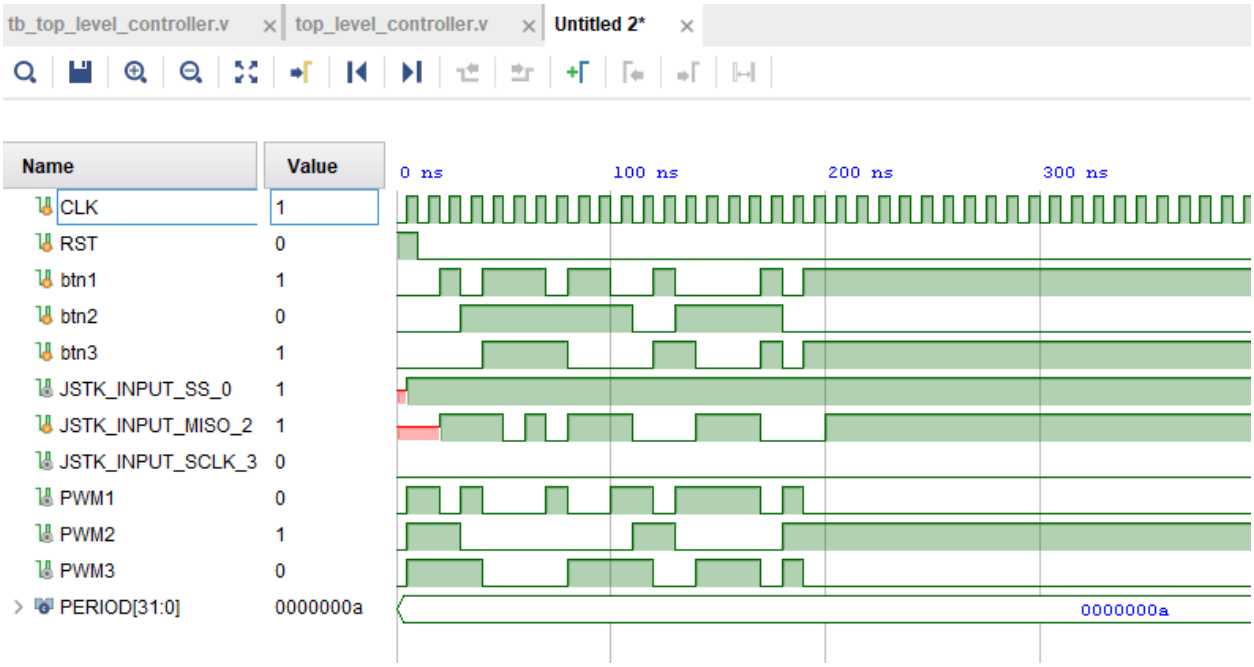

Fig. 4: Waveform Results of Top-Level Controller

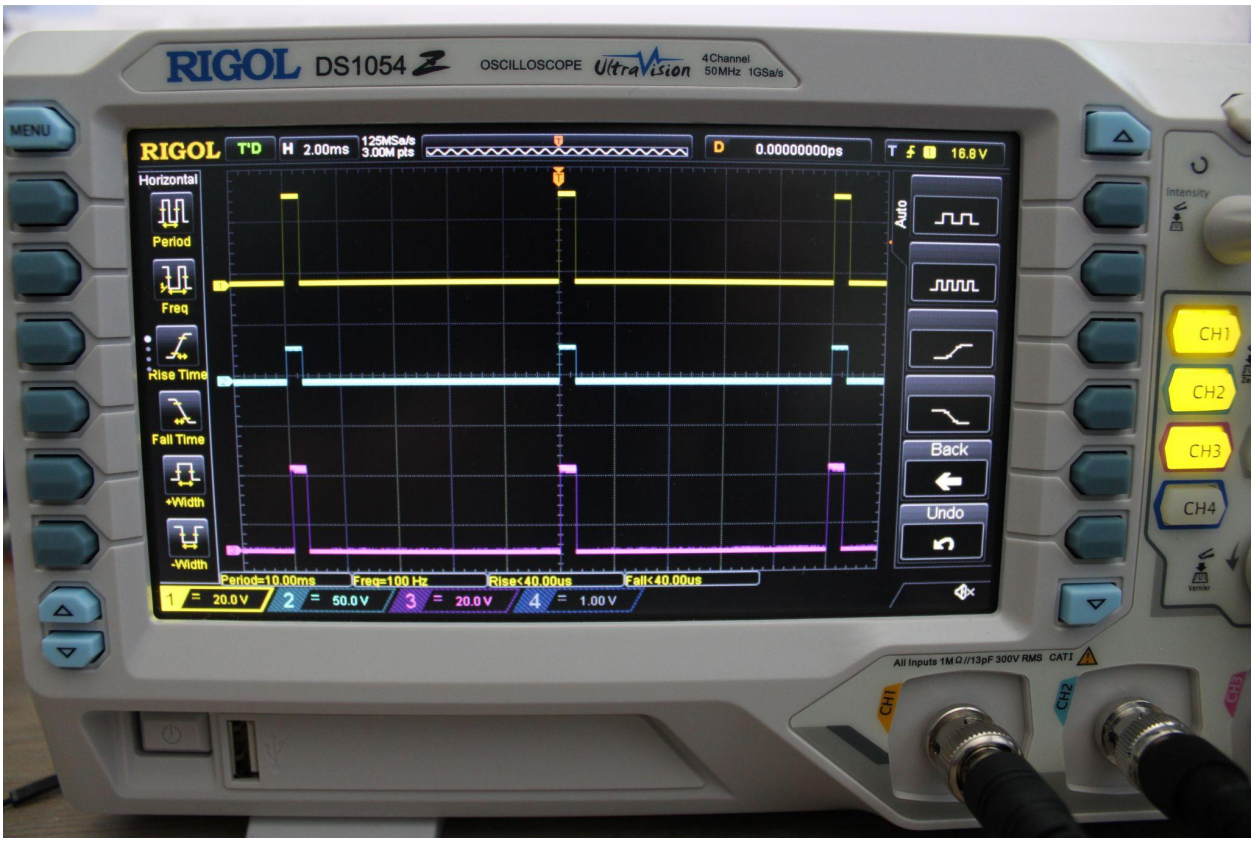

Fig. 5: PWM Results on Oscilloscope

Verilog Code

Top-Level Controller:

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: top_level_controller.v
 **
 ** Description: The top-level that serves as the main controller.
 ** This module instantiates the other lower-level servo interface
 ** modules. Currently, this controls three. Another servo module
 ** can be easily added and instantiate to control an additional servo.

 ** Note: When instantiating another servo module, be sure to
 ** update the Zedboard's constraints file to include
 ** the additional PWM signal
 **/

`timescale 1ns / 1ps

module top_level_controller(
    input CLK,
    input RST,
    input btn1, btn2, btn3,
    output JSTK_INPUT_SS_0,
    input JSTK_INPUT_MISO_2,
    output JSTK_INPUT_SCLK_3,
    output wire PWM1, PWM2, PWM3
    );

    wire [9:0] x_value_w, y_value_w;

    PmodJSTK2_Input Joystick_Input(
        .CLK(CLK),
        .RST(RST),
        .MISO(JSTK_INPUT_MISO_2),
        .SS(JSTK_INPUT_SS_0),
        .SCLK(JSTK_INPUT_SCLK_3),
        .x_value(x_value_w),
        .y_value(y_value_w)
        );
```

```
    // This servo (bottom) will rotate the base
    Servo_Controller Servo1(
        .x_value(x_value_w),
        .y_value(y_value_w),
        .RST(RST),
        .CLK(CLK),
        .btn(btn1),
        .PWM(PWM1)
         );

    // This servo (left) will move the arm forward or backward
    Servo_Controller Servo2(
        .x_value(x_value_w),
        .y_value(y_value_w),
        .RST(RST),
        .CLK(CLK),
        .btn(btn2),
        .PWM(PWM2)
        );

    // This servo (right) will lift the arm or bring it down
    Servo_Controller Servo3(
        .x_value(x_value_w),
        .y_value(y_value_w),
        .RST(RST),
        .CLK(CLK),
        .btn(btn3),
        .PWM(PWM3)
        );

endmodule
```

Top-Level Controller Lower-Level Modules

PmodJSTK2_Input

```
/** Aaron Joseph Nanas
**  ECE 526
**  Final Project: Servo-Controlled Robot Arm
**  Module: PmodJSTK2_Input.v
**  Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual

**  Description: This module serves to instantiate the Joystick interface
**  module and the 5Hz Clock Divider module. This will be used in a
**  top-level module that will serve as the primary controller
**  for both the servos and the joystick.
*/
`timescale 1ns / 1ps

module PmodJSTK2_Input(CLK, RST, MISO, SS, SCLK, x_value, y_value);

    input CLK, RST, MISO;
    output SS, SCLK;
    output [9:0] x_value;
    output [9:0] y_value;

    wire SCLK;
    wire sndRec;
    wire [39:0] JSTK_DATA;

    PmodJSTK2_Interface JSTK_Interface(.CLK(CLK), .RST(RST),
.sndRec(sndRec), .MISO(MISO), .SS(SS), .SCLK(SCLK), .DOUT(JSTK_DATA));
    Clk_Div_5Hz genSndRec(.CLK(CLK), .RST(RST), .CLKOUT(sndRec));

    assign x_value = {JSTK_DATA[9:8], JSTK_DATA[23:16]};
    assign y_value = {JSTK_DATA[25:24], JSTK_DATA[39:32]};

endmodule
```

PmodJSTK2_Interface

```
/** Aaron Joseph Nanas
**  ECE 526
**  Final Project: Servo-Controlled Robot Arm
**  Module: PmodJSTK2_Interface.v
**  Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual

**  Description: This module serves as the Pmod JSTK2 interface.
**  It consists of the 66.67khZ Serial Clock module, a SPI interface,
**  and a SPI controller. The SPI interface module sends and receives
**  bytes of data to and from the Pmod JSTK2, while the SPI controller
**  will handle all the data transfer request and the bytes of data
**  that are being transferred to the Pmod JSTK2.
*/
`timescale 1ns / 1ps

module PmodJSTK2_Interface(CLK, RST, sndRec, MISO, SS, SCLK, DOUT);

    input CLK;  // 100 MHz Clock
    input RST; // Reset
    input sndRec; // Signal that initiates read/write request
    input MISO;
    output SS; // Active low slave select signal
    output SCLK;
    output [39:0] DOUT; // Signal that stores the data from the slave

    wire SCLK;
    wire [39:0] DOUT;

    wire getByte; // Initiates the data transfer in SPI_Int
    wire [7:0] RxData; // Receive data from SPI_Int
    wire BUSY; // Handshake signal from SPI_Int to SPI_Ctrl
    wire iSCLK; // Internal Serial Clock
```

```
    // SPI Controller
    spiCtrl SPI_Ctrl(.CLK(iSCLK), .RST(RST), .sndRec(sndRec),
.BUSY(BUSY), .RxData(RxData), .SS(SS), .getByte(getByte),
.DOUT(DOUT));

    // SPI Mode 0
    SPImode0 SPI_Int(.CLK(iSCLK), .RST(RST), .sndRec(getByte),
.MISO(MISO), .SCLK(SCLK), .BUSY(BUSY), .DOUT(RxData));

    // SPI Controller Serial Clock
    Clk_Div_66_67kHz SerialClock(.CLK(CLK), .RESET(RST),
.CLK_OUT(iSCLK));

endmodule
```

spiCtrl

```
/*  Aaron Joseph Nanas
**  ECE 526:
**  Final Project: Servo-Controlled Robot Arm
**  Module: spiCtrl.v

**  Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual
**
**  Description:      This component manages all data transfer requests,
**                   and manages the data bytes being sent to the PmodJSTK.
**
**                       For more information on the contents of the bytes
being sent/received
**                              see page 2 in the PmodJSTK reference manual
found at the link provided
**                              below.
**
**
http://www.digilentinc.com/Data/Products/XUPV2P-COVERS/PmodJSTK_rm_RevC.pdf
*/

`timescale 1ns / 1ps

module spiCtrl(CLK, RST, sndRec, BUSY, RxData, SS, getByte, DOUT);
    input CLK, RST;
    input sndRec; // Send/Receive, initializes data read/write
    input BUSY; // If this is active, indicate that data transfer is
currently in progress
    input [7:0] RxData; // Last data byte received
    output reg SS; // Active low, slave select signal
    output getByte; // Initializes data trasfer
    output [39:0] DOUT; // All data read from the slave

    reg getByte = 1'b0;
    reg [39:0] DOUT = 40'h0000000000;

    // FSM States
    parameter [2:0]
```

```
        Idle = 3'd0,
        Init = 3'd1,
        Wait = 3'd2,
        Check = 3'd3,
        Done = 3'd4;

    // Present State
    reg [2:0] pState = Idle;
    reg [2:0] byteCnt = 3'd0; // Number of bits read/written
    parameter byteEndVal = 3'd5; // Number of bytes to send/receive
    reg [39:0] tmpSR = 40'h0000000000; // Temporary shift register to
accumulate all five data bytes

    always @(negedge CLK) begin
        if (RST) begin
            SS <= 1'b1; // Active low slave select signal
            getByte <= 1'b0;
            tmpSR <= 40'h0000000000;
            DOUT <= 40'h0000000000;
            byteCnt <= 3'd0;
            pState <= Idle;

        end else begin
            case(pState)
                Idle: begin
                    SS <= 1'b1; // Disable slave select
                    getByte <= 1'b0; // Data is not requested when getByte
is low
                    tmpSR <= 40'h0000000000; // Resets temporary data
                    DOUT <= DOUT; // Latch onto the output data
                    byteCnt <= 3'd0; // Clear byte count

                    // When send receive signal received, begin data
transmission
                    if (sndRec) begin
                        pState <= Init;
                    end else begin
                        pState <= Idle;
                    end
                end

                Init: begin
```

```
                    SS <= 1'b0; // Enable slave select
                    getByte <= 1'b1; // Initialize the data transfer
                    tmpSR <= tmpSR; // Latch onto the temporary data
                    DOUT <= DOUT; // Latch onto the output data

                    if (BUSY) begin
                        pState <= Wait;
                        byteCnt <= byteCnt + 1'b1; // When busy, begin
counting the incoming bits
                    end else begin
                        pState <= Init;
                    end
                end

                Wait: begin
                    SS <= 1'b0; // Enable slave select
                    getByte <= 1'b0; // Data transfer is active, so set
getByte low
                    tmpSR <= tmpSR; // Latch onto temporary data
                    DOUT <= DOUT; // Latch onto the output data
                    byteCnt <= byteCnt; // byteCnt will latch when it is in
WAIT state

                    // Finish reading byte -> get data
                    if (!BUSY) begin
                        pState <= Check;
                    end else begin
                        pState <= Wait;
                    end
                end

                Check: begin
                    SS <= 1'b0; // Enable slave select
                    getByte <= 1'b0; // Data transfer is active, so set
getByte low
                    tmpSR <= {tmpSR[31:0], RxData}; // Store the byte that
was just read
                    DOUT <= DOUT;
                    byteCnt <= byteCnt;

                    // When the 5 bytes of data are read, transition to
DONE
```

```
                    if (byteCnt == 3'd5) begin
                        pState <= Done;
                    end else begin
                        pState <= Init;
                    end
                end

                Done: begin
                    SS <= 1'b1; // Disable slave select
                    getByte <= 1'b0; // Data transfer is done
                    tmpSR <= tmpSR;
                    DOUT[39:0] <= tmpSR[39:0]; // Updates DOUT with the
temporarily stored data
                    byteCnt <= byteCnt;

                    if (!sndRec) begin
                        pState <= Idle;
                    end else begin
                        pState <= Done;
                    end
                end

                default: begin
                    pState <= Idle;
                    SS <= 1'b0;
                    DOUT <= DOUT; // Latch onto the output data
                end
            endcase
        end
    end

endmodule
```

SPImode0

```
/*  Aaron Joseph Nanas
**  ECE 526:
**  Final Project: Servo-Controlled Robot Arm
**  Module: SPImode0.v

** Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual
** Description:      This module provides the interface for sending and
receiving data
**                              to and from the PmodJSTK, SPI mode 0 is used
for communication.  The
**                              master (Nexys3) reads the data on the MISO
input on rising edges, the
**                              slave (PmodJSTK) reads the data on the MOSI
output on rising edges.
**                              Output data to the slave is changed on
falling edges, and input data
**                              from the slave changes on falling edges.
**
**                              To initialize a data transfer between the
master and the slave simply
**                              assert the sndRec input.  While the data
transfer is in progress the
**                              BUSY output is asserted to indicate to other
componenets that a data
**                              transfer is in progress.  Data to send to
the slave is input on the
**                              DIN input, and data read from the slave is
output on the DOUT output.
**
**                              Once a sndRec signal has been received a
byte of data will be sent
**                              to the PmodJSTK, and a byte will be read
from the PmodJSTK.  The
**                              data that is sent comes from the DIN input.
Received data is output
**                              on the DOUT output.
*/
```

```
`timescale 1ns / 1ps

module SPImode0(
    input CLK,                                  // 66.67kHz serial clock
    input RST,                          // Reset
    input sndRec,                      // Send receive, initializes data
read/write
    input MISO,                          // Master input slave output
    output wire SCLK,                     // Serial clock
    output reg BUSY,                     // Busy if sending/receiving data
    output wire [7:0] DOUT             // Current data byte read from the
slave
    );

    // FSM States
    parameter [1:0] Idle = 2'd0,
                    Init = 2'd1,
                    RxTx = 2'd2,
                    Done = 2'd3;

    reg [4:0] bitCount; // Number bits read/written
    reg [7:0] rSR = 8'h00; // Read shift register
    reg [1:0] pState = Idle; // Present state

    reg CE = 0; // Clock enable, controls serial clock signal sent to slave

      // Serial clock output, allow if clock enable asserted
    assign SCLK = (CE == 1'b1) ? CLK : 1'b0;
    // Master out slave in, value always stored in MSB of write shift
register
    // Connect data output bus to read shift register
    assign DOUT = rSR;

        /* Read Shift Register
        ** Master reads on rising edges,
        ** Slave changes data on falling edges
        */
        always @(posedge CLK) begin
            if (RST) begin
                rSR <= 8'h00;
```

```
            end else begin
                // Enable shift during RxTx state only
                case (pState)
                    Idle: begin
                        rSR <= rSR;
                    end

                    Init: begin
                        rSR <= rSR;
                    end

                    RxTx: begin
                        if (CE) begin
                            rSR <= {rSR[6:0], MISO};
                        end
                    end

                    Done: begin
                        rSR <= rSR;
                    end
                endcase
            end
        end

        /* SPI Mode 0 FSM */
        always @(negedge CLK) begin

            if (RST) begin
                CE <= 1'b0;
                BUSY <= 1'b0;
                bitCount <= 4'h0;
                pState <= Idle;

            end else begin

                case (pState)

                    Idle: begin
                        CE <= 1'b0;
                        BUSY <= 1'b0;
                        bitCount <= 4'h0;
```

```
                        if (sndRec) begin
                            pState <= Init;

                        end else begin
                            pState <= Idle;
                        end

                    end

                    Init: begin
                        BUSY <= 1'b1; // When initiated, it will output a
busy signal
                        bitCount <= 4'h0; // bitCount is 0 since no
Read/Write has happened yet
                        CE <= 1'b0; // Disable the serial clock
                        pState <= RxTx; // Next state, receive transmit
                    end

                    RxTx: begin
                        BUSY <= 1'b1;
                        bitCount <= bitCount + 1'b1; // Begin counting bits
received/written

                        // When all the bits to the slave are written,
prevent another falling edge
                        if (bitCount >= 4'd8) begin
                            CE <= 1'b0;
                        end else begin
                            CE <= 1'b1;
                        end

                        if (bitCount == 4'd8) begin
                            pState  <= Done;
                        end else begin
                            pState <= RxTx;
                        end
                    end

                    Done: begin
                        CE <= 1'b0; // When done, disable the serial clock
                        BUSY <= 1'b1; // Set the BUSY signal
                        bitCount <= 4'd0; // Clear the number of bits read
```

```
or written
                        pState <= Idle;
                    end

                    default: pState <= Idle;

            endcase
        end
    end

endmodule
```

Clk_Div_66_67kHz

```
/** Aaron Joseph Nanas
**  ECE 526
**  Final Project: Servo-Controlled Robot Arm
**  Module: Clk_Div_66_67kHz.v

**  Description: This module serves as a clock divider for
**  the Pmod JSTK2, and it will directly connect to the
**  internal serial clock. A 66.67kHz clock divider has a period of
**  15us. CLK_OUT will toggle between 0 and 1 every time
**  the counter reaches half of 15 us.
**
**  Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual
*/
`timescale 1ns / 1ps

module Clk_Div_66_67kHz(CLK, RESET, CLK_OUT);

    parameter COUNT_FINAL = 750;

    input CLK, RESET;
    output CLK_OUT;

    reg CLK_OUT = 1'b1; // Will connect directly to the internal serial
clock
    reg [9:0] COUNT = 0;

    always @(posedge CLK) begin
        if (RESET) begin
            CLK_OUT <= 1'b0; // Active high reset
            COUNT <= 0;

        end else begin
            if (COUNT == COUNT_FINAL) begin
                CLK_OUT <= ~CLK_OUT; // Toggle between high and low when it
reaches the final count
                COUNT <= 0; // Roll it back to 0 when it reaches the final
count of 750
```

```
            end else begin
                COUNT <= COUNT + 1'b1;
            end
        end
    end

endmodule
```

Clk_Div_5Hz

```
/*  Aaron Joseph Nanas
**  ECE 526:
**  Final Project: Servo-Controlled Robot Arm
**  Module: Clk_Div_5Hz

**  Credits to: Josh Sackos (Digilent Inc.)
**  Reference: Pmod JSTK Demo Project
**
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual

**  Description: This module will convert the onboard 100 MHz clock to 5
Hz.
*/

`timescale 1ns / 1ps

module Clk_Div_5Hz(CLK, RST, CLKOUT);
    input CLK, RST;
    output CLKOUT;

    reg CLKOUT;
    reg [23:0] COUNT = 24'd000000;

/** The period of the 100 MHz clock is 10 ns, while the period
**  of a 5 Hz frequency is 0.2 seconds.
**  If COUNT increments by 1, every 10 ns, what value must COUNT be
**  to get to 0.2 seconds? Since this clock divider is toggling between
**  0 and 1, it must be changed to 0.1.
**  Formula to find COUNT is: (100 MHz) / (1/(0.1s)) = 10000000
*/
```

```
    parameter cntEndVal = 24'd10000000;

    always @(posedge CLK) begin
        if (RST) begin
            CLKOUT <= 1'b0;
            COUNT <= 24'd000000;
        end else begin
            if (COUNT == cntEndVal) begin
                CLKOUT <= ~CLKOUT; // Toggle between 0 and 1 when the final
count value is reached
                COUNT <= 24'd000000;
            end else begin
                COUNT <= COUNT + 1'b1; // Otherwise, increment COUNT
            end
        end
    end


endmodule
```

Servo_Controller

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: Servo_Controller.v
 **
 ** Description: This module serves as the interface for a servo.
 ** Instantiating this module to a top-level module will add
 ** additional servos to the primary controller
**/

`timescale 1ns / 1ps

module Servo_Controller (x_value, y_value, RST, CLK, btn, PWM);
    input [9:0] x_value, y_value;
    input RST, CLK, btn;
    output PWM;

    // Wires needed to make connections to the
    // lower level modules
    wire [19:0] A_w, value_w;
    wire [8:0] angle_w;

    joystick_decoder joystick_read(
        .btn(btn),
        .x_value(x_value),
        .y_value(y_value),
        .angle(angle_w)
        );

    angle_decoder decode(
        .angle(angle_w),
        .value(value_w)
        );

    PWM_comparator compare_A_B(
        .A(A_w),
        .B(value_w),
        .PWM(PWM)
        );
```

```
    ClkDiv_100Hz Clk_100Hz(
        .RST(RST),
        .CLK(CLK),
        .COUNT(A_w)
        );

endmodule
```

Joystick_decoder

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: joystick_decoder.v
 **
 ** Description: This module will read the input of the joystick, and
 ** depending on which push button is pressed, the specified servo will
 ** rotate to the calculated angle value.
**/

`timescale 1ns / 1ps

module joystick_decoder(btn, x_value, y_value, angle);
    input btn;
    input [9:0] x_value;
    input [9:0] y_value;
    output reg [8:0] angle;

    reg [8:0] angle_temp;

    //
    always @(x_value, y_value, btn) begin
        // If the joystick moves right or up, update the angle to current
joystick input data
        if (x_value > 512 || y_value > 512)
            angle_temp <= (x_value + y_value)/10;
            if (btn)
                angle <= angle_temp;
            else if (!btn)
                angle <= 9'd90;
        // If the joystick moves left or down, reset angle to 0
        else if (x_value < 300 || y_value < 300)
            angle_temp <= 9'd0;

    end
endmodule
```

angle_decoder

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: angle_decoder.v
 **
 ** Description: This module will convert an angle value
 ** to a PWM constant. A servo requires a given PWM value
 ** in order to rotate to the desired degree angle
 **
 ** Note: Formula taken from:
https://blog.digilentinc.com/how-to-control-a-servo-with-fpga/
**/

`timescale 1ns / 1ps

module angle_decoder(
    input [8:0] angle,
    output reg [19:0] value
    );

    // Updates every time the angle is changed
    // This will convert the angle to a constant value,
    // and will work with continuous rotation servos as well
    always @ (angle) begin
        value = (10'd944)*(angle)+ 16'd60000;
    end
endmodule
```

PWM_comparator

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: PWM_comparator.v
 **
 ** Description: This module will output the PWM
 ** in order to drive the servos, creating a waveform.
 ** When A is less than B, the PWM output is 1.
 ** But when A is greater than B, the PWM output is 0
**/
`timescale 1ns / 1ps

module PWM_comparator (A, B, PWM);
      input [19:0] A;
      input [19:0] B;
      output reg PWM;

    // The PWM output is 1 when A < B; otherwise, it is 0
    always @ (A, B) begin
         if (A < B)
              PWM <= 1'b1;
         else
              PWM <= 1'b0;
      end
endmodule
```

ClkDiv_100Hz

```
/** Aaron Joseph Nanas
 ** ECE 526
 ** Professor Orod Haghighiara
 ** Final Project: Servo-Controlled Robot Arm
 ** Module: ClkDiv_100Hz.v
 ** Reference:
https://reference.digilentinc.com/reference/pmod/pmodjstk2/reference-manual

 ** Description: This module will create a refresh rate of 10ms.
 ** It will divide the board's 100 MHz clock to 100 Hz.
 ** According to the reference manual of the Pmod JSTK2, the raw
 ** measured data is stored at "a rate of 100 Hz Hertz (times per second)
 ** as a 16-bit right-justified variable in RAM with the upper 6 bits
masked with zeros."
**/
`timescale 1ns / 1ps

module ClkDiv_100Hz (RST, CLK, COUNT);
      input RST;
      input CLK;
      output reg [19:0] COUNT;

      always @ (posedge CLK) begin
          // The Zedboard has a system clock of 100 MHz. To obtain the
          // value needed to create a refresh rate of 10 ms, the formula
is:
          // (System Clock) / (1 / (Desired Refresh Rate))
          // (100 MHz) / (1 / (10 ms)) = 1000000
          if ((RST) || (COUNT == 20'd1000000))
                    COUNT <= 20'b0;
            else
                  COUNT <= COUNT + 1'b1;
      end
endmodule
```

Zedboard Constraints File:

```
# Clock Source - Bank 13
#
------------------------------------------------------------------------
-
set_property PACKAGE_PIN Y9 [get_ports CLK];  # "GCLK"
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
#set_property IOSTANDARD LVCMOS25 [get_ports sw]
#set_property IOSTANDARD LVCMOS25 [get_ports btn]
create_clock -period 10 [get_ports CLK]

#
------------------------------------------------------------------------
-
# JA Pmod - Bank 13
#
------------------------------------------------------------------------
-
set_property PACKAGE_PIN Y11  [get_ports {JSTK_INPUT_SS_0}];  # "JA1"
      set_property IOSTANDARD LVCMOS33 [get_ports {JSTK_INPUT_SS_0}]
#set_property PACKAGE_PIN AA8  [get_ports {JA10}];  # "JA10"
#set_property PACKAGE_PIN AA11 [get_ports {PWM2}];  # "JA2"
set_property PACKAGE_PIN Y10  [get_ports {JSTK_INPUT_MISO_2}];  # "JA3"
      set_property IOSTANDARD LVCMOS33 [get_ports {JSTK_INPUT_MISO_2}]
set_property PACKAGE_PIN AA9  [get_ports {JSTK_INPUT_SCLK_3}];  # "JA4"
      set_property IOSTANDARD LVCMOS33 [get_ports {JSTK_INPUT_SCLK_3}]
#set_property PACKAGE_PIN AB11 [get_ports {JA7}];  # "JA7"
#set_property PACKAGE_PIN AB10 [get_ports {JA8}];  # "JA8"
#set_property PACKAGE_PIN AB9  [get_ports {JA9}];  # "JA9"


#
------------------------------------------------------------------------
-
# JB Pmod - Bank 13
#
------------------------------------------------------------------------
-
set_property PACKAGE_PIN W12 [get_ports {PWM1}];  # "JB1"
      set_property IOSTANDARD LVCMOS33 [get_ports {PWM1}]
set_property PACKAGE_PIN W11 [get_ports {PWM2}];  # "JB2"
    set_property IOSTANDARD LVCMOS33 [get_ports {PWM2}]
```

```
set_property PACKAGE_PIN V10 [get_ports {PWM3}];  # "JB3"
    set_property IOSTANDARD LVCMOS33 [get_ports {PWM3}]
#set_property PACKAGE_PIN W8 [get_ports {JB4}];  # "JB4"
#set_property PACKAGE_PIN V12 [get_ports {JB7}];  # "JB7"
#set_property PACKAGE_PIN W10 [get_ports {JB8}];  # "JB8"
#set_property PACKAGE_PIN V9 [get_ports {JB9}];  # "JB9"
#set_property PACKAGE_PIN V8 [get_ports {JB10}];  # "JB10"

# User Push Buttons - Bank 34
#
------------------------------------------------------------------------
-
set_property PACKAGE_PIN P16 [get_ports {RST}];  # "BTNC"
    set_property IOSTANDARD LVCMOS33 [get_ports RST]
#set_property PACKAGE_PIN R16 [get_ports {btn4}];  # "BTND"
set_property PACKAGE_PIN N15 [get_ports {btn1}];  # "BTNL"
    set_property IOSTANDARD LVCMOS33 [get_ports btn1]
set_property PACKAGE_PIN R18 [get_ports {btn3}];  # "BTNR"
    set_property IOSTANDARD LVCMOS33 [get_ports btn3]
set_property PACKAGE_PIN T18 [get_ports {btn2}];  # "BTNU"
    set_property IOSTANDARD LVCMOS33 [get_ports btn2]
```

Testbench for Top_Level_Controller

```
`timescale 1ns / 1ps

module tb_top_level_controller();
    parameter PERIOD = 10;

    reg CLK;
    reg RST;
    reg btn1, btn2, btn3;
    wire JSTK_INPUT_SS_0;
    reg JSTK_INPUT_MISO_2;
    wire JSTK_INPUT_SCLK_3;
    wire PWM1, PWM2, PWM3;

    integer i;

    top_level_controller TOP_UUT(
        .CLK(CLK),
        .RST(RST),
        .btn1(btn1),
        .btn2(btn2),
        .btn3(btn3),
        .JSTK_INPUT_SS_0(JSTK_INPUT_SS_0),
        .JSTK_INPUT_MISO_2(JSTK_INPUT_MISO_2),
        .JSTK_INPUT_SCLK_3(JSTK_INPUT_SCLK_3),
        .PWM1(PWM1),
        .PWM2(PWM2),
        .PWM3(PWM3)
        );

    initial
        CLK = 1'b0;
    always #(PERIOD/2) CLK = ~CLK;

    initial begin
        btn1 = 0; btn2 = 0; btn3 = 0; RST = 1;

        #(PERIOD) RST = 0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn1 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn1 = 1'b0; btn2 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn3 = 1'b1; btn1 = 1'b1;
```

```
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn1 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn1 = 1'b0; btn2 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn3 = 1'b0; btn1 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn1 = 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn1 = 1'b0; btn2 = 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn3 = 1'b1; btn1 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn1 = 1'b0; btn2 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn3 = 1'b0; btn1 = 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1; btn1 = 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn1 = 1'b1; btn2 = 1'b1; btn3
= 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn1 = 1'b0; btn2 = 1'b0; btn3
= 1'b0;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b0; btn3 = 1'b1; btn1 = 1'b1;
        #(PERIOD) JSTK_INPUT_MISO_2 = 1'b1;
    end
endmodule
```

Assembly Pictures:

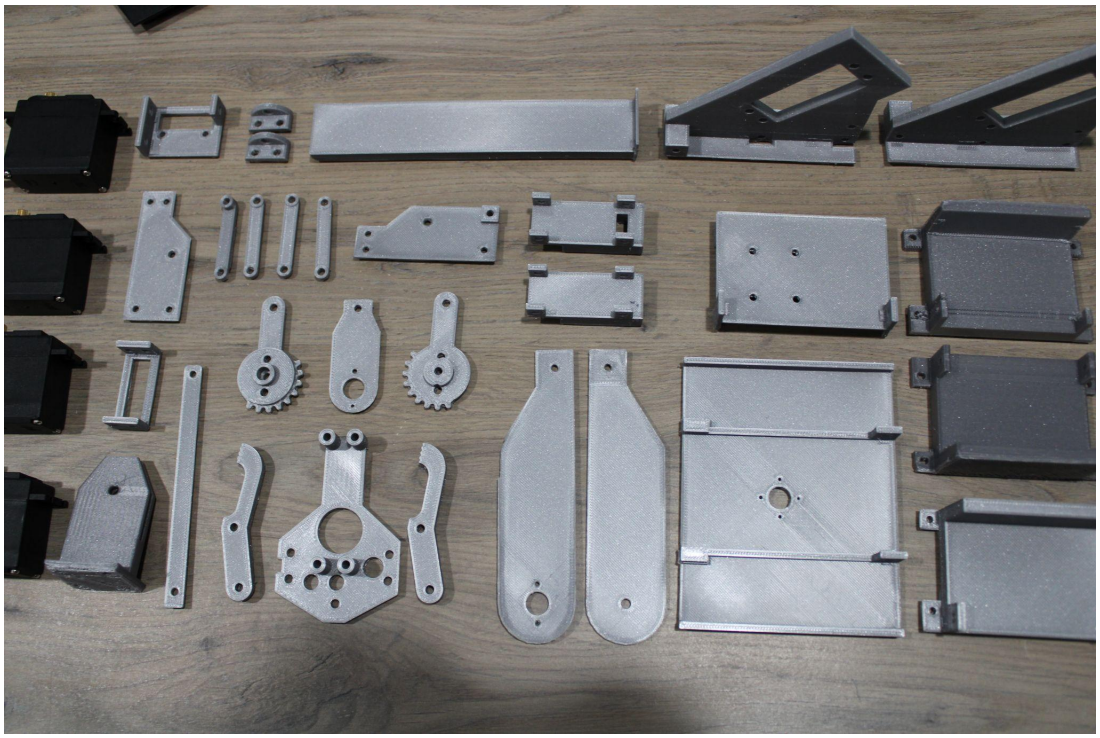

Fig. 6: Showing Servos and 3D Printed Materials Used

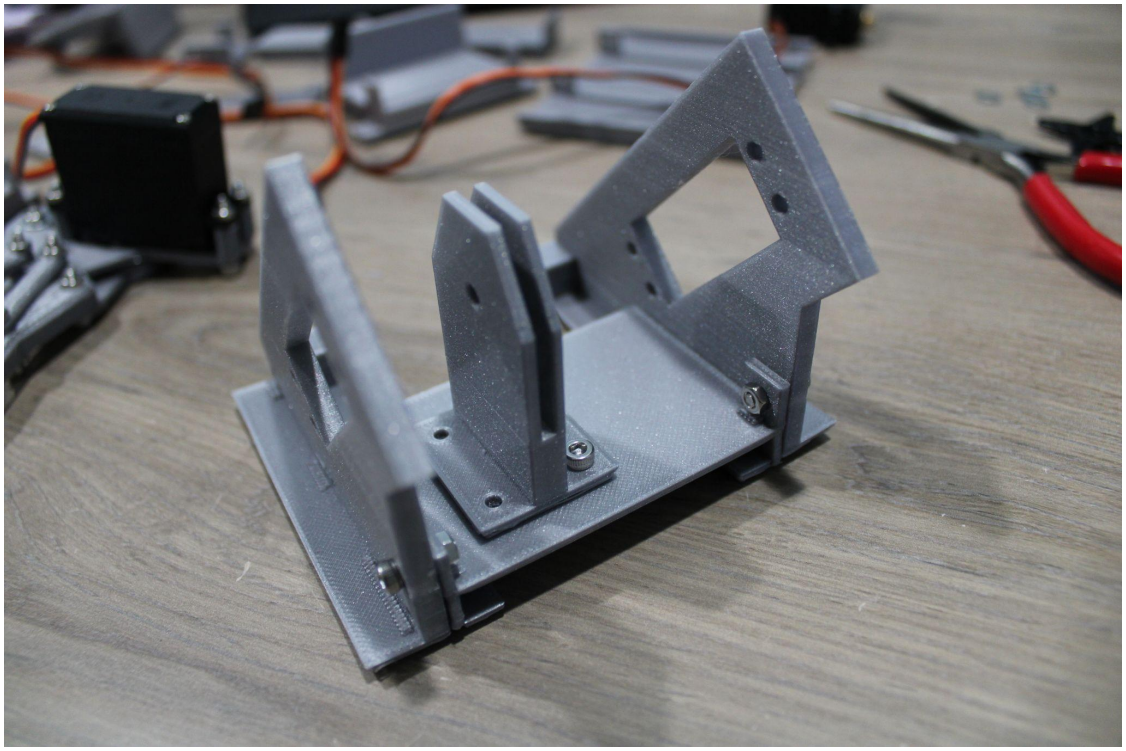

Fig. 7: Basic Foundation of Arm

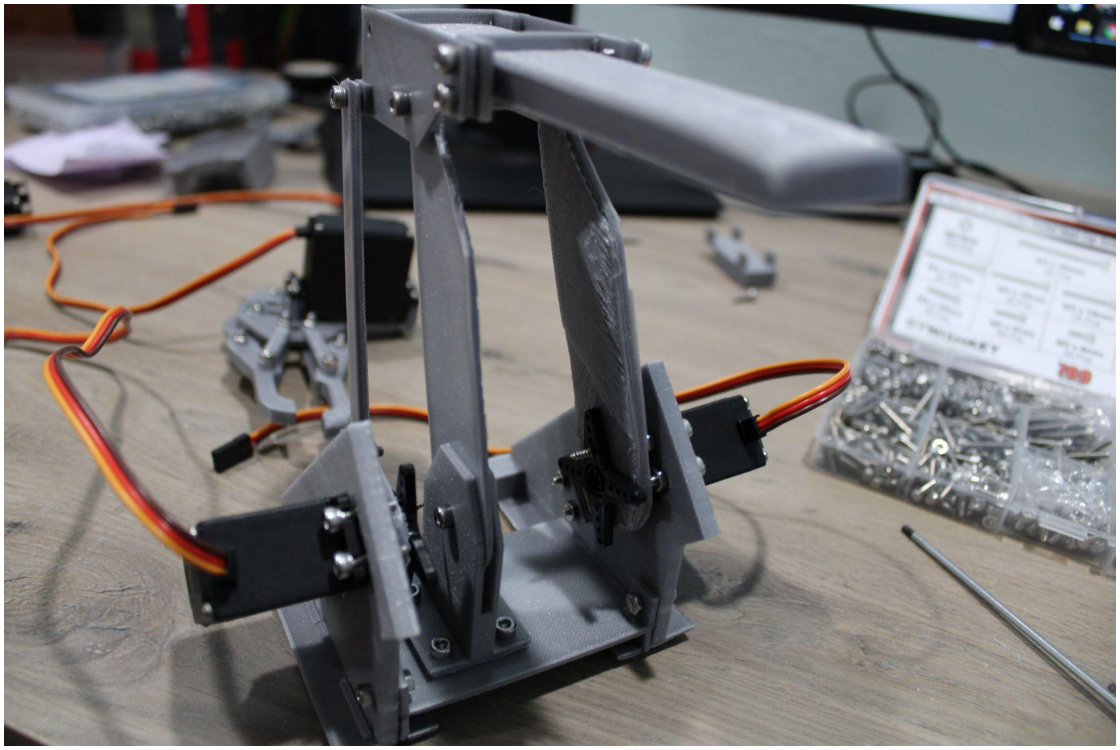

Fig. 8: Arm Without Rotating Base

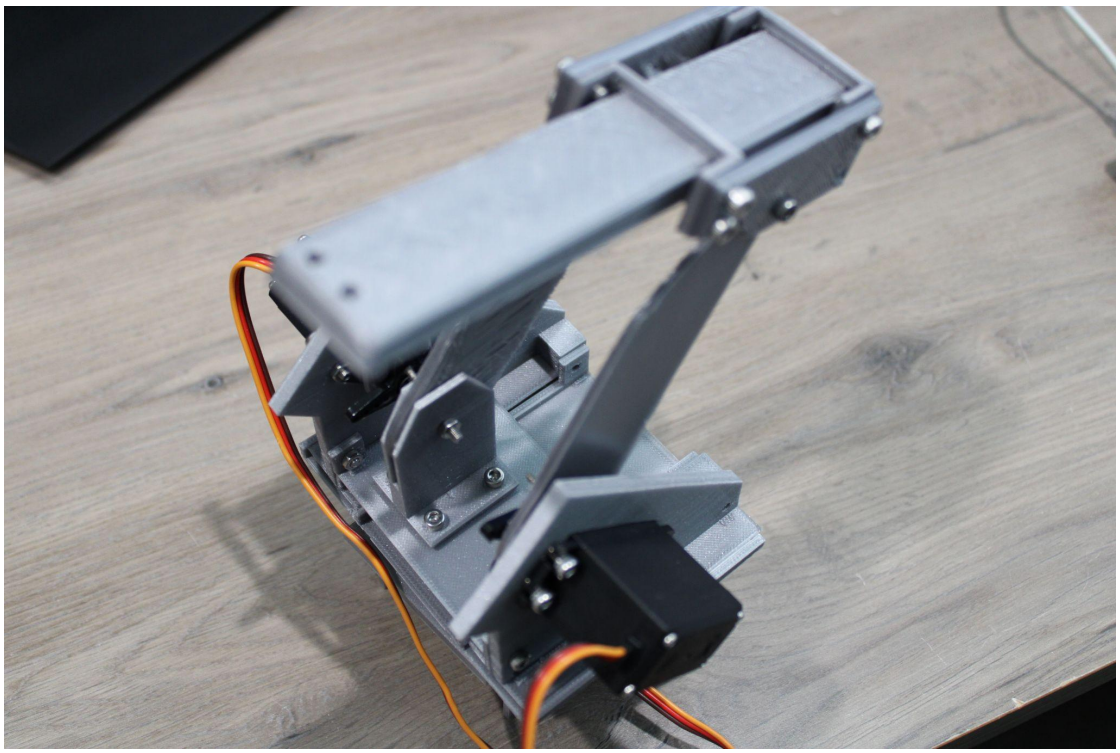

Fig. 9: Final Arm Structure

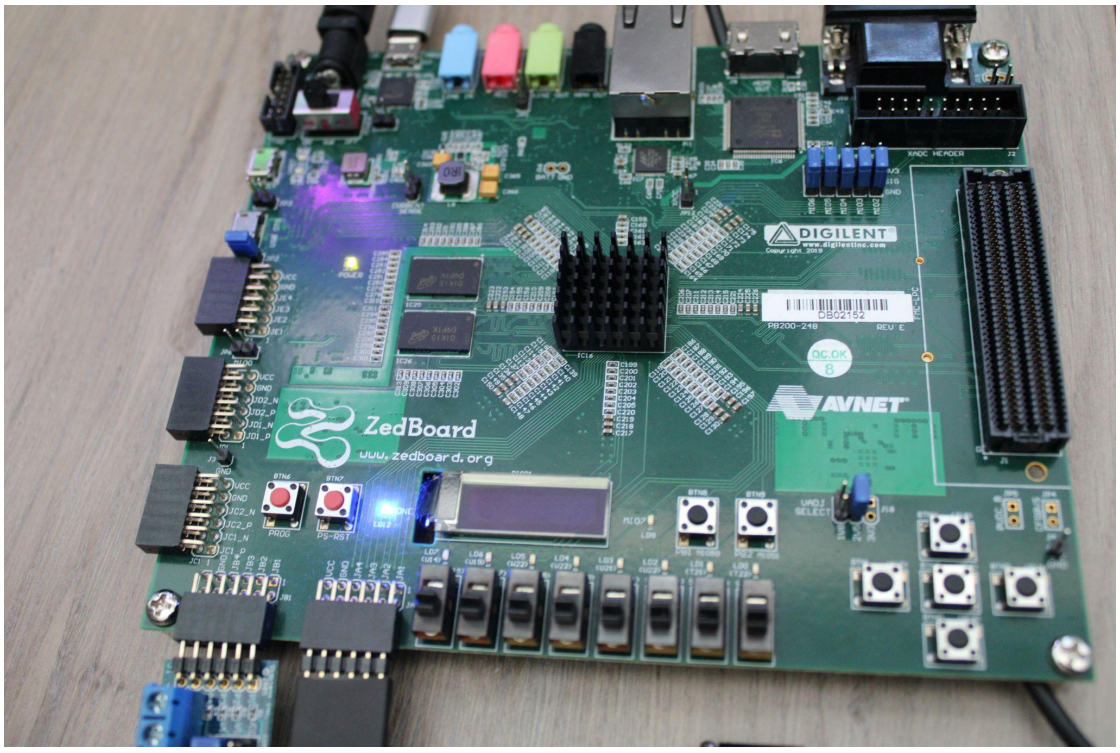

Fig. 10: Zedboard

References:

Servo Control and PWM:
https://learn.sparkfun.com/tutorials/pulse-width-modulation/all

Servo Types:
https://learn.adafruit.com/adafruit-motor-selection-guide/rc-servos

Controlling Servos with an FPGA:
https://www.instructables.com/Controlling-Servos-on-FPGA/

Open-Source 3D Print Files:
https://create.arduino.cc/projecthub/anova9347/roboarm-89b1cc

Pmod JSTK2 Reference Manual:
https://reference.digilentinc.com/reference/pmod/pmodjstk2/reference-manual

Pmod JSTK Reference Manual (with demo):
https://reference.digilentinc.com/reference/pmod/pmodjstk/reference-manual