Description: Description: Description: Description: Description: Description: Description: Description: Description: Description: Texas A&M University Logo - aTm

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:**

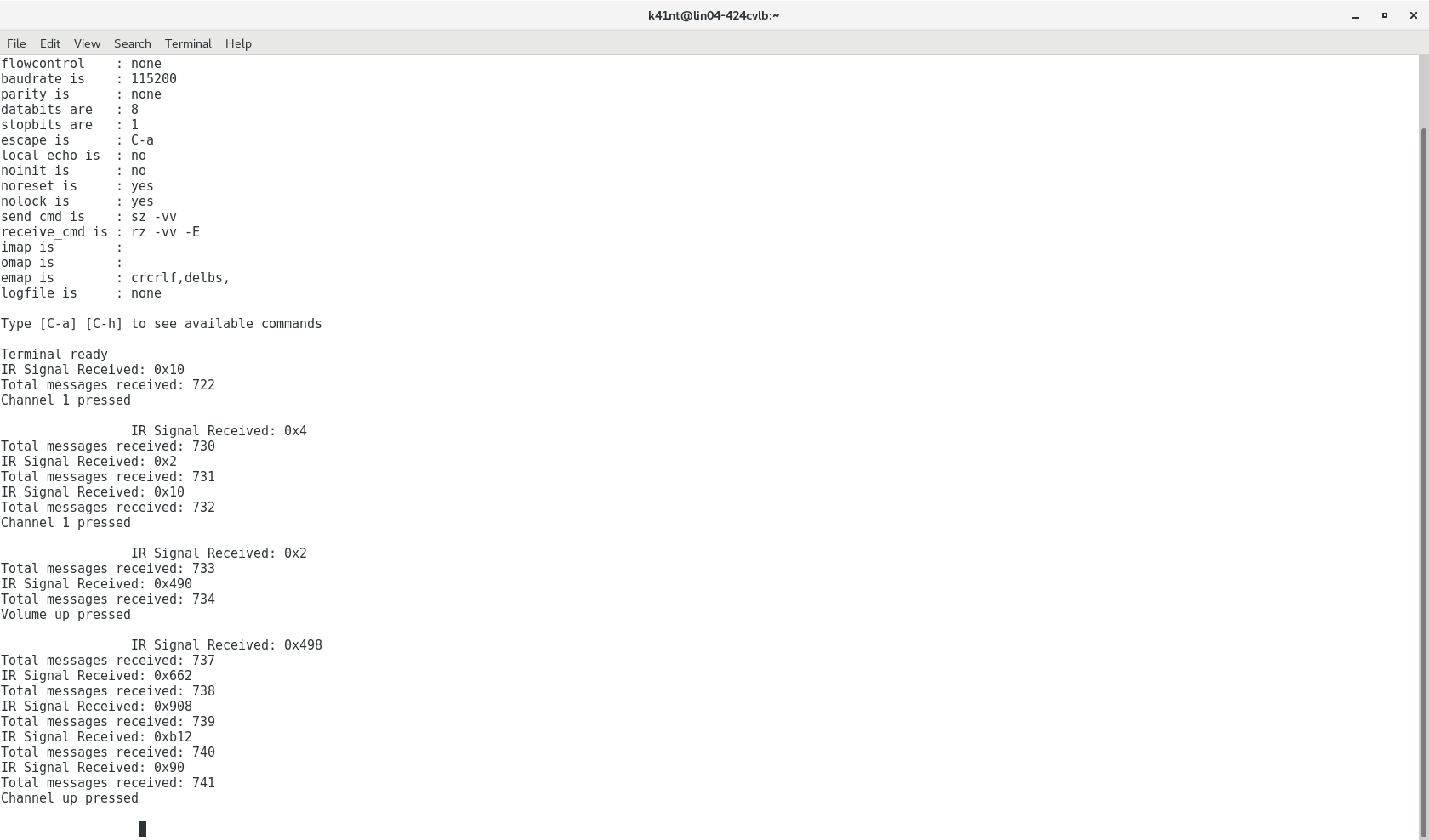
1/ The first part of the lab was to build the IR signal receiver circuit. The schematic for the circuit was provided and we used a LM339 comparator. Then we connected the circuit to the oscilloscope to observe the pulse signal generated from the TV remote. We observed the results for several buttons 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:



**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