ECEN 749 Lab 6 Report

Tong Lu
UIN 621007982

November 02, 2018

TEXAS A&M
UNIVERSITY®

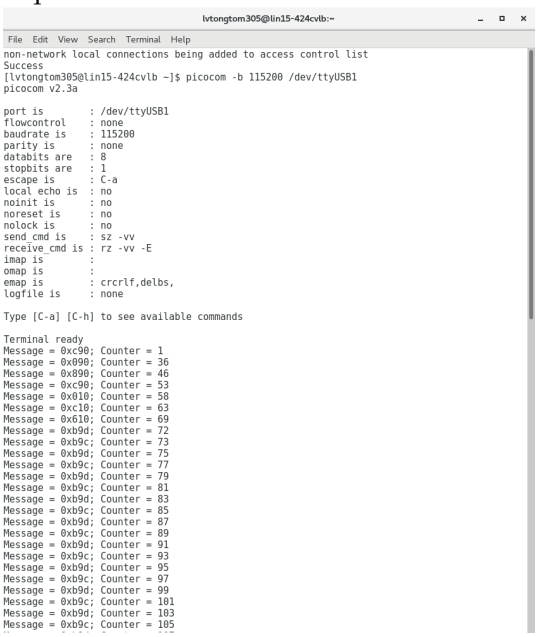
Introduction

Procedure

1. Following the lab manual and built the IR receiver circuit. Measure the output signal using oscilloscope and demonstrate to TA.
2. Based on the procedure from lab3 and lab4, build the i_demod (See Appendix) module and ZYBO processor system.
3. Generate bitstream, export the hardware including bitstream.
4. Launch SDK, write a simple C program (See Appendix) based on the helloworld example project to display register printer out on UART port.
5. Connect the IR_receiver circuit to ZYBO board, program FPGA, point the TV remote to the IR receiver, watch the printout using picocom.
6. Demonstrate the result to TA.

Result

All the programs was finished and demonstrated to TA. The programs are working well and meet all the requirement on lab manual.



```
lvtongtom305@lin15-424cvlb:~
File  Edit  View  Search  Terminal  Help
non-network local connections being added to access control list
Success
[lvtongtom305@lin15-424cvlb ~]$ picocom -b 115200 /dev/ttyUSB1
picocom v2.3a

port is        : /dev/ttyUSB1
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     : no
nolock is      : no
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
Message = 0xc90; Counter = 1
Message = 0x090; Counter = 36
Message = 0x890; Counter = 46
Message = 0xc90; Counter = 53
Message = 0x810; Counter = 58
Message = 0xc10; Counter = 63
Message = 0x610; Counter = 69
Message = 0xb9d; Counter = 72
Message = 0xb9c; Counter = 73
Message = 0xb9d; Counter = 75
Message = 0xb9c; Counter = 77
Message = 0xb9d; Counter = 79
Message = 0xb9c; Counter = 81
Message = 0xb9d; Counter = 83
Message = 0xb9c; Counter = 85
Message = 0xb9d; Counter = 87
Message = 0xb9c; Counter = 89
Message = 0xb9d; Counter = 91
Message = 0xb9c; Counter = 93
Message = 0xb9d; Counter = 95
Message = 0xb9c; Counter = 97
Message = 0xb9d; Counter = 99
Message = 0xb9c; Counter = 101
Message = 0xb9d; Counter = 103
Message = 0xb9c; Counter = 105
```

Conclusion

In this lab, I built a simple IR signal demodulation system based on OpAmp circuit and ZYBO FPGA system. Now I have a deeper understanding about FSM and verilog programming. This success of this lab will help me a lot in the following two labs.

Answer to Questions

(a) **What are the hexadecimal control codes for the following buttons: volume up/down, channel up/-down, stop/play, 1, 2, 3, and 4? Tabulate your results.**

$Button$	$ControlCode$
$volumeup/down$	$0x490/0xc90$
$channelup/down$	$0x090/0x890$
$stop/play$	$0x7b0/0xfb0$
$1, 2, 3, 4$	$0x010, 0x810, 0x410, 0xc10$

(b) **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.** Approxmately five same messages are transmitted after each press of a button. If the button is held, then more messages will be transmitted. This is because the command messages can contain noises, so sending muitiple messages can ensure at least one or two correct message is transmitted.

(c) **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 @(msg_cnt)  begin // check when the value of message counter changed
    prev_msg <= slv_reg0; // record the previous message
    if (prev_msg == slv_reg0) begin // if this is the same message
        slv_reg2 <= 0; // register 2 goes low
    end
    else begin // if this is a new message coming in
        slv_reg2 <= 1; // register 2 goes high
    end
end
```

This signal can be used for interrupt.

Appendix

```

`timescale 1 ns / 1 ps

    module ir_demod_v2_0 #
    (
        // Users to add parameters here

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


        // Parameters of Axi Slave Bus Interface S00_AXI
        parameter integer C_S00_AXI_DATA_WIDTH  = 32,
        parameter integer C_S00_AXI_ADDR_WIDTH  = 4
    )
    (
        // Users to add ports here
        input wire IR_signal,
        // User ports ends
        // Do not modify the ports beyond this line


        // Ports of Axi Slave Bus Interface S00_AXI
        input wire  s00_axi_aclk,
        input wire  s00_axi_aresetn,
        input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
        input wire [2 : 0] s00_axi_awprot,
        input wire  s00_axi_awvalid,
        output wire  s00_axi_awready,
        input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
        input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
        input wire  s00_axi_wvalid,
        output wire  s00_axi_wready,
        output wire [1 : 0] s00_axi_bresp,
        output wire  s00_axi_bvalid,
        input wire  s00_axi_bready,
        input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
        input wire [2 : 0] s00_axi_arprot,
        input wire  s00_axi_arvalid,
        output wire  s00_axi_arready,
        output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
        output wire [1 : 0] s00_axi_rresp,
        output wire  s00_axi_rvalid,
        input wire  s00_axi_rready
    );
// Instantiation of Axi Bus Interface S00_AXI
    ir_demod_v2_0_S00_AXI # (
        .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    ) ir_demod_v2_0_S00_AXI_inst (
        .IR_signal(IR_signal),
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr),
        .S_AXI_AWPROT(s00_axi_awprot),
        .S_AXI_AWVALID(s00_axi_awvalid),
```

```
        .S_AXI_AWREADY(s00_axi_awready),
        .S_AXI_WDATA(s00_axi_wdata),
        .S_AXI_WSTRB(s00_axi_wstrb),
        .S_AXI_WVALID(s00_axi_wvalid),
        .S_AXI_WREADY(s00_axi_wready),
        .S_AXI_BRESP(s00_axi_bresp),
        .S_AXI_BVALID(s00_axi_bvalid),
        .S_AXI_BREADY(s00_axi_bready),
        .S_AXI_ARADDR(s00_axi_araddr),
        .S_AXI_ARPROT(s00_axi_arprot),
        .S_AXI_ARVALID(s00_axi_arvalid),
        .S_AXI_ARREADY(s00_axi_arready),
        .S_AXI_RDATA(s00_axi_rdata),
        .S_AXI_RRESP(s00_axi_rresp),
        .S_AXI_RVALID(s00_axi_rvalid),
        .S_AXI_RREADY(s00_axi_rready)
    );

    // Add user logic here

    // User logic ends

    endmodule
```

src/ir_demod_v2_0.v

```

`timescale 1 ns / 1 ps

    module ir_demod_v2_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,
        /*output reg [C_S_AXI_DATA_WIDTH-1:0]   slv_reg0,
        output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1,
        output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2,
        output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg3,
        */
        // 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] pulse_length_counter; // measure the length of each pulse
        reg [1:0] state; // state for demodulate IR signal
        reg [3:0] pulse_idx; // measure the number of pulses for decoding
        reg [11:0] temp_reg0; // store temporary value
        parameter START = 2'b00, LEN0 = 2'b01, LEN1 = 2'b10, OTHER = 2'b11; // state parameter
        reg IR_signal_prev;
        always @(posedge S_AXI_ACLK) begin // set up counter
            if ( S_AXI_ARESETN == 1'b0 ) begin // sync reset
                pulse_length_counter <= 0; // reset counter
                state <= OTHER; // reset state to OTHER
            end
            else begin
                if (~IR_signal) begin // when IR_signal is 0
                    pulse_length_counter <= pulse_length_counter + 1; // counter increment
                end
                else begin // if IT_signal is 1
                    pulse_length_counter <= 0; // clear the counter
                end

                if (pulse_length_counter > 40_000 && pulse_length_counter <= 50_000) begin // around 0.6ms, or 45_000 cycles
                    state <= LEN0; // demodulate signal to 0
                end
                else if (pulse_length_counter > 85_000 && pulse_length_counter <= 95_000) begin // around 1.2ms, or 90_000 cycles
                    state <= LEN1; // demodulate signal to 1
                end
                else if (pulse_length_counter > 175_000 && pulse_length_counter <= 185_000) begin // around 2.4ms, or 180_000 cycles
                    state <= START; // demodulate signal to START
                end
                else begin
                    state <= OTHER; // If signal is not correct, send to OTHER state
                end
            end
```

```
        end

        always @(posedge IR_signal) begin // when IR signal rises, send out state
            case(state)
                START: begin
                    //slv_reg0  <= 0;
                    temp_reg0 <= 0; // reset temp register
                    pulse_idx <= 0; // reset bit counter
                end
                LEN0: begin // when signal is 0
                    if (pulse_idx < 11) begin// for first 11 bits
                        temp_reg0[11 - pulse_idx] <= 0; // put 0
                        pulse_idx <= pulse_idx + 1; // counter increment
                    end
                    if (pulse_idx == 11) begin // last bit
                        slv_reg0 <= temp_reg0; // put temp into register0, with last bit
                        if (slv_reg1 >= 1) begin
                            slv_reg1 <= slv_reg1 +1; // increment the message counter
                        end
                        else begin // initalize reg1
                            slv_reg1 <= 1;
                        end
                    end
                end
                LEN1: begin // if signal is 1
                    if (pulse_idx < 11) begin // for first 11 bits
                        temp_reg0[11 - pulse_idx] <= 1; // put 1
                        pulse_idx <= pulse_idx + 1; // counter increment
                    end
                    if (pulse_idx == 11) begin // last bit
                        slv_reg0 <= temp_reg0 + 1; // put temp into register with last bit
                        if (slv_reg1 >= 1) begin
                            slv_reg1 <= slv_reg1 +1; // increment message counter
                        end
                        else begin // initalize reg1
                            slv_reg1 <= 1;
                        end
                    end
                end
                OTHER: begin
                    pulse_idx <= 4'b1111; // set the pulse to a impossible state, prevent the future counting and recording.
                end
                default: ;
            endcase
        end
        // User logic ends

    endmodule
```

src/ir_demod_v2_0_S00_AXI.v

```
/*****************************************************************************
*
* Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.
*
* Permission is hereby granted, free of charge, to any person obtaining a copy
```

```
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in
* all copies or substantial portions of the Software.
*
* Use of the Software is limited solely to applications:
* (a) running on a Xilinx device, or
* (b) that interact with a Xilinx device through a bus or interconnect.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
* XILINX  BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*
* Except as contained in this notice, the name of the Xilinx shall not be used
* in advertising or otherwise to promote the sale, use or other dealings in
* this Software without prior written authorization from Xilinx.
*
******************************************************************************/

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ----------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ----------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "ir_demod.h"
void print(char *str);

int main()
{
    u32 reg0;
    u32 reg0_prev;
    u32 reg1;
    u32 reg1_prev;
    u32 reg3;
    u32 reg3_prev;

    init_platform();
    while (1){
```

```c
        //IR_DEMOD_mWriteReg(XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, IR_DEMOD_S00_AXI_SLV_REG0_OFFSET, 16);
        reg0 = IR_DEMOD_mReadReg(XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, IR_DEMOD_S00_AXI_SLV_REG0_OFFSET);
        reg1 = IR_DEMOD_mReadReg(XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, IR_DEMOD_S00_AXI_SLV_REG1_OFFSET);
        reg3 = IR_DEMOD_mReadReg(XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, IR_DEMOD_S00_AXI_SLV_REG3_OFFSET);
        if (reg0 != reg0_prev) {
            printf("Message = 0x%03x; ", reg0);
            printf("Counter = %d\n", reg1);
            reg0_prev = reg0;
        }
    }
    cleanup_platform();
    return 0;
}
```

src/helloworld.c