Target Hunter

Austin Phillips-Brown and Casey Wessel

December 10, 2014

Final Project Report
6.111 Introduction to Digital Systems Fall 2014
Massachusetts Institute of Technology

Abstract

For this project, the 6.111 lab kit FPGA, a gyroscopic sensor, and an external camera are used to create a compact and inexpensive version of an arcade shooting game. In the game, objects are displayed on a monitor, and the player has a plastic rifle, which they point at the screen to "shoot" the targets. Digital logic on the FPGA is used for two main purposes: the first is to analyze an image of the player aiming the controller, or rifle, to calculate where the player is aiming. The second has the FPGA reading data from the gyroscopic sensor, sent by an Arduino, to calculate pitch and yaw angle of the rifle. Finally, the BRAM on the 6.111 lab kit stores a variety of images to display as backgrounds and targets during the game.

Table of Contents

I. Introduction………………………………………………………………………...…4
   A. Project Overview…………………………………………………………….....4
   B. Motivation…………………………………………………………...………....4
II. Summary……………………………………………………………………………….4
   A. Display……………………………………………………………………….....5
   B. Camera Tracking…………………………………………………………....5
   C. Gyro Sensor……………………………………………………………………5
   D. Sound………………………………………………………………………...5
III. Implementation……………………………………………………….………………....6
   A. Display…………………………………………………………………………6
   B. Camera Tracking……………………………………………………………...8
   C. Rifle……………………………………………………………………………..9
   D. Gyro and Tangent Calculation…………………...…………....……………………10
   E. Finite State Machine…………….……………………………………….....10
IV. Testing…………………………………………………………………………………11
V. Review and Recommendations…………………………………………………...12
VI. Conclusion…………………………………………………………………………….13
VII. Glossary……………………………………………………………………………….15

Appendices
   A. Arduino Code…………………………………………………………...16
   B. Top Level Verilog(Camera Tracking and Display)......................................18
   C. Sound Verilog………………………………………………………......74
   D. Gyro Sensor and Tangent Calculation Verilog……………………………....92

Introduction

Project Overview

This project is a virtual target shooting game similar to a shooting video game typically played in an arcade. The standard version in an arcade is bulky - the rifle is permanently attached to the machine and the entire system costs thousands of dollars to manufacture as well as to maintain. Our version is a low-cost alternative that uses an FPGA and an Arduino to do the "heavy lifting" of the game. What makes our "light" version of the game possible are the methods that we use to track the movement of the rifle - a combination of image tracking and rifle angle calculations. The game first begins by displaying options for game levels - 'beginner', 'intermediate', and 'expert'. The player has a physical plastic rifle, which they use as a selector; the player selects their game level by aiming at the desired level and pulling the trigger. The screen displays an animal whose size is based on the level selected - higher levels have smaller animals. The player aims the rifle at the displayed targets and tries to "hit" them. As shown in Figure 1, a combination of image tracking and gyroscopic data are used to calculate where the player is aiming on the screen. Using that data, an image of a sight will be displayed on the screen so that the player can see where they are aiming. To fire, the player pushes a button on the rifle, which will be wired to the FPGA. If the player hits the target within the time limit, the screen with transition to a screen where the word "Hit!" appears. Once the player pushes the fire button again, they are returned to the game where the animal will begin moving faster. After the player advances through all speeds of their particular animal, the animal is switched to a smaller counterpart. The game also has a pause capability - using a toggle switch located on the rifle, the player can pause the timer and motion of the animal.

Motivation

Our motivation in building this game began with a shared interest in the sport of shooting. One of the challenges of practicing shooting with a rifle is the restriction on practicing in a safe environment, like a shooting range. For this reason, we both enjoy playing shooting games at arcades. These arcade games, however, are large, costly machines that we could never own ourselves. Implementing this game on the FPGA will allow us to simulate practicing a sport we enjoy, in a more readily accessible way. This project has the added benefit that it will be fun to play and test while working in the lab.

Summary

The game divides into four main modules: Display, Camera Tracking, Gryo Sensor, and Sound. As shown in Figure 1, these modules lead to central module called Finite State Machine (FSM), which will be discussed further in the Implementation section.

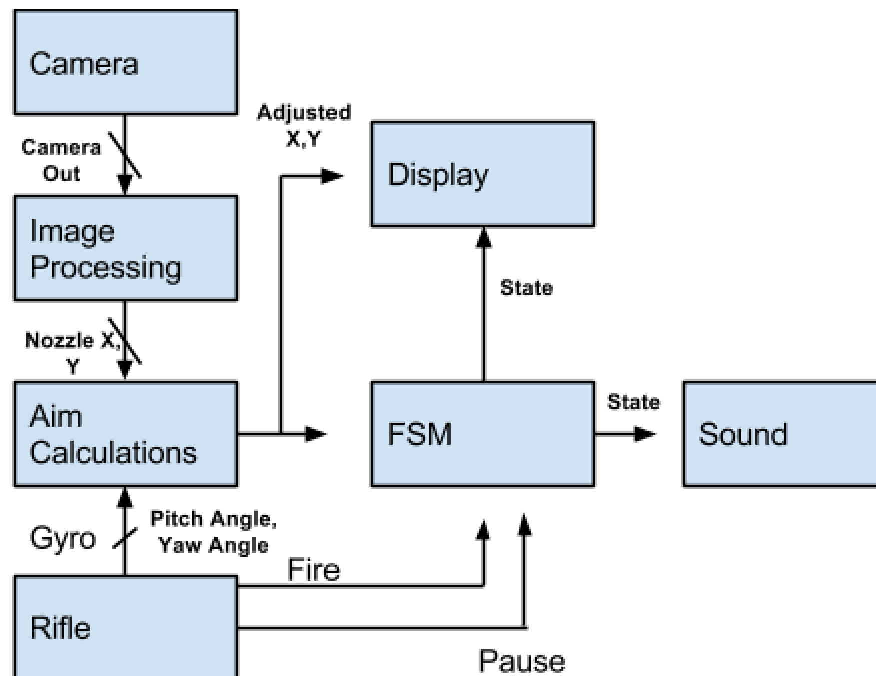

Figure 1. System Block Diagram
Outside hardware - Camera and Rifle Inputs, FPGA Modules - Aim/Target Overlap, Score, Display, FSM
Arduino Modules - Image Processing, Aim Calculations

Display

The Display module holds all of the images that will be displayed at different times in the game and also controls where on the screen the images will appear. The display modules also controls the movements of the sprites on the various levels.

Camera Tracking

The Camera Tracking modules takes in data from an NTSC camera. The module then converts this data to RGB color values and then to HSV color values. Searching through the data pixel by pixel, the module finds pixels that correspond to the color of the tip of the rifle, in this case, a neon green ping pong ball.

Gyro Sensor

The Gyro Sensor module collects data from the gyro sensor mounted on the rifle. This module calculates the displaced angle and then does a tangent calculation to calculate the aim position from the pitch or yaw angle of the rifle.

Sound

The sound module stores several different sound effects including a rifle firing, nature sounds, and a turkey gobble. This module plays the sound effects when called by a finite state machine.

Implementation

Display

Author:Casey Wessel

The main purpose of the display module is to provide the user interface for the player. To save as much memory as possible the display is in 640x480 resolution. Some adjustments were made to account for the 640x480 resolution. To maintain a 60Hz refresh rate, the display needs to use a 25 Mhz clock, rather than the typical 65 Mhz. All images displayed in the game are stored in rom, except for the sight, which is a real-time generated sprite.Therefore, the images are generated and implemented in the most frugal manner possible.

To convert images to the proper size and format to be used on the FPGA, we used the MATLAB code provided by the 6.111 staff. The code takes a bitmap image and converts the image to COE files that can be used by the ISE CORE Generator. The following images are in four-bit color and have four COE files associated with each one. Each image has one COE file which is four bits wide and [pixel-width x pixel_height] entries deep. Each image also has a red, green, and blue color table COE file.

- **Background Forest Image** 640x480
- **“Target Hunter” Title** 500x100
- **“Beginner” Block** 200x75
- **“Intermediate” Block** 200x75
- **“Expert” Block** 200x75
- **Deer** 100x100
- **Turkey** 50x50

- **Duck** 25x25
- **“Hit!” Block** 100x50

CORE Generator was used to generate the ROM for each COE file. Photo 1 shows the initial setup screen for creating a single port ROM.

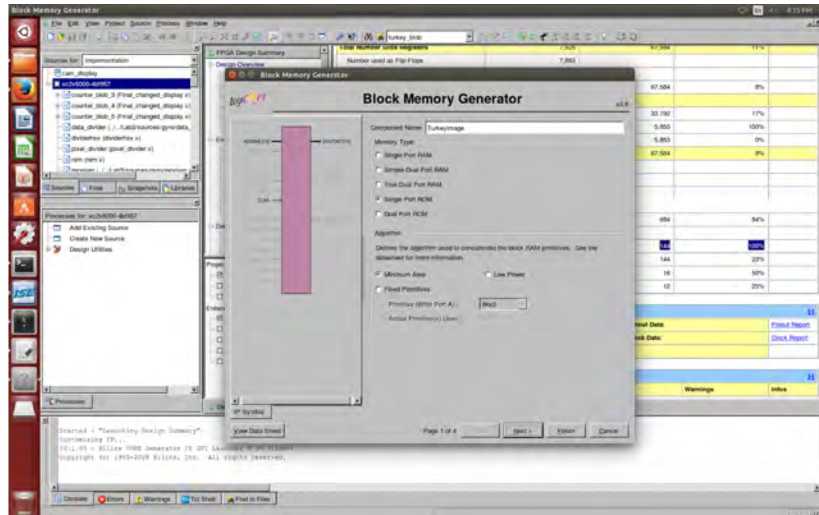

Photo 1. Setup to create single port ROM

Below Photo 2 shows the initial title screen for the game. The image is actually composed of 5 total images: the background, the title, and the three level blocks. This was done to save space because the background image is used again in each level.

Photo 2. Title Screen for Target Hunter
Allows player to select start level based on difficulty.

Below, Photo 3 shows the final design of the beginner level. The targets in all cases use the code for the “puck” movement from the Pong lab. While the Beginner level only displays one deer, the Intermediate level displays two turkeys, and the Expert level displays two ducks.

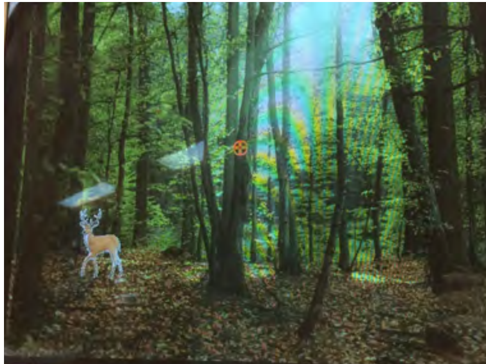

Photo 3. Deer Target for Beginner Level

Photo 4. Turkey Target for Intermediate Level

Photo 5. Duck Target for Expert Level

Camera Tracking

Author: Austin Phillips-Brown

The camera tracking module calculates the location of the tip of the rifle by visually tracking a specific color. Staff code provided on the 6.111 website was used to take the data from the NTSC camera and convert it to RGB values that could be displayed by the monitor. From there, we then converted the RGB values to HSV for ease of tracking[1]. The color image is then processed on the FPGA in the following manner: We search the pixels for the color of the green ping-pong ball that we set on the rifle nozzle and have a running sum of the x-value and y-value of those locations. We then divide those sums to find the average, or center, location that contains the green pixel. This center point is what we use for the location of the

[1] HSV places most of its “color” information in the “H” portion of its pixel value.

rifle nozzle. In Photo 6 below, this location is indicated by the intersection of the vertical and horizontal yellow lines.

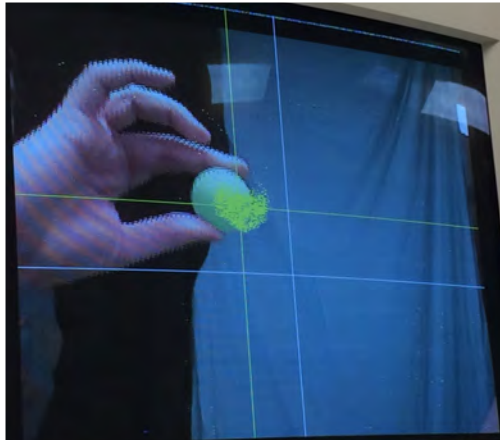

Photo 6. Tracking the Ping-Pong Ball
Yellow pixels displayed where the verilog code read “green”.

Rifle

Author: Casey Wessel

The rifle is the main controller used by the player. The rifle is how the player controls movement on the screen and indicates a “fire”.

Photo 7. Photo of Rifle Controls
Shows “fire” button, “safety” switch, and gyro sensor.

The Rifle module has three key aspects, the first two of which are relatively simple. There have a button/switch system on the rifle that relay the “fire” and “pause” commands to the FPGA respectively. The button is normally disconnected and connects 5v from the FPGA to the User1[0] io port of the FPGA when pressed. Because the io port on the FPGA is

normally floating, a pull-up circuit like the one pictured in Figure 2 was used. The setup for the Pause function is very similar but uses a switch to connect a different io port The idea behind the switch for the Pause function is that it is similar to a safety switch on a rifle. Finally, the gyro sensor is mounted on the stock of the rifle. The relevant pins from the gyro lead to an Arduino Uno board next to the lab kit.

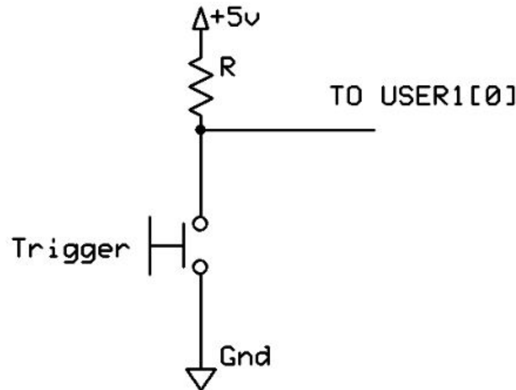

Figure 2. Pull-up Circuit Used for Fire

Gyro and Tangent Calculation

Author: Casey Wessel

The gyroscope finds the angle of the rifle with respect to the x and y axis. The Arduino interfaces with the Adafruit L3DG20H gyro sensor board so that the Arduino can execute the SPI protocols to interpret the data from the gyro. The Arduino reads the data for change in degrees per second sensed by the gyro and then, using the Arduino code in Appendix A, sends the data to the FPGA IO ports User3[0] and User3[1] on the lab kit via a serial protocol similar to the one implemented in the Infrared lab. A module implemented on the FPGA uses an altered version of the serial read code written in the Infrared lab to read the data sent from the Arduino.

A running total of the change in angle is kept throughout the game. Because trigonometric functions are tricky to implement on the FPGA, the angle is sent to a lookup table to find the tangent value. The lookup table is of reasonable size because range of movement for the player would typically be no more than plus or minus fifteen degrees in either the pitch or yaw direction. Finally, the value received from the lookup table is multiplied by a scaling factor to calculate the change in either the x or y direction due to the pitch or yaw angle of the gun. This final output is then added to the x and y pixel coordinates from image tracking to calculate the final (x,y) coordinate of the sight.

Finite State Machine (FSM)

Author: Austin Phillips-Brown

The FSM controls the main operation of the game. Each "level" of the game is assigned a state. The states are assigned as follows:

State 0: Title Screen
State 1: Beginner Level
State 2: Intermediate Level
State 3: Expert Level
State 4: "Hit!" Transition

The game begins on the title screen; this is where the player selects which level they would like to start. The player hovers the sight over which level they would like to play. When the "fire" trigger is pressed, the FSM checks to see which pixel range the center of the sight falls in. The FSM then changes state to the state of the selected level. When entering a new state, the FSM outputs a "start" signal which is sent to an external timer module. The timer module starts a ten second countdown. The player has ten seconds to hit the target, or else the player is booted out of the level and sent back to the start screen.

Sound

Author: Austin Phillips-Brown

The Sound Module controls the audio that is outputted for the various stages of the game. The Sound Module is loaded on a second FPGA which receives data from the main FPGA via wires connecting to the IO ports. Based upon what data the FSM Module sends, the Sound module decides what audio data must be sent to the speakers. To load the audio files onto the FPGA, we used the MATLAB script provided by the 6.111 staff to convert short .wav audio files into a COE file. We then loaded that COE file into RAM and then accessed the RAM via Verilog. We placed the following sound files into the FPGA:

TurkeySound - This is the background music that plays during most of the levels and the
title screen.

TurkeyGobble - The sound file that is played during the Intermediate level.

GunShot - the "shooting" sound that is played at the title screen and if a player hits a target.

Testing Plan

The visual aspect of this game makes testing fairly straight-forward. Because a large portion of the project is visual, it is easy to display many of the modules on VGA to see if the modules are producing correct data. This combined with the hex display with which values can be displayed in real time allowed us to easily run and test our code at the same time. For example, to test that all of the images were properly loaded into the ROM, instances of each image were created and called to display on VGA.

For the Camera Tracking module, we could simply have all pixels that matched our HSV values turn yellow so that we had visual confirmation with what the camera believed to match the values. For the Display Module, we could see on the VGA monitor how the images appeared and get a general idea about the possible problems, such as the need for clock delays.

To test the Gyro Module, we created a small square in the center of the VGA screen, which moved based upon the calculations of our tangent module. Based upon how the block moved, we could easily tell if our gyro code was actually interpreting the data from the gyroscope correctly.

Review and Recommendations

Upon completion of the game, there are several areas that

Originally, we were using 800 x 600 resolution for our display which caused us to have to transform the pixel coordinates given by the Camera Module via formula 1 due to the camera not providing a perfect 800 x 600.

$$\frac{Position_{Camera}}{Res_{Camera}} * Res_{Display}$$

Formula 1:
Converting camera position to VGA position.

Due to complications with image size and ram limitations, we were forced to downsize our display to 640 x 480 which largely eliminated the need for transforming the coordinates.

play.ise - [Design Summary]

Total Num... Slice Registers	7,925	67,584	11%	
Number used as Flip Flops	7,883			
Number used as Latches	42			
Number of 4 input LUTs	5,951	67,584	8%	
Logic Distribution				
Number of occupied Slices	5,853	33,792	17%	
Number of Slices containing only related logic	5,853	5,853	100%	
Number of Slices containing unrelated logic	0	5,853	0%	
Total Number of 4 input LUTs	6,418	67,584	9%	
Number used as logic	5,768			
Number used as a route-thru	467			
Number used as Shift registers	183			
Number of bonded IOBs				
Number of bonded	577	684	84%	
IOB Flip Flops	6			
Number of RAMB16s	144	144	100%	
Number of MULT18X18s	34	144	23%	
Number of BUFGMUXs	8	16	50%	
Number of DCMs	3	12	25%	

Performance Summary			
Final Timing Score:		**Pinout Data:**	Pinout Report
Routing Results:		**Clock Data:**	Clock Report
Timing Constraints:			

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**

Photo 8. Utilization of BRAM

Camera Tracking

We discovered that the camera picks up a little bit of noise due to all of the lights in the lab. In most cases, if the image you are tracking is fairly large and not too far away, this does not cause too many problems. However, our game tracks a ping-pong ball which, when about four feet away, takes up very few of the pixels that the camera is reading. In this scenario, the small spots of noise drastically alter the effectiveness of the camera tracking. The only way to solve this problem was to hang up sheets of solid colors as a background. This cut out almost all of the background noise that the camera was picking up.

Photo 9. Noise Reduction Technique

Gyro Sensor

Due to the complication described in the subsection above, the player needs to stand relatively close to the game (feet should be five to six feet away from the camera). In this proximity, the gyro sensor does not sense any change that is not accounted for in the change of movement on the camera. For our game, we did have a version both with and without the gyro and tangent calculations being incorporated. Both versions were almost identical, minus a small lag time in the version with the gyro sensor. This is because of the calculations occurring in the gyro sensor and tangent calculation modules. For future implementations of this game, we would suggest removing the gyro sensor entirely. While the gyro sensor does work, it is not necessary when the player must be in such close proximity to the camera.

Target Movement

Through playing the game, we discovered that the game can be easy because the animal follows a pong-like path as it moves around the screen. For future implementations of this game, we suggest that a “random teleportation” movement be implemented where the animal appears for one or two seconds, disappears, and randomly appears in a new location on the screen. We think this would make the game much more challenging.

Counter Implementation and Permanent Memory

Our original goal was to implement a counter on the display that would show the player how long they had left in the level, but we used all of the BRAM to store our main images. Eventually, we simply discarded the counter images to save space. It was later suggested by Professor Gim Hom that some of the images could be stored in the ZBT permanent memory. In that way, we would not have to use as much BRAM to store the files as a majority could be stored in the permanent memory. This would have saved space in the BRAM and allowed us to have our visual counters.

Conclusion

Though we had significant problems during the five weeks we were allotted for the project, we managed to successfully finish our game. Using a combination of camera tracking, gyroscope data measurements, and basic VGA display, our game worked above and beyond our expectations. This does not mean that there is nothing to improve upon; our game had notable flaws that we were forced to ignore. First, though we had a tentative version of the gyroscope code working, it was buggy and gave little of value to the aiming mechanism as a whole. Second, we were forced to remove the visual notification of time left on the display due to running out of available RAM on the FPGA. Nevertheless, we did manage to meet almost all of our proposed goals including the extra sound module. As such, we can easily say that our proposed project was a success and a fun and entertaining game to play.

Glossary

Fire
The signal which says a player has pressed the trigger to fire the rifle.

Gyroscopic (Gyro) Sensor
A sensor which detects rate of change in degrees or radians per second in all three axis (x, y, z).

HSV (Hue Saturation Value)
HSV is way to express the value of a color in terms of hue saturation and value. It is useful because most of the ‘color’ information is expressed in the “hue” term.

RGB (Red Green Blue)
HSV is way to express the value of a color in terms of the ratio of red, green, and blue within a color.

Sprite
A small image that is used several times within a game that is usually overlaid over a larger background image.

Target
In each level the “target” is the animal sprite that appears and moves around the screen.

Pause

Signal from the rifle that says the player wished to pause the game. Signal is activated by the player flipping the “safety” switch on the rifle.

VGA
Video Graphics Array. This is the protocol used by the monitor to display images.

Appendices

Appendix A: Arduino Code

```
#include <Wire.h>
#include <Adafruit_L3GD20.h>

byte transmit_one = 10;
byte transmit_two = 11;
byte mask = 1; //our bitmask

// To use SPI, you have to define the pins
#define GYRO_CS 4 // labeled CS
#define GYRO_DO 5 // labeled SA0
#define GYRO_DI 6  // labeled SDA
#define GYRO_CLK 7 // labeled SCL
Adafruit_L3GD20 gyro(GYRO_CS, GYRO_DO, GYRO_DI, GYRO_CLK);

void setup()
{
  Serial.begin(9600);
  pinMode(transmit_one,OUTPUT);
  pinMode(transmit_two,OUTPUT);

  // Try to initialise and warn if we couldn't detect the chip
   if (!gyro.begin(gyro.L3DS20_RANGE_250DPS))
```

```
   pinMode(transmit_one,OUTPUT);
   pinMode(transmit_two, OUTPUT);
}

void loop()
{
  gyro.read();
  int data_one = gyro.data.z;
  int data_two = gyro.data.y;

  digitalWrite(transmit_one,HIGH);
  delayMicroseconds(2000);
  digitalWrite(transmit_one,LOW);
  delayMicroseconds(200);

  for (mask = 00000001; mask>0; mask<<=1) { //iterate through bit mask
    if (data_one & mask){ // if bitwise AND resolves to true
     digitalWrite(transmit_one,HIGH); // send 1
     delayMicroseconds(600);
     digitalWrite(transmit_one,LOW);
    }

    else { // if bitwise AND resolves to true
      digitalWrite(transmit_one,HIGH); // send 0
      delayMicroseconds(300);
      digitalWrite(transmit_one,LOW);
    }
    delayMicroseconds(200);
  }

  digitalWrite(transmit_two,HIGH);
  delayMicroseconds(2000);
  digitalWrite(transmit_two,LOW);
  delayMicroseconds(200);

  for (mask = 00000001; mask>0; mask<<=1) { //iterate through bit mask
    if (data_two & mask){ // if bitwise AND resolves to true
      digitalWrite(transmit_two,HIGH); // send 1
      delayMicroseconds(600);
      digitalWrite(transmit_two,LOW);
    }

    else { // if bitwise AND resolves to true
```

```
    digitalWrite(transmit_two,HIGH); // send 0
    delayMicroseconds(300);
    digitalWrite(transmit_two,LOW);
   }

   delayMicroseconds(200);
  }
   delayMicroseconds(10000);
}
```

Appendix B: Top Level Module (Camera Tracking and Display)

```
//
// 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] select 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
//switch
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
module debounce(input reset, clock, noisy,
                output reg clean);

  reg [19:0] count;
  reg new;

  always @(posedge clock)
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 650000) clean <= new;
        else count <= count+1;

endmodule
// 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);  //cwessel

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

   // 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;

/*   ////////////////////////////////////////////////////////////////////////////
   // 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 12
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 11
   // 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),
                                .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;
 wire fire;
// ENTER button is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clk, ~button_enter, user_reset);
 wire up, down, right, left, but0, but1,but2,but3;
 debounce db2(power_on_reset, clk, ~button_down, down);
 debounce db3(power_on_reset, clk, ~button_up, up);
 debounce db4(power_on_reset, clk, ~button_right, right);
 debounce db5(power_on_reset, clk, ~button_left, left);
 debounce db6(power_on_reset, clk, ~button0, but0);
 debounce db7(power_on_reset, clk, ~button1, but1);
 debounce db8(power_on_reset, clk, ~button2, but2);
 debounce db9(power_on_reset, clk, ~button3, but3);
 debounce db10(power_on_reset, clk, user1[0], fire);
assign reset = user_reset | power_on_reset;
 reg [10:0] hcounter = 400;
 reg [9:0] vcounter = 300;
 reg [7:0] huecompare=128;
 reg [7:0] satcomparelow = 128;
 reg [7:0] satcomparehigh = 255;
// display module for debugging
 reg [24:0] clockdelay=0;
 always @(posedge clk) begin
        clockdelay<=clockdelay+1;
        if(clockdelay==500000) begin
                clockdelay<=0;
                if(up) begin
                        huecompare<=huecompare+1;
                        if (vcounter>1)
                                vcounter<=vcounter-1;
```

```
				else
					vcounter<=vcounter;
			end
			else if (down) begin
				huecompare<=huecompare-1;
				if(vcounter<600)
					vcounter<=vcounter+1;
				else
					vcounter<=vcounter;
			end
			if (right) begin
				if(hcounter<800)
					hcounter<=hcounter+1;
				else
					hcounter<=hcounter;
			end
			else if (left) begin
				if(hcounter>1)
					hcounter<=hcounter-1;
				else
					hcounter<=hcounter;
			end
			if (but0) begin
				satcomparelow<=satcomparelow-1;
			end
			else if (but1) begin
				satcomparelow<=satcomparelow+1;
			end
			if (but2) begin
				satcomparehigh<=satcomparehigh-1;
			end
			else if (but3) begin
				satcomparehigh<=satcomparehigh+1;
			end
		end
 end
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,hsyncdelay,vsyncdelay,blankdelay;
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;
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
 reg [23:0] rgb_reg;
```

```
wire [23:0] rgb, rgbdone;
wire [23:0] hsv,hsv_done;
reg [30:0] xacc;
reg [30:0] yacc;
reg [30:0] xacc1=0;
reg [30:0] yacc1=0;
reg [22:0] counter_hold=1;
wire [15:0] poop, pooptastic;
wire [30:0] yavg, xavg;
wire [10:0] yavg_reg;
wire [10:0] xavg_reg;
reg [23:0] hsvreg;
wire yready, xready;
wire hsyncdelay2, vsyncdelay2, blankdelay2;

reg [22:0] counter=1;
reg sawit = 1;
YCrCb2RGB work(.R(rgb[23:16]), .B(rgb[7:0]), .G(rgb[15:8]), .clk(clk),
.rst(reset), .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]));
delayN#(.NDELAY(35)) hsync2(.clk(clk), .in(hsync), .out(hsyncdelay));
delayN#(.NDELAY(35)) vsync2(.clk(clk), .in(vsync), .out(vsyncdelay));
delayN#(.NDELAY(35)) blank2(.clk(clk), .in(blank), .out(blankdelay));
        always @(posedge clk) begin
                rgb_reg <= rgb;
        end

assign rgbdone = rgb_reg;

//(rgb_reg[23:16] >230 && rgb_reg[15:8] >60 && rgb_reg[15:8] <110) ? {8'd255, 8'd255,
8'd0} :
wire [17:0] rgbdata = {rgbdone[23:18], rgbdone[15:10], rgbdone[7:2]};
wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire        ntsc_we;
ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, rgbdata,
            ntsc_addr, ntsc_data, ntsc_we, 0);

// 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 = {4{count[3+4:4],4'b0}};
```

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

  wire     sw_ntsc = 1;
  wire     my_we = sw_ntsc ? (hcount[0]==1'd1) : blank;
  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;  //Caseydidn't comment
this section out
//   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] iseepixel;
   reg [63:0] dummyval;
   reg [63:0] dummyval1;

  reg [23:0] pixel;
   reg [23:0] pixel1;

   wire [23:0] pixel_game;
   reg [23:0] pixel_dummy;

   wire phsync,pvsync,pblank;
   wire start_on, expire, second;
   wire[3:0] delay_dis;

   wire signed[7:0] x_value, y_value;
   reg [10:0] storex, storey;
   wire [10:0] x_newer, y_newer;
   wire clock;
   wire start_x, start_y;
   wire signed[5:0] angle_x, angle_y;
   wire signed[11:0] x_point, y_point;
   reg [2:0] level=1;

   timer_divider divide(.clock(clock_40mhz), .second(second));
```

```
   timer countdown(.start(start_on),.stop(user1[1]), .clock(second), .expire(expire),
   .delay_dis(delay_dis));


   gyro_divider newClock(.oldclock(clock_27mhz), .clock(clock));

   data_interp data_read (.data(user3[0]), .clock(clock), .start(start_x), .val_out(x_value));
   data_interp data_read_two(.data(user3[1]), .clock(clock),.start(start_y), .val_out(y_value));

   calculator calc (.data(x_value), .start(start_x), .angle(angle_x));
   calculator calc1(.data(y_value), .start(start_y), .angle(angle_y));
   aim_calc point_x (.angle(angle_x), .clock(clock),
    .level(level), .new_distance(x_point));

    aim_calc point_y(.angle(angle_y), .clock(clock),
    .level(level), .new_distance(y_point));
   assign x_newer = switch[1] ? (xavg_reg + x_point) : xavg_reg;
   assign y_newer = switch[1] ? (yavg_reg + y_point) : yavg_reg;
   always @(posedge clock_40mhz) begin
         storex<=x_newer;
         storey<=y_newer;
   end
  pong_game pg(.vclock(clock_40mhz),.reset(reset),
                 .up(up),.down(down),.pspeed(2),
         .hcount(hcount),.vcount(vcount),.music(user1[3:2]),
                 .hsync(hsync),.vsync(vsync),.blank(blank),.gun(user1[4]),

.stop(user1[1]),.phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel_game),.switch(s
witch[1]),.delay_dis(delay_dis),
.fire(fire),    .expire(expire),.start_on(start_on), .x(storex), .y(storey));
  reg    b,hs,vs, b1, b2, hs2, vs2, hs1, vs1;
   reg [4:0] averagecount=0;
   reg [26:0] totalpix=0;
   reg [63:0] showmepixel;
   reg [191:0] storeme=0;
   reg [9:0] hsub, vsub;
   reg happy;
  reg [7:0] vga_out_red_reg, vga_out_green_reg, vga_out_blue_reg, vga_out_sync_b_reg,
   vga_out_blank_b_reg, vga_out_hsync_reg, vga_out_vsync_reg;
   always @(*) begin
         totalpix = storeme[23:0] +storeme[47:24]+storeme[71:48]+storeme[95:72]
         +storeme[119:96]+storeme[143:120]+storeme[167:144] + storeme[191:168];
   end
```

```
   reg [23:0] hsvreg1;
   rgb2hsv color(.r(pixel1[23:16]), .b(pixel1[7:0]), .g(pixel1[15:8]),
   .clock(clk), .reset(reset), .v(hsv[7:0]), .s(hsv[15:8]), .h(hsv[23:16]));
   delayN#(.NDELAY(35)) hsync1(.clk(clk), .in(hsyncdelay), .out(hsyncdelay2));
   delayN#(.NDELAY(35)) vsync1(.clk(clk), .in(vsyncdelay), .out(vsyncdelay2));
   delayN#(.NDELAY(35)) blank1(.clk(clk), .in(blankdelay), .out(blankdelay2));
   always @(posedge clk) begin
         hsvreg<=hsv;
   end
   assign hsv_done=hsvreg;
   reg [23:0] hsv_stor;
   always @(posedge clk) begin
         hsv_stor<=hsvreg1;
         if(hcount==0 && vcount==0) begin
               counter<=1;
               xacc<=0;
               yacc<=0;
         end
         else if(hcount<640 && vcount<480) begin
               if(hsvreg1[23:16]>8'h25 && hsvreg1[23:16]<8'h40 && hsvreg1[15:8]> 8'h51 &&
hsvreg1[15:8]<8'h7a && hsvreg1[7:0]> 110 ) begin
                     if(hsv_stor[23:16]>8'h25 && hsv_stor[23:16]<8'h40 && hsv_stor[15:8]>
8'h51 && hsv_stor[15:8]<8'h7a && hsv_stor[7:0]> 110 ) begin
                     //hsv_stor[23:16] >= 100 && && hsv_stor[7:0] >240
                           counter<=counter+1;
                           xacc <= xacc +hcount;
                           yacc<= yacc+vcount;
                     end
               end
         end
         if(hcount==640 && vcount==480) begin
                     xacc1<=xacc;
                     yacc1<=yacc;
                     counter_hold<=counter;
         end
   end

   divider quot(.clk(clk), .dividend(xacc1), .quotient(xavg),
.divisor(counter_hold),.fractional(poop), .rfd(xready));
   divider quot1(.clk(clk), .dividend(yacc1), .quotient(yavg),
.divisor(counter_hold),.fractional(pooptastic), .rfd(yready));
```

```
        //(rgb_reg[23:16] >230 && rgb_reg[15:8] >50 && rgb_reg[15:8] <220) ? {8'd255,
8'd255, 8'd0} :
   assign xavg_reg = (xready)?xavg[10:0] : xavg_reg;
   assign yavg_reg = (yready)? yavg[9:0] : yavg_reg;
   reg pooptastical;
   always @(posedge clk) begin
        hsvreg1<=hsv_done;
        if (vcounter == vcount && hcounter==hcount) begin
                storeme[191:24] <= storeme[167:0];
                storeme[23:0] <= hsvreg1;
                iseepixel<= totalpix[26:3];
            end
        pixel1 <= vr_pixel;
        pixel<=(hsvreg1[23:16]>37 && hsvreg1[23:16]<64 && hsvreg1[15:8]> 81 &&
hsvreg1[15:8]<122 && hsvreg1[7:0]> 110 ) ?{8'd255, 8'd255, 8'd0}: pixel1;
        b1 <= blankdelay2;
        b<=b1;
        hs1 <= hsyncdelay2;
        hs<=hs1;
        vs1 <= vsyncdelay2;
        vs<=vs1;

        vga_out_red_reg <= (    vcount==vcounter ||
hcount==hcounter)?255:(vcount==yavg_reg | hcount==xavg_reg)?255:pixel[23:16];
        vga_out_green_reg<= (    vcount==vcounter ||
hcount==hcounter)?255:(vcount==yavg_reg | hcount==xavg_reg)?255:pixel[15:8];
        vga_out_blue_reg<= (    vcount==vcounter ||
hcount==hcounter)?255:(vcount==yavg_reg | hcount==xavg_reg)?0:pixel[7:0];
        vga_out_sync_b_reg<= 1'b1;
        vga_out_blank_b_reg<=~b;
        vga_out_hsync_reg<=hs;
        vga_out_vsync_reg<=vs;
        showmepixel<= {iseepixel, 8'd0,satcomparehigh,4'd0,satcomparelow,4'd0,
huecompare};
       end
wire[63:0] poopy = {x_value, y_value,{angle_x[5],angle_x[5],
angle_x[5]},angle_x,{angle_y[5],angle_y[5], angle_y[5]}, angle_y, 31'd0};

   display_16hex hex(.reset(reset), .clock_27mhz(clock_27mhz), .data_in(poopy) ,
        .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
        .disp_ce_b(disp_ce_b), .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));
   reg hspong, vspong, bpong;
  always @(posedge clock_40mhz) begin
```

```
        // default: pong
     hspong <= phsync;
     vspong <= pvsync;
     bpong <= pblank;
     pixel_dummy <= pixel_game;
   end

//(pixel[23:16] > 100  & pixel[23:16] <160 &pixel[15:8] >35 & pixel[15:8] <50 & pixel[7:0] >35 &
pixel[7:0] <50)? 255 :
//(pixel[23:16] > 100 & pixel[23:16] <160 & pixel[15:8] >35 & pixel[15:8] <50 & pixel[7:0] >35 &
pixel[7:0] <50)? 255:
    // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clk.
   assign vga_out_red = switch[0] ?pixel_dummy[23:16] : vga_out_red_reg;
   assign vga_out_green = switch[0] ?pixel_dummy[15:8] : vga_out_green_reg;
   assign vga_out_blue = switch[0] ? pixel_dummy[7:0]: vga_out_blue_reg;
   assign vga_out_sync_b = 1'b1;     // not used
   assign vga_out_pixel_clock =~clock_40mhz;
   assign vga_out_blank_b =switch[0] ?~bpong: ~b;
   assign vga_out_hsync = switch[0]? hspong: hs;
   assign vga_out_vsync = switch[0] ?vspong : vs;


   // debugging

   assign led = ~{vram_addr[18:13],reset,0};

   always @(posedge clk)
        // dispdata <= {vram_read_data,9'b0,vram_addr};
        dispdata <= {ntsc_data,9'b0,ntsc_addr};

endmodule

////////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 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: 1344 pixels total
        display 1024 pixels per line
  wire             hsyncon,hsyncoff,hreset,hblankon;
  assign        hblankon = (hcount == 1023);
  assign        hsyncon = (hcount == 1047);
  assign        hsyncoff = (hcount == 1183);
  assign        hreset = (hcount == 1343);

        vertical: 806 lines total
        display 768 lines
  wire             vsyncon,vsyncoff,vreset,vblankon;
  assign        vblankon = hreset & (vcount == 767);
  assign        vsyncon = hreset & (vcount == 776);
  assign        vsyncoff = hreset & (vcount == 782);
  assign        vreset = hreset & (vcount == 805);

        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
*/

////////////////////////////////////////////////////////////////////////////////
// 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 blank0=0;
 reg vsync0=0;
 reg   hsync0=0;
reg [10:0]     hcount;        // pixel number on current line
reg [9:0] vcount;    // line number

// horizontal: 800 pixels total
// display 640 pixels per line
wire            hsyncon,hsyncoff,hreset,hblankon;
assign          hblankon = (hcount == 639);
assign          hsyncon = (hcount == 655);
assign          hsyncoff = (hcount == 751);
assign          hreset = (hcount == 799);

// vertical: 524 lines total
// display 480 lines
wire            vsyncon,vsyncoff,vreset,vblankon;
assign          vblankon = hreset & (vcount == 479);
assign          vsyncon = hreset & (vcount == 490);
assign          vsyncoff = hreset & (vcount == 492);
assign          vreset = hreset & (vcount == 523);

// 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;
        hsync0 <= hsyncon ? 0 : hsyncoff ? 1 : hsync0;  // active low
         hsync<=hsync0;
        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= next_vblank;
        vsync0 <= vsyncon ? 0 : vsyncoff ? 1 : vsync0;  // active low
         vsync<=vsync0;
```

```
        blank0 <= next_vblank | (next_hblank & ~hreset);
         blank<=blank0;
  end
endmodule

//module counter_blob_0 #(parameter WIDTH = 100, HEIGHT =100)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//    wire [18:0] image_addr;                    //num of bits for 128*256 ROM
//    wire [3:0] image_bits;
//    reg [3:0] image_bit_reg, image_bit_reg0;
//    wire [7:0] red_mapped,green_mapped,blue_mapped;
//    reg [18:0] image_addr_reg, image_addr_reg0;
//    // note the two clock cycle delay in pixel!
//
//    always @ (posedge pixel_clk) begin
//        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//                pixel <= {red_mapped,green_mapped,blue_mapped};
//        else pixel <= 0;
//        image_bit_reg <= image_bits;
//        image_bit_reg0 <= image_bit_reg;
//        image_addr_reg <= image_addr;
//        image_addr_reg0<= image_addr_reg;
//    end
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    zero_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule
//
//module counter_blob_1 #(parameter WIDTH = 100, HEIGHT =100)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//    wire [18:0] image_addr;                    //num of bits for 128*256 ROM
//    wire [3:0] image_bits;
//    reg [3:0] image_bit_reg, image_bit_reg0;
//    wire [7:0] red_mapped,green_mapped,blue_mapped;
```

```
//    reg [18:0] image_addr_reg, image_addr_reg0;
//    // note the two clock cycle delay in pixel!
//
//    always @ (posedge pixel_clk) begin
//         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//                   pixel <= {red_mapped,green_mapped,blue_mapped};
//         else pixel <= 0;
//         image_bit_reg <= image_bits;
//         image_bit_reg0 <= image_bit_reg;
//         image_addr_reg <= image_addr;
//         image_addr_reg0<= image_addr_reg;
//    end
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    one_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule
//
module deer_blob #(parameter WIDTH = 100, HEIGHT =100)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;                //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                   pixel <= {red_mapped,green_mapped,blue_mapped};
         else pixel <= 0;
         image_bit_reg <= image_bits;
         image_bit_reg0 <= image_bit_reg;
         image_addr_reg <= image_addr;
         image_addr_reg0<= image_addr_reg;
   end
```

```
   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   deer_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   deer_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   deer_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   deer_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule
//
//module counter_blob_2 #(parameter WIDTH = 100, HEIGHT =100)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//    wire [18:0] image_addr;                    //num of bits for 128*256 ROM
//    wire [3:0] image_bits;
//    reg [3:0] image_bit_reg, image_bit_reg0;
//    wire [7:0] red_mapped,green_mapped,blue_mapped;
//    reg [18:0] image_addr_reg, image_addr_reg0;
//    // note the two clock cycle delay in pixel!
//
//    always @ (posedge pixel_clk) begin
//        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//                pixel <= {red_mapped,green_mapped,blue_mapped};
//        else pixel <= 0;
//        image_bit_reg <= image_bits;
//        image_bit_reg0 <= image_bit_reg;
//        image_addr_reg <= image_addr;
//        image_addr_reg0<= image_addr_reg;
//    end
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    two_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule
```

```
module counter_blob_3 #(parameter WIDTH = 100, HEIGHT =100)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;              //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                 pixel <= {red_mapped,green_mapped,blue_mapped};
         else pixel <= 0;
         image_bit_reg <= image_bits;
         image_bit_reg0 <= image_bit_reg;
         image_addr_reg <= image_addr;
         image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   three_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module counter_blob_4 #(parameter WIDTH = 100, HEIGHT =100)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;              //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
```

```
                pixel <= {red_mapped,green_mapped,blue_mapped};
        else pixel <= 0;
        image_bit_reg <= image_bits;
        image_bit_reg0 <= image_bit_reg;
        image_addr_reg <= image_addr;
        image_addr_reg0<= image_addr_reg;
    end

    // calculate rom address and read the location
    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
    four_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

    // use color map to create 8bits R, 8bits G, 8 bits B;
    eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
    eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
    eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module counter_blob_5 #(parameter WIDTH = 100, HEIGHT =100)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
    wire [18:0] image_addr;             //num of bits for 128*256 ROM
    wire [3:0] image_bits;
    reg [3:0] image_bit_reg, image_bit_reg0;
    wire [7:0] red_mapped,green_mapped,blue_mapped;
    reg [18:0] image_addr_reg, image_addr_reg0;
    // note the two clock cycle delay in pixel!

    always @ (posedge pixel_clk) begin
        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                pixel <= {red_mapped,green_mapped,blue_mapped};
        else pixel <= 0;
        image_bit_reg <= image_bits;
        image_bit_reg0 <= image_bit_reg;
        image_addr_reg <= image_addr;
        image_addr_reg0<= image_addr_reg;
    end

    // calculate rom address and read the location
    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
    five_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?
```

```
    // use color map to create 8bits R, 8bits G, 8 bits B;
    eight_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
    eight_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
    eight_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module picture_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
    wire [18:0] image_addr;             //num of bits for 128*256 ROM
    wire [3:0] image_bits;
    reg [3:0] image_bit_reg, image_bit_reg0;
    wire [7:0] red_mapped,green_mapped,blue_mapped;
    reg [18:0] image_addr_reg, image_addr_reg0;
    // note the two clock cycle delay in pixel!

    always @ (posedge pixel_clk) begin
            if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                    pixel <= {red_mapped,green_mapped,blue_mapped};
            else pixel <= 0;
            image_bit_reg <= image_bits;
            image_bit_reg0 <= image_bit_reg;
            image_addr_reg <= image_addr;
            image_addr_reg0<= image_addr_reg;
    end

    // calculate rom address and read the location
    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
    background_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?

    // use color map to create 8bits R, 8bits G, 8 bits B;
    hills_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
    hills_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
    hills_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

//module title_blob #(parameter WIDTH = 600, HEIGHT =400)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//    wire [18:0] image_addr;                   //num of bits for 128*256 ROM
//    wire [3:0] image_bits;
```

```
//    reg [3:0] image_bit_reg, image_bit_reg0;
//    wire [7:0] red_mapped,green_mapped,blue_mapped;
//    reg [18:0] image_addr_reg, image_addr_reg0;
//    // note the two clock cycle delay in pixel!
//
//    always @ (posedge pixel_clk) begin
//         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//                   pixel <= {red_mapped,green_mapped,blue_mapped};
//         else pixel <= 0;
//         image_bit_reg <= image_bits;
//         image_bit_reg0 <= image_bit_reg;
//         image_addr_reg <= image_addr;
//         image_addr_reg0<= image_addr_reg;
//    end
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    Title_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    Title_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    Title_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    Title_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule



//module calibration_blob #(parameter WIDTH = 600, HEIGHT =400)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//    wire [18:0] image_addr;                        //num of bits for 128*256 ROM
//    wire [3:0] image_bits;
//    reg [3:0] image_bit_reg, image_bit_reg0;
//    wire [7:0] red_mapped,green_mapped,blue_mapped;
//    reg [18:0] image_addr_reg, image_addr_reg0;
//    // note the two clock cycle delay in pixel!
//
//    always @ (posedge pixel_clk) begin
//         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//                   pixel <= {red_mapped,green_mapped,blue_mapped};
//         else pixel <= 0;
//         image_bit_reg <= image_bits;
```

```
//          image_bit_reg0 <= image_bit_reg;
//          image_addr_reg <= image_addr;
//          image_addr_reg0<= image_addr_reg;
//    end
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    calibration_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    calibration_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    calibration_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    calibration_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule

module turkey_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;               //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
          if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                 pixel <= {red_mapped,green_mapped,blue_mapped};
          else pixel <= 0;
          image_bit_reg <= image_bits;
          image_bit_reg0 <= image_bit_reg;
          image_addr_reg <= image_addr;
          image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   turkey_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What
is this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   turkey_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
```

```
   turkey_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   turkey_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module duck_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;              //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                 pixel <= {red_mapped,green_mapped,blue_mapped};
         else pixel <= 0;
         image_bit_reg <= image_bits;
         image_bit_reg0 <= image_bit_reg;
         image_addr_reg <= image_addr;
         image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   duck_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   duck_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   duck_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   duck_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module hit_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;              //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
```

```
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
              pixel <= {red_mapped,green_mapped,blue_mapped};
        else pixel <= 0;
        image_bit_reg <= image_bits;
        image_bit_reg0 <= image_bit_reg;
        image_addr_reg <= image_addr;
        image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   hit_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   hit_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   hit_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   hit_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule


//module hunter_blob #(parameter WIDTH = 600, HEIGHT =400)
//(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
//output reg [23:0] pixel);
//   wire [18:0] image_addr;                    //num of bits for 128*256 ROM
//   wire [3:0] image_bits;
//   reg [3:0] image_bit_reg, image_bit_reg0;
//   wire [7:0] red_mapped,green_mapped,blue_mapped;
//   reg [18:0] image_addr_reg, image_addr_reg0;
//   // note the two clock cycle delay in pixel!
//
//   always @ (posedge pixel_clk) begin
//        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
//              pixel <= {red_mapped,green_mapped,blue_mapped};
//        else pixel <= 0;
//        image_bit_reg <= image_bits;
//        image_bit_reg0 <= image_bit_reg;
//        image_addr_reg <= image_addr;
//        image_addr_reg0<= image_addr_reg;
//   end
```

```
//
//    // calculate rom address and read the location
//    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
//    hunter_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What
is this?
//
//    // use color map to create 8bits R, 8bits G, 8 bits B;
//    hunter_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
//    hunter_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
//    hunter_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
//endmodule


module beginner_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;                //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
               pixel <= {red_mapped,green_mapped,blue_mapped};
         else pixel <= 0;
         image_bit_reg <= image_bits;
         image_bit_reg0 <= image_bit_reg;
         image_addr_reg <= image_addr;
         image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   beginner_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   beginner_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   beginner_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   beginner_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule
```

```
module intermediate_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;          //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
         if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                  pixel <= {red_mapped,green_mapped,blue_mapped};
         else pixel <= 0;
         image_bit_reg <= image_bits;
         image_bit_reg0 <= image_bit_reg;
         image_addr_reg <= image_addr;
         image_addr_reg0<= image_addr_reg;
   end

   // calculate rom address and read the location
   assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
   intermediate_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?

   // use color map to create 8bits R, 8bits G, 8 bits B;
   intermediate_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
   intermediate_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
   intermediate_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module expert_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
   wire [18:0] image_addr;          //num of bits for 128*256 ROM
   wire [3:0] image_bits;
   reg [3:0] image_bit_reg, image_bit_reg0;
   wire [7:0] red_mapped,green_mapped,blue_mapped;
   reg [18:0] image_addr_reg, image_addr_reg0;
   // note the two clock cycle delay in pixel!

   always @ (posedge pixel_clk) begin
```

```
        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                pixel <= {red_mapped,green_mapped,blue_mapped};
        else pixel <= 0;
        image_bit_reg <= image_bits;
        image_bit_reg0 <= image_bit_reg;
        image_addr_reg <= image_addr;
        image_addr_reg0<= image_addr_reg;
    end

    // calculate rom address and read the location
    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
    expert_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What
is this?

    // use color map to create 8bits R, 8bits G, 8 bits B;
    expert_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
    expert_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
    expert_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule

module title1_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input [10:0] x,hcount,input [9:0] y,vcount,
output reg [23:0] pixel);
    wire [18:0] image_addr;             //num of bits for 128*256 ROM
    wire [3:0] image_bits;
    reg [3:0] image_bit_reg, image_bit_reg0;
    wire [7:0] red_mapped,green_mapped,blue_mapped;
    reg [18:0] image_addr_reg, image_addr_reg0;
    // note the two clock cycle delay in pixel!

    always @ (posedge pixel_clk) begin
        if ((hcount>= x && hcount< (x+WIDTH)) &&(vcount>= y &&vcount< (y+HEIGHT)))
                pixel <= {red_mapped,green_mapped,blue_mapped};
        else pixel <= 0;
        image_bit_reg <= image_bits;
        image_bit_reg0 <= image_bit_reg;
        image_addr_reg <= image_addr;
        image_addr_reg0<= image_addr_reg;
    end

    // calculate rom address and read the location
    assign image_addr= (hcount-x) + (vcount-y) * WIDTH;
```

```
    title1_image rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What is
this?

    // use color map to create 8bits R, 8bits G, 8 bits B;
    title1_red rcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(red_mapped));
    title1_green gcm( .addra(image_bit_reg0), .clka(pixel_clk), .douta(green_mapped));
    title1_blue bcm(.addra(image_bit_reg0), .clka(pixel_clk), .douta(blue_mapped));
endmodule
```

```
module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
           HEIGHT = 64,           // default height: 64 pixels
           COLOR = 24'hFF_FF_FF)  // default color: white
   (input [10:0] x,hcount,
           input [9:0] y,vcount,
           output reg [23:0] pixel);

   always @ * begin
           if ((hcount >= x && hcount < (x+WIDTH)) &&
     (vcount >= y && vcount < (y+HEIGHT)))
    pixel = COLOR;
           else pixel = 0;
   end
endmodule

//module target
//#(parameter RADIUS1 = 25, RADIUS2 = 45, RADIUS3 = 65) // default picture width //
default picture height
//(
//input [10:0] x, hcount,
//input [9:0] y,vcount,
//input vclock,
//output reg [23:0] pixel
//);
//
//reg [18:0] deltax1;
//reg [18:0] deltay1;
//reg [18:0] deltax2;
```

```
//reg [18:0] deltay2;
//reg [18:0] deltax3;
//reg [18:0] deltay3;
//reg [18:0] radiussquared1 = RADIUS1*RADIUS1;
//reg [18:0] radiussquared2 = RADIUS2*RADIUS2;
//reg [18:0] radiussquared3 = RADIUS3*RADIUS3;
//
//
//   alwaeltax1 <= (hcount > (x)) ? (hcount-(x)) : ((x)-hcount);
//    deltay1 <= (vcount > (y)) ? (vcount-(y)) : ((y)-vcount);
//    deltax2 <= (hcount > (x)) ? (hcount-(x)) : ((x)-hcount);
//    deltay2 <= (vcount > (y)) ? (vcount-(y)) : ((y)-vcount);
//    deltax3 <= (hcount > (x)) ? (hcount-(x)) : ((x)-hcount);
//    deltay3 <= (vcount > (y)) ? (vcount-(y)) : ((y)-vcount);
//    if(deltax1*deltax1+deltay1*deltay1 <= radiussquared1)
//    pixel <= 24'hFF_00_00;
//    else if(deltax2*deltax2+deltay2*deltay2 <= radiussquared2 )
//    pixel <= 24'h00_00_01;
//    else if(deltax3*deltax3+deltay3*deltay3 <= radiussquared3)
//    pixel <= 24'hFF_FF_FF;
//
//else pixel <= 0;
//end
//endmodule

module circle #(parameter RADIUS1 =8, RADIUS2= 10) // default picture width // default
picture height
(
input [10:0] x, hcount,
input [9:0] y,vcount,
input vclock,
output reg [23:0] pixel
);

reg [18:0] deltax1;
reg [18:0] deltay1;
reg [18:0] radiussquared1 = RADIUS1*RADIUS1;
reg [18:0] radiussquared2 = RADIUS2*RADIUS2;

always @ (posedge vclock) begin
   deltax1 <= (hcount > (x)) ? (hcount-(x)) : ((x)-hcount);
   deltay1 <= (vcount > (y)) ? (vcount-(y)) : ((y)-vcount);
```

```
   if(deltax1*deltax1+deltay1*deltay1 >= radiussquared1 & deltax1*deltax1+deltay1*deltay1
<= radiussquared2)
          pixel <= 24'hFF_80_00;
   else
          pixel <=0;
end
endmodule


module sight #(parameter RADIUS1=8)// default picture width // default picture height
(
input [10:0] x, hcount,
input [9:0] y,vcount,
input vclock,
output reg [23:0] pixel
);
   wire [23:0] crosshairs1, crosshairs2, circle_pix;
   blob #(.WIDTH(4), .HEIGHT(RADIUS1*2),.COLOR(24'hFF_80_00))
   cross1 (.x(x), .y(y-RADIUS1), .vcount(vcount), .hcount(hcount), .pixel(crosshairs1));
   blob #(.WIDTH(RADIUS1*2), .HEIGHT(4), .COLOR(24'hFF_80_00))
   cross2 (.x(x-RADIUS1+2), .y(y-2), .vcount(vcount), .hcount(hcount), .pixel(crosshairs2));

   circle ring (.x(x), .y(y), .vcount(vcount), .hcount(hcount), .vclock(vclock),
   .pixel(circle_pix));
   always @* begin
          pixel = crosshairs1+ crosshairs2+ circle_pix;
   end
endmodule

module pong_game (
  input vclock,    // 65MHz clock
  input reset,              // 1 to initialize module
  input up,       // 1 when paddle should move up
  input down,   // 1 when paddle should move down
  input [3:0] pspeed,  // puck speed in pixels/tick
  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 switch,
  input blank,             // XVGA blanking (1 means output black pixel)
   input expire,
   input stop,
```

```
    input[10:0] x,
    input[10:0] y,
    input[3:0] delay_dis,
    input fire,
    output [1:0] music,
    output start_on,
    output gun,
   output phsync,    // pong game's horizontal sync
   output pvsync,    // pong game's vertical sync
   output pblank,    // pong game's blanking
   output [23:0] pixel    // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );


//puck pixel and control

wire [23:0] puck_pixel, puck_pixel1, puck_pixel2;
reg [9:0]  paddle_y;
reg [9:0] puck_x1 = 'd500;
reg [9:0] puck_y1 = 'd400;
reg [9:0] puck_x2 = 'd50;
reg [9:0] puck_y2 = 'd200;
reg [9:0] puck_x3 = 'd500;
reg [9:0] puck_y3 = 'd400;
reg gunner=0;


reg [0:0] vel_y1='b0;
reg [0:0] vel_x1='b0;
reg [0:0] vel_y2='b1;
reg [0:0] vel_x2='b1;
reg [0:0] vel_y3='b0;
reg [0:0] vel_x3='b0;


reg hsync1;
reg vsync1;
reg [10:0] y_extend = 1170;
reg [10:0] x_extend =  1084;
reg [19:0] x_changed;
reg [19:0] y_changed;
reg [10:0] x_new;
reg [10:0] y_new;
```

```
wire [23:0] beginner;
wire [23:0] turkey;
wire [23:0] turkey1;
wire [23:0] title;
wire [23:0] background;
wire [23:0] duck;
wire [23:0] duck1;
wire [23:0] duck2;
wire [23:0] counter_pixel_first;
wire [23:0] counter_pixel_second;
wire [23:0] hit;
wire [23:0] deer;
wire [23:0] deer1;
wire [23:0] intermediate;
wire [23:0] expert;
wire [23:0] title_1;


    reg hit0, hit1;
    reg[3:0] state =0;
    reg[3:0] level_hold =0;
    reg start_holder =0;
    reg[31:0] count =0;
    reg [4:0] speed =1;
    reg [31:0] slowdownbitch;
    reg restart =0;
    reg [31:0] counter1=0;
    reg start=0;
    reg[23:0] pixel_holder;
    reg [1:0] music_reg;
    reg [10:0] holderx, holdery;
    always @(*) begin
        x_changed=(x-20)*x_extend;
        y_changed=(y-40)*y_extend;
   end
    always @(posedge vclock) begin
                        if (reset)
                                state<=0;
                        else begin
                        case(state)
```

```
0: begin
	music_reg <= 2;
	restart<=0;
	speed<=1;
	hit0<=0;
	hit1<=0;
	pixel_holder <= puck_pixel ? puck_pixel : beginner ?
beginner : title_1 == 24'hffffff ? background: title_1 ? title_1 : expert ? expert : intermediate ?
intermediate: background;
	if (~fire && slowdownbitch>24000000) begin
		gunner<=1;
		slowdownbitch<=0;
		if (640-x_changed[19:10]>240 &&
640-x_changed[19:10]<440 && y_changed[19:10]>200 && y_changed[19:10]<280 ) begin
				state <=1;
				count<=0;
		end
		else if (640-x_changed[19:10]>240 &&
640-x_changed[19:10]<440 && y_changed[19:10]>300 && y_changed[19:10]<380 ) begin
				state <=2;
				count<=0;
		end
		else if((640-x_changed[19:10])>240 &&
(640-x_changed[19:10])<440 && y_changed[19:10]>400 && y_changed[19:10]<480 ) begin
				state <=3;
				count<=0;
	end
	end
	else begin
			if (counter1>17500000) begin
				gunner<=0;
				counter1<=0;
			end
			else if (gunner) begin
				counter1<=counter1+1;
			end
			state<=0;
			slowdownbitch<=slowdownbitch+1;
	end
end

1: begin
	if (counter1>17500000) begin
```

```
                    gunner<=0;
                    counter1<=0;
                    end
                else if (gunner) begin
                    counter1<=counter1+1;
                end
                music_reg <= 2;
                restart<=0;
                level_hold <=1;
                    pixel_holder <= puck_pixel ? puck_pixel:(deer
==24'hffffff) ?
        (background): deer ? deer:
         background;
                if (~expire) begin
                    if (count < 25000000) begin
                        start <=1;
                        count<=count+1;
                    end
                    else begin
                        start <=0;
                end
                end
                if (~fire & ~expire && ~stop) begin
                    if (|deer && |puck_pixel) begin
                        //if(hit1==1) begin
                            gunner<=1;
                            state <=4;
                            count<=0;
                        //end
                        //else
                            //hit0<=1;
                    end
 //             if (|deer1 && |puck_pixel) begin
 //                 if(hit0==1) begin
 //                     state<=4;
 //                     count<=0;
 //                 end
 //                 else
 //                     hit1<=1;
 //             end
 //             else
 //                 state <=1;
                    end
```

```
                    if (expire) begin
                              state <=0;
                              count<=0;
                    end
                    end

          2: begin
                    music_reg <= 1;
                              if (counter1>17500000) begin
                                                  gunner<=0;
                                                  counter1<=0;
                              end
                              else if (gunner) begin
                                        counter1<=counter1+1;
                    end
                    restart<=0;
                    level_hold<=2;
                              pixel_holder <= puck_pixel ? puck_pixel: (turkey
==24'hffffff || turkey1 ==
                              24'hffffff) ?
                              (background): (turkey && ~hit1) ? turkey :
(turkey1 && ~hit0) ? turkey1:
                              background;
                    if (~expire) begin
                              if (count < 25000000) begin
                                        start <=1;
                                        count<=count+1;
                              end
                              else begin
                                        start <=0;
                    end
                    end
                    if (~fire & ~expire && ~stop) begin
                              if ((|turkey && |puck_pixel)) begin
                                        gunner<=1;
                                        if(hit0) begin
                                                  state <=4;
                                                  count<=0;
                                        end
                                        else
                                                  hit1<=1;
```

```
					end
					else if (|turkey1 && puck_pixel) begin
						gunner<=1;
						if(hit1) begin
							state<=4;
							count<=0;
						end
						else
							hit0<=1;
					end
					else
						state <=2;
					end

				if (expire) begin
					count<=0;
					state <= 0;
				end
				end

			3: begin
				if (counter1>17500000) begin
						gunner<=0;
						counter1<=0;
					end
				else if (gunner) begin
					counter1<=counter1+1;
				end
				music_reg <= 2;
				restart<=0;
				level_hold <=3;
					pixel_holder <= puck_pixel ? puck_pixel: (duck
==24'hffffff || duck1==24'hffffff) ?
		(background) : (duck && ~hit1) ? duck : (duck1 && ~hit0) ? duck1:
		background;
				if (~expire) begin
					if (count < 25000000) begin
						start <=1;
						count<=count+1;
					end
					else begin
						start <=0;
				end
```

```
				end
				if (~fire & ~expire && ~stop) begin
					if ((|duck && |puck_pixel)) begin
						gunner<=1;
						if(hit0) begin
							state <=4;
							count<=0;
						end
						else
							hit1<=1;
					end
					else if (|duck1 && puck_pixel) begin
						gunner<=1;
						if(hit1) begin
							state<=4;
							count<=0;
						end
						else
							hit0<=1;
					end
					else
						state <=3;
					end
				if (expire) begin
					state <=0;
					count<=0;
				end
				end
			4: begin
				restart<=1;
				hit0<=0;
				hit1<=0;
				if (counter1>17500000) begin
					gunner<=0;
					counter1<=0;
				end
				else if (gunner) begin
					counter1<=counter1+1;
				end
				pixel_holder <= hit==24'hfcfbfb ? background: hit ? hit :
background;
				if(~fire && slowdownbitch>24000000 && ~stop) begin
					hit0<=0;
```

```
                                        hit1<=0;
                                        slowdownbitch<=0;
                                        //puck_x <= 'd320;
                                        //puck_y <= 'd240;
                                        if(speed>27) begin
                                             case (level_hold)
                                                  1:
                                                       state<=2;
                                                  2:
                                                       state<=3;
                                                  3:
                                                       state<=0;
                                             endcase
                                             speed<=1;
                                        end
                                        else begin
                                             state<=level_hold;
                                             speed<=speed+2;
                                        end
                                   end
                                   else
                                        slowdownbitch<=slowdownbitch+1;
                              end


               endcase
          end
    end

assign start_on = start;
always @ (negedge vsync) begin   //puck controls
          //hsync1 <= hsync;
         //vsync1 <= vsync;
          if (reset) begin
          puck_x1 <= 'd320;
          puck_y1 <= 'd240;
          end
          else if (stop) begin
               puck_x1<=790-x_changed[19:10];
               puck_y1<=100;
          end

          if (puck_y1 >= 'd410)begin
```

```
			vel_y1 <= 'b1;
		end

		if (puck_x1 >= 'd560)begin
			vel_x1 <= 'b0;
		end

		if (puck_x1<= 70)begin
			vel_x1<='b1;

		end

		if (puck_y1<= 70)begin
			vel_y1 <='b0;

		end

		if (vel_x1&~vel_y1&~stop) begin
			puck_x1 <= puck_x1+speed;
			puck_y1 <= puck_y1+speed;
			end

		if (vel_x1&vel_y1&~stop) begin
			puck_x1<= puck_x1+speed;
			puck_y1 <= puck_y1-speed;
			end

		if (stop) begin
			puck_x1<=puck_x1;
			puck_y1<=puck_y1;
			end

		if (~vel_x1&vel_y1) begin
			puck_x1<= puck_x1-speed;
			puck_y1 <= puck_y1-speed;
			end

		if (~vel_x1&~vel_y1) begin
			puck_x1<= puck_x1-speed;
			puck_y1 <= puck_y1+speed;
			end

	end
```

```
always @ (negedge vsync) begin   //puck controls
	//hsync1 <= hsync;
	//vsync1 <= vsync;
	if (reset) begin
	puck_x2 <= 'd320;
	puck_y2<= 'd240;
	end
	else if (stop) begin
		puck_x2<=590-x_changed[19:10];
		puck_y2<=300;
	end

	if (puck_y2 >= 'd410)begin
		vel_y2 <= 'b1;
	end

	if (puck_x2 >= 'd560)begin
		vel_x2 <= 'b0;
	end

	if (puck_x2<= 70)begin
		vel_x2 <='b1;

	end

	if (puck_y2<= 70)begin
		vel_y2 <='b0;

	end

	if (vel_x2&~vel_y2&~stop) begin
		puck_x2 <= puck_x2+speed;
		puck_y2 <= puck_y2+speed;
		end

	if (vel_x2&vel_y2&~stop) begin
		puck_x2<= puck_x2+speed;
		puck_y2 <= puck_y2-speed;
		end

	if (stop) begin
		puck_x2<=puck_x2;
		puck_y2<=puck_y2;
```

```
			end

		if (~vel_x2&vel_y2) begin
			puck_x2<= puck_x2-speed;
			puck_y2 <= puck_y2-speed;
			end

		if (~vel_x2&~vel_y2) begin
			puck_x2<= puck_x2-speed;
			puck_y2 <= puck_y2+speed;
			end

end
always @ (negedge vsync) begin   //puck controls
		//hsync1 <= hsync;
		//vsync1 <= vsync;
		if (reset) begin
		puck_x3 <= 'd320;
		puck_y3 <= 'd240;
		end
		else if (restart) begin
			puck_x3<=790-x_changed[19:10];
			puck_y3<=100;
		end

		if (puck_y3 >= 'd410)begin
			vel_y3 <= 'b1;
		end

		if (puck_x3 >= 'd560)begin
			vel_x3 <= 'b0;
		end

		if (puck_x3<= 30)begin
			vel_x3 <='b1;

		end

		if (puck_y3<= 30)begin
			vel_y3 <='b0;

		end
```

```
        if (vel_x3&~vel_y3&~stop) begin
                puck_x3 <= puck_x3+speed;
                puck_y3 <= puck_y3+speed;
                end

        if (vel_x3&vel_y3&~stop) begin
                puck_x3<= puck_x3+speed;
                puck_y3 <= puck_y3-speed;
                end

        if (stop) begin
                puck_x3<=puck_x3;
                puck_y3<=puck_y3;
                end

       if (~vel_x3&vel_y3) begin
                puck_x3<= puck_x3-speed;
                puck_y3 <= puck_y3-speed;
                end

        if (~vel_x3&~vel_y3) begin
                puck_x3<= puck_x3-speed;
                puck_y3 <= puck_y3+speed;
                end

end

always @(posedge vclock) begin
   if (~stop) begin
        holderx <= 640-x_changed[19:10];
        holdery <= y_changed[19:10];
   end
   else if (stop && state != 0) begin
        holderx <= holderx;
        holdery <= holdery;
   end
end

wire [23:0] blended_pixel;
  blob #(.WIDTH(64), .HEIGHT(64), .COLOR(24'hFE_7F_7F)) //displays blended
 blended(.x(puck_x_start+puck_x),.y(puck_y_start+puck_y),.hcount(hcount),.vcount(vcount),
   .pixel(blended_pixel));
```

```
  sight //displays puck
  puck(.x(holderx),.y(holdery),.hcount(hcount),.vcount(vcount),
    .vclock(vclock), .pixel(puck_pixel));


    picture_blob #(.HEIGHT(480), .WIDTH(640))
    photo(.x(0), .y(0), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(background));


//    title_blob #(.HEIGHT(600), .WIDTH(800))
//    start_screen(.x(0), .y(0), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(title));



//wire [23:0] calibration;
//    calibration_blob #(.HEIGHT(600), .WIDTH(800))
//    calibrate(.x(0), .y(0), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(calibration));


    deer_blob #(.HEIGHT(100), .WIDTH(100))
    deer_1(.x(puck_x1), .y(puck_y1), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(deer));


    turkey_blob #(.HEIGHT(50), .WIDTH(50))
    turkey_1(.x(puck_x1), .y(puck_y1), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(turkey));
    turkey_blob #(.HEIGHT(50), .WIDTH(50))
    turkey_2(.x(puck_x2), .y(puck_y2), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(turkey1));

    duck_blob #(.HEIGHT(25), .WIDTH(25))
    duck_1(.x(puck_x1), .y(puck_y1), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(duck));
    duck_blob #(.HEIGHT(25), .WIDTH(25))
    duck_2(.x(puck_x2), .y(puck_y2), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(duck1));
    duck_blob #(.HEIGHT(25), .WIDTH(25))
    duck_3(.x(puck_x3), .y(puck_y3), .hcount(hcount), .vcount(vcount),
```

```
    .pixel_clk(vclock), .pixel(duck2));



    hit_blob #(.HEIGHT(50), .WIDTH(100))
    hit_1(.x(270), .y(200), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(hit));

//used in the case of double digits
reg [2:0] counter_count=0;    // perhaps a way to designate specific counters?
                                                                                    // probably
complement this method with something in the main file
                                                                                    // that establishes
the start correct start location for each number
                                                                                    // so that we can
properly show double digits.


//wire [23:0] hunter;
    //hunter_blob #(.HEIGHT(200), .WIDTH(200))
    //hunt(.x(0), .y(150), .hcount(hcount), .vcount(vcount),
    //.pixel_clk(vclock), .pixel(hunter));

    beginner_blob #(.HEIGHT(75), .WIDTH(200))
    beg(.x(210), .y(200), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(beginner));


    intermediate_blob #(.HEIGHT(75), .WIDTH(200))
    interm(.x(210), .y(300), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(intermediate));


    expert_blob #(.HEIGHT(75), .WIDTH(200))
    exper(.x(210), .y(400), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(expert));

    title1_blob #(.HEIGHT(100), .WIDTH(500))
    title1(.x(70), .y(50), .hcount(hcount), .vcount(vcount),
    .pixel_clk(vclock), .pixel(title_1));

//wire [23:0] counter_pixel_0;
//    counter_blob_0 #(.HEIGHT(75), .WIDTH(75))
```

```
//    counterpix0(.x(400), .y(300), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_0));
//
//wire [23:0] counter_pixel_1;
//    counter_blob_1 #(.HEIGHT(75), .WIDTH(75))
//    counterpix1(.x(400), .y(300), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_1));
//
//wire [23:0] counter_pixel_2;
//    counter_blob_2 #(.HEIGHT(75), .WIDTH(75))
//    counterpix2(.x(400), .y(300), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_2));

//wire [23:0] counter_pixel_3;
//    counter_blob_3 #(.HEIGHT(75), .WIDTH(75))
//    counterpix3(.x(400), .y(300), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_3));
//
//wire [23:0] counter_pixel_4;
//    counter_blob_4 #(.HEIGHT(75), .WIDTH(75))
//    counterpix4(.x(400), .y(60), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_4));
//
//wire [23:0] counter_pixel_5;
//    counter_blob_5 #(.HEIGHT(75), .WIDTH(75))
//    counterpix5(.x(400), .y(60), .hcount(hcount), .vcount(vcount),
//    .pixel_clk(vclock), .pixel(counter_pixel_5));
//     assign counter_pixel_first = delay_dis == 5 ? counter_pixel_5  : delay_dis == 4 ?
counter_pixel_4 :counter_pixel_3;
    //delay_dis==2 ? counter_pixel_2
    //: delay_dis==1 ? counter_pixel_1 : delay_dis==0 ? counter_pixel_0: counter_pixel_first;
    //assign counter_pixel_second = ~counter_count? counter_pixel_5 :
counter_pixel_second;
    reg [23:0] holder0, holder1;
//     always @(posedge vclock) begin
//         holder0 <= puck_pixel ? puck_pixel:(deer ==24'hffffff)?
//     (background +counter_pixel_first): deer ? deer :
//     (counter_pixel_first ==24'hffffff) ? background:counter_pixel_first
//     ?counter_pixel_first:
//     background;
//         holder1 <= puck_pixel ? puck_pixel : beginner ? beginner : title_1 == 24'hffffff ?
background: title_1 ? title_1 : expert ? expert : intermediate ? intermediate: background;
//     end
```

```
    assign phsync = hsync;
         assign pvsync = vsync;
         assign pblank = blank;
    assign pixel = pixel_holder;
    assign music = music_reg;
    assign gun = gunner;


   //assign pixel = puck_pixel? puck_pixel :(|puck_pixel || |center_pixel) ? puck_pixel
+center_pixel:
    //background;


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 >= 840) ? (hcount - 840) : (hcount + 8);
  wire [9:0] vcount_f = (hcount >= 840) ? ((vcount == 624) ? 0 : vcount + 1) : vcount;

  wire [18:0]      vram_addr = { vcount_f, hcount_f[9:1]};

  wire hc2 = hcount[0];
  reg [23:0]      vr_pixel;
  reg [35:0]      vr_data_latched;
  reg [35:0]      last_vr_data;

  always @(posedge clk)
        last_vr_data <= (hc2==1'd1) ? vr_data_latched : last_vr_data;

  always @(posedge clk)
        vr_data_latched <= (hc2==1'd0) ? vram_read_data : vr_data_latched;

  always @(*)            // each 36-bit word from RAM is decoded to 2 bytes
        case (hc2)
        1'd1: vr_pixel = {last_vr_data[17:12],2'b00, last_vr_data[11:6], 2'b00, last_vr_data[5:0],
2'b00};
        1'd0: vr_pixel = {last_vr_data[35:30], 2'b00, last_vr_data[29:24], 2'b00,
last_vr_data[23:18], 2'b00};
        endcase
```

```
endmodule // vram_display

///////////////////////////////////////////////////////////////////////////////
// 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
//
///////////////////////////////////////////////////////////////////////////////
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
```

Appendix C: Sound Module

```
`default_nettype none

///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (
  input wire reset, clock, noisy,
  output reg clean
);
  reg [18:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
```

```
    else if (noisy != new) begin
      // noisy input changed, restart the .01 sec clock
      new <= noisy;
      count <= 0;
    end
    else if (count == 270000)
      // noisy input stable for .01 secs, pass it along!
      clean <= new;
    else
      // waiting for .01 sec to pass
      count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// bi-directional monaural interface to AC97
//
///////////////////////////////////////////////////////////////////////////////

module lab5audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [7:0] audio_in_data,
  input wire [7:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);

  wire [7:0] command_address;
  wire [15:0] command_data;
  wire command_valid;
  wire [19:0] left_in_data, right_in_data;
  wire [19:0] left_out_data, right_out_data;

  // wait a little before enabling the AC97 codec
  reg [9:0] reset_count;
  always @(posedge clock_27mhz) begin
```

```
    if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
    end else if (reset_count == 1023)
      audio_reset_b = 1'b1;
    else
      reset_count = reset_count+1;
  end

  wire ac97_ready;
  ac97 ac97(.ready(ac97_ready),
            .command_address(command_address),
            .command_data(command_data),
            .command_valid(command_valid),
            .left_data(left_out_data), .left_valid(1'b1),
            .right_data(right_out_data), .right_valid(1'b1),
            .left_in_data(left_in_data), .right_in_data(right_in_data),
            .ac97_sdata_out(ac97_sdata_out),
            .ac97_sdata_in(ac97_sdata_in),
            .ac97_synch(ac97_synch),
            .ac97_bit_clock(ac97_bit_clock));

  // ready: one cycle pulse synchronous with clock_27mhz
  reg [2:0] ready_sync;
  always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
  assign ready = ready_sync[1] & ~ready_sync[2];

  reg [7:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:12];
  assign left_out_data = {out_data, 12'b000000000000};
  assign right_out_data = left_out_data;

  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule
```

```
// assemble/disassemble AC97 serial frames
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
  reg [7:0] bit_count;

  reg [19:0] l_cmd_addr;
  reg [19:0] l_cmd_data;
  reg [19:0] l_left_data, l_right_data;
  reg l_cmd_v, l_left_v, l_right_v;

  initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";

    bit_count <= 8'h00;
    // synthesis attribute init of bit_count is "0000";
    l_cmd_v <= 1'b0;
    // synthesis attribute init of l_cmd_v is "0";
    l_left_v <= 1'b0;
    // synthesis attribute init of l_left_v is "0";
    l_right_v <= 1'b0;
    // synthesis attribute init of l_right_v is "0";

    left_in_data <= 20'h00000;
    // synthesis attribute init of left_in_data is "00000";
    right_in_data <= 20'h00000;
```

```
  // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
  // Generate the sync signal
  if (bit_count == 255)
    ac97_synch <= 1'b1;
  if (bit_count == 15)
    ac97_synch <= 1'b0;

  // Generate the ready signal
  if (bit_count == 128)
    ready <= 1'b1;
  if (bit_count == 2)
    ready <= 1'b0;

  // Latch user data at the end of each frame. This ensures that the
  // first frame after reset will be empty.
  if (bit_count == 255) begin
    l_cmd_addr <= {command_address, 12'h000};
    l_cmd_data <= {command_data, 4'h0};
    l_cmd_v <= command_valid;
    l_left_data <= left_data;
    l_left_v <= left_valid;
    l_right_data <= right_data;
    l_right_v <= right_valid;
  end

  if ((bit_count >= 0) && (bit_count <= 15))
    // Slot 0: Tags
    case (bit_count[3:0])
      4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
      4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
      4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
      4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
      4'h4: ac97_sdata_out <= l_right_v; // Right data valid
      default: ac97_sdata_out <= 1'b0;
    endcase
  else if ((bit_count >= 16) && (bit_count <= 35))
    // Slot 1: Command address (8-bits, left justified)
    ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
  else if ((bit_count >= 36) && (bit_count <= 55))
    // Slot 2: Command data (16-bits, left justified)
```

```
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
    else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;
  end // always @ (posedge ac97_bit_clock)

  always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
  end
endmodule

// issue initialization commands to AC97
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
  input wire [4:0] volume,
  input wire [2:0] source
);
  reg [23:0] command;

  reg [3:0] state;
  initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
```

```
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
  end

  assign command_address = command[23:16];
  assign command_data = command[15:0];

  wire [4:0] vol;
  assign vol = 31-volume;  // convert to attenuation

  always @(posedge clock) begin
    if (ready) state <= state+1;

    case (state)
      4'h0: // Read ID
        begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
        end
      4'h1: // Read ID
        command <= 24'h80_0000;
      4'h3: // headphone volume
        command <= { 8'h04, 3'b000, vol, 3'b000, vol };
      4'h5: // PCM volume
        command <= 24'h18_0808;
      4'h6: // Record source select
        command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
      4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
      4'h9: // set +20db mic gain
        command <= 24'h0E_8048;
      4'hA: // Set beep volume
        command <= 24'h0A_0000;
      4'hB: // PCM out bypass mix1
        command <= 24'h20_8000;
      default:
        command <= 24'h80_0000;
    endcase // case(state)
  end // always @ (posedge clock)
endmodule // ac97commands


///////////////////////////////////////////////////////////////////////////////
```

```
////
//// 6.111 FPGA Labkit -- Template Toplevel Module
////
//// For Labkit Revision 004
//// Created: October 31, 2004, from revision 003 file
//// Author: Nathan Ickes, 6.111 staff
////
///////////////////////////////////////////////////////////////////////////////

module lab5   (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;
//lab5 assign audio_reset_b = 1'b0;
//lab5 assign ac97_synch = 1'b0;
//lab5 assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 10'h0;
assign vga_out_green = 10'h0;
assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// 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'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
```

```
assign tv_in_clock = 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
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
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;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// 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
//lab5 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
//lab5 assign analyzer1_data = 16'h0;
//lab5 assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
//lab5 assign analyzer3_data = 16'h0;
//lab5 assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;
```

```
//   wire [7:0] from_ac97_data, to_ac97_data;
//   wire ready;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   //
   ////////////////////////////////////////////////////////////////////////////
   wire reset;
   SRL16 #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                                     .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),.clean(vup));
   debounce bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),.clean(vdown));
   reg [4:0] 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

   // AC97 driver
   lab5audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);

   // push ENTER button to record, release to playback
   wire playback;
   debounce benter(.reset(reset),.clock(clock_27mhz),.noisy(button_enter),.clean(playback));
```

```
  // switch 0 up for filtering, down for no filtering
  wire filter;
  debounce sw0(.reset(reset),.clock(clock_27mhz),.noisy(switch[0]),.clean(filter));

  // light up LEDs when recording, show volume during playback.
  // led is active low
  assign led = playback ? ~{filter,2'b00, volume} : ~{filter,7'hFF};
	reg start=1;
	wire [7:0] background, turkey, gunner;
	intermediate_blob go(.pixel_clk(clock_27mhz), .start(user1[0]), .ready(ready),
.out(background));
  // record module
	gobble_blob go1(.pixel_clk(clock_27mhz), .start(user1[1]), .ready(ready), .out(turkey));
	gun_blob go2(.pixel_clk(clock_27mhz), .start(user1[2]), .ready(ready), .out(gunner));
	assign to_ac97_data = user1[2] ? gunner : user1[0] ? background : user1[1] ? turkey :
0;
  // output useful things to the logic analyzer connectors
  assign analyzer1_clock = ac97_bit_clock;
  assign analyzer1_data[0] = audio_reset_b;
  assign analyzer1_data[1] = ac97_sdata_out;
  assign analyzer1_data[2] = ac97_sdata_in;
  assign analyzer1_data[3] = ac97_synch;
  assign analyzer1_data[15:4] = 0;

  assign analyzer3_clock = ready;
  assign analyzer3_data = {from_ac97_data, to_ac97_data};
endmodule



///////////////////////////////////////////////////////////////////////////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
///////////////////////////////////////////////////////////////////////////////

module mybram #(parameter LOGSIZE=14, WIDTH=1)
	      (input wire [LOGSIZE-1:0] addr,
	       input wire clk,
	       input wire [WIDTH-1:0] din,
```

```
            output reg [WIDTH-1:0] dout,
            input wire we);
  // let the tools infer the right number of BRAMs
  (* ram_style = "block" *)
  reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
  always @(posedge clk) begin
    if (we) mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Coefficients for a 31-tap low-pass FIR filter with Wn=.125 (eg, 3kHz for a
// 48kHz sample rate).  Since we're doing integer arithmetic, we've scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,.125)*1024)
//
///////////////////////////////////////////////////////////////////////////////

module intermediate_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input ready, input start,
output [7:0] out);                          //num of bits for 128*256 ROM
        wire [15:0] image_bits;
        reg [7:0] image_bit_reg, image_bit_reg0;
        reg [17:0] image_addr_reg=0;
        reg [17:0] image_addr_reg0;
        reg written=0;
        reg [17:0] done = 110000;
        // note the two clock cycle delay in audio out!

        always @ (posedge ready) begin
                if ((start && ~written && image_addr_reg<109999)) begin
                        image_addr_reg<= image_addr_reg+1;
                        written<=1;
                end
                else if (start && ~written && image_addr_reg==109999) begin
                        image_addr_reg<=0;
                        written<=1;
                end
                else begin
                        written<=0;
```

```
            end
            image_bit_reg <= image_bits[15:8];
            image_bit_reg0 <= image_bit_reg;
            image_addr_reg0<= image_addr_reg;
        end

        // calculate rom address and read the location
        turkey_sound rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?
        assign out= image_bit_reg0;
        // use color map to create 8bits R, 8bits G, 8 bits B;
endmodule

module gun_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input ready, input start,
output [7:0] out);                    //num of bits for 128*256 ROM
        wire [15:0] image_bits;
        reg [7:0] image_bit_reg, image_bit_reg0;
        reg [17:0] image_addr_reg=0;
        reg [17:0] image_addr_reg0;
        reg written=0;
        reg [17:0] done = 22000;
        // note the two clock cycle delay in audio out!

        always @ (posedge ready) begin
            if ((start && ~written && image_addr_reg<21999)) begin
                image_addr_reg<= image_addr_reg+1;
                written<=1;
            end
            else if (start && ~written && image_addr_reg==21999) begin
                image_addr_reg<=0;
                written<=1;
            end
            else begin
                written<=0;
            end
            image_bit_reg <= image_bits[15:8];
            image_bit_reg0 <= image_bit_reg;
            image_addr_reg0<= image_addr_reg;
        end

        // calculate rom address and read the location
```

```
	gunsound rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));  //What
is this?
	assign out= image_bit_reg0;
	// use color map to create 8bits R, 8bits G, 8 bits B;
endmodule

module gobble_blob #(parameter WIDTH = 600, HEIGHT =400)
(input pixel_clk, input ready, input start,
output [7:0] out);                              //num of bits for 128*256 ROM
	wire [15:0] image_bits;
	reg [7:0] image_bit_reg, image_bit_reg0;
	reg [17:0] image_addr_reg=0;
	reg [17:0] image_addr_reg0;
	reg written=0;
	reg [17:0] done = 110000;
	// note the two clock cycle delay in audio out!

	always @ (posedge ready) begin
		if ((start && ~written && image_addr_reg<109999)) begin
			image_addr_reg<= image_addr_reg+1;
			written<=1;
		end
		else if (start && ~written && image_addr_reg==109999) begin
			image_addr_reg<=0;
			written<=1;
		end
		else begin
			written<=0;
		end
		image_bit_reg <= image_bits[15:8];
		image_bit_reg0 <= image_bit_reg;
		image_addr_reg0<= image_addr_reg;
	end

	// calculate rom address and read the location
	turkeylevel rom1(.addra(image_addr_reg0),.clka(pixel_clk),.douta(image_bits));
//What is this?
	assign out= image_bit_reg0;
	// use color map to create 8bits R, 8bits G, 8 bits B;
endmodule
```

Appendix D: Gyro Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:28:34 11/13/2014
// Design Name:
// Module Name:    aim_calc
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module aim_calc(
        input signed[5:0] angle ,
    input clock,
        input[2:0] level,
        output signed[11:0] new_distance
        );
    reg signed[25:0] new_point = 0;
    reg signed[25:0] new_point_shifted=0;
    wire signed[16:0] tan;
    reg signed[16:0] tanreg;
    lookup_table tabl(.clk(clock), .angle(angle), .tan(tan));
    always @*
         new_point_shifted=new_point>>>13;
    always @ (posedge clock) begin
                tanreg<=tan;
                case(level)
                        1: new_point<= tanreg*100;

                        default: new_point<=new_point;
                endcase
   end


   assign new_distance = new_point_shifted[11:0];

endmodule

 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:06:43 11/12/2014
// Design Name:
// Module Name:    calculator
// Project Name:
// Target Devices:
// Tool versions:
// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module calculator(
        input signed[7:0] data,
    input start,
        output signed[5:0] angle
        );
    reg signed[7:0] hold =0;
    reg signed[9:0] dis =0;
    reg signed[9:0] datashift;

   // always @(*) begin
        //datashift= data>>>5;
   //end
    always @ (negedge start) begin
    hold <= data;
        if (hold != data)begin
                if(dis+data>=30)
                        dis<=30;
                else if (dis+data<=-30)
                        dis<=-30;
                else
                        dis <= dis + data;
        end
        else
                dis<=dis;
   end

    assign angle = dis;

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
```

```
// Create Date:         18:05:18 11/13/2014
// Design Name:
// Module Name:         data_divider
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module data_divider(
        input oldclock,
        output quarter_clock
        );

    reg[31:0] count = 0;
    reg holder =0;

    always @(posedge oldclock) begin         //counter goes high every 75microsec
         if (count==1012) begin
                 count <=0;
                 holder <=~holder;
                 end
         else begin
                 count <= count+1;
         end
   end
   assign clock=holder;

endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:         17:17:27 11/13/2014
// Design Name:
```

```
// Module Name:    data_interp
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module data_interp(

        input data,
    input clock,
    output start,
        output signed[7:0] val_out);

    reg signed[7:0] val = 0;
    reg[3:0] bits = 0;
    reg[5:0] count =0;
    reg[127:0] val_total=0;
    reg signed[7:0] val1, val2, val3, val4, val5, val6, val7,
    val8, val9, val10, val11, val12, val13, val14, val15, val16;

    reg signed[12:0] val_shift=0;
    reg reading;
    always @* begin
        val1 = val_total[127:120];
        val2 = val_total[119:112];
        val3 = val_total[111:104];
        val4 = val_total[103:96];
        val5 = val_total[95:88];
        val6 = val_total[87:80];
        val7 = val_total[79:72];
        val8 = val_total[71:64];
        val9 = val_total[63:56];
        val10 = val_total[55:48];
        val11 = val_total[47:40];
        val12 = val_total[39:32];
        val13 = val_total[31:24];
```

```
        val14 = val_total[23:16];
        val15 = val_total[15:8];
        val16 = val_total[7:0];
    end
        always @ (posedge clock) begin

                if (data & bits<8) begin     //when the signal is high, time how long it is high
                count <=count+1;
                                reading <=1;
                            end

                            if (~data & count>2 & bits<30) begin  //if it is a 1 add it to the
reg
                                if (count>6 & count<13) begin
                                        bits <= bits+1;
                                        val[6:0]<= val[7:1];
                                        val[7]<= 1;
                                        count <=0;
                                    end
                                else if (count>2 & count<6) begin //if it is a 0 add it to
the reg
                                        bits <= bits+1;
                                        val[6:0]<= val[7:1];
                                        val[7]<= 0;
                                        count <=0;
                                    end
                                    else if (count>15) begin
                                        count <=0;
                                    end

                                end
                            if (bits==8) begin
                                bits<=0;
                                val_total[127:8]<=val_total[120:0];
                                val_total[7:0] <= val;
                                reading <=0;
                            end
                        val_shift <=
val1+val2+val3+val4+val5+val6+val7+val8+val9+val10+val11
                        +val12+val13+val14+val15+val16;
        end

    assign val_out = (bits==8)? (val_shift[12:5]): val_out;
```

```
    assign start = reading;

endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:01:01 11/12/2014
// Design Name:
// Module Name:    divider
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module gyro_divider(
        input oldclock,
        output clock
        );

     reg[31:0] count = 0;
     reg holder =0;

     always @(posedge oldclock) begin        //counter goes high every 75microsec
          if (count==1012) begin
               count <=0;
               holder <=~holder;
               end
          else begin
               count <= count+1;
          end
    end
    assign clock=holder;
```

```
endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:51:09 11/19/2014
// Design Name:
// Module Name:    lookup_table
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module lookup_table(
    input clk,
        input signed[5:0] angle,
        output signed[16:0] tan
        );
    reg signed[16:0] out;
    reg signed[5:0] anglechanged;
    always @(*) begin
        if (angle[5] ==1)
                anglechanged=angle*(-1);
        else
                anglechanged=angle;
   end
    always @(clk) begin
        case(anglechanged)
                0: out<=0;
                1: out<=175;
                2: out<=349;
                3: out<=524;
                4: out<=699;
                5: out<=875;
```

```
                6: out<= 1051;
                7: out<=1228;
                8: out<=1405;
                9: out<= 1584;
                10: out<= 1763;
                11: out<= 1943;
                12: out<= 2126;
                13: out<=2309;
                14: out<=2493;
                15: out<=2679;
                16: out<=2867;
                17: out<=3057;
                18: out<=3249;
                19: out<=3443;
                20: out<=3639;
                21: out<=3839;
                22: out<= 4040;
                23: out<= 4245;
                24: out<= 4452;
                25: out<= 4663;
                26: out<= 4817;
                27: out<= 5095;
                28: out<= 5317;
                29: out<= 5543;
                30: out<= 5773;
                default: out<=out;
        endcase
    end
    assign tan = (angle[5]==1)?(out*-1):out;
endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:        13:37:13 11/12/2014
// Design Name:
// Module Name:        receiver
// Project Name:
// Target Devices:
// Tool versions:
// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module receiver(
        input data,
    input clock,
        output[7:0] val_out)

    reg[7:0] val = 0;
    reg bits = 0;
    reg[3:0] count =0;

        always @ (posedge clock) begin

                if (signal & bits<8) begin     //when the signal is high, time how long it is high
                count <=count+1;
                                end

                                if (~signal & count>5 & bits<8) begin  //if it is a 1 add it to the
reg
                                        if (count>14 & count<25) begin
                                                bits <= bits+1;
                                                com[6:0]<= com[7:1];
                                                val[7]<= 1;
                                                count <=0;
                                            end
                                        else if (count>6 & count<12) begin //if it is a 0 add it to
the reg
                                                bits <= bits+1;
                                                val[6:0]<= val[7:1];
                                                val[7]<= 0;
                                                count <=0;
                                            end

                                    end
                                if (bits==8) begin
                                    bits<=0;
                                end
```

```
        end
    assign val_out = (bits==8? val: val_out);

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//


module debounce (input reset, clock, noisy,
                output reg clean);

  reg [19:0] count;
  reg new;

  always @(posedge clock)
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 650000) clean <= new;
        else count <= count+1;

endmodule
///////////////////////////////////////////////////////////////////////////////
//
// 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 fe edback 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)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//                    and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 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 labkit (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

// VGA Output
//assign vga_out_red = 8'h0;
//assign vga_out_green = 8'h0;
//assign vga_out_blue = 8'h0;
//assign vga_out_sync_b = 1'b1;
//assign vga_out_blank_b = 1'b1;
//assign vga_out_pixel_clock = 1'b0;
//assign vga_out_hsync = 1'b0;
//assign vga_out_vsync = 1'b0;

// 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'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 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
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
```

```
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
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;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// 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
  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;

////////////////////////////////////////////////////////////////////////////
 //
 // Reset Generation
 //
 // A shift register primitive is used to generate an active-high reset
 // signal that remains high for 16 clock cycles after configuration finishes
 // and the FPGA's internal clocks begin toggling.
 //
 ////////////////////////////////////////////////////////////////////////////

 ////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
//
// lab3 : a simple pong game
//
////////////////////////////////////////////////////////////////////////////////

// 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));

// power-on reset generation
wire power_on_reset;       // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .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(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
assign reset = user_reset | power_on_reset;

// UP and DOWN buttons for pong paddle
wire up,down;
debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(up));
debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(down));
 wire signed[7:0] x_value, y_value;
 wire clock;
 wire start_x, start_y;
 wire signed[5:0] angle_x, angle_y;
 wire signed[11:0] x_point, y_point;
 reg [2:0] level=1;
// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
```

```
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
        .hsync(hsync),.vsync(vsync),.blank(blank));

// feed XVGA signals to user's pong game
wire [23:0] pixel;
wire phsync,pvsync,pblank;
gyro_test pg(.x_new(x_point), .y_new(y_point), .vclock(clock_65mhz),.reset(reset),
         .hcount(hcount),.vcount(vcount),
                   .hsync(hsync),.vsync(vsync),.blank(blank),
         .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel));




 divider newClock(.oldclock(clock_27mhz), .clock(clock));

 data_interp data_read (.data(user3[0]), .clock(clock), .start(start_x), .val_out(x_value));
 data_interp data_read_two(.data(user3[1]), .clock(clock),.start(start_y), .val_out(y_value));

 calculator calc (.data(x_value), .start(start_x), .angle(angle_x));
 calculator calc1(.data(y_value), .start(start_y), .angle(angle_y));
 aim_calc point_x (.angle(angle_x), .clock(clock),
  .level(level), .new_distance(x_point));

  aim_calc point_y(.angle(angle_y), .clock(clock),
  .level(level), .new_distance(y_point));

 wire[63:0] poopy = {x_value, y_value,{angle_x[5],angle_x[5],
angle_x[5]},angle_x,{angle_y[5],angle_y[5], angle_y[5]}, angle_y, 31'd0};

 display_16hex hex(.reset(reset), .clock_27mhz(clock_27mhz), .data(poopy) ,
         .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
         .disp_ce_b(disp_ce_b), .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));

        //switch[1:0] selects which video generator to use:
//  00: user's pong game
//  01: 1 pixel outline of active video area (adjust screen controls)
//  10: color bars
reg [23:0] rgb;
wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

reg b,hs,vs;
always @(posedge clock_65mhz) begin
```

```
        if (switch[1:0] == 2'b01) begin
  // 1 pixel outline of visible area (white)
                hs <= hsync;
                vs <= vsync;
                b <= blank;
                rgb <= {24{border}};
        end
         else if (switch[1:0] == 2'b10) begin
  // color bars
                hs <= hsync;
                vs <= vsync;
                b <= blank;
                rgb <= {{8{hcount[8]}}, {8{hcount[7]}}, {8{hcount[6]}}} ;
        end
         else begin
        // default: pong
                hs <= phsync;
                vs <= pvsync;
                b <= pblank;
                rgb <= pixel;
        end
  end

  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
  assign vga_out_red = rgb[23:16];
  assign vga_out_green = rgb[15:8];
  assign vga_out_blue = rgb[7:0];
  assign vga_out_sync_b = 1'b1;     // not used
  assign vga_out_blank_b = ~b;
  assign vga_out_pixel_clock = ~clock_65mhz;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;

  //assign led = ~{3'b000,up,down,reset,switch[1:0]};


endmodule

////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
```

```
//
//////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
        output reg [10:0] hcount,    // pixel number on current line
        output reg [9:0] vcount,   // line number
        output reg vsync,hsync,blank);

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  reg hblank,vblank;
  wire hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount == 1023);
  assign hsyncon = (hcount == 1047);
  assign hsyncoff = (hcount == 1183);
  assign hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount == 767);
  assign vsyncon = hreset & (vcount == 776);
  assign vsyncoff = hreset & (vcount == 782);
  assign vreset = hreset & (vcount == 805);

  // 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
////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////
module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
          HEIGHT = 64,           // default height: 64 pixels
          COLOR = 24'hFF_FF_FF)  // default color: white
   (input [10:0] x,hcount,
          input [9:0] y,vcount,
          output reg [23:0] pixel);

   always @ * begin
          if ((hcount >= x && hcount < (x+WIDTH)) &&
     (vcount >= y && vcount < (y+HEIGHT)))
    pixel = COLOR;
          else pixel = 0;
   end
endmodule


module gyro_test (
    input[11:0] x_new,
    input[11:0] y_new,
   input vclock,    // 65MHz clock
   input reset,             // 1 to initialize module
   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 phsync,    // pong game's horizontal sync
   output pvsync,    // pong game's vertical sync
   output pblank,    // pong game's blanking
   output [23:0] pixel    // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );



    reg[10:0] x_start = 524;
    reg[10:0] y_start= 368;
    wire[23:0] puck_pixel;
```

```
  blob #(.WIDTH(64), .HEIGHT(64), .COLOR(24'hFF_FF_FF)) //displays puck
 puck(.x(x_start+x_new),.y(y_start+y_new),.hcount(hcount),.vcount(vcount),
   .pixel(puck_pixel));



    assign phsync = hsync;
          assign pvsync = vsync;
          assign pblank = blank;
    assign pixel = puck_pixel;



endmodule
```