ECEN 749- Microprocessor System Design

Section 603

TA: Mr. Kunal Bharathi

LAB 7

IR-Remote Peripheral for Linux System

Date of Performance: 10/17/2019

Student Name: Dhiraj Dinesh Kudva

UIN: 829009538

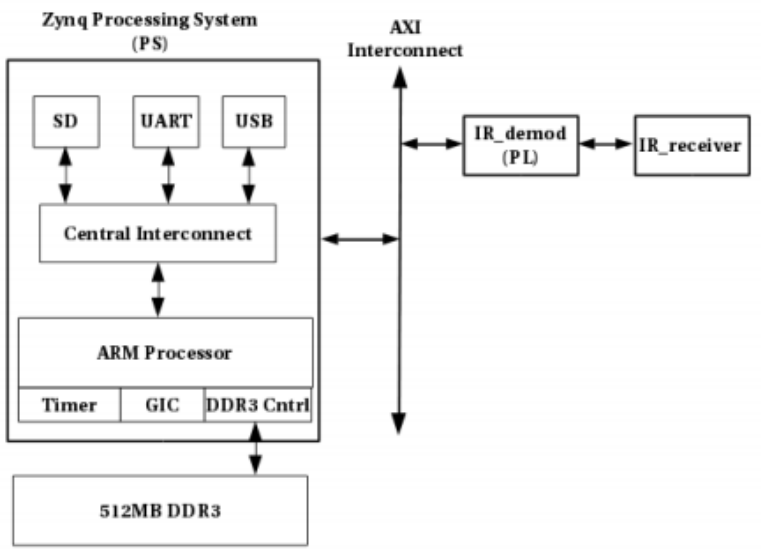
Introduction

The purpose of this lab to implement a IR detection hardware on breadboard and decode the IR signals with the help of FPGA and Verilog.

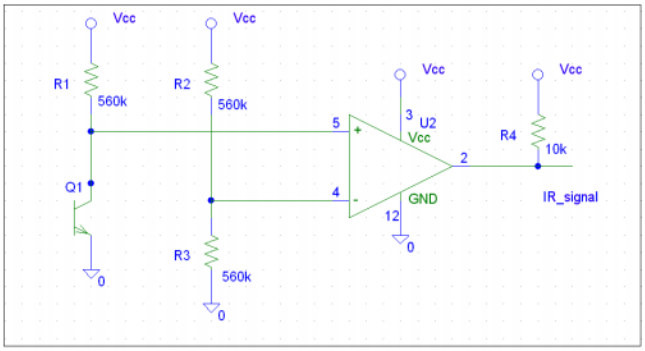
The demodulation hardware is to be designed using FPGA and Verilog and the demodulated message must be software accessible (to show on Picocom console).

Design:

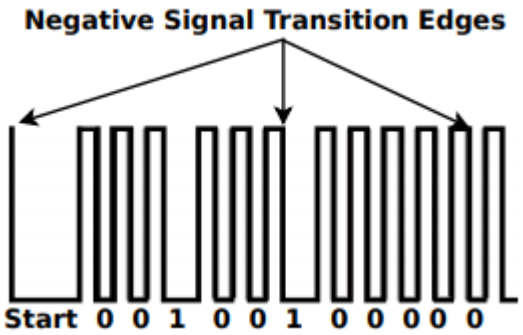
The below diagram represents the overall IR Demodulator to be designed in this lab. The Zynq processing system and IR demodulator will be configured on Vivado, while the IR receiver circuit would be hardware connected on the breadboard. The signal received form the breadboard will be decoded and shown on the terminal output. Demodulation logic is written inside the IR demodulator IP peripheral that is connected to the Zynq Processing System.



The connection for IR receiver signal is to be done as per the below figure on the breadboard. The output from the hardware is checked on the Oscilloscope. (Vcc=+3.3 V.)



The basic principle of an IR remote control is to utilize infrared light for transmitting short control messages from the remote control to the IR receiver. Pulses of infrared light which represent specific binary codes are transmitted from the IR remote.



The above figure is the output of the IR Receiver signal. Check this on the oscilloscope.

Part 1: IR Receiver Circuit Assembly

1. In this lab, as per the design of the IR receiver circuit shown above, the components must be connected on breadboard.
2. Check the output waveform on the oscilloscope whenever a button is pressed from the IR remote.
3. It can be observed that each button transmits 12 bits.
4. Freeze the waveform on the CRO and measure the width of the 'START', 1 and 0. The one bit is generally twice as wide as Zero state and START is twice of one.

Part 2: Creating IR peripheral

1. Create a block design in vivado and select the Zynq Processing System.
2. The IR demodulator is implemented on FPGA. Configure the Zynq Processing System as follows:
    - Select 'Clock Configuration' and expand 'PL Fabric Clocks'.
    - Set the frequency of FCLK CLK0 to 75MHz for this lab.
3. Create the template code for a custom peripheral with four software accessible and label the peripheral 'ir demod'.
4. In the 'ir demod v1 SOO AXI.v' file, disable all AXI write capabilities to the software accessible registers and ensure the AXI read capabilities remain.
5. Add IR_signal as the port name in the 'ir demod v1.v' file and add the logic written to demodulate the code in 'ir demod v1 SOO AXI.v'.
6. The Verilog code written must detect the start edge as it indicates the start of the message and then count the ones and zeroes based on the width in PWM output of the IR receiver. (This IR receiver output is connected to the V15 on the Zybo board and constraints for the same must be written).
7. Slv_reg0 must hold the latest demodulated message and the slv_reg1 must hold the number of messages transmitted.
8. Clock and Reset must be taken from the auto generated code for the IR Demodulator peripheral. Reducing the clock will provide better decoded message and also reduce the effect of any noise while displaying the output.
9. Repackage the IP and merge ports and interfaces if any changes are made while declaring in the user logic of IR peripheral.
10. Create HDL wrapper, generate bitstream and export the file including bitstream.
11. Then launch SDK.

12. Create new application project and open hello world file.
13. C code for displaying the demodulated message in hexadecimal form on the picocom terminal must be written here.
14. Do not forget to cast the base address of IP to type ‘u32’.
15. Connect Vcc = 3.3V , GND, and the appropriate IR signal pin on the ZYBO Z7-10 board to IR receiver circuit. Ensure the JP5 pin is in JTAG mode.

Program:

Verilog Code:

```
    // Add user logic here
reg [31:0] Main_clock_count; // Clock counter
        reg reduced_clock; // Device counter
        assign reset = ~S_AXI_ARESETN; // Active Low Reset
        assign mainClock = S_AXI_ACLK; // System clock
        // Modifying mainClock for Counting
        always@(posedge mainClock) begin
        if (Main_clock_count == 1000 && ~reset) begin
        reduced_clock <= 1;
        Main_clock_count <= 0;
        end
        else if (reset) begin
        Main_clock_count <= 0;
        reduced_clock <= 0;
        end
        else begin
        Main_clock_count <= Main_clock_count + 1;
        reduced_clock <= 0;
        end
        end
        reg prev_main_signal; // detect edge
        reg [31:0] msg_select; // msg selector start, 1 or
zero
        reg State; // msg bit
        reg [11:0] final_msg; // 12 bit msg
        reg signal_start; // start signal flag
        reg [31:0] number_bits; // max 12 bits per msg
        reg flag_count; // flag to start counting


always@(posedge reduced_clock) begin
        prev_main_signal <= IR_signal;
        if (prev_main_signal && ~IR_signal) begin
        // neg edge found, set the flag to start counting of
start bit
```

```
        flag_count <= 1'b1;
        end
        else if (~prev_main_signal && IR_signal) begin // if
pos edge then 1 bit is counted

        number_bits <= number_bits + 1;
        flag_count <= 1'b0;
        msg_select <= 0;

        if (signal_start && number_bits >= 12) begin // once
12 bits found, send it to the output
        slv_reg0 <= final_msg;
        slv_reg1 <= slv_reg1 + 1;
        number_bits <= 0;
        signal_start <= 0;
        end

        else if (signal_start && number_bits < 12 &&
number_bits != 0) begin //if 12 bits not over, keeep reading
the messages
        final_msg[11 - (number_bits - 1)] <= State;
        end
        end

        if (flag_count && ~IR_signal) begin
        msg_select <= msg_select + 1;
        // Start signal time of 2.3 ms, 0 signal time 0.6ms, 1
signal time 1.19ms.
        // Start signal low for 180 cycles
        // Zero Signal = 45 cycles
        // One Signal = 90 cycles

        if (msg_select >= 20 && msg_select <= 68) begin
        State <=0;
        end
        else if (msg_select >= 69 && msg_select <= 134) begin
        State <= 1;
        end
        else if (msg_select >= 135 && msg_select <= 250) begin
        signal_start <= 1;

        number_bits <= 0;
        end
        else begin
        State <= 0;
        end
        end
```

```
   end

// User logic ends

endmodule
```

C Code:

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "ir_demodulator.h"
#include "xil_io.h"
int main()
{
u32 actual_msg, msg_count, prev_actual_msg, prev_msg_count;
init_platform(); // Initialize FPGA platform
while(1)
{
actual_msg=IR_DEMODULATOR_mReadReg(0x43C00000,0); // actual_msg
read
msg_count=IR_DEMODULATOR_mReadReg(0x43C00000,4); // msg_count
read
if(actual_msg!=prev_actual_msg)
{ // Print demodulated message based on obtained values in
actual_msg
if(actual_msg==0x490)
{
printf("Demodulated message is VOL +\n");
}
if(actual_msg==0xc90)
{
printf("Demodulated message is VOL - \n");
}
if(actual_msg==0x90)
{
printf("Demodulated message is CH + \n");
}
if(actual_msg==0x890)
{
printf("Demodulated message is CH - \n");
}
else
{
printf("Demodulated message is %d \n", actual_msg);
}
```

```
}
if(msg_count!=prev_msg_count)
{
printf("Msg # %d\n",msg_count);
}
prev_actual_msg=actual_msg;
prev_msg_count=msg_count;
}
cleanup_platform(); // Deinitialize resources
return 0;
}
```

Output:

```
Type [C-a] [C-h] to see available comman

Terminal ready
Msg # 1
Demodulated message is CH +
Demodulated message is 144
Msg # 2
Msg # 3
Msg # 4
Demodulated message is VOL +
Demodulated message is 1168
Msg # 5
Msg # 6
Msg # 7
Msg # 8
Msg # 9
Msg # 10
Msg # 11
Msg # 12
Demodulated message is VOL -
Demodulated message is 3216
Msg # 13
Msg # 14
Msg # 15
Demodulated message is CH -
Msg # 17
Msg # 18
Msg # 19
```

```
Demodulated message is 1040
Msg # 294
Msg # 295
Msg # 296
Msg # 297
Msg # 298
Msg # 299
Msg # 300
Msg # 301
Msg # 302
Msg # 303
Demodulated message is 3088
Msg # 304
Msg # 305
Msg # 306
Msg # 307
Msg # 308
Msg # 309
Msg # 310
Msg # 311
Msg # 312
Demodulated message is 16
Msg # 313
Msg # 314
Msg # 315
Msg # 316
Msg # 317
Msg # 318
Msg # 319
Msg # 320
Demodulated message is 2064
Msg # 321
Msg # 322
Msg # 323
Msg # 324
Msg # 325
Msg # 326
Msg # 327
Msg # 328
Msg # 329
Demodulated message is 1040
Msg # 330
Demodulated message is 1296
```

Result:

In this lab, the hardware IR circuit was first designed and output was measured. If all the connections are proper then the waveform would follow the ideal case. Width of start bit = 2 x width of bit when 1 is transmitted and width of bit when 1 is transmitted =2 x width of bit when 0 is transmitted. After checking the hardware connection, the Verilog must be first checked with a standard test bench (generates a PWM waveform similar to the hardware) to check the logic. As the test bench generated waveform would not have any noise, we must take into account the noise factor in the code which is to be loaded on the IR peripheral.

Conclusion:

This lab helped in integrating the hardware with software and in understanding the PWM waveform transmitted by the remote and the unique code format for each button.

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

Ans.

Button	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

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.

Ans. Whenever a button is pressed, 4 to 5 messages of the same command are sent. The multiple messages are sent to ensure that the correct command is received at the receiver and it reduces the error that might be present at the receiver end because of the noise. Hence transmitting multiple messages reduces the probability of error.

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. If this signal was made available to the processor, what might this signal be used for??

Ans. A new register can be used for this purpose. For example, a new register called new_msg_counter is used. Whenever a new message arises, the slv_reg1 gets updated and this update sets the new_msg_counter to 1, to indicate the arrival of a new message. After the new message is arrived, during the next clock cycle it is again reset to zero, so that it can again be set only by arrival of a new message.

If this signal is made available to the processor, the signal can acts as an interrupt to indicate a presence of a new message.

Edited Verilog Code:

```
reg [31:0] Main_clock_count; // Clock counter
        reg reduced_clock; // Device counter
        assign reset = ~S_AXI_ARESETN; // Active Low Reset
        assign mainClock = S_AXI_ACLK; // System clock
        // Modifying mainClock for Counting
        always@(posedge mainClock) begin
        if (Main_clock_count == 1000 && ~reset) begin
        reduced_clock <= 1;
        Main_clock_count <= 0;
        end
        else if (reset) begin
        Main_clock_count <= 0;
        reduced_clock <= 0;
        end
        else begin
        Main_clock_count <= Main_clock_count + 1;
        reduced_clock <= 0;
        end
        end
        reg prev_main_signal; // detect edge
        reg [31:0] msg_select; // msg selector start, 1 or
zero
        reg State; // msg bit
        reg [11:0] final_msg; // 12 bit msg
        reg signal_start; // start signal flag
        reg [31:0] number_bits; // max 12 bits per msg
        reg flag_count; // flag to start counting
  reg new_msg_counter; //new reg added to indicate new message

always@(posedge reduced_clock) begin
     new_msg_counter<=1'b0; /*set to zero, changed to 1 when
new message arrives*/
        prev_main_signal <= IR_signal;
        if (prev_main_signal && ~IR_signal) begin
        // neg edge found, set the flag to start counting of
start bit
        flag_count <= 1'b1;
        end
        else if (~prev_main_signal && IR_signal) begin // if
pos edge then 1 bit is counted

        number_bits <= number_bits + 1;
        flag_count <= 1'b0;
        msg_select <= 0;
```

```
        if (signal_start && number_bits >= 12) begin // once
12 bits found, send it to the output
        slv_reg0 <= final_msg;
        slv_reg1 <= slv_reg1 + 1;
        new_msg_counter<=1'b1; /*set to one when new message
is arrived*/
        number_bits <= 0;
        signal_start <= 0;
        end

        else if (signal_start && number_bits < 12 &&
number_bits != 0) begin //if 12 bits not over, keeep reading
the messages
        final_msg[11 - (number_bits - 1)] <= State;
        end
        end

        if (flag_count && ~IR_signal) begin
        msg_select <= msg_select + 1;
        // Start signal time of 2.3 ms, 0 signal time 0.6ms, 1
signal time 1.19ms.
        // Start signal low for 180 cycles
        // Zero Signal = 45 cycles
        // One Signal = 90 cycles

        if (msg_select >= 20 && msg_select <= 68) begin
        State <=0;
        end
        else if (msg_select >= 69 && msg_select <= 134) begin
        State <= 1;
        end
        else if (msg_select >= 135 && msg_select <= 250) begin
        signal_start <= 1;

        number_bits <= 0;
        end
        else begin
        State <= 0;
        end
        end
        end

    // User logic ends

    endmodule
```