Virtual Drum Set

6.111 Final Project Report

Rishi Naidu
Luis Fernandez
12/12/12

Abstract:

It is said that air drumming provides the same level of satisfaction to the user as drumming on a physical drum set does. Knowing this, why should everyone have to own a drum set to start drumming? For our final project, we implemented a gesture-controlled virtual drum set on the 6.111 FPGA lab kit to provide a simple and fun way for people to enjoy drumming.

Our Virtual Drum Set provides users with three elements of a real drum set that are the snare, floor tom and cymbal. The user wears two distinct colored gloves and his motions are tracked by the camera system. By just air drumming, the user can enjoy playing the drum elements where the playback is through a speaker. In order to provide good feedback the user is provided with a user interface to keep track of his hand motions over different elements, which also gives user a feel for playing drums in reality. The user can also play the drum elements along with a song stored in the lab kit.

Overview

With an increase in the number of applications that use gesture control, the virtual drum set is another step towards the implementation of gesture recognition features in musical instruments. Our virtual drum set gives the user the freedom of drumming at basic level without owning a drum set, which is expensive, noisy and occupies a lot of space. The user can control volume of drumbeats that are played back through a speaker and can even limit the playing to oneself by using headphones. It also allows the user to play the drumbeats along with a two-minute song that is stored in the flash rom of the lab kit.

The implementation of our project consist of camera system that is used to track the user’s motion, a limited number of drum set elements and basic playback of sounds of the drums that user plays. The user wears two distinct color gloves to simplify the motion tracking by the camera. The motion tracking system distinguishes between hitting motions in different areas in front of the user and emulates sound of distinct drums based on hitting area. A user interface allows user to keep track of hand motions over the drums to provide a real experience of drumming.

Overall, the user would air drum freely and hear playback from the drums he is virtually playing, along with listening to a song in tandem to his actions.

Modules:

The project is partitioned into two distinct sections: the *video module* and the *audio module*. As the name suggests the video module controls camera, display and gestures while the audio module controls the sound features for drum sound playback.

The video module is divided into two separate parts: the *gesture detection module* and the *collision engine & display module*. The *gesture detection module* takes in the input stream of the video camera to detect the position and velocity of the user's hands and output a specific pair of coordinates corresponding to the location of the center of mass of the user's hands. The *collision engine and display module* then takes this information and not only displays it on the screen along with a user interface that represents the hands and the drums but also determines at what point a specific drum is being played. In short, this module is responsible for generating the graphical interface of the project and is also responsible with determining what drum is being played at what time by the user.

The audio module is also divided into two separate parts: the *game audio module* and the *music audio module*. The game audio module takes in the input from the output of the video module to determine what drum sound to play at what time and correctly output it to the ac97 interface for playback in the lab kit. The music audio module is responsible for reading a predetermined song from flash memory and also interfacing with the ac97 interface for playback in the lab kit.

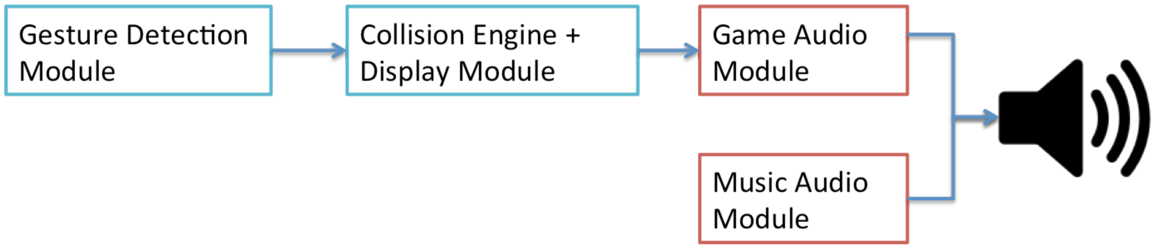

Figure 1. Overall Block Diagram

Gesture Detection Module (Luis)

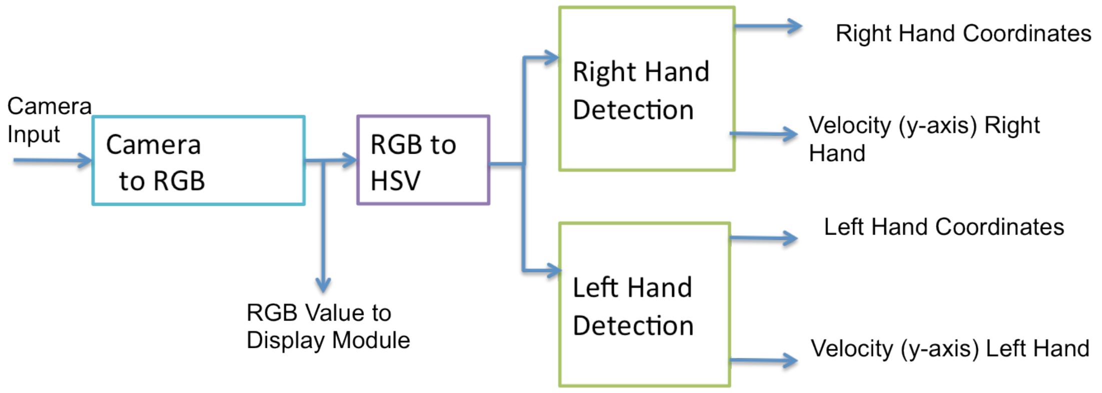

Figure 2. Gesture Detection Module

The Gesture Detection module is responsible for tracking the user's movements throughout space. Supplying the user with red and green colored gloves allows our camera system to recognize and filter their positions in free space.

The module uses a modified version of the camera modules supplied by the 6.111 staff in order to correctly transform the camera NTSC signal into a full range 24 bit RGB stream, which is consequently stored in ZBT ram inside the lab kit. Regarding the use of ZBT ram, one video frame is stored at any given moment in ZBT memory. Each frame consists of 786,432 (1024x768) pixels, where each pixel is stored as 3 bytes (for RGB representation). We store two pixels per address in ZBT RAM, but we truncate each pixel to 18 bits instead of 24 bits (3 bytes) to accommodate to the maximum of 36 bits per address line of storage. This yields a total ZBT memory use of 4.5*786432 = 3.53 MB (out of 4MB available). After this, we also used the staff supplied RGB to HSV conversion module in order to be able to use a more robust filtering method in HSV color space.

The detection of the user's hands is done in the *hand_detection* module, which is responsible for taking in the HSV stream of bits captured by the video camera and detecting which pixels correspond to the hand of the user. This is done by comparing the hue of the incoming pixels to the specific color of the glove that the user is wearing. If a pixel is determined to match the color of the glove, then its coordinates are added to an accumulator register that corresponds to the total addition of the coordinates of all the pixels that match our filter. Then, at the end of every video frame, the total number in the accumulator register is divided by the total number of matching pixels, in order to obtain an average of the location of the center of mass of all the matching pixels. Furthermore, the module averages the position of the current center of mass and the three previous ones, in order to stabilize the position of the center of mass and reduce noise present in the system.

The *hand_detection* module also keeps track of the previous location of the center of mass and then compares the change in location for every frame in order to calculate the velocity at which the hand is moving.

In the actual project two of these modules are instantiated, changing a parameter in each to specify which color they are looking to match (red for the left hand and green for the right hand). They are responsible for supplying the user interface modules with the information about the user's hand movements.

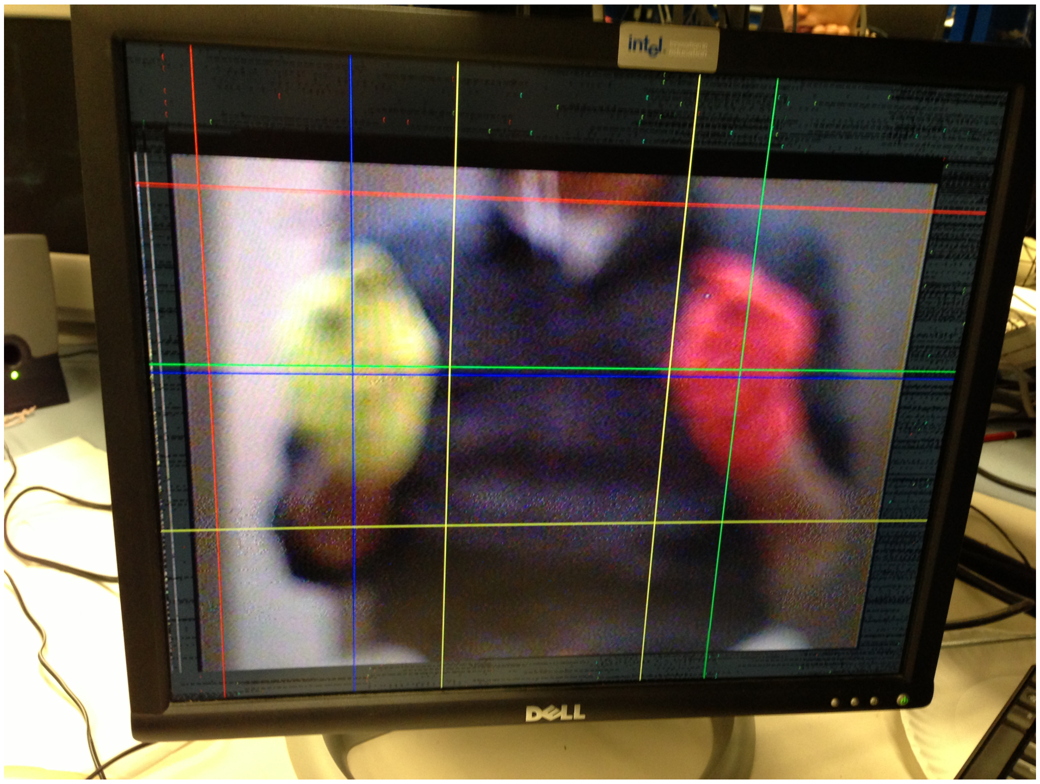

Figure 3. Hand Tracking

Apart from this, several debugging modules were created to help with the development process. Most notably, the *crosshairs* module is responsible for outputting the video feed together with crosshairs that allow the developer to see what the HSV value is of a specific location of the video feed as well as the location of the center of masses that are being calculated by the *hand_detection* module (see Figure 3).

Collision Engine Module (Rishi)

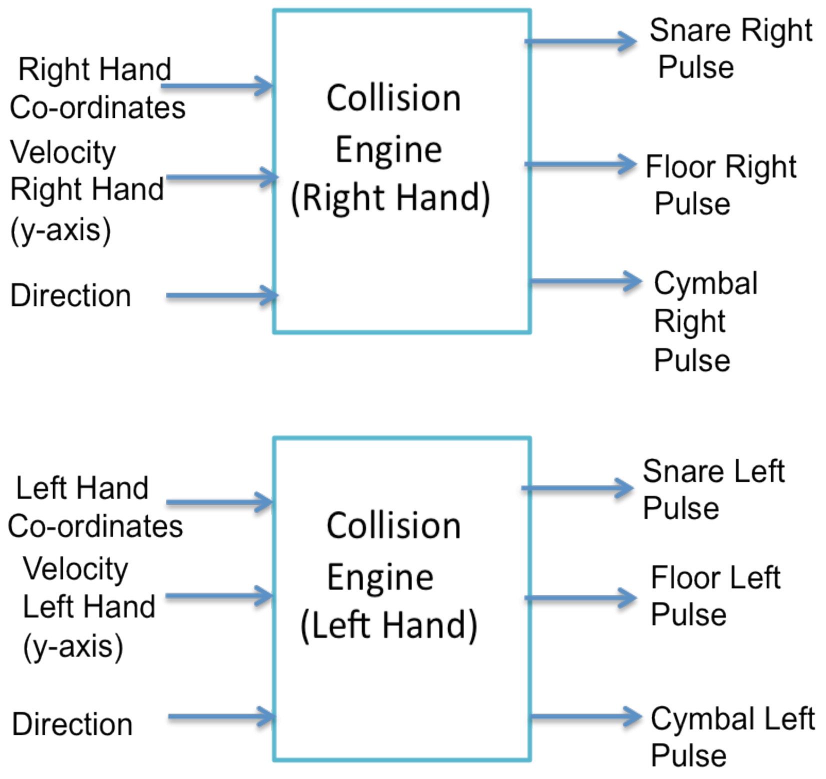

Figure 4 Collision Engine Block Diagram

The Collision Engine Module is responsible to detect the drum hit based on the inputs from the Gesture Detection Module. The inputs to this module are the x and y coordinates of the center of mass of the hand of the user, its velocity and the direction of movement of the hand, which specifies if the hand is going in downward direction or upward direction.

This module is a two state Finite State Machine (FSM) that detects a hit on a drum based on the velocity, direction and a threshold level along y-axis of the video frame (See Figure 5). Once the drum hit is determined it detects the drum that has been actually hit based on the position of hand along the x-axis.

There are two instantiations of this module (one for each hand) in the main lab kit module. This module then outputs a pulse for each of the drum elements, which specifies the hit on the drum. The outputs from this module are the cymbal pulse, snare pulse and floor pulse.

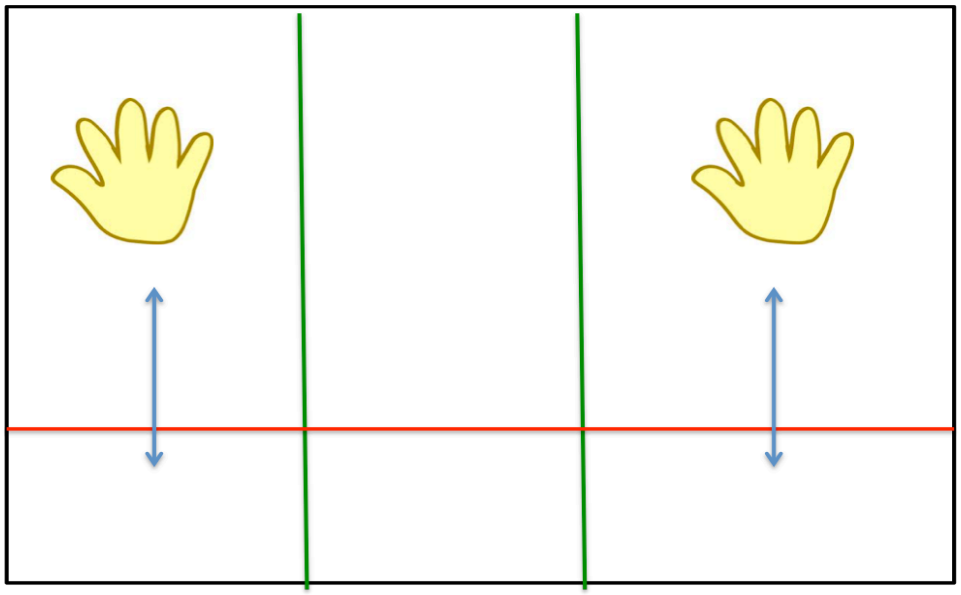

Figure 5. Collision Engine Threshold levels

Figure 5 shows how the screen is divided into different sections. The left most section determines snare drum, the middle section determines floor hat and the right most section determines the cymbal.

A drum hit is determined when a hand crosses the specific threshold (depicted in the figure 5 by a red line) with a certain velocity.

Display Module (Rishi)

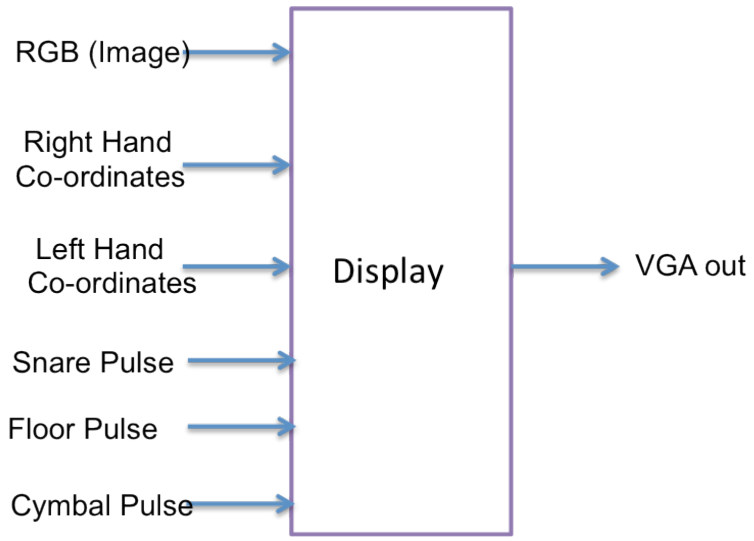

Figure 6. Display Module

The Display Module generates the user interface that provides the user with proper feedback of the position of the sticks along each of the drum elements. The display consists of images of different drum elements and the stick images. The images are of the snare drum, floor hat and cymbal.

All these images were first edited using GIMP and then converted to 256-color bmp format. This bmp image was then converted to a COE file using a MATLAB script created by Edgar Twigg of Conductor Hero Project. This COE file was then used to store the information in BRAM, created by using the IP Core Gen in Xilinx tools.

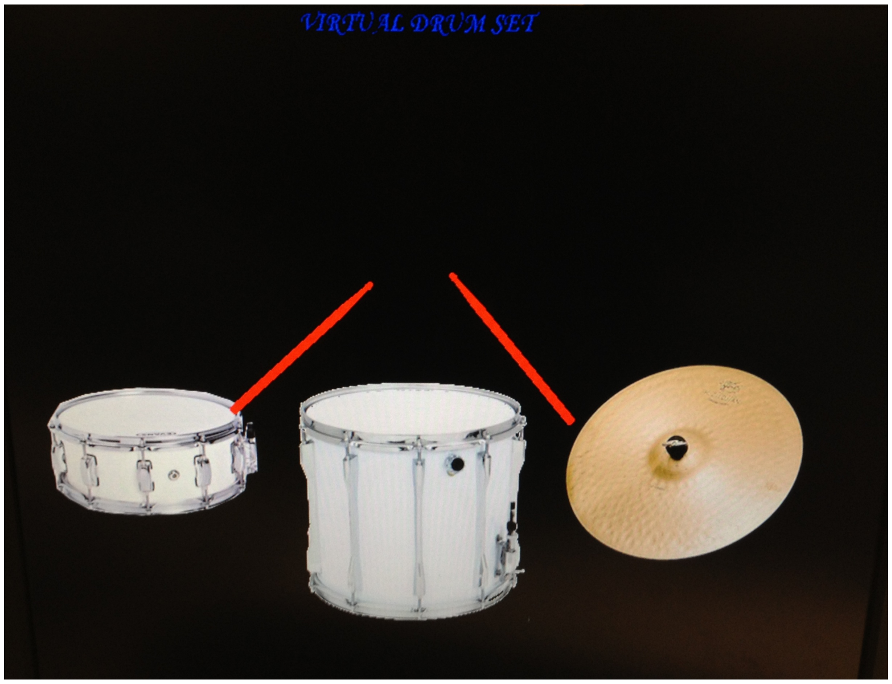

Figure 7. User Interface

The MATLAB script generates the image address that has to be passed through a look up table for red, green and blue in order to generate the required 24-bit value for each pixel.

In the implementation of the project a single stick image stored in BRAM was used to view both sticks, which saved space in BRAM. To achieve this the stick image was made a perfect square with the stick (red in color) across the diagonal of the square, using GIMP. Then the image address given to BRAM module to fetch image data was changed in such a way that the image obtained looked like the stick on the right.

Similarly an animation for motion of the stick was done in such a way that when there is a hit, the image address given to BRAM module changes for the duration of the drum hit to obtain the corresponding hit motion effect.

Figure 8. Left Stick

Figure 9. Right Stick

Figure 10. Left Stick Animation

Figure 11. Right Stick Animation

BRAM Memory for Display Images

BRAM total space = 2952k bits = 369k bytes.

Color Mapping Memory = 3*256bytes (For Red, Green & Blue) = 768 bytes.

Display Images	Width (pixels)	Height (Pixels)	Pixels Memory (bytes)	Color Mapping Memory (bytes)	Total Memory (bytes)
Snare Drum	225	180	40.5k	768	41.268k
Floor Tom	225	225	50.625k	768	51.393k
Cymbal	225	200	45k	768	45.768k
Stick	140	140	19.6k	768	20.368k
Cymbal (For motion)	225	200	45k	768	45.768k
Title Image	300	50	15k	768	15.768k
				Memory Required	220.333k

Table 1. BRAM Memory Usage

Memory required in BRAM = 220.33k bytes
Memory available in BRAM = 369k bytes
Memory remaining in BRAM = 148.67k bytes

Design Tradeoff with animation of stick

The stick animation obtained was a nice way to show the motion of the stick at time of a drum hit. However this limited the space along the y-axis for user feedback, as the stick was 140 pixels long and the motion could be detected only if the stick is 140 pixels above the threshold of the drum hit. This made us to move away from the animation for a drum hit and have the hit be represented when tip of stick corresponds to the threshold level, which added more space for user feedback.

Game Audio Module (Rishi)

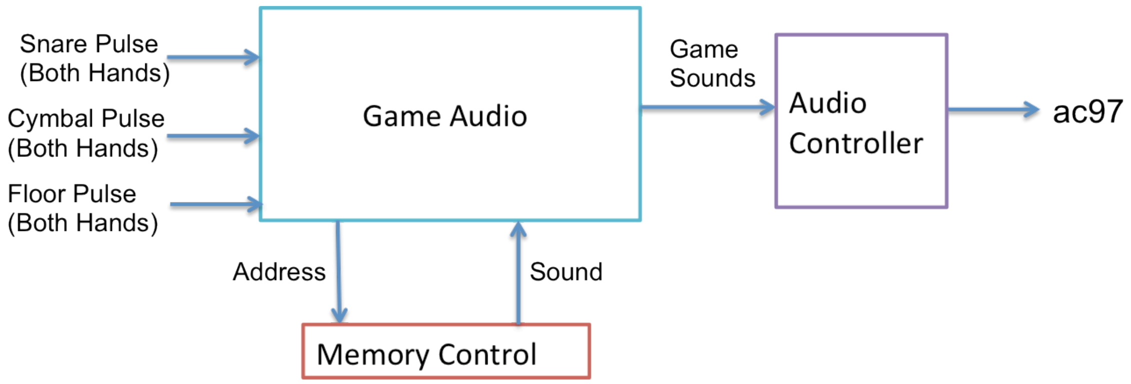

Figure 12. Game Audio Module

The game audio module is responsible to playback the drum sounds stored in BRAM at the time of hit along with the song stored in flash ROM. The drum sounds are first converted to 8-bit WAV file with a sampling rate of 48kHz using Audacity. The sampling rate is 48kHz as this makes it easy to send data to AC97 (Audio Chip in Lab Kit), which samples data at 48kHz. This WAV file is then converted to COE file using a MATLAB script that is then stored in BRAM using the IP Core Gen of Xilinx tools.

Input to this module includes pulse signals obtained from the Collision Engine Module for different drum elements. The sound stream from individual BRAM are then added along with the song data from the flash ROM and fed into AC97 for playback. The option of playing the song is controlled using a switch on lab kit.

Having an individual state machine for each element of the drum set allows us to play two drums together at a time, which gives it a feel of playing a real drum set.

Music Audio Module (Luis)

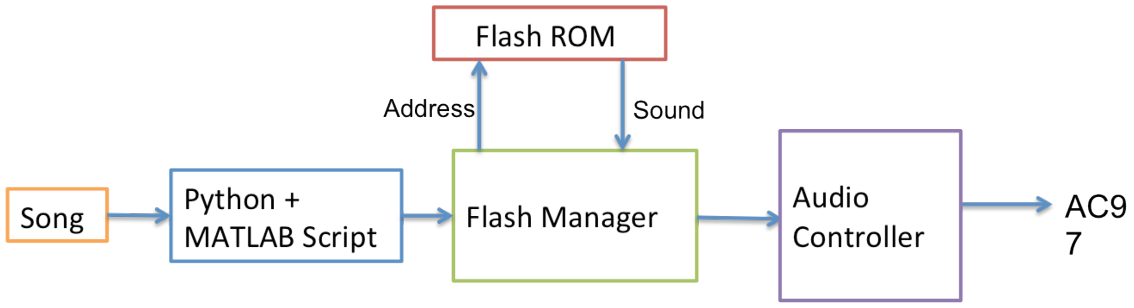

Figure 13. Music Audio Module

The Music Audio Module is responsible for transmitting a song from a computer to the labkit's Flash ROM memory in order for it to be played back during the game.

The first part of the module consists of various software scripts on the computer side, which are responsible for preprocessing the song in order to be able to correctly transmit it to the lab kit. The song is first converted into a 48000Hz wav file using Dylan Sherry's handy MATLAB script, which assures that the playback speed on the AC97 interface will be correct. After this, we modified Yuta Kuboyama's *soundDemo.m* MATLAB script in order to generate a custom *coe* file to be fed into the lab kit. This custom *coe* file is read by a Python script we wrote that communicates with a USB FIFO that is responsible for physically connecting our computer with the lab kit.

On the lab kit side, the module instantiates copies of the staff provided *usb_input* and *flash_manager* modules with some modifications made in order to make sure that they are able to interact efficiently and are able to accommodate the information being transmitted by the computer. Some logic is also added to be able to read the memory once all the information is written in order to be able to correctly output a stream of bits to be supplied to the AC97 controller.

Regarding memory use, the lab kit Flash ROM has 16M bytes of space available, corresponding to roughly 8M memory addresses with 16 bits available per address. In our project, we store 8 bits per address and use about 5M addresses, which correspond to a two minute long song. If necessary, the module can be easily modified to use all 16 bits of space per address and thus increase storage capacity.

Testing Methodology

Each module was first developed and debugged independently, through a combination of ModelSIM simulations and real world incremental testing. We extensively used the alphanumeric display on the lab kit to output values relevant to the operation of the modules. In particular for the video modules, we tested the tracking system in a variety of scenarios (changing lightning, environment, etc), in order to increase its robustness. After every sub-module was independently developed and tested, we incrementally integrated the system, carefully debugging every integration via real world tests. We This ‘piece by piece’ debugging and integration approach helped us minimize bugs and allowed us to be more efficient when putting the whole system together.

Conclusion

We have successfully implemented a robust and fun way to enjoy a virtual drum set by air drumming freely. We believe we successfully achieved our design goals and are satisfied by the performance of the system after realizing how much fun it is to actually use it. The Gesture Detection Module was able to correctly identify the presence and motions of the user’s hands through the video camera. The Collision Engine successfully recognized the presence of drum hits in the user’s motion. The Game and Music Audio Modules were able to correctly reproduce their corresponding sound effects. Finally, the Display Module was able to put everything together into an effective user interface that finalizes what we consider is a very novel and entertaining virtual drum set.

Acknowledgements

We would like to thank Gim Hom, Kevin Zheng and Dylan Sherry for their continued help, advice and company during the long development process.

Schedule

Week of 11/05:

- Start implementing video modules
- Block diagram conference

Week of 11/12:

- Finish implementation of video modules
- Project design presentation
- Checklist conference
- Start implementing audio modules

Week of 11/19:

- Debug and testing of video modules
- Debug audio modules

Week of 11/26:

- Finish implementing audio modules
- Wrap up video modules testing
- Start testing and debug of audio modules

Week of 12/3:

- Debug all modules
- Test whole design
- Prepare final presentation

Appendix: Verilog Source Code

Top Level Module:

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//          during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)
//
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09
//
// Use ramclock module to deskew clocks;  GPH
// To change display from 1024*787 to 800*600, use clock_40mhz and change
// accordingly. Verilog ntsc2zbt.v will also need changes to change resolution.
//
```

```
// Date   : 10-Nov-11

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2011-Nov-10: Changed resolution to 1024 * 768.
//                                    Added back ramclok to deskew RAM clock
//
```

```
// 2009-May-11: Fixed memory management bug by 8 clock cycle forecast.
//            Changed resolution to  800 * 600.
//            Reduced clock speed to 40MHz.
//            Disconnected zbt_6111's ram_clk signal.
//            Added ramclock to control RAM.
//            Added notes about ram1 default values.
//            Commented out clock_feedback_out assignment.
//            Removed delayN modules because ZBT's latency has no more effect.
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//            "disp_data_out", "analyzer[2-3]_clock" and
//            "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//            actually populated on the boards. (The boards support up to
//            256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//            value. (Previous versions of this file declared this port to
//            be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb
devices
//            actually populated on the boards. (The boards support up to
//            72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

module zbt_6111_sample(beep, audio_reset_b,
                        ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
		tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
		tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
		tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
		tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

		ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
		ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

		ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
		ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

		clock_feedback_out, clock_feedback_in,

		flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
		flash_reset_b, flash_sts, flash_byte_b,

		rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

		mouse_clock, mouse_data, keyboard_clock, keyboard_data,

		clock_27mhz, clock1, clock2,

		disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
		disp_reset_b, disp_data_in,

		button0, button1, button2, button3, button_enter, button_right,
		button_left, button_down, button_up,

		switch,

		led,

		user1, user2, user3, user4,

		daughtercard,

		systemace_data, systemace_address, systemace_ce_b,
		systemace_we_b,		systemace_oe_b,		systemace_irq,
systemace_mpbrdy,

		analyzer1_data, analyzer1_clock,
		analyzer2_data, analyzer2_clock,
		analyzer3_data, analyzer3_clock,
		analyzer4_data, analyzer4_clock);
```

```
	output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
	input  ac97_bit_clock, ac97_sdata_in;

	output [7:0] vga_out_red, vga_out_green, vga_out_blue;
	output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
		vga_out_hsync, vga_out_vsync;

	output [9:0] tv_out_ycrcb;
	output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
		tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
		tv_out_subcar_reset;

	input  [19:0] tv_in_ycrcb;
	input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
		tv_in_hff, tv_in_aff;
	output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
		tv_in_reset_b, tv_in_clock;
	inout  tv_in_i2c_data;

	inout  [35:0] ram0_data;
	output [18:0] ram0_address;
	output    ram0_adv_ld,    ram0_clk,    ram0_cen_b,    ram0_ce_b,    ram0_oe_b,
ram0_we_b;
	output [3:0] ram0_bwe_b;

	inout  [35:0] ram1_data;
	output [18:0] ram1_address;
	output    ram1_adv_ld,    ram1_clk,    ram1_cen_b,    ram1_ce_b,    ram1_oe_b,
ram1_we_b;
	output [3:0] ram1_bwe_b;

	input  clock_feedback_in;
	output clock_feedback_out;

	inout  [15:0] flash_data;
	output [23:0] flash_address;
	output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
	input  flash_sts;

	output rs232_txd, rs232_rts;
	input  rs232_rxd, rs232_cts;

	input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;
```

```
  input  clock_27mhz, clock1, clock2;

  output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
  input  disp_data_in;
  output  disp_data_out;

  input  button0, button1, button2, button3, button_enter, button_right,
         button_left, button_down, button_up;
  input  [7:0] switch;
  output [7:0] led;

  inout [31:0] user1, user2, user3, user4;

  inout [43:0] daughtercard;

  inout  [15:0] systemace_data;
  output [6:0]  systemace_address;
  output systemace_ce_b, systemace_we_b, systemace_oe_b;
  input  systemace_irq, systemace_mpbrdy;

  output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                analyzer4_data;
  output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

  ////////////////////////////////////////////////////////////////////////////
  //
  // I/O Assignments
  //
  ////////////////////////////////////////////////////////////////////////////

  // Audio Input and Output
  assign beep= 1'b0;
  //assign audio_reset_b = 1'b0;
  //assign ac97_synch = 1'b0;
  //assign ac97_sdata_out = 1'b0;
/*
*/
  // ac97_sdata_in is an input

  // Video Output
  assign tv_out_ycrcb = 10'h0;
  assign tv_out_reset_b = 1'b0;
  assign tv_out_clock = 1'b0;
  assign tv_out_i2c_clock = 1'b0;
  assign tv_out_i2c_data = 1'b0;
  assign tv_out_pal_ntsc = 1'b0;
  assign tv_out_hsync_b = 1'b1;
  assign tv_out_vsync_b = 1'b1;
  assign tv_out_blank_b = 1'b1;
  assign tv_out_subcar_reset = 1'b0;

  // Video Input
  //assign tv_in_i2c_clock = 1'b0;
  assign tv_in_fifo_read = 1'b1;
  assign tv_in_fifo_clock = 1'b0;
  assign tv_in_iso = 1'b1;
  //assign tv_in_reset_b = 1'b0;
  assign tv_in_clock = clock_27mhz;//1'b0;
  //assign tv_in_i2c_data = 1'bZ;
  // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
  // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

  // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
  assign ram0_data = 36'hZ;
  assign ram0_address = 19'h0;
  assign ram0_clk = 1'b0;
  assign ram0_we_b = 1'b1;
  assign ram0_cen_b = 1'b0;  // clock enable
*/

/* enable RAM pins */

  assign ram0_ce_b = 1'b0;
  assign ram0_oe_b = 1'b0;
  assign ram0_adv_ld = 1'b0;
  assign ram0_bwe_b = 4'h0;

/**********/

  assign ram1_data = 36'hZ;
  assign ram1_address = 19'h0;
  assign ram1_adv_ld = 1'b0;
  assign ram1_clk = 1'b0;
  //These values has to be set to 0 like ram0 if ram1 is used.
  assign ram1_cen_b = 1'b1;
  assign ram1_ce_b = 1'b1;
  assign ram1_oe_b = 1'b1;
  assign ram1_we_b = 1'b1;
  assign ram1_bwe_b = 4'hF;

  // clock_feedback_out will be assigned by ramclock
  // assign clock_feedback_out = 1'b0;  //2011-Nov-10
  // clock_feedback_in is an input

        //esto lo comente out
  /*
  // Flash ROM
  assign flash_data = 16'hZ;
  assign flash_address = 24'h0;
  assign flash_ce_b = 1'b1;
  assign flash_oe_b = 1'b1;
  assign flash_we_b = 1'b1;
  assign flash_reset_b = 1'b0;
  assign flash_byte_b = 1'b1;
  // flash_sts is an input
        */


  // RS-232 Interface
  assign rs232_txd = 1'b1;
  assign rs232_rts = 1'b1;
  // rs232_rxd and rs232_cts are inputs

  // PS/2 Ports
  // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

  // LED Displays
/*
  assign disp_blank = 1'b1;
  assign disp_clock = 1'b0;
  assign disp_rs = 1'b0;
  assign disp_ce_b = 1'b1;
  assign disp_reset_b = 1'b0;
  assign disp_data_out = 1'b0;
*/
  // disp_data_in is an input
  // Buttons, Switches, and Individual LEDs
  //lab3 assign led = 8'hFF;
  // button0, button1, button2, button3, button_enter, button_right,
  // button_left, button_down, button_up, and switches are inputs

  // User I/Os
  assign user1 = 32'hZ;
  assign user2 = 32'hZ;
  assign user3 = 32'hZ;
  assign user4 = 32'hZ;

  // Daughtercard Connectors
  assign daughtercard = 44'hZ;

  // SystemACE Microprocessor Port
  assign systemace_data = 16'hZ;
  assign systemace_address = 7'h0;
  assign systemace_ce_b = 1'b1;
  assign systemace_we_b = 1'b1;
  assign systemace_oe_b = 1'b1;
  // systemace_irq and systemace_mpbrdy are inputs

  // Logic Analyzer
  assign analyzer1_data = 16'h0;
  assign analyzer1_clock = 1'b1;
  assign analyzer2_data = 16'h0;
  assign analyzer2_clock = 1'b1;
  assign analyzer3_data = 16'h0;
  assign analyzer3_clock = 1'b1;
  assign analyzer4_data = 16'h0;
  assign analyzer4_clock = 1'b1;


        //aqui hice los cambios del clock!!
/*
  ////////////////////////////////////////////////////////////////////////////
  // Demonstration of ZBT RAM as video memory

  // use FPGA's digital clock manager to produce a
  // 65MHz clock (actually 64.8MHz)
  wire clock_65mhz_unbuf,clock_65mhz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
  // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
  // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk1 is 37
  BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
*/


//   wire clk = clock_65mhz;  // gph 2011-Nov-10

  ////////////////////////////////////////////////////////////////////////////
  // Demonstration of ZBT RAM as video memory

  // use FPGA's digital clock manager to produce a
  // 40MHz clock (actually 40.5MHz)
  wire clock_40mhz_unbuf,clock_40mhz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 2
  // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
  // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk1 is 37
  BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

  //wire clk = clock_40mhz;


        wire locked;

        //assign clock_feedback_out = 0; // gph 2011-Nov-10

  ramclock rc(.ref_clock(clock_40mhz), .fpga_clock(clk), //aqui cambie de 65 a 40
                          .ram0_clock(ram0_clk),
                          //.ram1_clock(ram1_clk),    //uncomment if
ram1 is used
                          .clock_feedback_in(clock_feedback_in),
                          .clock_feedback_out(clock_feedback_out),
.locked(locked));


  // power-on reset generation
  wire power_on_reset;    // remain high for first 16 clocks
  SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
                  .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
  defparam reset_sr.INIT = 16'hFFFF;
```

```
	// ENTER button is user reset
	wire reset,user_reset;
	debounce db1(power_on_reset, clk, ~button_enter, user_reset);
	assign reset = user_reset | power_on_reset;
		//////////////////////////////////////////////////////////////////////
	///////////////// Display module for debugging/////////////////////
		///////////////////////////////////////////////////////////////////////

	reg [63:0] dispdata;
	display_16hex hexdisp1(reset, clk, dispdata,
				disp_blank, disp_clock, disp_rs, disp_ce_b,
				disp_reset_b, disp_data_out);

	// generate basic XVGA video signals
	wire [10:0] hcount;
	wire [9:0]  vcount;
	wire hsync,vsync,blank;
	xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

	// wire up to ZBT ram

	wire [35:0] vram_write_data;
	wire [35:0] vram_read_data;
	wire [18:0] vram_addr;
	wire       vram_we;

	wire ram0_clk_not_used;
	zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
			vram_write_data, vram_read_data,
			ram0_clk_not_used,       //to get good timing, don't connect
ram_clk to zbt_6111
			ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

	// generate pixel value from reading ZBT memory
	wire [23:0] vr_pixel; //[7:0] vr_pixel;
	wire [18:0]    vram_addr1;

	vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
			 vram_addr1,vram_read_data);

	// ADV7185 NTSC decoder interface code
	// adv7185 initialization module
	adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
			  .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                    .tv_in_i2c_clock(tv_in_i2c_clock),
                    .tv_in_i2c_data(tv_in_i2c_data));

  wire [29:0] ycrcb;    // video data (luminance, chrominance)
  wire [2:0] fvh;       // sync for field, vertical, horizontal
  wire     dv;  // data valid

  ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                    .ycrcb(ycrcb), .f(fvh[2]),
                    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

  // code to write NTSC data to video memory

  wire [18:0] ntsc_addr;
  wire [35:0] ntsc_data;
  wire      ntsc_we;
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb,
                  ntsc_addr, ntsc_data, ntsc_we, switch[6]);

  // code to write pattern to ZBT memory
  reg [31:0]    count;
  always @(posedge clk) count <= reset ? 0 : count + 1;

  wire [18:0]   vram_addr2 = count[0+18:0];
  wire [35:0]   vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
                       : {4{count[3+4:4],4'b0}} );

  // mux selecting read/write to memory based on which write-enable is chosen

  wire  sw_ntsc = ~switch[7];
  wire  my_we = sw_ntsc ? (hcount[0]==1'd1) : blank; //era hcount[1:0]
  wire [18:0]   write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
  wire [35:0]   write_data = sw_ntsc ? ntsc_data : vpat;

//  wire        write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//  assign      vram_addr = write_enable ? write_addr : vram_addr1;
//  assign      vram_we = write_enable;

  assign        vram_addr = my_we ? write_addr : vram_addr1;
  assign        vram_we = my_we;
  assign        vram_write_data = write_data;

  // select output pixel data
```

```
	reg [23:0]	pixel; //esto hay q cambiar

		reg [23:0]	delayed_pixel; //esto hay q cambiar
	reg	b,hs,vs;

		//hsv conversion

		wire [23:0] hsv_data;

		rgb2hsv
hsvconverter(clk,1'b0,pixel[23:16],pixel[15:8],pixel[7:0],hsv_data[23:16],
			hsv_data[15:8],hsv_data[7:0]);


	always @(posedge clk)
	 begin
		pixel <= switch[0] ? {hcount[8:6],5'b0} : vr_pixel; //esto hay q cambiar
		b <= blank;
		hs <= hsync;
		vs <= vsync;
	 end

		 ///////////////
		// project starts
		//////////////



		//delay hcount and vcount to account for delay in hsv converter
		wire [10:0] hcount_delayed;
		wire [9:0] vcount_delayed;
		delayNsize #(22,11) delay_h(clk, hcount, hcount_delayed);
		delayNsize #(22,10) delay_v(clk, vcount, vcount_delayed);

		//aqui hago el detection de una mano!
		wire interesting;
		wire [10:0] x_coord;
		wire [9:0] y_coord;
		wire [23:0] colored_pixel;

		//direction and velocity for 1 hand
		wire direction;
		wire [9:0] velocity;
		wire [24:0] x_accumulator;
		wire [24:0] y_accumulator;

		hand_detection
detect(clk,1,hsv_data,hcount_delayed,vcount_delayed,pixel,interesting,colored_pixel,x_
coord,y_coord,direction,velocity,x_accumulator,y_accumulator);

		//aqui viene la deteccion de la otra mano
		wire interesting2;
		wire [10:0] x_coord2;
		wire [9:0] y_coord2;
		wire [23:0] colored_pixel2;

		//direction and velocity for second hand
		wire direction2;
		wire [9:0] velocity2;

		wire [24:0] x_accumulator2;
		wire [24:0] y_accumulator2;

		hand_detection
detect2(clk,0,hsv_data,hcount_delayed,vcount_delayed,pixel,interesting2,colored_pixel
2,x_coord2,y_coord2,direction2,velocity2,x_accumulator2,y_accumulator2);

		//aqui viene la parte de cross hairs

		wire [23:0] rgb_value;
		wire [23:0] hsv_value;
		wire [23:0] cross_pixel;
		wire [20:0] xy_cross;

		debounce d_up(1'd0, clk, button_up, buttonup);
		debounce d_down(1'd0, clk, button_down, buttondown);
		debounce d_left(1'd0, clk, button_left, buttonleft);
		debounce d_right(1'd0, clk, button_right, buttonright);

		crosshairs                                                    cross(clk,
buttonup,buttondown,buttonleft,buttonright,hcount,vcount,x_coord,y_coord,x_coord2,
y_coord2,pixel,hsv_data,rgb_value,hsv_value, xy_cross, cross_pixel);

		//final pixel to be outputted
		wire [23:0]	final_pixel;
		wire [23:0] drum_display_pixel;
```

```
			assign final_pixel = switch[1] ? cross_pixel :drum_display_pixel; //cambiar
esto dependiendo q quiero output.

		// VGA Output.  In order to meet the setup and hold times of the
		// AD7125, we send it ~clk.
		assign vga_out_red = final_pixel[23:16]; //esto hay q cambiar
		assign vga_out_green = final_pixel[15:8]; //esto hay q cambiar
		assign vga_out_blue = final_pixel[7:0]; // esto hay q cambiar
		assign vga_out_sync_b = 1'b1;    // not used
		assign vga_out_pixel_clock = ~clk;
		assign vga_out_blank_b = ~b;
		assign vga_out_hsync = hs;
		assign vga_out_vsync = vs;
			////////////////////////////Collision                              Engine
Instantiation/////////////////////////////////////
			wire
cymbal_red_pulse,cymbal_yellow_pulse,snare_red_pulse,snare_yellow_pulse,
				floor_red_pulse,floor_yellow_pulse;
			///////////Detect red Hit//////////////////////////
			collision_engine
detect_red_hit(.clock(clk),.reset(reset),.direction(direction),
				.velocity(velocity),.x_hand(x_coord),.y_hand(y_coord),
				.cymbal_pulse(cymbal_red_pulse),.floor_pulse(floor_red_pulse),
						.snare_pulse(snare_red_pulse));
	//		/////////////Detect yellow hit/////////////////////
			collision_engine
detect_yellow_hit(.clock(clk),.reset(reset),.direction(direction2),
				.velocity(velocity2),.x_hand(x_coord2),.y_hand(y_coord2),

	.cymbal_pulse(cymbal_yellow_pulse),.floor_pulse(floor_yellow_pulse),
					.snare_pulse(snare_yellow_pulse));
			////////////////Volume Control//////////////////////////////////
		wire vup,vdown;
		reg old_vup,old_vdown;
		debounce bup(.reset(reset),.clk(clock_27mhz),.noisy(~button0),.clean(vup));
		debounce
bdown(.reset(reset),.clk(clock_27mhz),.noisy(~button1),.clean(vdown));
		reg [4:0] volume;
			reg [4:0] new_volume;
		always @ (posedge clock_27mhz) begin
		  if (reset) volume <= 5'd8;
		  else begin
			if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
```

```
                if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
            end
            old_vup <= vup;
            old_vdown <= vdown;
          end
                wire ready;
                wire [7:0] from_ac97_data;
                reg[9:0] to_ac97_data;
                lab5audio    a(clock_27mhz,    reset,    new_volume,    from_ac97_data,
to_ac97_data, ready,
                    audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                    ac97_synch, ac97_bit_clock);

                wire [7:0] cymbal_audio_out,snare_audio_out,floor_audio_out;

                wire cymbal_pulse,floor_pulse,snare_pulse;

                assign                        red_hand_hit                        =
cymbal_red_pulse|floor_red_pulse|snare_red_pulse;
                assign                       yellow_hand_hit                      =
cymbal_yellow_pulse|floor_yellow_pulse|snare_yellow_pulse;

                assign
cymbal_pulse=cymbal_red_pulse|cymbal_yellow_pulse;//|cymbal_yellow_pulse;
                assign
floor_pulse=floor_red_pulse|floor_yellow_pulse;//|floor_yellow_pulse;
                assign
snare_pulse=snare_red_pulse|snare_yellow_pulse;//|snare_yellow_pulse;
                /////////////////Assigning     Volume     based     on     Velocity
Threshold/////////////
                always @(posedge clk) begin
                        if (switch[4]) begin
                                if (red_hand_hit) begin
                                        if   (velocity>=10&&velocity<20)   new_volume   <=
volume-5'd16;
                                        else if(velocity>=20&&velocity<40) new_volume <=
volume-5'd7;
                                        else if (velocity>=40) new_volume <= volume;
                                end
                                else if (yellow_hand_hit) begin
                                        if  (velocity2>=10&&velocity2<20)  new_volume  <=
volume-5'd16;
                                        else  if(velocity2>=20&&velocity2<40)  new_volume
<= volume-5'd7;
```

```
                                        else if (velocity2>=40) new_volume <= volume;
                                end
                        end
                        else new_volume<=volume;
                end
        parameter cymbal_max_address=70000;
        parameter snare_max_address=7000;
        parameter floor_max_address=25000;
        wire [16:0] cymbal_sound_addr,floor_sound_addr,snare_sound_addr;

        cymbal_sound_rom
cymbal_one(.clka(clock_27mhz),.addra(cymbal_sound_addr),
                                                .douta(cymbal_audio_out));

        floor_sound_rom floor_one(.clka(clock_27mhz),.addra(floor_sound_addr),
                                                .douta(floor_audio_out));

        snare_sound_rom sound_one(.clka(clock_27mhz),.addra(snare_sound_addr),
                                                .douta(snare_audio_out));

        sound_address_fetch
cymbal_address_out(.clock(clock_27mhz),.reset(reset),.ready(ready),

        .sound_hit_pulse(cymbal_pulse),.sound_max_address(cymbal_max_address),
                        .sound_out_address(cymbal_sound_addr));

        sound_address_fetch
floor_address_out(.clock(clock_27mhz),.reset(reset),.ready(ready),

        .sound_hit_pulse(floor_pulse),.sound_max_address(floor_max_address),
                        .sound_out_address(floor_sound_addr));

        sound_address_fetch
snare_address_out(.clock(clock_27mhz),.reset(reset),.ready(ready),

        .sound_hit_pulse(snare_pulse),.sound_max_address(snare_max_address),
                        .sound_out_address(snare_sound_addr));



        //////////////////////////
        //   FLASH PART STARTS HERE
        //////////////////////////
```

```
//   assign analyzer3_clock = ready;
//   assign analyzer3_data = {from_ac97_data, to_ac97_data};

          //initialize the usb receiver
          wire [7:0] data_in = user1[7:0];
          wire [7:0] data_out;

          wire flash_busy;

          wire newout;
          wire hold = 1'b0;
          wire [3:0] state;

          wire reset_full;

          debounce
bzero(.reset(reset),.clk(clock_27mhz),.noisy(button3),.clean(reset_full));



          usb_input usb(clock_27mhz,reset,data_in,user1[8],user1[9],data_out,
                              newout,flash_busy,state);  //tenia  hold  en
vez de flash busy


          //initialize the flash manager

          wire memory_busy;
          wire memory_mode;
          reg doread;
          wire [639:0] dots;

          //calculate the number of samples im inputting

          reg [22:0] amount_of_frames = 23'd0;
          reg prev_newout;
          always @ (posedge clock_27mhz) prev_newout <= newout;

          always @ (posedge clock_27mhz) begin
               //we are in write mode

               if (~reset_full) begin
```

```
                                amount_of_frames <= 0;
                        end

                        else if (memory_mode) begin
                                //pulse for new frame
                                if (newout && ~prev_newout) begin
                                        amount_of_frames <= amount_of_frames + 1;
                                end
                        end

                end


                //initialize the flash manager



                debounce  db_mm(0,clock_27mhz,switch[5],memory_mode);  //era  7  en
original
                reg [22:0] read_addr;
                wire [15:0] write_data_flash;
                wire [15:0] read_data;
                wire [11:0] fsm_state;

                assign write_data_flash = {8'd0,data_out};

                flash_manager                                   manager(clock_27mhz,
~reset_full,dots,memory_mode,write_data_flash,

        newout,read_addr,read_data,doread,flash_busy,flash_data,

        flash_address,flash_ce_b,flash_oe_b,flash_we_b,flash_reset_b,

        flash_sts,flash_byte_b,fsm_state);

                /*
                module  flash_manager(clock,  reset,  dots,  writemode,  wdata,  dowrite,
raddr,  frdata,  doread,  busy,  flash_data,  flash_address,  flash_ce_b,  flash_oe_b,
flash_we_b, flash_reset_b, flash_sts, flash_byte_b, fsmstate);
                input reset, clock;                     //clock and reset
                output [639:0] dots;            //outputs to dot-matrix to help debug flash,
not necessary
                input writemode;                        //if  true  then  we're  in  write  mode,
else we're in read mode
                input [15:0] wdata;                     //data to be written
```

```
			input dowrite;				//putting this high tells the manager
the data it has is new, write it
			input [22:0] raddr;			//address to read from
			output[15:0] frdata;		//data being read
			reg[15:0]    rdata;
			input doread;				//putting this high tells the manager
to perform a read on the current address
			output busy;				//and an output to tell folks we're
still working on the last thing
			reg busy;

			inout [15:0] flash_data;						//direct
passthrough from labkit to low-level modules (flash_int and test_fsm)
		output [23:0] flash_address;
		output flash_ce_b, flash_oe_b, flash_we_b;
		output flash_reset_b, flash_byte_b;
		input  flash_sts;
	*/

			// light up LEDs when recording, show volume during playback.
		// led is active low
			//leds output direct data being received

		//assign		led		=		switch[5]		?
~{5'd0,~reset_full,flash_busy,newout,memory_mode}:	~data_out;//playback	?
~{filter,2'b00, volume} : ~{filter,7'hFF};


			//loop to assign read addr
			always@(posedge clock_27mhz) begin
				//we are on read mode
				if(~memory_mode) begin

					//play back sounds from memory
					if (switch[2]) begin //era 4 en orignal

						if(ready) begin
							if	(read_addr	<	amount_of_frames)
begin//23'd50000) begin
								read_addr <= read_addr + 1;
								doread <= 1;
							end
```

```
                                        else  if  (read_addr  ==  amount_of_frames)
begin//23'd50000) begin
                                             read_addr <= 0;

                                        end
                                   end
                              end

                              //reset addr while in silent mode
                              else begin

                                   read_addr <= 0;
                                   doread <= 0;

                              end

                         end

                    end

                    reg [7:0] audio_out;

                    always @ (posedge clock_27mhz) begin
                         if(ready) begin
                              audio_out <= read_data[7:0];
                         end
                    end

                // record module
                /*recorder r(.clock(clock_27mhz), .reset(reset), .ready(ready),
                    .playback(playback), .filter(filter),

.from_ac97_data(from_ac97_data),.from_flash_data(audio_out),.to_ac97_data(to_ac97
_data));
                    */

               //        assign analyzer1_clock = clock_27mhz;
               //        assign                          analyzer1_data                          =
{memory_mode,user1[9],flash_busy,newout,doread,
               //
               data_out,1'b0,2'd0};

               ////////////////////////
               //FLASH PART ENDS HERE
```

```
        /////////////////////////
        always@(posedge clock_27mhz) begin
                if(ready) begin
                        to_ac97_data<={{2{cymbal_audio_out[7]}},cymbal_audio_out}+

        {{2{floor_audio_out[7]}},floor_audio_out}+

        {{2{snare_audio_out[7]}},snare_audio_out}+

        {{2{audio_out[7]}},audio_out};//+floor_audio_out+snare_audio_out;
                end
        end

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////Game Display Module////////////////////////////
        ////////////////////////////////////////////////////////////////////////
        wire [10:0] xtemp_right_stick,xtemp_left_stick;
        wire [9:0] ytemp_right_stick,ytemp_left_stick;
        game_display drum_visual(.vclock(clk),.reset(reset),
          .x_coord_left(x_coord),
          .x_coord_right(x_coord2),
          .y_coord_left(y_coord),
                .y_coord_right(y_coord2),
                .cymbal_right_hit(cymbal_yellow_pulse),
                .floor_right_hit(floor_yellow_pulse),
                .snare_right_hit(snare_yellow_pulse),
                .cymbal_left_hit(cymbal_red_pulse),
                .floor_left_hit(floor_red_pulse),
                .snare_left_hit(snare_red_pulse),
                .hcount(hcount),
                .vcount(vcount),
          .hsync(hsync),
          .vsync(vsync),
                .blank(blank),
                .xtemp_right_stick(xtemp_right_stick),
                .xtemp_left_stick(xtemp_left_stick),
                .ytemp_right_stick(ytemp_right_stick),
                .ytemp_left_stick(ytemp_left_stick),
         .pixel  (drum_display_pixel)
          );

          // debugging
                assign       led       =       (cymbal_pulse|snare_pulse|floor_pulse)       ?
~{0,7'hFF}:~{flash_busy,2'b00, volume};
```

```
  //assign led = ~{vram_addr[18:13],reset,switch[0]};
        always @(posedge clk) begin

dispdata<=switch[3]?{1'b0,xtemp_left_stick,2'd0,ytemp_left_stick,

16'd0,1'b0,xtemp_right_stick,2'd0,ytemp_right_stick}

:{2'd0,velocity,1'd0,x_coord,2'd0,y_coord,16'd0,3'd0,cymbal_pulse,3'd0,floor_pu
lse,3'd0,snare_pulse};//{ntsc_data,9'b0,ntsc_addr}; este estaba antes
        end

//   always @(posedge clk) begin
//              if (drum_control_pulse==1) begin
//                      // dispdata <= {vram_read_data,9'b0,vram_addr};
//                      dispdata
<=switch[3]?{1'b0,xy_cross[20:10],2'd0,xy_cross[9:0],16'd0,hsv_value}
//
:{2'd0,velocity,1'd0,x_coord,2'd0,y_coord,24'd0,2'd0,drum_select_signal};//{ntsc
_data,9'b0,ntsc_addr}; este estaba antes
//                      //dispdata         <=          switch[3]          ?
{1'b0,xy_cross[20:10],2'd0,xy_cross[9:0],16'd0,hsv_value}                  :
{2'd0,velocity,3'd0,x_accumulator,hsv_value};//{ntsc_data,9'b0,ntsc_addr}; este estaba
antes
//              end
//      end


endmodule




///////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (800 x 600 @ 60Hz)
///////////////////////////////////////////////////////////////////////////////
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output        vsync;
  output        hsync;
  output        blank;

  reg    hsync,vsync,hblank,vblank,blank;
  reg [10:0]      hcount;    // pixel number on current line
  reg [9:0] vcount;        // line number

  // horizontal: 1056 pixels total
  // display 800 pixels per line
  wire      hsyncon,hsyncoff,hreset,hblankon;
  assign    hblankon = (hcount == 799);
  assign    hsyncon = (hcount == 839);
  assign    hsyncoff = (hcount == 967);
  assign    hreset = (hcount == 1055);

  // vertical: 628 lines total
  // display 600 lines
  wire      vsyncon,vsyncoff,vreset,vblankon;
  assign    vblankon = hreset & (vcount == 599);
  assign    vsyncon = hreset & (vcount == 600);
  assign    vsyncoff = hreset & (vcount == 604);
  assign    vreset = hreset & (vcount == 627);

  // sync and blanking
  wire      next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

    blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule

////////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
```

```
//
// Bug due to memory management will be fixed. The bug happens because
// memory is called based on current hcount & vcount, which will actually
// shows up 2 cycle in the future. Not to mention that these incoming data
// are latched for 2 cycles before they are used. Also remember that the
// ntsc2zbt's addressing protocol has been fixed.

// The original bug:
// -. At (hcount, vcount) = (100, 201) data at memory address(0,100,49)
//    arrives at vram_read_data, latch it to vr_data_latched.
// -. At (hcount, vcount) = (100, 203) data at memory address(0,100,49)
//    is latched to last_vr_data to be used for display.
// -. Remember that memory address(0,100,49) contains camera data
//    pixel(100,192) - pixel(100,195).
// -. At (hcount, vcount) = (100, 204) camera pixel data(100,192) is shown.
// -. At (hcount, vcount) = (100, 205) camera pixel data(100,193) is shown.
// -. At (hcount, vcount) = (100, 206) camera pixel data(100,194) is shown.
// -. At (hcount, vcount) = (100, 207) camera pixel data(100,195) is shown.
//
// Unfortunately this means that at (hcount == 0) to (hcount == 11) data from
// the right side of the camera is shown instead (including possible sync signals).

// To fix this, two corrections has been made:
// -. Fix addressing protocol in ntsc_to_zbt module.
// -. Forecast hcount & vcount 8 clock cycles ahead and use that
//    instead to call data from ZBT.


module vram_display(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr,vram_read_data);

  input reset, clk;
  input [10:0] hcount;
  input [9:0]    vcount;
  output [23:0] vr_pixel;
  output [18:0] vram_addr;
  input [35:0]  vram_read_data;

  //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
  wire [10:0] hcount_f = (hcount >= 1048) ? (hcount - 1048) : (hcount + 8);
  wire [9:0] vcount_f = (hcount >= 1048) ? ((vcount == 805) ? 0 : vcount + 1) :
vcount;
```

```
        wire [18:0]    vram_addr = {vcount_f, hcount_f[9:1]}; // cambiar a [9:1] y quitar
padding bit

        wire hc4 = hcount[0]; // era hcount
        reg [23:0]      vr_pixel;
        reg [35:0]      vr_data_latched;
        reg [35:0]      last_vr_data;

        always @(posedge clk)
          last_vr_data <= (hc4 == 1'd1) ? vr_data_latched : last_vr_data;

        always @(posedge clk)
          vr_data_latched <= (hc4 == 1'd0) ? vram_read_data : vr_data_latched;

        always @(*)             // each 36-bit word from RAM is decoded to 4 bytes
          case (hc4) //solo deberian de ser 2 cases para color
            //2'd3: vr_pixel = last_vr_data[7:0];
            //2'd2: vr_pixel = last_vr_data[7+8:0+8];
            1'd1:                              vr_pixel                              =
{last_vr_data[17:12],2'd0,last_vr_data[11:6],2'd0,last_vr_data[5:0],2'd0};
            1'd0:                              vr_pixel                              =
{last_vr_data[35:30],2'd0,last_vr_data[29:24],2'd0,last_vr_data[23:18],2'd0};
          endcase

      endmodule // vram_display


      ////////////////////////////////////////////////////////////////////////////////
      // parameterized delay line CON TAMANO
      ////////////////////////////////////////////////////////////////////////////////

      module delayNsize #(parameter NDELAY = 3, parameter SIZE =10)(clk,in,out);
        input clk;
        input [SIZE-1:0] in;
        output [SIZE-1:0] out;


              reg [SIZE-1:0] shift_regs [NDELAY:0];
              reg [4:0] i;

              always @ (posedge clk) begin
                    shift_regs[0] <= in;
                    for(i=1;i<NDELAY+1; i= i+1) begin
                          shift_regs[i] <= shift_regs[i-1];
            end
        end

        assign out = shift_regs[NDELAY];

endmodule // delayN




///////////////////////////////////////////////////////////////////////////
// parameterized delay line
///////////////////////////////////////////////////////////////////////////
module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;

   reg [NDELAY-1:0] shiftreg;
   wire     out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN

//////////////////////////////////////////////////////////////////////////
// ramclock module

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
///////////////////////////////////////////////////////////////////////////////

module ramclock(ref_clock, fpga_clock, ram0_clock, ram1_clock,
                clock_feedback_in, clock_feedback_out, locked);

  input ref_clock;                  // Reference clock input
  output fpga_clock;                // Output clock to drive FPGA logic
  output ram0_clock, ram1_clock;   // Output clocks for each RAM chip
  input  clock_feedback_in;         // Output to feedback trace
  output clock_feedback_out;        // Input from feedback trace
  output locked;                    // Indicates that clock outputs are stable

  wire  ref_clk, fpga_clk, ram_clk, fb_clk, lock1, lock2, dcm_reset;

  ////////////////////////////////////////////////////////////////////////////

  //To force ISE to compile the ramclock, this line has to be removed.
  //IBUFG ref_buf (.O(ref_clk), .I(ref_clock));

        assign ref_clk = ref_clock;

  BUFG int_buf (.O(fpga_clock), .I(fpga_clk));

  DCM int_dcm (.CLKFB(fpga_clock),
               .CLKIN(ref_clk),
               .RST(dcm_reset),
               .CLK0(fpga_clk),
               .LOCKED(lock1));
```

```
    // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"
    // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"
    // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"
    // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"
    // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"
    // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"
    // synthesis attribute PHASE_SHIFT of int_dcm is 0

    BUFG ext_buf (.O(ram_clock), .I(ram_clk));

    IBUFG fb_buf (.O(fb_clk), .I(clock_feedback_in));

    DCM ext_dcm (.CLKFB(fb_clk),
                 .CLKIN(ref_clk),
                 .RST(dcm_reset),
                 .CLK0(ram_clk),
                 .LOCKED(lock2));
    // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"
    // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"
    // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"
    // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"
    // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"
    // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"
    // synthesis attribute PHASE_SHIFT of ext_dcm is 0

    SRL16 dcm_rst_sr (.D(1'b0), .CLK(ref_clk), .Q(dcm_reset),
                      .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
    // synthesis attribute init of dcm_rst_sr is "000F";


    OFDDRRSE ddr_reg0 (.Q(ram0_clock), .C0(ram_clock), .C1(~ram_clock),
                       .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
    OFDDRRSE ddr_reg1 (.Q(ram1_clock), .C0(ram_clock), .C1(~ram_clock),
                       .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
    OFDDRRSE     ddr_reg2     (.Q(clock_feedback_out),     .C0(ram_clock),
.C1(~ram_clock),
                       .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));

    assign locked = lock1 && lock2;

   endmodule
```

Gesture Detection Module:

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09  // gph mod 11/3/2011
//
//
// Bug due to memory management will be fixed. It happens because
// the memory addressing protocol is off between ntsc2zbt.v and
// vram_display.v. There are 2 solutions:
// -. Fix the memory addressing in this module (neat addressing protocol)
//    and do memory forecast in vram_display module.
// -. Do nothing in this module and do memory forecast in vram_display
//    module (different forecast count) while cutting off reading from
//    address(0,0,0).
//
// Bug in this module causes 4 pixel on the rightmost side of the camera
// to be stored in the address that belongs to the leftmost side of the
// screen.
//
// In this example, the second method is used. NOTICE will be provided
// on the crucial source of the bug.
//
/////////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

   input            clk;     // system clock
   input           vclk;   // video clock from camera
   input [2:0]     fvh;
   input           dv;
   input [29:0] din;//[7:0]        din; //esto hay q cambiar
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output          ntsc_we;        // write enable for NTSC data
   input           sw;             // switch which determines mode (for debugging)

   parameter       COL_START = 10'd30; //era 30
   parameter       ROW_START = 10'd30; //era 30

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 * 788 XGA display

   reg [9:0]       col = 0;
   reg [9:0]       row = 0;
   reg [29:0]      vdata = 0; // esto hay q cambiar
   reg             vwe;
   reg             old_dv;
   reg             old_frame;      // frames are even / odd interlaced
   reg             even_odd;       // decode interlaced frame to this wire

   wire   frame = fvh[2];
   wire   frame_edge = frame & ~old_frame;

   always @ (posedge vclk) //LLC1 is reference
    begin
           old_dv <= dv;
           vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
           old_frame <= frame;
           even_odd = frame_edge ? ~even_odd : even_odd;

           if (!fvh[2])
             begin
               col <= fvh[0] ? COL_START :
                           (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
               row <= fvh[1] ? ROW_START :
                           (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
               vdata <= (dv && !fvh[2]) ? din : vdata;
             end
    end

   // synchronize with system clock
```

```
            reg [9:0] x[1:0],y[1:0];
            reg [29:0] data[1:0]; //esto hay q cambiar
            reg      we[1:0];
            reg        eo[1:0];

            always @(posedge clk)
              begin
                    {x[1],x[0]} <= {x[0],col};
                    {y[1],y[0]} <= {y[0],row};
                    {data[1],data[0]} <= {data[0],vdata};
                    {we[1],we[0]} <= {we[0],vwe};
                    {eo[1],eo[0]} <= {eo[0],even_odd};
              end

            // edge detection on write enable signal

            reg old_we;
            wire we_edge = we[1] & ~old_we;
            always @(posedge clk) old_we <= we[1];

            // shift each set of four bytes into a large register for the ZBT
                              // here we implement the rgb conversion     wire                    [7:0]
RGB_data[2:0];           YCrCb2RGB
convert1(RGB_data[2],RGB_data[1],RGB_data[0],clk,0,data[1][29:20],data[1][19:10],
          data[1][9:0]);
            reg [35:0] mydata;
            always @(posedge clk)
              if (we_edge)
                mydata     <=     {     mydata[17:0],     RGB_data[2][7:2],     RGB_data[1][7:2],
RGB_data[0][7:2] };

            // NOTICE : Here we have put 4 pixel delay on mydata. For example, when:
            // (x[1], y[1]) = (60, 80) and eo[1] = 0, then:
            // mydata[31:0] = ( pixel(56,160), pixel(57,160), pixel(58,160), pixel(59,160) )
            // This is the root of the original addressing bug.


            // NOTICE : Notice that we have decided to store mydata, which
            //        contains pixel(56,160) to pixel(59,160) in address
            //        (0, 160 (10 bits), 60 >> 2 = 15 (8 bits)).
            //
            //        This protocol is dangerous, because it means
            //        pixel(0,0) to pixel(3,0) is NOT stored in address
```

```
	//	(0, 0 (10 bits), 0 (8 bits)) but is rather stored
	//	in address (0, 0 (10 bits), 4 >> 2 = 1 (8 bits)). This
	//	calculation ignores COL_START & ROW_START.
	//
	//	4 pixels from the right side of the camera input will
	//	be stored in address corresponding to x = 0.
	//
	//	To fix, delay col & row by 4 clock cycles.
	//	Delay other signals as well.

	reg [39:0] x_delay;
	reg [39:0] y_delay;
	reg [3:0] we_delay;
	reg [3:0] eo_delay;

	always @ (posedge clk)
	begin
	  x_delay <= {x_delay[29:0], x[1]};
	  y_delay <= {y_delay[29:0], y[1]};
	  we_delay <= {we_delay[2:0], we[1]};
	  eo_delay <= {eo_delay[2:0], eo[1]};
	end

	// compute address to store data in
	wire [8:0] y_addr = y_delay[38:30];
	    wire [9:0] x_addr = x_delay[39:30];

	wire [18:0] myaddr = {y_addr[8:0], eo_delay[3], x_addr[9:1]};

	// Now address (0,0,0) contains pixel data(0,0) etc.


	// alternate (256x192) image data and address
	wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
	wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};

	// update the output address and data only when four bytes ready

	reg [18:0] ntsc_addr;
	reg [35:0] ntsc_data;
	wire	ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0)); // esto
hayq  cambiarlo para q writes every 2

	always @(posedge clk)
```

```
          if ( ntsc_we )
            begin
                    ntsc_addr <= sw ? myaddr2 : myaddr;         //    normal    and    expanded
modes
                    ntsc_data <= sw ? {4'b0,mydata2} : mydata; //esto hay q cambiarlo
            end

     endmodule // ntsc_to_zbt



     `timescale 1ns / 1ps
     ////////////////////////////////////////////////////////////////////////////////
     // Company:
     // Engineer: Luis Fernandez
     //
     // Create Date:    20:19:34 11/04/2012
     // Design Name:
     // Module Name:    hand_detection
     // Project Name:
     // Target Devices:
     // Tool versions:
     // Description:
     //
     // Dependencies:
     //
     // Revision:
     // Revision 0.01 - File Created
     // Additional Comments:
     //
     ////////////////////////////////////////////////////////////////////////////////
     module hand_detection(
         input clk,
                 input hand, // decides que color estoy buscando
         input [23:0] hsv,
                 input [10:0] h_count,
                 input [9:0] v_count,
                 input [23:0] pixel,
                 output reg interesting,
                 output reg [23:0] colored_pixel,
                 output [10:0] x_coord,
         output [9:0] y_coord,
                 output reg down,
                 output reg [9:0] velocity,
                 output reg [24:0] x_accumulator,
                 output reg [24:0] y_accumulator
         );

     //red parameters
     parameter RED_HUE_MAX = 0;//10;
     parameter RED_HUE_MIN = 210;//era 240
     parameter RED_SAT_MIN = 160;//150;//180;
     parameter RED_VAL_MIN = 100;
     //yellow parameters
     parameter YELLOW_HUE_MAX = 48;//80;
     parameter YELLOW_HUE_MIN = 32;//0;//42;
     parameter YELLOW_SAT_MIN = 80;//130;//120;//90;//180;//90; //era 100
     parameter YELLOW_VAL_MIN = 100;
     //frame limits
     parameter [9:0] F_UP = 10'd108;
     parameter [9:0] F_DOWN = 10'd542;
     parameter [10:0] F_LEFT = 11'd55;
     parameter [10:0] F_RIGHT = 11'd717;

     //decide which hand we are detecting
     wire [7:0] HUE_MAX = hand ? RED_HUE_MAX : YELLOW_HUE_MAX;
     wire [7:0] HUE_MIN = hand ? RED_HUE_MIN : YELLOW_HUE_MIN;
     wire [7:0] SAT_MIN = hand ? RED_SAT_MIN : YELLOW_SAT_MIN;
     wire [7:0] VAL_MIN = hand ? RED_VAL_MIN : YELLOW_VAL_MIN;

     reg [24:0] count_x = 0;
     reg [24:0] count_y = 0;
     reg [24:0] dividend_x = 0;
     reg [24:0] dividend_y = 0;
     reg [24:0] divisor_x = 0;
     reg [24:0] divisor_y = 0;
     wire [24:0] x_quotient;
     wire [24:0] y_quotient;
     wire [24:0] x_remainder;
     wire [24:0] y_remainder;
     wire x_rfd;
     wire y_rfd;

     //para calcular center of mass
     reg [9:0] y_memory;

     //wire direction;
     //wire [9:0] velocity;
```

```
	//instantiate divisors
			mass_divider x_div(
			.clk(clk),
			.dividend(dividend_x),
			.divisor(divisor_x),
			.quotient(x_quotient),
			.fractional(x_remainder),
			.rfd(x_rfd)
			);

			mass_divider y_div(
			.clk(clk),
			.dividend(dividend_y),
			.divisor(divisor_y),
			.quotient(y_quotient),
			.fractional(y_remainder),
			.rfd(y_rfd)
			);


	always @ (posedge clk) begin

		//colored_pixel <= pixel;

		// reset everything in new frame
		if (h_count == 11'd0 && v_count == 10'd0) begin
			x_accumulator <= 0;
			y_accumulator <= 0;
			count_x <= 0;
			count_y <= 0;

			colored_pixel <= pixel;

		end

		//detect colors if in frame
		else if (h_count >= F_LEFT && h_count <= F_RIGHT && v_count >= F_UP
&& v_count <= F_DOWN) begin

			//detect red
			if(hand && (hsv[23:16] > HUE_MIN || hsv[23:16] < HUE_MAX) &&
hsv[15:8] > SAT_MIN && hsv[7:0] > VAL_MIN) begin
```

```
                interesting <= 1'd1;
                //x_coord <= h_count;
                //y_coord <= v_count;

                colored_pixel[23:16] <= 8'd255;
                colored_pixel[15:0] <= 16'd0;

                x_accumulator <= x_accumulator + h_count;
                y_accumulator <= y_accumulator + v_count;
                count_x <= count_x + 1;
                count_y <= count_y + 1;


        end

        //detect yellow
        else if(~hand && hsv[23:16] >= HUE_MIN && hsv[23:16] <
HUE_MAX && hsv[15:8] > SAT_MIN && hsv[7:0] > VAL_MIN) begin

                interesting <= 1'd1;
                //x_coord <= h_count;
                //y_coord <= v_count;

                colored_pixel[23:16] <= 8'd255;
                colored_pixel[15:8] <= 8'd255;
                colored_pixel[7:0] <= 8'd0;

                x_accumulator <= x_accumulator + h_count;
                y_accumulator <= y_accumulator + v_count;
                count_x <= count_x + 1;
                count_y <= count_y + 1;

        end

        else begin
                interesting <=1'd0;
                //x_coord <= 0;
                //y_coord <= 0;
                colored_pixel <= pixel;

                x_accumulator <= x_accumulator;
                y_accumulator <= y_accumulator;
                count_x <= count_x;
                count_y <= count_y;
                divisor_x <= divisor_x;
                divisor_y <= divisor_y;
                dividend_x <= dividend_x;
                dividend_y <= dividend_y;

        end

    end

    //start divisors
    else if (h_count == 11'd0 && v_count == 10'd543) begin

        colored_pixel <= pixel;

        dividend_x <= (count_x > 50) ? x_accumulator : 0;
        dividend_y <= (count_y > 50) ? y_accumulator : 0;


        divisor_x <= (count_x == 0) ? 1 : count_x;
        divisor_y <= (count_y == 0) ? 1 : count_y;

    end


    //catch it all clause
    else begin
        interesting <= 0;
        colored_pixel <= pixel;
        /*
        x_accumulator <= x_accumulator;
        y_accumulator <= y_accumulator;
        count_x <= count_x;
        count_y <= count_y;

        divisor_x <= divisor_x;
        divisor_y <= divisor_y;
        dividend_x <= dividend_x;
        dividend_y <= dividend_y;
        */
    end

end
```

```
	//always @ (posedge clk) y_memory <= y_coord;

	wire [10:0] x_coord_noisy;
	wire [9:0] y_coord_noisy;

	assign x_coord_noisy = x_rfd ? x_quotient[10:0] : x_coord_noisy;
	assign y_coord_noisy = y_rfd ? y_quotient[9:0] : y_coord_noisy; //era x

	wire new_value;

	assign new_value = (h_count == 11'd1 && v_count == 10'd542);

	// instantiate here, input is y_coord_noisy, output is y_coord

	avg_center
average(clk,new_value,x_coord_noisy,y_coord_noisy,x_coord,y_coord);



	always @ (posedge clk) begin

		if (h_count == 11'd0 && v_count == 10'd542) begin//(y_rfd &&
~y_rfd_delay) begin


			//y_coord <= y_quotient[9:0];
			//x_coord <= x_quotient[10:0];

			y_memory <= y_coord; //este lo cambie el loop superior

			// 0 for up/stationary
			// 1 for down
			if (y_memory > y_coord + 10'd5) begin //moving up
				down <= 1'b0; //up
				velocity <=y_memory[9:0] - y_quotient[9:0];
			end
			else if (y_memory + 10'd5 < y_coord) begin //moving down
				down <= 1'b1; //down
				velocity <=y_coord - y_memory[9:0];
			end
			else begin //staying 'stationary'
				down <= 1'b0;
				velocity <= 10'd0;
```

```
                    end

            end

    end


    endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:16:21 11/08/2012
// Design Name:
// Module Name:    crosshairs
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module crosshairs(
   input clk,
         input up,
         input down,
         input left,
         input right,
   input [10:0] hcount,
         input [9:0] vcount,
         input [10:0] x_coord,
         input [9:0] y_coord,
         input [10:0] x_coord2,
         input [9:0] y_coord2,
   input [23:0] pixel,
         input [23:0] hsv,
            output reg [23:0] rgb_value,
            output reg [23:0] hsv_value,
            output [20:0] xycross,
            output reg [23:0] final_pixel
    );

reg [10:0] x_loc = 11'd300;
reg [9:0] y_loc = 10'd300;

always @ (posedge clk) begin

            final_pixel <= pixel;
            if (vcount == 10'd0 && hcount == 11'd0) begin
                        if (~up) begin
                                    y_loc <= y_loc - 1;
                        end
                        else if (~down) begin
                                    y_loc <= y_loc + 1;
                        end
                        if (~left) begin
                                    x_loc <= x_loc -1;
                        end
                        else if (~right) begin
                                    x_loc <= x_loc + 1;
                        end
            end

            else begin
                        y_loc <= y_loc;
                        x_loc <= x_loc;
            end

            if (x_loc == hcount) begin
                        final_pixel[23:16] <= 8'd255;
                        final_pixel[15:0] <= 16'd0;
            end
                        if (y_loc == vcount) begin
                        final_pixel[23:16] <= 8'd255;
                        final_pixel[15:0] <= 16'd0;
            end

            if (x_loc == hcount && y_loc == vcount) begin
                        rgb_value <= pixel;
                        hsv_value <= hsv;
                        final_pixel <= pixel;
            end


            //to display crosshairs to center of mass of 1 hand
            if (hcount == x_coord) begin

                        final_pixel[23:16] <= 8'd0;
                        final_pixel[15:8] <= 8'd255;
                        final_pixel[7:0] <= 8'd0;

            end
            if (vcount == y_coord) begin

                        final_pixel[23:16] <= 8'd0;
                        final_pixel[15:8] <= 8'd255;
                        final_pixel[7:0] <= 8'd0;

            end
            if (hcount == x_coord && vcount == y_coord) begin

                        final_pixel <= pixel;

            end

            //to display crosshairs to center of mass of second hand
            if (hcount == x_coord2) begin

                        final_pixel[23:16] <= 8'd0;
                        final_pixel[15:8] <= 8'd0;
                        final_pixel[7:0] <= 8'd255;

            end
            if (vcount == y_coord2) begin

                        final_pixel[23:16] <= 8'd0;
                        final_pixel[15:8] <= 8'd0;
                        final_pixel[7:0] <= 8'd255;

            end
            if (hcount == x_coord2 && vcount == y_coord2) begin

                        final_pixel <= pixel;
```

```
        end

        //to display trigger
        if (vcount == 10'd420) begin
                final_pixel[23:16] <= 8'd255;
                final_pixel[15:8] <= 8'd255;
                final_pixel[7:0] <= 8'd0;

        end

        //to display left boundary
        if (hcount == 11'd276) begin
                final_pixel[23:16] <= 8'd255;
                final_pixel[15:8] <= 8'd255;
                final_pixel[7:0] <= 8'd0;

        end

        //to display right boundary
        if (hcount == 11'd496) begin
                final_pixel[23:16] <= 8'd255;
                final_pixel[15:8] <= 8'd255;
                final_pixel[7:0] <= 8'd0;

        end

end


assign xycross = {x_loc, y_loc};

//assign final_pixel_2 = final_pixel;
//assign rgb_value_2 = rgb_value;

endmodule



`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:01:07 11/26/2012
// Design Name:
// Module Name:    avg_center
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module avg_center(
   input clk,
          input new_value,
   input [10:0] x_coord_noisy,
   input [9:0] y_coord_noisy,
   output [10:0] x_coord,
   output [9:0] y_coord
   );

wire [12:0] x_accumulator;
wire [11:0] y_accumulator;
reg [10:0] buffer_x [3:0];
reg [9:0] buffer_y [3:0];
reg [1:0] offset;

always @ (posedge clk) begin

          if (new_value) begin
                    offset <= offset + 1;
                    buffer_x[offset] <= x_coord_noisy;
                    buffer_y[offset] <= y_coord_noisy;

          end

end

assign x_accumulator = buffer_x[0] + buffer_x[1] + buffer_x[2] + buffer_x[3];
assign y_accumulator = buffer_y[0] + buffer_y[1] + buffer_y[2] + buffer_y[3];

//division by 4
```

```
assign x_coord = x_accumulator >> 2;
assign y_coord = y_accumulator >> 2;

endmodule
```

Display Module

```
///////////////////////////////////////////////////////////////////////////////
//
//Display Module
//
///////////////////////////////////////////////////////////////////////////////

module game_display (
  input vclock, // 40MHz clock
  input reset,              // 1 to initialize module
  input [10:0] x_coord_left,              // 1 when stick should move up
  input [10:0] x_coord_right,             // 1 when stick should move down
  input [9:0] y_coord_left,  // 1when stick should move Left
        input [9:0] y_coord_right,      // 1when stick should move right
        input cymbal_right_hit,
        input floor_right_hit,
        input snare_right_hit,
        input cymbal_left_hit,
        input floor_left_hit,
        input snare_left_hit,
  input [10:0] hcount, // horizontal index of current pixel (0..1023)
  input [9:0]    vcount, // vertical index of current pixel (0..767)
  input hsync,              // XVGA horizontal sync signal (active low)
  input vsync,              // XVGA vertical sync signal (active low)
  input blank,              // XVGA blanking (1 means output black pixel)
        output reg [10:0] xtemp_right_stick,
        output reg [10:0] xtemp_left_stick,
        output reg [9:0] ytemp_right_stick,
        output reg [9:0] ytemp_left_stick,
  output [23:0] pixel   // pong game's pixel  // r=23:16, g=15:8, b=7:0
  );

        wire [23:0] pixel_drum,pixel_left_stick,pixel_right_stick,pixel_hit,
                pixel_nohit,pixel_floortom,pixel_cymbal,pixel_heading;

  // REPLACE ME! The code below just generates a color checkerboard
		// using 64 pixel by 64 pixel squares.

			parameter drum_first_horizontal=35;
			parameter drum_first_vertical=340;
			//reg		[10:0]		xtemp_right_stick,xtemp_left_stick,xhit_left-
position,xhit_right_position;
			//reg								[9:0]
ytemp_right_stick,ytemp_left_stick,yhit_left_position,yhit_right_position;

			wire displayhit_left_cymbal,displayhit_left_floor,displayhit_left_snare;
			wire
displayhit_right_cymbal,displayhit_right_floor,displayhit_right_snare;

			always @(posedge vsync) begin
				if ((y_coord_left>108)&&(y_coord_left<395)) begin
					ytemp_left_stick<=y_coord_left;
				end
				if ((y_coord_right>108)&&y_coord_right<395) begin
					ytemp_right_stick<=y_coord_right;
				end
				if (((785-x_coord_left)<660)&&((785-x_coord_left)>0)) begin
					xtemp_left_stick<=displayhit_left_snare?16+140:
						displayhit_left_floor?216+140:
						displayhit_left_cymbal?452+140:(785-
x_coord_left);
				end
				if (((785-x_coord_right)<660)&&((785-x_coord_right)>0)) begin
					xtemp_right_stick<=displayhit_right_snare?152:
						displayhit_right_floor?376:
						displayhit_right_cymbal?600:(785-x_coord_right);
				end
			end
	//		display_hit_detect lefthand_detect_hit(.reset(reset),.vsync(vsync),
	//			.cymbal_hit(cymbal_left_hit),.floor_hit(floor_left_hit),
	//			.snare_hit(snare_left_hit),
	//			.display_cymbal_hit(displayhit_left_cymbal),
	//			.display_snare_hit(displayhit_left_snare),
	//			.display_floor_hit(displayhit_left_floor));
	//
	//			display_hit_detect
righthand_detect_hit(.reset(reset),.vsync(vsync),
	//			.cymbal_hit(cymbal_right_hit),.floor_hit(floor_right_hit),
	//			.snare_hit(snare_right_hit),
	//			.display_cymbal_hit(displayhit_right_cymbal),
	//		.display_snare_hit(displayhit_right_snare),
	//		.display_floor_hit(displayhit_right_floor));

		reg [24:0] new_pixel;
		always @ (posedge vclock) begin
			if(|pixel_drum &&(|pixel_left_stick)) new_pixel<=pixel_left_stick;
			else		if(|pixel_cymbal		&&(|pixel_left_stick))
new_pixel<=pixel_left_stick;
			else		if(|pixel_floortom		&&(|pixel_left_stick))
new_pixel<=pixel_left_stick;
			else		if(|pixel_drum		&&(|pixel_right_stick))
new_pixel<=pixel_right_stick;
			else		if(|pixel_cymbal		&&(|pixel_right_stick))
new_pixel<=pixel_right_stick;
			else		if(|pixel_floortom		&&(|pixel_right_stick))
new_pixel<=pixel_right_stick;
			else
new_pixel<=pixel_right_stick|pixel_drum|pixel_floortom|pixel_cymbal|pixel_left_stick;
		end
	 wire display_left_hit,display_right_hit;

	//								assign
display_left_hit=displayhit_left_snare|displayhit_left_floor|displayhit_left_cymbal;
	//								assign
display_right_hit=displayhit_right_snare|displayhit_right_floor|displayhit_right_cymbal
;
	//
		picture_blob drum_first_pic(.pixel_clk(vclock),
			.x(drum_first_horizontal),.y(drum_first_vertical),
			.hcount(hcount),.vcount(vcount),.pixel(pixel_drum));
	//	heading_image heading_first_pic(.pixel_clk(vclock),
	//		.x(270),.y(2),
	//		.hcount(hcount),.vcount(vcount),.pixel(pixel_heading));
		floor_tom_pic floor_pic(.pixel_clk(vclock),
			.x(270),.y(375),
			.hcount(hcount),.vcount(vcount),.pixel(pixel_floortom));
		pic_cymbal cymbal_first_pic(.pixel_clk(vclock),
			.x(520),.y(350),
			.hcount(hcount),.vcount(vcount),.pixel(pixel_cymbal));

		wire [10:0] x_adjusted;
		assign x_adjusted = xtemp_left_stick - 140;
		pic_right_stick stick_right_pic(.pixel_clk(vclock),
			.x(xtemp_right_stick),.y(ytemp_right_stick),.hit(0),
                .hcount(hcount),.vcount(vcount),.pixel(pixel_right_stick));
        pic_left_stick stick_left_pic(.pixel_clk(vclock),
                .x(x_adjusted),.y(ytemp_left_stick),.hit(0),
                .hcount(hcount),.vcount(vcount),.pixel(pixel_left_stick));


        assign pixel=new_pixel;



endmodule

//////////////////////////////////////////////////

 ////////picture_blob: Heading

////////////////////////////////////////////////
module heading_image
        #(parameter WIDTH = 195,
// default picture width
                                HEIGHT =17)
        // default picture height
        (input pixel_clk,
         input [10:0] x,hcount,
         input [9:0] y,vcount,
         output reg [23:0] pixel);

        wire [15:0] image_addr;
                                    // num of bits for 128*256 ROM
        wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
  reg [7:0] image_bits_reg;

        // calculate rom address and read the location
        assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
        heading_image_rom
rom1(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

        // use color map to create 8bits R, 8bits G, 8 bits B;
        red_heading_map                                            rcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
        green_heading_map                                          gcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
        blue_heading_map                                           bcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));
```

```
                // note the one clock cycle delay in pixel!
                always @ (posedge pixel_clk) begin
                  image_bits_reg <= image_bits;      //This stabilizes the address before
feeding it into Color maps
                        if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
                                        (vcount >= y && vcount < (y+HEIGHT)))
                                        pixel     <=     {red_mapped,     green_mapped,
blue_mapped};

                        else pixel <= 24'h000000;
                end


        endmodule



        ///////////////////////////////////////////////

         ////////picture_blob: Display Snare drum

        ///////////////////////////////////////////////
        module picture_blob
                #(parameter WIDTH = 225,
        // default picture width
                                                HEIGHT =180)
                // default picture height
                (input pixel_clk,
                 input [10:0] x,hcount,
                 input [9:0] y,vcount,
                 output reg [23:0] pixel);

                wire [15:0] image_addr;
                                                        // num of bits for 128*256 ROM
                wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
          reg [7:0] image_bits_reg;

                // calculate rom address and read the location
                assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
                moscow_image_rom
rom1(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

                // use color map to create 8bits R, 8bits G, 8 bits B;
                red_color_map                                                      rcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
                green_color_map                                                    gcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
                blue_color_map                                                     bcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));

                // note the one clock cycle delay in pixel!
                always @ (posedge pixel_clk) begin
                  image_bits_reg <= image_bits;      //This stabilizes the address before
feeding it into Color maps
                        if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
                                        (vcount >= y && vcount < (y+HEIGHT)))
                                        pixel       <=       {red_mapped,       green_mapped,
blue_mapped};

                        else pixel <= 24'h000000;
                end


        endmodule


        ////////////////////////////////////////////////////
        //
        // picture_blob: Display Stick_right
        //
        ////////////////////////////////////////////////////

        module pic_right_stick
                #(parameter WIDTH = 140,
        // default picture width
                                                HEIGHT =140)
                // default picture height
                (input pixel_clk,hit,
                 input [10:0] x,hcount,
                 input [9:0] y,vcount,
                 output reg [23:0] pixel);

                wire [14:0] image_addr,image_addr_nohit,image_addr_hit;

        // num of bits for 128*256 ROM
                wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
          reg [7:0] image_bits_reg;
```

```
			// calculate rom address and read the location
			assign image_addr_nohit= (hcount-x) + (vcount-y) * WIDTH;
			assign image_addr_hit = WIDTH-((vcount-y)+1)+((hcount-x)*WIDTH);
			assign image_addr=hit?image_addr_hit:image_addr_nohit;
			stick_image_rom2
romstick(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

			// use color map to create 8bits R, 8bits G, 8 bits B;
			red_color_map_stick                                    rcm_stick
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
			green_color_map_stick
gcm_stick(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
			blue_color_map_stick
bcm_stick(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));

			// note the one clock cycle delay in pixel!
			always @ (posedge pixel_clk) begin
			  image_bits_reg <= image_bits;      //This stabilizes the address before
feeding it into Color maps
				if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
						(vcount >= y && vcount < (y+HEIGHT)))
						pixel     <=     {red_mapped,     green_mapped,
blue_mapped};
				else pixel <= 24'h000000;
			end


		endmodule
		////
		//////////////////////////////////////////////////////
		////
		//// picture_blob: Display Stick_left
		////
		//////////////////////////////////////////////////////

		module pic_left_stick
			#(parameter WIDTH = 140,
		// default picture width
								HEIGHT =140)
			// default picture height
			(input pixel_clk,hit,
			 input [10:0] x,hcount,
			 input [9:0] y,vcount,
			 output reg [23:0] pixel);
```

```
                wire [14:0] image_addr,image_addr_nohit,image_addr_hit;

        // num of bits for 128*256 ROM
                wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
          reg [7:0] image_bits_reg;
                // calculate rom address and read the location
                assign image_addr_nohit=(vcount-y)+((WIDTH-((hcount-x)+1)) * WIDTH);
                assign    image_addr_hit    =    (WIDTH-((hcount-x)+1))+((WIDTH-((vcount-
y)+1))*WIDTH);
                assign image_addr=hit?image_addr_hit:image_addr_nohit;
                stick_image_rom2
romstick(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

                // use color map to create 8bits R, 8bits G, 8 bits B;
                red_color_map_stick                                        rcm_stick
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
                green_color_map_stick
gcm_stick(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
                blue_color_map_stick
bcm_stick(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));

                // note the one clock cycle delay in pixel!
                always @ (posedge pixel_clk) begin
                  image_bits_reg  <=  image_bits;      //This  stabilizes  the  address  before
feeding it into Color maps
                        if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
                                        (vcount >= y && vcount < (y+HEIGHT)))
                                        pixel      <=      {red_mapped,      green_mapped,
blue_mapped};
                        else pixel <= 24'h000000;
                end


        endmodule

        ////////////////////////////////////////////////

         ////////picture_blob: floor tom

        ///////////////////////////////////////////////
        module floor_tom_pic
                #(parameter WIDTH = 225,
        // default picture width
```

```
                                    HEIGHT =225)
                // default picture height
                (input pixel_clk,
                 input [10:0] x,hcount,
                 input [9:0] y,vcount,
                 output reg [23:0] pixel);

                wire [15:0] image_addr;
                                                        // num of bits for 128*256 ROM
                wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
            reg [7:0] image_bits_reg;

                // calculate rom address and read the location
                assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
                floortom_image_rom
rom1(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

                // use color map to create 8bits R, 8bits G, 8 bits B;
                red_floortom_map                                                    rcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
                green_floortom_map                                                  gcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
                blue_floortom_map                                                   bcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));

                // note the one clock cycle delay in pixel!
                always @ (posedge pixel_clk) begin
                  image_bits_reg <= image_bits;     //This stabilizes the address before
feeding it into Color maps
                    if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
                                (vcount >= y && vcount < (y+HEIGHT)))
                                pixel    <=    {red_mapped,    green_mapped,
blue_mapped};

                    else pixel <= 24'h000000;
                end


        endmodule

        ///////////////////////////////////////////////

         ////////picture_blob: cymbal
        ///////////////////////////////////////////
        module pic_cymbal
                #(parameter WIDTH = 225,
        // default picture width
                                        HEIGHT =200)
                // default picture height
                (input pixel_clk,
                 input [10:0] x,hcount,
                 input [9:0] y,vcount,
                 output reg [23:0] pixel);

                wire [15:0] image_addr;
                                                        // num of bits for 128*256 ROM
                wire [7:0] image_bits, red_mapped,green_mapped, blue_mapped;
          reg [7:0] image_bits_reg;

                // calculate rom address and read the location
                assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
                cymbal_image_rom
rom1(.addra(image_addr),.clka(pixel_clk),.douta(image_bits));

                // use color map to create 8bits R, 8bits G, 8 bits B;
                red_cymbal_map                                                  rcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(red_mapped));
                green_cymbal_map                                                gcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(green_mapped));
                blue_cymbal_map                                                 bcm
(.addra(image_bits_reg),.clka(pixel_clk),.douta(blue_mapped));

                // note the one clock cycle delay in pixel!
                always @ (posedge pixel_clk) begin
                  image_bits_reg <= image_bits;      //This stabilizes the address before
feeding it into Color maps
                        if ((hcount-3 >= x && hcount-3 < (x+WIDTH)) &&
                                (vcount >= y && vcount < (y+HEIGHT)))
                                pixel      <=      {red_mapped,      green_mapped,
blue_mapped};

                        else pixel <= 24'h000000;
                end


        endmodule
        module collision_engine(
                input clock,
                input reset,
                input direction,
                input [9:0] velocity,
                input [10:0] x_hand,
                input [9:0] y_hand,
        //      output [9:0] gain,
                //output [9:0] gain_yellow,
                output reg cymbal_pulse,
                output reg floor_pulse,
                output reg snare_pulse);

                parameter state_wait=0;
                parameter state_hit=1;

                reg [1:0] state,next_state;

                always @(*) begin
                        if (reset) next_state=state_wait;       //Default state WAIT state
                        else case (state)

        state_wait:next_state=(velocity>10&&direction==1&&y_hand>420)?state_hit
                                        :state_wait;
                                state_hit:next_state=state_wait;
                        endcase
                end

                always  @(posedge  clock)  begin       //Using  the  40  Mhz  clock  of  video
module
                state<=next_state;
                        if (next_state==state_wait) begin
                                cymbal_pulse<=0;
                                floor_pulse<=0;
                                snare_pulse<=0;
                        end
                        else if(next_state==state_hit) begin
                                if (x_hand<276) cymbal_pulse<=1;
                                else if(x_hand>276&&x_hand<496)floor_pulse<=1;
                                else if(x_hand>496) snare_pulse<=1;
                        end
                end

        endmodule
```

Game Audio

```
module sound_fetch_playback(
  input clock,              // 27mhz system clock
  input reset,              // 1 to reset to initial state
  input ready,
  input sound_hit_pulse,            // 1 for cymbal
  input sound_max_address,
  output[7:0] sound_out_address    // 8-bit PCM data to headphone
);

parameter state_idle=0;
parameter state_play=1;
parameter state_replay=2;


reg [1:0] state,next_state;
always @(*) begin
          if (reset) next_state=state_idle;
          else case(state)
                    state_idle:next_state=sound_hit_pulse?state_play
                                        :state_idle;

state_play:next_state=(cymbal_sound_addr==sound_max_address)

?state_idle:((sound_hit_pulse)?state_replay:state_play);
                    state_replay:next_state=state_play;
          endcase
end

always @(posedge clock) begin
state<=next_state;
          if (next_state==state_idle) begin
                    sound_out_address<=0;
          end
          else if(next_state==state_play) begin
                    if (ready) begin
                              sound_out_address<=sound_out_address+1;
                    end
          end
          else if(next_state==state_replay) begin
                    sound_out_addressr<=0;
          end
end
```

```
endmodule
```

Music Audio

```
#! /usr/bin/env python

import serial
import wave
import struct
import scipy

#w = wave.open('lion_resampled_48000Hz.wav','rb')

#luis's labkit
#ser = serial.Serial(port='/dev/tty.usbserial-FTDHKA57')#, baudrate=921600)

#rishi's labkit
ser = serial.Serial(port='/dev/tty.usbserial-A900a0YF')

a = open('Petrucci_test1.coe','r')

for line in a:

  line = line.rstrip()
  line = int(line)
  a = struct.pack("<H", line)

  #print line#repr(a[0])

  r = ser.write(a[0])#'\x01')#line)

#print repr(b_array[0])

ser.close()
```