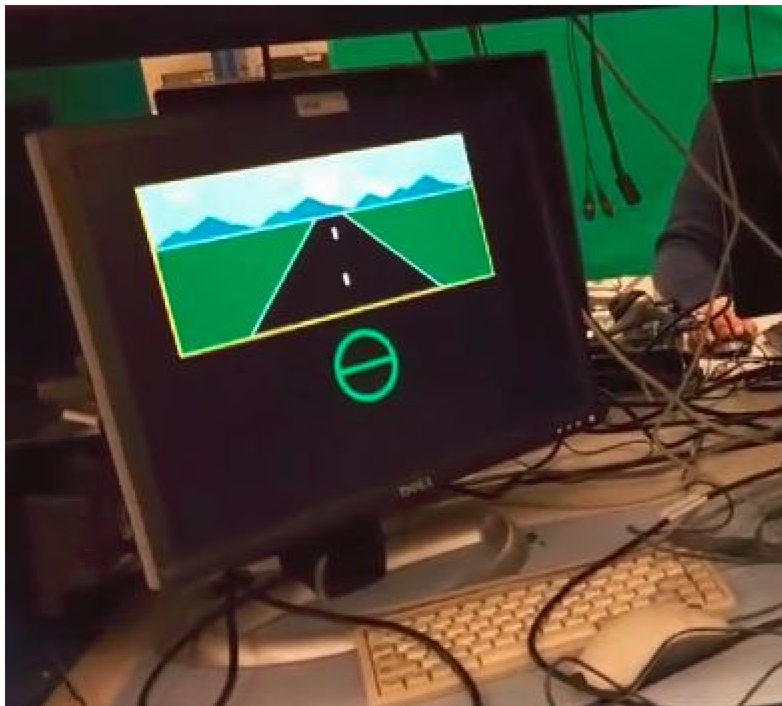
Motion Controlled Driving Game

Venita Boodhoo, Janie Liu, Jing Wang



6.111 Final Project Report

Table of Contents

1. **Overview** 3
2. **Design** 3
    a. **Block diagram** 3
    b. **Steering wheel detection** 4
        i. **Color video display (Jing)** 4
        ii. **Image processing (Jing)** 4
    c. **Game functionality** 5
        i. **Brakes and acceleration (Venita)** 5
        ii. **Game logic (Venita)** 5
    d. **Graphics** 6
        i. **Dashboard (Jing and Venita)** 6
        ii. **Road (Jing)** 7
        iii. **Markers (Venita)** 7
        iv. **Sky (Janie)** 7
    e. **Audio (Janie)** 8
3. **Integration** 8
4. **Challenges** 8
5. **Testing** 9
6. **Timeline** 9
7. **Stretch Goals** 9
8. **Advice** 10
9. **Conclusion** 10
10. **Appendix** 11

1. Overview

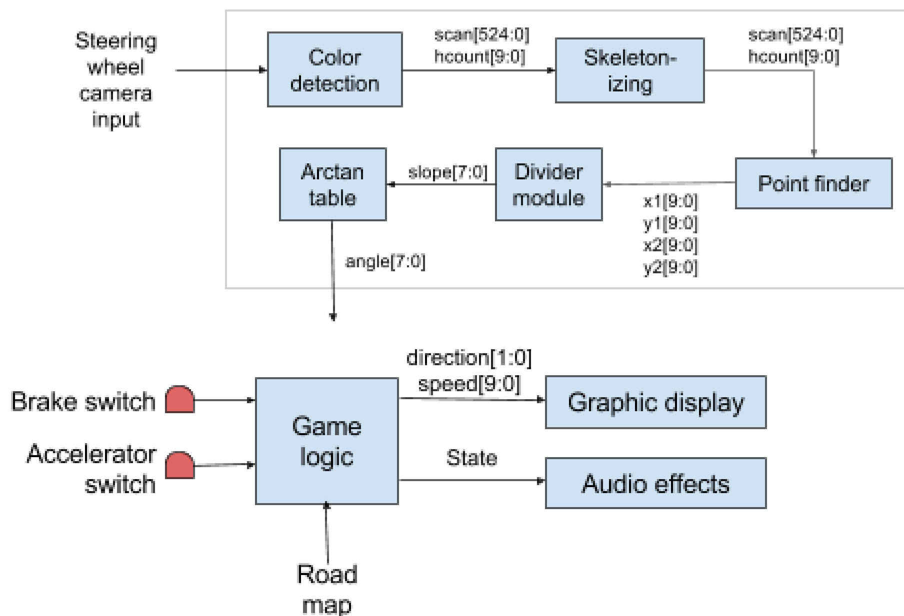
"You are a tired computer engineer coming home from work. Due to extreme sleep deprivation, the world around you looks strange…"

We would like to design a game that uses motion tracking, graphics, and audio effects. The game is a augmented reality driving game that will be displayed using graphics generated by the FPGA onto the computer monitor. The user is inside the car and can control how the car moves by holding out their two hands and moving them around as if they are holding the steering wheel. There will also be brakes and acceleration pedals for the feet. We can use a VGA camera or an accelerometer to track the movements of the hands and feet. The goal of the game is to reach the final destination without hitting any objects on the road. Sound effects will be played when you crash. Game logic may include assessing how well the driver drove i.e. not crashing into many obstacles.

2. Design

A. Block Diagram

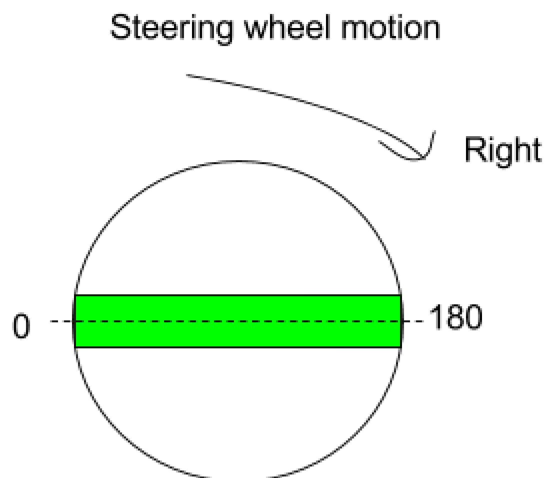
Our project can be divided into four main parts: steering wheel detection, game logic, graphic display, and audio output. The block diagram for the system is shown below.



B. Steering wheel

In this game, the driver can control how the car in the display moves along the road by turning a physical steering wheel. The steering wheel will be built by cutting out a plastic or wooden circle and adding a rectangular bar across the middle, which will be of a different color from the circle. As the driver turns the wheel, the colored rectangle will change its angle, which will provide the angle of rotation. The video camera will be aimed at the wheel as the driver plays to provide motion information in real time.

If the colored bar is horizontal, then the angle of the perpendicular to the bar would be 90° and the resulting direction of the car should be straight. The angle from the horizontal would determine how much the car would deviate from driving straight on the one-lane road. We are restricting the angles from -90° to 90° from the steering wheel to range across the direction on the road. Thus, it is not possible for the car to reverse or completely turn around.



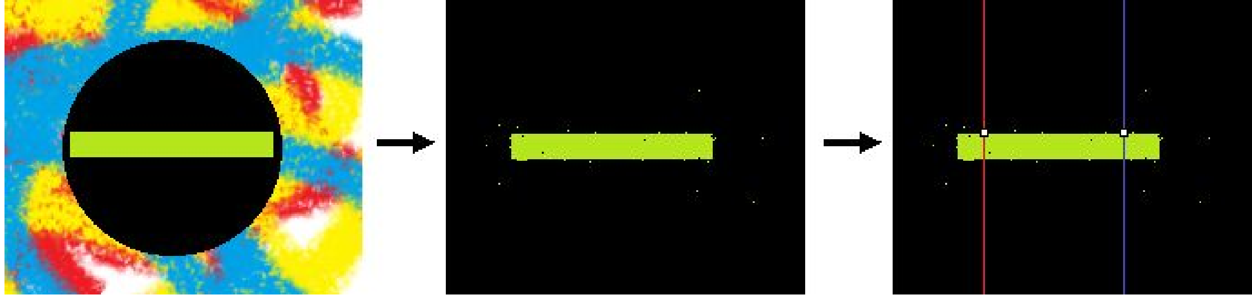
(i) Color video display (Jing)

In order to track the movement of a steering wheel, color image data was displayed by the FPGA. The video inputs were stored in ZBT memory and then converted into color values for each pixel and displayed onto the monitor.

We also tried transforming the RGB values into HSV color values, but ultimately decided to use RGB values because there was more noise with HSV, likely due to either randomness in hues for very black or very white pixels, or mathematical errors when performing the transformation.

(ii) Image processing (Jing)

Once the video input is displayed through the FPGA, we can isolate the green bar of the steering wheel by scanning through each pixel and setting it to black if it does not lie within the valid RGB range, as demonstrated in the pictures below.



From these green pixels, we scan down each row and record the first green pixel seen along two lines, one on the left and one on the right. Depending on whether the left pixel is higher, lower, or around the same height as the right pixel, we can determine whether the driver is moving right, left, or straight. This 2-bit direction value is then passed along into the game logic.

C (i). Brakes and acceleration (Venita)

For the brake and acceleration, we used the up and down push buttons - up for acceleration and down for deceleration. The change in speed is constant so long as the switches are pressed - if the driver steps on the brake, the car will decelerate at a constant rate (and vice versa for the accelerator) but continue moving at a constant speed if the switch is released. The output values from these switches (0 or 1 for on or off) will be sent into the Verilog game logic.

C (ii). Game logic (Venita)

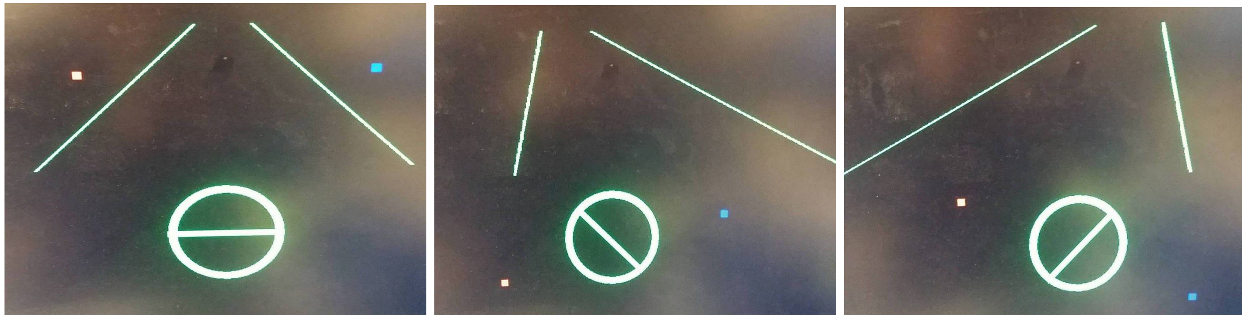
There are 4 inputs that will determine what it is displayed on the screen. These are the brake switch, accelerator switch, road map and steering wheel motion. The switches' values are parameters that the state of the game, a register can have. If the brake switch is on, then the car gradually slows down on the screen. If the accelerator switch is on, the car speeds up. Just as with a real car, one cannot press both the accelerator and brake at the same time so this is an invalid state for our system so we have a default case of the car moving at a constant slow speed. The steering wheel input determines the car's direction: left, right or straight. If the car is at a non-zero speed and the steering wheel does not change direction to the direction of the road in a few seconds, then the car has crashed and the game ends.

These states can be represented as summarized as:

Accelerator Switch	Brake Switch	Crashed	Motion of car on screen
OFF	OFF	NO	Constant speed

OFF	ON	NO	Decelerating
ON	OFF	NO	Accelerating
X	X	YES	Crashed (no motion)

The roadmap is comprised of 4 parts where each part is defined by the user through the buttons 0-2. Each button press represents a particular type of road - button 0 is a straight road, button 1 is a left road and button 2 is a right road. These values are also converted to binary and displayed on the leds so that the user can remember what type of road they created. As previously mentioned, a car is considered to have driven off the road if its direction does not correspond to a bend in the road. For example, if we consider the first image below, the road is straight and if the steering wheel turns right or left, then the car will crash off of the road.



Straight Left Right

The game logic module will then send to the graphics module 3 pieces of information: the state of the game which includes information of the car's motion and if it has crashed, a map of the road and the speed of the car.

D. Graphics

i) Dashboard (Jing and Venita)

A vertical red bar served as a speedometer for the user. It was displayed on the screen so that the user would have an indication of how fast they were going on the road. When the game begins, the speed is zero and there is no red vertical bar. When the user accelerates by pressing the up push button, the red bar increases in height to a maximum height with each button press. When the user decelerates, the red bar decreases until it disappears.

To display the steering wheel, we drew a circle using the basic circle equation. Because of the amount of time needed to perform the multiplication, we pipelined this module so that it would draw properly. With direction as input, three different bars were

drawn through the center of the steering wheel to indicate different turnings - if the player turned their physical steering wheel to the left, then the graphical wheel would also turn left.

ii) Road (Jing)

To display straight, left, and right portions of the road, we need to be able to draw lines of different slopes that approach each other toward the horizon to give the illusion of distance. We made simple sketches of what the road would look like in each state, and then calculated an equation for every line in terms of the variables hcount and vcount.

iii) Markers (Venita)

Besides the speedometer, there needed to be some other sort of indication of speed for the user and dashes were chosen to simulate the dashes seen on roads in real life. Initially, the road is still so the markers do not move. As speed increases, the markers would move more quickly downwards on the screen and their speed would slow down if the user decelerates.

iv) Sky (Janie)

The most exciting moment was when the sky picture actually showed up on the screen.

(The next exciting moment was when it showed up correctly).

Beforehand, process images in MATLAB and store them in BROM

- Sky:
    - Find picture (minimalist, flat picture preferred)
    - Reduce to 16 color indexed picture with GIMP
    - Export as a BMP
    - Fire up the MATLAB script provided online
    - Create a COE file from the BMP
    - Create three more COE files for the color maps (one red, green, blue)
    - Create block memories in verilog, and initialize them with these COE files

  The process in the end was not difficult. However, understanding the color tables through the online powerpoints took a while. I think showing an example of a bitmap file, and showing how it maps to different colors, would be the clearest way to show how this worked.

  Asking a TA/Instructor to explain is also great, because it’s a simple, but clever concept.

E. Audio (Janie)

Input: Takes in the speed
Output: Beeps that get louder and faster when you go faster, and quieter and slower when you drive more slowly.

Originally, this was intended to be a more complex implementation. Due to personal events, there was not a lot of time to implement it. The current implementation largely uses the tone generator from Lab 5. It plays the tone, followed by silence, the AC97 headphones.

If I had more time, I would have liked using Lab 5's recorder functionality to record certain sounds to places in memory (in block memory). There would be a separate block memory for each sound. The choice of what memory to write to, or read from, would be determined from a state input.

Even though this would not create a realistic sound profile (the most realistic one would use a lot of lookup tables), it would at least create a flexible and interesting playground for sound.

In the future, I would try the simplest implementation first, and then move onto more complicated implementations. I did not know enough to break the task down into smaller parts at the time, but doing a basic implementation of tone generative and sending it to the ac97 would have helped acclimate me to the sound modules.

3. Integration

We first integrated the steering wheel detection module and its accompanying graphic with the game logic and the speedometer. There was some timing issues with figuring out a good amount of time needed before the user would crash but this was resolved with a few testing runs. There was also some random lines of color but this was due to us not initially clocking some of the drawing modules. When we had the motion and the direction of the car, we added the markers on the road and then, the sky for a nice view. Finally, the audio effect was added to create a more realistic user experience.

4. Challenges

Most of the project will not require complex arithmetic with the exception of drawing the road and the dashboard, which will require multiplication to calculate the correct slope or radius. Therefore, pipelining is required for the modules that draw the road and circles. Within the game logic, it took longer than expected to have the user defined roadmap via the buttons because it was easy to not remember that pressing down the button could actually be multiple button

presses. As described in earlier sections, flash memory will be used to store the graphics and sound effects for the game.

5. Testing

Initially, the individual modules were unit tested to ensure that they work independently:

- To test the steering wheel, we displayed different shapes on the screen depending on what direction was passed in and checked that the program correctly recognizes the angle that the wheel is turned.
- To test the game logic, values were hard coded to pass into the module and the direction of the user, the direction of the road and the state of the car were all displayed on the display leds
- To test the graphic outputs, we manually inputted data about the speed and angle of the car.
- To test the audio output, we manually inputted data about the state of the game.

Once we ensure that the individual modules functioned, we integrated them together one at a time and test that they work together as a whole as described in 3.

6. Timeline

11/7-11/9: Project presentation

11/19: Implement modules with base level performance

11/26: Unit testing and continued work on individual modules

12/03: Integration of modules

12/06: Debugging

12/11 Project checkoff

7. Stretch Goals

- Allowing the user to choose between a horse, bike or car
- Allowing the user to change a road while driving
- Fuel light indicator on the dashboard
- Speed limit warning indicator
- Reversing the vehicle
- Obstacles in the road

8. Advice

Janie:

- Start early, even if it's just a tiny task

Venita:

- Always keep backup copies of your last working code
- When all else fails, restart the computer

Jing:

- Pipelining is your friend
- Taking the time to calculate the math saves time over just guessing

9. Conclusion

Ultimately, we were able to reach our commitment goals and successfully create a simple driving game that had left, right, and straight portions on the road. We were able to detect a physical steering wheel and allow a player to “drive” through a displayed graphical road that had markers to indicate speed as well as a dashboard that showed a speedometer and wheel. If a driver didn't turn in time, a crash screen was displayed. Different sound effects were produced upon changing speeds or crashing.

Although the project ended up being more complex than we originally imagined, working on the motion controlled driving game taught us many lessons about design, pipelining, drawing graphics, and state machines, as well as team skills such as communication and time management. We would like to thank Professor Gim Hom and the TAs for guiding us through an exciting semester of digital electronics.

Appendix (Verilog source file)

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

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

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

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

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

               clock_feedback_out, clock_feedback_in,

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

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
```

```
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input

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

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

   // SRAMs

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

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

/* enable RAM pins */

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

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

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;

   //These values has to be set to 0 like ram0 if ram1 is used.
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   // clock_feedback_out will be assigned by ramclock
   // assign clock_feedback_out = 1'b0;  //2011-Nov-10
```

```
   // clock_feedback_in is an input

   // 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 2
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

   wire clk = clock_40mhz;
*/
	wire locked;
	//assign clock_feedback_out = 0; // gph 2011-Nov-10

   ramclock rc(.ref_clock(clock_65mhz), .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;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(.reset(power_on_reset),.clk(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
   assign reset = user_reset | power_on_reset;

	// direction and speeds inputs
   wire up,down;
	wire straight,left,right;
   debounce db2(.reset(reset),.clk(clock_65mhz),.noisy(~button_up),.clean(up));
   debounce db3(.reset(reset),.clk(clock_65mhz),.noisy(~button_down),.clean(down));
	debounce db4(.reset(reset),.clk(clock_65mhz),.noisy(~button0),.clean(straight));
	debounce db5(.reset(reset),.clk(clock_65mhz),.noisy(~button1),.clean(left));
	debounce db6(.reset(reset),.clk(clock_65mhz),.noisy(~button2),.clean(right));

   // display module for debugging

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

   // generate basic XVGA video signals
   wire [10:0] hcount;
```

```
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

// wire up to ZBT ram

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

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

// generate pixel value from reading ZBT memory
wire [17:0]         vr_pixel; // Was 8 bits
wire [18:0]         vram_addr1;

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

      // neighbor pixels
      //wire [17:0]         vr_pixel_up, vr_pixel_down, vr_pixel_left, vr_pixel_right;
      //wire [18:0]         vram_addr_up, vram_addr_down, vram_addr_left, vram_addr_right;
      //vram_display vd_up(reset,clk,hcount,vcount+1,vr_pixel_up,
      //            vram_addr_up,vram_read_data);
      //vram_display vd_down(reset,clk,hcount,vcount-1,vr_pixel_down,
      //            vram_addr_down,vram_read_data);
      //vram_display vd_left(reset,clk,hcount+1,vcount,vr_pixel_left,
      //            vram_addr_left,vram_read_data);
      //vram_display vd_right(reset,clk,hcount-1,vcount,vr_pixel_right,
      //            vram_addr_right,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));

      // Convert from YcRcB to RGB
      wire [7:0] red, green, blue;
      YCrCb2RGB toRGB( red, green, blue, tv_in_line_clock1, reset,
                                    ycrcb[29:20], ycrcb[19:10], ycrcb[9:0] );

      // Convert RGB to HSV
      //wire [7:0] hue, sat, val;
      //rgb2hsv toHSV(tv_in_line_clock1, reset, red, green, blue, hue, sat, val);

      // Color filter
      // If color is green, let through; else, display black.
      reg [7:0] red2, green2, blue2;
      reg [7:0] red3, green3;
      reg [7:0] prev_green;
```

```
		always @(posedge clk) begin
			prev_green <= green2;
			//Color filtering
			if (green > red && green > blue
					&& green > 8'h5F
					&& red < 8'hCF
					&& blue < 8'hCF) begin
				red2 <= 8'h00;
				green2 <= 8'hFF;
				blue2 <= 8'h00;
			end else begin
				red2 <= 8'h00;
				green2 <= 8'h00;
				blue2 <= 8'h00;
			end

			// Skeletonization
			if (green2 == 8'hFF)
				if (0/*(vr_pixel_up[11:6] == 8'h00) || (vr_pixel_down[11:6] == 8'h00)
				  || (vr_pixel_left[11:6] == 8'h00) || (vr_pixel_right[11:6] == 8'h00)*/)
					green3 <= 8'h00;
				else
					green3 <= 8'hFF;
			else
				green3 <= 9'h00;
		end


		reg signed [15:0] x1, y1, x2, y2;
//		// code for getting angle
//		always @(posedge clk) begin
//			if (green2 == 8'hFF && vcount > y2 && hcount > x2) begin
//				x2 <= hcount;
//				red3 <= 8'hFF;
//			end else if (green2 == 8'hFF && vcount < y1 && hcount > x1) begin
//				x1 <= hcount;
//				red3 <= 8'hFF;
//			end else
//				red3 <= 8'h00;
//		end

		// TESTING PURPOSES
		wire [23:0] left_pixel, right_pixel, left_point, right_point;
		blob #(.HEIGHT(500), .WIDTH(5), .COLOR(24'hFF_FF_FF))		// white
			left_border(.x(200),.y(70),.hcount(hcount),.vcount(vcount),
							.pixel(left_pixel));
		blob #(.HEIGHT(500), .WIDTH(5), .COLOR(24'hFF_FF_FF))		// white
			right_border(.x(550),.y(70),.hcount(hcount),.vcount(vcount),
							.pixel(right_pixel));

		blob #(.HEIGHT(10), .WIDTH(10), .COLOR(24'hFF_FF_FF))		// white
			point1(.x(200),.y(y1),.hcount(hcount),.vcount(vcount),
							.pixel(left_point));
		blob #(.HEIGHT(10), .WIDTH(10), .COLOR(24'hFF_FF_FF))		// white
			point2(.x(550),.y(y2),.hcount(hcount),.vcount(vcount),
							.pixel(right_point));

   // code to write NTSC data to video memory

   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {red2[7:2], green3[7:2], blue2[7:2]},
			 ntsc_addr, ntsc_data, ntsc_we, switch[6]);

   // code to write pattern to ZBT memory
```

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

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

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

   wire      sw_ntsc = ~switch[7];
   wire      my_we = sw_ntsc ? (hcount[0]==1'b1) : 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;
//   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 [17:0]            pixel; // Was 8 bits
   reg       b,hs,vs;

         //reg [15:0] y1, y2, slope;
         reg left_found, right_found;
         reg [22:0] frame_count;

         reg signed [15:0] y_diff, x_diff;
         reg pos_slope;
         reg [1:0] direction;

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

                 // GETS THE ENDPOINTS
                 if (frame_count == 0) begin
                         left_found <= 0;
                         right_found <= 0;
                 end else   if (!left_found && left_pixel[17:10]>0 && vr_pixel[11:6]>0 && vcount > 100) begin
                         y1 <= vcount;
                         left_found <= 1;
                 end else if (!right_found && right_pixel[17:10]>0 && vr_pixel[11:6]>0 && vcount > 100) begin
                         y2 <= vcount;
                         right_found <= 1;
                 end
                 frame_count <= frame_count + 1;
                 if (y2 >= y1) begin
                         y_diff <= y2 - y1;
                         pos_slope <= 1;
                 end else begin
                         y_diff <= y1 - y2;
                         pos_slope <= 0;
                 end
                 x_diff <= x2 - x1;

                 // p1 = (x1, y1) x1 = 200
                 // p2 = (x2, y2) x2 = 550
                 // Turn left if y2>y1, right if y1>y2
```

```
			if (y2 < y1 + 15 && y2 > y1 - 15)
				direction <= 2'b00;
			else if (y2 < y1)
				direction <= 2'b10;
			else
				direction <= 2'b01;

end


		wire [15:0] slope;
		divider slope_calc(
			.clk(clk),
			.dividend(y_diff),
			.divisor(x_diff),
			.quotient(slope)
			);

		//wire [23:0] steering_wheel, steer_circle, road_pixel, dash_pixel, horizon_pixel;
		//wheel_bar steer_bar(.hcount(hcount),.vcount(vcount),.dir(direction),.pixel(steering_wheel));
		//circle circle1(.hcount(hcount),.vcount(vcount),.pixel(steer_circle));
		//road road1(.hcount(hcount),.vcount(vcount),.dir(direction),.pixel(road_pixel));
		//board board1(.hcount(hcount),.vcount(vcount),.pixel(dash_pixel));
		//horizon horizon1(.hcount(hcount),.vcount(vcount),.pixel(horizon_pixel));

		wire [23:0] pixel1;
		wire phsync,pvsync,pblank;
		reg [23:0] rgb;
		reg [7:0] roadmap = 8'b0;
		reg [2:0] i;
		reg [1:0] last_straight;
		reg [1:0] last_left;
		reg [1:0] last_right;
		reg straight_pressed;
		reg left_pressed;
		reg right_pressed;

		//Roadmap logic
always @(posedge clock_65mhz)
			begin
				last_straight <= straight;
				last_left <= left;
				last_right <= right;

				if (straight && !last_straight)
					straight_pressed <= 1;
				if (left && !last_left)
					left_pressed <= 1;
				if (right && !last_right)
					right_pressed <= 1;
				if (i != 3'd4)
					begin
					 if (straight_pressed)
						begin
							roadmap <= {roadmap[5:0],2'b00};
							straight_pressed <= 0;
						end
					 else if(left_pressed)
						begin
							roadmap <= {roadmap[5:0],2'b01};
							left_pressed <= 0;
						end
					 else if (right_pressed)
						begin
							roadmap <= {roadmap[5:0],2'b10};
							right_pressed <= 0;
```

```
                                        end
                                i <= i + 1;
                                end
                        else
                                i <= 0;

                        // default: driving game
                         hs <= phsync;
                         vs <= pvsync;
                         b <= pblank;
                         rgb <= pixel1;
                end


        //Outputs
        wire [1:0] state;
        wire [1:0] road_direction;
        /////////////////////////////
        //Driving Game
        /////////////////////////////
        driving_game game_logic(.vclock(clock_65mhz),.reset(reset),
        .acceleration(up),.brake(down),.roadmap(roadmap),.direction(direction),.pspeed(switch[7:4]),
                .hcount(hcount),.vcount(vcount),
        .hsync(hsync),.vsync(vsync),.blank(blank),
                .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel1),.state(state),.road_direction(road_direction));


   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clk.

        /*
   //assign vga_out_red = {vr_pixel[17:12], 2'b0} + left_pixel[17:10] + left_point[17:10];
        assign vga_out_red = road_pixel[23:16] + dash_pixel[23:16] + horizon_pixel[23:16]; // + left_point[17:10];
   //assign vga_out_green = {vr_pixel[11:6], 2'b0};
        assign   vga_out_green   =   steering_wheel[15:8]   +   road_pixel[15:8]   +   steer_circle[15:8]   +   dash_pixel[15:8]   +
horizon_pixel[17:10];
   //assign vga_out_blue = {vr_pixel[5:0], 2'b0} + right_pixel[17:10] + right_point[17:10];
        assign vga_out_blue = road_pixel[7:0] + horizon_pixel[7:0]; // + right_point[17:10];

   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clk;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
*/
        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 = ~roadmap;
   // debugging

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

        wire [63:0] my_hex_data = {52'b0,2'b0,road_direction, 2'b0,direction,2'b0,state};

        display_16hex disp(reset, clock_27mhz, my_hex_data,disp_blank, disp_clock, disp_rs, disp_ce_b,

        disp_reset_b, disp_data_out);

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

```
    ////dispdata <= {ntsc_data,9'b0,ntsc_addr};

endmodule

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

module driving_game (
   input vclock,        // 65MHz clock
   input reset,                 // 1 to initialize module
   input acceleration,          // 1 when car is accelerating
   input brake,         // 1 when car is decelerating
          input [7:0] roadmap,
          input [1:0] direction,
   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 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
          output reg [1:0] state,
          output reg [1:0] road_direction
   );

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

          //COLORS
          parameter WHITE    = 24'hFF_FF_FF;
          parameter RED = 24'hFF_00_00;
          parameter GREEN = 24'h00_FF_00;
          parameter YELLOW = 24'hffff00;
          parameter BLACK = 24'h0;
          parameter BLUE = 24'h00_00_7F;
          parameter GRASS_COLOR = 24'h00_7F_00;

          ////SPEED
          parameter ACCELERATING = 2'b10;
   parameter DECELERATING =  2'b01;
   parameter CONSTANT_SPEED = 2'b00;
          parameter CRASH = 2'b11;
          parameter RESTART = 3'b100;

          //LEVEL
          parameter max_level_y = 10'd450;
          reg [9:0]  level_y;
          wire [23:0] level_pixel;

          //blob #(.WIDTH(16),.HEIGHT(128),.COLOR(RED))   // red level, x position is fixed
          //              level(.x(11'd100),.y(level_y),.hcount(hcount),.vcount(vcount),.marker(1'b1),.pixel(level_pixel));
          speed_level #(.COLOR(RED)) how_fast_am_i_going(.hcount(hcount),.vcount(vcount),.level_y(level_y),.pixel(level_pixel));

          //OBJECTS

          reg [9:0]  object_y;
          reg [9:0] delta_y;
          wire [23:0] straight_object_pixel;
          wire [23:0] straight_object_pixel2;
          wire [23:0] straight_object_pixel3;
```

```
	wire [23:0] straight_object_pixel4;
	wire [23:0] straight_object_pixel5;
	wire [23:0] straight_object_pixel6;
	wire [23:0] straight_object_pixel7;
	wire [23:0] straight_object_pixel8;
	wire [23:0] straight_object_pixel9;
	wire [23:0] straight_object_pixel10;

	//BLOB STRAIGHT PARAMETERS
	parameter DASH_WIDTH =    8;
	parameter DASH_HEIGHT = 20;
	parameter STRAIGHT_X = 11'd475;

	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object1(.x(STRAIGHT_X),.y(object_y),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object2(.x(STRAIGHT_X),.y(object_y+130),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel2));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object3(.x(STRAIGHT_X),.y(object_y+228),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel3));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object4(.x(STRAIGHT_X),.y(object_y+326),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel4));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object5(.x(STRAIGHT_X),.y(object_y+424),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel5));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object6(.x(STRAIGHT_X),.y(object_y+522),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel6));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object7(.x(STRAIGHT_X),.y(object_y+620),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel7));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object8(.x(STRAIGHT_X),.y(object_y+718),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel8));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object9(.x(STRAIGHT_X),.y(object_y+816),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel9));
	blob #(.WIDTH(DASH_WIDTH),.HEIGHT(DASH_HEIGHT),.COLOR(WHITE))   // white object

object10(.x(STRAIGHT_X),.y(object_y+914),.hcount(hcount),.vcount(vcount),.marker(1'b0),.pixel(straight_object_pixel10));

	//DIRECTIONS
	reg [10:0] dir_y;
	//LEFT
	wire [23:0] left_object_pixel;
	wire [23:0] left_object_pixel2;
	wire [23:0] left_object_pixel3;
	wire [23:0] left_object_pixel4;
	wire [23:0] left_object_pixel5;
	wire [23:0] left_object_pixel6;
	wire [23:0] left_object_pixel7;
	wire [23:0] left_object_pixel8;
	wire [23:0] left_object_pixel9;
	wire [23:0] left_object_pixel10;
	wire [23:0] left_object_pixel11;
	wire [23:0] left_object_pixel12;
	wire [23:0] left_object_pixel13;
	wire [23:0] left_object_pixel14;
	wire [23:0] left_object_pixel15;
	wire [23:0] left_object_pixel16;
	wire [23:0] left_object_pixel17;
	wire [23:0] left_object_pixel18;
	wire [23:0] left_object_pixel19;
```

```
wire [23:0] left_object_pixel20;

dash  #(.COLOR(WHITE))
        object1_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y),.pixel(left_object_pixel));
dash  #(.COLOR(WHITE))
        object2_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+130),.pixel(left_object_pixel2));
dash  #(.COLOR(WHITE))
        object3_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+228),.pixel(left_object_pixel3));
dash  #(.COLOR(WHITE))
        object4_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+326),.pixel(left_object_pixel4));
dash  #(.COLOR(WHITE))
        object5_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+424),.pixel(left_object_pixel5));
dash  #(.COLOR(WHITE))
        object6_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+522),.pixel(left_object_pixel6));
dash  #(.COLOR(WHITE))
        object7_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+620),.pixel(left_object_pixel7));
dash  #(.COLOR(WHITE))
        object8_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+718),.pixel(left_object_pixel8));
dash  #(.COLOR(WHITE))
        object9_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+816),.pixel(left_object_pixel9));
dash  #(.COLOR(WHITE))
        object10_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+914),.pixel(left_object_pixel10));
dash  #(.COLOR(WHITE))
        object11_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1012),.pixel(left_object_pixel11));
dash  #(.COLOR(WHITE))
        object12_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1110),.pixel(left_object_pixel12));
dash  #(.COLOR(WHITE))
        object13_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1208),.pixel(left_object_pixel13));
dash  #(.COLOR(WHITE))
        object14_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1306),.pixel(left_object_pixel14));
dash  #(.COLOR(WHITE))
        object15_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1404),.pixel(left_object_pixel15));
dash  #(.COLOR(WHITE))
        object16_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1502),.pixel(left_object_pixel16));
dash  #(.COLOR(WHITE))
        object17_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1600),.pixel(left_object_pixel17));
dash  #(.COLOR(WHITE))
        object18_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1698),.pixel(left_object_pixel18));
dash  #(.COLOR(WHITE))
        object19_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1796),.pixel(left_object_pixel19));
dash  #(.COLOR(WHITE))
        object20_left(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1894),.pixel(left_object_pixel20));

//RIGHT
wire [23:0] right_object_pixel;
wire [23:0] right_object_pixel2;
wire [23:0] right_object_pixel3;
wire [23:0] right_object_pixel4;
wire [23:0] right_object_pixel5;
wire [23:0] right_object_pixel6;
wire [23:0] right_object_pixel7;
wire [23:0] right_object_pixel8;
wire [23:0] right_object_pixel9;
wire [23:0] right_object_pixel10;
wire [23:0] right_object_pixel11;
wire [23:0] right_object_pixel12;
wire [23:0] right_object_pixel13;
wire [23:0] right_object_pixel14;
wire [23:0] right_object_pixel15;
wire [23:0] right_object_pixel16;
wire [23:0] right_object_pixel17;
wire [23:0] right_object_pixel18;
wire [23:0] right_object_pixel19;
wire [23:0] right_object_pixel20;

dash  #(.COLOR(WHITE))
```

```
		object1_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y),.pixel(right_object_pixel));
	dash #(.COLOR(WHITE))
		object2_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+130),.pixel(right_object_pixel2));
	dash #(.COLOR(WHITE))
		object3_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+228),.pixel(right_object_pixel3));
	dash #(.COLOR(WHITE))
		object4_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+326),.pixel(right_object_pixel4));
	dash #(.COLOR(WHITE))
		object5_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+424),.pixel(right_object_pixel5));
	dash #(.COLOR(WHITE))
		object6_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+522),.pixel(right_object_pixel6));
	dash #(.COLOR(WHITE))
		object7_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+620),.pixel(right_object_pixel7));
	dash #(.COLOR(WHITE))
		object8_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+718),.pixel(right_object_pixel8));
	dash #(.COLOR(WHITE))
		object9_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+816),.pixel(right_object_pixel9));
	dash #(.COLOR(WHITE))
		object10_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+914),.pixel(right_object_pixel10));
	dash #(.COLOR(WHITE))
		object11_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1012),.pixel(right_object_pixel11));
	dash #(.COLOR(WHITE))
		object12_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1110),.pixel(right_object_pixel12));
	dash #(.COLOR(WHITE))
		object13_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1208),.pixel(right_object_pixel13));
	dash #(.COLOR(WHITE))
		object14_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1306),.pixel(right_object_pixel14));
	dash #(.COLOR(WHITE))
		object15_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1404),.pixel(right_object_pixel15));
	dash #(.COLOR(WHITE))
		object16_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1502),.pixel(right_object_pixel16));
	dash #(.COLOR(WHITE))
		object17_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1600),.pixel(right_object_pixel17));
	dash #(.COLOR(WHITE))
		object18_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1698),.pixel(right_object_pixel18));
	dash #(.COLOR(WHITE))
		object19_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1796),.pixel(right_object_pixel19));
	dash #(.COLOR(WHITE))
		object20_right(.hcount(hcount),.vcount(vcount),.dir(road_direction),.x(dir_y+1894),.pixel(right_object_pixel20));

	//ROAD
	wire [23:0] road_pixel;
	road #(.ROAD_COLOR(WHITE))
			road1
(.hcount(hcount),.vcount(vcount),.clock(vclock),.crash(state),.dir(road_direction),.pixel(road_pixel));

	reg [23:0] object_pixel;
	wire [23:0] straight_object_pixels;
	wire [23:0] left_object_pixels;
	wire [23:0] right_object_pixels;

	assign straight_object_pixels = (straight_object_pixel | straight_object_pixel2 | straight_object_pixel3 |
straight_object_pixel4 |

straight_object_pixel5 | straight_object_pixel6 | straight_object_pixel7 | straight_object_pixel8 |

straight_object_pixel9 | straight_object_pixel10);

	assign left_object_pixels = (left_object_pixel | left_object_pixel2 | left_object_pixel3 | left_object_pixel4 |

left_object_pixel5 | left_object_pixel6 | left_object_pixel7 | left_object_pixel8 |

left_object_pixel9 | left_object_pixel10 | left_object_pixel11 | left_object_pixel12 |

left_object_pixel13 | left_object_pixel14 | left_object_pixel15 | left_object_pixel16 |
```

```
left_object_pixel17 | left_object_pixel18 | left_object_pixel19 | left_object_pixel20);

        assign right_object_pixels = (right_object_pixel| right_object_pixel2 | right_object_pixel3 | right_object_pixel4 |

right_object_pixel5 | right_object_pixel6 | right_object_pixel7 | right_object_pixel8 |

right_object_pixel9 | right_object_pixel10 | | right_object_pixel11 | | right_object_pixel12 |

right_object_pixel13 | right_object_pixel14 | right_object_pixel15 | right_object_pixel16 |

right_object_pixel17 | right_object_pixel18 | right_object_pixel19 | right_object_pixel20);


        always @ (posedge vclock)
                begin
                if (road_direction == 2'b00)
                        object_pixel = straight_object_pixels;
                else if (road_direction == 2'b01)
                        object_pixel = left_object_pixels;
                else
                        object_pixel = right_object_pixels;
                end

        //WHEEL
        wire [23:0] wheel_pixel;
        wheel_bar #(.COLOR(GREEN))  // default color: white
                wheel(.hcount(hcount),.vcount(vcount),.dir(direction),.pixel(wheel_pixel));

        //HORIZON
        wire [23:0] horizon_pixel;
        horizon #(.COLOR(WHITE))  // default color: white
                horizon1(.hcount(hcount),.vcount(vcount),.pixel(horizon_pixel));

        //BOARD
        wire [23:0] board_pixel;
        board #(.COLOR(YELLOW))  // default color: yellow
                board1(.hcount(hcount),.vcount(vcount),.pixel(board_pixel));

        //CIRCLE
        wire [23:0] circle_pixel;
        circle #(.COLOR(GREEN))  // default color: green
                circle1(.hcount(hcount),.vcount(vcount),.clock(vclock),.pixel(circle_pixel));

        //SKY
        wire [23:0]sky_pixel;
        wire [10:0] sky_x = 11'd100;
        wire [9:0] sky_y = 10'd84;
        sky_blob #(.WIDTH(750),.HEIGHT(113))
                sky(.pixel_clock(vclock),.x(sky_x),.hcount(hcount),.crash(state),.y(sky_y),.vcount(vcount),.pixel(sky_pixel));

        //SCREEN
        wire [23:0] pixel_color;
        wire black_pixel;
        wire change_to_red;
        assign pixel_color = ( level_pixel | road_pixel | object_pixel | wheel_pixel | horizon_pixel |
                                                board_pixel | circle_pixel | sky_pixel );

        assign black_pixel = ((pixel_color == BLACK | pixel_color == BLUE ))? 1: 0;
        assign change_to_red = ((black_pixel) & state == CRASH);

        assign pixel = change_to_red ? RED : pixel_color ;


        //DIRECTION
        reg [3:0] i;
```

```
//Button press
reg [6:0] count;

//CRASH LOGIC
reg [31:0] time_counter = 0;
parameter LIMIT = 100_000_000;

//Clock time
always @(posedge vclock)
        begin
                if (reset)
                        begin
                                level_y <= 600;
                                object_y <= 0;
                                delta_y <= 0;
                                dir_y <= 0;
                                road_direction <= 0;
                                state <= CONSTANT_SPEED;
                                count <= 0;
                                time_counter <= 0;
                        end
                else
                        begin

                                if (time_counter < LIMIT)
                                        time_counter <= time_counter + 1;
                                else if (time_counter == LIMIT)
                                         begin
                                                time_counter <= 0;
                                                if (i != 4'd7)
                                                        begin
                                                                road_direction                              <=
{roadmap[i+1],roadmap[i]};
                                                                if              (({direction[1:0]}              !=
{road_direction[1:0]}) && level_y != 600)
                                                                        state <= CRASH;
                                                         //Check next part of road
                                                         i <= i + 2;
                                                        end
                                                end
                                                else
                                                        //Reset i
                                                        i  <= 0;

                                if (hcount == 1023 & vcount == 600)   //refresh every frame/pixel
                                        begin
                                        //////////////
                                        //STATE LOGIC
                                        ////////////
                                         case(state)
                                                 ACCELERATING:
                                                                begin

                                                                        if (~acceleration)
                                                                                begin
                                                                                        count   <=
0;
                                                                                        if
(~brake)

state <= CONSTANT_SPEED;
                                                                                        else

state <= DECELERATING;
                                                                                end
```

```
				//Visuals
				else if (level_y != max_level_y)
					level_y <= level_y - 1;

				count <= count + 1;
				if (count == 7'd128)
					begin
						if (level_y != max_level_y)
							begin
								delta_y <= delta_y+1;
							end
						object_y <= object_y + delta_y;
						dir_y <= dir_y + delta_y;
					end
				else
					count <= 0;
			end

		DECELERATING:
			begin
				if (~brake)
					begin
					count <= 0;
						if (~acceleration)
							state <= CONSTANT_SPEED;
						else
							state <= ACCELERATING;
					end
				//Visuals
				else if (level_y >= 600)
					begin
					delta_y <= 0;
					state <= CONSTANT_SPEED;
					end
				else
					level_y <= level_y + 1;
				//
				count <= count + 1;
				if (count == 7'd128)
					begin
					if (delta_y >= 1 && level_y < 600)
						delta_y <= delta_y-1;
					object_y <= object_y + delta_y;
					dir_y <= dir_y + delta_y;
					end
				else
					count <= 0;
			end
```

```
                                                            CONSTANT_SPEED:
                                                                    begin
                                                                            if (acceleration && ~brake)
                                                                                    state           <=
ACCELERATING;
                                                                            else    if    (~acceleration    &&
brake )
                                                                                    state           <=
DECELERATING;
                                                                            object_y    <=    object_y    +
delta_y;
                                                                            dir_y <= dir_y + delta_y;
                                                                    end

                                                            CRASH:
                                                                    if (reset)
                                                                            state <= RESTART;

                                                            RESTART:
                                                                    begin
                                                                            level_y <= 600;
                                                                            object_y <= 0;
                                                                            delta_y <= 0;
                                                                            dir_y <= 0;
                                                                            count <= 0;
                                                                            time_counter <= 0;
                                                                            state                   <=
CONSTANT_SPEED;
                                                                    end
                                                            default:
                                                                    state <= CONSTANT_SPEED;
                                                    endcase
                        end//if
                end //else
        end // always 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,
           input marker, output reg [23:0] pixel);

  always @ * begin
    if ((hcount >= x && hcount < (x+WIDTH)) &&
                        (vcount >= y && vcount < (y+HEIGHT)))
                        begin
                                //Dash
                                if (~marker && vcount > 200 && vcount < 400)
                                        pixel = COLOR;
                                if(marker)
                                        pixel = COLOR;
                        end
                else
                        pixel = 0;
  end
endmodule


module dash  #(parameter COLOR = 24'hFF_FF_FF)
  (input [10:0] hcount,
   input [9:0] vcount,
    input [1:0] dir,
```

```
              input [10:0] x,
   output reg [23:0] pixel);

             always @(*) begin
                        if (dir==2'b01)
                                begin
                                        if ((((vcount > hcount - 205 )&&(vcount < hcount - 197))
            && (vcount > x && vcount < (x + 15)) && (x >= 200 && x <= 380)))
            pixel = COLOR;
          else pixel = 0;
      end
                        else
                                begin
                                        if ((((vcount > 750 - hcount)&&(vcount < 758 - hcount))
            && (vcount > x && vcount < (x + 15)) && (x >= 200 && x <= 380)))
            pixel = COLOR;
          else pixel = 0;
      end
   end
endmodule

module wheel_bar
  #(parameter COLOR = 24'hFF_FF_FF)  // default color: white
  (input [10:0] hcount,
   input [9:0] vcount,
             input [1:0] dir,
   output reg [23:0] pixel);

            reg signed [16:0] x, y, b;

  always @(*) begin
                        x <= 475;
                        y <= 520;

                        if (dir==2'b00) begin
                                if (vcount > (y-5) && vcount < (y+5) &&
                                         hcount > (x-70) && hcount < (x+75))
                                        pixel = COLOR;
                                else
                                        pixel = 0;
                        end else if (dir==2'b10) begin // right tilt
                                b <= y - x;
                                if ((vcount > hcount + b)&& (vcount < hcount + b + 13)
                                                && hcount > (x-52) && hcount < (x+47))
                                        pixel = COLOR;
                                else pixel = 0;
                        end else begin          // left tilt
                                b <= y + x;
                                if ((vcount > b - hcount) && (vcount < b - hcount + 13)
                                                && hcount > (x-46) && hcount < (x+53))
                                        pixel = COLOR;
                                else pixel = 0;
                        end
  end
endmodule

module circle
            #(parameter COLOR = 24'hFF_FF_FF)  // default color: white
  (input [10:0] hcount,
   input [9:0] vcount,
             input clock,
   output reg [23:0] pixel);

            reg [21:0] xs, ys;
            reg signed [16:0] x, y;
```

```
	always @(posedge clock) begin
		// WHEEL CIRCLE
		x <= 475;
		y <= 520;
		xs <= (hcount-x)*(hcount-x);
		ys <= (vcount-y)*(vcount-y);
		if ((xs+ys>4000) && (xs+ys<5500)) begin
			pixel = COLOR;
		end else
			pixel = 0;
  end
endmodule

module board
	#(parameter COLOR = 24'hFF_FF_FF)  // default color: white
  (input [10:0] hcount,
   input [9:0] vcount,
   output reg [23:0] pixel);

	always @(*) begin
		if (((((vcount>400)&&(vcount<406))||((vcount>79)&&(vcount<84)))&&((hcount>95&&hcount<855)))
			||(((vcount>79)&&(vcount<=401))&&(((hcount>=95&&hcount<=100))||(hcount>850&&hcount<855))))
begin
			pixel = COLOR;
		end else
			pixel = 0;
  end
endmodule

module road
	#(parameter ROAD_COLOR = 24'hFF_FF_FF, GRASS_COLOR = 24'h00_7F_00)  // default color: white
  (input [10:0] hcount,
   input [9:0] vcount,
	 input [1:0] dir,
	 input [1:0] crash,
	 input clock,
   output reg [23:0] pixel);

	 always @(posedge clock) begin
		// STRAIGHT
		if (dir==2'b00) begin
			if ((((vcount > 640 - hcount)&&(vcount < 645 - hcount))		// left line, slope: -1
				|| ((vcount > hcount - 310)&&(vcount < hcount - 305)))		// right line, slope: 1
				&& (vcount > 200 && vcount <= 400))
				pixel = ROAD_COLOR;
			// Grass
			else if ((crash!=2'b11)&&(((vcount < 645 - hcount)&&(hcount >100)&&(hcount<470))
				|| ((vcount < hcount - 305)&&(hcount<=850)&&(hcount>470)))
				&& (vcount > 200 && vcount <= 400))
				pixel = GRASS_COLOR;
			else pixel = 0;

		// RIGHT SWERVE
		end else if (dir==2'b01) begin
			if (((((vcount>>2) > 425 - hcount)&&((vcount>>2) < 430 - hcount))		// left line, slope: -4
				|| ((2*vcount > hcount - 50)&&(2*vcount < hcount - 45)))		// right line, slope: 1/2
				&& (vcount > 200 && vcount <= 400))
				pixel = ROAD_COLOR;
			// Grass
			else if ((crash!=2'b11)&&((((vcount>>2) < 430 - hcount)&&(hcount>100)&&(hcount<400))
				|| ((2*vcount < hcount - 45)&&(hcount<=850)&&(hcount>400)))
				&& (vcount > 200 && vcount <= 400))
				pixel = GRASS_COLOR;
			else pixel = 0;
```

```
                // LEFT SWERVE
                end else begin      // left swerve
                        if ((((2*vcount > 900 - hcount)&&(2*vcount < 905 - hcount))          // left line, slope: -1/2
                                || (((vcount>>2) + 45 > hcount - 500)&&((vcount>>2) + 45 < hcount - 495)))       //      right
line, slope: 4
                                && (vcount > 200 && vcount <= 400))
                                pixel = ROAD_COLOR;
                        // Grass
                        else if ((crash!=2'b11)&&(((2*vcount < 905 - hcount)&&(hcount>100)&&(hcount<540))
                                || (((vcount>>2) + 45 < hcount - 495)&&(hcount<=850)&&(hcount>=540)))
                                && (vcount > 200 && vcount <= 400))
                                pixel = GRASS_COLOR;
                        else pixel = 0;
                end
        end

endmodule

module horizon
        #(parameter COLOR = 24'hFF_FF_FF)  // default color: white
   (input [10:0] hcount,
    input [9:0] vcount,
    output reg [23:0] pixel);

        always @(*) begin
                if (((vcount>197&&vcount<201)&&((hcount>=100)&&(hcount<=850))))
                        pixel = COLOR;
                else
                        pixel = 0;
   end
endmodule

module speed_level
        #(parameter COLOR = 24'hFF_FF_FF)  // default color: white
   (input [10:0] hcount,
    input [9:0] vcount,
         input [9:0] level_y,
    output reg [23:0] pixel);

        always @(*) begin
                if ((vcount>level_y&&vcount<600)&&((hcount>230)&&(hcount<270)))
                        pixel = COLOR;
                else
                        pixel = 0;
   end
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 [10:0]            hcount;    // pixel number on current line
  reg [9:0] vcount;     // line number

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

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

  // sync and blanking
  wire      next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
```

```
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
//
// Bug due to memory management will be fixed. The bug happens because
// memory is called based on current hcount & vcount, which will actually
// shows up 2 cycle in the future. Not to mention that these incoming data
// are latched for 2 cycles before they are used. Also remember that the
// ntsc2zbt's addressing protocol has been fixed.

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

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


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

  input reset, clk;
  input [10:0] hcount;
  input [9:0]        vcount;
  output [17:0] vr_pixel; // Was 8 bits
  output [18:0] vram_addr;
  input [35:0]  vram_read_data;

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

  wire [18:0]        vram_addr = {vcount_f, hcount_f[9:1]}; //{1'b0, vcount_f, hcount_f[9:2]

  wire hc4 = hcount[0]; // was [1:0]
  reg [17:0]         vr_pixel; // Was 8 bits
```

```
   reg [35:0]              vr_data_latched;
   reg [35:0]              last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc4==1'b1) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==1'b0) ? vram_read_data : vr_data_latched;

   always @(*)                         // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4)
       1'b1: vr_pixel = last_vr_data[17:0];
       1'b0: vr_pixel = last_vr_data[35:18];
       //2'd1: vr_pixel = last_vr_data[7+16:0+16];
       //2'd0: vr_pixel = last_vr_data[7+24:0+24];
     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
//
///////////////////////////////////////////////////////////////////////////////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
```

```
// <fpga_clock> should probably be held inreset until <locked> is high.
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

	assign ref_clk = ref_clock;

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

   DCM int_dcm (.CLKFB(fpga_clock),
		.CLKIN(ref_clk),
		.RST(dcm_reset),
		.CLK0(fpga_clk),
		.LOCKED(lock1));
   // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of int_dcm is 0

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

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

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

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


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

```
   assign locked = lock1 && lock2;

endmodule
```