Mauro Lopez

LAB 7 Report

ECEN-449

Section 502

November 1, 2018

Introduction:

In this lab we are going to build a circuit that receives an IR-signal, reads it, and sends out a signal to our programmed FPGA board to print out in our screen. To read this, we will create a custom IP peripheral like we did with our multiply IP before.

Procedures:

We start of by first analyzing the diagram that was given to us in the lab manual, to model our block diagram. We will be following similar steps that we did, when we build our multiply IP.

We start off by creating our “Zynq processor “ block and load the settings from the 449NeededFiles directory. Once we have this, we need to go back and change the clock rate of our processor.

Once we have our processor ready, we need to create our own custom IP. We do this by going to the tool section in the top part of the window. Click on the create and repackage option, and it will launch a new vivado window. Once we have named and set up our IP, we need to edit its Verilog code.

We need to change look at the code and make sure nothing is writing to our slv_registers. We need to make sure of this, otherwise we will encounter bugs.

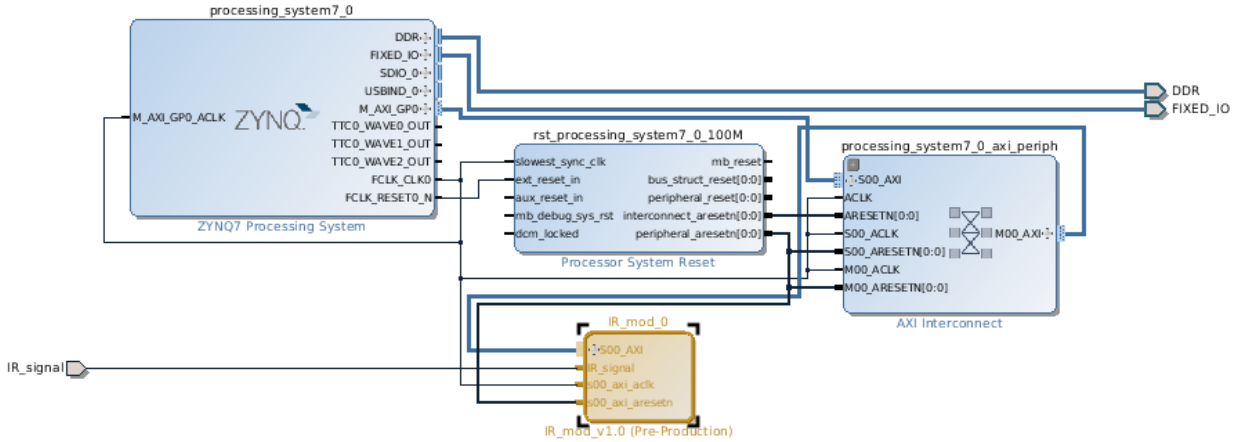
Once we have updated our Verilog code, we should end up with something similar to this:

```
1.          reg [64:0] counter;
2.          reg Possible_bit;
3.
4.          reg [11:0] message;
5.          reg startSignal;
6.          reg [7:0] total_bits;
7.          reg continue_count;
8.          reg previous_signal;
9.
10.         always@(posedge S_AXI_ACLK) begin
11.             previous_signal <= IR_signal;
12.             /*
13.             This is used as a buffer gate, to compare the change in the signal.
14.             */
15.
16.             if ( (previous_signal==1'b1)  && (IR_signal==1'b0 ) ) begin
17.             // we have a negative edge
18.
19.                 continue_count <= 1'b1;
20.             end
21.
22.             else if ((previous_signal==1'b0)  && (IR_signal==1'b1 )) begin // posedge
23.                 /*
24.                  * we stop the count on posedge, and we add another bit.
25.                 */
26.                 total_bits <= total_bits + 1;
27.                 continue_count <= 1'b0;
28.                 counter <= 0;
```

```
29.
30.                 if (startSignal && total_bits >= 12) begin // we have a full message.
31.                     slv_reg0 <= message; // send out message to register0
32.                     slv_reg1 <= slv_reg1 + 1; // keep track of ho many messages we got
33.                     total_bits <= 0;
34.                     startSignal <= 0;
35. // our signal flag is set to off, meaning we need to wait for another start signal
36.                 end
37.
38.                 else if (startSignal && total_bits < 12 && total_bits != 0) begin
39.             /* If less than 12 bits, and our starsignal is on, and we atleast have some bits.
40.                 */
41.                     message[11 - (total_bits - 1)] <= Possible_bit;
42.                     // sends the bit to a specific index.
43.                     // to start at the leftmost, we need to set the index from 11 to 0;
44.
45.                 end
46.             end
47.
48.
49.             if (continue_count && ~IR_signal) begin
50.             /* if our signal is positive we dont count,
51.              * this will help prevent from our counter to go over our ranges.
52.              * Ranges
53.              * zero >= .2us to .68 us.
54.              * one >= 1 ms to 1.34 ms.
55.              * start >= 1.35ms to 2.5 ms.
56.             */
57.                 counter <= counter + 1;
58.
59.                     if (counter >= 20000 && counter <= 68000) begin
60.                         Possible_bit <=0;
61.                     end
62.
63.                     else if (counter >= 69000 && counter <= 134000) begin
64.                         Possible_bit <= 1;
65.                     end
66.
67.                     else if (counter >= 135000 && counter <= 250000) begin
68.                             startSignal <= 1;
69.                             total_bits <= 0;
70.                     end
71.
72.
73.             end
74.
75.
76.         end
```

Once we have our code ready, we can repackage our IP and use it in our block diagram.

We should get something similar to this:



Notice that we have an “IR_signal” attach to our IP. We will be using this to feed a signal to our FPGA board. Before we do this, we will need to create an .XDC file to set this to a pin in the board.

We need to add the following to our .xdc file:

```
1.  set_property PACKAGE_PIN T20 [get_ports {IR_signal} ]
2.  set_property IOSTANDARD LVCMOS33 [ get_ports {IR_signal} ]
```

Once we have this, we need to generate a HDL wrapper, and then generate a bitstream.

Before moving on the lab, we need to make sure our circuit is working and that we measure the pulses with their respected values.

We then export our bitstream, into our SDK and launch our SDK. Once we are in SDK we need to create a new project. I used a templated “helloworld” code to make sure things worked.

We simply need to write a program, that outputs the message and number of messages we got.

My code end it up looking like this:

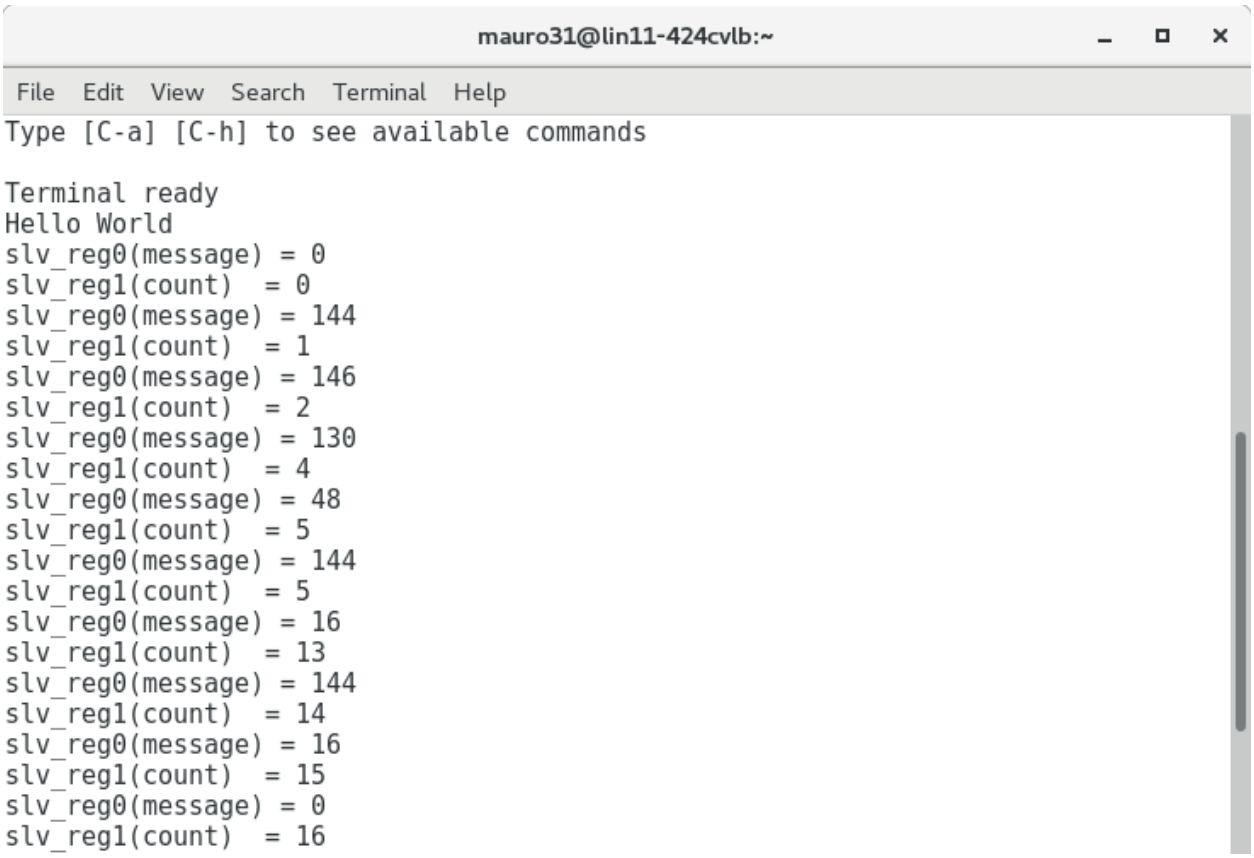
```
1.  #include <stdio.h>
2.  #include "platform.h"
3.  #include "xparameters.h"
4.  #include "IR_mod.h"
5.
6.  #define BASE_ADDRESS XPAR_IR_MOD_0_S00_AXI_BASEADDR
7.
8.  void print(char *str);
9.
10. int main()
11. {
12.
13.     print("Hello World\n\r");
```

```
14.     /*
15.      * declaring our variables
16.      */
17.     init_platform();
18.     unsigned long slv_reg0;
19.     unsigned long slv_reg1;
20.     unsigned long slv_reg2;
21.     unsigned long slv_reg3;
22.     unsigned long oldreg;
23.
24.
25.     while(1){
26.         oldreg= slv_reg0;
27.         slv_reg0 = IR_MOD_mReadReg(BASE_ADDRESS, 0);
28.         slv_reg1 = IR_MOD_mReadReg(BASE_ADDRESS, 4);
29.         if(slv_reg0!= oldreg){ // if a change, print it our.
30.             printf("slv_reg0(message) = %d \n",slv_reg0);
31.             printf("slv_reg1(count)  = %d \n",slv_reg1);
32.
33.         }
34.     }
35.
36.
37.
38.     cleanup_platform();
39.     return 0;
40. }
```

With the code finish, we need to program our FPGA board and launch the GDB through picocom.

Results:

Once we have successfully program our FPGA board, we are ready to point the remote to our circuit and read its inputs.



Looking at the results, I was using the channel up button which is 114 once you convert its 12 bits into a decimal number. You can see with every message, our register 1 updates. It is also important to know, that sometimes I miss pointed the remote, so we ended with values that were quite off.

Conclusion:

Over all the lab was a success. I was able to create a custom IP, that would take in an IR_signal and decode it. There was some issues, at the beginning when I try to read the remote signals. I had to go back and adjust the ranges for some of my values. Having to use different circuits meant that I had to make sure, I was using reasonable ranges. Some times when you miss-point the remote, you can incorrect values, and sometimes you need to put the remote real close to the IR_reader. This caused some problems, and I feel like the circuit can be improve upon, however the lab was a good learning experience.

Questions:

What are the hexadecimal control codes for the following buttons:

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.

Looking at the results from my lab, I can see that that I got a range of 2 to 3 messages. Although, some of my messages were interrupted, we can still see that the remote sends more than 1 code.

(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?

To add this feature, I can simply check the old message with the new incoming message, if those messages are not the same, then I set the flag off otherwise it on. This signal could be used to queue new commands to a system such as a tv. This will prevent from messages interrupting each other.