Gesture Recognition Remote Control

Abstract

The Gesture Recognition Remote Control system effectively replaces the functionality of the remote control for a Sony brand television. The user interface for the system is a wand, which the user can wave in one of eight gestures to send a specific command to the television. A color NTSC video camera tracks the movement of the wand while the user is making a gesture, and hardware programmed on the Field Programmable Gate Arrays of a 6.111 Labkit analyzes the video to determine which gesture the wand is making and responds by sending the corresponding infrared signal to the television using the correct bit encoding. The resulting behavior of this design turned out to be easy to use due to the simplistic nature of the gestures.

Table of Contents

Abstract i
1. OVERVIEW 1
2. DESCRIPTION 2
2.1 Frame Storage and Retrieval 3
2.2 Wand Filter Module 3
2.2.1 Pixel Filters 4
2.2.2 Finding Coordinates 5
2.3 Gesture Generator Module 7
2.3.1 The Start Button 7
2.3.2 The End Button 7
2.3.3 Testing 8
2.4 Infrared Signal Generator Module 8
2.4.1 Gesture to Command 8
2.4.2 Debugging the Infrared Signal 9
2.5 Infrared LED Circuit 10
3. CONCLUSION 10
4. REFERENCES 12
5. APPENDICES 13
5.1 Wand Filter Module 13
5.2 Gesture Generator Module 17
5.3 Infrared Signal Generator Module 19

LIST OF FIGURES

Figure 1: Above: Only red filter applied, noisy video. Below: Red filter and band-pass intensity filter removes extra noise. 4
Figure 2 : A map relating the direction of the gesture made by the user with the wand with its corresponding command. 7
Figure 3: Sony SIRC signals use a pulse-width encoding. 8
Figure 4: The entire signal, with its three phases. 9
Figure 5: The infrared LED in its circuit. 10

1. OVERVIEW

Remote controls for televisions have become fairly similar in the past years. The user presses a button on the remote which results in circuitry inside the remote changing the button press into a waveform which is fed to an infrared LED. This signal propagates throughout the room and the television receives and decodes the infrared signal into a simple command. My project aims to redesign the way television viewers interact with their TV sets, by changing the basic method of command generation from button pressing to gesturing with an LED wand device.

The Gesture Recognition Remote Control functions like a cross between a Wiimote and a conventional remote control. Its purpose is to combine both the functionality of a remote control and the movement recognition of the Wiimote into one device. This combination allows the user to make gestures by waving a red LED wand in one of eight gestures to generate commands such as change volume or channel, mute or power on/off within the receiving device which are sent through the device's infrared LED to the television. The receiving device consists of a video camera to capture the red LED's movements, a field-programmable gate array (FPGA) programmed to turn the video feed into a command waveform, and an infrared LED to turn this waveform into a signal that can be received by the TV. For televisions with multiple potential users, this allows for those who prefer to use this device over a conventional remote to be able to use it without preventing others from being able to use a conventional remote.

The design of the Gesture Recognition Remote Control system makes some assumptions about the space where it will be used. The first and most important assumption is that the general area around which the user will be using the red LED wand will be free of other red colors of the same intensity as the LED. This means that the red color generated by bright lights and the red of clothing will not be a problem. However, other red lights that fall in a narrow intensity band within the intensity of a red LED may cause problems for the device in recognizing gestures being made. Another assumption is that the receiving device is placed such that it has an unobstructed view of the television, or that at the very least, the infrared transmitting LED be able to have line of sight with the television.

In considering different solutions to remotely controlling a television, other options seemed to have some promise. One of these was the possibility of using a game controller, such as one from an Xbox 360, to create commands. The benefit of this idea was that gamers would only need to find one item instead of two, only needing their game controller to play their game on a television. The drawback of this design was that it targeted a limited market,

Xbox gamers. A different design would have to be made for each of the different game consoles. One of the strengths of the design I implemented was that it could be used by anyone, young or old, gamer or non-gamer.

The design of the Gesture Recognition Remote Control system will be examined in its entirety in the following section, focusing on the different modules which process the video feed input and produce the infrared signal output. Problems encountered while designing particular modules of the system and their solutions will also be explained. The problems encountered in the design often resulted with elegant solutions which required a minimum of adjustment by the user and his or her environment. Tests on different parts of the design will also be explained where applicable.

2. DESCRIPTION

The Gesture Recognition Remote Control system was written in Verilog and programmed on to a Field Programmable Gate Array (FPGA). Several modules of code transform the video feed input from the camera into a command waveform sent as output to a small circuit containing an infrared LED, which transmits the signal to a nearby television. The device used to create the gestures is a simple red LED with a thin covering of tissue paper to diffuse the emitted light. The video feed contains not only information about the presence of the red LED, but also contains a color image of the surrounding environment. Therefore, the code which analyzes this image must first extract the red LED from its background, and then determine the gesture being made.

Before describing each module of the code in depth, I will summarize each module and how the modules are connected. The video feed from the video camera cannot be analyzed directly, so the first step is to buffer the video feed into a zero bus turnaround (ZBT) memory unit, which can be either written to or read from continuously. Once the signal is buffered, it can be read from the memory, one pixel at a time. The first module's task is to recognize and black-out all pixels that are not part of the red LED wand. The second task is to find the coordinates of the remaining pixels which compose the red LED's image on the camera. Once these coordinates are determined, they can be passed to a gesture generating module. This module uses two buttons, one to indicate the start of a gesture, and one to indicate the end of a gesture. The direction of movement between the presses of these two buttons, start and end, is used to create a gesture, which is passed to the infrared signal generator module. This module creates a waveform which it sends to a small circuit containing an infrared LED. The LED transmits this signal, and a television within line of sight receives and decodes the signal into a command such as (volume up/down, channel up/down, power on/off, mute, etc).

2.1 Frame Storage and Retrieval

This module is responsible for the buffering of the analog video signal coming from the NTSC video camera in a zero-bus turnaround random access memory (ZBT RAM). This is the only module in this system in which the majority of the code was copied from a previously written module. This code was copied from the course website for the fall 2005 term of 6.111 (Introduction to Digital Systems Laboratory), in which several modules are offered as help for starting a project in several different directions. The code which was copied only recorded the intensity, and not the red and blue chrominance values associated with each pixel. This meant that the memory returned black and white video. The module was modified to store and retrieve the chrominance values for each pixel. From these three values, the red, green, and blue (RGB) values for each pixel were also calculated, while putting the other signals involved in the video feed through an appropriate delay to account for the time required to calculate the RGB values of the pixel. This module prepares all the necessary signals for the wand filter module to be able to filter out all but the red LED wand held by the user.

2.2 Wand Filter Module

Taking in the six values about each pixel, along with the pixels coordinates and other signals relating to the video feed, this module attempts to filter out all pixels which aren't part of the red LED wand in the video feed. Once it has only the relevant pixels, it uses an efficient algorithm to find the coordinates of the red LED wand in the video. The two pictures of the external monitor below show the image with only a red filter, and then the image again with both the red filter and a narrow band-pass intensity filter. It makes it very clear that the red LED is lost amid the other red noise in the background due to the lights above and the reflection of the lights off the white table below. The efficiency of the the band-pass filter in eliminating the noise should also be noted. Lastly, the stability of the coordinates found by the wand filter module is shown quite clearly here over the exposure time of a digital camera's picture.

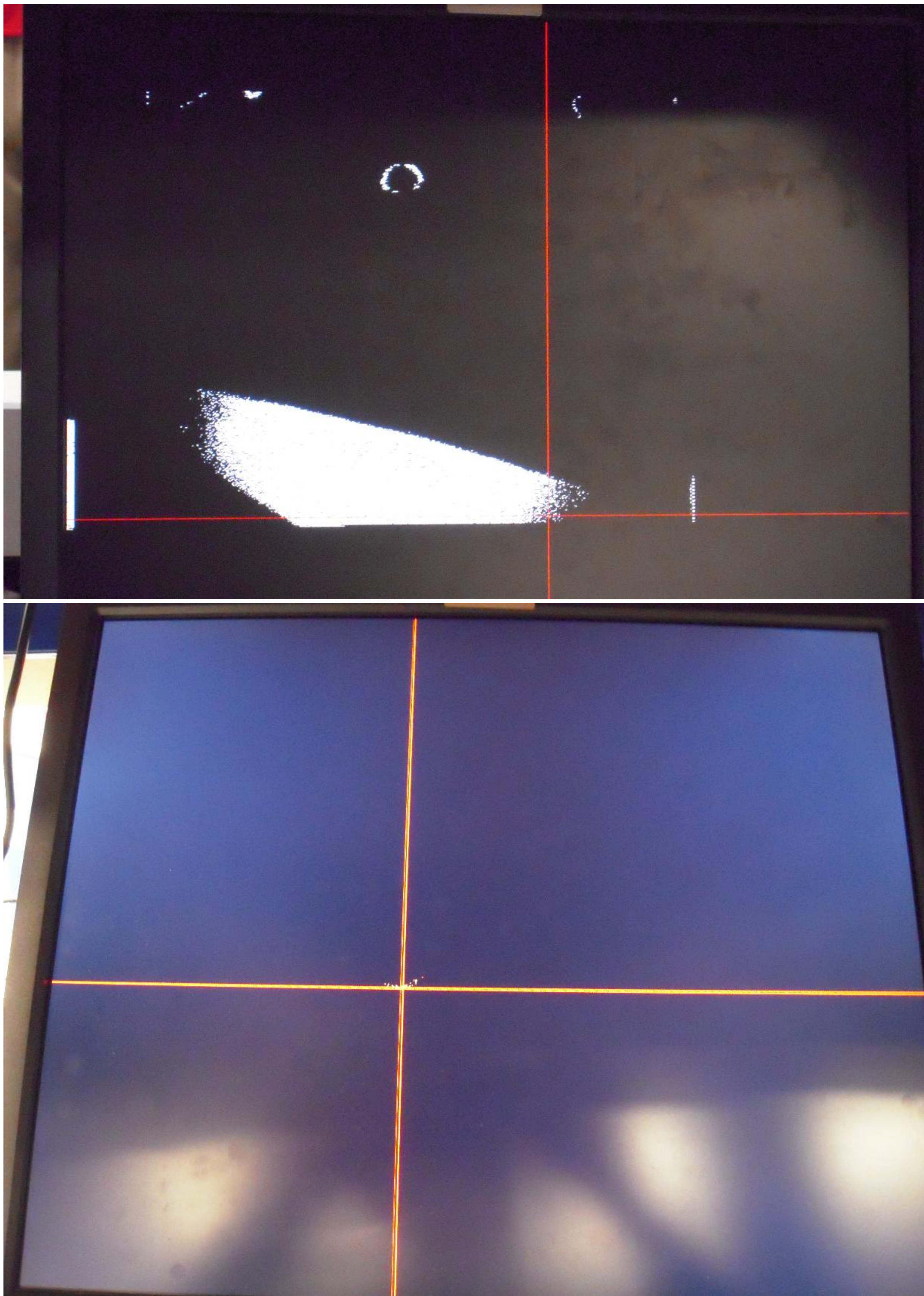

Figure 1: Above: Only red filter applied, noisy video. Below: Red filter and band-pass intensity filter removes extra noise.

2.2.1 Pixel Filters

The color video given to the wand filter module contains more than just the red LED wand. The background and surrounding environment around the wand need to be removed such that a clear picture of the red LED is left. While this could be done by physically manipulating the background to be black, requiring the user to wear a black glove and a black shirt while gesturing with the red LED. This poses a large restriction on the user, and removes a lot of the freedom inherent in the original design.

To get around this issue, I used a combination of two filters to filter out everything except the red LED. The first filter was a high-pass filter on the color red. Only pixels containing a very high red pixel color were let through this filter. This filter meant that most of the background and surrounding environment were removed from the picture, leaving the red LED untouched. However, the red LED wand was not the only source of red in the room. The lights put out a high intensity red color focused around the rim of the light fixtures. This red light also reflects off white surfaces, such as the lab tables, to create more background noise. On the other end of the spectrum, low intensity red colors, such as those on clothes, were also let through this filter. This is where you could get rid of these sources of noise by using a black background cloth, but there is a better way.

To isolate the red LED from all of the other sources of the color red in the video feed, a luminosity band-pass filter can be used. By filtering out all but a very narrow range of intensities, the high-intensity red of the lights and the low-intensity red of clothing can be successfully filtered out. This leaves only the pixels representing the wand, with the rest of the screen blacked out.

To monitor the progress of my code in filtering out all but the red LED, the pixels returning from the wand filter were passed to an external computer monitor. By using buttons on the FPGA lab kit, I was able to adjust the values of the high-pass and band-pass filters to optimize the quality of the filtering. The end result of these filters was a very distinct display of where the LED was with practically no extra noise from other sources of red light in the room.

2.2.2 Finding Coordinates

With just the pixels representing the wand remaining on the screen, it is now an easier task to find the coordinates of the wand on the screen. There are several ways to find the coordinates of the wand on the screen. One possible method would be to take the first pixel found, which would represent the top of a perfectly spherical wand representation on the screen. This method would be highly influenced by any added noise however, as any pixel that passed through the filter above the wand would then be taken to be the new top of the wand. This method's flaws make it easier to see the flaws in attempting to use a center of mass

algorithm on all of the pixels found. While a center of mass algorithm would be less influenced by added noise, it would still jump around more than is necessary.

The question of how to find the coordinates of the red LED wand without being influenced by relatively smaller quantities of noise was answered by a two algorithms that searched for the largest grouping of dots. These algorithms proved to be very efficient in not requiring massive calculations at any one time and were very immune to extra noise from the background.

The first algorithm analyzed each row of the video frame. A counter kept track of how many pixels passed through the filter in the row being counted, and at the end of the row, this value was compared with the last row. The larger number of pixels was stored, along with the row it came from, for each row. By the time the algorithm reached the last row, it had found the row with the largest number of pixels that passed though the two filters. This value was saved as the y-coordinate of the wand to be displayed as a horizontal part of the crosshairs displayed on the screen during the next frame.

To find the x-coordinate of the wand, the second algorithm examines the pixels on each row. For a given row, the algorithm searches for the largest series of uninterrupted pixels let through the filter. Every time the algorithm meets a pixel blocked by the filter, it waits for another pixel let through to begin a new count of a series of pixels. For a given frame, the algorithm has two counters. One counter keeps count of the number of pixels in the current series of pixels being counted as the frame is analyzed row by row, while the other records the hcount, or x-coordinate, of the end of the largest series of uninterrupted pixels found in its search of the current frame.

To test the accuracy of these two algorithms, crosshairs marking the position of the wand were added to the video output to the external LCD monitor. Although these two algorithms run independently of each other for each frame, they consistently and accurately pinpoint the location of the wand. Because the filters remove practically all extra noise from the signal, the crosshairs marking the location of the wand do not jump around either when a gesture is being made, or when the wand is not being used.

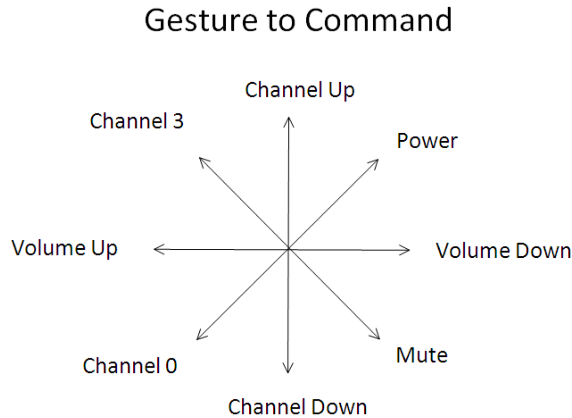

Figure 2 : A map relating the direction of the gesture made by the user with the wand with its corresponding command.

2.3 Gesture Generator Module

The purpose of this module is to determine when the user is making a gesture, and which gesture the wand is making while the user is making a gesture. Two buttons are used to signal the start and end of a gesture. These two buttons control the functionality of the gesture generator module, which outputs a gesture to the infrared signal generator module upon successful gesture recognition. There are 8 gestures which can be made, moving the wand in one of 8 cardinal directions, and one default gesture "NONE" which represents the absence of a gesture being made. The following figure shows the eight directions the user can move the wand while making a gesture. Each direction is labeled with the corresponding command scheduled to be sent to the television for each gesture.

2.3.1 The Start Button

When the start button is pressed, this module records the position of the wand into two sets of registers, one set which stores the start coordinates of the gesture, and one set which stores the end coordinates of a gesture. In this way, the algorithm that computes the gesture sees the start and end locations as the same, and outputs "NONE" as the gesture. This behavior is desirable, because without refreshing the end location, a gesture would be created from the difference between the new start location to the old end location.

2.3.2 The End Button

The end button, while being pressed, performs two important tasks. Like the start button, the end button places the location of the wand into the set of registers which store the end location of the wand. The start location registers are untouched, which creates a difference between these two sets of registers, assuming the user actually moved the wand a little. While the end button is held down, the output of this module is calculated through an algorithm that figures out approximately which of the 8 cardinal directions the wand traveled

from start to end locations. This direction is then output as a gesture. When the end button is released, the output gesture changes back to “NONE”, to signal the end of the gesture. As added functionality, if a user wishes to make a gesture repeatedly, they can press the end button repeatedly, or hold it down, to send the same command to the television repeatedly.

2.3.3 Testing

Testing this module was fairly straightforward. Two hex digits on the 16-hex display were used by this module for debugging. One of the hex digits output the current gesture being made, which allowed for the checking that the gesture went from the performed gesture back to “NONE” after the end button was depressed. The other hex digit displayed the last gesture which was made. This was more useful during testing of earlier models of the code, which didn’t hold the gesture constant while the end button was being held down, causing the gesture to flicker too quickly to see on the other hex digit.

2.4 Infrared Signal Generator Module

The purpose of the Infrared Signal Generator module is to take as input a gesture from the Gesture Generator module and return as output to a small circuit containing an infrared LED a waveform which encodes command and address bits as part of a message. This message is encoded using the Sony Infrared Control (SIRC) encoding scheme, such that a Sony television will respond appropriately to the infrared signal being sent.

2.4.1 Gesture to Command

One aspect of the encoding for the SIRC signal requires the commands being sent to be spaced such that there is 45ms time between the start of a signal and the start of the following signal. To ensure this, the code incorporates a counter that goes becomes non-zero one clock cycle every 45ms. By using a temporary register that is continually updated with the command bits to be sent via the infrared LED, when the 45ms counter goes high, the current command is loaded into a register that is only changed in this manner. This means that the command being sent over the infrared LED won’t suddenly change if a new gesture is given partway through the sending of one message through the infrared LED.

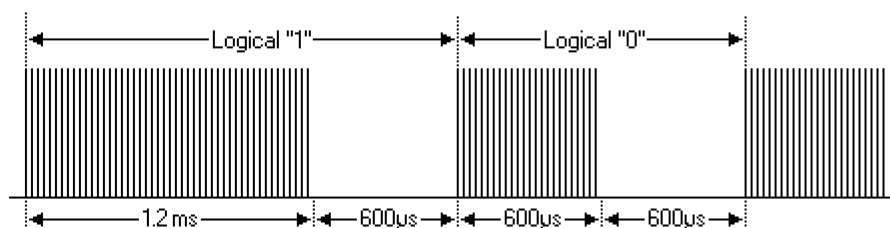

Figure 3: Sony SIRC signals use a pulse-width encoding.

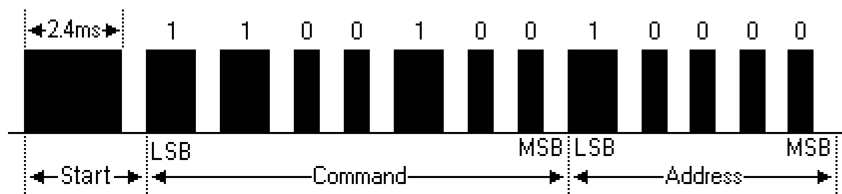

Figure 4: The entire signal, with its three phases.

The SIRC encoding is formatted such that there is always a high pulse followed by a low signal for each bit in the signal. The duration of the high pulse indicates whether the high-low pair represents a one or a zero. A duration of the high pulse of 1.2ms encodes a high pulse, while 0.6ms encodes a zero. The low pulse is always 0.6ms in duration. The entire signal is divided into three parts: the start, the command, and the address phases of the message. The start phase is always the same, and consists of a 2.4ms high signal followed by a 0.6ms low signal. The command signal, encoding which command (change volume, channel, etc) is sent following the start signal, with the least significant bit being sent first. Following this 7-bit command signal is the 5-bit address signal. This signal denotes which type of Sony machine the command is intended for (1 for TV, 2 for VCR, etc). Furthermore, the high signals sent to the infrared LED are not constant high signals, but are pulsed on and off at 40 kHz. This is achieved by using the logical AND on the signal and on a clock that alternates value between 1 and 0 with a frequency of 40 kHz. While the gesture is “NONE”, the output is held low and the infrared LED does not send any signals.

2.4.2 Debugging the Infrared Signal

The first version of this module was debugged using ModelSim’s simulation of the output given a gesture input. After successfully changing the module to produce the right output in ModelSim, I tried connecting the circuit containing the infrared LED to the output waveform of this module, which looked correct in ModelSim. What was found while examining the waveform with a digital oscilloscope was that only the start and the first bit of the signal encoding were making it to the circuit. After several changes, the digital oscilloscope showed the desired results, of a start bit followed by 12 bits representing the command and address phases of the message.

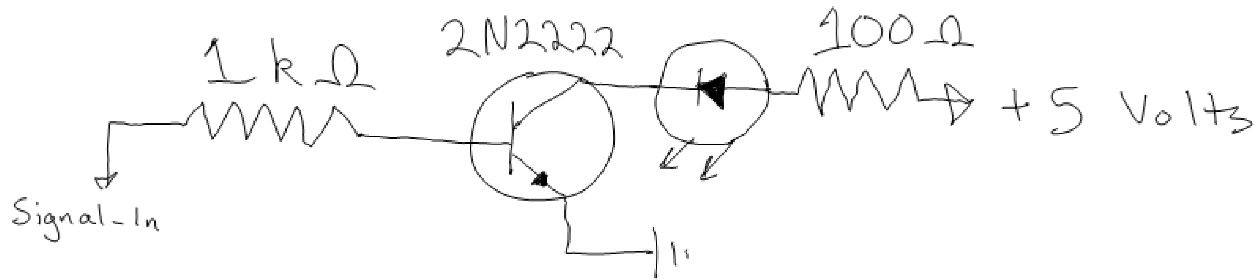

Figure 5: The infrared LED in its circuit.

2.5 Infrared LED Circuit

The circuit containing the infrared LED consists of a 5.0 volt power supply, the infrared LED, a bipolar junction transistor, and the signal output from the Infrared Signal Generator module through the 6.111 labkit. The reason for using the bipolar junction transistor in the circuit was an uncertainty over the current supplied by the output pins on the labkit. The circuit made it possible for the digital output of the labkit to allow current to either halt or pass through the infrared LED, turning it on and off following the signal supplied by the labkit which enters the above diagram through the path labeled Signal_In.

3. CONCLUSION

The Gesture Recognition Remote Control system turns a user's movement of a red LED wand into an infrared signal encoding the command associated with the gesture made by the user. The movement of the red LED wand is recorded by a NTSC video camera. The video camera has its analog signal converted into a digital signal which is stored in a zero-bus turnaround (ZBT) memory buffer. The information about each frame of the video camera feed is retrieved from the memory buffer one pixel at a time. These pixels are analyzed by the Wand Filter module, which filters out pixels not associated with the LED wand and finds the coordinates of the remaining pixels which represent the location of the wand. This location is passed to the Gesture Generator module, which uses button presses to signal the start and end of a gesture. At the end of a gesture, the module calculates which of eight gestures was made by the user. The gesture being made, or gesture "NONE" if the user is not making a gesture, is passed to the IR Signal Generator module. This module takes in the gesture information, and sends a signal encoded with the SIRC encoding scheme to an infrared LED. The infrared LED broadcasts the signal, and a nearby Sony television responds to one of the eight commands that it is given.

Testing was conducted for each module to ensure the proper functioning of each part of each module. The wand filter module had two main tests. The first allowed for the tester to change the cut-off values on the high-pass color and band-pass intensity filters. The results of these changes were seen in real time on an external monitor, so the optimal filter values could

be found efficiently without the need to recompile the code for each new set of filter values. The second test was the displaying of crosshairs marking the location of the wand as found by the module. This test easily showed that earlier algorithms attempting to find the coordinates of the wand had trouble without the intensity filter, with the crosshairs often jumping around the screen. This prompted the need for better filtering, which led to the band-pass intensity filter. Testing for the Gesture Generator module was more straightforward, in that it involved printing the current and last gesture seen to two hex digits on the 16-digit hex display on the 6.111 labkit. This allowed for the checking of whether the code accurately discovered which gesture the user was making with the wand. The Infrared Signal Generator module was more complicated to debug. ModelSim was utilized to perfect small imperfections in the waveform, but when the signal looked perfect in ModelSim, the signal failed when given to the LED wand. A digital oscilloscope revealed the problem, and a second version of the module was able to produce the correct output on the digital oscilloscope, at which point the infrared LED was able to successfully send commands to the television.

The Gesture Recognition Remote Control succeeded in its goal of allowing a user to make gestures with a LED wand which could be changed into infrared signals sent to a television. One shortcoming was the need for button presses to indicate the start and end of a gesture being made. An improvement for the future of this system would be to detect a jump in the number of pixels meeting the filter requirements, and thus detecting the turning on and off of the wand in its starting and ending of a gesture.

4. REFERENCES

Bergmans, San. “Sony SIRC Protocol. ”Knowledge Base. 23 Apr 2008. 25 Jan 2009
http://www.sbprojects.com/knowledge/ir/sirc.htm

Terman, Chris. “zbt_6111_sample.zip” 6.111 Fall 2005.
<http://web.mit.edu/6.111/www/f2005/index.html>

5. APPENDICES

The appendices contain the three modules in which the majority of the code written was my own. Other modules which were either copied entirely from the course website listed in references or copied and modified slightly.

5.1 Wand Filter Module

```
`timescale 1ns / 1ps

module WandFilter(clk, reset, dButton_L, dButton_R, dButton_U, dButton_D, switchC,
pixel_Y_curr, pixel_Cr_curr, pixel_Cb_curr,
pixel_R, pixel_G, pixel_B,
hcount, vcount, hsync, vsync,
pixel_RGB, x, y, Y_Val_Crosshairs, Y_Val_T, R_Val_T, Y_Val_Max,
Wand_X, Wand_Y);
   input clk;
   input reset;
         input dButton_L;
         input dButton_R;
         input dButton_U;
         input dButton_D;
         input switchC;
   input [7:0] pixel_Y_curr;
   input [7:0] pixel_Cr_curr;
   input [7:0] pixel_Cb_curr;
   input [7:0] pixel_R;
   input [7:0] pixel_G;
   input [7:0] pixel_B;
   input [10:0] hcount;
   input [9:0] vcount;
   input hsync;
         input vsync;
         output [23:0] pixel_RGB;
         output [10:0] x;
         output [9:0] y;
         output [7:0] Y_Val_Crosshairs;
         output [7:0] Y_Val_T;
         output [7:0] R_Val_T;
         output [7:0] Y_Val_Max;
         output reg [10:0] Wand_X;
         output reg [10:0] Wand_Y;

        reg [23:0] reg_pixel_RGB;
        assign pixel_RGB = reg_pixel_RGB;

        reg [10:0] reg_x = 0;
        reg [9:0] reg_y = 0;
```

```
reg [7:0] Y_Val_Crosshairs = 8'b0;

reg [7:0] R_Val_T = 8'b0;
reg [7:0] Y_Val_T = 8'h02;  // was 8'h9E, new usage
reg [7:0] Y_Val_Max = 8'h90; // was A0, dont use notecard covering on LED

reg [10:0] Last_Hcount = 0;
reg [9:0] Last_Vcount = 0;


reg [10:0] X_Red_Most;          // x coord of most pixels illuminated up to current vcount
reg [10:0] X_Red_Draw;          // x coordinate of last frame's position of wand
reg [10:0] Red_Pixels_InRow;
reg [20:0] Running_Max_In_Rows;
reg [9:0] Row_Red_Most;
reg [9:0] Draw_Row_Wand;

reg [10:0] First_Pixels_In_Row[2:0];
reg [2:0] First_RowPixels_Found = 0;
reg [10:0] Last_Pixels_In_Row[2:0];
reg [2:0] Last_RowPixels_Found = 0;

reg [24:0] Sum_Xcoord_Row = 0;
reg [10:0] Draw_Col_Wand = 0;

reg Hsync_Low_Delay = 0;

reg [12:0] Running_Sum_Row_Pixels = 0;
reg [12:0] Old_Running_Sum_Row_Pixels = 0;
reg [10:0] Running_Sum_Row_A = 0;
reg [10:0] Running_Sum_Row_B = 0;
reg [10:0] Running_Sum_Row_C = 0;

reg Found_Red_Pixel;
reg [10:0] Pixels_Red_This_Bunch;
reg [10:0] Pixels_Red_Last_Bunch;
reg [10:0] This_Bunch_Hcount;
reg [10:0] Last_Bunch_Hcount;

always @(hcount) begin
        if (!hsync) begin
                Hsync_Low_Delay <= 1; // enter waiting phase, eval new Row_Red_Most

                Running_Sum_Row_A <= Red_Pixels_InRow;
                Running_Sum_Row_B <= Running_Sum_Row_A;
                Running_Sum_Row_C <= Running_Sum_Row_B;
                Old_Running_Sum_Row_Pixels <= Running_Sum_Row_Pixels;
```

```
				Running_Sum_Row_Pixels <= (Red_Pixels_InRow + Running_Sum_Row_A +
Running_Sum_Row_B);

				if (Running_Sum_Row_Pixels > Old_Running_Sum_Row_Pixels) begin
					Row_Red_Most <= vcount;			// stores vcount of row with
most pixels illuminated
					Running_Max_In_Rows <= Red_Pixels_InRow;
				end
			end
			else if (Hsync_Low_Delay) begin // lose one pixel's data, OK because we don't use first
50 columns
				Red_Pixels_InRow <= 0;// new row: reset # red pixels found in row
				Hsync_Low_Delay <= 0; // exit waiting phase during low hsync, reset
Red_Pixels_InRow
			end

			else if ((hcount > 50) && (hcount < 750) && (vcount > 90) && (vcount < 560)) begin
				if ((pixel_Y_curr >= (Y_Val_Max - Y_Val_T)) && (pixel_R >= 8'b1111_1110) &&
(pixel_Y_curr <= Y_Val_Max)) begin
					Red_Pixels_InRow <= Red_Pixels_InRow + 1;

					// Find largest bunch or grouping of pixels in a row
					if (!Found_Red_Pixel) begin  // start "bunch"
						Found_Red_Pixel <= 1;
					end
					else begin
						This_Bunch_Hcount <= hcount;
						Pixels_Red_This_Bunch <= Pixels_Red_This_Bunch + 1;
					end
				end
				else begin // end bunch, compare size with last largest bunch, record if this
bunch is larger
					Found_Red_Pixel <= 0;
					Pixels_Red_This_Bunch <= 0;
					Pixels_Red_Last_Bunch <= (Pixels_Red_This_Bunch >
Pixels_Red_Last_Bunch)
													?
Pixels_Red_This_Bunch : Pixels_Red_Last_Bunch;
					Last_Bunch_Hcount <= (Pixels_Red_This_Bunch >
Pixels_Red_Last_Bunch)
													?
This_Bunch_Hcount : Last_Bunch_Hcount;
				end
			end
		end


		always @(negedge vsync) begin				// 60 times a second
```

```
			Draw_Row_Wand <= Row_Red_Most;		// store last frame's y-coord for red
wand, to draw this frame
			Draw_Col_Wand <= Last_Bunch_Hcount; // same with x-coord



			if (switchC)	// Change coordinates of crosshairs
				begin
					if (dButton_L)
						reg_x <= reg_x - 1;
					else if (dButton_R)
						reg_x <= reg_x + 1;
					if (dButton_U)
						reg_y <= reg_y - 1;
					else if (dButton_D)
						reg_y <= reg_y + 1;
				end
			else	// Change Luminance Threshold (D/U) or L Max (L/R)
				begin
					if (dButton_L)
						Y_Val_Max <= (Y_Val_Max > 0) ?  Y_Val_Max - 1 : Y_Val_Max;
					else if (dButton_R)
						Y_Val_Max <= (Y_Val_Max < 8'b1111_1111) ?  Y_Val_Max + 1 :
Y_Val_Max;
					if (dButton_U)
						Y_Val_T <= (Y_Val_T < 8'b1111_1111) ?  Y_Val_T + 1 : Y_Val_T;
					else if (dButton_D)
						Y_Val_T <= (Y_Val_T > 0) ?  Y_Val_T - 1 : Y_Val_T;
				end

		end


		always @(posedge clk) begin
			if ((hcount == reg_x) && (vcount == reg_y)) begin // give luminance at center of
crosshairs
				reg_pixel_RGB <= {8'b1111_1111, 8'b0, 8'b0};
				Y_Val_Crosshairs <= pixel_Y_curr;
				end
			// Draw Crosshairs
			else if ((hcount == reg_x) || (vcount == reg_y) || (vcount == Draw_Row_Wand) ||
(hcount == Draw_Col_Wand))
				reg_pixel_RGB <= {8'b1111_1111, 8'b0, 8'b0};
			// Draw red dot if pixel is detected as part of LED
			else if ((pixel_Y_curr >= (Y_Val_Max - Y_Val_T)) && (pixel_R >= 8'b1111_1110) &&
(pixel_Y_curr <= Y_Val_Max))
				reg_pixel_RGB <= {pixel_R, pixel_G, pixel_B};		// pass through red
pixels above threshold & under max luminance
```

```
            else
                reg_pixel_RGB <= 24'b0; // black out all pixels that don't meet threshold
    end

    assign x = reg_x;  // crosshair coords
    assign y = reg_y;

    always @(posedge clk) begin  // LED coords, passed to GesGenB
            Wand_X <= Draw_Col_Wand;
            Wand_Y <= {1'b0, Draw_Row_Wand[9:0]};
    end

endmodule
```

5.2 Gesture Generator Module

```
`timescale 1ns / 1ps

module GesGenB(clk, reset, vsync, ButtonA, ButtonB, ButtonC, ButtonD, Wand_X,
Wand_Y, Gesture, First_Gesture, Pos_X, Pos_Y);
    input clk;
    input reset;
       input vsync;
       input ButtonA;
       input ButtonB;
       input ButtonC;
       input ButtonD;
    input [10:0] Wand_X;
    input [10:0] Wand_Y;
    output [3:0] Gesture;
    output reg [3:0] First_Gesture;
       output reg [10:0] Pos_X;
       output reg [10:0] Pos_Y;

      // gestures
      parameter UP = 0;
      parameter UP_RIGHT = 1;
      parameter RIGHT = 2;
      parameter DOWN_RIGHT = 3;
      parameter DOWN = 4;
      parameter DOWN_LEFT = 5;
      parameter LEFT = 6;
      parameter UP_LEFT = 7;
      parameter STILL = 8;
      parameter NONE = 9;


      reg [3:0] Gesture = NONE;

      reg [10:0] Start_X;
      reg [10:0] Start_Y;
```

```
	reg [10:0] End_X;
	reg [10:0] End_Y;
	//reg [10:0] Third_X;
	//reg [10:0] Third_Y;

	reg Waiting = 1;

	reg [26:0] Half_Sec = 0;


	/*
		Records start and end of gesture,
		Show last location stored (Pos_X, Pos_Y) on hex display
	*/
	always @(posedge clk) begin
		Start_X <= (ButtonA) ? Wand_X : Start_X;
		Start_Y <= (ButtonA) ? Wand_Y : Start_Y;
		End_X <= (ButtonA) ? Wand_X : ((ButtonB) ? Wand_X : End_X);
		End_Y <= (ButtonA) ? Wand_Y : ((ButtonB) ? Wand_Y : End_Y);
		//Third_X <= (ButtonA) ? Wand_X : ((ButtonB) ? Wand_X :
((ButtonC) ?  Wand_X : Third_X));
		//Third_Y <= (ButtonA) ? Wand_Y : ((ButtonB) ? Wand_Y :
((ButtonC) ?  Wand_Y : Third_Y));
		//Pos_X <= (ButtonA) ? Start_X : ((ButtonB) ? End_X : ((ButtonC)
? Third_X : Pos_X));
		//Pos_Y <= (ButtonA) ? Start_Y : ((ButtonB) ? End_Y : ((ButtonC)
? Third_Y : Pos_Y));
		Pos_X <= (ButtonA) ? Start_X : ((ButtonB) ? End_X : Pos_X);
		Pos_Y <= (ButtonA) ? Start_Y : ((ButtonB) ? End_Y : Pos_Y);
	end
	/*
		If a gesture is being made, calculate which gesture it is and
send gesture to irSigGen.
		Record last gesture entered in First_Gesture.
	*/
	always @(*) begin		//(*)
		Waiting = (ButtonB) ? 0 : 1;

		if (Waiting)  begin
			Gesture = NONE;
		end
			// debugging
		else if (ButtonD) begin
			Gesture = UP_RIGHT;
			First_Gesture = UP_RIGHT;
		end
		//end debugging
		else if (End_Y < Start_Y - 50) begin //--UP--
			if (End_X < Start_X - 50) begin  // LEFT
				Gesture = UP_LEFT;
				First_Gesture = UP_LEFT;
				Waiting = 1;
			end
			else if (End_X > Start_X + 50) begin // RIGHT
				Gesture = UP_RIGHT;
				First_Gesture = UP_RIGHT;
				Waiting = 1;
```

```
                end
                else begin
                        Gesture = UP;
                        First_Gesture = UP;
                        Waiting = 1;
                end
        end
        else if (End_Y > Start_Y + 50) begin // --DOWN--
                if (End_X < Start_X - 50) begin  // LEFT
                        Gesture = DOWN_LEFT;
                        First_Gesture = DOWN_LEFT;
                        Waiting = 1;
                end
                else if (End_X > Start_X + 50) begin // RIGHT
                        Gesture = DOWN_RIGHT;
                        First_Gesture = DOWN_RIGHT;
                        Waiting = 1;
                end
                else begin
                        Gesture = DOWN;
                        First_Gesture = DOWN;
                        Waiting = 1;
                end
        end
        else begin // no UP/DOWN
                if (End_X < Start_X - 50) begin  // LEFT
                        Gesture = LEFT;
                        First_Gesture = LEFT;
                        Waiting = 1;
                end
                else if (End_X > Start_X + 50) begin // RIGHT
                        Gesture = RIGHT;
                        First_Gesture = RIGHT;
                        Waiting = 1;
                end
        end
    end

endmodule
```

5.3 Infrared Signal Generator Module

```
`timescale 1ns / 1ps

module irSigGen(clk, reset, ButtonC, ButtonD, Gesture, Signal_OutW, Cmd_Out);
    input clk;
    input reset;
    input ButtonC;
    input ButtonD;
    input [3:0] Gesture;
    output Signal_OutW;
    output [6:0] Cmd_Out;

       parameter NONE = 9;  // gesture NONE = 9
```

```
	reg [32:0] tv_cmd;
	reg Signal_Out;  // irLED signal output

	reg [18:0] Shift; // 39000 clock for each bit change
	reg [18:0] Shift_Pulse_Clock;
	reg Reset_Shift;

	reg [22:0] Repeat_Counter;
	reg Reset_Repeat;

	reg [10:0] Carrier_Signal;  //40khz carrier signal 0 to 1624 repeating
	reg Carrier_Clock = 0;         //carrier envelope

	reg [32:0] Signal_Sent = 33'b0;  // Stored signal to be sent

	reg [32:0] Power = 33'b11110_110_10_110_10_110_10_10__110_10_10_10_10
;
	reg [32:0] Chan_Up =
33'b11110_10_10_10_10_110_10_10__110_10_10_10_10___00;
	reg [32:0] Chan_Dn =
33'b11110_110_10_10_10_110_10_10__110_10_10_10_10___0;
	reg [32:0] Vol_Up =
      33'b11110_10_110_10_10_110_10_10__110_10_10_10_10___0;
	reg [32:0] Vol_Dn =
      33'b11110_110_110_10_10_110_10_10__110_10_10_10_10;
	reg [32:0] Mute =
      33'b11110_10_10_110_10_110_10_10__110_10_10_10_10___0;
	reg [32:0] Zero =
      33'b11110_110_10_10_110_10_10_10__110_10_10_10_10___0;
	reg [32:0] Three =
      33'b11110_10_110_10_10_10_10_10__110_10_10_10_10___00;

	//Command  #  bits
	//zero          9  0001001
	//three    2  0000010
	//chan+    16 0010000
	//chan-    17 0010001
	//vol+          18 0010010
	//vol-          19 0010011
	//mute          20 0010100

	/*
		Shift Clock shifts bits in command being processed at
appropriate time (600us).
		Carrier Clock provides a 40khz carrier envelope
		for the signal being transmitted to the irLED.
	*/
	always @(posedge clk) begin
		// pulse clock
		Shift <= ((Shift == 39_000) || Reset_Shift) ? 0 : Shift +
1;
		Shift_Pulse_Clock <= (Shift == 39_000);
		Repeat_Counter = ((Repeat_Counter == 2_925_000) ||
Reset_Repeat) ? 0 : Repeat_Counter + 1;
		// 40khz carrier
```

```
                  Carrier_Signal = (Carrier_Signal == 1624) ? 0 :
Carrier_Signal + 1;
                  Carrier_Clock = (Carrier_Signal == 1624) ? ~Carrier_Clock :
Carrier_Clock;
       end

      /*
            Continually update tv_cmd with the gesture being given to
irSigGen.
      */
      always @(*) begin                               // (*)
            case (Gesture)
                  0: tv_cmd = Chan_Up;                      // up
            //
                  1: tv_cmd = Three;                        // up-right
      //chan up   //five
                  2: tv_cmd = Vol_Up;                       // right
            //
                  3: tv_cmd = Zero;                         // down-right
      // chan down
                  4: tv_cmd = Chan_Dn;                      // down
            //
                  5: tv_cmd = Mute;                         // down-left
      // vol down
                  6: tv_cmd = Vol_Dn;                       // left
            //
                  7: tv_cmd = Power;                        // up-left
      // vol up
                  8: tv_cmd = 33'b0;
      // still //not used
                  9: tv_cmd = 33'b0;
      // NONE -- low output
            endcase
      end

      /*
            If time to deliver a new command through irLED, store tv_cmd in
Signal_Sent,
            and hold that constant for the duuration of sending the signal.
            While sending signal, shift bit being sent every shift clock
pulse,
            while using 40khz carrier envelope for the signal being sent.
      */
      always @(posedge clk) begin
            Signal_Out <= (Carrier_Clock && Signal_Sent[32]);
            if (Shift_Pulse_Clock) begin
                  Signal_Sent <= {Signal_Sent[31:0], 1'b0};
            end
            else if (Repeat_Counter == 2_925_000) begin
                  Signal_Sent <= tv_cmd;
            end
      end

      assign Cmd_Out = 7'b0;
      assign Signal_OutW = Signal_Out;

endmodule
```