Gesture Based Programming
6.111 Final Project

Alex Hornstein

Abstract:
This project is an FPGA-based system that can control the motion of a robot using gestures. A vision system based on a video camera tracks a colored object and records it x and y positions at various times, and then a robot on the ground follows the path that the object traced out.

Table of Contents

List of Figures

Introduction:

One of the greatest obstacles in human/machine interaction is the communication barrier. We, as humans, speak a different language from machines. In the past, we have overcome this barrier by inventing language structures such as programming languages, which allow us to speak in a finite vocabulary that is easily translated into machine language. The problem with this is that it requires anyone who wants to communicate with computers to learn and be familiar with programming languages and concepts. We compromise by having programmers write user interfaces intended for most users, so they can use the computer without 'speaking it's language.' This allows people to use a computer, but only to the level of complexity that the programmer designs the interface, and so there are some intrinsic limitations in this system.

This project will create a new language structure that is more intuitive for humans to use. Users will use gestures to describe a path for a robot to take, as well as conditional reactions for the robot(i.e. go straight. If you hit a wall, turn left. Otherwise, keep on going straight). An FPGA will interpret these gestures using a video camera, and then direct a physical robot according to the user's instructions.

In this project, an analog color camera is pointed at a user. The camera will output a NTSC composite signal to the FPGA, which will interface to it via the onboard ADV7185 video decoder. The FPGA will track the user's gestures by tracking a finger, which will be uniquely colored, and store the set of (x,y,t) triplets in RAM. It will also

identify gestures signifying conditionals, and then compile the user's traced paths into a sort of program, which it will then ``run'' on a two-wheeled robot.

The robot itself has two drive wheels and two balancing castors. Position feedback is given by an optical mouse mounted on the robot slightly off the floor, communicating with the FPGA via the PS/2 protocol. Power is supplied by a tether, and the robot has a brushed motor amplifier onboard for each motor. Each motor will only require two wires to control.

This project is an implementation of a new concept for human/computer interaction. Using this gesture language, someone could easily create a robot with behavior such as wall-following in a few minutes, rather than learning a new set of programming commands and then writing equivalent code for the robot. By implementing it in harware, we can have a fairly sophisticated system running on a single FPGA and a few interface chips, rather than on a full PC. This could lead to a small, cheap system that allows intuitive gesture-based programming, that canbe added on to existing robot systems.

Design:

There are two major components to this system: The video interface, and the robot control. Block diagrams of both systems are shown below.

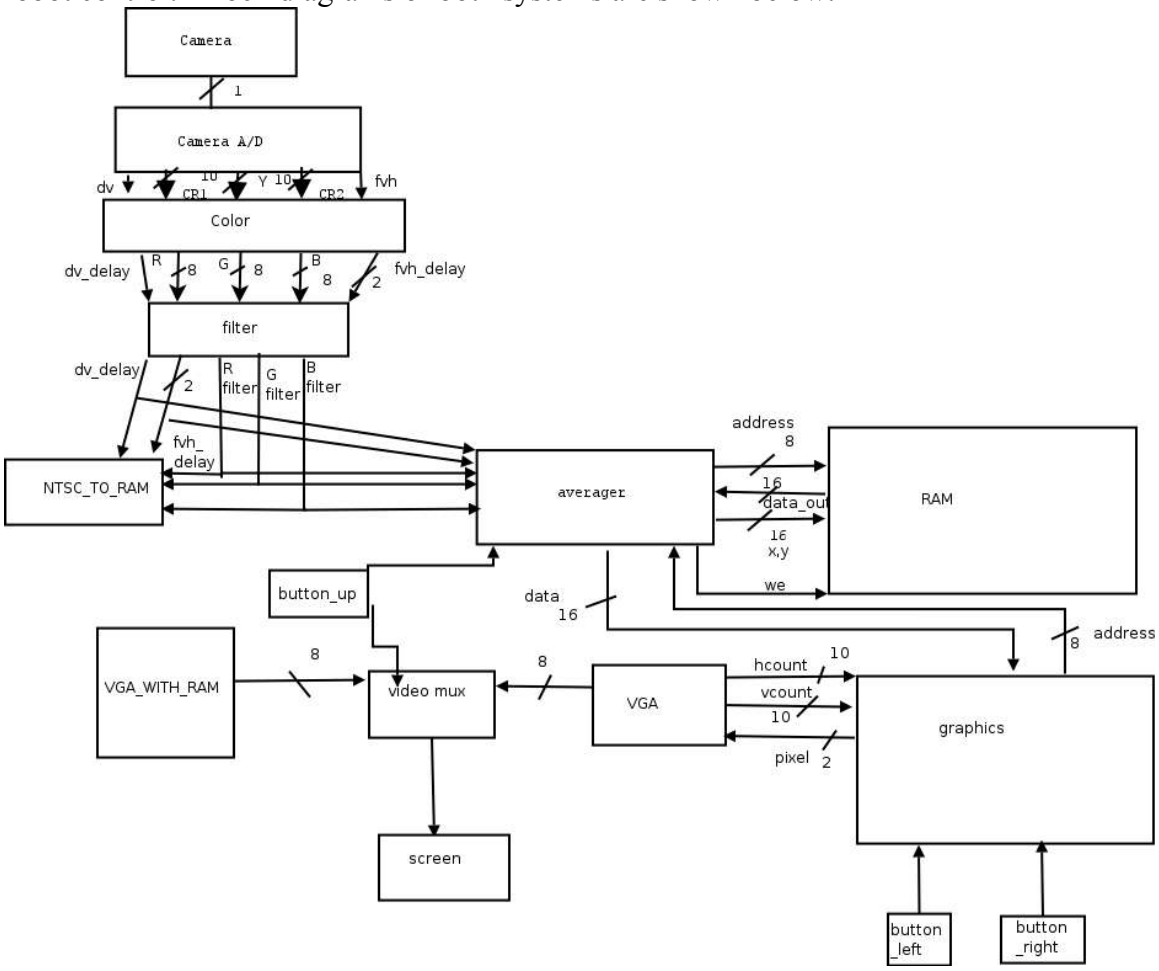

Figure 1. Video system

For the video interface, a module written by Javier Castro interfaces with the ADC1785 video decoder, and streams out the Y, Cr, and Cb data. This data goes into the

module Color, which is a wrapper for a Y, Cr, Cb to R, G, B decoder module downloaded from the xilinx website. Color also provides a 5 cycle delay on the field, vertical, horizontal, and data valid signals, to account for the time it takes the decoder to process data. The RGB data is then passed to the module Filter, which interfaces with the labkit's switches and buttons to allow the user to set maxima and minima on the red, green, and blue values, and so select only a range of colors. Filter passes through all values within the range of the filter, and sets all others to zero. The RGB values then pass through to the module Video_sample, which sums up the x and y positions of the first 256 nonzero color values, and then bitshift by 8, taking the average of the colored pixels. This average is stored in a 256-element RAM. When the user presses the enter button, the module computes the average once every 100 frames, and then increments the address in RAM, storing 256 sample points. The RGB data from Filter is also passed to the module NTSC_to_RAM, which is written by Javier. The RAM contains all 256X198 pixels of the video display, with 8 bits of data at each pixel. The three most significant bits of red and green data are stored in the RAM, along with the two most significan bits of blue. This data is then output to the screen by the module VGA_with_RAM. The display data from the VGA_with_RAM module is muxed with VGA data generated by the module Graphics, which draws a blob at the current X,Y value stored in memory, and allows the user to step through the memory addresses. The WE signal for the RAM that stores the averages is ANDed with the mux signal for the Graphics module, so that it will not be overwritten while the user is reading the data.

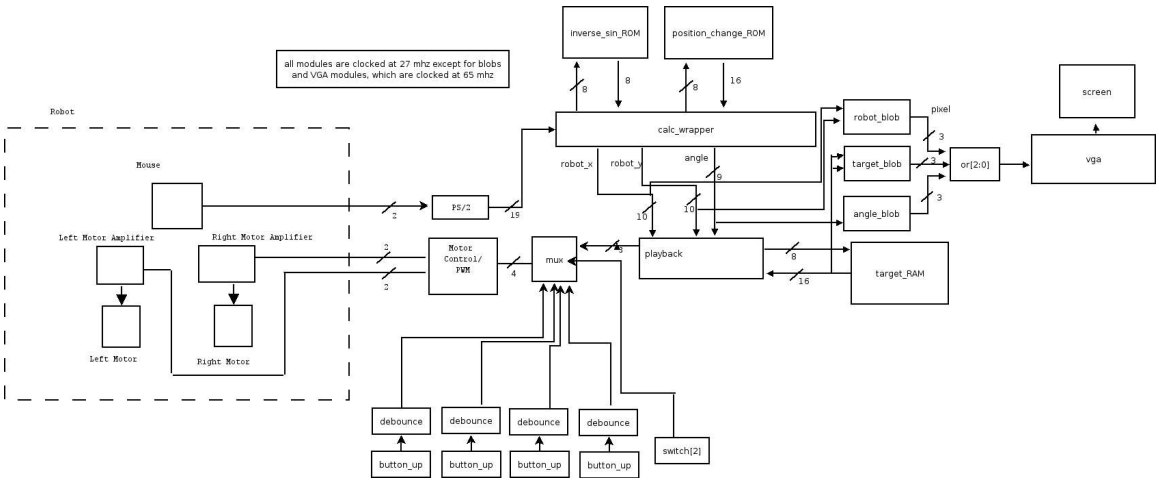

Figure 2. Robot system

The robot control module(above) centers around the Playback module. The playback module gets a target coordinate, either from the RAM that the graphics module writes to, or hardcoded into the module. It then generates the control signals necessary to make to robot go to the target, using a simple FSM. The ps2_mouse_xy module in Final_mouse.v provides an interface with a PS/2 mouse mounted in the robot. The module takes in a 27 mhz clock, reset, and the mouse data and clock lines on the labkit, and gives out 9-bit X and Y signals, with an 8 bit magnitude and one sign bit. The module also provides a data_ready signal to signify the time at which the mouse value is valid. The X, Y, and data_ready signals go to the module Calc_wrapper.

The Calc_wrapper module keeps track of the robot's angle and position by using precalculated ROMs to compute the changes in the angle of the robot due to x movements of the mouse, which correspond to a rotation of the robot. The module then multiplies the y movement by either the sine or cosine of that angle to get a change of position in the global coordinate frame. Finally, it adds the position change values to the robot's current position. Blob modules in the labkit module display the current angle and position of the robot.

The robot(shown below) itself is very simple. When the robot turns, it turns in place, which sweeps the mouse in a circle. The mouse sees this as a change in its x position, which we can use trigonometry if we know the distance from the mouse to the center of the robot to find the change in the robot's angle. To find changes in the robot's absolute position, we multiply the mouse's change in Y by the sin or cosine of the angular change, which gives us values in the global cartestian coordinate system.

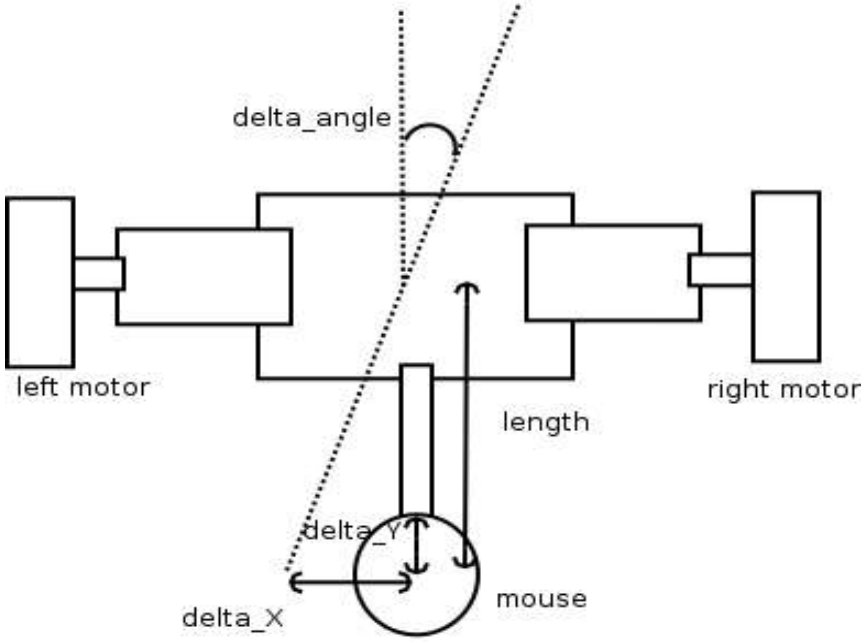

Figure 3: The physical robot

The design for the playback module calls for an FSM to drive the robot to the target. The FSM(shown below) has four states: TURN_RIGHT, TURN_LEFT, GO, and ADVANCE TARGET. The module will calculate the angle from the robot to the target with respect to a vertical line, and it takes in the robot current angle with respect to a vertical line. The FSM will cause the robot to rotate until it is pointing at the target, at which point it will drive straight ahead until it either hits the target or gets out of alignment. When the robot hits the target within a margin of error, the module increments the memory address of the RAM storing the target coordinates, moving on to

the next target.

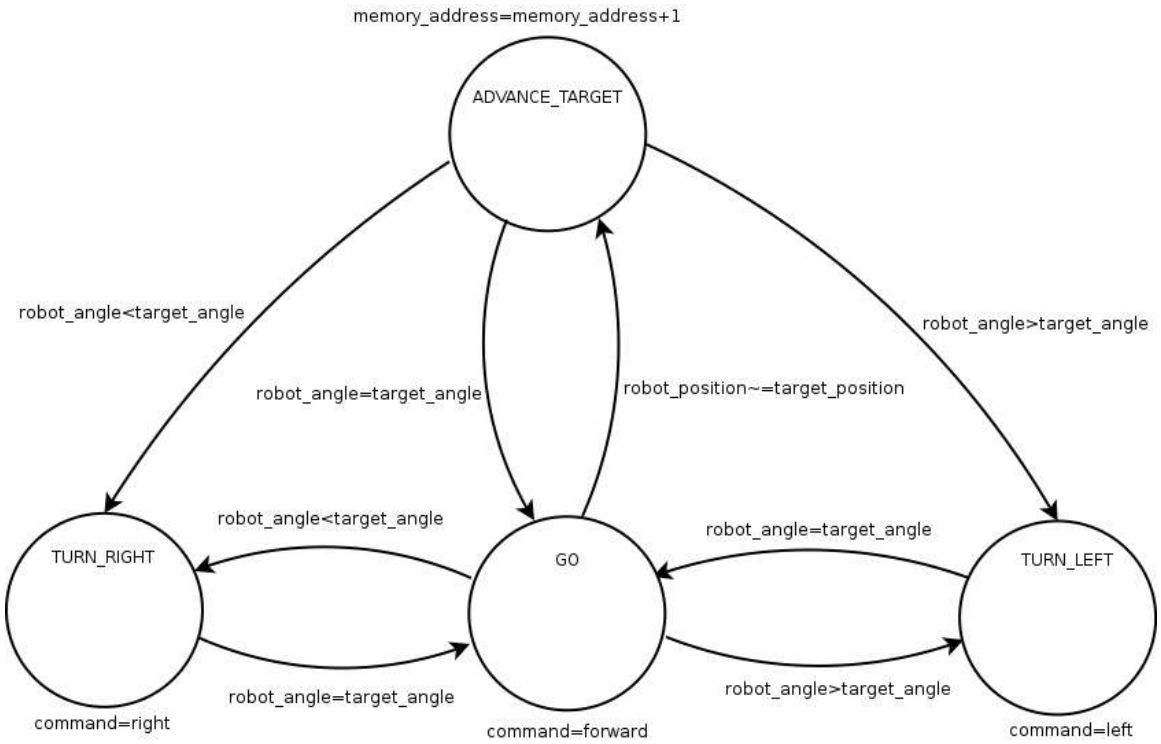

Figure 4. Robot FSM

Testing and Debugging:

The majority of debugging was done on the FPGA, using the logic analyzer and the LEDs and display on the labkit. I also used the VGA output extensively to debug the various modules. To debug the video portion of the project, I interfaced the Filter module with the switches and buttons on the labkit, to allow me to set color filters and see the results on the screen. Later, when I built the finger detection module, I added a VGA interface to allow me to step through the locations in RAM and display the recorded location of my finger for a given sample.

I also used the VGA display for debugging the robot. I added 'blobs' on the display for the position of my mouse in its coordinate frame, the angular changes of the mouse, the absolute angle of the mouse in the global coordinate frame, and the absolute position of the mouse in the global coordinate frame. I also used the labkit's LEDs to display the commands being sent from the Playback module.

Conclusion:

I found this project more difficult than I had anticipated. Most of the problems I encountered were over controlling the robot. The robot was very fast, and was very difficult to control dynamically. Part of this was due to the motor drivers only accepting input voltages in the upper part of the motors' operating range, and also not responding well to PWM control signals. Because of the control issues, I was not able to get the robot to trace out a path.

However, I was able to implement two capabilities separately. I was able to fully implement the video system, and I was able to implement a simple robot control system that would rotate the robot to a given angle, using closed-loop feedback.

After working on this project for uncountable hours over three weeks, I think that an FPGA is not an appropriate choice for a project of this type. This control is fairly computationally intensive, and it would be much simpler to program this sort of system in a high-level language. As it was, I used C to generate the ROMs with the results for most complicated computations.

I did learn a lot about the capabilities and limitations of the verilog HDL. The modularity of verilog is very useful in design and testing. However, when performing high-level computations, it is very easy to get bogged down in low-level details such as format of binary numbers.

I think that a more precisely controllable robot would have greatly facilitated this project. I would like to continue this project, either using an FPGA or a different platform, and add more functionality to the system.

Appendices

Appendix A: Color_video.v

```
//
//  based on javier castro's video_decoder_sample.v
//
// Sample top-level file for the MIT 6.111 labkit, demonstrating
// b&w NTSC video input from the ADV7185.  The video is digitized
// and made available, together with sync signals from the ntsc_decode
// module.  This sample demo displays a 256 x 192 pixel black and white
// image undersampled subset of the video in a SVGA (1024 x 768) window.
//
// Switch[0] is used for testing: 0 = digitized video, 1 = b&w test bars
// button_enter is reset
//
// The dot LEDs display the a version number, and the
// 8 discrete LEDs display the video data read from RAM.

`include "video_decoder.v"
`include "display_16hex.v"
`include "videoram.v"
`include "color.v"
`include "filter.v"
`include "graphics.v"
`include "vga_sync.v"
`include "sample_simple.v"
`include "video_sample.v"
`include "debounce.v"
`include "level_to_pulse.v"


//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
```

```
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 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 = 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'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
  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 analyzer1_clock = clock_27mhz;
  //assign analyzer2_data = 16'h0000;
  //assign analyzer2_clock = 1'b1;
  assign analyzer2_clock = tv_in_line_clock1;
  assign analyzer3_data = 16'h0;
  assign analyzer3_clock = 1'b1;
  assign analyzer4_data = 16'h0;
  assign analyzer4_clock = 1'b1;

  // VGA Output: this vga driver for displaying video from RAM

  wire reset = ~button_enter;
  reg [15:0] vram_addr;
  reg [7:0] vram_data_in;
  wire [7:0] vram_data_out;
  reg vram_clk;
  reg vram_we;


  wire [2:0] pixel;
  wire       blank_gen;
  wire       pix_clk_gen;
  wire [9:0] hcount_gen, vcount_gen;
  wire[7:0] vga_red_gen, vga_green_gen, vga_blue_gen;
  wire[7:0] vga_red_ram, vga_green_ram, vga_blue_ram;
  wire vga_sync_ram, vga_hsync_ram, vga_vsync_ram, vga_pix_clk_ram;
  wire vga_sync_gen, vga_hsync_gen, vga_vsync_gen, vga_pix_clk_gen;
  wire vga_blank_ram;
  wire vga_blank_gen;

wire up, down, left, right;

  wire mode;
  assign mode=up;


  assign vga_out_red=mode? vga_red_ram : vga_red_gen;
  assign vga_out_green=mode? vga_green_ram : vga_green_gen;
  assign vga_out_blue=mode? vga_blue_ram : vga_blue_gen;
  assign vga_out_sync_b =mode? vga_sync_ram : vga_sync_gen;
  assign vga_out_blank_b = mode? vga_blank_ram : vga_blank_gen;
```

```
  assign vga_out_pixel_clock = mode? vga_pix_clk_ram : pix_clk_gen;
  assign vga_out_hsync = mode? vga_hsync_ram : vga_hsync_gen;
  assign vga_out_vsync = mode? vga_vsync_ram : vga_vsync_gen;



        //video friver for generating video

  // use FPGA's digital clock manager to produce a 50 Mhz clock from 27 Mhz
  // actual frequency: 49.85 MHz
  wire clock_50mhz_unbuf,clock_50MHz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 13
  // 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_50MHz),.I(clock_50mhz_unbuf));

/*
  assign vga_out_red=vga_red_gen;
  assign vga_out_green=vga_green_gen;
  assign vga_out_blue=vga_blue_gen;
  assign vga_out_sync_b =vga_sync_gen;
  assign vga_out_blank_b =vga_blank_gen;
  assign vga_out_pixel_clock =pix_clk_gen;
  assign vga_out_hsync =vga_hsync_gen;
  assign vga_out_vsync =vga_vsync_gen;
 */



  debounce db1(reset, clock_27mhz, ~button_up, up);
  debounce db2(reset, clock_27mhz, ~button_left, left);
  debounce db3(reset, clock_27mhz, ~button_right, right);
  debounce db4(reset, clock_27mhz, ~button_down, down);

  wire left_pulse, right_pulse;
  level_to_pulse left_pullllu(clock_27mhz, left, reset, left_pulse);
  level_to_pulse right_pullllu(clock_27mhz, right, reset, right_pulse);

  vga_with_ram vr(reset, clock_27mhz, vga_red_ram, vga_green_ram,
                  vga_blue_ram,
                  vga_sync_ram, vga_blank_ram, vga_pix_clk_ram,
                  vga_hsync_ram, vga_vsync_ram,
                  vram_addr, vram_data_in, vram_data_out, vram_clk, vram_we);

  // 640x480 VGA display
```

```
  assign     vga_red_gen = {8{pixel[0]}};
  assign     vga_green_gen = {8{pixel[1]}};
  assign     vga_blue_gen = {8{pixel[2]}};
  assign     vga_blank_gen = ~blank_gen;

  vga_sync vga1
(clock_50MHz,vga_hsync_gen,vga_vsync_gen,hcount_gen,vcount_gen,pix_clk_gen,blan
k_gen);

  // 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));

  // display memory: test pattern or NTSC video
  wire[7:0] status;

  wire [15:0] vaddr3;
  wire [7:0]  vdata3;
  wire        vclk3;
  wire        vwe3;
  vid_test_pat vp3 (clock_27mhz,vaddr3,vdata3,vclk3,vwe3);

  wire [15:0] vaddr4;
  wire [7:0]  vdata4;
  wire        vwe4;
  wire        vclk4;

  wire[2:0] fvh_color_delay, fvh_filter_delay;
  wire dv_color_delay, dv_dilter_delay;
  wire[7:0] color_data;
  wire[7:0] bw_data;
```

```
  wire[7:0] red, green, blue, red_filter, green_filter, blue_filter;

  assign bw_data=ycrcb[29:22];
  wire[7:0] sample_address;
  wire[15:0] sample_data;
  wire[4:0] point_counter;



        wire[15:0] data_in;
        wire[16:0] hsum;
        wire[16:0] vsum;
        wire[7:0] counting_address;

  video_sample simple(.clock(tv_in_line_clock1), .reset(reset),
                                .fvh(fvh_filter_delay),          .data(color_data),
                                .inhibit(mode), .address(sample_address),
                                .data_out(sample_data), .dv(dv_filter_delay),
                                .point_counter(point_counter),
                                .data_in(data_in), .hsum(hsum), .vsum(vsum),
                                .sample(counting_address));

  color remaster(.clk(tv_in_line_clock1), .reset(reset), .Y(ycrcb[29:20]),
            .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]), .red(red), .green(green), .blue(blue),
                        .dv(dv),. fvh(fvh), .dv_delay(dv_color_delay), .fvh_delay
(fvh_color_delay));

  filter gefilta(.clk(tv_in_line_clock1), .reset(reset), .red(red),
.green(green), .blue(blue),
                        .r_out(red_filter), .g_out(green_filter),
                        .b_out(blue_filter, .r_select(~button2),
                        .g_select(~button1), .b_select(~button0),
                        .filter(switch[7:0]),.compare_select(~button3),
                        .dv_in(dv_color_delay), .dv_out(dv_filter_delay),
                        .fvh_in(fvh_color_delay),
                        .fvh_out(fvh_filter_delay), .status(status));

  graphics hooray(.clock(clock_50MHz), .reset(reset), .next(right_pulse), .previous
(left_pulse),
                                .address(sample_address), .data(sample_data), .video_data
(sample_video), .hcount(hcount_gen),
                                .vcount(vcount_gen),.pixel(pixel));

  assign color_data={red_filter[7:5], green_filter[7:5], blue_filter[7:6]};

  ntsc_to_ram vp4 (.clk(tv_in_line_clock1),
                  .fvh(fvh_filter_delay), .dv(dv_filter_delay),
```

```
                    .din(color_data), .vaddr(vaddr4), .vwe(vwe4),
                    .vdata(vdata4));

  assign analyzer1_clock={tv_in_line_clock1};
  assign analyzer1_data={hsum[15:0]};

  // select video source

  always @(down)
    case (down)
      1'b0: begin                   // fill video RAM with NTSC video data
            vram_addr = vaddr4;
            vram_data_in = vdata4; // luminance data
            vram_clk = tv_in_line_clock1; //vclk4;
            vram_we = vwe4;
      end

      1'b1: begin                   // fill video RAM with b/w bars
            vram_addr = vaddr3;
            vram_data_in = vdata3;
            vram_clk = vclk3;
            vram_we = vwe3;
      end
    endcase

 assign led = switch[0]? ~counting_address : ~sample_address;


  // debugging

  assign      analyzer2_data[15:0] = {2'b00, vwe4, fvh, tv_in_ycrcb[19:10]};

  parameter   VERSION = 8'd51;
  display_16hex my_display(reset, clock_27mhz, {48'b0,sample_data},
                           disp_blank, disp_clock, disp_rs,
                           disp_ce_b, disp_reset_b, disp_data_out);
endmodule


// Fill video RAM with a video pattern: vertical bars

module vid_test_pat(clk,vaddr,vdata,vclk,vwe);

  output vclk,vwe;
  output [7:0] vdata;
  output [15:0] vaddr;
  input          clk;
```

```
  // generate 90 degree phase shifted signal for write-enable
  // so that we is stable and high on rising edge of count[0]

  reg [31:0] count;
  always @(posedge clk) count <= count + 1;

  reg        vwe;
  always @(posedge clk) vwe <= count[0] ^ vwe;

  assign     vaddr = count[17:2];
  assign     vclk = count[0];
  // assign     vwe = count[1];
  assign     vdata = { count[7:2], 2'b00 };

endmodule

// Fill video RAM from NTSC decoded video grabbed data

module ntsc_to_ram(clk, fvh, dv, din, vdata, vaddr, vwe);

  output vwe;
  output [15:0] vaddr;
  output [7:0]  vdata;
  input [2:0]    fvh;
  input          dv;
  input [7:0]    din;
  input          clk;

  //   parameter          MAX_ROW = 191;
  //   parameter          MAX_COL = 255;

  // here put the luminance data from the ntsc decoder into the ram
  reg [7:0]      col = 0;
  reg [7:0]      row = 0;
  reg [7:0]      vdata = 0;
  reg            vwe = 0;

  always @ (posedge clk) //LLC1 is reference
    begin
        if (!fvh[2])
          begin
            col <= fvh[0] ? 8'h00 :
                   (!fvh[2] && !fvh[1] && dv && (col < 255)) ? col + 1 : col;
            row <= fvh[1] ? 8'h00 :
                   (!fvh[2] && fvh[0] && (row < 191)) ? row + 1 : row;
            vwe <= dv && !fvh[2]; // if the data is valid, we can write it
```

```
            vdata <= (dv && !fvh[2]) ? din : vdata;
          end
    end

  assign vaddr = {row,col};

endmodule

// SVGA video module with dual-port video ram
// the vram_* lines are used to read/write to the video RAM
// this module is configured to display a b&w 256x192 pixel image
// stored in the video RAM.

module vga_with_ram (reset, clock_27mhz, 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,
                     vram_addr, vram_data_in, vram_data_out, vram_clk,
                     vram_we);

  input reset; // Active high reset, synchronous with 27MHz clock
  input clock_27mhz; // 27MHz input clock
  output [7:0] vga_out_red, vga_out_green, vga_out_blue; // Outputs to DAC
  output vga_out_sync_b, vga_out_blank_b; // Composite sync/blank outputs to DAC
  output vga_out_pixel_clock; // Pixel clock for DAC
  output vga_out_hsync, vga_out_vsync; // Sync outputs to VGA connector

  input [15:0] vram_addr;     // video ram address
  input [7:0] vram_data_in;   // video ram data input
  input  vram_clk, vram_we; // video ram clock and write enable
  output [7:0] vram_data_out;         // video ram data output

  //////////////////////////////////////////////////////////////////////
  //
  // Timing values
  //
  //////////////////////////////////////////////////////////////////////

  // 1024 X 768 @ 75Hz with a 78.750MHz pixel clock

`define H_ACTIVE    1024 // pixels
`define H_FRONT_PORCH    16 // pixels
`define H_SYNCH          96 // pixels
`define H_BACK_PORCH    176 // pixels
`define H_TOTAL         1312 // pixels

`define V_ACTIVE     768 // lines
```

```
`define V_FRONT_PORCH     1 // lines
`define V_SYNCH           3 // lines
`define V_BACK_PORCH     28 // lines
`define V_TOTAL         800 // lines

  ////////////////////////////////////////////////////////////////////////
  //
  // Internal signals
  //
  ////////////////////////////////////////////////////////////////////////

  wire pixel_clock;
  reg prst, pixel_reset; // Active high reset, synchronous with pixel clock

  reg [7:0] vga_out_red, vga_out_blue, vga_out_green;
  wire vga_out_sync_b, vga_out_blank_b;
  reg hsync1, hsync2, vga_out_hsync, vsync1, vsync2, vga_out_vsync;

  reg [10:0] pixel_count; // Counts pixels in each line
  reg [10:0] line_count; // Counts lines in each frame

  reg [9:0] xpos;        // horizontal image pixel count
  reg [9:0] ypos;        // vertical image pixel count

  ////////////////////////////////////////////////////////////////////////
  //
  // Generate the pixel clock (78.750MHz)
  //
  ////////////////////////////////////////////////////////////////////////

  // synthesis attribute period of clock_27mhz is 37ns;

  DCM vga_dcm (.CLKIN(clock_27mhz),
                          .RST(1'b0),
                          .CLKFX(pixel_clock));
  // synthesis attribute DLL_FREQUENCY_MODE of vga_dcm is "LOW"
  // synthesis attribute DUTY_CYCLE_CORRECTION of vga_dcm is "TRUE"
  // synthesis attribute STARTUP_WAIT of vga_dcm is "TRUE"
  // synthesis attribute DFS_FREQUENCY_MODE of vga_dcm is "LOW"
  // synthesis attribute CLKFX_DIVIDE of vga_dcm is 9
  // synthesis attribute CLKFX_MULTIPLY of vga_dcm is 26
  // synthesis attribute CLK_FEEDBACK of vga_dcm  is "NONE"
  // synthesis attribute CLKOUT_PHASE_SHIFT of vga_dcm is "NONE"
  // synthesis attribute PHASE_SHIFT of vga_dcm is 0
  // synthesis attribute clkin_period of vga_dcm is 37

  assign vga_out_pixel_clock = ~pixel_clock;
```

```
always @(posedge pixel_clock)
  begin
        prst <= reset;
        pixel_reset <= prst;
  end

//////////////////////////////////////////////////////////////////////////
//
// Pixel and Line Counters
//
//////////////////////////////////////////////////////////////////////////

always @(posedge pixel_clock)
  if (pixel_reset)
    begin
          pixel_count <= 0;
          line_count <= 0;
    end
  else if (pixel_count == (`H_TOTAL-1)) // last pixel in the line
    begin
          pixel_count <= 0;
          if (line_count == (`V_TOTAL-1)) // last line of the frame
            line_count <= 0;
          else
            line_count <= line_count + 1;
    end
  else
    pixel_count <= pixel_count +1;

always @(pixel_reset or pixel_count or line_count)
  if (pixel_reset)
    begin
          xpos <= 0;
          ypos <= 0;
    end
  else begin
     if (pixel_count > `H_FRONT_PORCH)
          xpos <= pixel_count - `H_FRONT_PORCH;
     if (line_count > `V_FRONT_PORCH)
          ypos <= line_count - `V_FRONT_PORCH;
  end

//////////////////////////////////////////////////////////////////////////
//
// Sync and Blank Signals
//
```

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

always @ (posedge pixel_clock)
  begin
        if (pixel_reset)
          begin
            hsync1 <= 1;
            hsync2 <= 1;
            vga_out_hsync <= 1;
            vsync1 <= 1;
            vsync2 <= 1;
            vga_out_vsync <= 1;
          end
        else
          begin

            // Horizontal sync
            if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH))
              hsync1 <= 0; // start of h_sync
            else if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH+`H_SYNCH))
              hsync1 <= 1; // end of h_sync

            // Vertical sync
            if (pixel_count == (`H_TOTAL-1))
              begin
                    if (line_count == (`V_ACTIVE+`V_FRONT_PORCH))
                      vsync1 <= 0; // start of v_sync
                    else if (line_count ==
(`V_ACTIVE+`V_FRONT_PORCH+`V_SYNCH))
                      vsync1 <= 1; // end of v_sync
              end

          end

        // Delay hsync and vsync by two cycles to compensate for 2 cycles of
        // pipeline delay in the DAC.
        hsync2 <= hsync1;
        vga_out_hsync <= hsync2;
        vsync2 <= vsync1;
        vga_out_vsync <= vsync2;

  end

  // Blanking
  assign vga_out_blank_b = ((pixel_count<`H_ACTIVE) & (line_count<`V_ACTIVE));

  // Composite sync
```

```
  assign vga_out_sync_b = hsync1 ^ vsync1;

  //////////////////////////////////////////////////////////////////////
  //
  // Display a 256x192 pixel image from dual-port RAM
  //
  //////////////////////////////////////////////////////////////////////

  reg [7:0] pixdata;     // for the memory read pipeline
  reg [15:0] memaddr;           // for memory read address
  wire [7:0] memdata;  // for memory output data 8 bits: BBGGGRRR

  // read data from memory at pixel clock, with one pipeline stage

  always @(posedge pixel_clock)
    begin
     memaddr <= xpos[9:2] + ypos[9:2]*256;          // oversample
     pixdata <= memdata;                  // latch in last value
    end

  // RAM with a 256 x 192 b&w image, 8-bits per pixel
  // the vga output is 1024x768, so we skip 2 bits of xpos and ypos, each

  videoram vram( .addra(memaddr), .clka(pixel_clock), .douta(memdata),
                 .addrb(vram_addr),
                 .doutb(vram_data_out), .dinb(vram_data_in),
                 .clkb(vram_clk), .web(vram_we));

  // feed pixel values to VGA machine

  always @ (posedge pixel_clock)
    begin


        vga_out_red   <= {pixdata[7:5], 5'b00000};
        vga_out_green <= {pixdata[4:2], 5'b00000};
        vga_out_blue  <= {pixdata[1:0], 6'b000000};

        /*
        vga_out_red <= pixdata;                 // for black and white display
        vga_out_green <= pixdata;
        vga_out_blue <= pixdata;
         */
    end

endmodule
```

Appendix B: Color.v

```
module color(clk, reset, Y, Cr, Cb, red, green, blue, fvh, dv, fvh_delay, dv_delay);
        input clk, reset;
        input [9:0] Y, Cr, Cb;
        output [7:0] red, green, blue;
        input[2:0] fvh;
        input dv;
        output[2:0] fvh_delay;
        output dv_delay;
        reg[2:0] fvh_1, fvh_2, fvh_3, fvh_4, fvh_5, fvh_delay;
        reg dv_1, dv_2, dv_3, dv_4, dv_5, dv_delay;
    wire [7:0] red, green, blue;


    YCrCb2RGB convert(red, green, blue, clk, reset, Y, Cr, Cb);

        always@(posedge clk)
        begin
                if(reset) begin
                        fvh_1<=fvh;
                        fvh_2<=0;
                        fvh_3<=0;
                        fvh_4<=0;
                        fvh_5<=0;
                        fvh_delay<=0;
                        dv_1<=dv;
                        dv_2<=0;
                        dv_3<=0;
                        dv_4<=0;
                        dv_5<=0;
                        dv_delay<=0;
                end
                else begin
                        dv_1<=dv;
                        dv_2<=dv_1;
                        dv_3<=dv_2;
                        dv_4<=dv_3;
//                      dv_5<=dv_4;
                        dv_delay<=dv_4;
```

```
                        fvh_1<=fvh;
                        fvh_2<=fvh_1;
                        fvh_3<=fvh_2;
                        fvh_4<=fvh_3;
//                      fvh_5<=fvh_4;
                        fvh_delay<=fvh_4;
                end
        end
endmodule
```

Appendix C: graphics.v

```
module graphics(clock, reset, next, previous, address, data, video_data, hcount, vcount,
pixel);
        input clock, reset, next, previous;
        input [15:0] data;
        input [9:0] hcount, vcount;
        output[7:0] address;
        output[7:0] video_data;
        reg[7:0] address;
        output[2:0] pixel;

        wire[2:0] avg_pixel, border_pixel;
        assign pixel=avg_pixel | border_pixel;
//      blob the_blob(10'd100, 10'd100, hcount, vcount, pixel);

        blob the_blob({2'b0, data[15:8]}, {2'b0, data[7:0]}, hcount, vcount, avg_pixel);
        border south_of_the_border(hcount,vcount, border_pixel);
        always@(posedge clock) begin
                if(reset) begin
                        address<=0;
                end
                else begin
                        if(next) address<=address+1;
                        if(previous) address<=address-1;

                end
        end
```

```
endmodule

module border(hcount,vcount,pixel);
	input [9:0] hcount, vcount;
	output[2:0] pixel;
	parameter Y=8'd192;
	parameter X=8'd255;
	reg [2:0] pixel;
	always@(hcount or vcount)
	begin
		if((hcount==X)||(vcount==Y))
			pixel<=3'b111;
		else
			pixel<=3'b0;
	end
endmodule



module blob(x,y,hcount,vcount,pixel);

  parameter WIDTH = 5;
  parameter HEIGHT = 5;
  parameter COLOR = 3'b100;

  input [9:0] x,hcount;		// x,y specifies upper left corner
  input [9:0] y,vcount;
  output [2:0] pixel;

  reg [2:0]	pixel;
  always @(x or y or hcount or vcount) begin
    if ((hcount >= x && hcount < (x+WIDTH)) &&
	  (vcount >= y && vcount < (y+HEIGHT)))
	pixel = COLOR;
    else pixel = 0;
  end
endmodule
```

Appendix D: filter.v

```
module filter(clk, reset, red, green, blue, r_out, g_out, b_out, r_select, b_select, g_select,
filter,
                    compare_select, dv_in, dv_out, fvh_in, fvh_out, status);

        input clk, reset, r_select, g_select,b_select, compare_select;
        input[7:0] red, green, blue, filter;
        input dv_in;
        output dv_out;
        reg dv_out;
        input[2:0] fvh_in;
        output[2:0] fvh_out;
        reg[2:0] fvh_out;
        output[7:0] r_out, g_out, b_out;
```

```
	reg[7:0] r_out, g_out, b_out;
	reg[7:0] r_in, g_in, b_in, filter_val;
	reg r_mode, g_mode, b_mode;
	reg[7:0] r_filter, g_filter, b_filter;
	reg r_pass, g_pass, b_pass;
	output[7:0] status;

	assign status={r_mode, g_mode, b_mode, r_select, g_select, b_select,
compare_select, 1'b0};
	always @( posedge clk)
	begin
		r_in=red;
		g_in=green;
		b_in=blue;
		filter_val=filter;

		if(r_select) begin
			r_filter=filter;
			if(compare_select)
				r_mode=1'b1;
			else
				r_mode=1'b0;
		end
		if(g_select) begin
			g_filter=filter;
				if (compare_select)
					g_mode=1'b1;
				else
					g_mode=1'b0;
		end
		if(b_select) begin
			b_filter=filter;
				if(compare_select)
					b_mode=1'b1;
				else
					b_mode=1'b0;
		end

	    r_pass=r_mode? (r_in>r_filter) : (r_in<r_filter);
	    g_pass=g_mode? (g_in>g_filter) : (g_in<g_filter);
	    b_pass=b_mode? (b_in>b_filter) : (b_in<b_filter);

		r_out<=(r_pass&g_pass&b_pass)? r_in : 8'b0;
		g_out<=(r_pass&g_pass&b_pass)? g_in : 8'b0;
		b_out<=(r_pass&g_pass&b_pass)? b_in : 8'b0;

		dv_out<=dv_in;
```

```
            fvh_out<=fvh_in;
        end
endmodule
```

Appendix E: robot_labkit.v

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

`include "display_16hex.v"
`include "final_mouse.v"
`include "debounce.v"
`include "vga_sync.v"
`include "graphics.v"
`include "motor.v"
`include "playback_final.v"


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;

inout  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 = 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

  // button0, button1, button2, button3, button_enter, button_right,
  // button_left, button_down, button_up, and switches are inputs

  // User I/Os
  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;

/* mouse stuff*/
  // use FPGA's digital clock manager to produce a 50 Mhz clock from 27 Mhz
  // actual frequency: 49.85 MHz

  wire clock_50mhz_unbuf,clock_50MHz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
```

```
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 13
  // 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_50MHz),.I(clock_50mhz_unbuf));

  ////////////////////////////////////////
  // The ps2 mouse test: display 640x480 screen with mouse controlled cursor

  wire   clk = clock_50MHz;
  wire power_on_reset;
  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 user_reset;
  debounce dbreset(1'b0,clk,~button_enter,user_reset);

  wire reset = power_on_reset | user_reset;



  // mouse
//   wire [8:0] front_mx,front_my;
//   wire [8:0] back_mx,back_my;
  wire[15:0] ysum;
  wire [8:0] front_mx,front_my;
  wire [2:0]  front_btn_click;
  wire [8:0] front_mx_o,front_my_o;

  wire mouse_data_ready;
  ps2_mouse_xy m1(clk, reset, mouse_clock, mouse_data, front_mx_o, front_my_o,
front_btn_click, mouse_data_ready);
  defparam    m1.MAX_X = 640-10;          // max - blob size
  defparam    m1.MAX_Y = 480-10;


/* display stuff*/
  wire [9:0] calc_out;

  wire [63:0] dispdata = {front_mx,front_my,30'b0,ysum};
  display_16hex d1(reset, clk, dispdata,
                   disp_blank, disp_clock, disp_rs, disp_ce_b,
                   disp_reset_b, disp_data_out);

/* vga stuff */
  // 640x480 VGA display
```

```
  wire [2:0] pixel;
  wire       blank;
  wire       pix_clk;
  wire [9:0] hcount, vcount;
  assign     vga_out_red = {8{pixel[0]}};
  assign     vga_out_green = {8{pixel[1]}};
  assign     vga_out_blue = {8{pixel[2]}};
  assign     vga_out_blank_b = ~blank;
  assign     vga_out_pixel_clock = pix_clk;   // vga pixel clock

  vga_sync vga1(clk,vga_out_hsync,vga_out_vsync,hcount,vcount,pix_clk,blank);


/* buttons and motor wires */
    wire up,down,left,right, motor_up, motor_down, motor_left, motor_right;
    wire leftA, leftB, rightA, rightB;
    wire[2:0] command;

  debounce db1(reset, clock_27mhz, ~button_up, up);
  debounce db2(reset, clock_27mhz, ~button_left, left);
  debounce db3(reset, clock_27mhz, ~button_right, right);
  debounce db4(reset, clock_27mhz, ~button_down, down);


  assign motor_up=(switch[2] ? up : command[2]);
  assign motor_down=(switch[2] ? down : 0);
  assign motor_left=(switch[2] ? left : command[1]);
  assign motor_right=(switch[2] ? right : command[0]);



  assign user1={28'b0,leftA,leftB,rightA,rightB};


/* graphics */


  wire [11:0] target_x, target_y;
  wire [11:0] angley;
  assign angley=11'd400;

  assign target_x=11'd500;
  assign target_y=11'd200;
  // video output
```

```
   wire [2:0] front_pixel;
   wire [2:0] robot_pixel;
   wire [2:0] tpixel;
   wire[2:0] change_pixel;
   not_the_same nx(front_mx_o, mouse_sampler, reset, front_mx);
   //not_the_same ny(front_my_o, mouse_sampler, reset, front_my);
   assign front_mx=front_mx_o;
   assign front_my=front_my_o;
  wire[9:0] robotx, roboty;
   wire[7:0] change_x, change_y;
   wire[9:0] scaled_x, scaled_y;
   wire[8:0] angle;
//module calc_wrapper(x, y, reset, clock, delta_x, delta_y, robotx, roboty);

   calc_wrapper calcdoodle(.x(front_mx), .y(front_my), .reset(reset), .clock
(clock_27mhz),
   .delta_x(change_x), .delta_y(change_y), .robotx(robotx), .roboty(roboty), .angle(angle)
   ,.dr(mouse_data_ready),.y_sum(ysum));

   wire       fpixel = (hcount==0 | hcount==639 | vcount==0 | vcount==479);
   wire[2:0] angle_boundary;

   assign angle_boundary=((hcount==10'd180)&&(vcount>10'd200));


   //blob front_mouse({2'b0, front_mx[7:0]},{2'b0,front_my[7:0]},
hcount,vcount,front_pixel);
   //defparam front_mouse.COLOR=3'b100;

   blob front_mouse({1'b0, angle},angley[9:0],hcount,vcount,front_pixel);
   defparam front_mouse.COLOR=3'b100;

   blob target(target_x[9:0], target_y[9:0], hcount, vcount, tpixel);
   defparam target.COLOR=3'b010;

   blob robotblob(robotx, roboty, hcount, vcount, robot_pixel);
   defparam robotblob.COLOR=3'b011;


   wire[7:0] lookup;

   assign   pixel = (switch[0] ? hcount[7:5] : {3{fpixel}} | front_pixel | tpixel | robot_pixel
|change_pixel
                                    | angle_boundary) & ~blank;

   playback playtime(clock_27mhz, reset, robotx, roboty, angle, command, target_x[9:0],
target_y[9:0]);
```

```
        wire[2:0] communist;


  motor_control zoom(motor_up, motor_down, motor_left, motor_right, leftA, leftB,
rightA, rightB);



        assign analyzer1_clock=clock_27mhz;
        assign analyzer1_data={front_mx[7:0], mouse_data_ready, 7'b0};
        assign led=~{5'b0, command[2:0]};
//      assign led=~{calc_out[7:0]};
endmodule



module calc_wrapper(x, y, reset, clock, delta_x, delta_y, robotx, roboty, angle, dr,
y_sum);
        input dr;
        input[8:0] x,y;
        input clock, reset;
        wire[7:0] data;
        reg[7:0] doody;
        output [15:0] y_sum;
        output[7:0] delta_x, delta_y;
        output[8:0] angle;
        reg[8:0] angle;
        reg[8:0] delta_angle;
    calc another_calc(x[7:0], clock, data);
        delta_rom chchchchchanges({y[7:0],          data[6:0]}, clock, {delta_x,
delta_y});
        output[9:0] robotx, roboty;
        reg[9:0] robotx, roboty;
        reg[15:0] yval;
        reg[9:0] scaled_x, scaled_y;
        reg[15:0] y_sum;
        always@(posedge clock)
        begin
                if(reset)
                begin
                        robotx<=0;
                        roboty<=0;
                        angle<=0;
                        y_sum<=0;
                end
                else begin
```

```
                delta_angle<=dr? data : 0;
                if(x[8])
                begin
                        if(angle>delta_angle)
                                angle<=angle+(~delta_angle)+1;
                        else
                                angle<=10'd360+angle+(~delta_angle)+1;
                end
                if(y[8])
                begin
                        if((angle+delta_angle)>10'd360)
                                angle<=angle+delta_angle+(~10'd360)+1;
                        else
                                angle<=angle+delta_angle;
                end

                scaled_x=dr ? (delta_x>>4):0;
                scaled_y=dr ? (delta_y>>4):0;

                yval<=dr? {8'b0, y[7:0]}:16'b0;
                y_sum<=y_sum+yval;
                if(x[8])
                begin
                        robotx<=robotx+(~scaled_x+1);
                end
                else
                begin
                        robotx<=robotx+scaled_x;
                end
                if(y[8])
                begin
                        roboty<=roboty+(~scaled_y+1);
                end
                else
                begin
                        roboty<=roboty+scaled_y;
                end


        end
    end
endmodule
```

Appendix F: motor.v

```
module motor_control(up, down, left, right, leftA, leftB, rightA, rightB);
	input up,down,left,right;
	output leftA,leftB,rightA,rightB;
	reg leftA,leftB,rightA,rightB;

	always @(up or down or left or right)
	begin
		if(up)
		begin
			leftA=1;
			leftB=0;
			rightA=1;
			rightB=0;
		end

		else if(down)
		begin
			leftA=0;
			leftB=1;
			rightA=0;
			rightB=1;
		end

		else if(right)
		begin
			leftA=1;
			leftB=0;
			rightA=0;
			rightB=1;
		end

		else if(left)
		begin
			leftA=0;
			leftB=1;
			rightA=1;
			rightB=0;
		end

		else
		begin
			leftA=0;
```

```
                leftB=0;
                rightA=0;
                rightB=0;
        end

    end
endmodule
```

Appendix G: playback_final.v

```
module playback(clock, reset, x, y, angle, command, targetx, targety);
    input clock, reset;
    input[9:0] x,y,targetx,targety;
    input[8:0] angle;
    output[2:0] command;  //up, left, right
    reg[2:0] command;
    parameter up=3'b100;
    parameter left=3'b010;
    parameter right=3'b001;
    always@(posedge clock)
    begin
        if(reset)
            command<=0;
        else
        begin
            if(angle>9'd180)
                command<=left;
            else
                command<=right;
        end
    end
endmodule
```

Appendix H: final_mouse.v

```
// ps2_mouse_xy gives a high-level interface to the mouse, which
// keeps track of the "absolute" x,y position (within a parameterized
// range) and also returns button presses.

module ps2_mouse_xy(clk, reset, ps2_clk, ps2_data, mx, my, btn_click,
data_ready_delay);

  input clk, reset;
  inout ps2_clk, ps2_data;     // data to/from PS/2 mouse
  output [8:0] mx, my;         // current mouse position, 12 bits
  output [2:0]  btn_click;     // button click: Left-Middle-Right

  // module parameters
  parameter     MAX_X = 1023;
  parameter     MAX_Y = 767;

  // low level mouse driver

  wire [8:0]     dx, dy;
  wire [2:0]     btn_click;
  output         data_ready_delay;
  reg data_ready_delay;
  wire   error_no_ack;
  wire [1:0]     ovf_xy;
  wire   streaming;

  ps2_mouse m1(clk,reset,ps2_clk,ps2_data,dx,dy,ovf_xy, btn_click,
```

```
                    data_ready,streaming);

  // Update "absolute" position of mouse

  reg [8:0]  mx, my;
  wire       sx = dx[8];                    // signs
  wire       sy = dy[8];
//   wire [8:0]  ndx =  {sx,dx[7:0]};   // magnitudes
 //  wire [8:0]  ndy =  {sy,dy[7:0]};
 //   wire [8:0]  ndx = sx ? {sx,(~dx[7:0])+1} : {sx,dx[7:0]};          // magnitudes
   //wire [8:0]  ndy = sy ? {sy,(~dy[7:0])+1} : {sy,dy[7:0]};
  //       wire[8:0] ndx=dx;
  //       wire[8:0] ndy=dy;

  always @(posedge clk) begin
          data_ready_delay<=data_ready;
          mx<= reset? 0 : data_ready ? (sx ? {sx, (~dx[7:0])+1} : {sx, dx[7:0]}) : mx;
          my<= reset? 0 : data_ready ? (sy ? {sy, (~dy[7:0])+1} : {sy, dy[7:0]})  : my;


//     mx <= reset ? 0 :
//             data_ready ? (sx ? (mx>ndx ? mx - ndx : 0)
//                           : (mx < MAX_X - ndx ? mx+ndx : MAX_X)) : mx;
     // note Y is flipped for video cursor use of mouse
//      my <= reset ? 0 :
//             data_ready ? (sy ? (my < MAX_Y - ndy ? my+ndy : MAX_Y)
//                           : (my>ndy ? my - ndy : 0))  : my;
//             data_ready ? (sy ? (my>ndy ? my - ndy : 0)
//                           : (my < MAX_Y - ndy ? my+ndy : MAX_Y)) : my;
  end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//  PS/2 MOUSE
//
//  6.111 Fall 2005
//
//  NOTE:  make sure to change the mouse ports (mouse_clock, mouse_data) to
//                  bi-directional 'inout' ports in the top-level module
//
//  specifically, labkit.v should have the line
//
//     inout  mouse_clock, mouse_data;
//
// This module interfaces to a mouse connected to the labkit's PS2 port.
// The outputs provided give dx and dy movement values (9 bits 2's comp)
```

```
// and three button click signals.
//
// NOTE: change the following parameters for a different system clock
//       (current configuration for 50 MHz clock)
//              CLK_HOLD                        : 100 usec hold to bring PS2_CLK
low
//              RCV_WATCHDOG_TIMER_VALUE : (PS/2 RECEIVER) 2 msec count
//              RCV_WATCHDOG_TIMER_BITS  :                      bits needed for
timer
//              INIT_TIMER_VALUE                : (INIT process) 0.5 sec count
//              INIT_TIMER_BITS         :                       bits needed for timer
////////////////////////////////////////////////////////////////////////////////////
//
// Nov-8-2005: Registered the outputs (dout_dx, dout_dy, ovf_xy, btn_click) in
[ps2_mouse]
//                      Added output "streaming"
// Nov-9-2005: synchronized ps2_clk to local clock for transmitter in [ps2_interface]
//                      Programmed watchdog_timer for [ps2] (receiver module)
//                      Programmed init_timer for [ps2_mouse] (resets initialization)
////////////////////////////////////////////////////////////////////////////////////

module ps2_mouse(clock, reset, ps2_clk, ps2_data, dout_dx,
                 dout_dy, ovf_xy, btn_click, ready, streaming);

  input clock, reset;
  inout ps2_clk, ps2_data;              //data to/from PS/2 mouse
  output [8:0] dout_dx, dout_dy;        //9-bit 2's compl, indicates movement of mouse
  output [1:0] ovf_xy;                  //==1 if overflow: dx, dy
  output [2:0] btn_click;               //button click: Left-Middle-Right
  output ready;                                 //synchronous 1 cycle ready flag
  output streaming;                             //==1 if mouse is in stream mode

  ////////////////////////////////////////////////////
  // PARAMETERS
  // # of cycles for clock=50 MHz
  parameter CLK_HOLD = 5000;                                    //100 usec hold to
bring PS2_CLK low
  parameter RCV_WATCHDOG_TIMER_VALUE = 100000;      // For PS/2
RECEIVER : # of sys_clks for 2msec.
  parameter RCV_WATCHDOG_TIMER_BITS  = 17;      //                                 :
bits needed for timer
  parameter INIT_TIMER_VALUE = 25000000;                // For INIT process : sys_clks
for 0.5 sec.(SELF-TEST phase takes several miliseconds)
  parameter INIT_TIMER_BITS  = 28;                      //                              : bits
needed for timer
  ////////////////////////////////////////////////////
  wire reset_init_timer;
```

```
  ////////////////////////////////////////////////////////
  //CONTROLLER:
  //-initialization process:
  //       Host:  FF  Reset command
  //       Mouse: FA  Acknowledge
  //       Mouse: AA  Self-test passed
  //       Mouse: 00  Mouse ID
  //       Host:  F4  Enable
  //       Mouse: FA  Acknowledge
  parameter SND_RESET = 0,  RCV_ACK1 = 1,  RCV_STEST = 2, RCV_ID = 3;
  parameter SND_ENABLE =4,  RCV_ACK2 = 5,  STREAM = 6;
  reg [2:0] state;

  wire send, ack;
  wire [7:0] packet;
  wire [7:0] curkey;
  wire key_ready;

  //NOTE: no support for scrolling wheel, extra buttons
  always @(posedge clock) begin
    if (reset || reset_init_timer) state <= SND_RESET;
    else case (state)
      SND_RESET:    state <= ack ? RCV_ACK1 : state;
      RCV_ACK1:     state <= (key_ready && curkey==8'hFA) ? RCV_STEST : state;
      RCV_STEST:    state <= (key_ready && curkey==8'hAA) ? RCV_ID : state;
      RCV_ID:       state <= (key_ready) ? SND_ENABLE : state;                  //any
device type
      SND_ENABLE:  state <= ack ? RCV_ACK2 : state;
      RCV_ACK2:     state <= (key_ready && curkey==8'hFA) ? STREAM :state;
      STREAM:               state <= state;
           default:       state <= SND_RESET;
    endcase
  end

  assign send = (state==SND_RESET) || (state==SND_ENABLE);
  assign packet = (state==SND_RESET) ? 8'hFF :
                              (state==SND_ENABLE) ? 8'hF4 :
                                                                  8'h00;
  assign streaming = (state==STREAM);


  // Connect PS/2 interface module
  ps2_interface ps2_mouse(.reset(reset), .clock(clock),
                                          .ps2c(ps2_clk), .ps2d(ps2_data),
                                          .send(send), .snd_packet(packet), .ack(ack),
```

```
                                        .rcv_packet(curkey), .key_ready(key_ready)  );
  defparam ps2_mouse.CLK_HOLD                    = CLK_HOLD;
  defparam ps2_mouse.WATCHDOG_TIMER_VALUE =
RCV_WATCHDOG_TIMER_VALUE;
  defparam ps2_mouse.WATCHDOG_TIMER_BITS  =
RCV_WATCHDOG_TIMER_BITS;

  ////////////////////////////////////////////////////////
  // DECODER
  //http://www.computer-engineering.org/ps2mouse/
  //                        bit-7                                3                bit-0
  //Byte 1:  Y-ovf  X-ovf  Y-sign  X-sign  1  Btn-M  Btn-R  Btn-L
  //Byte 2:  X movement
  //Byte 3:  Y movement
  reg [1:0] bindex, old_bindex;
  reg [7:0] status, dx, dy;              //temporary storage of mouse status
  reg [8:0] dout_dx, dout_dy;            //Clock the outputs
  reg [1:0] ovf_xy;
  reg [2:0] btn_click;
  wire ready;

  always @(posedge clock) begin
          if (reset) begin
            bindex <= 0;
            status <= 0;
            dx <= 0;
            dy <= 0;
          end else if (key_ready && state==STREAM) begin
            case (bindex)
                    2'b00:  status <= curkey;
                    2'b01:  dx <= curkey;
                    2'b10:  dy <= curkey;
                    default:          status <= curkey;
            endcase

            bindex <= (bindex==2'b10) ? 0 : bindex + 1;
            if (bindex == 2'b10) begin                                  //Now, dy is ready
                    dout_dx   <= {status[4], dx};                               //2's compl 9-
bit
                    dout_dy   <= {status[5], curkey};                   //2's compl 9-bit
                    ovf_xy    <= {status[6], status[7]};                //overflow: x, y
                    btn_click <= {status[0], status[2], status[1]};  //button click: Left-Middle-
Right
          end
          end     //end else-if (key_ready)
  end
```

```
  always @(posedge clock)
    old_bindex <= bindex;

  assign ready = (bindex==2'b00) && old_bindex==2'b10;

  ////////////////////////////////////////////////////////
  // INITIALIZATION TIMER
  //       ==> RESET if processs hangs during initialization
  reg [INIT_TIMER_BITS-1:0] init_timer_count;
  assign reset_init_timer = (state != STREAM) &&
(init_timer_count==INIT_TIMER_VALUE-1);
  always @(posedge clock)
  begin
          init_timer_count <= (reset || reset_init_timer || state==STREAM) ?
                                                                  0 : init_timer_count +
1;
  end

endmodule



////////////////////////////////////////////////////////////////////////
// PS/2 INTERFACE: transmit or receive data from ps/2

module ps2_interface(reset, clock, ps2c, ps2d, send, snd_packet, ack, rcv_packet,
key_ready);
   input clock,reset;
   inout ps2c;                          // ps2 clock     (BI-DIRECTIONAL)
   inout ps2d;                          // ps2 data      (BI-DIRECTIONAL)
   input send;                          //flag: send packet                    _
   output ack;                          // end of transmission                | for
transmitting
   input [7:0] snd_packet;      // data packet to send to PS/2          _|
   output [7:0] rcv_packet;     //packet received from PS/2            _
   output key_ready;            // new data ready (rcv_packet)         _| for receiving

   ////////////////////////////////////////////////////////////////////////
   // MAIN CONTROL
   ////////////////////////////////////////////////////////////////////////
   parameter CLK_HOLD = 5005;                           //hold PS2_CLK low for 100
usec (@50 Mhz)
   parameter WATCHDOG_TIMER_VALUE = 100000;    // Number of sys_clks for
2msec. _
   parameter WATCHDOG_TIMER_BITS  = 17;             // Number of bits needed for
timer    _| for RECEIVER
```

```
  wire serial_dout;                     //output (to ps2d)
  wire rcv_ack;                                 //ACK from ps/2 mouse after data
transmission

  wire we_clk, we_data;

  assign ps2c = we_clk  ? 0 : 1'bZ;
  assign ps2d = we_data ? serial_dout : 1'bZ;

  assign ack = rcv_ack;


  //////////////////////////////////////////////////////////////////////////////
  // TRANSMITTER MODULE
  //////////////////////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////
  // COUNTER: 100 usec hold
  reg [15:0] counter;
  wire en_cnt;
  wire cnt_ready;
  always @(posedge clock) begin
    counter <= reset ? 0 :
                        en_cnt ? counter+1 :
                                 0;
  end
  assign cnt_ready = (counter>=CLK_HOLD);

  ////////////////////////////////////////////////////
  // SEND DATA
  //      hold CLK low for at least 100 usec
  //      DATA low
  //      Release CLK
  //      (on negedge of ps2_clock) - device brings clock LOW
  //              REPEAT:       SEND data
  //      Release DATA
  //      Wait for device to bring DATA low
  //      Wait for device to bring CLK low
  //      Wait for device to release CLK, DATA
  reg [3:0] index;

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire falling_edge = ps2c_sync[2] & ~ps2c_sync[1];

  always @(posedge clock) begin
```

```
        if (reset) begin
                index <= 0;
        end
        else if (falling_edge) begin                    //falling edge of ps2c
          if (send) begin                                       //transmission mode
                if (index==0)
                        index <= cnt_ready ? 1 : 0;             //index=0: CLK low
                else
                        index <= index + 1;                     //index=1: snd_packet
[0], =8: snd_packet[7],
                                                                //      9: odd
parity, =10: stop bit
                                                                //          11:
wait for ack
            end else
                index <= 0;
        end else
          index <= (send) ? index : 0;
  end

  assign en_cnt = (index==0 && ~reset && send);
  assign serial_dout = (index==0 && cnt_ready) ? 0 :                    //bring
DATA low before releasing CLK
                                (index>=1 && index <=8) ? snd_packet[index-1] :
                                (index==9) ? ~(^snd_packet) :
        //odd parity
                                                1;
                //including last '1' stop bit

  assign we_clk = (send && !cnt_ready && index==0);                     //Enable
when counter is counting up
  assign we_data = (index==0 && cnt_ready) || (index>=1 && index<=9);//Enable after
100usec CLK hold
  assign rcv_ack = (index==11 && ps2d==0);
        //use to reset RECEIVER module


  ////////////////////////////////////////////////////////////////////////
  // RECEIVER MODULE
  ////////////////////////////////////////////////////////////////////////
  reg [7:0]   rcv_packet;               // current keycode
  reg             key_ready;            // new data
  wire        fifo_rd;                  // read request
  wire [7:0]  fifo_data;                // data from mouse
  wire        fifo_empty;               // flag: no data
  //wire        fifo_overflow;    // keyboard data overflow
```

```
  assign      fifo_rd = ~fifo_empty;    // continuous read

  always @(posedge clock)
  begin
        // get key if ready
        rcv_packet <= ~fifo_empty ? fifo_data : rcv_packet;
        key_ready  <= ~fifo_empty;
  end

  ////////////////////////////////////////////////////
  // connect ps2 FIFO module
  reg [WATCHDOG_TIMER_BITS-1 : 0] watchdog_timer_count;
  wire [3:0] rcv_count;                                  //count incoming data bits from ps/2
(0-11)

  wire watchdog_timer_done = watchdog_timer_count==
(WATCHDOG_TIMER_VALUE-1);
  always @(posedge clock)
  begin
        if (reset || send || rcv_count==0) watchdog_timer_count <= 0;
        else if (~watchdog_timer_done)
          watchdog_timer_count <= watchdog_timer_count + 1;
  end

  ps2 ps2_receiver(.clock(clock), .reset(!send && (reset || rcv_ack) ),       //RESET on
reset or End of Transmission
                                        .ps2c(ps2c), .ps2d(ps2d),
                                        .fifo_rd(fifo_rd), .fifo_data(fifo_data),
        //in1, out8
                                        .fifo_empty(fifo_empty) , .fifo_overflow(),
        //out1, out1
                                        .watchdog(watchdog_timer_done), .count
(rcv_count) );

endmodule



////////////////////////////////////////////////////////////////////////
// PS/2 FIFO receiver module (from 6.111 Fall 2004)

module ps2(reset, clock, ps2c, ps2d, fifo_rd, fifo_data, fifo_empty,fifo_overflow,
watchdog, count);
  input clock,reset,watchdog,ps2c,ps2d;
  input fifo_rd;
  output [7:0] fifo_data;
  output fifo_empty;
```

```
output fifo_overflow;
output [3:0] count;

reg [3:0] count;      // count incoming data bits
reg [9:0] shift;      // accumulate incoming data bits

reg [7:0] fifo[7:0];   // 8 element data fifo
reg fifo_overflow;
reg [2:0] wptr,rptr;   // fifo write and read pointers

wire [2:0] wptr_inc = wptr + 1;

assign fifo_empty = (wptr == rptr);
assign fifo_data = fifo[rptr];

// synchronize PS2 clock to local clock and look for falling edge
reg [2:0] ps2c_sync;
always @ (posedge clock) ps2c_sync <= {ps2c_sync[1:0],ps2c};
wire sample = ps2c_sync[2] & ~ps2c_sync[1];

reg timeout;
always @ (posedge clock) begin
  if (reset) begin
    count <= 0;
    wptr <= 0;
    rptr <= 0;
    timeout <= 0;
    fifo_overflow <= 0;
  end else if (sample) begin
    // order of arrival: 0,8 bits of data (LSB first),odd parity,1
    if (count==10) begin
      // just received what should be the stop bit
      if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
        fifo[wptr] <= shift[8:1];
        wptr <= wptr_inc;
           fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
      end
      count <= 0;
      timeout <= 0;
    end else begin
      shift <= {ps2d,shift[9:1]};
      count <= count + 1;
    end
  end else if (watchdog && count!=0) begin
    if (timeout) begin
      // second tick of watchdog while trying to read PS2 data
         count <= 0;
```

```
        timeout <= 0;
      end else timeout <= 1;
    end

    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end
endmodule
```

Appendix I: