Lab #7:

IR Remote HW

Khanh Nguyen

UIN# 525000335

ECEN 449– 503

Date: April 6, 2018

INTRODUCTION:

The purpose of this lab is to introduce us the pulse signal and how to create pulse demodulation hardware that can receive message from a TV remote. We learn to create an IR signal receiver circuit and use a customized IP module with Verilog to send signal generated from the TV remote to Zybo board.

PROCEDURE:

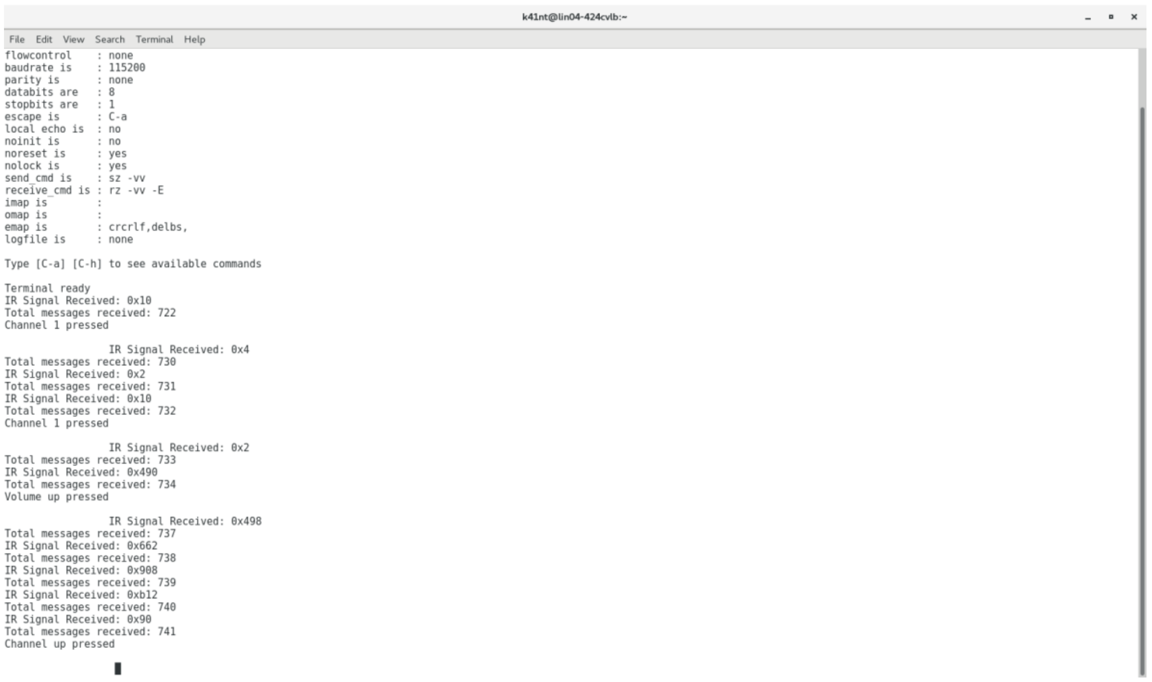
1/ The first part of the lab was to build the IR signal receiver circuit. The schematic for the circuit was provided and we used a LM339 comparator. Then we connected the circuit to the oscilloscope to observe the pulse signal generated from the TV remote. We observed the results for several buttons on the remote: volume up/down, channel up/down, stop/play, 1, 2, 3, 4.

2/ After having all the PWM codes, next step was to create a IP peripheral similar to what we did in lab 3 and lab 4:

- Created ir_demod module that with clock frequency of 75 MHz is implemented on the FPGA.
- Modified the ir_demod_v1_SOO_AXI.v Verilog template code by disabling all the AXI write capabilities and adding a new port call ‘IR_signal’
- Created user logic for demodulating with the guidelines provided in the lab manual.
- Created a new C code with ‘Hello World’ template similar to the procedures in lab 2 to test the hardware.
- Programed the FPGA board with SDK and connected the circuit from part 1 to the FPGA board.
- Observed the results from picocom terminal.

RESULT:

The results showed from Picocom terminal were identical to the PWM data we collected from part 1 of the lab:



```
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : yes
nolock is      : yes
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none

Type [C-a] [C-h] to see available commands

Terminal ready
IR Signal Received: 0x10
Total messages received: 722
Channel 1 pressed

                 IR Signal Received: 0x4
Total messages received: 730
IR Signal Received: 0x2
Total messages received: 731
IR Signal Received: 0x10
Total messages received: 732
Channel 1 pressed

                 IR Signal Received: 0x2
Total messages received: 733
IR Signal Received: 0x490
Total messages received: 734
Volume up pressed

                 IR Signal Received: 0x498
Total messages received: 737
IR Signal Received: 0x662
Total messages received: 738
IR Signal Received: 0x908
Total messages received: 739
IR Signal Received: 0xb12
Total messages received: 740
IR Signal Received: 0x90
Total messages received: 741
Channel up pressed
```

CONCLUSION:

This lab is a good review of what we did in lab 2, 3 and 4. Building the circuit and the creating the IP peripheral were pretty simple. However, coding the user logic and test software was quite demanding and took me a lot of time. There were so many error occurred during the process (syntax, logical etc.). Debugging was a very time-consuming part of this lab. Despite of some problems I had, it was a good learning experience. I learned how the Pulse Width Modulation signal works and built a system that can detect PWM signal from a TV remote.

QUESTIONS:

1/ Hexadecimal control codes for the PWM signals:

Buttons	Hexadecimal code
Volume up	0x490
Volume down	0xc90
Channel up	0x90
Channel down	0x890
Stop	0x7b0
Play	0xfb0
1	0x10
2	0x810
3	0x410
4	0xc10

2/ When a button is pressed on the remote, multiple copies of the same command message are sent. Approximately how many of the same command message are transmitted after each press of a button? Provide some intuition as to why multiple messages are sent.

About 4 or 5 times the command message was sent when a button was pressed (it depends on how long it is pressed). It is to ensure the hardware will receive the message at least once (in case of some signals can be faulty due noise or error). Sending multiple messages in one press is also useful for continuous actions. For example, when we are surfing all TV channels we press and hold Channel up (or Channel down), the TV will change the channels continuously until we release the button. Another example is when we press a hold volume up (or volume down), we the volume will increase (or decrease) based on how long we press the button.

3/ What modifications would you make to your code to provide an internal signal that goes high when a new message comes in? You do not have to synthesize this modification, but please provide the Verilog code that would do this. Hint: you can use the message count register. If this signal was made available to the processor, what might this signal be used for?

```
always@(posedge userDefinedCLK)
if (newMessage==1)
        begin
                internalSignal =1;
                timer=timer+1;
                if (timer==100)
                        newMessage = 0;
        end
```

The Verilog code above is for the internal signal that goes high when a new message comes in. When a new message arrives, I set the internal signal to 1 for short period then after that, I set the new message back to 0. It is useful to have internal signal to send to other components when the system needs to perform some other tasks when it receives a message. It is also useful for an interrupt system to stop some running process when a new message received.

CODE:

Helloworld.c (used to test the hardware)

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "ir_demod.h"

void print(char *str);

int main() {
   init_platform();



   /* Reading from slv_reg0 and slave_reg1 */
   u32 count = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR,
IR_DEMOD_S00_AXI_SLV_REG1_OFFSET);
   u32 message =
IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR,
IR_DEMOD_S00_AXI_SLV_REG0_OFFSET);

   // if theres a change between oldMsg and message, output to terminal
   u32 oldMsg =  message;

   for (;;) {
         message =
IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR,
IR_DEMOD_S00_AXI_SLV_REG0_OFFSET);
         count = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR,
IR_DEMOD_S00_AXI_SLV_REG1_OFFSET);
         if (message != oldMsg) {
                  printf("IR Signal Received: 0x%x\n", (unsigned int)message);
                  printf("Total messages received: %d\n", (unsigned int)count);

                  if (message == 0x10)
                           print("Channel 1 pressed\n\n");
```

```
                else if (message == 0x810)
                        print("Channel 2 pressed\n\n");
                else if (message == 0x410)
                        print("Channel 3 pressed\n\n");
                else if (message == 0xc10)
                        print("Channel 4 pressed\n\n");
                else if (message == 0x490)
                        print("Volume up pressed\n\n");
                else if (message == 0xc90)
                        print("Volume down pressed\n\n");
                else if (message == 0x90)
                        print("Channel up pressed\n\n");
                else if (message == 0x890)
                        print("Channel down pressed\n\n");
                else if (message == 0x7b0)
                        print("Stop pressed\n\n");
                else if (message == 0xfb0)
                        print("Play pressed\n\n");
        }
        oldMsg = message;

   }
   cleanup_platform();
   return 0;
}
```

ir_demod_v1_0_S00_AXI.v

```

`timescale 1 ns / 1 ps

        module ir_demod_v1_0_S00_AXI #
        (
                // Users to add parameters here

                // User parameters ends
                // Do not modify the parameters beyond this line

                // Width of S_AXI data bus
                parameter integer C_S_AXI_DATA_WIDTH        = 32,
                // Width of S_AXI address bus
                parameter integer C_S_AXI_ADDR_WIDTH        = 4
        )
```

```
   (
		// Users to add ports here
input wire IR_signal,
		// User ports ends
		// Do not modify the ports beyond this line

		// Global Clock Signal
		input wire  S_AXI_ACLK,
		// Global Reset Signal. This Signal is Active LOW
		input wire  S_AXI_ARESETN,
		// Write address (issued by master, acceped by Slave)
		input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
		// Write channel Protection type. This signal indicates the
		// privilege and security level of the transaction, and whether
		// the transaction is a data access or an instruction access.
		input wire [2 : 0] S_AXI_AWPROT,
		// Write address valid. This signal indicates that the master signaling
		// valid write address and control information.
		input wire  S_AXI_AWVALID,
		// Write address ready. This signal indicates that the slave is ready
		// to accept an address and associated control signals.
		output wire  S_AXI_AWREADY,
		// Write data (issued by master, acceped by Slave)
		input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
		// Write strobes. This signal indicates which byte lanes hold
		// valid data. There is one write strobe bit for each eight
		// bits of the write data bus.
		input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
		// Write valid. This signal indicates that valid write
		// data and strobes are available.
		input wire  S_AXI_WVALID,
		// Write ready. This signal indicates that the slave
		// can accept the write data.
		output wire  S_AXI_WREADY,
		// Write response. This signal indicates the status
		// of the write transaction.
		output wire [1 : 0] S_AXI_BRESP,
		// Write response valid. This signal indicates that the channel
		// is signaling a valid write response.
		output wire  S_AXI_BVALID,
		// Response ready. This signal indicates that the master
		// can accept a write response.
		input wire  S_AXI_BREADY,
		// Read address (issued by master, acceped by Slave)
		input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
		// Protection type. This signal indicates the privilege
		// and security level of the transaction, and whether the
		// transaction is a data access or an instruction access.
		input wire [2 : 0] S_AXI_ARPROT,
		// Read address valid. This signal indicates that the channel
		// is signaling valid read address and control information.
		input wire  S_AXI_ARVALID,
		// Read address ready. This signal indicates that the slave is
		// ready to accept an address and associated control signals.
		output wire  S_AXI_ARREADY,
		// Read data (issued by slave)
		output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
		// Read response. This signal indicates the status of the
		// read transfer.
		output wire [1 : 0] S_AXI_RRESP,
		// Read valid. This signal indicates that the channel is
		// signaling the required read data.
		output wire  S_AXI_RVALID,
		// Read ready. This signal indicates that the master can
		// accept the read data and response information.
		input wire  S_AXI_RREADY
	);

	// AXI4LITE signals
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_awaddr;
	reg  	axi_awready;
	reg  	axi_wready;
	reg [1 : 0] 	axi_bresp;
	reg  	axi_bvalid;
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_araddr;
	reg  	axi_arready;
	reg [C_S_AXI_DATA_WIDTH-1 : 0] 	axi_rdata;
	reg [1 : 0] 	axi_rresp;
	reg  	axi_rvalid;

	// Example-specific design signals
	// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
	// ADDR_LSB is used for addressing 32/64 bit registers/memories
	// ADDR_LSB = 2 for 32 bits (n downto 2)
	// ADDR_LSB = 3 for 64 bits (n downto 3)
	localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
```

```
localparam integer OPT_MEM_ADDR_BITS = 1;
//----------------------------------------------
//-- Signals for user logic register space example
//------------------------------------------------
//-- Number of Slave Registers 4
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]slv_reg3;
wire      slv_reg_rden;
wire      slv_reg_wren;
reg [C_S_AXI_DATA_WIDTH-1:0] reg_data_out;
integer  byte_index;

// I/O Connections assignments

assign S_AXI_AWREADY   = axi_awready;
assign S_AXI_WREADY    = axi_wready;
assign S_AXI_BRESP     = axi_bresp;
assign S_AXI_BVALID    = axi_bvalid;
assign S_AXI_ARREADY   = axi_arready;
assign S_AXI_RDATA     = axi_rdata;
assign S_AXI_RRESP     = axi_rresp;
assign S_AXI_RVALID    = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awready <= 1'b0;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
        begin
          // slave is ready to accept write address when
          // there is a valid write address and write data
          // on the write address and data bus. This design
          // expects no outstanding transactions.
          axi_awready <= 1'b1;
        end
      else
        begin
          axi_awready <= 1'b0;
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_awaddr <= 0;
    end
  else
    begin
      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
        begin
          // Write Address latching
          axi_awaddr <= S_AXI_AWADDR;
        end
    end
end

// Implement axi_wready generation
// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_wready <= 1'b0;
    end
  else
    begin
      if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
```

```
	          begin
	            // slave is ready to accept write data when 
	            // there is a valid write address and write data
	            // on the write address and data bus. This design 
	            // expects no outstanding transactions. 
	            axi_wready <= 1'b1;
	          end
	        else
	          begin
	            axi_wready <= 1'b0;
	          end
	      end 
	end       

	// Implement memory mapped register select and write logic generation
	// The write data is accepted and written to memory mapped registers when
	// axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. 
Write strobes are used to
	// select byte enables of slave registers while writing.
	// These registers are cleared when reset (active low) is applied.
	// Slave register write enable is asserted when valid address and data are available
	// and the slave is ready to accept the write address and write data.
	assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && 
S_AXI_AWVALID;

	always @( posedge S_AXI_ACLK )
	begin
	  if ( S_AXI_ARESETN == 1'b0 )
	    begin
	      //slv_reg0 <= 0;
	      //slv_reg1 <= 0;
	      //slv_reg2 <= 0;
	      //slv_reg3 <= 0;
	    end 
	  else begin
	    if (slv_reg_wren)
	      begin
	        case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
	          2'h0:
	            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index 
= byte_index+1 )
	              if ( S_AXI_WSTRB[byte_index] == 1 ) begin
	                // Respective byte enables are asserted as per write strobes
```

```
              // Slave register 0
              //slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
             end
           2'h1:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
= byte_index+1 )
             if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 1
              //slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
             end
           2'h2:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
= byte_index+1 )
             if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 2
              //slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
             end
           2'h3:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
= byte_index+1 )
             if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 3
              //slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
             end
           default : begin
                   //slv_reg0 <= slv_reg0;
                   //slv_reg1 <= slv_reg1;
                   //slv_reg2 <= slv_reg2;
                   //slv_reg3 <= slv_reg3;
                  end
          endcase
         end
       end
      end

      // Implement write response logic generation
      // The write response and response valid signals are asserted by the slave
      // when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
      // This marks the acceptance of address and indicates the status of
      // write transaction.
```

```
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_bvalid  <= 0;
      axi_bresp   <= 2'b0;
    end
  else
    begin
      if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
S_AXI_WVALID)
        begin
          // indicates a valid write response is available
          axi_bvalid <= 1'b1;
          axi_bresp  <= 2'b0; // 'OKAY' response
        end                   // work error responses in future
      else
        begin
          if (S_AXI_BREADY && axi_bvalid)
            //check if bready is asserted while bvalid is high)
            //(there is a possibility that bready is always asserted high)
            begin
              axi_bvalid <= 1'b0;
            end
        end
    end
end

// Implement axi_arready generation
// axi_arready is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_awready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.

always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_arready <= 1'b0;
      axi_araddr  <= 32'b0;
    end
  else
    begin
      if (~axi_arready && S_AXI_ARVALID)
        begin
          // indicates that the slave has acceped the valid read address
          axi_arready <= 1'b1;
          // Read address latching
          axi_araddr  <= S_AXI_ARADDR;
        end
      else
        begin
          axi_arready <= 1'b0;
        end
    end
end

// Implement axi_arvalid generation
// axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_ARVALID and axi_arready are asserted. The slave registers
// data are available on the axi_rdata bus at this instance. The
// assertion of axi_rvalid marks the validity of read data on the
// bus and axi_rresp indicates the status of read transaction.axi_rvalid
// is deasserted on reset (active low). axi_rresp and axi_rdata are
// cleared to zero on reset (active low).
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_rvalid <= 0;
      axi_rresp  <= 0;
    end
  else
    begin
      if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
        begin
          // Valid read data is available at the read data bus
          axi_rvalid <= 1'b1;
          axi_rresp  <= 2'b0; // 'OKAY' response
        end
      else if (axi_rvalid && S_AXI_RREADY)
        begin
          // Read data is accepted by the master
          axi_rvalid <= 1'b0;
        end
    end
end

// Implement memory mapped register select and read logic generation
// Slave register read enable is asserted when valid address is available
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0   : reg_data_out <= slv_reg0;
        2'h1   : reg_data_out <= slv_reg1;
        2'h2   : reg_data_out <= slv_reg2;
        2'h3   : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
      endcase
end

// Output register or memory read data
always @( posedge S_AXI_ACLK )
begin
  if ( S_AXI_ARESETN == 1'b0 )
    begin
      axi_rdata  <= 0;
    end
  else
    begin
      // When there is a valid read address (S_AXI_ARVALID) with
      // acceptance of read address by the slave (axi_arready),
      // output the read dada
      if (slv_reg_rden)
        begin
          axi_rdata <= reg_data_out;     // register read data
        end
    end
end

// Add user logic here
reg [31:0] clockCount;
reg clock;
assign reset = ~S_AXI_ARESETN; //S_AXI_ARESETN is active low
assign mainClock = S_AXI_ACLK;
/* Dividing clock by 1,000 to manage counter better */
always@(posedge mainClock) begin
        if (clockCount == 1000 && ~reset) begin
                clock <= 1;
                clockCount <= 0;
        end

        else if (reset) begin
                clockCount <= 0;
                clock <= 0;
        end

        else begin
                clockCount <= clockCount + 1;
                clock <= 0;
        end
end

reg oldSignal;/* Old Signal used for edge detection */
reg [31:0] counter;/* Counter to count time of ~IR_signal */
reg State;
reg NextState;
reg [11:0] demodulatedMessage;
reg startSignal;
reg [31:0] bitCount;
reg keepCounting;

always@(posedge clock) begin
        oldSignal <= IR_signal;
        if (oldSignal && ~IR_signal) begin
                /* We start counting on the negative edge of the clock */
                keepCounting <= 1'b1;
        end

        else if (~oldSignal && IR_signal) begin //if we just resolve a bit
                /* We stop counting when we reach the positive edge again */
                bitCount <= bitCount + 1; //increament bitCount (we just read a bit)
                keepCounting <= 1'b0; //stop counting
                counter <= 0; //reset counter
                //store the bit we just read into current message
```

```
            if (startSignal && bitCount >= 12) begin //if we have received a full
message
                slv_reg0 <= demodulatedMessage;
                slv_reg1 <= slv_reg1 + 1;
                bitCount <= 0;
                startSignal <= 0;
            end

            else if (startSignal && bitCount < 12 && bitCount != 0) begin //in order
to prevent having to use blocking assignments
                demodulatedMessage[11 - (bitCount - 1)] <= State;
            end
        end

        /* IR Signal is Active Low */
        if (keepCounting && ~IR_signal) begin
            counter <= counter + 1;
            /* With a start signal time of 2.4 ms, IR_signal
                shoud be low for approximately 180 cycles
                Zero Signal = 0.6 ms = 45 cycles
                One Signal = 1.2 = 90 cycles
                I'm using the halfway point between 0 and 1
                to create a fine line between them and resolving
                an appropriate signal (.9 ms = 68 cycles)
                Halfway point between Start and 1 = 1.8 ms = 135 cycles */
                if (counter >= 20 && counter <= 68) begin
                    State <=0;
                end

                else if (counter >= 69 && counter <= 134) begin
                    State <= 1;
                end

                else if (counter >= 135 && counter <= 250) begin
                        startSignal <= 1;
                        /* Initialize bit count to 0 */
                        bitCount <= 0;
                end

                else begin
                    /* Just to be safe (it's probably impossible
                        to get to here anyway) */
                    State <= 0;
```

```
                        end

                end


            end
            // User logic ends

endmodule
```