ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Sunil P Khatri
(Lab exercise created and tested by Ramu Endluri, He Zhou and Sunil P Khatri)

Laboratory Exercise #7

IR-Remote Peripheral for Linux System

Objective

The objective of lab this week is to create and test hardware which receives and decodes Infrared (IR) signals from a universal television remote. You will create the IR detection hardware on a bread board and use Verilog to implement the decoding (i.e. demodulating) hardware within the FPGA. The demodulation hardware will be encapsulated within a custom IP peripheral so that messages from the TV remote are software accessible. An oscilloscope will be used to test the bread boarded hardware, while the demodulation process will be tested via a software routine executing on the Zynq Processor.

System Overview

The hardware system you will be creating in this lab is very similar to that of Lab 4, except the multiply block will be replaced by a custom peripheral for demodulating the IR signal from the universal remote. The IR_receive block refers to the analog hardware that will be implemented on the bread board. The labs that follow will integrate the IR peripheral into a more complex Linux System like that of Lab 4. However, for this lab, we want to keep the hardware as simple as possible. The sections that follow explain the theory of operation for the IR remote and the corresponding decoding logic.

- **IR remote control:** 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

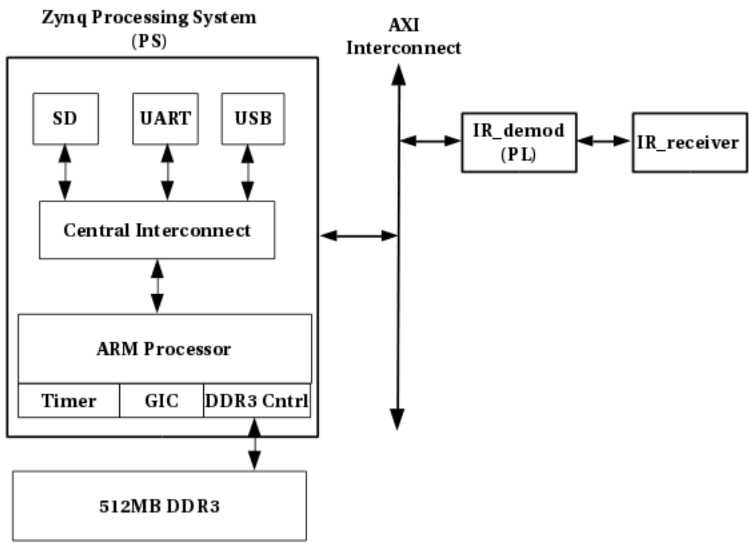

Figure 1: Zynq System Diagram

represent specific binary codes are transmitted from the IR remote . These binary codes correspond to commands such as 'Power On/Off', 'Volume Up/Down', etc. The receiving hardware decodes the pulses of light back into binary codes and uses them to command the device under control. The IR remote we will be using encodes the binary data using Pulse Width Modulation as seen in Figure 2. In addition to the binary data pulses, each pulse train consists of a 'START' pulse. A negative edge transition separates each pulse.

- **IR Receiver:** We will use an IR phototransistor to detect the pulse train generated by the remote control. The IR phototransistor is a transistor such that the emitter-collector current is proportional to the amount of infrared light that strikes the junction. Thus, the current through the emitter-collector junction increases when light falls on the phototransistor. We will create a simple detection circuit which converts the light pulses into a binary signal. The output of our detection circuit will feed into a demodulator circuit which will use the length of the various pulses to demodulate each control message.

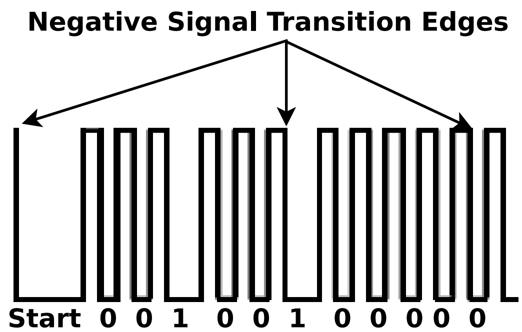

Figure 2: PWM Encoded Pulse Train for IR Remote

Procedure

1. You do not need to use the ZYBO board for the first part of lab. You will build your circuit on the breadboard provided and utilize the multimeter and oscilloscope as necessary.

   (a) Build the circuit shown in Figure 3. Use the voltage source on your workbench to provide V_{cc} = 3.3V to your circuit. The comparator pin-out diagram is provided for you in Figure 4, and you may use any of the four available comparators. Note the 10k Ω pull up resistor connected to the output terminal. This resistor is necessary because the output of the comparator is an open-collector output.

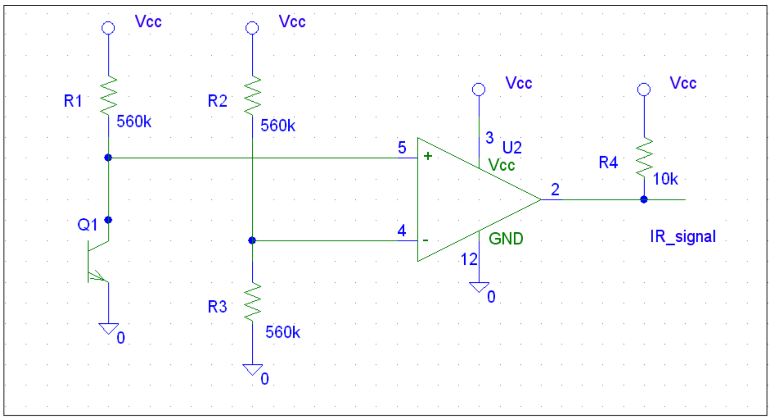

Figure 3: IR Receiver Circuit

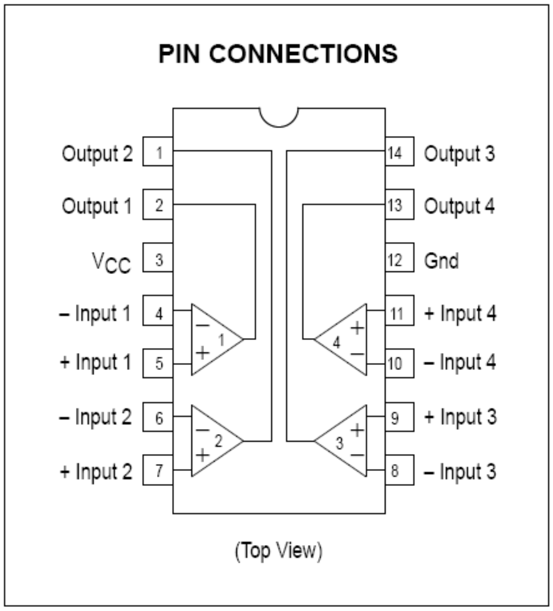

Figure 4: LM339 Comparator

(b) Observe the signal *IR_signal* on the oscilloscope, while pointing the remote control towards the IR phototransistor and pressing any key. You should see a pulse train on the oscilloscope where high and low are at *3.3V* and *0V* respectively. Demonstrate your progress to the TA.

(c) Each control message from the remote is 12-bits in length. The remote sends a sequence of messages when a button is held down. Hold down a button on the remote and hit 'Run/Stop' on the oscilloscope to freeze an entire message pulse train.

(d) Using Figure 2 as a guide, measure the length of the 'Start', '0', and '1' pulses. You will need to use the time axis cursors on the oscilloscope to do this. Try to make your measurements as accurate as possible and note them in your lab manual.

(e) Use the sample codes provided in Table 1 to ensure you are interpreting the pulse train properly. Read up on Pulse Width Modulation if the modulation scheme is unclear.

2. At this point, you are ready to begin building the FPGA hardware. The next few steps will guide you through the process of creating a base system for testing the custom IR peripheral and provide you with some guidelines for creating the IR demodulation hardware.

   (a) Create a block design in Vivado with everything shown in Figure1, except the IR hardware. If any doubts about the procedure refer to lab 4.

Key	Code
Volume Up	010010010000
Channel Up	000010010000
Channel 1	000000010000
Channel 2	100000010000

Table 1: Sample Command Codes

(b) The ir_demod module will be implemented on the FPGA. To set the clock frequency for the module follow the instructions below.

- Double click on the Zynq Processing System IP to customize it.
- Select 'Clock Configuration' and expand 'PL Fabric Clocks'.
- Set the frequency of FCLK_CLK0 to 75MHz for this lab.

(c) Use Vivado to create the template code for a custom peripheral with four software accessible registered as outlined in Lab 3. Label the peripheral 'ir_demod'.

(d) 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.

(e) Since our peripheral has an input pin (i.e. *IR_signal* from Figure 3), we must add it to the port list of our 'ir_demod_v1_SOO_AXI.v' file. Do this now using *IR_signal* as the port name. Use the auto generated comments as a guide for where to insert your changes.

(f) The 'ir_demod_v1_SOO_AXI' Verilog module is instantiated in the 'ir_demod_v1.v' file. Define the user port IR_signal here and connect it to the IR_signal defined in 'ir_demod_v1_SOO_AXI.v'.

(g) Now use the following guidelines to create user logic for demodulating the digital signal from the first part of the lab.

- Use a counter to measure the length of each pulse in order to determine its significance. Pulses in order of decreasing pulse length are as follows: 'Start', '1', and '0'. The appropriate count value for each pulse must be predetermined based on the rate of the system clock and the oscilloscope measurements made in Part 1 of the lab.
- Use a separate state register to keep track of which count value the counter has hit. For example, since a 'start' pulse is the longest, the counter will hit the count value for the '0' first, followed by the count value for '1', and finally the count value for 'start'.
- You will need to detect both negative and positive edges using edge detection circuitry. Negative edges signify the beginning of a pulse while positive edges signify the end of a pulse.
- Your logic will need an intermediate register to hold the bits of the IR message as they arrive. Depending on your implementation, your logic may also need a register to keep track of the current bit position of the arriving bit stream.

- Use slv_reg0 to hold the latest demodulated message. This register shall by loaded after the arrival of a complete 12-bit message.
- Use slv_reg1 to hold a running count of the number of messages that have been received.
- The third software accessible register may be used for debugging.
- Read the auto-generated codes in ‘ir_demod_v1_SOO_AXI.v’ carefully. If you use ‘S_AXI_ARESETN’ in your code, please notice that ‘S_AXI_ARESETN’ is active LOW.

(h) Once your source code modifications are complete, select the ‘Ports and Interfaces’ tab in the Package IP window. Click on the ‘Merge changes from Ports and interfaces Wizard’ at the top of the window.

(i) Re-Package IP and import your custom IR peripheral into your Zynq system.

(j) Make the ‘IR_signal’ port of your IR peripheral external by right clicking on the IR_signal and select ‘Make External’. This will create an external port with name ’IR_signal’. Then create an XDC file to connect the external signal to pin T20 on the ZYBO board.

(k) Create HDL wrapper, generate bitstream, and export to the SDK.

(l) Create a simple test application that continually polls your IR hardware registers and prints them in hexadecimal format to the screen when a change is detected. You do not need to implement a delay for this part. Do not forget to cast the base address of IP to type ‘u32’.

(m) Do NOT connect any hardware to the ZYBO board until the TA has inspected your setup. Have the TA verify this is correct.

(n) Connect $V_{cc} = 3.3V$, *GND*, and the appropriate *IR_signal* pin on the ZYBO board to your IR receiver circuit built in Part 1 of the lab.

(o) Make sure the JP5 is set to JTAG mode and verify that your system is working by creating a bitstream and downloading it to your ZYBO board. If your hardware is not operating properly, you may be required to use Vivado and SDK to debug your user logic. Within Vivado you can create a test module that produces a sample pulse train and then debug your hardware using the Vivado simulator.

Deliverables

1. [5 points.] Demo the working IR demodulation hardware to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [4 points.] Commented Verilog and C files.
4. [2 points.] The output of the Picocom terminal.

5. [4 points.] Answers to the following 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.

   (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.

   (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?