SkiFree

Figure 1: Screenshot of Windows SkiFree
(Retrieved from http://ski.ihoc.net/)

Eric Grebing and Erik Stafl

TA: Cassie Huang
6.111 – Introductory Digital Systems Laboratory
December 12, 2006

Abstract
The following implementation of SkiFree integrates a visual controller with a rewritten version of the classic Windows PC game. Two hardware components are used in the system – a camera and the 6.111 labkit. Over 20 module blocks provide functionality, but the project can be divided into two main parts – the controller and the game itself. The controller consists of a small wooden block covered with a piece of pink construction paper. The user turns the block in his hand to control the motion of the skier. A camera tracks the position and angle of the block and the image is processed using color recognition and center of mass calculation to show changes in orientation. The data from this controller is then fed into the game and changes the orientation skier sprite on-screen. This version of SkiFree has the same general gameplay of the original Windows application, including slalom mode.
# Table of Contents

Abstract........................................................................................................................................... i

Table of Contents.......................................................................................................................... ii
  Table of Figures.......................................................................................................................... iii
  Table of Tables........................................................................................................................... iii

Project Overview............................................................................................................................ 1

Implementation and Modular Description.......................................................................................... 2
  Displaying Graphics on Screen........................................................................................................ 2
  Timer Module................................................................................................................................ 4
  Skier Video Module....................................................................................................................... 5
  Distance Display Module............................................................................................................. 5
  Landscape Video Module.............................................................................................................. 6
  Collision Detection Module......................................................................................................... 8
  Timer Toggle Module................................................................................................................... 8
  Game Video Output Module......................................................................................................... 8
  Slalom Gate Detect Module......................................................................................................... 8
  Time Display Module.................................................................................................................. 9
  Detecting the Orientation of the Controller................................................................................... 10
  NTSC to ZBT Module.................................................................................................................. 12
  Display Detect Module............................................................................................................... 12
  Center of Mass Module............................................................................................................... 13
  Angle Detect Module.................................................................................................................. 13
  Final Compositor Module............................................................................................................ 13

Testing and Debugging....................................................................................................................... 14

Conclusion....................................................................................................................................... 15

Acknowledgements.......................................................................................................................... 16

Appendix A: Sprite Converter Matlab Script..................................................................................... 17

Appendix B: Project Verilog Source Code........................................................................................... 18
Table of Figures

Figure 1: Screenshot of Windows SkiFree.................................................................1
Figure 2: A Screenshot from our Implementation of SkiFree.................................. 1
Figure 3: High-level Block Diagram..................................................................... 2
Figure 4: Detailed Block Diagram for the Game Module.................................... 4

Figure 5: YCbCr Example.................................................................................... 10

Figure 6: The UV Color Space........................................................................... 11
Figure 7: Block Diagram for Video Modules....................................................... 12

Table of Tables

Table 1: Landscape_object Index Values............................................................. 7
Table 2: Digit_to_display Index Values................................................................. 10
Project Overview

The SkiFree game involves a skier in downhill motion, trying to avoid obstacles in the landscape while navigating a slalom course. In order to simulate this downhill motion, the skier image remains relatively static near the center of the screen, while the objects in the landscape move around him. The individual sprites move from the bottom to the top of the screen and then regenerate at the bottom to make a continuous map. The images for our implementation of the game are taken from actual screenshots from the original SkiFree game for Windows, and the video output of the playing area is a standard 1024x768 XVGA signal.

As opposed to using a mouse and keyboard to control the skier as is common in Windows SkiFree, our implementation of the game uses a camera to detect the angle of a colored block held by the player. If a player wants to go straight downhill, then he or she simply needs to hold the block in a vertical position. Likewise, if the player would like to move either left or right, the block can be tilted in the desired direction to make the skier follow that path. Our implementation provides a calibration mode for the player to see the image analysis that the detection logic is performing on the camera input to determine the angle of the block. If switch1 is on, then the player will simply see this full screen calibration view. However, to allow gameplay while enabling the player to make sure that the block is within view of the camera, a small overlay image of this calibration view is provided in the upper right corner of the screen. This mode is enabled with switch0.

As in the original game, if the skier hits an obstacle, such as one of the many trees or rocks that are scattered about the slope, then he will crash and remain stuck for about one second. There is a timer and distance meter on the screen to indicate how long it has taken the player to ski a certain distance. The slalom gates on the course also change color to indicate whether the skier has passed on the correct side or not, and if the skier has not, a five second penalty is added to the skier’s time. Figure 2 shows an image of the screen with important features highlighted.

![Figure 2: A Screenshot from our Implementation of SkiFree](image-url)
Implementation and Modular Description

The general structure of our implementation of SkiFree can be broken down into two major modules: the game module, and the video detection module. The game module, written by Eric Grebing, handles the actual gameplay and drawing of the game graphics to the screen. The video detection module, written by Erik Stafl, takes a video signal from a camera and performs an analysis to determine the angular orientation of the colored block. The only communication between these two modules is the simple 3-bit `skier_orientation` signal, which dictates the angle that the skier should be facing, and the 24-bit `game_pixel` signal, which is simply the RGB pixel data from the game.

Since the interface between these two major modules is so simple, each was developed independently until they both performed satisfactorily. They were then integrated onto a single 6.111 FPGA Labkit.

The high-level block diagram of our implementation is shown in Figure 3. The modules located within the gray box are those that are part of the game major module, while the rest is part of the video detection and handling module.

![Figure 3: High-level Block Diagram](image-url)
Displaying Graphics on Screen
(By Eric Grebing)

All of the graphics used in this version of SkiFree are simple two-dimensional sprites. Each image used for the sprites was obtained using screen shots from a 24-bit color Windows XP version of the original SkiFree for Windows posted at http://ski.ihoc.net/. Each sprite image was isolated from various screenshots using Microsoft Paint and then was saved as a 24-bit bitmap image. ROMs were created in order to load image files into the FPGA. Each ROM holds a .coe file which contains a 24-bit hexadecimal color value for each pixel in the image. This .coe file was obtained with a Matlab script that took a bitmap image as input and output comma-separated color values. Over 20 ROMs were used for the SkiFree graphics. Many of the ROMs are instantiated multiple times throughout the game, especially images that make up the skiing landscape.

The skier_video, landscape_video, time_display, and distance_display modules provide the pixels to the game_video_output module for display on the screen. Each module uses a similar strategy to determine the value of each pixel in the 1024x768 XVGA display. At the beginning of each of these graphics modules, the roms containing each sprite image are instantiated. Each instantiation of a rom is assigned a unique register to hold the pixel’s address, in which the size of the register corresponds to the number of pixels in the image. Each instantiation of a sprite is also assigned a unique 24-bit data wire to transmit the RGB value for the current pixel addressed. Each sprite is also assigned parameter values that correspond to the image’s height and width.

In order to display an image on the monitor, the XVGA system refreshes all pixels continuously, refreshing the entire screen at a rate of 60 Hz. Two variables, \textit{hcount} and \textit{vcount}, track the horizontal and vertical coordinates of the current pixel being refreshed. These \textit{hcount} and \textit{vcount} values are inputs to each of the graphics modules and are used as comparison values to the locations of the sprites.

To determine which values to send for each display pixel, a four stage pipeline handles a series of calculations to determine if the module should output a pixel for the given refresh locations of \textit{hcount} and \textit{vcount}, calculate the address for the correct rom if a pixel should be displayed by the module, and then pass the RGB color data to the game_video_output module so that the image can be seen on the screen.

In the first stage of the pipeline, the coordinate values as well as the height and width of each sprite are compared to the current \textit{hcount} and \textit{vcount} on the screen. If the current \textit{hcount} and \textit{vcount} fall within the values occupied by a particular sprite, the rom corresponding to that sprite is identified and passed by means of a register value to the next stage of the pipeline.

Stages two and three deal with determining the correct address for an image ROM that matches the pixel value that should be displayed on screen. Stage two performs a multiplication step that determines the row of pixels wanted in the ROM of interest. Since multiplication is a very computationally intense operation within the FPGA and causes a significant delay, this operation is isolated in its own pipeline stage. The third stage of the pipeline uses the result of the multiplication and adds it to the difference in \textit{hcount} from the current sprite’s x coordinate. This fully determines the correct address for the current pixel.

Finally, the last pipeline stage assigns the module’s output pixel to be the 24-bit RGB value stored at the address in the ROM of interest. Additionally, if the module outputs a pixel value for the current \textit{hcount} and \textit{vcount}, the most significant bit is set to one to indicate that the
pixel is active. Game_video_output handles inputs from the four pixel-producing modules to combine the video output for the SkiFree game. Please see Figure 4 for a detailed block diagram.

**Figure 4: Detailed Block Diagram for the Game Module**

**Timer Module**

The timer module splits the signal \(clk\) from the global 65 Mhz clock into signals of lesser frequency by using a series of counting registers. The module outputs a 100 Hz, 10 Hz, 1 Hz, 1/10 Hz, and 1/60 Hz signals, named \(hundred\_hz\_signal\), \(ten\_hz\_signal\), \(one\_hz\_signal\), \(tenth\_hz\_signal\), and \(sixtieth\_hz\_signal\) respectively. When the \(timer\_run\) signal, an input from the boundary_detect module, is high signifying that the timer should be running, \(hundred\_hz\_signal\) goes high once every 650,000 clock cycles. This is achieved by incrementing \(hundred\_hz\_counter\), a 20-bit register. \(Hundred\_hz\_counter\) resets to zero whenever it holds a value of 650,000. The next three output signals, \(ten\_hz\_signal\), \(one\_hz\_signal\), and \(tenth\_hz\_signal\), have associated 4-bit counters that increment according to the \(hundred\_hz\_signal\). For example, the 4-bit register \(ten\_hz\_counter\) increments each time that \(hundred\_hz\_signal\) is asserted. After the \(ten\_hz\_counter\) reaches ten, \(ten\_hz\_signal\) is asserted since this corresponds to 6,500,000 clock cycles of the 65 Mhz clock. Likewise, \(one\_hz\_counter\), \(tenth\_hz\_counter\), and \(sixtieth\_hz\_counter\) increment when the \(ten\_hz\_signal\) and \(one\_hz\_signal\) are asserted respectively. In the same manner, \(sixtieth\_hz\_signal\) is asserted whenever the 3-bit \(sixtieth\_hz\_counter\) reaches a value of six. The output signals from the timer module are passed to the time_display module to drive the changes in the digit roms.
corresponding to time. When the system reset signal is asserted, all registers and signals in the timer module are set to zero and the change affects the text output of the time display.

This module also assigns a five second penalty each time a slalom gate is missed. Thus, the missed_left_gate and missed_right_gate signals from the slalom_gate_detection module are used as inputs to the timer module. If either of the missed gate signals are high, the one_hz_signal is asserted five times to add five seconds to the skier’s time. In addition to asserting one_hz_signal five times, the internal variable tenth_hz_counter is also incremented by five. This causes the timer display to show a correct increase in the skier’s downhill time.

Skier Video Module

The skier_video module is responsible for taking in the skier’s orientation from the controller and displaying the correct sprite on the screen. Skier_video takes inputs from the 65 Mhz clock, the XVGA hcount and vcount, the signal crash_mode (signifying that the skier has crashed into an object), and a 3-bit value skier_orientation from the controller. The module then outputs a 25-bit value skier_pixel, which is sent to the game_video_output module, and the signals skier_x and skier_y, which correspond to the skier’s x and y coordinates on the screen.

Before a pixel value can be produced, the skier_video module must instantiate each sprite rom corresponding to individual skier images. This module uses five different sprites of the skier – headon (skier_headon_rom), turned thirty degrees right (skier_pos_30_rom), turned sixty degrees right (skier_pos_60_rom), turned ninety degrees right (skier_pos_90_rom), and the skier crashed (skier_crashed_rom). The roms holding the images corresponding to the turned skier’s are instantiated twice and addressed in reverse to display the skier turned to the left at the three different angles.

Two 10-bit registers, skier_x and skier_y, hold the x and y pixel coordinates of the skier’s position on the screen. In normal non-jumping mode, these positions are fixed at their initial values of 400 on the horizontal and 250 on the vertical. The skier’s movement is therefore governed by the movement of the landscape around him.

At every positive clock edge, the value of skier_pixel is updated to correspond to the current RGB value of the pixel. First, the system checks to see if the signal crash_mode is high, indicating that the skier has crashed into a landscape object. (This signal is sent from the collision_detection_module). If the system is in crash mode, the skier_crashed_rom is addressed and the appropriate sprite depicting a crashed skier is displayed. If the system is not in crash mode, an appropriate rom is selected based on the current value of skier_orientation. Skier_orientation is a 3-bit register, sent from the controller, whose value corresponds to the direction the skier is currently facing. Table 1 explains the different states of the FSM for the skier’s orientation. Based on the value of skier_orientation, the case statement outputs the corresponding sprite onto the screen. Whenever the vcount and hcount match the placement of the skier on screen, skier_pixel[24] takes a value of one to signify it is active. Skier_pixel[23:0] contains the 24-bit RGB value for the current pixel. The entire 25-bit value is output to the game_video_output module for displaying on the screen and the collision_detection_module to compare to the value of the pixel of the landscape objects to detect crashes.

Distance Display Module

The distance_display module outputs the current value of the skier’s distance remaining on the course. This module takes in data from the distance_tracker module to control the four digits of the distance remaining. The four signals, distance_thousands, distance_hundreds,
distance_tens, and distance_ones, correspond to the four digits in the distance output. Each digit displayed references a different instantiation of digits_rom. Digits_rom contains a monochrome bitmap that has images of the digits zero through nine arranged vertically. The rom is then addressed to the proper digit based on the value needed. The process of determining the output pixel is passed through a four-stage pipeline, very similar to other modules that produce an output pixel. The same technique is also used to display the digits for the time value in the time_display module.

Landscape Video Module

The landscape_video module generates all of the pixels corresponding to objects in the ski hill’s landscape. Features such as the pine tree, bare tree, rocks, tree stumps, slalom gates, and ramps are all generated in landscape_video. This module also controls the movements of each of the landscape sprites, to simulate the skier’s movement down the hill. Landscape_video outputs the 25-bit pixel signals landscape_pixel, slalom_left_pixel, slalom_right_pixel, and ramp_pixel. Each of these outputs is passed to the game_video_output module so that their RGB values can be displayed on the screen. The module also outputs left_slalom_gate_x, left_slalom_gate_y, right_slalom_gate_x, and right_slalom_gate_y, all of which correspond to the current x and y coordinates of the location of the slalom gates. These signals are passed to the slalom_detect module to determine whether or not the skier correctly navigates the slalom gates.

The video pixels are determined using the four stage pipeline system used throughout the system to output sprites. Seven different roms, big_tree_rom, bare_tree_rom, rock_rom, tree_stump_rom, ramp_rom, left_gate_rom, and right_gate_rom are used to store the RGB values for each landscape object. All of the landscape roms except for the slalom gate roms are instantiated multiple times to completely populated the screen with obstacles.

Once the roms are instantiated, the current hcount and vcount values are compared to each object’s location and area to determine whether or not the sprite should be displayed. In the first stage of the pipeline an object should be displayed, the 5-bit register landscape_object takes the value of the index of that object. Table 1 shows the index value for each landscape object. If no object should be displayed at the current pixel location, landscape_object takes the value zero.

<table>
<thead>
<tr>
<th>Landscape_object Index</th>
<th>Object Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>big_tree_one</td>
</tr>
<tr>
<td>2</td>
<td>bare_tree_one</td>
</tr>
<tr>
<td>3</td>
<td>tree_stump_one</td>
</tr>
<tr>
<td>4</td>
<td>big_tree_two</td>
</tr>
<tr>
<td>5</td>
<td>bare_tree_two</td>
</tr>
<tr>
<td>6</td>
<td>tree_stump_two</td>
</tr>
<tr>
<td>7</td>
<td>big_tree_three</td>
</tr>
<tr>
<td>8</td>
<td>bare_tree_three</td>
</tr>
<tr>
<td>Landscape_object Index</td>
<td>Object Name</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>9</td>
<td>tree_stump_three</td>
</tr>
<tr>
<td>10</td>
<td>rock_one</td>
</tr>
<tr>
<td>11</td>
<td>rock_two</td>
</tr>
<tr>
<td>12</td>
<td>ramp_one</td>
</tr>
<tr>
<td>13</td>
<td>ramp_two</td>
</tr>
<tr>
<td>14</td>
<td>left_slalom_gate</td>
</tr>
<tr>
<td>15</td>
<td>left_slalom_gate (frowning sign)</td>
</tr>
<tr>
<td>16</td>
<td>left_slalom_gate (smiling sign)</td>
</tr>
<tr>
<td>17</td>
<td>right_slalom_gate</td>
</tr>
<tr>
<td>18</td>
<td>right_slalom_gate (frowning sign)</td>
</tr>
<tr>
<td>19</td>
<td>right_slalom_gate (smiling sign)</td>
</tr>
</tbody>
</table>

**Table 1:** Landscape_object Index Values

The values of landscape_object are then passed to the second stage of the pipeline, a case statement that determines the multiplication stage of the object’s pixel memory address. This stage of the pipeline is necessarily isolated due to the fact that multiplication causes a substantial delay on the FPGA. From there, the multiply_stage value is passed to another case statement chosen by the one clock cycle delay of landscape_object. In this third stage of the pipeline, the value of multiply_stage is added to the difference of the object’s location and hcount to determine the address of the object.

The fourth stage of the pipeline takes the value of landscape_object delayed by two clock cycles and outputs the corresponding RGB value from the rom over the object’s data wire. The most significant bit of the output is set to one for the type of pixel to which the object corresponds. The 25-bit value of the active object is then passed to the game_video_output module to be displayed on the screen.

The landscape_video module also controls the motion of all of the landscape sprites. Each moves according to the skier’s downhill velocity and the angle of the skier. Simple vector addition is used to determine the vertical component, skier_vel, and the horizontal component, horiz_change. When the skier is not in crash mode, the sprites move from bottom to top according to these values. When the skier is in crash mode, determined by the signal crash_mode from the collision_detection module, all motion of the sprites stops. When crash mode ends, the landscape shifts to the right and the skier has a free path to continue down the hill.

In order to allow for the tracking of each object’s location, each landscape sprite is assigned two 10-bit registers – one for the x coordinate of it’s current pixel location and one for the y coordinate. Each object coordinate register is set with an initial value to spread out the landscape images on the map. Whenever the skier is moving, these values change according to skier_vel and horiz_change, as the sprites approach the top of the screen. When an object’s y coordinate value is less than five, the y register is reset to a specific value between 768 and 1023 (these values will not immediately be displayed on screen but can be held by a 10-bit register).
The fact that each of the objects resets to a different value allows for the ‘randomization’ of the map each time the sprites reappear at the bottom of the screen. Within this motion simulation, the distance traveled by the skier is also generated. The distance_counter register counts each vsync signal and decrements the distance_to_go signal by one at every other instance of a change in vsync. Distance_to_go is initially set to 2000 and the course ends when this value reaches zero. Distance_to_go is passed to the distance_tracker module that outputs signals to control the distance display.

Collision Detection Module

The collision_detection module detects crashes between the skier and objects in the landscape. In order to accomplish this detection, the module takes as input bit 24 of both landscape_pixel and skier_pixel and assigns them the names landscape_pixel_active and skier_pixel_active. The module also takes in the one bit value detect_crashes from landscape_video. When this value is one, collision_detection works to determine crashes, but otherwise will not. This insures that the skier will not become stuck in an infinite loop in crash_mode.

If detect_crashes, skier_pixel_active and landscape_pixel_active each has a value of one on a clock cycle, the system will enter crash mode and the register crash_mode takes on a value of one. The 27-bit crash_counter register holds the skier in crash_mode for 50,000,000 clock cycles, or a little less than one minute in real time. Once this time has passed, the module returns to a crash_mode value of zero and the skier will continue down the hill again.

Timer Toggle Module

The timer_toggle module is responsible for activating and deactiving the function of the timer running. The module takes in the one bit value course_started from landscape_video, which signifies whether or not the skier has started the course. The module also takes in the 11-bit register distance_to_go from landscape_video, that determines how much further the skier must go before reaching the end of the course.

When the game first loads, the timer_run signal will go to one, indicating that the timer and time_display should be outputting values. Once the skier’s value of distance_to_go reaches zero, timer_run is deasserted and the skier’s time for the course stays constant on the screen. If a user reset occurs, timer_run will go to one again and the timer will start.

Game Video Output Module

The game_video_output module takes in 25-bit pixel values from each of the sprite-producing modules to combine all pixels for the video output. The hsync, vsync, and blank signals are all delayed by six clock cycles to compensate for the pipeline delays of each of the sprite-producing modules. A hierarchical cascade of if statements govern the value of the game’s pixel output. At each clock cycle, the module checks the bit 24 value of time_display_pixel, ramp_pixel, distance_display_pixel, slalom_right_pixel, slalom_left_pixel, landscape_pixel, and skier_pixel in order. At the first instance of one of these pixels being active, the 24-bit RGB value in bits 23:0 of the register are transferred to the 24-bit register game_pixel_out. Game_pixel_out is then split into 8-bit values for red, green, and blue color in the xvga module.
**Slalom Gate Detect Module**

Slalom_gate_detect compares the values of skier_x and skier_y with the x and y coordinate values of the slalom gates represented by left_slalom_gate_x, left_slalom_gate_y, right_slalom_gate_x, and right_slalom_gate_y. Based on a system of checking at each positive edge of the vsync signal, the module determines whether gates have been encountered and missed on the path. The output values left_gate_encountered, right_gate_encountered, missed_left_gate, and missed_right_gate send signals to other modules to indicate how the gate should appear on screen and whether or not to assign a time penalty. If a gate is missed, the landscape_video module makes it appear as a red frowning sign. If the gate is navigated correctly, landscape_video replaces the slalom gate with a green smiling sign.

Once the skier passes the threshold of a slalom gate (when the y coordinate of the slalom gate plus the gate’s height is less than the y coordinate of the skier, skier_y), the appropriate gate_encountered signal is asserted for either the left or right slalom gate. At this same point, the module checks the skier_x coordinate to see if it is either to the left or right of the corresponding slalom gate. If the skier is not on the appropriate side, the gate is ‘missed’ and either missed_left_gate or missed_right_gate is assigned a value of one. Whenever a slalom gate reaches the top of the screen and disappears, the values of its gate_encountered and missed_gate signals are reset to zero for when the gate reemerges at the bottom of the screen.

**Time Display Module**

Time_display takes in the divider pulses from the timer – hundred_hz_signal, ten_hz_signal, one_hz_signal, tenth_hz_signal, and sixtieth_hz_signal and uses the signals to control the output display of the timer. The module’s final output is the 25-bit time_display_pixel to the game_video_output module. The x and y coordinates of each digit are fixed to specific values in the right corner of the screen. The five digits, those representing 1/100 second, 1/10 second, one second, ten seconds, and one minute, each have two 10-bit registers that refer to and hold the values of these coordinates.

In order to determine the value for each digit at each clock cycle, five internal registers – hundredth_digit_counter, tenth_digit_counter, onesec_digit_counter, tensec_digit_counter, and onemin_digit_counter – increment according to the various frequency signals from the timer module. The values of these counters help to determine the address of the digits_rom, which in turn outputs a particular number.

Time_display uses a four-stage pipeline similar to that in landscape_video to identify if a pixel is in the range of the current hcount and vcount, address the correct rom, and output the RGB values for the pixel associated with the address. The first stage checks the coordinates of the time digit positions against the values of hcount and vcount. If one of the digits is in range, the appropriate index of the variable digit_to_display described in Table 2 is used. If one of the digits is in range, the second stage of the pipeline performs one of the multiplication steps in addressing the rom. Depending on which digit is being displayed, the appropriate counter is multiplied by the parameter DIGIT_HEIGHT. The output value of multiply_stage is then passed to the next stage of the pipeline, where the final address is determined. After the address is known, the final stage of the pipeline outputs the data from the appropriate address of the appropriate rom. If none of the time_display pixels are in range for the current hcount and vcount, bit 24 of time_display_pixel is set to zero to signify inactivity.
Detecting the Orientation of the Controller
(By Erik Stafl)

The process of determining the orientation of the controller block is comprised of two steps. First, the logic must be able to filter out pixels that do not correspond to the pink controller block by using some sort of color detection. Then, once the desired pixels have been selected, multiple center of mass calculations must be made to determine where the block is located, and the angle at which the block is being held.

In our implementation, NTSC video data coming in from the camera is decoded into 30-bit per pixel YcrCb data. This encoding corresponds to 10 bits for each of the vectors in the YUV color space: Luminance, Chroma-Red and Chroma-Blue. The 6 highest order bits are taken from each of these signals so that the encoding can be compressed to 18 bits per pixel. This is necessary because the ZBT RAM that houses the main frame buffer only stores 36 pixels per address, and it must alternate read and write cycles. If pixels are stored with only 18 bits per pixels, this means that two pixels can be stored per address, and then read and write cycles can alternate while resulting in the appearance of a smooth, continuous frame buffer, without a significant sacrifice in quality.

When the video data is being read out of the ZBT frame buffer by the display_detect module, a comparison is run to determine whether or not the current pixel’s Cr value exceeds a certain threshold. If it does, then the pixel is marked as needing to be factored into this frame’s center of mass calculation, by asserting the selected signal.

The reasoning behind using the Cr value for color detection instead of luminance or some calculation in RGB space is because the Cr value is much more resilient to lighting fluctuations and shading (Please see Figure 5). YCrCb, which is simply a digital encoding representing the YUV color space, represents color as a vector with one

<table>
<thead>
<tr>
<th>Digit_to_display Index</th>
<th>Corresponding Digit/Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time_text_display</td>
</tr>
<tr>
<td>2</td>
<td>Hundredth of a second digit</td>
</tr>
<tr>
<td>3</td>
<td>Tenth of a second digit</td>
</tr>
<tr>
<td>4</td>
<td>One second digit</td>
</tr>
<tr>
<td>5</td>
<td>Ten second digit</td>
</tr>
<tr>
<td>6</td>
<td>One minute digit</td>
</tr>
</tbody>
</table>

Table 2: Digit_to_display Index Values

**Figure 5:** YCbCr Example
component relating to intensity and two components relating to its chrominance value. Specifically, the Cr value maps to V and the Cb value maps to U. With this encoding, a solid red object, for example, would generate pixels with similar chrominance values, even if the lighting on it were non-uniform. This quality is highly desirable when tracking the colored block for this controller, because inevitably the player will be using the block in somewhat uneven lighting conditions and will tilt the angle of the block as her or she plays.

A bright pink color was chosen for the block because its chrominance value is quite far outside that of normal colors, such as skin tones or the colors of most clothing. As shown in Figure 6, the bright pink color used in this project has an approximate (U,V) value of (0.05,0.4). Since a V or Cr value of 0.4 is far out of the range of normal, the results were quite good from using this color detection scheme.

Once the appropriate colors were selected, the second step is to use this added data to perform the necessary calculations to determine the position and orientation of the block. Calculating the position of the block is accomplished with a simple two-axis center of mass calculation to determine the average x and y values for the pixels that have been identified as belonging to the controller block. Each frame, every pixel with an active selected signal has its hcount and vcount signals added to two large registers, and a counter which keeps track of how many pixels have been selected is incremented. At the beginning of the next frame, the x and y totals are sent to two 34 stage pipelined dividers to be divided by the number of selected pixels for that frame. After the dividers complete their calculations, the result is stored for an entire frame and drawn on the screen as an overlay to show where the center of mass for the selected pixels is.

Calculating one center of mass is helpful, but does not give enough data to determine the orientation of the block. In order to do this, four other center of mass locations are calculated very frame. These correspond to all the pixels above, below, left and right of the overall center of mass. So in addition to having data describing the position of the block, these other calculations give the logic the center for the top half of the block, left half, right half, and bottom half. With this, simple comparisons of the x and y values of these center of masses can determine the orientation of the block. These few simple comparisons produce the 3-bit skier_orientation signal that needs to be sent to the game logic.

The last major component relating to video in our implementation is the final compositor, which allows for a small overlay of the video feed in the upper right hand corner of the screen. This is accomplished by writing to a 21600 address by 24 bit depth Block RAM with every 4th pixel that is read from the ZBT memory. The location of the output of the large camera image and the small overlay image are different, so reading and writing never have to take place at the same time.

Please refer to the Figure 7 for a block diagram describing these components. More specific implementation details are provided below in the descriptions of the most significant modules.
The ntsc_to_zbt module prepares the write enable, address and data signals to write to the ZBT RAM. This module features a few small FIFO buffers made out of registers that help synchronize the data across the different clock speeds of the video camera and the XVGA clock. Every other 65 MHz clock cycle, two pixel values are outputted on the ntsc_data[35:0] signal with their address being stored as a combination of the current x and y coordinate in the ntsc_addr[18:0] signal. Finally, the ntsc_we signal is true when this data is ready to be written, otherwise it is low to allow the data to be read from the ZBT by the display_detect module.

Display Detect Module

The display_detect module is crucial to the entire process of the video analysis, because it reads the video data from the ZBT every other clock cycle, performs the color detection of the Cr value of the pixel, and calls the center of mass and angle detect modules. This module takes in values such as hcount, vcount, and threshold to use and pass on to the center of mass module. It reads the pixel data every other clock cycle from the vram_data[35:0] signal. Since the ZBT has a two clock cycle latency, the display detect module has a two stage pipeline.

To turn on the pixel selection by color, switch5 must be enabled. Then, if pixels are selected, the display detect module will output their color as black, so it is easy for the player to see what is being identified as the block. The threshold value can then be controlled by using the up and down pushbuttons and watching the hexadecimal value on the alphanumeric display increment or decrement. Furthermore, the main center of mass is identified with a horizontal and vertical pink line intersecting at that point, while the other four center of masses that are received...
from the angle detect module are shown are 4 pixel wide squares on the screen.

**Center of Mass Module**

The center of mass module contains the registers for counting how many pixels have met the color detection criteria per frame, and passes this value on as the divisor to the divider cores that are instantiated in this module. It also contains the x and y registers that store the total of the x and y coordinates of the selected pixels. So for example, if only 2 pixels were selected in one frame, and they were at the locations (0,0), (100,200), then the total of the x register would be 100, and that of the y register would be 200. When these are divided by the total pixelcount of 2, it returns the center of mass of all the selected pixels on the screen, which in this case is (50,100).

This module simply outputs the x and y coordinates of the center of mass of the block in the signals `haverage[10:0]` and `vaverage[9:0]`. Since the divider has a 34 stage pipeline, the values are not available for the first 34 clock cycles of each frame, but this delay does not affect the overall system performance or look.

**Angle Detect Module**

The angle detect module is where the final computation is performed to determine the value of the `skier_orientation` (referred to internally as `angle_signal`) signal, which is the entire purpose of the whole video detector block. This module takes in the `haverage[10:0]` and `vaverage[9:0]` signals from the center of mass module as well as the `selected` signal from the display detect module to perform its own center of mass calculations, but for the four “halves” of the screen. This results in the center of mass signals `leftave_x[10:0]`, `leftave_y[9:0]`, `rightave_x[10:0]`, `rightave_y[9:0]`, `topave_x[10:0]`, `topave_y[9:0]`, `bottomave_x[10:0]` and `bottomave_y[9:0]` that describe the center of masses of the four halves.

This module then calculates the difference between the x and y coordinates of these points and uses these values to determine the value of `angle_signal` at the beginning of each frame. There is also a register that stores the old value of the signal to make sure that any change in `angle_signal` is constant across two frames, because the control should not exhibit any noise.

**Final Compositor Module**

The final compositor module takes in the two video signals from the game module (`game_pixel[23:0]`) and video module (`video_pixel[23:0]`), and provides a mux for them to be switchable. In addition, the module instantiates a 21600x24bit BRAM to store small frames for the overlay image in the upper left corner of the screen if enabled. It is also important to mention that the final compositor module outputs a signal that is correctly lined up and pipelined to finish the total of a 6 stage pipeline that is in both the game and video modules.

If switch0 is on, then the overlay will appear, but if switch1 is on, a full screen view of the camera video will appear.
Testing and Debugging

A project with the magnitude of our SkiFree implementation required the testing and debugging of each module along the design path. In addition, the integration of the controller with the game required many debugging steps.  

For the debugging of the game's appearance, compiling, building, and viewing the final output served as the best indicator of problems with the design. Building test benches to look at the pixel values of sprites is not nearly as useful as looking at the output of the sprites themselves. Whenever a visual part of the game was imperfect, a thorough test was run by just playing the game, trying to invoke every possible scenario. When the system failed, a hypothesis about the cause for the failure was made and the code was changed accordingly. This proved to be a useful method in testing the functionality of the game's video output.

Some signals, especially those associated with the timer and distance displays, required more careful debugging. To see how signals were being passed between modules, the signals in question were assigned to logic analyzer outputs. By viewing the timing of waveforms on the analyzer, we could develop a visual representation of when problems in the passing of signals would occur. Often errors would occur simply because a wire was not declared for a signal connecting two different blocks in the labkit file.

The biggest testing and debugging challenge occurred at the integration stage. When the controller and game logic were combined, we found that timing constraints were not being met and many flaws became apparent when the system tried to build on the FPGA. Design and synthesis reports served as important tools that allowed us to pinpoint exactly which data paths were causing the timing constraints to fail. In the end, after many builds with failed timing contraints, we found that pipelining was needed for many of the calculations in the video display.

Although pipelining helped, other courses of action needed to be taken in getting the project to work correctly. We read through the relevant Xilinx documentation and found that we needed to constrain the area occupied by our logic on the FPGA. After constraining both area and time considerations, our project worked correctly according to the programmed logic.
Conclusion

Throughout this project, it became increasingly apparent that good planning is the key to any successful engineering design. Planning out a detailed block diagram before doing much coding proved to be immensely helpful. This allowed us to visualize all of the pieces of the project, making the actual implementation much easier. Planning an aggressive timeline, with a goal of finishing ten days ahead of schedule was also very important. Even with sticking to the goals for the most part, our design required a decent amount of last-minute debugging.

We also learned to never underestimate the complexity of the logic we were designing and the number of subtleties involved when programming onto the labkit. Testing before compiling became an important skill as our project took over 30 minutes to synthesize, compile, and program on the labkit.

There are a few features that we wish we had the time to implement such as reliable jumping over ramps, sound effects, and other video features to test our digital design skills. However, the timeframe for the class made it difficult to extend our game far beyond essential functionality.

In closing, the amount of work that this project required made it very satisfying to see it successfully come to fruition. We now feel empowered as digital designers and are ready to meet new challenges in our development as electrical engineers.
Acknowledgements

We would first of all like to thank the 6.111 teaching staff this term for their support and aid through the arduous process of the final project. As we progressed through the class, we found that experience is one of the keys to finding flaws in a design, so having the combined experience at our disposal was quite helpful.

In creating the project, some files were borrowed and modified from code available on the 6.111 course website. Zbt_6.111.v, debounce.v, and the labkit.v file from Lab 5 were for all intents and purposes kept intact for use with our game. Also the ntsc_to_zbt.v file was looked at, but completely rewritten for our purposes.

Finally, we would like to thank Chris Pirih for creating the original SkiFree game and being the inspiration for our project. We also were able to obtain the bitmaps for our sprites from screenshots of a current version of the game.
Appendix A: Sprite Converter Matlab Script

function BMPtoCOE(image_name)
% Converts a 24-bit color bitmap image to a Xilinx .COE file
% Was written so students could use a FPGA to display images on a VGA monitor

% read bmp data in and display it to the screen
imdata = imread(image_name);
image(imdata);
% colormap(immap);

imdata_red = transpose(double(imdata(:,:,1)));
imdata_green = transpose(double(imdata(:,:,2)));
imdata_blue = transpose(double(imdata(:,:,3)));

numpixels = numel(imdata_red);
imdata_red = dec2bin(imdata_red);
imdata_green = dec2bin(imdata_green);
imdata_blue = dec2bin(imdata_blue);

% create .COE file
COE_file = image_name;
COE_file(end-2:end) = 'coe';
fid = fopen(COE_file, 'w');

% write header information
fprintf(fid, ';******************************************************************
');printf(fid, ';****                 BMP file in .COE Format                 *****
');printf(fid, ';******************************************************************
');printf(fid, '; This .COE file specifies initialization values for a
');printf(fid, '\n');printf(fid, 'memory_initialization_radix=16;
');printf(fid, 'memory_initialization_vector=
');

% concatenate the values of RGB into one value
newimdata = [imdata_red imdata_green imdata_blue];

% change binary values to decimal so they can be written as hex values
newimdata = bin2dec(newimdata);

% write image data to file
for j = 1:(numpixels-1)
    fprintf(fid, '\n', dec2hex(newimdata(j)));
end
% last data value supposed to have a semicolon instead of a comma
fprintf(fid, '\n', dec2hex(newimdata(numpixels)));
% clean shutdown
fclose(fid)
Appendix B: Project Verilog Source Code

Ski Free Main Labkit Module:

.isHiddenCommentString

6.111 Final Project
Authors: Erik Stafl and Eric Grebing

Created From 6.111 FPGA Labkit -- Template Toplevel Module
For Labkit Revision 004

Created: October 31, 2004, from revision 003 file
Author: Nathan Ickes

DESCRIPTION

Provides functionality for the game "SkiFree", using a video color selector
for the game input, and displaying the video on a 1024x768 XVGA monitor. The
project uses the ZBT memory to store the input video in a frame buffer, so it
can be analyzed to determine the movement commands.

CHANGES FOR BOARD REVISION 004

1) Added signals for logic analyzer pods 2-4.
2) Expanded "tv_in_yrcrb" 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)

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

2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
"disp_data_out", "analyzer[2-3]_clock" and
"analyzer[2-3]_data".

2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
actually populated on the boards. (The boards support up to
256Mb devices, with 25 address lines.)

2004-Oct-31: Adapted to new revision 004 board.

2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
value. (Previous versions of this file declared this port to
be an input.)
module ski_free (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_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_vsnsync_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_aaff,
    tv_in_i2c_clock, tv_in_i2c_data, tv_in_i2c_read,
    tv_in_fifo0_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,
    raml_data, raml_address, raml_adv_ld, raml_clk, raml_cen_b,
    raml_ce_b, raml_oe_b, raml_we_b, raml_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;
in  put 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_vsnsync;
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;
in flash_sts;
output rs232_txd, rs232_rts;
in 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;
in disp_data_in;
output disp_data_out;
in button0, button1, button2, button3, button_enter, button_right,
button_left, button_down, button_up;
inout [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;
in 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_sync = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input
// VGA Output
//assign vga_out_red = 8'h0;
//assign vga_out_green = 8'h0;
//assign vga_out_blue = 8'h0;
//assign vga_out_sync_b = 1'b1;
//assign vga_out_blank_b = 1'b1;
//assign vga_out_pixel_clock = 1'b0;
//assign vga_out_hsync = 1'b0;
//assign vga_out_vsync = 1'b0;

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

// Video Input
//assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz;
// 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_we_b = 1'b1;
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 rami_data = 36'hZ;
assign rami_address = 19'h0;
assign rami_adv_ld = 1'b0;
assign rami_clk = 1'b0;
assign rami_cen_b = 1'b1;
assign rami_ce_b = 1'b1;
assign rami_oe_b = 1'b1;
assign rami_we_b = 1'b1;
assign rami_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

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

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs
// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
//assign disp_blank = 1'b1;
//assign disp_clock = 1'b0;
//assign disp_rs = 1'b0;
//assign disp_ce_b = 1'b1;
//assign disp_reset_b = 1'b0;
//assign disp_data_out = 1'b0;
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

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

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

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

 chaotic seqeunce generator
wire jump_button;

debounce db8(reset, clock_65mhz, ~button0, jump_button);

////////////////////////////////////////////////////////////////////
// Use hex display module for debugging
reg [63:0] dispdata;
display_16hex hexdisp1(reset, clock_65mhz, 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;
hvga xvga1(clock_65mhz,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;
zbt_6111 zbt1(clock_65mhz, 1'b1, vram_we, vram_addr,
  vram_write_data, vram_read_data,
  ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

////////////////////////////////////////////////////////////////////
// Generate pixel value from reading ZBT memory
// and detect colored pixels from this data to determine
// angle of the skier
wire [17:0] vr_pixel;
wire [18:0] vram_addr1;
wire [2:0] anglesignal;
reg [5:0] threshold;
display_detect vd1(reset,clock_65mhz,hcount,vcount,vr_pixel,
  vram_addr1,vram_read_data,switch[5],threshold,anglesignal);

// Use pushbutton up and down inputs to change display threshold
wire threshold_up, threshold_down;
steptopulse step1(reset, clock_65mhz, ~button_up, threshold_up);
steptopulse step2(reset, clock_65mhz, ~button_down, threshold_down);
always @ (posedge clock_65mhz) begin
  if ( reset )
    threshold <= 6'h2B;
  else if ( threshold_up )
    threshold <= threshold + 1;
  else if ( threshold_down )
    threshold <= threshold - 1;
end

////////////////////////////////////////////////////////////////////
// Convert Colors:
wire [9:0] Y, Cr, Cb;
wire [7:0] Rval, Gval, Bval;
assign Y = (vr_pixel[17:12], 4'b0);
assign Cr = (vr_pixel[11:6], 4'b0);
assign Cb = (vr_pixel[5:0], 4'b0);
wire [23:0] video_pixel = (Rval, Gval, Bval);
color_converter cv1( Rval, Gval, Bval, clock_65mhz, reset, Y, Cr, Cb );
// ADV7185 NTSC decoder interface code
// adv7185 initialization module
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
    .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
    .tv_in_i2c_clock(tv_in_i2c_clock),
    .tv_in_i2c_data(tv_in_i2c_data));

////////////////////////////////////////////////////////////////////
wire [29:0] ycrcb; // video data (luminance, chrominance)
wire [2:0] fvh; // sync for field, vertical, horizontal
wire dv; // data valid
ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
    .ycrcb(ycrcb), .f(fvh[2]),
    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

////////////////////////////////////////////////////////////////////
// Code to write NTSC data to video memory
// Convert 30-bit ycrcb signal to 18-bits, for 2 pixels/address.
wire [17:0] ycrcb_small;
assign ycrcb_small = {ycrcb[29:24], ycrcb[19:14], ycrcb[9:4]};
// Convert NTSC data into a format more easily stored in the RAM
wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire        ntsc_we;
ntsc_to_zbt n2z (clock_65mhz, tv_in_line_clock1, fvh, dv, ycrcb_small,
    ntsc_addr, ntsc_data, ntsc_we, switch[6]);

////////////////////////////////////////////////////////////////////
// Code to write black to ZBT memory
reg [31:0] count;
always @(posedge clock_65mhz)
    count <= reset ? 0 : count + 1;
wire [18:0] vram_addr2 = count[18:0];
wire [35:0] vblack = {2{6'd10,6'd6,6'd8}};

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

////////////////////////////////////////////////////////////////////
// Display Info on the hex display
always @(posedge clock_65mhz)
begin
    // Display Cr threshold and current angle on hex display
    dispdata <= {54'b0, threshold[5:0], 1'b0, anglesignal[2:0]};
end

////////////////////////////////////////////////////////////////////
// Code to handle game logic
// feed XVGA signals to SkiFree game
wire [23:0] game_pixel;
wire phsync,pvsync,pblank;
wire [24:0] skier_pixel;
wire [24:0] landscape_pixel;
wire [24:0] time_display_pixel;
wire [24:0] distance_display_pixel;
wire [24:0] slalom_left_pixel;
wire [24:0] slalom_right_pixel;
wire [24:0] left_boundary_gate_pixel;
wire [24:0] right_boundary_gate_pixel;
wire [24:0] ramp_pixel;
wire crash_mode;
wire detect_crashes;
wire [2:0] skier_orientation;
assign skier_orientation = anglesignal;
wire jump_mode;
wire tenth_hz_signal;
wire one_hz_signal;
wire ten_hz_signal;
wire hundred_hz_signal;
wire sixtieth_hz_signal;
wire left_gate_encountered;
wire right_gate_encountered;
wire missed_left_gate;
wire missed_right_gate;
wire [9:0] left_slalom_gate_x;
wire [9:0] left_slalom_gate_y;
wire [9:0] right_slalom_gate_x;
wire [9:0] right_slalom_gate_y;
wire [9:0] left_start_gate_y;
wire [9:0] ramp_one_x;
wire [9:0] ramp_one_y;
wire [9:0] ramp_two_x;
wire [9:0] ramp_two_y;
wire [9:0] skier_x;
wire [9:0] skier_y;
wire [10:0] distance_to_go;
wire timer_run;
wire course_started;
wire [3:0] led_val;
wire [1:0] distance_thousands;
wire [3:0] distance_hundreds;
wire [3:0] distance_tens;
wire [3:0] distance_ones;
wire left,right;
debounce db6(reset, clock_65mhz, ~button_left, left);
debounce db7(reset, clock_65mhz, ~button_right, right);
assign led = ~{4'b0,led_val[3:0]};
skier_video ski_vid(.clk(clock_65mhz),.reset(reset),.hcount(hcount),
    .vcount(vcount),.vsync(vsync),.left(left),.right(right),.up(jump_button),
    .crash_mode(crash_mode),.ramp_one_x(ramp_one_x),.ramp_one_y(ramp_one_y),
    .ramp_two_x(ramp_two_x),.ramp_two_y(ramp_two_y),.skier_pixel(skier_pixel),
    .skier_orientation(skier_orientation),.skier_x(skier_x),.skier_y(skier_y),
    .jump_mode(jump_mode));
landscape_video land_vid(.clk(clock_65mhz),.reset(reset),.hcount(hcount),.vcount(vcount),
    .vsync(vsync),.crash_mode(crash_mode),.skier_orientation(skier_orientation),
    .left_gate_encountered(left_gate_encountered),
    .right_gate_encountered(right_gate_encountered),
    .crash_mode(crash_mode),.jump_mode(jump_mode),
    .skier_x(skier_x),.skier_y(skier_y),
    .left_start_gate_y(left_start_gate_y),
    .ramp_one_x(ramp_one_x),.ramp_one_y(ramp_one_y),
    .ramp_two_x(ramp_two_x),.ramp_two_y(ramp_two_y),
    .skier_pixel(skier_pixel),
    .skier_orientation(skier_orientation),
    .left(left),.right(right),.up(jump_button),
    .crash_mode(crash_mode),.jump_mode(jump_mode));
.right_gate_encountered(right_gate_encountered), .missed_left_gate(missed_left_gate),
    .missed_right_gate(missed_right_gate), .timer_run(timer_run),

.landscape_pixel(landscape_pixel), .slalom_left_pixel(slalom_left_pixel),
    .slalom_right_pixel(slalom_right_pixel),
    .left_boundary_gate_pixel(left_boundary_gate_pixel),
    .right_boundary_gate_pixel(right_boundary_gate_pixel),
    .ramp_pixel(ramp_pixel), .detect_crashes(detect_crashes),
    .left_slalom_gate_x(left_slalom_gate_x),
    .left_slalom_gate_y(left_slalom_gate_y),
    .right_slalom_gate_x(right_slalom_gate_x),
    .right_slalom_gate_y(right_slalom_gate_y),

.ramp_one_y(ramp_one_y), .ramp_two_x(ramp_two_x), .ramp_two_y(ramp_two_y),

.left_start_gate_y(left_start_gate_y), .distance_to_go(distance_to_go),
    .course_started(course_started));

game_video_output game_vid_out(.clk(clock_65mhz), .hsync(hsync),
    .vsync(vsync), .blank(blank), .skier_pixel(skier_pixel), .landscape_pixel(landscape_pixel),
    .time_display_pixel(time_display_pixel), .slalom_left_pixel(slalom_left_pixel),

    .slalom_right_pixel(slalom_right_pixel), .left_boundary_gate_pixel(left_boundary_gate_pixel),
    .right_boundary_gate_pixel(right_boundary_gate_pixel), .ramp_pixel(ramp_pixel),
    .distance_display_pixel(distance_display_pixel), .phsync(phsync), .pvsync(pvsync),
    .pblank(pblank), .game_pixel_out(game_pixel));

collision_detection col_detect(.clk(clock_65mhz), .reset(reset), .vsync(vsync),
    .one_hz_signal(one_hz_signal), .skier_pixel_active(skier_pixel[24]),
    .landscape_pixel_active(landscape_pixel[24]), .detect_crashes(detect_crashes),
    .jump_mode(jump_mode), .crash_mode(crash_mode));

timer timer1(.clk(clock_65mhz), .reset(reset), .timer_run(timer_run),
    .sixtieth_hz_signal(sixtieth_hz_signal),
    .tenth_hz_signal(tenth_hz_signal), .missed_left_gate(missed_left_gate),
    .missed_right_gate(missed_right_gate), .one_hz_signal(one_hz_signal),
    .ten_hz_signal(ten_hz_signal), .hundred_hz_signal(hundred_hz_signal));

time_display time_disp1(.clk(clock_65mhz), .reset(reset), .vcount(vcount), .hcount(hcount),
    .sixtieth_hz_signal(sixtieth_hz_signal), .tenth_hz_signal(tenth_hz_signal), .hundred_hz_signal(hundred_hz_signal),
    .one_hz_signal(one_hz_signal), .ten_hz_signal(ten_hz_signal),
    .hundred_hz_signal(hundred_hz_signal), .timer_run(timer_run),
    .time_display_pixel(time_display_pixel));

slalom_gate_detect slalom_gate_det(.clk(clock_65mhz), .vsync(vsync), .vcount(vcount),
    .hcount(hcount), .skier_x(skier_x), .skier_y(skier_y),
    .left_slalom_gate_x(left_slalom_gate_x), .left_slalom_gate_y(left_slalom_gate_y),
    .right_slalom_gate_x(right_slalom_gate_x), .right_slalom_gate_y(right_slalom_gate_y),
    .left_gate_encountered(left_gate_encountered),
    .right_gate_encountered(right_gate_encountered), .missed_left_gate(missed_left_gate),
    .missed_right_gate(missed_right_gate), .led_val(led_val));

timer_toggle
timertog(.clk(clock_65mhz), .reset(reset), .left_start_gate_y(left_start_gate_y),
    .skier_y(skier_y), .course_started(course_started), .distance_to_go(distance_to_go),
    .timer_run(timer_run));
distance_tracker distance_track(.clk(clock_65mhz),.reset(reset),.vsync(vsync),
.distance_to_go(distance_to_go),.course_started(course_started),
.distance_thousands(distance_thousands),.distance_hundreds(distance_hundreds),
.distance_tens(distance_tens),.distance_ones(distance_ones));

distance_display distance_display1(.clk(clock_65mhz),.reset(reset),.vcount(vcount),
 .hcount(hcount),.distance_thousands(distance_thousands),
 .distance_hundreds(distance_hundreds),.distance_tens(distance_tens),.distance_ones(distance_ones),
.distance_to_go(distance_to_go),
.distance_display_pixel(distance_display_pixel));

////////////////////////////////////////////////////////////////////
// Final Video Image compositor

// Delay hcount and vcount by 6 to compensate for the pipeline
wire [10:0] hcount_delayed;
wire [9:0] vcount_delayed;

// This extra delay is necessary to match pipeline of video module
reg [23:0] rgb;
reg notb,hs,vs;
always @ (posedge clock_65mhz) begin
    hs <= phsync;
    vs <= pvsync;
    notb <= ~pblank;
    rgb <= game_pixel;
end

// Final Output Signals
wire [23:0] final_pixel;
wire final_hs, final_vs, final_b;

// Switch 0 shows overlay, switch 1 shows full screen video output
final_compositor cmp1(.reset(reset),.clk(clock_65mhz),.hcount(hcount),
 .vcount(vcount),.video_pixel(video_pixel),.game_pixel(rgb),
 .show_overlay(switch[0]),.full_video_output(switch[1]),
 .pixel_out(final_pixel),.hs_in(hs),.vs_in(vs),.blank_in(notb),
 .hs_out(final_hs),.vs_out(final_vs),.blank_out(final_b));

////////////////////////////////////////////////////////////////////
// Final Video output to VGA port

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

endmodule

Debounce Module:
////////////////////////////////////////////////////////////////////
// Pushbutton Debounce Module
// Makes sure that the button is stable for an acceptable amount of time before
// allowing the signal to propagate into the circuit
///
/// Inputs:
/// reset - the global or user reset signal to clear all previous data

Grebing, Stafl
27
clk - whatever clock signal the circuit is operating on
noisy - signal directly from the pushbutton

Outputs:
clean - the cleaned output that only shows true button pushes

Parameters:
NDELAY - how many clock cycles to delay the signal by
NBITS - number of bits required to count up to NDELAY

module debounce (reset, clk, noisy, clean);
  input reset, clk, noisy;
  output clean;
  parameter NDELAY = 650000;
  parameter NBITS = 20;
  reg [NBITS-1:0] count;
  reg xnew, clean;

  always @(posedge clk)
    if (reset) begin xnew <= noisy; clean <= noisy; count <= 0; end
    else if (noisy != xnew) begin xnew <= noisy; count <= 0; end
    else if (count == NDELAY) clean <= xnew;
    else count <= count+1;
endmodule
Timer Toggle Module:

```verilog
module timer_toggle(clk, reset, course_started, distance_to_go, timer_run);

input clk;
input reset;
input course_started;
input [10:0] distance_to_go;
output timer_run;

reg timer_run = 0;

always @(posedge clk) begin
  if (reset) timer_run <= 1; // timer restarts immediately after a reset
  if (course_started == 0)
    timer_run <= 1; // timer starts as soon as the image appears on screen
  else if (distance_to_go == 0)
    timer_run <= 0; // timer stops running once the end of the course is reached
end
endmodule
```

XVGA Module:

```verilog
module xvga(vclocak, hcount, vcount, hsync, vsync, blank);

input vclocak;
output [10:0] hcount;
output [9:0] vcount;
output vsync;
output hsync;
output blank;
```

Grebing, Stafl

29
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
Landscape Video Module:

// Landscape_video module
// Controls the output of sprites on the landscape, controls movement, and
determines the distance to go.

// Inputs:
// clk - 65 MHz clock signal, synchronous with pixels
// reset - user or global reset
// hcount[10:0] - horizontal (x) coordinate of current pixel
// vcount[9:0] - vertical (y) coordinate of current pixel
// crash_mode - indicates that a skier has just crashed into a landscape
// skier_orientation[2:0] - 3-bit value that indicates which direction
// skier is facing
// left_gate_encountered - 1 if the skier has passed the left slalom gate
// right_gate_encountered - 1 if the skier has passed the right slalom gate
// missed_left_gate - 1 if the skier is not to the left of the left slalom
// gate when he passes it
// missed_right_gate - 1 if the skier is not to the right of the right
// slalom gate when he passes it
// timer_run - 1 if the timer is running and the skier is on the course

// Outputs:
// landscape_pixel[24:0] - bit 24 indicates that a landscape object should
// be displayed on the
// current pixel bits [23:0]
// indicate the RGB value
// of the landscape object's
// pixel
// slalom_left_pixel[24:0] - bit 24 indicates that the left slalom gate should
// be displayed on the
// current pixel bits [23:0] indicate
// the RGB value of the
// left slalom gate's pixel
// slalom_right_pixel[24:0] - bit 24 indicates that the right slalom gate should
// be displayed on the
// current pixel bits [23:0] indicate
// the RGB value of the
// right slalom gate's pixel
// detect_crashes - 1 if the collision_detection module should detect crashes
// left_slalom_gate_x[9:0] - x and y coordinates of the two slalom gates are
// passed to the
// slalom_detection module
// left_slalom_gate_y[9:0]
// right_slalom_gate_x[9:0]
// right_slalom_gate_y[9:0]
// ramp_one_x[9:0] - x and y coordinates of the ramps - output to skier_video
// for jumping
// ramp_one_y[9:0]
// ramp_two_x[9:0]
// ramp_two_y[9:0]
// distance_to_go[10:0] - tells how many more 'meters' the skier has to go
down the hill (starts
at 2000)

// Parameters:
// BIG_TREE_WIDTH
// BIG_TREE_HEIGHT
// BARE_TREE_WIDTH
// BARE_TREE_HEIGHT
// TREE_STUMP_WIDTH
// TREE_STUMP_HEIGHT
// SLALOM_GATE_WIDTH
// SLALOM_GATE_HEIGHT
// START_GATE_WIDTH
module landscape_video(clk, reset, hcount, vcount, vsync, crash_mode, skier_orientation, left_gate_encountered, right_gate_encountered, missed_left_gate, missed_right_gate, timer_run, landscape_pixel, slalom_left_pixel, slalom_right_pixel, left_boundary_gate_pixel, right_boundary_gate_pixel, ramp_pixel, detect_crashes, left_slalom_gate_x, left_slalom_gate_y, right_slalom_gate_x, right_slalom_gate_y, ramp_one_x, ramp_one_y, ramp_two_x, ramp_two_y, distance_to_go, course_started);

input clk; // video clock
input reset;
input [10:0] hcount; // current x,y location of pixel
input [9:0] vcount;
input vsync; // XVGA vertical sync signal (active low)

input crash_mode;
input [2:0] skier_orientation; // indicates which direction the skier is facing (0 = -90, 1 = -60, 2 = -30, 3 = headon, 4 = 30, 5 = 60, 6 = 90)

input left_gate_encountered;
input right_gate_encountered;
input missed_left_gate;
input missed_right_gate;
input timer_run;

output [24:0] landscape_pixel; // landscape object pixel value output
output [24:0] slalom_left_pixel; // slalom gate pixel values
output [24:0] slalom_right_pixel;
output [24:0] left_boundary_gate_pixel;
output [24:0] right_boundary_gate_pixel;
output [24:0] ramp_pixel;

output detect_crashes;

output [9:0] left_slalom_gate_x;
output [9:0] left_slalom_gate_y;
output [9:0] right_slalom_gate_x;
output [9:0] right_slalom_gate_y;
output [9:0] ramp_one_x;
output [9:0] ramp_one_y;
output [9:0] ramp_two_x;
output [9:0] ramp_two_y;

output [10:0] distance_to_go;
output course_started;

// output pixels to be passed to the game_video_output module
reg [24:0] landscape_pixel = 0;
reg [24:0] slalom_left_pixel = 0;
reg [24:0] slalom_right_pixel = 0;
reg [24:0] left_boundary_gate_pixel = 0;
reg [24:0] right_boundary_gate_pixel = 0;
reg [24:0] ramp_pixel = 0;

// size parameters for the landscape objects
parameter BIG_TREE_WIDTH = 6'd32;
parameter BIG_TREE_HEIGHT = 7'd64;
parameter BARE_TREE_WIDTH = 5'd22;
parameter BARE_TREE_HEIGHT = 5'd27;

parameter TREE_STUMP_WIDTH = 5'd16;
parameter TREE_STUMP_HEIGHT = 4'd11;

parameter SLALOM_GATE_WIDTH = 4'd14;
parameter SLALOM_GATE_HEIGHT = 5'd25;

parameter START_GATE_WIDTH = 6'd42;
parameter START_GATE_HEIGHT = 5'd27;

parameter ROCK_WIDTH = 5'd23;
parameter ROCK_HEIGHT = 4'd11;

parameter RAMP_WIDTH = 5'd31;
parameter RAMP_HEIGHT = 4'd8;

// landscape object address registers
reg [11:0] big_tree_one_add = 0; // depth of 2048
reg [9:0] bare_tree_one_add = 0; // depth of 594
reg [7:0] tree_stump_one_add = 0; // depth of 176
reg [11:0] big_tree_two_add = 0; // depth of 2048
reg [9:0] bare_tree_two_add = 0; // depth of 594
reg [7:0] tree_stump_two_add = 0; // depth of 176
reg [11:0] big_tree_three_add = 0; // depth of 2048
reg [9:0] bare_tree_three_add = 0; // depth of 594
reg [7:0] tree_stump_three_add = 0; // depth of 176
reg [7:0] rock_one_add = 0; // depth of 253
reg [7:0] rock_two_add = 0; // depth of 253

// slalom gate and smile/frown sign address registers
reg [8:0] left_slalom_gate_add = 0; // depth of 350 for all roms
reg [8:0] right_slalom_gate_add = 0;
reg [8:0] left_smile_sign_add = 0;
reg [8:0] right_smile_sign_add = 0;
reg [8:0] left_frown_sign_add = 0;
reg [8:0] right_frown_sign_add = 0;

// start and finish gate address registers
reg [10:0] left_start_gate_add = 0;
reg [10:0] right_start_gate_add = 0;
reg [10:0] left_finish_gate_add = 0;
reg [10:0] right_finish_gate_add = 0;

// jump ramp address registers
reg [7:0] ramp_one_add = 0; // depth of 248
reg [7:0] ramp_two_add = 0; // depth of 248
reg [7:0] ramp_three_add = 0; // depth of 248

// landscape object data wires (24-bit RGB values)
wire [23:0] big_tree_one_data;
wire [23:0] bare_tree_one_data;
wire [23:0] tree_stump_one_data;
wire [23:0] big_tree_two_data;
wire [23:0] bare_tree_two_data;
wire [23:0] tree_stump_two_data;
wire [23:0] big_tree_three_data;
wire [23:0] bare_tree_three_data;
wire [23:0] tree_stump_three_data;
wire [23:0] rock_one_data;
wire [23:0] rock_two_data;

// slalom gate and smile/frown sign data wires (24-bit RGB values)
wire [23:0] left_slalom_gate_data;
wire [23:0] right_slalom_gate_data;
wire [23:0] left_smile_sign_data;
wire [23:0] right_smile_sign_data;
wire [23:0] left_frown_sign_data;
wire [23:0] right_frown_sign_data;
// start and finish gate data wires
wire [23:0] left_start_gate_data;
wire [23:0] right_start_gate_data;
wire [23:0] left_finish_gate_data;
wire [23:0] right_finish_gate_data;

// jump ramp data wire
wire [23:0] ramp_one_data;
wire [23:0] ramp_two_data;
wire [23:0] ramp_three_data;

// instantiate the landscape ROMs (same ROMs are instantiated multiple times)
big_tree_rom big_tree_one(big_tree_one_add, clk, big_tree_one_data);
bare_tree_rom bare_tree_one(bare_tree_one_add, clk, bare_tree_one_data);
tree_stump_rom tree_stump_one(tree_stump_one_add, clk, tree_stump_one_data);
big_tree_rom big_tree_two(big_tree_two_add, clk, big_tree_two_data);
bare_tree_rom bare_tree_two(bare_tree_two_add, clk, bare_tree_two_data);
tree_stump_rom tree_stump_two(tree_stump_two_add, clk, tree_stump_two_data);
big_tree_rom big_tree_three(big_tree_three_add, clk, big_tree_three_data);
bare_tree_rom bare_tree_three(bare_tree_three_add, clk, bare_tree_three_data);
tree_stump_rom tree_stump_three(tree_stump_three_add, clk, tree_stump_three_data);
rock_rom rock_one(rock_one_add, clk, rock_one_data);
rock_rom rock_two(rock_two_add, clk, rock_two_data);

// instantiate the slalom gate ROMs
left_gate_rom left_gate_one(left_slalom_gate_add, clk, left_slalom_gate_data);
right_gate_rom right_gate_one(right_slalom_gate_add, clk, right_slalom_gate_data);
smile_sign_rom left_smile_sign_one(left_smile_sign_add, clk, left_smile_sign_data);
smile_sign_rom right_smile_sign_one(right_smile_sign_add, clk, right_smile_sign_data);
frown_sign_rom left_frown_sign_one(left_frown_sign_add, clk, left_frown_sign_data);
frown_sign_rom right_frown_sign_one(right_frown_sign_add, clk, right_frown_sign_data);

// instantiate the start and finish gate ROMs
left_start_gate_rom left_start(left_start_gate_add, clk, left_start_gate_data);
right_start_gate_rom right_start(right_start_gate_add, clk, right_start_gate_data);
left_finish_gate_rom left_finish(left_finish_gate_add, clk, left_finish_gate_data);
right_finish_gate_rom right_finish(right_finish_gate_add, clk, right_finish_gate_data);

// instantiate the jump ramp ROMs
jump_ramp_rom ramp_one(ramp_one_add, clk, ramp_one_data);
jump_ramp_rom ramp_two(ramp_two_add, clk, ramp_two_data);
jump_ramp_rom ramp_three(ramp_three_add, clk, ramp_three_data);

// 1 if the system should detect crashes
reg detect_crashes = 1;

wire [10:0] sprite_hcount;
assign sprite_hcount = hcount;

// tracks the remaining distance to the end of the course
reg [10:0] distance_to_go = 2000;

// initial landscape object positions (x and y coordinates)
reg [9:0] big_tree_one_x = 500;
reg [9:0] big_tree_one_y = 600;

reg [9:0] bare_tree_one_x = 280;
reg [9:0] bare_tree_one_y = 570;

reg [9:0] tree_stump_one_x = 460;
reg [9:0] tree_stump_one_y = 250;

reg [9:0] big_tree_two_x = 320;
reg [9:0] big_tree_two_y = 400;

reg [9:0] bare_tree_two_x = 650;
reg [9:0] bare_tree_two_y = 340;

reg [9:0] tree_stump_two_x = 160;
reg [9:0] tree_stump_two_y = 450;
reg [9:0] big_tree_three_x = 800;
reg [9:0] big_tree_three_y = 450;
reg [9:0] bare_tree_three_x = 870;
reg [9:0] bare_tree_three_y = 240;
reg [9:0] tree_stump_three_x = 930;
reg [9:0] tree_stump_three_y = 100;
reg [9:0] rock_one_x = 100;
reg [9:0] rock_one_y = 50;
reg [9:0] rock_two_x = 520;
reg [9:0] rock_two_y = 340;
reg [9:0] ramp_one_x = 100;
reg [9:0] ramp_one_y = 250;
reg [9:0] ramp_two_x = 700;
reg [9:0] ramp_two_y = 650;
reg [9:0] ramp_three_x = 400;
reg [9:0] ramp_three_y = 350;

// initial slalom gate positions (x and y coordinates)
reg [9:0] left_slalom_gate_x = 300;
reg [9:0] left_slalom_gate_y = 100;
reg [9:0] right_slalom_gate_x = 450;
reg [9:0] right_slalom_gate_y = 600;

// initial start and finish gate positions (x and y coordinates)
reg [9:0] left_start_gate_x = 200;
reg [9:0] left_start_gate_y = 600;
reg [9:0] right_start_gate_x = 600;
reg [9:0] right_start_gate_y = 600;
reg [9:0] left_finish_gate_x = 100;
reg [9:0] left_finish_gate_y = 1000;
reg [9:0] right_finish_gate_x = 800;
reg [9:0] right_finish_gate_y = 1000;
reg [4:0] landscape_object = 31;
reg [4:0] landscape_object_delay_one = 31;
reg [4:0] landscape_object_delay_two = 31;

// indicates the skier's velocity down the hill in pixels per refresh cycle
reg [3:0] skier_vel = 4;

// indicates the horizontal change of each object depending on the skier's orientation
reg signed [6:0] horiz_change = 0;

// 1 if the system went into crash mode on the last refresh cycle
reg just_crashed = 0;

// 1 if the skier has entered the course through the start gates
reg course_started = 0;

reg [1:0] distance_counter = 1; // initialized to 1 because of delay
reg [10:0] hcount_delay_one = 0;
reg [10:0] hcount_delay_two = 0;
reg [9:0] vcount_delay_one = 0;
reg [9:0] vcount_delay_two = 0;
reg [11:0] multiply_stage = 0;

// Controls movement of the sprites
always @(posedge vsync) begin
  // Places all object in their original locations and sets the distance
  // back at 2000m to go upon reset
if (reset) begin
  big_tree_one_x <= 500;
  big_tree_one_y <= 600;
  bare_tree_one_x <= 280;
  bare_tree_one_y <= 570;
  tree_stump_one_x <= 460;
  tree_stump_one_y <= 250;
  big_tree_two_x <= 320;
  big_tree_two_y <= 400;
  bare_tree_two_x <= 650;
  bare_tree_two_y <= 340;
  tree_stump_two_x <= 160;
  tree_stump_two_y <= 450;
  big_tree_three_x <= 800;
  big_tree_three_y <= 450;
  bare_tree_three_x <= 870;
  bare_tree_three_y <= 240;
  tree_stump_three_x <= 930;
  tree_stump_three_y <= 100;
  rock_one_x <= 100;
  rock_one_y <= 50;
  rock_two_x <= 520;
  rock_two_y <= 340;
  ramp_one_x <= 100;
  ramp_one_y <= 250;
  ramp_two_x <= 700;
  ramp_two_y <= 650;
  ramp_three_x <= 400;
  ramp_three_y <= 350;
  left_slalom_gate_x <= 300;
  left_slalom_gate_y <= 100;
  right_slalom_gate_x <= 450;
  right_slalom_gate_y <= 600;
  distance_to_go <= 2000;
end
else begin
  if (crash_mode == 1) begin
    just_crashed <= 1;
    // Does not allow the system to detect any crashes once the skier
    // has already crashed
detect_crashes <= 0;
  end
else begin
  // indicates that the crash mode has just ended
  if (just_crashed == 1) begin
    // Shifts all landscape objects over 63 pixels to the right to
    // allow the skier a clear path after crashing
    horiz_change <= 7'b0111111;
    just_crashed <= 0;
  end
  // delays reintroduction of crash detection until the skier's position has
  // changed
  if (just_crashed == 0) detect_crashes <= 1;
  // Changes the horizontal and vertical positions of the landscape objects
  // and slalom gates based on the skier's velocity and the skier's angle
  left_slalom_gate_x <= left_slalom_gate_x + horiz_change;
  left_slalom_gate_y <= left_slalom_gate_y - skier_vel;
  right_slalom_gate_x <= right_slalom_gate_x + horiz_change;
  right_slalom_gate_y <= right_slalom_gate_y - skier_vel;
  // whenever landscape objects reach the top of the screen, resets their
  // position
  // so that they recycle and reappear at the bottom of the screen - the
  // different
  // values on each recycle causes the landscape to appear randomized

Grebing, Stafl 36
if (big_tree_one_y <= 5) big_tree_one_y <= 870;
else begin
    big_tree_one_x <= big_tree_one_x + horiz_change;
    big_tree_one_y <= big_tree_one_y - skier_vel;
end
if (bare_tree_one_y <= 5) bare_tree_one_y <= 800;
else begin
    bare_tree_one_x <= bare_tree_one_x + horiz_change;
    bare_tree_one_y <= bare_tree_one_y - skier_vel;
end
if (tree_stump_one_y <= 5) tree_stump_one_y <= 810;
else begin
    tree_stump_one_x <= tree_stump_one_x + horiz_change;
    tree_stump_one_y <= tree_stump_one_y - skier_vel;
end
if (big_tree_two_y <= 5) big_tree_two_y <= 960;
else begin
    big_tree_two_x <= big_tree_two_x + horiz_change;
    big_tree_two_y <= big_tree_two_y - skier_vel;
end
if (bare_tree_two_y <= 5) bare_tree_two_y <= 900;
else begin
    bare_tree_two_x <= bare_tree_two_x + horiz_change;
    bare_tree_two_y <= bare_tree_two_y - skier_vel;
end
if (tree_stump_two_y <= 5) tree_stump_two_y <= 800;
else begin
    tree_stump_two_x <= tree_stump_two_x + horiz_change;
    tree_stump_two_y <= tree_stump_two_y - skier_vel;
end
if (big_tree_three_y <= 5) big_tree_three_y <= 800;
else begin
    big_tree_three_x <= big_tree_three_x + horiz_change;
    big_tree_three_y <= big_tree_three_y - skier_vel;
end
if (bare_tree_three_y <= 5) bare_tree_three_y <= 800;
else begin
    bare_tree_three_x <= bare_tree_three_x + horiz_change;
    bare_tree_three_y <= bare_tree_three_y - skier_vel;
end
if (tree_stump_three_y <= 5) tree_stump_three_y <= 768;
else begin
    tree_stump_three_x <= tree_stump_three_x + horiz_change;
    tree_stump_three_y <= tree_stump_three_y - skier_vel;
end
if (rock_one_y <= 5) rock_one_y <= 780;
else begin
    rock_one_x <= rock_one_x + horiz_change;
    rock_one_y <= rock_one_y - skier_vel;
end
if (rock_two_y <= 5) rock_two_y <= 940;
else begin
    rock_two_x <= rock_two_x + horiz_change;
    rock_two_y <= rock_two_y - skier_vel;
end
if (ramp_one_y <= 5) ramp_one_y <= 860;
else begin
    ramp_one_x <= ramp_one_x + horiz_change;
    ramp_one_y <= ramp_one_y - skier_vel;
end
if (ramp_two_y <= 5) ramp_two_y <= 790;
else begin
    ramp_two_x <= ramp_two_x + horiz_change;
    ramp_two_y <= ramp_two_y - skier_vel;
end
if (ramp_three_y <= 5) ramp_three_y <= 800;
else begin
    ramp_three_x <= ramp_three_x + horiz_change;
    ramp_three_y <= ramp_three_y - skier_vel;
end
// When the skier is not crashed and the timer is running, the distance
// decreases by 1 each vsync
if (timer_run && (skier_orientation != 0) && (skier_orientation != 6))
    distance_to_go <= distance_to_go + 1;

if (course_started == 0) begin
    left_start_gate_x <= left_start_gate_x + horiz_change;
    left_start_gate_y <= left_start_gate_y - skier_vel;
    right_start_gate_x <= right_start_gate_x + horiz_change;
    right_start_gate_y <= right_start_gate_y - skier_vel;
    if (timer_run && left_start_gate_y >= 768) course_started <= 1;
end
else begin
    // guarantees that these gates will not appear on screen again
    left_start_gate_y <= 1000;
    right_start_gate_y <= 1000;
end

if (distance_to_go <= 50) begin // when skier nears the end of the course
    left_finish_gate_x <= left_finish_gate_x + horiz_change;
    left_finish_gate_y <= left_finish_gate_y - skier_vel;
    right_finish_gate_x <= right_finish_gate_x + horiz_change;
    right_finish_gate_y <= right_finish_gate_y - skier_vel;
end

if (just_crashed == 0) begin
    case (skier_orientation)
        0 : begin
            horiz_change <= 0; // 0
            skier_vel <= 0;
        end
        1 : begin
            horiz_change <= 7'b0000010; // 2
            skier_vel <= 4;
        end
        2 : begin
            horiz_change <= 7'b0000001; // 1
            skier_vel <= 4;
        end
        3 : begin
            horiz_change <= 7'b0000000; // 0
            skier_vel <= 4;
        end
        4 : begin
            horiz_change <= 7'b1111111; // -1
            skier_vel <= 4;
        end
        5 : begin
            horiz_change <= 7'b1111110; // -2
            skier_vel <= 4;
        end
        6 : begin
            horiz_change <= 0; // 0
            skier_vel <= 0;
        end
default : begin
            horiz_change <= 0;
            skier_vel <= 0;
        end
    endcase
end

always @(posedge clk) begin
    // sets landscape and ramp pixels inactive by default
    landscape_pixel[24] <= 0;
    landscape_pixel[23:0] <= 24'hFFFFFFFF;
ramp_pixel[24] <= 0;
ramp_pixel[23:0] <= 24'hFFFFFF;

// delays set to be used in pipeline
hcount_delay_one <= hcount;
vcount_delay_one <= vcount;
hcount_delay_two <= hcount_delay_one;
vcount_delay_two <= vcount_delay_one;
landscape_object_delay_one <= landscape_object;
landscape_object_delay_two <= landscape_object_delay_one;

// first stage of the pipeline checks the position of different landscape objects
and compares

// it to the current vcount and sprite_hcount

if ((vcount >= big_tree_one_y && vcount < big_tree_one_y + BIG_TREE_HEIGHT) &&
    (sprite_hcount >= big_tree_one_x && sprite_hcount < big_tree_one_x +
    BIG_TREE_WIDTH))
landscape_object <= 1;
else if ((vcount >= bare_tree_one_y && vcount < bare_tree_one_y +
    BARE_TREE_HEIGHT) &&
    (sprite_hcount >= bare_tree_one_x && sprite_hcount < bare_tree_one_x +
    BARE_TREE_WIDTH))
landscape_object <= 2;
else if ((vcount >= tree_stump_one_y && vcount < tree_stump_one_y +
    TREE_STUMP_HEIGHT) &&
    (sprite_hcount >= tree_stump_one_x && sprite_hcount < tree_stump_one_x +
    TREE_STUMP_WIDTH))
landscape_object <= 3;
else if ((vcount >= big_tree_two_y && vcount < big_tree_two_y + BIG_TREE_HEIGHT) &&
    (sprite_hcount >= big_tree_two_x && sprite_hcount < big_tree_two_x +
    BIG_TREE_WIDTH))
landscape_object <= 4;
else if ((vcount >= bare_tree_two_y && vcount < bare_tree_two_y +
    BARE_TREE_HEIGHT) &&
    (sprite_hcount >= bare_tree_two_x && sprite_hcount < bare_tree_two_x +
    BARE_TREE_WIDTH))
landscape_object <= 5;
else if ((vcount >= tree_stump_two_y && vcount < tree_stump_two_y +
    TREE_STUMP_HEIGHT) &&
    (sprite_hcount >= tree_stump_two_x && sprite_hcount < tree_stump_two_x +
    TREE_STUMP_WIDTH))
landscape_object <= 6;
else if ((vcount >= big_tree_three_y && vcount < big_tree_three_y +
    BIG_TREE_HEIGHT) &&
    (sprite_hcount >= big_tree_three_x && sprite_hcount < big_tree_three_x +
    BIG_TREE_WIDTH))
landscape_object <= 7;
else if ((vcount >= bare_tree_three_y && vcount < bare_tree_three_y +
    BARE_TREE_HEIGHT) &&
    (sprite_hcount >= bare_tree_three_x && sprite_hcount < bare_tree_three_x +
    BARE_TREE_WIDTH))
landscape_object <= 8;
else if ((vcount >= tree_stump_three_y && vcount < tree_stump_three_y +
    TREE_STUMP_HEIGHT) &&
    (sprite_hcount >= tree_stump_three_x && sprite_hcount < tree_stump_three_x +
    TREE_STUMP_WIDTH))
landscape_object <= 9;
else if ((vcount >= rock_one_y && vcount < rock_one_y + ROCK_HEIGHT) &&
    (sprite_hcount >= rock_one_x && sprite_hcount < rock_one_x + ROCK_WIDTH))
landscape_object <= 10;
else if ((vcount >= rock_two_y && vcount < rock_two_y + ROCK_HEIGHT) &&
  (sprite_hcount >= rock_two_x && sprite_hcount < rock_two_x + ROCK_WIDTH))
landscape_object <= 11;
else if ((vcount >= ramp_one_y && vcount < ramp_one_y + RAMP_HEIGHT) &&
  (sprite_hcount >= ramp_one_x && sprite_hcount < ramp_one_x + RAMP_WIDTH))
landscape_object <= 12;
else if ((vcount >= ramp_two_y && vcount < ramp_two_y + RAMP_HEIGHT) &&
  (sprite_hcount >= ramp_two_x && sprite_hcount < ramp_two_x + RAMP_WIDTH))
landscape_object <= 13;

// assigns the slalom gates to either be a flag, frowning sign, or smiling sign
// depending on if
// the gates have been encountered and whether or not
else if ((vcount >= left_slalom_gate_y && vcount < left_slalom_gate_y +
  SLALOM_GATE_HEIGHT) &&
  (sprite_hcount >= left_slalom_gate_x && sprite_hcount < left_slalom_gate_x +
  SLALOM_GATE_WIDTH))
begin
  if (!left_gate_encountered)
    landscape_object <= 14;
else
begin
  if (missed_left_gate)
    landscape_object <= 15;
  else
    landscape_object <= 16;
end
end

else if ((vcount >= right_slalom_gate_y && vcount < right_slalom_gate_y +
  SLALOM_GATE_HEIGHT) &&
  (sprite_hcount >= right_slalom_gate_x && sprite_hcount <
  right_slalom_gate_x + SLALOM_GATE_WIDTH))
begin
  if (!right_gate_encountered)
    landscape_object <= 17;
else
begin
  if (missed_right_gate)
    landscape_object <= 18;
  else
    landscape_object <= 19;
end
end

else landscape_object <= 0;

// the second stage of the pipeline calculates the multiplication term for the
determination of an
// image’s address - the landscape_object variable uniquely indexes each landscape
sprite

case (landscape_object)
1 : multiply_stage <= (vcount_delay_one-big_tree_one_y)*BIG_TREE_WIDTH;
2 : multiply_stage <= (vcount_delay_one-bare_tree_one_y)*BARE_TREE_WIDTH;
3 : multiply_stage <= (vcount_delay_one-tree_stump_one_y)*TREE_STUMP_WIDTH;
4 : multiply_stage <= (vcount_delay_one-big_tree_two_y)*BIG_TREE_WIDTH;
5 : multiply_stage <= (vcount_delay_one-bare_tree_two_y)*BARE_TREE_WIDTH;
6 : multiply_stage <= (vcount_delay_one-tree_stump_two_y)*TREE_STUMP_WIDTH;
7 : multiply_stage <= (vcount_delay_one-big_tree_three_y)*BIG_TREE_WIDTH;
8 : multiply_stage <= (vcount_delay_one-bare_tree_three_y)*BARE_TREE_WIDTH;
9 : multiply_stage <= (vcount_delay_one-
  tree_stump_three_y)*TREE_STUMP_WIDTH;
10 : multiply_stage <= (vcount_delay_one-rock_one_y)*ROCK_WIDTH;
11 : multiply_stage <= (vcount_delay_one-rock_two_y)*ROCK_WIDTH;
12 : multiply_stage <= (vcount_delay_one-ramp_one_y)*RAMP_WIDTH;
13 : multiply_stage <= (vcount_delay_one-ramp_two_y)*RAMP_WIDTH;
14 : multiply_stage <= (vcount_delay_one-
  left_slalom_gate_y)*SLALOM_GATE_WIDTH;
15 : multiply_stage <= (vcount_delay_one-
  left_slalom_gate_y)*SLALOM_GATE_WIDTH;
16 : multiply_stage <= (vcount_delay_one-
  left_slalom_gate_y)*SLALOM_GATE_WIDTH;
17 : multiply_stage <= (vcount_delay_one-
  right_slalom_gate_y)*SLALOM_GATE_WIDTH;
18 : multiply_stage <= (vcount_delay_one-
  right_slalom_gate_y)*SLALOM_GATE_WIDTH;
19 : multiply_stage <= (vcount_delay_one-
  right_slalom_gate_y)*SLALOM_GATE_WIDTH;
default : multiply_stage <= 0;

endcase

// the third stage of the pipeline determines the address of each object based on
// the delayed version of landscape_object
case (landscape_object_delay_one)
  1 : big_tree_one_add <= multiply_stage + hcount_delay_two-big_tree_one_x;
  2 : bare_tree_one_add <= multiply_stage + hcount_delay_two-bare_tree_one_x;
  3 : tree_stump_one_add <= multiply_stage + hcount_delay_two-
    tree_stump_one_x;
  4 : big_tree_two_add <= multiply_stage + hcount_delay_two-big_tree_two_x;
  5 : bare_tree_two_add <= multiply_stage + hcount_delay_two-bare_tree_two_x;
  6 : tree_stump_two_add <= multiply_stage + hcount_delay_two-
    tree_stump_two_x;
  7 : big_tree_three_add <= multiply_stage + hcount_delay_two-
    big_tree_three_x;
  8 : bare_tree_three_add <= multiply_stage + hcount_delay_two-
    bare_tree_three_x;
  9 : tree_stump_three_add <= multiply_stage + hcount_delay_two-
    tree_stump_three_x;
 10 : rock_one_add <= multiply_stage + hcount_delay_two-rock_one_x;
 11 : rock_two_add <= multiply_stage + hcount_delay_two-rock_two_x;
 12 : ramp_one_add <= multiply_stage + hcount_delay_two-ramp_one_x;
 13 : ramp_two_add <= multiply_stage + hcount_delay_two-ramp_two_x;
 14 : left_slalom_gate_add <= multiply_stage + hcount_delay_two-
    left_slalom_gate_x;
 15 : left_frown_sign_add <= multiply_stage + hcount_delay_two-
    left_slalom_gate_x;
 16 : left_smile_sign_add <= multiply_stage + hcount_delay_two-
    left_slalom_gate_x;
 17 : right_slalom_gate_add <= multiply_stage + hcount_delay_two-
    right_slalom_gate_x;
 18 : right_frown_sign_add <= multiply_stage + hcount_delay_two-
    right_slalom_gate_x;
 19 : right_smile_sign_add <= multiply_stage + hcount_delay_two-
    right_slalom_gate_x;
endcase

// the final stage of the pipeline sets the type of the object's pixel active and
// the rest inactive
// the appropriate pixel is also given data from the appropriate rom

case (landscape_object_delay_two)
  1 : begin
    landscape_pixel[24] <= 1;
    ramp_pixel[24] <= 0;
    slalom_left_pixel[24] <= 0;
    slalom_right_pixel[24] <= 0;
    landscape_pixel[23:0] <= big_tree_one_data;
  end
  2 : begin
    landscape_pixel[24] <= 1;
    ramp_pixel[24] <= 0;
    slalom_left_pixel[24] <= 0;
    slalom_right_pixel[24] <= 0;
    landscape_pixel[23:0] <= bare_tree_one_data;
  end
  3 : begin
    landscape_pixel[24] <= 1;
    ramp_pixel[24] <= 0;
    slalom_left_pixel[24] <= 0;
    slalom_right_pixel[24] <= 0;
    landscape_pixel[23:0] <= tree_stump_one_data;
  end
  4 : begin
    landscape_pixel[24] <= 1;
    ramp_pixel[24] <= 0;
    slalom_left_pixel[24] <= 0;
    slalom_right_pixel[24] <= 0;
    landscape_pixel[23:0] <= big_tree_two_data;
  end
  5 : begin
  end
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= bare_tree_two_data;
end
6 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= tree_stump_two_data;
end
7 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= big_tree_three_data;
end
8 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= bare_tree_three_data;
end
9 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= tree_stump_three_data;
end
10 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
ramp_pixel[23:0] <= ramp_one_data;
end
11 : begin
landscape_pixel[24] <= 1;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
landscape_pixel[23:0] <= ramp_two_data;
end
12 : begin
ramp_pixel[24] <= 1;
landscape_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
ramp_pixel[23:0] <= ramp_one_data;
end
13 : begin
ramp_pixel[24] <= 1;
landscape_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
ramp_pixel[23:0] <= ramp_two_data;
end
14 : begin
slalom_left_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
slalom_left_pixel[23:0] <= left_slalom_gate_data;
end
15 : begin
slalom_left_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
slalom_left_pixel[23:0] <= left_frown_sign_data;

16 : begin
   slalom_left_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
slalom_left_pixel[23:0] <= left_smile_sign_data;
end

17 : begin
   slalom_right_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[23:0] <= right_slalom_gate_data;
end

18 : begin
   slalom_right_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[23:0] <= right_frown_sign_data;
end

19 : begin
   slalom_right_pixel[24] <= 1;
ramp_pixel[24] <= 0;
landscape_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[23:0] <= right_smile_sign_data;
end

default : begin
   landscape_pixel[24] <= 0;
ramp_pixel[24] <= 0;
slalom_left_pixel[24] <= 0;
slalom_right_pixel[24] <= 0;
end

endcase

// reset landscape object addresses
if (vcount == 0) begin
   big_tree_one_add <= 0;
bare_tree_one_add <= 0;
tree_stump_one_add <= 0;
big_tree_two_add <= 0;
bare_tree_two_add <= 0;
tree_stump_two_add <= 0;
big_tree_three_add <= 0;
bare_tree_three_add <= 0;
tree_stump_three_add <= 0;
rock_one_add <= 0;
rock_two_add <= 0;
ramp_one_add <= 0;
ramp_two_add <= 0;
ramp_three_add <= 0;
left_slalom_gate_add <= 0;
right_slalom_gate_add <= 0;
left_smile_sign_add <= 0;
right_smile_sign_add <= 0;
left_frown_sign_add <= 0;
right_frown_sign_add <= 0;
end

endmodule // landscape_video

Distance Display Module:

Grebing, Stafl
// Distance_display module
// Takes in signals from the distance_tracker module to control the output of
// the distance value on the screen. Outputs the value of the current pixel
// corresponding to the distance to the game_video_output module.
// Inputs:
// clk - 65 MHz clock signal, synchronous with pixels
// reset - user or global reset
// hcount[10:0] - horizontal (x) coordinate of current pixel
// vcount[9:0] - vertical (y) coordinate of current pixel
// distance_thousands[1:0] - holds value of the thousands digit of the distance
// remaining
// distance_hundreds[3:0] - holds value of the hundreds digit of the distance
// remaining
// distance_tens[3:0] - holds value of the tens digit of the distance remaining
// distance_ones[3:0] - holds value of the ones digit of the distance remaining
// distance_to_go[10:0] - number of 'meters' remaining in the course
// (generated by the landscape_video module)
// Outputs:
// distance_display_pixel[24:0] - Bit 24 signifies that the screen should display
// a distance number for the current hcount/vcount
// - Bits [23:0] are passed as the RGB values
// of the XVGA display
// module distance_display(clk,reset,vcount,hcount,distance_thousands,distance_hundreds,
// distance_tens,distance_ones,distance_to_go,distance_display_pixel);

input clk;
input reset;
input [9:0] vcount;
input [10:0] hcount;
input [1:0] distance_thousands;
input [3:0] distance_hundreds;
input [3:0] distance_tens;
input [3:0] distance_ones;
input [10:0] distance_to_go;
output [24:0] distance_display_pixel;

// x and y coordinates for distance display values
reg [9:0] display_dist_thousand_x = 924;
reg [9:0] display_dist_thousand_y = 25;

reg [9:0] display_dist_hundred_x = 933;
reg [9:0] display_dist_hundred_y = 25;

reg [9:0] display_dist_ten_x = 942;
reg [9:0] display_dist_ten_y = 25;

reg [9:0] display_dist_one_x = 951;
reg [9:0] display_dist_one_y = 25;

// distance display rom address registers
reg [9:0] thousand_dist_add = 0;
reg [9:0] hundred_dist_add = 0;
reg [9:0] ten_dist_add = 0;
reg [9:0] one_dist_add = 0;

// module's output pixel register - bit 24 means active, 23:0 holds RGB value
reg [24:0] distance_display_pixel = 0;

// hcount used for sprites to compensate for system delays
wire [10:0] sprite_hcount;
assign sprite_hcount = hcount - 1;
// data wires for pixel information from roms - 1 bit because the images are monochrome
wire thousand_dist_data;
wire hundred_dist_data;
wire ten_dist_data;
wire one_dist_data;

// digit sizes in pixels
parameter DIGIT_WIDTH = 8;
parameter DIGIT_HEIGHT = 11;

// hcount and vcount registers for use in the pipeline
reg [10:0] hcount_delay_one = 0;
reg [10:0] hcount_delay_two = 0;
reg [9:0] vcount_delay_one = 0;

reg [9:0] multiply_stage = 0;

// variables to control case statements in the pipeline -
// the value corresponds to the actual value to be displayed
reg [2:0] digit_to_display = 0;
reg [2:0] digit_to_display_delay_one = 0;
reg [2:0] digit_to_display_delay_two = 0;

always @(posedge clk) begin
    // set the distance_display_pixel to default inactive
    distance_display_pixel[24] <= 0;
    distance_display_pixel[23:0] <= 24'hFFFFFF;

    // controls the delays of various signals for the pipeline
    digit_to_display_delay_one <= digit_to_display;
    digit_to_display_delay_two <= digit_to_display_delay_one;
    hcount_delay_one <= hcount;
    hcount_delay_two <= hcount_delay_one;
    vcount_delay_one <= vcount;

    // series of statements determines if particular digits should be displayed at the
current sprite_hcount
    // and vcount
    if ((vcount >= display_dist_thousand_y && vcount < display_dist_thousand_y +
        DIGIT_HEIGHT) &&
        (sprite_hcount >= display_dist_thousand_x && sprite_hcount <
        display_dist_thousand_x + DIGIT_WIDTH))
        digit_to_display <= 4;
    else if ((vcount >= display_dist_hundred_y && vcount < display_dist_hundred_y +
        DIGIT_HEIGHT) &&
        (sprite_hcount >= display_dist_hundred_x && sprite_hcount <
        display_dist_hundred_x + DIGIT_WIDTH))
        digit_to_display <= 3;
    else if ((vcount >= display_dist_ten_y && vcount < display_dist_ten_y +
        DIGIT_HEIGHT) &&
        (sprite_hcount >= display_dist_ten_x && sprite_hcount < display_dist_ten_x +
        DIGIT_WIDTH))
        digit_to_display <= 2;
    else if ((vcount >= display_dist_one_y && vcount < display_dist_one_y +
        DIGIT_HEIGHT) &&
        (sprite_hcount >= display_dist_one_x && sprite_hcount < display_dist_one_x +
        DIGIT_WIDTH))
        digit_to_display <= 1;
    else digit_to_display <= 0;

    // second stage of the pipeline separates one multiply operation from the
determination
    // of the rom’s address - if digit_to_display is zero, no digits will be

Grebing, Stafl
case (digit_to_display)
  1 : multiply_stage <= (vcount_delay_one-display_dist_one_y + distance_ones*DIGIT_HEIGHT);
  2 : multiply_stage <= (vcount_delay_one-display_dist_thousand_y + distance_tens*DIGIT_HEIGHT);
  3 : multiply_stage <= (vcount_delay_one-display_dist_hundred_y + distance_hundreds*DIGIT_HEIGHT);
  4 : multiply_stage <= (vcount_delay_one-display_dist_thousand_y + distance_thousands*DIGIT_HEIGHT);
  default : multiply_stage <= 0;
endcase

// third stage of the pipeline uses the multiply_stage value to help determine the
// address for the
delayed hcount and vcount

// Display and Detect Module:

// Display and Detect Module
// Generate display pixels from reading the ZBT ram and select the desired
// pixels and use them to determine the requested angle of the skier
//
// Note: The ZBT ram has 2 cycles of read (and write) latency
// This is taken care of by latching the data an an appropriate time.
//
// Note: The ZBT stores 36 bits per word, so we store two pixels (18-bits each)
// in each address location.
//
// Inputs:
// reset - any reset signal
// clk - 65 MHz clock signal
// hcount[10:0] - horizontal (x) coordinate of current pixel
// vcount[9:0] - vertical (y) coordinate of current pixel
// vram_read_data[35:0] - Read data from the ZBT ram, two pixels/addr
// contrast_mode - Draw selected pixel color red if true
threshold[5:0] - Middle 6 bits for chroma key threshold
// Outputs:
vr_pixel[17:0] - The output pixel value in ycrcb_small format
vram_addr[18:0] - Address to read from: {vcount[9:0], hcount[9:1]}
anglesignal[2:0] - The current requested angle for the skier

module display_detect(reset, clk, hcount, vcount, vr_pixel, vram_addr, vram_read_data, contrast_mode, threshold, anglesignal);

input reset, clk;
input [10:0] hcount;
input [9:0] vcount;
output [17:0] vr_pixel;
output [18:0] vram_addr;
input [35:0] vram_read_data;
input contrast_mode;
output [2:0] anglesignal;
input [5:0] threshold;

wire [18:0] vram_addr = {vcount, hcount[9:1]};
wire [1:0] hc4 = hcount[1:0];
reg [17:0] vr_pixel;
reg [35:0] vr_data_latched;
reg [35:0] last_vr_data;

always @ (posedge clk)
last_vr_data <= (hc4[0]==1'b1) ? vr_data_latched : last_vr_data;
always @ (posedge clk)
vr_data_latched <= (hc4[0]==1'b0) ? vram_read_data : vr_data_latched;
reg [17:0] vr_pixel_temp;

always @ (*) // each 36-bit word from RAM is decoded to two 18-bit pixels
if (hc4[0])
    vr_pixel_temp = last_vr_data[35:18];
else
    vr_pixel_temp = last_vr_data[17:0];

// Handle Color Detection by cr value
// hcount bounds are necessary to ignore pixels at far left that should not be there
wire selected = (contrast_mode && (hcount > 3) && (hcount < 1024) && (vcount > 3) &&
    (vcount < 768) &&
    ({2'b10, vr_pixel_temp[11:6], 2'b00} >
    {2'b10, threshold[5:0], 2'b00}) >
    {2'b10, threshold[5:0], 2'b00});

// Wires for the various average values
wire [10:0] haverage;
wire [9:0] vaverage;
wire [10:0] leftave_x, rightave_x, topave_x, bottomave_x;
wire [9:0] leftave_y, rightave_y, topave_y, bottomave_y;

always @ (*) begin
    vr_pixel = (6'b111100, 6'b111100, 6'b111110);
    vr_pixel = (6'b111100, 6'b001000, 6'b111110);
    vr_pixel = (6'b111110, 6'b111110, 6'b001100);
    vr_pixel = (6'b111010, 6'b001110, 6'b111110);
    vr_pixel = (6'b001100, 6'b001110, 6'b110010);
else if ( selected )
    vr_pixel = (6'b000100, 6'b100000, 6'b100000);
else
    vr_pixel = vr_pixel_temp;
end
// Send current pixel info to centerofmass module and receive COM for old frame
centerofmass com1(reset, clk, hcount, vcount, selected, haverage, vaverage);

// Send current pixel and COM info and to the angledetect module for command recognition
angledetect ad1(reset, clk, hcount, vcount, selected, haverage, vaverage,
leftave_x, leftave_y, rightave_x, rightave_y, topave_x,
topave_y, bottomave_x, bottomave_y, anglesignal);

endmodule

**Skier Video Module:**

```vhdl
module skier_video(clk, reset, hcount, vcount, vsync, left, right, up, ramp_one_x, ramp_one_y,
ramp_two_x, ramp_two_y, crash_mode, skier_orientation, ramp_one_x, ramp_one_y,
ramp_two_x, ramp_two_y, skier_pixel, skier_x, skier_y, jump_mode);
```

```vhdl
input clk; // video clock
input [10:0] hcount; // current x, y location of pixel
input [9:0] vcount;
input vsync; // XVGA vertical sync signal (active low)
input reset;
```

Grebing, Stafl 48
input left;       // temporarily controls skier orientation (until controller is completed)
input right;
input up;       // triggers a jump

// coordinate locations of the ramps for use in jumping
input [9:0] ramp_one_x;
input [9:0] ramp_one_y;
input [9:0] ramp_two_x;
input [9:0] ramp_two_y;

input crash_mode;       // 1 if the skier is currently crashed

input [2:0] skier_orientation; // will be the signal from the controller to determine angular position

// bit 24 tells if the skier should be displayed at the current location
// bits 23:0 tell the current RGB value of the skier pixel
output [24:0] skier_pixel;

// skier's x and y coordinate position
output [9:0] skier_x;
output [9:0] skier_y;

output jump_mode;

// set of parameters for the size of each skier sprite
parameter SKIER_HEADON_WIDTH = 15;
parameter SKIER_HEADON_HEIGHT = 32;

parameter SKIER_POS_30_WIDTH = 16;
parameter SKIER_POS_30_HEIGHT = 31;

parameter SKIER_POS_60_WIDTH = 22;
parameter SKIER_POS_60_HEIGHT = 28;

parameter SKIER_POS_90_WIDTH = 23;
parameter SKIER_POS_90_HEIGHT = 26;

parameter SKIER_CRASHED_WIDTH = 31;
parameter SKIER_CRASHED_HEIGHT = 30;

parameter SKIER_JUMPING_WIDTH = 32;
parameter SKIER_JUMPING_HEIGHT = 32;

parameter SKIER_GROUNDED_X = 400;
parameter SKIER_GROUNDED_Y = 250;

parameter RAMP_WIDTH = 31;

reg [2:0] skier_orientation_delay_one = 0;
reg [2:0] skier_orientation_delay_two = 0;
reg [2:0] skier_orientation_delay_three = 0;
reg jumper_in_range = 0;
reg jumper_in_range_delay_one = 0;
reg jumper_in_range_delay_two = 0;
reg crash_in_range = 0;
reg crash_in_range_delay_one = 0;
reg crash_in_range_delay_two = 0;
reg skier_in_range = 0;
reg skier_in_range_delay_one = 0;
reg skier_in_range_delay_two = 0;
reg [9:0] vcount_delay_one = 0;
reg [9:0] vcount_delay_two = 0;
reg [10:0] hcount_delay_one = 0;
reg [10:0] hcount_delay_two = 0;

reg [11:0] multiply_stage = 0;

reg [24:0] skier_pixel;       // 24 tells if skier is active for a particular pixel;
                            // [23:0] stores RGB value for the skier's current pixel

Grebing, Stafl
wire [10:0] sprite_hcount; // delays the sprite's hcount to compensate for clock delay
assign sprite_hcount = hcount - 1;

// skier address registers
reg [8:0] skier_headon_add = 0; // depth of 480
reg [9:0] skier_left_90_add = 0; // depth of 598
reg [9:0] skier_left_60_add = 0; // depth of 616
reg [8:0] skier_left_30_add = 0; // depth of 496
reg [9:0] skier_right_90_add = 0; // depth of 598
reg [9:0] skier_right_60_add = 0; // depth of 616
reg [8:0] skier_right_30_add = 0; // depth of 496
reg [9:0] skier_crashed_add = 0; // depth of 930
reg [10:0] skier_jumping_add = 0; // depth of 1024

// skier data wires - each passes a 24-bit RGB value from an image ROM
wire [23:0] skier_headon_data;
wire [23:0] skier_left_90_data;
wire [23:0] skier_left_60_data;
wire [23:0] skier_left_30_data;
wire [23:0] skier_right_90_data;
wire [23:0] skier_right_60_data;
wire [23:0] skier_right_30_data;
wire [23:0] skier_crashed_data;
wire [23:0] skier_jumping_data;

// instantiate the skier roms - each of the left facing skiers are just reflected versions of
// the right facing
// version
skier_left_90_rom skier0(skier_left_90_add,clk,skier_left_90_data);
skier_left_60_rom skier1(skier_left_60_add,clk,skier_left_60_data);
skier_left_30_rom skier2(skier_left_30_add,clk,skier_left_30_data);
skier_headon_rom skier3(skier_headon_add,clk,skier_headon_data);
skier_right_30_rom skier4(skier_right_30_add,clk,skier_right_30_data);
skier_right_60_rom skier5(skier_right_60_add,clk,skier_right_60_data);
skier_right_90_rom skier6(skier_right_90_add,clk,skier_right_90_data);
skier_crashed_rom skiercrashed(skier_crashed_add,clk,skier_crashed_data);
skier_jumping_rom skierjump(skier_jumping_add,clk,skier_jumping_data);

// skier position registers - the skier is static in this position on-screen
// while the landscape moves past him
reg [9:0] skier_x = SKIER_GROUNDED_X;
reg [9:0] skier_y = SKIER_GROUNDED_Y;
reg jump_up = 0; // 1 if the the button is pressed and the skier is jumping
reg jump_down = 0; // 1 if the skier is landing from a jump
wire jump_mode;
assign jump_mode = jump_up | jump_down;

// controls display of the skier - compares current hcount and vcount to the position of
the skier
always @(posedge clk) begin
  // Reset all values to defaults
  if (reset) begin
    crash_in_range <= 0;
    crash_in_range_delay_one <= 0;
    crash_in_range_delay_two <= 0;
    jumper_in_range <= 0;
    jumper_in_range_delay_one <= 0;
    jumper_in_range_delay_two <= 0;
    skier_in_range <= 0;
    skier_in_range_delay_one <= 0;
    skier_in_range_delay_two <= 0;
    skier_x <= SKIER_GROUNDED_X;
    skier_y <= SKIER_GROUNDED_Y;
  end
end
// Check if the skier is currently jumping (not used in final presentation)
else begin
    /*if (vsync) begin
        if ((skier_y + SKIER_HEADON_HEIGHT <= ramp_one_y) && (skier_y >=
ramp_one_y - 50) && (skier_x + SKIER_HEADON_WIDTH >= ramp_one_x) && (skier_x <=
ramp_one_x + RAMP_WIDTH)) jump_up <= 1;
        else if ((skier_y + SKIER_HEADON_HEIGHT <= ramp_two_y) && (skier_y
        >= ramp_two_y - 50) && (skier_x + SKIER_HEADON_WIDTH >= ramp_two_x) && (skier_x <=
ramp_two_x + RAMP_WIDTH)) jump_up <= 1;
        // Skier is currently going up from jump
        if (jump_up) begin
            if (skier_y > SKIER_GROUNDED_Y - 100) skier_y <= skier_y -
4;
            else begin
                jump_up <= 0;
                jump_down <= 1;
            end
        end
        // Skier is currently going down from jump
        if (jump_down) begin
            if (skier_y < SKIER_GROUNDED_Y) skier_y <= skier_y + 2;
            else jump_down <= 0;
        end
    end*/
    crash_in_range_delay_one <= crash_in_range;
    jumper_in_range_delay_one <= jumper_in_range;
    crash_in_range_delay_two <= crash_in_range_delay_one;
    jumper_in_range_delay_two <= jumper_in_range_delay_one;
    skier_in_range_delay_one <= skier_in_range;
    skier_in_range_delay_two <= skier_in_range_delay_one;
    if (jumper_in_range) multiply_stage <= (vcount-
    skier_y)*SKIER_JUMPING_WIDTH;
        if (jumper_in_range_delay_one) skier_jumping_add <= multiply_stage +
        hcount-skier_x;
    end skier_pixel[24] <= 0;
        if (crash_in_range) multiply_stage <= (vcount-skier_y)*SKIER_CRASHED_WIDTH;
        if (crash_in_range_delay_one) skier_crashed_add <= multiply_stage +
        hcount-skier_x;
    end skier_pixel[24] <= 0;
end
    crash_in_range <= (crash_mode && (vcount >= skier_y && vcount < skier_y +
    SKIER_CRASHED_HEIGHT) &&
        (sprite_hcount >= skier_x && sprite_hcount < skier_x +
    SKIER_CRASHED_WIDTH));
    if (crash_mode == 0) begin
        // if not in crash mode
        jumper_in_range <= ((jump_up || jump_down) &&
            (vcount >= skier_y && vcount < skier_y + SKIER_JUMPING_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x +
    SKIER_JUMPING_WIDTH));
        */
Grebing, Stafl 51
skier_orientation_delay_one <= skier_orientation;
skier_orientation_delay_two <= skier_orientation_delay_one;
skier_orientation_delay_three <= skier_orientation_delay_two;
vcount_delay_one <= vcount;
hcount_delay_one <= hcount;
vcount_delay_two <= vcount_delay_one;
hcount_delay_two <= hcount_delay_one;

```
case (skier_orientation)
  0 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_90_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_90_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  1 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_60_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_60_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  2 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_30_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_30_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  3 : if ((vcount >= skier_y && vcount < skier_y + SKIER_HEADON_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_HEADON_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  4 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_30_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_30_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  5 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_60_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_60_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  6 : if ((vcount >= skier_y && vcount < skier_y + SKIER_POS_90_HEIGHT) &&
            (sprite_hcount >= skier_x && sprite_hcount < skier_x + SKIER_POS_90_WIDTH))
          skier_in_range <= 1;
          else skier_in_range <= 0;
  default : skier_in_range <= 0;
endcase
```

if (skier_in_range) begin
  case (skier_orientation_delay_one)
    0 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_90_WIDTH;
    1 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_60_WIDTH;
    2 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_30_WIDTH;
    3 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_HEADON_WIDTH;
    4 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_30_WIDTH;
    5 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_60_WIDTH;
    6 : multiply_stage <= (vcount_delay_one - skier_y)*SKIER_POS_90_WIDTH;
  endcase
end

if (skier_in_range_delay_one) begin
case (skier_orientation_delay_two)
  0 : skier_left_90_add <= multiply_stage + hcount_delay_two-skier_x;
  1 : skier_left_60_add <= multiply_stage + hcount_delay_two-skier_x;
  2 : skier_left_30_add <= multiply_stage + hcount_delay_two-skier_x;
  3 : skier_headon_add <= multiply_stage + hcount_delay_two-skier_x;
  4 : skier_right_30_add <= multiply_stage + hcount_delay_two-skier_x;
  5 : skier_right_60_add <= multiply_stage + hcount_delay_two-skier_x;
  6 : skier_right_90_add <= multiply_stage + hcount_delay_two-skier_x;
endcase
end

if (skier_in_range_delay_two) begin
  case (skier_orientation_delay_three)
    0 : begin
      skier_pixel[23:0] <= skier_left_90_data;
      skier_pixel[24] <= 1;
    end
    1 : begin
      skier_pixel[23:0] <= skier_left_60_data;
      skier_pixel[24] <= 1;
    end
    2 : begin
      skier_pixel[23:0] <= skier_left_30_data;
      skier_pixel[24] <= 1;
    end
    3 : begin
      skier_pixel[23:0] <= skier_headon_data;
      skier_pixel[24] <= 1;
    end
    4 : begin
      skier_pixel[23:0] <= skier_right_30_data;
      skier_pixel[24] <= 1;
    end
    5 : begin
      skier_pixel[23:0] <= skier_right_60_data;
      skier_pixel[24] <= 1;
    end
    6 : begin
      skier_pixel[23:0] <= skier_right_90_data;
      skier_pixel[24] <= 1;
    end
    default : skier_pixel[23:0] <= 24'hFFFFFF;
  endcase
end
else begin
  skier_pixel[24] <= 0;
  skier_pixel[23:0] <= 24'hFFFFFF;
end

end
if (!vsync) begin
  // reset skier addresses on each screen refresh
  skier_headon_add <= 0;
  skier_left_90_add <= 0;
  skier_left_60_add <= 0;
  skier_left_30_add <= 0;
  skier_right_90_add <= 0;
  skier_right_60_add <= 0;
  skier_right_30_add <= 0;
end

Grebing, Stafl
ZBT 6.111 Driver Module:

```
module zbt_6111(clk, cen, we, addr, write_data, read_data, ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);
```

```
// ZBT 6.111 Driver Module
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user. The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//
// Note: Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Note: Read requests are processed immediately, but the read data is not available
// until two cycles after the initial request.
//
// Inputs:
//
// clk - system clock
// cen - clock enable for gating ZBT cycles
// we - write enable (active HIGH)
// addr[18:0] - memory address
// write_data[35:0] - data to write
//
// Outputs:
//
// read_data[35:0] - data read from memory
// ram_clk - physical line to ram clock
// ram_we_b - physical line to ram we_b
// ram_address[18:0] - physical line to ram address
// ram_cen_b - physical line to ram clock enable
//
// Inouts:
//
// ram_data[35:0] - physical line to ram data
```

```
Grebing, Stafl
```
// wire to ZBT RAM signals
assign ram_we_b = ~we; // RAM is not happy with our data hold
assign ram_clk = ~clk; // times if its clk edges equal FPGA's
// so we clock it on the falling edges
// and thus let data stabilize longer
assign ram_address = addr;
assign ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
assign read_data = ram_data;
endmodule

Step to Pulse:
getPostinstance:------------------------// Step to Pulse Module
// Simply convert any step signal (such as a button press) to a one
// clock cycle pulse
//
// Inputs:
//   reset - any reset signal
//   clk - any clock signal
//   dirtystep - the step function that will be used to create a pulse
//
// Outputs:
//   pulse - the one clock cycle pulse, high when step goes 0->1
//--
module steptopulse(reset, clk, dirtystep, pulse);
input reset, clk, dirtystep;
output pulse;
// Store last value of step function for comparison
reg oldvalue = 0;
// Debounce signal first
wire step;
debounce db(reset, clk, dirtystep, step);
// True only when step just went from 0->1
assign pulse = (oldvalue == 0 && step == 1);
// Keep track of recent step value
always @(posedge clk)
  oldvalue <= step;
endmodule

Color Converter Module:
getPostinstance:------------------------// YCrCb -> RGB Color Converter Module
// Takes Y, Cr, Cb color values and converts them into R, G and B values to
// show a pixel with the same color, but in RGB space.
//
// Note: This module has a three stage pipeline, so the RGB output values
// are delayed by three from the input values. Make sure to compensate by
// delaying all other video signals by 3 clock cycles to sync it up.
//
// Inputs:
//   clk - any clock signal, most likely 65 MHz for video
//   rst - reset signal
//   Y[9:0] - luminance value of Pixel
//   Cr[9:0] - red chrominance value of pixel
//   Cb[9:0] - blue chrominance value of pixel
//
// Outputs:
//   R[7:0] - red color value of pixel
//   G[7:0] - green color value of pixel

Grebing, Stafl
module color_converter ( R, G, B, clk, rst, Y, Cr, Cb );

output [7:0]  R, G, B;
input clk,rst;
input [9:0] Y, Cr, Cb;
wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
reg [9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
const1 = 10'b 0100101010; //1.164 = 01.00101010
const2 = 10'b 0110011000; //1.596 = 01.10011000
const3 = 10'b 0011010000; //0.813 = 00.11010000
const4 = 10'b 0001100100; //0.392 = 00.01100100
const5 = 10'b 1000000100; //2.017 = 10.00000100
end
always @ (posedge clk or posedge rst)
if (rst)
begin
Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
end
else
begin
Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
end
always @ (posedge clk or posedge rst)
if (rst)
begin
A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
end
else
begin
X_int <= (const1 * (Y_reg - 'd64)) ;
A_int <= (const2 * (Cr_reg - 'd512));
B1_int <= (const3 * (Cr_reg - 'd512));
B2_int <= (const4 * (Cb_reg - 'd512));
C_int <= (const5 * (Cb_reg - 'd512));
end
always @ (posedge clk or posedge rst)
if (rst)
begin
R_int <= 0; G_int <= 0; B_int <= 0;
end
else
begin
R_int <= X_int + A_int;
G_int <= X_int - B1_int - B2_int;
B_int <= X_int + C_int;
end

// limit output to 0 = 4095, <0 equals 0 and >4095 equals 4095
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;
endmodule

NTSC to ZBT Module:

Grebing, Stafl
// Prepare data and address value to fill ZBT memory with NTSC data
//
// Note: The ZBT memory is 36 bits wide, so we store two 18-bit pixel values
// in the memory.
//
// Inputs:
// clk - system clock
// vclk - video clock from camera
// fvh[2:0] - field, vertical and horizontal signals
// dv - data valid signal
// data_in[17:0] - input data
// sw - switch to enable debugging mode
//
// Outputs:
// ntsc_addr[18:0] - address to store pixel values in
// ntsc_data[35:0] - two pixel values of data to store
// ntsc_we - write enable for the NTSC data
//
// Parameters:
// COL_START - where on the screen's x position the storage should start
// ROW_START - where on the screen's y position the storage should start
//
module ntsc_to_zbt(clk, vclk, fvh, dv, data_in, ntsc_addr, ntsc_data, ntsc_we, sw);

input clk;    // system clock
input vclk;   // video clock from camera
input [2:0] fvh;    // field, vertical and horizontal signals
input dv;     // data valid signal
input [17:0] data_in;  // input data
output [18:0] ntsc_addr; // address to store pixel values in
output [35:0] ntsc_data; // two pixel values of data to store
output ntsc_we; // write enable for NTSC data
input sw;     // switch to enable debugging mode

parameter COL_CENTER_START = 10'b1110001000;
parameter ROW_CENTER_START = 10'd62;

// here put the luminance data from the ntsc decoder into the ram
// this is for 1024 x 768 XGA display

reg [9:0] col = 0;
reg [9:0] row = 0;
reg [17:0] vdata = 0;
reg vwe;
reg old_dv;
reg old_frame; // frames are even / odd interlaced
reg even_odd;  // decode interlaced frame to this wire
wire frame = fvh[2];
wire frame_edge = frame & ~old_frame;

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

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

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

// edge detection on write enable signal
reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];

// Shift each set of two-byte data forward.
reg [35:0] mydata;
always @(posedge clk)
    mydata <= { mydata[17:0], data[1] };

// compute address to store data in
wire [18:0] myaddr = {y[1][8:0], eo[1], x[1][9:1]};

// alternate (256x192) image data and address
wire [35:0] mydata2 = {data[1], data[0]};
wire [18:0] myaddr2 = {y[1][8:0], 1'b0, x[1][9:1]};

// update the output address and data only when the two pixels are ready
reg [18:0] ntsc_addr;
reg [35:0] ntsc_data;
wire ntsc_we = sw ? we_edge : (we_edge & (x[1][0] == 1'b0));
always @(posedge clk)
if ( ntsc_we )
begin
    ntsc_addr <= sw ? myaddr2 : myaddr; // normal and expanded modes
    ntsc_data <= sw ? mydata2 : mydata;
end
endmodule // ntsc_to_zbt

**Game Video Output Module:**
------------------------------------------------------------------------------------------------------------------
// Game_video_output module
//
// Takes in pixel values from all of the sprite generating modules and combines them for the final game output.
//
// Inputs:
//     clk - 65 MHz clock signal, synchronous with pixels
//     reset - user or global reset
//     hcount[10:0] - horizontal (x) coordinate of current pixel
//     vcount[9:0] - vertical (y) coordinate of current pixel
//     hsync - XVGA horizontal sync signal (active low)
//     vsync - XVGA vertical sync signal (active low)
//     blank - XVGA blanking (1 means output black pixel)
//     skier_pixel[24:0] - (for each of the pixel signals) - bit 24 indicates if the pixel is active, bits [23:0] indicate RGB values
//     landscape_pixel[24:0]
//     time_display_pixel[24:0]
//     slalom_left_pixel[24:0]
//     slalom_right_pixel[24:0]
// ramp_pixel[24:0]
// distance_display_pixel[24:0]

// Outputs:
// phsync - horizontal sync for game video
// pvsync - vertical sync for game video
// pblank - blanking signal for game video
// game_pixel[23:0] - 24-bit RGB value for each pixel on screen,
// passed to the XVGA display

module game_video_output(clk, hsync, vsync, blank, skier_pixel, landscape_pixel,
                         time_display_pixel, slalom_left_pixel, slalom_right_pixel, left_boundary_gate_pixel,
                         right_boundary_gate_pixel, ramp_pixel, distance_display_pixel, phsync, pvsync,
                         pblank, game_pixel_out);

input clk;  // video clock
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)

// pixel inputs from various modules
input [24:0] skier_pixel;
input [24:0] landscape_pixel;
input [24:0] time_display_pixel;
input [24:0] slalom_left_pixel;
input [24:0] slalom_right_pixel;
input [24:0] left_boundary_gate_pixel;
input [24:0] right_boundary_gate_pixel;
input [24:0] ramp_pixel;
input [24:0] distance_display_pixel;

output phsync;  // game's horizontal sync
output pvsync;  // game's vertical sync
output pblank;  // game's blanking

output [23:0] game_pixel_out;  // pixel value output

// delayed versions of the hsync, vsync, and blank signals
wire phsync;
wire pvsync;
wire pblank;

// delays the hsync, vsync, and blank signals by six clock cycles
delayN dn4(clk, hsync, phsync);
delayN dn5(clk, vsync, pvsync);
delayN dn6(clk, blank, pblank);

defparam dn4.NDELAY = 5;
defparam dn5.NDELAY = 5;
defparam dn6.NDELAY = 5;

// register to hold the final RGB value of the game's pixel for the current hcount and vcount
reg [23:0] game_pixel_out;

// systematically checks to see if each module's pixel is active - if active,
game_pixel_out takes
// the value of the object's pixel
always @(posedge clk) begin
  if (time_display_pixel[24] == 1)
    game_pixel_out <= time_display_pixel[23:0];
  else if (ramp_pixel[24] == 1)
    game_pixel_out <= ramp_pixel[23:0];
  else if (distance_display_pixel[24] == 1)
    game_pixel_out <= distance_display_pixel[23:0];
  else if (slalom_right_pixel[24] == 1)
    game_pixel_out <= slalom_right_pixel[23:0];
  else if (slalom_left_pixel[24] == 1)
    game_pixel_out <= slalom_left_pixel[23:0];
else if (landscape_pixel[24] == 1)
    game_pixel_out <=
    landscape_pixel[23:0];
else if (skier_pixel[24] == 1)
    game_pixel_out <=
    skier_pixel[23:0];
else if
    left_boundary_gate_pixel[24] == 1)
    game_pixel_out <=
    left_boundary_gate_pixel[23:0];
else if
    right_boundary_gate_pixel[24] == 1)
    game_pixel_out <=
    right_boundary_gate_pixel[23:0];
else
    game_pixel_out <= 24'hFFFFFF;
end
endmodule  // game_video_output

Collision Detection Module:

Collision Detection Module

------
// Collision_detection module
//
// Compares the value of landscape_pixel and skier_pixel to see if the two
// occupy the same pixel simultaneously. This signifies a crash.
//
// Inputs:
//    clk - 65 MHz clock signal, synchronous with pixels
//    reset - global reset signal
//    vsync - XvGA vertical sync signal (active low)
//    one_hz_signal - pulses high every 65,000,000 clock cycles (each second)
//    skier_pixel_active - bit 24 of skier_pixel (1 if pixel is active)
//    landscape_pixel_active - bit 24 of landscape_pixel (1 if pixel is active)
//
// Outputs:
//    crash_mode - 1 if the skier has just crashed into a landscape object
//
//------
module collision_detection(clk,reset,vsync,one_hz_signal,skier_pixel_active,
    landscape_pixel_active,detect_crashes,jump_mode,crash_mode);

    input clk;
    input reset;
    input vsync;
    input one_hz_signal;
    input skier_pixel_active;  // takes bit 24 value from skier_pixel (1 indicates it is active)
    input landscape_pixel_active;  // takes bit 24 value from landscape_pixel (1 indicates it is active)
    input detect_crashes;
    input jump_mode;

    output crash_mode;

    reg crash_mode = 0;

    // times one second after the skier crashes before he can get up again
    reg [26:0] crash_counter = 0;

    always @ (posedge clk) begin
        // Handle reset signals
        if (reset) begin
            crash_mode <= 0;
            crash_counter <= 0;
        end
        // Recover from previous crash after one second
        else begin
            if (crash_mode) begin
                if (crash_counter == 5000000) begin
                    // Crash detected, reset
                    crash_mode <= 1;
                    crash_counter <= 0;
                end
            end
        end
    end
endmodule
crash_mode <= 0;
crash_counter <= 0;
end
else
   crash_counter <= crash_counter + 1;
end
// Detect a new crash
else if (detect_crashes && !jump_mode &&
    skier_pixel_active && landscape_pixel_active) begin
   crash_mode <= 1;
crash_counter <= 0;
end
end
endmodule

**Timer Module:**

////////////////////////////////////////////
// Timer module
// Divides the 65 MHz clock into 100, 10, 1, 1/10, and 1/100 Hz signals and
// outputs them to time_display and other modules
// Inputs:
// clk - 65 MHz clock signal, synchronous with pixels
// reset - user or global reset
// timer_run - high when timer should be running
// Outputs:
// sixtieth_hz_signal - pulses high once per minute
// tenth_hz_signal - pulses high once every ten seconds
// one_hz_signal - pulses high once per second
// ten_hz_signal - pulses high once per 1/10 second
// hundred_hz_signal - pulses high once per 1/100 second
////////////////////////////////////////////

module timer(clk,timer_run,reset,missed_left_gate,missed_right_gate,
sixtieth_hz_signal,tenth_hz_signal,one_hz_signal,ten_hz_signal,hundred_hz_signal);
input clk;
input timer_run;
input reset;
input missed_left_gate;
input missed_right_gate;
output sixtieth_hz_signal;
output tenth_hz_signal;
output one_hz_signal;
output ten_hz_signal;
output hundred_hz_signal;

always @(posedge clk) begin
    sixtieth_hz_signal <= 0;
    tenth_hz_signal <= 0;
    one_hz_signal <= 0;
    ten_hz_signal <= 0;
    hundred_hz_signal <= 0;
end

Grebing, Stafl
// Handle resets
if (reset) begin
    hundred_hz_counter <= 0;
ten_hz_counter <= 0;
one_hz_counter <= 0;
tenth_hz_counter <= 0;
sixtieth_hz_counter <= 0;
end

// Run when the timer is enabled
else if (timer_run) begin
    // Increment 1/100ths digit
    if (hundred_hz_counter == 650000) begin // splits the 65 MHz clock into
        100 Hz
        hundred_hz_signal <= 1;
        hundred_hz_counter <= 0;
        if (ten_hz_counter < 10 )
            ten_hz_counter <= ten_hz_counter + 1;
    end
    else begin
        hundred_hz_counter <= hundred_hz_counter + 1;
        hundred_hz_signal <= 0;
    end
    // Increment 1/10ths digit
    if (ten_hz_counter == 10) begin
        ten_hz_signal <= 1;
        ten_hz_counter <= 0;
        if (one_hz_counter < 10)
            one_hz_counter <= one_hz_counter + 1;
    end
    else ten_hz_signal <= 0;
    // Increment 1 second digit
    if (one_hz_counter == 10) begin
        one_hz_signal <= 1;
        one_hz_counter <= 0;
        if (tenth_hz_counter < 10 )
            tenth_hz_counter <= tenth_hz_counter + 1;
    end
    else one_hz_signal <= 0;
    // Increment 10 second digit
    if (tenth_hz_counter >= 10) begin // due to changes with the five second
        missed gate penalty
        tenth_hz_signal <= 1;
        tenth_hz_counter <= tenth_hz_counter - 10;
        if (sixtieth_hz_counter < 6 )
            sixtieth_hz_counter <= sixtieth_hz_counter + 1;
    end
    else tenth_hz_signal <= 0;
    // Increment 1 minute digit
    if (sixtieth_hz_counter == 6) begin
        sixtieth_hz_signal <= 1;
        sixtieth_hz_counter <= 0;
    end
    else sixtieth_hz_signal <= 0;
    // Assign penalties for missed gates
    if (!missed_left_gate && !missed_right_gate)
        penalty_assigned <= 0;
    if ((missed_left_gate || missed_right_gate) &&
        !penalty_assigned && (penalty_counter < 10)) begin
        penalty_counter <= penalty_counter + 1;
        one_hz_signal <= ~one_hz_signal;
    end
    else if (penalty_counter == 10) begin
        tenth_hz_counter <= tenth_hz_counter + 5;
        penalty_assigned <= 1;
        penalty_counter <= 0;
    end
end
Time Display Module:

// Time_display module
//
// Takes in signals from the timer module to control the output display of the time.
//
// Inputs:
//   clk - 65 MHz clock signal, synchronous with pixels
//   reset - user or global reset
//   hcount[10:0] - horizontal (x) coordinate of current pixel
//   vcount[9:0] - vertical (y) coordinate of current pixel
//   sixtieth_hz_signal - signal from timer module is high for one clock cycle each
//   minute
//   seconds
//   tenth_hz_signal - signal from timer module is high for one clock cycle every ten
//   seconds
//   one_hz_signal - signal from timer module is high for one clock cycle each second
//   ten_hz_signal - signal from timer module is high for one clock cycle ten times per
//   second
//   hundred_hz_signal - signal from timer module is high for one clock cycle 100 times
//   per second
//
// Outputs:
//   time_display_pixel[24:0] - Bit 24 signifies that the screen should display a time
//   number for
//   hcount/vcount
//   - Bits [23:0] are
//   passed as the RGB values of the XVGA display
//
module time_display(clk,reset,vcount,hcount,sixtieth_hz_signal,tenth_hz_signal,
   one_hz_signal,ten_hz_signal,hundred_hz_signal,time_display_pixel);

   input clk;
   input reset;
   input [10:0] hcount;
   input [9:0] vcount;
   input sixtieth_hz_signal;
   input tenth_hz_signal;
   input one_hz_signal;
   input ten_hz_signal;
   input hundred_hz_signal;

   output [24:0] time_display_pixel;

   // size parameters for the each digit and the text display rom
   parameter DIGIT_WIDTH = 8;
   parameter DIGIT_HEIGHT = 11;
   parameter TEXT_DISPLAY_WIDTH = 45;
   parameter TEXT_DISPLAY_HEIGHT = 29;

   // sprite count compensates for a delay in the logic
   wire [10:0] sprite_hcount;
   assign sprite_hcount = hcount - 1;

   // register holds the current value of the module's output pixel - bit 24 is 1 if the
   // pixel is active
   // for time_display
   reg [24:0] time_display_pixel;

   // x and y coordinates for time display values
   reg [9:0] display_text_x = 875;
   reg [9:0] display_text_y = 8;

   reg [9:0] display_onemin_x = 924;
   reg [9:0] display_onemin_y = 10;

Grebing, Stafl 63
```verilog
reg [9:0] display_colon_one_x = 933;
reg [9:0] display_colon_one_y = 10;
reg [9:0] display_tensec_x = 938;
reg [9:0] display_tensec_y = 10;
reg [9:0] display_onesec_x = 947;
reg [9:0] display_onesec_y = 10;
reg [9:0] display_colon_two_x = 956;
reg [9:0] display_colon_two_y = 10;
reg [9:0] display_tenth_x = 962;
reg [9:0] display_tenth_y = 10;
reg [9:0] display_hundredth_x = 971;
reg [9:0] display_hundredth_y = 10;

// time display rom address registers
reg [10:0] time_display_text_add = 0;
reg [9:0] hundredth_digit_add = 0;
reg [9:0] tenth_digit_add = 0;
reg [9:0] onesec_digit_add = 0;
reg [9:0] tensec_digit_add = 0;
reg [9:0] onemin_digit_add = 0;
reg [5:0] colon_one_add = 0;
reg [5:0] colon_two_add = 0;

// digit counters to tell rom address which digit image to use
reg [3:0] hundredth_digit_counter = 0; // keeps track of the digit displayed for 1/100 sec
reg [3:0] tenth_digit_counter = 0; // keeps track of the digit displayed for 1/10 sec
reg [3:0] onesec_digit_counter = 0; // keeps track of the digit displayed for 1 sec
reg [2:0] tensec_digit_counter = 0; // keeps track of the digit displayed for 10 sec
reg [3:0] onemin_digit_counter = 0;

// data wires for reading pixel values from roms (1-bit because images are monochrome)
wire hundredth_digit_data;
wire tenth_digit_data;
wire onesec_digit_data;
wire tensec_digit_data;
wire onemin_digit_data;
wire time_display_text_data;

// instantiates the digit and text roms for each place value for time and distance display
digits_rom hundredth_digit_rom(hundredth_digit_add,clk,hundredth_digit_data);
digits_rom tenth_digit_rom(tenth_digit_add,clk,tenth_digit_data);
digits_rom onesec_digit_rom(onesec_digit_add,clk,onesec_digit_data);
digits_rom tensec_digit_rom(tensec_digit_add,clk,tensec_digit_data);
digits_rom onemin_digit_rom(onemin_digit_add,clk,onemin_digit_data);
digits_rom gates_missed_ones_rom(gates_missed_ones_add,clk,gates_missed_ones_data);
digits_rom gates_missed_tens_rom(gates_missed_tens_add,clk,gates_missed_tens_data);
time_display_text_rom text_rom(time_display_text_add,clk,time_display_text_data);

// variables to control case statements in the pipeline -
// the value corresponds to the actual value to be displayed
reg [2:0] digit_to_display = 0;
reg [2:0] digit_to_display_delay_one = 0;
reg [2:0] digit_to_display_delay_two = 0;

// delay registers to hold hcount and vcount values for use in the pipeline
reg [9:0] vcount_delay_one = 0;
reg [10:0] hcount_delay_one = 0;
reg [10:0] hcount_delay_two = 0;

// register holds the value of a multiplication step in stage two of the output pipeline
reg [11:0] multiply_stage = 0;

always @(posedge clk) begin
    // sets delay registers to previous values, moving them through the pipeline
    digit_to_display_delay_one <= digit_to_display;
    digit_to_display_delay_two <= digit_to_display_delay_one;
end
```
digit_to_display_delay_two <= digit_to_display_delay_one;
vcount_delay_one <= vcount;
hcount_delay_one <= hcount;
hcount_delay_two <= hcount_delay_one;

// each digit’s value goes back to zero in the event of a user reset
if (reset) begin
  hundredth_digit_counter <= 0;
tenth_digit_counter <= 0;
one_sec_digit_counter <= 0;
tensec_digit_counter <= 0;
onemin_digit_counter <= 0;
end
else begin

  // the time_display_pixel is inactive by default
  time_display_pixel[24] <= 0;
time_display_pixel[23:0] <= 24'hFFFFFF;

  // upon each signal from the timer, the digit_counter variables identify which
  address of the digits rom
  // should be used to display the proper digit
  if (hundred_hz_signal) begin
    if (hundredth_digit_counter == 9) hundredth_digit_counter <= 0;
    else hundredth_digit_counter <= hundredth_digit_counter + 1;
  end
  if (ten_hz_signal) begin
    if (tenth_digit_counter == 9) tenth_digit_counter <= 0;
    else tenth_digit_counter <= tenth_digit_counter + 1;
  end
  if (one_hz_signal) begin
    if (onesec_digit_counter == 9) onesec_digit_counter <= 0;
    else onesec_digit_counter <= onesec_digit_counter + 1;
  end
  if (tenth_hz_signal) begin
    if (tensec_digit_counter == 5) tensec_digit_counter <= 0;
    else tensec_digit_counter <= tensec_digit_counter + 1;
  end
  if (sixtieth_hz_signal) begin
    if (onemin_digit_counter == 9) onemin_digit_counter <= 0;
    else onemin_digit_counter <= onemin_digit_counter + 1;
  end

  // first stage of the pipeline determines if a particular digit should appear at
  the current values
  // of sprite_hcount and vcount
  if ((vcount >= display_text_y && vcount < display_text_y + TEXT_DISPLAY_HEIGHT) &&
      (sprite_hcount >= display_text_x && sprite_hcount < display_text_x +
       TEXT_DISPLAY_WIDTH)) digit_to_display <= 1;
  else if ((vcount >= display_hundredth_y && vcount < display_hundredth_y +
            DIGIT_HEIGHT) &&
           (sprite_hcount >= display_hundredth_x && sprite_hcount <
            display_hundredth_x + DIGIT_WIDTH)) digit_to_display <= 2;
  else if ((vcount >= display_tenth_y && vcount < display_tenth_y + DIGIT_HEIGHT) &&
            (sprite_hcount >= display_tenth_x && sprite_hcount <
             display_tenth_x + DIGIT_WIDTH)) digit_to_display <= 3;
  else if ((vcount >= display_onesec_y && vcount < display_onesec_y + DIGIT_HEIGHT) &&
            (sprite_hcount >= display_onesec_x && sprite_hcount <
             display_onesec_x + DIGIT_WIDTH)) digit_to_display <= 4;
  else if ((vcount >= display_tensec_y && vcount < display_tensec_y + DIGIT_HEIGHT) &&
            (sprite_hcount >= display_tensec_x && sprite_hcount <
             display_tensec_x + DIGIT_WIDTH)) digit_to_display <= 5;
  else if ((vcount >= display_onemin_y && vcount < display_onemin_y + DIGIT_HEIGHT) &&
            (sprite_hcount >= display_onemin_x && sprite_hcount <
             display_onemin_x + DIGIT_WIDTH)) digit_to_display <= 6;
{sprite_hcount >= display_onemin_x && sprite_hcount < display_onemin_x + DIGIT_WIDTH}) digit_to_display <= 6;

else digit_to_display <= 0;

// second stage of the pipeline performs a multiplication process for use in
determining the rom address
// of the object
// case (digit_to_display)
// 1 : multiply_stage <= (vcount_delay_one-display_text_y)*TEXT_DISPLAY_WIDTH;
// 2 : multiply_stage <= (vcount_delay_one-display_hundredth_y +
// hundredth_digit_counter*DIGIT_HEIGHT);
// 3 : multiply_stage <= (vcount_delay_one-display_tenth_y +
// tenth_digit_counter*DIGIT_HEIGHT);
// 4 : multiply_stage <= (vcount_delay_one-display_onesec_y +
// onesec_digit_counter*DIGIT_HEIGHT);
// 5 : multiply_stage <= (vcount_delay_one-display_tensec_y +
// tensec_digit_counter*DIGIT_HEIGHT);
// 6 : multiply_stage <= (vcount_delay_one-display_onemin_y +
// onemin_digit_counter*DIGIT_HEIGHT);
// default : multiply_stage <= 0;
// endcase

// third stage of the pipeline assigns rom addresses if a digit should be
displayed for the current pixel
// case (digit_to_display_delay_one)
// 1 : time_display_text_add <= multiply_stage + hcount_delay_two-
// display_text_x;
// 2 : hundredth_digit_add <= multiply_stage*DIGIT_WIDTH + (hcount_delay_two-
// display_hundredth_x);
// 3 : tenth_digit_add <= multiply_stage*DIGIT_WIDTH + (hcount_delay_two-
// display_tenth_x);
// 4 : onesec_digit_add <= multiply_stage*DIGIT_WIDTH + (hcount_delay_two-
// display_onesec_x);
// 5 : tensec_digit_add <= multiply_stage*DIGIT_WIDTH + (hcount_delay_two-
// display_tensec_x);
// 6 : onemin_digit_add <= multiply_stage*DIGIT_WIDTH + (hcount_delay_two-
// display_onemin_x);
// endcase

// final pipeline stage makes the time_display_pixel active and transmits data
// from the rom into
// time_display_pixel[23:0]
// case (digit_to_display_delay_two)
// 1 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{time_display_text_data}};
end
// 2 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{hundredth_digit_data}};
end
// 3 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{tenth_digit_data}};
end
// 4 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{onesec_digit_data}};
end
// 5 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{tensec_digit_data}};
end
// 6 : begin
time_display_pixel[24] <= 1;
time_display_pixel[23:0] <= {24{onemin_digit_data}};
end
// default : begin
/* time_display_pixel[24] <= 0;
time_display_pixel[23:0] <= 24'hFFFFFF; */
Slalom Gate Detect Module:

This module compares the location of slalom gates to that of the skier and determines if each gate has been encountered and missed.

Inputs:
- clk - 65 MHz clock signal, synchronous with pixels
- reset - user or global reset
- vcount[9:0] - vertical (y) coordinate of current pixel
- skier_x[9:0]
- skier_y[9:0]
- left_slalom_gate_x[9:0]
- left_slalom_gate_y[9:0]
- right_slalom_gate_x[9:0]
- right_slalom_gate_y[9:0]

Outputs:
- left_gate_encountered - 1 if the skier has passed the left slalom gate on screen when he passes it
- right_gate_encountered - 1 if the skier has passed the right slalom gate on screen when he passes it
- missed_left_gate - 1 if the skier is not to the left of the left gate when he passes it
- missed_right_gate - 1 if the skier is not to the right of the right gate when he passes it

Parameters:
- various widths and heights of the skier and slalom gate sprites

module slalom_gate_detect(clk, vsync, vcount, hcount, skier_x, skier_y, left_slalom_gate_x, left_slalom_gate_y, right_slalom_gate_x, right_slalom_gate_y, left_gate_encountered, right_gate_encountered, missed_left_gate, missed_right_gate, led_val);

input clk;
input vsync;
input [9:0] vcount;
input [10:0] hcount;

// used to compare against location of slalom gates and determine whether or not a penalty should be assessed
input [9:0] skier_x;
input [9:0] skier_y;
input [9:0] left_slalom_gate_x;
input [9:0] left_slalom_gate_y;
input [9:0] right_slalom_gate_x;
input [9:0] right_slalom_gate_y;

output left_gate_encountered; // 1 if skier is at same vertical position as a slalom gate
output right_gate_encountered;
output missed_left_gate; // 1 if gate is missed, 0 if gate
output missed_right_gate;
output [3:0] led_val; // temporary for debugging

reg left_gate_encountered = 0;
reg right_gate_encountered = 0;
reg missed_left_gate = 0;
reg missed_right_gate = 0;
parameter SKIER_WIDTH = 16;
parameter SKIER_HEIGHT = 31;

parameter SLALOM_GATE_WIDTH = 14;
parameter SLALOM_GATE_HEIGHT = 25;

always @(posedge vsync) begin
    // checks skier's position at the center of the slalom gate
    if ((left_slalom_gate_y + SLALOM_GATE_HEIGHT) <= skier_y &&
        (left_slalom_gate_y + SLALOM_GATE_HEIGHT) >= (skier_y-8)) begin
        left_gate_encountered <= 1;
        // skier must be completely to the left of the gate
        if (skier_x <= left_slalom_gate_x) missed_left_gate <= 1;
    end

    // checks skier's position at the center of the slalom gate
    if ((right_slalom_gate_y + SLALOM_GATE_HEIGHT) <= skier_y &&
        (right_slalom_gate_y + SLALOM_GATE_HEIGHT) >= (skier_y-8)) begin
        right_gate_encountered <= 1;
        // skier must be completely to the right of the gate
        if (skier_x <= right_slalom_gate_x) missed_right_gate <= 1;
    end

    if (left_slalom_gate_y >= 768) begin
        // when the gate is not
        left_gate_encountered <= 0;
        // resets values of
        missed_left_gate <= 0;
        // position of the gate is off screen
        end

    if (right_slalom_gate_y >= 768) begin
        right_gate_encountered <= 0;
        missed_right_gate <= 0;
        end

    end

    // temporary use of the leds for debugging
    assign led_val[3] = left_gate_encountered;
    assign led_val[2] = right_gate_encountered;
    assign led_val[1] = missed_left_gate;
    assign led_val[0] = missed_right_gate;
endmodule

Timer Toggle Module:

Timertoggle module

// Asserts and deasserts the timer_run signal based on the reset signal and the
// end of the course.
//
// Inputs:
//    clk - 65 MHz clock signal, synchronous with pixels
//    reset - user or global reset
//    hcount[10:0] - horizontal (x) coordinate of current pixel
//    vcount[9:0] - vertical (y) coordinate of current pixel
//    skier_x[9:0] - x coordinate of the skier's position
//    skier_y[9:0] - y coordinate of the skier's position
//    course_started - 1 if the skier has entered the course
//    distance_to_go[10:0] - value of the skier's remaining distance down the hill
//
// Outputs:
//    timer_run - 1 when the timer should be running (turned on and off by the start
//                and finish gates)
//
module timer_toggle(clk,reset,course_started,distance_to_go,timer_run);
input clk;
input reset;
// course_started;
input [10:0] distance_to_go;

output timer_run;

// signifies if the timer should be running
reg timer_run = 0;

always @(posedge clk) begin
  if (reset) timer_run <= 1; // timer restarts immediately after a reset
  if (course_started == 0) 
    timer_run <= 1; // timer starts as soon as the image appears on screen
  else if (distance_to_go == 0)
    timer_run <= 0; // timer stops running once the end of the course is reached
end
endmodule

Distance Tracker Module:

Distance_tracker module
/////////////////////////////////////////////////////////////////////////////////
// Distance_tracker module
// Tracks the distance to go and outputs signals to control the digits in the distance display.
// // Inputs:
// clk - 65 MHz clock signal, synchronous with pixels
// reset - user or global reset
// // Outputs:
// sixtieth_hz_signal - pulses high once per minute
// tenth_hz_signal - pulses high once every ten seconds
// one_hz_signal - pulses high once per second
// ten_hz_signal - pulses high once per 1/10 second
// hundred_hz_signal - pulses high once per 1/100 second
// //
module distance_tracker(clk,reset,vsync,distance_to_go,course_started, 
distance_thousands,distance_hundreds,distance_tens,distance_ones);

input clk;
input reset;
input vsync;
input course_started;
input [10:0] distance_to_go;

// outputs each of the digits so that they can be displayed in distance_display
output [1:0] distance_thousands;
output [3:0] distance_hundreds;
output [3:0] distance_tens;
output [3:0] distance_ones;

reg [1:0] distance_thousands = 2;
reg [3:0] distance_hundreds = 0;
reg [3:0] distance_tens = 0;
reg [3:0] distance_ones = 0;

reg [10:0] previous_distance_to_go = 2000;

always @(posedge vsync) begin
  if (reset) begin
    distance_thousands <= 2;
    distance_hundreds <= 0;
    distance_tens <= 0;
    distance_ones <= 0;
  end

  if (distance_to_go != previous_distance_to_go && course_started) begin
    // Grebing, Stafl
  end
end
endmodule
Final Compositor Module:

module final_compositor(reset, clk, hcount, vcount, hs_in, vs_in, blank_in, video_pixel, game_pixel, show_overlay, full_video_output, pixel_out, hs_out, vs_out, blank_out);

input reset, clk;
input hs_in, vs_in, blank_in;
input [10:0] hcount;
input [9:0] vcount;
input [23:0] video_pixel, game_pixel;
input show_overlay, full_video_output;

output [23:0] pixel_out;
output hs_out, vs_out, blank_out;

// Bounding box of video output display in full screen
parameter UPPERX = 11'd180;

parameter UPPERY = 10'd140;
parameter LOWERX = 11'd710;
parameter LOWERY = 10'd540;

reg hs_out, vs_out, blank_out;
reg [23:0] pixel_out;
wire [10:0] hcount;
wire [9:0] vcount;

// Signals for accessing BRAM
reg [14:0] bram_address;
reg write_enable;
reg [23:0] data_in;
wire [23:0] data_out;

// Is the current pixel in the overlay box?
wire overlay_box = (hcount > 11'd0 && hcount < 11'd140 && vcount >= 10'd0 && vcount < 10'd88);

// The rectangle describing the video shown full screen
wire video_box = (hcount < LOWERX && hcount > UPPERX && vcount < LOWERY && vcount > UPPERY);

// Adjusted values to address every 4th pixel to store in memory
reg [7:0] adjusted_h;
reg [6:0] adjusted_v;

always @ (posedge clk) begin
    // Set up addresses for next cycle
    adjusted_h <= (hcount - UPPERX) >> 2;
    adjusted_v <= (vcount - UPPERY) >> 2;
    // If current pixel is in overlay
    if( overlay_box ) begin
        bram_address <= {vcount[6:0], hcount[7:0]};
        write_enable <= 0;
    end
    // If current pixel is in video rectangle
    else if( video_box ) begin
        bram_address <= {adjusted_v, adjusted_h};
        write_enable <= 1;
        data_in <= video_pixel;
    end
    else
        write_enable <= 0;

    // Shift all signals over to line up for the 1 clk cycle delay
    hs_out <= hs_in;
    vs_out <= vs_in;
    blank_out <= blank_in;

    // Show full screen output from camera
    if ( full_video_output ) begin
        pixel_out <= video_pixel;
    end
    // Show video overlay in lower left corner on top of the game screen
    else if( show_overlay ) begin
        // Read from BRAM and smaller image
        if( overlay_box )
            pixel_out <= data_out;
        else
            pixel_out <= game_pixel;
    end
    else if( hcount > 1030 || vcount > 768 )
        pixel_out <= 24'b0;
    // Simply show the game output
    else
        pixel_out <= game_pixel;
end

// BRAM for storing small image of video output
videoram

Grebing, Stafl
Center of Mass Module:

module centerofmass(reset, clk, hcount, vcount, selected, haverage, vaverage);

input reset, clk;
input [10:0] hcount;
input [9:0] vcount;
input selected;
output [10:0] haverage;
output [9:0] vaverage;

parameter DIVIDER_DELAY = 6'd33;

// Count for the divider delay:
reg [5:0] delaycount = 0;

// Total number of pixels that have fit the criteria this frame
reg [19:0] pixelcount = 0;

// Weighted total for x positions and y positions of selected pixels
reg [31:0] xcount = 0;
reg [31:0] ycount = 0;

// Output from divider, only sampled once per frame
wire [31:0] haverage_temp;
wire [31:0] vaverage_temp;

// Resulting x and y center of mass from previous frame
reg [10:0] haverage = 0;
reg [9:0] vaverage = 0;

// Signal the beginning of the new frame
wire newframe = (hcount == 0 && vcount == 0);

// Add each pixel to count, and reset at the beginning of each frame
always @ (posedge clk)
begin
    // Handle systemwide reset
    if (reset) begin
        pixelcount <= 0;
xcount <= 0;
ycount <= 0;
haverage <= 0;
vaverage <= 0;
delaycount <= 0;
end
    // Reset pixel count at the beginning of each frame
    else if(newframe) begin

end

endmodule
// If the current pixel is selected, add its position to the totals
else begin
    pixelcount <= pixelcount + selected;
    if ( selected ) begin
        xcount <= xcount + hcount;
        ycount <= ycount + vcount;
    end
    // Increment delay counter, and extract average values at end of cycle
    if ( delaycount == DIVIDER_DELAY ) begin
        haverage <= haverage_temp[10:0];
        vaverage <= vaverage_temp[9:0];
        delaycount <= delaycount + 1; // Will remain at DELAY + 1 until
    end
    else if( delaycount < DIVIDER_DELAY )
        delaycount <= delaycount + 1;
end
end

// Divider for x position
wire [19:0] remainderx;
wire readyfordatax;
comdivider comx(.divisor(pixelcount),.dividend(xcount),.clk(clk),.quot(haverage_temp),
                   .remd(remainderx),.rfd(readyfordatax));

// Divider for y position
wire [19:0] remaindery;
wire readyfordatay;
comdivider comy(.divisor(pixelcount),.dividend(ycount),.clk(clk),.quot(vaverage_temp),
                   .remd(remaindery),.rfd(readyfordatay));

endmodule

Angle Detect Module:

// Angle Detect Module
// Take in x and y positions of pixels, and will average the locations of all the
// pixels to the left and right of the x center of mass provided, and will output
// the extrapolated block angle from that information.
//
// Inputs:
// reset - any reset signal
// clk - 65 MHz clock signal, synchronous with pixels
// hcount[10:0] - horizontal (x) coordinate of current pixel
// vcount[9:0] - vertical (y) coordinate of current pixel
// selected - one bit value determining whether pixel fits selection criteria
// haverage[10:0] - horizontal position of center of mass of last frame
// vaverage[9:0] - vertical position of center of mass of last frame
//
// Outputs:
// leftave_x[10:0] - x position of left COM
// leftave_y[9:0] - y position of left COM
// rightave_x[10:0] - x position of right COM
// rightave_y[9:0] - y position of right COM
// topave_x[10:0] - x position of top COM
// topave_y[9:0] - y position of top COM
// bottomave_x[10:0] - x position of bottom COM
// bottomave_y[9:0] - y position of bottom COM
// anglesignal[2:0] - which of the 7 possible angular positions the skier is in
//
// Parameters:
// DIVIDER_DELAY - Latency of the divider module used, typically M+2 cycles
//
module angledetect(reset,clk,hcount,vcount,selected,haverage,vaverage,
                   leftave_x,leftave_y,rightave_x,rightave_y,topave_x,
                   bottomave_x,bottomave_y,anglesignal);

// Divider for x position
wire [19:0] remainderx;
wire readyfordatax;
comdivider comx(.divisor(pixelcount),.dividend(xcount),.clk(clk),.quot(haverage_temp),
                   .remd(remainderx),.rfd(readyfordatax));

// Divider for y position
wire [19:0] remaindery;
wire readyfordatay;
comdivider comy(.divisor(pixelcount),.dividend(ycount),.clk(clk),.quot(vaverage_temp),
                   .remd(remaindery),.rfd(readyfordatay));

endmodule
input reset, clk;
input [10:0] hcount;
input [9:0] vcount;
input selected;
input [10:0] haverage;
input [9:0] vaverage;
output [10:0] leftave_x, rightave_x, topave_x, bottomave_x;
output [9:0] leftave_y, rightave_y, topave_y, bottomave_y;
output [2:0] anglesignal;

parameter DIVIDER_DELAY = 7'd34;

// Count for the divider delay:
reg [6:0] delaycount = 0;

// Total number of pixels that have fit the criteria this frame
reg [19:0] pixelcount_left, pixelcount_right, pixelcount_top, pixelcount_bottom = 0;
reg [19:0] pixelcount_right_old, pixelcount_top_old, pixelcount_bottom_old = 0;

// Weighted total for x positions and y positions of selected pixels, left or right of overall COM
reg [31:0] xcount_left = 0;
reg [31:0] xcount_right, xcount_right_old = 0;
reg [31:0] ycount_left = 0;
reg [31:0] ycount_right, ycount_right_old = 0;

// Top and Bottom
reg [31:0] xcount_top, xcount_top_old = 0;
reg [31:0] xcount_bottom, xcount_bottom_old = 0;
reg [31:0] ycount_top, ycount_top_old = 0;
reg [31:0] ycount_bottom, ycount_bottom_old = 0;

// Input to divider, sampled every clock cycle
reg [19:0] divider_in_pixels;
reg [31:0] divider_in_xcount;
reg [31:0] divider_in_ycount;

// Output from divider, sampled every clock cycle
// but values are only stored once per frame
wire [31:0] divider_out_x;
wire [31:0] divider_out_y;

// Resulting x and y center of masses (right and left) from previous frame
reg [10:0] leftave_x, rightave_x, topave_x, bottomave_x = 0;
reg [9:0] leftave_y, rightave_y, topave_y, bottomave_y = 0;

// Signal the beginning of the new frame
wire newframe = (hcount == 0 && vcount == 0);

// Add each pixel to count, and reset at the beginning of each frame
always @ (posedge clk) begin
// Handle systemwide reset, and reset at the beginning of each frame
if (reset || newframe) begin
// Shift pixel values over so they can be divided later
pixelcount_right_old <= pixelcount_right;
pixelcount_top_old <= pixelcount_top;
pixelcount_bottom_old <= pixelcount_bottom;
// Reset the current values so counting can begin
pixelcount_left <= 0;
pixelcount_right <= 0;
pixelcount_top <= 0;
pixelcount_bottom <= 0;

// Shift total count values over so they can be divided later
xcount_right_old <= xcount_right;
xcount_top_old <= xcount_top;
xcount_bottom_old <= xcount_bottom;
ycount_right_old <= ycount_right;
ycount_top_old <= ycount_top;
end
end

// Handle systemwide reset, and reset at the beginning of each frame
if (reset || newframe) begin
// Shift pixel values over so they can be divided later
pixelcount_right_old <= pixelcount_right;
pixelcount_top_old <= pixelcount_top;
pixelcount_bottom_old <= pixelcount_bottom;
// Reset the current values so counting can begin
pixelcount_left <= 0;
pixelcount_right <= 0;
pixelcount_top <= 0;
pixelcount_bottom <= 0;

// Shift total count values over so they can be divided later
xcount_right_old <= xcount_right;
xcount_top_old <= xcount_top;
xcount_bottom_old <= xcount_bottom;
ycount_right_old <= ycount_right;
ycount_top_old <= ycount_top;
end
end

Grebing, Stafl

74
ycount_bottom_old <= ycount_bottom;
// Reset the current values so counting can begin
xcount_left <= 0;
xcount_right <= 0;
ycount_left <= 0;
ycount_right <= 0;
xcount_top <= 0;
xcount_bottom <= 0;
ycount_top <= 0;
ycount_bottom <= 0;

leftave_x <= 0;
leftave_y <= 0;
rightave_x <= 0;
rightave_y <= 0;
topave_x <= 0;
topave_y <= 0;
bottomave_x <= 0;
bottomave_y <= 0;
delaycount <= 0;

// Start Calculating x and y averages for left right away
divider_in_pixels <= pixelcount_left;
divider_in_xcount <= xcount_left;
divider_in_ycount <= ycount_left;
end

// If the current pixel is selected, add its position to the totals
else begin
  // Handle pixels to the left
  if ( hcount < haverage ) begin
    pixelcount_left <= pixelcount_left + selected;
    if ( selected ) begin
      xcount_left <= xcount_left + hcount;
ycount_left <= ycount_left + vcount;
    end
  end

  // Handle pixels to the right
  else if ( hcount > haverage ) begin
    pixelcount_right <= pixelcount_right + selected;
    if ( selected ) begin
      xcount_right <= xcount_right + hcount;
ycount_right <= ycount_right + vcount;
    end
  end

  // Handle pixels at the top
  if ( vcount > vaverage ) begin
    pixelcount_top <= pixelcount_top + selected;
    if ( selected ) begin
      xcount_top <= xcount_top + hcount;
ycount_top <= ycount_top + vcount;
    end
  end

  // Handle pixels to the right
  else if ( vcount < vaverage ) begin
    pixelcount_bottom <= pixelcount_bottom + selected;
    if ( selected ) begin
      xcount_bottom <= xcount_bottom + hcount;
ycount_bottom <= ycount_bottom + vcount;
    end
  end

// Retrieve divided values, one at a time in following order:
// leftave, rightave, topave, bottomave
if ( delaycount == DIVIDER_DELAY ) begin
  leftave_x <= divider_out_x[10:0];
  leftave_y <= divider_out_y[9:0];
delaycount <= delaycount + 1;
end
else if ( delaycount == DIVIDER_DELAY + 1 ) begin
  rightave_x <= divider_out_x[10:0];
end
rightave_y <= divider_out_y[9:0];
delaycount <= delaycount + 1;
end
else if ( delaycount == DIVIDER_DELAY + 2 ) begin
topave_x <= divider_out_x[10:0];
topave_y <= divider_out_y[9:0];
delaycount <= delaycount + 1;
end
else if ( delaycount == DIVIDER_DELAY + 3 ) begin
bottomave_x <= divider_out_x[10:0];
bottomave_y <= divider_out_y[9:0];
delaycount <= delaycount + 1; // Will remain at DELAY + 4 until
reset again
end

// Send values in the be divided
else if( delaycount < DIVIDER_DELAY ) begin
// Increment by 1 in every case
delaycount <= delaycount + 1;
// Put in values for right
if ( delaycount == 0 ) begin
divider_in_pixels <= pixelcount_right_old;
divider_in_xcount <= xcount_right_old;
divider_in_ycount <= ycount_right_old;
end
// Put in values for top
else if ( delaycount == 1 ) begin
divider_in_pixels <= pixelcount_top_old;
divider_in_xcount <= xcount_top_old;
divider_in_ycount <= ycount_top_old;
end
// Put in values for bottom
else if ( delaycount == 2 ) begin
divider_in_pixels <= pixelcount_bottom_old;
divider_in_xcount <= xcount_bottom_old;
divider_in_ycount <= ycount_bottom_old;
end
end
end

///////////////////////////////////////////////////////////////////////////////
// Two Dividers for the 4 points, top, bottom, left and right. Values
// submitted every clock cycle, one at a time.

// Divider for x positions
wire [19:0] remaindery;
wire readyfordatay;
comdivider
comyl(.divisor(divider_in_pixels),.dividend(divider_in_ycount),.clk(clk),.quot(divider_out_y),
.remde(remaindery),.rfd(readyfordatay));

// Divider for y positions
wire [19:0] remaindery;
wire readyfordatay;
comdivider
comyl(.divisor(divider_in_pixels),.dividend(divider_in_ycount),.clk(clk),.quot(divider_out_y),
.remde(remaindery),.rfd(readyfordatay));

///////////////////////////////////////////////////////////////////////////////
// Angle Detection Code

// Remember which direction the skier has been facing, for continuity
reg [2:0] anglesignal = 0;
reg [2:0] newangle, oldangle = 0;

// Differential positions of the left, right, top and bottom averages
// Must be signed because diffy can be < 0, and difftopx < 0
reg signed [11:0] diffx;
reg signed [11:0] diffy;
reg signed [11:0] difftopy;
reg signed [11:0] difftopx;

// Get signed versions of the x, y positions
wire signed [11:0] leftave_xs, rightave_xs, topave_xs, bottomave_xs;
wire signed [11:0] rightave_ys, leftave_ys, topave_ys, bottomave_ys;

// Convert the unsigned signals to the signed signals
assign leftave_xs = {1'b0, leftave_x};
assign rightave_xs = {1'b0, rightave_x};
assign topave xs = {1'b0, topave_x};
assign bottomave_xs = {1'b0, bottomave_x};

assign leftave_ys = {2'b00, leftave_y};
assign rightave_ys = {2'b00, rightave_y};
assign topave_ys = {2'b00, topave_y};
assign bottomave_ys = {2'b00, bottomave_y};

always @ (posedge clk)
if (newframe)
beg
  diffx <= rightave_xs - leftave_xs;
diffy <= rightave_ys - leftave_ys;
difftopx <= bottomave_xs - topave_xs;
difftopy <= bottomave_ys - topave_ys;

  // Shift angles over
  oldangle <= newangle;

  // Determine new requested angle
  // Downhill left
  if (diffy > 15) begin
    if (diffy < -difftopx) begin
      newangle <= 1;
    end
    else
      newangle <= 2;
  end

  // Downhill right
  else if (diffy < -15) begin
    if (diffy > -difftopx) begin
      newangle <= 5;
    end
    else
      newangle <= 4;
  end

  // Either straight downhill, or to the left or right
  else if (diffy < 15 && diffy > -15) begin
    if (diffx < -difftopy) begin
      newangle <= 3;
    end
    else
      begin
        if (anglesignal == 6 || anglesignal == 5 || anglesignal == 4)
          newangle <= 6;
        else
          newangle <= 0;
      end
  end

  // If the new angle and old angle are the same, then make that the output
  if (newangle == oldangle)
    anglesignal <= newangle;
end