Pen-Raised Quail Hunt
6.111 Final Project

Daniel Lopuch and Zachary Remscrim

December 13, 2006

Department of Electrical Engineering
and Computer Science

Massachusetts Institute of Technology
Cambridge, MA 02139
**ABSTRACT**

Duck Hunt is among the most famous of the classic console games. This project sought to recreate the Nintendo classic using the MIT 6.111 FPGA Labkit and a video camera instead of the memorable Nintendo gun. To aim, the player wears a glove with two different color regions which the video camera uses to track the player’s in-game gun-sight and to shoot. Tracking is accomplished using video processing, and the color regions that the image processing is sensitive to is completely customizable by the user at run-time using the labkit’s buttons as inputs and VFD display as feedback. The game engine renders a recreation of the classic Duck Hunt playing field, but unlike the original, it is upsampled to be displayed on a 1024x768 XVGA display. The final implemented game met all given specifications and proved to be a faithful recreation of the original classic except for certain elements which added a socio-political commentary that expressed the views of the authors on the current state of American politics.

Rev 1: Added Verilog Appendix
# Table of Contents

List of Figures .......................................................................................................................... 5  
List of Tables ............................................................................................................................ 5  
Overview..................................................................................................................................... 6  
Overview..................................................................................................................................... 6  
System Inputs and Outputs ........................................................................................................ 6  
Video Decoding and Processing Overview .............................................................................. 6  
AI Overview .............................................................................................................................. 11  
Game Module Overview .......................................................................................................... 11  
Graphics Pipeline Overview ..................................................................................................... 11  
Project Description.................................................................................................................. 12  
Overview of Video Decoding and Processing Modules .......................................................... 12  
Filter Configuration Module .................................................................................................... 15  
RGB To HSV Converter .......................................................................................................... 19  
NTSC to ZBT and Filtering ........................................................................................................ 21  
Filtering ...................................................................................................................................... 21  
Centroid Calculation ................................................................................................................ 21  
Position Smoothening ............................................................................................................. 21  
Output Debouncing .................................................................................................................. 22  
Game Module ............................................................................................................................ 22  
AI Module .................................................................................................................................... 23  
Graphics Pipeline ..................................................................................................................... 24  
Duck Module .............................................................................................................................. 25  
Crosshair Module ...................................................................................................................... 25  
Tree and Grass Modules .......................................................................................................... 26  
Background Module .................................................................................................................. 26  
Score Overlay Module ............................................................................................................. 26  
Round-Over Overlay Module .................................................................................................... 26  
Title Screen Module .................................................................................................................. 27  
Round Timer Module ................................................................................................................ 27  
Testing and Debugging ............................................................................................................. 27  
Video Decoding and Processing ............................................................................................... 27  
Game Engine Debugging .......................................................................................................... 28  
Conclusions ............................................................................................................................... 29
APPENDIX

Appendix 1: Verilog Code................................................................. 30
  Base Labkit File ................................................................. 30
  XVGA Signal Generator ...................................................... 41
  VRAM Display ................................................................. 41
  Filter Limits Configuration .................................................. 43
  NTSC TO ZBT And Filtering ................................................ 55
  RGB To HSV Color Converter .............................................. 66
  Game Engine ................................................................. 72
  Graphics Pipeline ............................................................ 80
    Background Sky ............................................................ 80
    Duck ................................................................. 80
    Tree and Grass ........................................................... 82
    Crosshair ................................................................. 83
    AI ................................................................. 84
  Round Timer ................................................................. 88
  Score Overlay ............................................................... 89
  Round Over Overlay ........................................................ 92
  Title Screen ............................................................... 93
LIST OF FIGURES

Figure 1: Summary of System Inputs and Outputs............................................................. 7
Figure 2: Summary of Video Decoding and Processing Blocks Logic .............................. 7
Figure 3a: 3D Conical Representation of the HSV Color Space ........................................ 9
Figure 3b: 2D Representation of the HSV Color Space ..................................................... 9
Figure 4: A Color Filter Adjusted To Pass “Blue” Pixels .................................................. 11
Figure 5: Overview of Video Decoding and Processing Verilog Modules ...................... 14
Figure 6: Block Diagram of Filter Configuration Module................................................ 16
Figure 7: Slow-Fast FSM State Transition Diagram ........................................................ 17
Figure 9: RGB to HSV Color Converter.......................................................................... 20

LIST OF TABLES

Table 1: VFD Menu Options for Filter Configuration Parameter Selection ..................... 15
**OVERVIEW**

**System Inputs and Outputs**

The primary input to the game which moves the player’s gun sight is a glove with two different color regions. The player moves his hand in front of a video camera, and the video camera tracks the glove’s color regions to create the player’s in-game coordinates. Although the game is played at a low resolution to maintain the original Nintendo look, it is rendered as a 1024x768 XVGA signal. Push-buttons are used to select between game-play and camera calibration.

The player can use switches and another set of buttons on the labkit to customize the input camera filters to be sensitive to any two color regions he desires. This customization allows the user to fine-tune the default color filter settings to suit his particular lighting conditions and camera. This also allows the player to use any input device he can conceive of that has two distinct color regions; the ambitious enthusiast is encouraged to tweak the system for maximum accuracy. Filter settings are displayed on the labkit’s VFD display. The filtered video signal is displayed onto the XVGA output through an in-game calibration screen.

Figure 1 is a summary of the system’s input and output.

**Video Decoding and Processing Overview**

The primary objective of the Video Decoding and Processing blocks is to generate player coordinates and a fire control signal for the game engine. This is done by looking at the input NTSC video stream from the camera and filtering for regions of user-defined color. The coordinates of pixels that pass the filters are added to a center-of-mass calculator which simply averages all passing pixels’ coordinates. These center-of-mass averages, updated once per NTSC frame, are fed into the game engine as the player’s coordinates.

The secondary objective of the Video Decoding and Processing blocks is to display the results of the color filtering and the center-of-mass calculation to the user. This proved to be one of the initial significant design challenges. The main problem associated with displaying the video feed to the user is that the user’s display is a XVGA signal clocked at 60mhz while the video feed operates on a 27mhz clock. Because these two different clock rates are not simple multiples of each other, they had to be operated asynchronously. A ZBT RAM module was used as a frame buffer to solve this problem, but because the ZBT module is a single-port memory chip, a multiplexing scheme had to be designed to allow proper operation. A summary of the Video Decoding and Processing blocks is shown in Figure 2.
Figure 1: Summary of System Inputs and Outputs

Figure 2: Summary of Video Decoding and Processing Blocks Logic
Filtering Overview

The system actually needs to look for two separate regions of color: one corresponding to a “track” hand gesture and one corresponding to a “fire” hand gesture. For example, in the project demonstration, a glove was provided that had a yellow-colored square on the part of the hand that was visible to the camera when the hand was open; this region corresponded to the game engine merely tracking the user’s hand and moving his gun-sight appropriately. When the user closed his hand into a fist, a blue-colored square was attached to the part of the hand that was visible to the camera; this region corresponded to the game engine shooting at the flying duck. The Video Decoding and Processing blocks actually contain two color-region filters which can be independently adjusted to be sensitive to different regions of color. The filter which had the most passing pixels (and therefore tells the logic which coordinates to send to the game engine and whether or not the user is firing or just tracking) is represented using the WINNING_FILTER line.

The goal of the color filtering is to be able to specify regions of colors that are of interest to the user; if the region on the player’s glove corresponding to tracking is a yellow-colored square, we want to be able to tweak the color filter to be sensitive only to regions of yellow. We are not interested in the intensity of the yellow since this can change due to different lighting conditions or camera. What is of interest to us is to specify the filter to be sensitive to all shades of “yellow.” The problem now at hand is which color-space would give the solution to this problem the most intuitive customization. Three color-spaces were considered: RGB, YCrCb, and HSV.

RGB color space is defined as the “intensity” of the red, green, and blue elements of a pixel. This makes the RGB color space poorly-suited to easily select a region of a particular “color”; to ignore intensity and concentrate on the desired color, one must look at the differential across the three channels. The RGB color space was quickly disregarded as a potential color space to do the filtering work.

The next color space considered was the camera’s native YCrCb color space. This color space was attractive because intensity is defined in the Y (or luminance) channel and “color” is defined using the Cr and Cb (or Chrominance-red and Chrominance-blue, respectively) channels. If intensity is not the main parameter, it could easily be ignored. However, specifying a particular “color” is initially non-intuitive because the “color” is a function of two variables – Cr and Cb. One of the design objectives of the project was to allow very simple and intuitive control over the color region of interest, and so this color space was disregarded on the grounds of it being too difficult to specify a particular color region.

The final color space that was considered was the HSV, or Hue-Saturation-Value color space. In HSV, the value of the Hue channel specifies a color, the value of the Saturation channel specifies how much “white” there is in the pixel, and the value of the Value channel specifies how much “black” there is in the pixel (equal Saturation and Value in a color shifts the color towards neutral-gray). Thus, specifying a region
of “color” can be easily accomplished by specifying a range of Hues and ignoring the Saturation and Value channels. A visualization of the HSV color space is shown in Figures 3a and 3b.

**Figure 3a: 3D Conical Representation of the HSV Color Space**


**Figure 3b: 2D Representation of the HSV Color Space**

Due to the simplicity of specifying a particular “color” in the HSV color space (a color is the function of just one parameter instead of two, as in YCrCb), it was
decided that filtering would be done with a transformation into the HSV coordinate space.

Filtering was originally implemented by just specifying a low cut-off for Hue and a high cut-off for Hue. However, during testing, it was discovered that whiter pixels tended to transform into the full spectrum of hues; a pure white pixel technically has an undefined hue, but as soon as a little amount of noise is added, the Hue value of the HSV transformation can easily become almost any hue. In other words, it was found that when a white-ish region was picked up by the camera, it tended to transform into some pixels in every hue-pass region, creating noise for the center-of-mass calculator. It was decided that a way to get rid of this noise was to filter out low saturation pixels, in essence adding a Saturation-Pass region to the Hue-Pass region.

Thus, each of the system’s Color Filters is in fact a customizable Hue-Pass Filter and a customizable Sat-Pass Filter. For a pixel to pass one of the Color Filters, it must pass both the Hue-Pass Filter (if enabled) AND the Saturation-Pass Filter (if enabled). It was found that near-black pixels did not create a significant amount of the same problem, and so no filtering is done on the Value channel (although future revisions of the project should include Value-Pass filter option). Each color filter is therefore parameterized by four values: a Hue-Pass Low Mark, a Hue-Pass High Mark, a Saturation-Pass Low Mark, and a Saturation-Pass High Mark. The system has two independent color filters, so for the user to be able to completely customize the filters, the Filter Configuration module must be capable of adjusting eight independent parameters. Figure 4 is a visualization of one of the system’s two color filters, adjusted to pass “blue” pixels.
Figure 4: A Color Filter Adjusted To Pass “Blue” Pixels

**AI Overview**

The AI Module is responsible for controlling all actions of the AI player. This module receives state information from the Game Module, such as the duck’s position and heading, and generates a decision on what action to take next. This decision will include both considering where to move its crosshair as well as whether or not to fire. The AI has multiple difficulty settings, which affect its behavior. On higher difficulty settings, it tracks the duck extremely well and fires accurately, while on lower difficulty settings, it tracks the duck poorly and misses frequently. This is implemented by adding a small error term (the magnitude of which varies by difficulty) to the AI’s perception of the duck’s current and future location. By making the AI purposely miss and track without perfect precision, a more realistic, human-like opponent is simulated.

**Game Module Overview**

The Game Module is responsible for storing and updating all game state information, enforcing all game rules, and providing display data to the display controller. The game state, which includes position and status of all game objects, is updated once per frame of video. For example, at the beginning of every new frame of video, the duck is moved to a new position, any attempted shots are processed, etc. Game rules include game time limit, shots per round, scoring, etc. This module does not generate the display data internally, but rather sends state information to the various graphics modules, which return appropriate graphics information.

**Graphics Pipeline Overview**

The graphics pipeline is not a single module, but rather a collection of related graphics modules organized into a pipeline. Each module stores the bitmaps related to a single game object, for example the duck or a crosshair. The Game Module then sends an hCount, vCount pair to each module, which represents the coordinate of a pixel on screen. If a given module represents an object which has a pixel at that location, it will output the RGB value of that pixel, and assert a hasPixel signal which indicates that it has color information; otherwise, it will output nothing, and not assert hasPixel. All modules compute the RGB color value of a given pixel in parallel, and the module highest in the pipeline that asserts hasPixel will have its color information displayed on screen, for the pixel in question.
PROJECT DESCRIPTION

Overview of Video Decoding and Processing Modules

Authorship: Daniel Lopuch

The overall architecture of the Video Decoding and Processing Modules is based loosely off of the “ZBT RAM Example” sample code from the 6.111 Fall 2005 website. Although this sample file proved to be a good reference for code on decoding the raw NTSC data stream into YCrCb pixels and for interfacing with the ZBT memory, the shortcomings of the sample file were made painfully obvious when the jump from black and white to color was made. An overview of this project’s video decoding and processing modules is illustrated in Figure 5.

The main issues that had to be resolved involved multiplexing the writing and reading of the pixel data. The NTSC decoder (which generates the data to be written) operates at 27mhz while the XVGA display (which reads the data) operates at 60mhz. These different clock rates result in asynchronous read and write requests. The “ZBT RAM Example” source took advantage of the fact that when interested only at the black and white data, it is possible to store 4 pixels in one word of ZBT RAM. The XVGA display would read from the ZBT RAM once every four pixels and let the NTSC-to-ZBT module write during the off-time.

When color information is introduced, it is no longer possible to store four pixels in one word of ZBT RAM. ZBT RAM words are 36-bits long. It is possible to store two color pixels per ZBT RAM word if one decides to truncate the two least significant bits from the 8-bit RGB channels, but this approach was not considered because maximum precision was desired for the color filtering and this approach would lose some color depth. It was therefore decided that the system would store one pixel per word, and the problem became how to allow the NTSC processing module to write to the frame buffer when the XVGA display was reading from it during every new pixel.

The solution to this problem was developed when the realization was made that the XVGA display of the NTSC video stream is for the player’s feedback only and the true accuracy of the picture really does not matter as long as the player doesn’t perceive any imperfections. It was decided to allow the NTSC processing module to write to the ZBT buffer whenever it had new data. Whenever it did not have any new data, it would allow the display module to read whatever pixels it needed. During those pixels where the NTSC processing module wrote new data, the display module would read out junk data because it was overwritten by a write-request. This resulted in “snow” noise that would drift up along the displayed XVGA signal.

The “snow” was fixed by using a simple interpolator. The junk pixels could be predicted by delaying the write-enable line from the NTSC processor. Whenever the display module would read a junk pixel, it would simply override that pixel with the
previous clock cycle’s pixel. This is accomplished with the “Frame Buffer Write-Conflict Interpolator” in Figure 5. This proved to be a very effective solution – although the displayed picture was technically not the exact NTSC picture, the interpolation was on such a fine scale that the error was unperceivable.

Another significant difference between the “ZBT RAM Example” source and the developed system was in the number of pixels stored in RAM. Because the example source used four pixels per word, it was able to store the entire 1024x768 XVGA frame in RAM; the display module simply read off and displayed each pixel. However, the 512k ZBT RAM chip is not big enough to do the same storing one pixel per word. Instead, only the 720x480 NTSC frame is stored in the RAM, and any XVGA pixels outside of this range are simply masked to black after the display module receives the pixel from the RAM (see “NTSC Mask” in Figure 5).

After the pixel information is retrieved from RAM and passes through the Frame Buffer Write-Conflict Interpolator and the NTSC Mask, it is passed to a Filter Display Control module. This module has two purposes. The first is to read the center-of-mass averages from the Filtering module and make a visualization of them across the entire NTSC frame, and the second purpose is to determine which center-of-mass should be used by the game engine and to scale it to the appropriate coordinate space. The center-of-mass visualization is just vertical and horizontal lines that form crosshairs onto the center-of-masses of each of the two color filters in the NTSC video frame. The crosshairs change color depending on which color filter passed the most amount of pixels (logically corresponding to which of the two center-of-mass averages the game engine should use as the player coordinates and whether the game should interpret the position as a tracking position or as a firing position). The filter that has the most passing pixels is represented using the winningFilter control line. Finally, the Filter Display and Control scales the winning average from the 720x480 coordinate space into the 1024x768 coordinate space used by the Game Engine and gives these coordinates to the Game Engine.

The VRAM Display Module simply uses hcount and vcount to generate an address with appropriate NTSC clipping and then passes the RAM data onto the other logic, and the Filter Display and Control is simply a collection of combinational logic that overrides the current pixel with an appropriately colored line if the pixel lines on a centroid’s x or y coordinate. Because the simplicity of these modules, they will not be discussed in any detail.
Figure 5: Overview of Video Decoding and Processing Verilog Modules
**Filter Configuration Module**

Authorship: Daniel Lopuch

As described in the Filtering Overview section, each of the system’s two color filters must have an adjustable Hue-Pass Low Mark, Hue-Pass High Mark, Saturation-Pass Low Mark, and Saturation Pass High Mark. These values are modified by the user on the fly using the Filter Configuration Module. The Filter Configuration Module uses the four directional push-buttons on the labkit as inputs and the labkit’s VFD display as a feedback to the user.

Selection of the particular parameter of interest is accomplished using a menu-system on the VFD display. Each menu item on the VFD display is one of the two color filter’s hue or saturation low and high marks. The VFD display can show one line of 16 characters, and the possible menu items are listed in Table 1:

<table>
<thead>
<tr>
<th>VFD Character #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
<td>h</td>
<td>1</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>h</td>
<td>i</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Item 2</td>
<td>s</td>
<td>1</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>h</td>
<td>i</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Item 3</td>
<td>h</td>
<td>2</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>h</td>
<td>i</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Item 4</td>
<td>s</td>
<td>2</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>h</td>
<td>i</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Each Menu Item will from here on be referred to as the Configuration Mode. Each item (Mode, Low-Mark, High-Mark) will be from here on referred to as the Configuration Object. Objects are selected by using the left and right buttons to move left or right. As a new Object is selected, its display changes from lower-case to upper-case and it begins blinking. The Object is changed by selecting it and pressing the up or down buttons. Because the range of the selection values is significant but precision is required in adjust the value, it is desirable to be able to go through the possible values both quickly and slowly. This is accomplished using a Slow/Fast FSM. When the FSM detects the user has pressed and holds a button, it initially increments or decrements the value by one at a slow interval. If the user keeps holding the button, it begins to increment or decrement the value rather quickly. Figure 6 shows a block diagram of the individual modules in the Filter Configuration Module, and Figure 7 is a state-transition diagram of the Slow/Fast FSM.
Figure 6: Block Diagram of Filter Configuration Module

Controls the Object and the Mode that we're changing.

Object refers to whether we're changing the Mode, the low mark, the high mark, or both the low and high marks at once.

Mode refers to Hue1, Sat1, Hue2 or Sat2.

Together, the Object and the Mode address each of the marks.
Figure 7: Slow-Fast FSM State Transition Diagram

Note: Transitions occur on the rising edge of each clock cycle. Shaded blocks are states. Non-shaded blocks are combinational logic pieces that determine the particular transition path.

The value of each mark is stored in the appropriate Mark Control Module. Each Mark Control Module has increment and decrement inputs. For every clock cycle in which its increment and decrement inputs are high, it increments or decrements its stored value appropriately. The Mark Control Verilog modules can be parameterized to use any minimum or maximum limits for their values, and they can be parameterized to use any value as their default start values. Thus, even though the Hue Marks go from 0 to 360 and the Saturation Marks go from 0 to 100, the same Verilog module was used but instantiated with different parameters.
Because the end-user experience was a priority for the filter customization, it was important that the VFD display show the filter parameters in common base-10 rather than a radix that only computer programmers would be familiar with. To generate the appropriate digits to be displayed, a divider module was used with a pipelined dividend/divisor multiplexer scheme. Figure 8 briefly illustrates the scheme used.

![Diagram of Pipelined Base-10 Digit Generator]

Figure 8: Pipelined Base-10 Digit Generator

NOTE: count increments on every clock cycle. highMark and lowMark are the current low and high marks of interest.
RGB To HSV Converter

Authorship: Daniel Lopuch

An RGB to HSV transformation is done using the following formulas:

\[
H = \begin{cases} 
    \text{undefined}, & \text{if } MAX = MIN \\
    60 \times \frac{G-B}{MAX-MIN} + 0, & \text{if } MAX = R \\
    60 \times \frac{G-B}{MAX-MIN} + 360, & \text{if } MAX = R \\
    60 \times \frac{B-R}{MAX-MIN} + 120, & \text{if } MAX = G \\
    60 \times \frac{R-G}{MAX-MIN} + 240, & \text{if } MAX = B 
\end{cases}
\]

\[
S = \begin{cases} 
    0, & \text{if } MAX = 0 \\
    1 - \frac{MIN}{MAX}, & \text{otherwise} 
\end{cases}
\]


Lacking a generalized ALU, each of the embedded operations were performed using an independent math module. In total, 4 adder/subtractors, 2 multipliers, and 2 dividers were used along with many registers that generated the appropriate synchronization delays to make a pipelined RGB to HSV Converter. Figure 9 shows a block-diagram of the implementation used. Note that because it was deemed unnecessary to add Value-checking to the filters, RGB to Value conversion is not implemented.

The total latency of the converter is 24 clock cycles. However, the converter is pipelined, so the throughput is still one conversion per clock cycle. The Filtering Module works by checking to see if the HSV values are within specified filter ranges, and if they are, it simply passes the RGB values through to the display pipeline. However, because the HSV results are delayed by 24 clock cycles, a 24-cycle pipelined delay was added to the RGB line to synchronize it with the results from the RGB to HSV transformation.
Figure 9: RGB to HSV Color Converter
**NTSC to ZBT and Filtering**

Authorship: Daniel Lopuch

The NTSC to ZBT and Filtering module is based initially off of the “ZBT RAM Example” from the 2005 website. The example code does the conversion from YCrCb to RGB, generates the appropriate address, and synchronizes the RGB with the XVGA clock, and generates the appropriate write-enable signal when the new data is ready.

Some minor changes were done in the address generation to fit the new 1-pixel-per-word scheme. In addition, the appropriate delays were added to the RGB, address, and write-enable lines to synchronize them with the added HSV converter.

**Filtering**

In addition to the Hue-Pass and Saturation-Pass filters, a noise filter was added. During testing it was found that many small regions were picked up by the camera that happened to be inside the defined Color-Pass region. A noise filter was added that checks whether or not the last seven pixels passed the Color-Pass filters. If they did, and if the current pixel did as well, the current pixel passed the noise filter. This was implemented using a simple shift register for the Color-Pass result. When the color filters were tuned properly, the noise filter was effective at eliminating a lot of the noise without eliminating much of the regions of interest.

The filter-pass logic was wired such that each of the hue-pass, saturation-pass, and noise filters could be turned on or off using the labkit switches.

**Centroid Calculation**

The centroid, or center of mass, of each of the two filters was calculated by simply adding the x- and y-coordinates of each pixel that passed the respective filter to a separate sum. A counter kept track of how many pixels passed, and when the frame was over, and averager calculated the average. The delay in calculating the average was over 25 cycles at 60mhz, but because the value was updated only once per NTSC frame and because it was not really important when the average would update itself, this was not a problem.

The count of passing pixels was used later to determine which filter had the most passing pixels, and so which filter should be used to calculate the player’s coordinates and determine whether the player is shooting or not.

**Position Smoothening**

During testing, it was observed that the centroids would jitter very quickly from frame to frame. A smoothening averager was added to address this problem. The smoothener simply took the average of the current centroid position versus the average of the previous frame’s centroid position. Mathematically, this results in an infinite average of past centroids, but each location further back in time is weighted less and less.
Output Debouncing

In testing, it was observed that the reported winning filter would sometimes quickly switch when the player’s gesture did not change. The cause of this was determined to be slight increases of noise for one filter would temporarily push its filter’s passing count above the other filter’s passing count. A debouncer that sampled once per NTSC frame was added to debounce the winning filter output and eliminate false-positives.

Game Module

Authorship: Zachary Remscrim

The purpose of this module is to store the current state of the game, update the state of the game based on the current state and current inputs, and provide data to the display controller. The game state includes: the duck’s position and heading, player’s score, AI’s score, number of shots the player has remaining this round, etc. The game state is updated once per frame of video (there are 60 frames of video per second). This is necessary since, if the state of the game were to change in the middle of a frame, part of the displayed frame would correspond to the old state, and the other part would correspond to the new state. This could, possibly, create a completely incorrect displayed image. For example, the duck could appear in two places at one time. To avoid this, state updates occur once per frame of video and are synchronized to the transition between frames of video. This is done by observing the vertical sync signal from the XVGA module, and executing a state transition whenever the vertical sync signal first goes low.

During a game state update, the states of all game objects are updated in parallel. The duck is moved to a new position, which is determined by its current position plus a small offset in the direction of its current heading. If this would move the duck outside of its valid region (which is bounded by the top, left, and right sides of the screen, as well as by the horizontal line which represents the upper boundary of the dirt image), then the duck would instead be reflected off the boundary. Since the duck is an animated sprite, the game module must also update the state of the animation. When the duck is flying, the animation has three possible states, which correspond to three different positions of the duck’s wings. A given state is maintained for several frames of video, then the animation enters the next state. There are also animation states which correspond to the image to be displayed when the duck is first shot, as well as when the duck falls from the sky. These animation states are entered after the duck has been shot by either the player or the AI. In order to generate proper animation state information, a simple FSM was implemented in which each animation position was a state. State transitions are governed by the rules outlined above. This animation state information, as well as the duck’s current position, is sent to the Duck Module (one of the graphics module) which will then return the proper image data. More details on this process will be included later in this section as well as in the section concerning the Duck Module.
Additionally, during a game state update, any shots fired by either the human player or the AI player are processed. If the firing player has no shots remaining this round, then nothing will occur. Otherwise, the number of shots that that player has will be reduced by one; and, if that player’s cursor currently overlaps any portion of the duck, then the duck will be killed and that player’s score will be updated accordingly. The score will be increased by a number of points inversely proportional to the time that has elapsed since the beginning of the round. For example, if less than 1 second has passed, the player will earn 300 points, while if 3 seconds have passed the player will only earn 100 points. Time is kept through instances of the RoundTimer Module. The score information and number of shots left will be sent to the ScoreOverlay module, which will be responsible for producing the appropriate display data.

A round can end in one of two ways. Either the duck can be killed, or the round timer could expire. If the duck has been killed, it will fall until it reaches the ground. If sufficiently long period of time has passed (specifically, 10 seconds) and neither the human player nor the AI player has successfully shot the duck, then the round will end and the duck will be removed from the screen. In either case, a special graphic will popup at the end of the round. If the player killed the duck, the popup will depict Cheney holding a dead duck. If the AI killed the duck, or the round ended due to time expiring, the popup will depict Cheney holding a gun. This will be accomplished by providing the RoundOverOverlay module with a signal that specifies which player killed the duck, if any, which will cause the RoundOverOverlay module to produce the appropriate display data. After a period of time has passed, the duck will return to the starting position, and a new round will begin. A game consists of a total of ten rounds, at which point the player can start a fresh game if they choose to.

As mentioned earlier, part of the purpose of this module is to provide data to the display controller. The Game Module does not produce this data on its own. Instead, it makes use of a series of graphics modules, each of which corresponds to a single game object, to produce the appropriate display information. These modules include the Duck Module, ScoreOverlay Module, and RoundOverOverlay Module mentioned already, as well as the TreeGrass Module, Background Module, and Crosshair Module. The game engine sends appropriate state information, as well as the coordinates of a pixel to these modules, and they each return the appropriate RGB color information, if the object they represent has a pixel at that location, to the Game Module. The Game Module then outputs the pixel that corresponds to the object closest to the foreground at the coordinates in question. Specific details concerning the operation of these modules can be found in the section dealing with the graphics pipeline.

**AI Module**

Authorship: Zachary Remscrim
The purpose of this module is to control the actions of the AI player. Specifically, this module will determine where the AI’s crosshair will be moved at the next game state update, as well as whether or not the AI will fire at any given time. These decisions are sent to the game engine and are processed in an identical manner to the player’s decision on movement and firing. This assures that the AI is given no special advantages or disadvantages. The game AI has the ability to be set to one of several difficulty settings, which affects how well the AI tracks the ducks movements and how accurate the AI’s shots are.

To determine where to move the crosshair, the module examines the duck’s current location and heading. It then estimates the location that the duck will be in at the next game state update. A small amount of error is purposely introduced here to cause the AI to not track the duck perfectly. This is done to make the AI’s play more human-like. The amount of error varies by difficulty level. After the estimated location has been found, the AI will move its crosshair in the direction of the estimated location. The distance moved is limited to avoid the crosshair moving immediately to the duck’s location. This limit also varies by difficulty level.

If the AI has any shots left (it has the same limit of three shots per round that the player has), then it will decide whether or not to shoot. If the location to which the AI has decided to move its crosshair at the next game state update places the crosshair over the duck’s estimated position, and a sufficient amount of time has passed since the beginning of the round, then the AI will fire. Since the duck’s estimated position had a small amount of error purposely introduced, it is possible that the AI will miss. Again, this is done to make the AI’s play more human-like. The AI waits a small amount of time after a new round begins before it takes a shot in order to give the player more of a chance to kill the duck. The amount of time varies by difficulty level and is kept through an instance of the RoundTimer module.

**Graphics Pipeline**

Authorship: Zachary Remscrim

Due to the complexity of display data and the short clock period (15.4 nS), the various graphics modules mentioned earlier were organized into a graphics pipeline. They are each given the appropriate state information and values for hCount and vCount (the horizontal and vertical coordinates, respectively, of the pixel whose value is currently being requested) simultaneously, and compute the appropriate RGB value for that pixel in parallel (if the given module corresponds to an object that has a pixel at those coordinates). For a given hCount and vCount, the pixel color that will actually be displayed is the RGB value produced by the module highest in the pipeline that has color information for that pixel. This produces a layering effect whereby objects higher in the graphics pipeline are displayed above objects lower in the graphics pipeline. For example, the TreeGrass Module is above the Duck Module, which is above the Background Module in the graphics pipeline. Therefore, the tree and grass will be displayed over the duck, which in turn will be displayed
over the background. This will have the desired effect of having the duck appear as though it is flying behind the tree and grass, while still appearing above the sky.

Each module (with the exception of the background) consists of one or more single-port ROM modules, created using the Xilinx tools, that stores the bitmaps of the various objects. In general, these ROMs are 8 bits wide, and each entry stores the RGB values of a single pixel. For dichromatic images, such as the crosshair, the ROM is only 1 bit wide. In that case, each entry would be a 1 to represent one color, and a 0 to represent the other. The module stores the full 8 bit RGB values for both the “1” color and the “0” color, and returns the appropriate RGB value when either a 1 or 0 is encountered in the ROM. This was done to save memory. Each address corresponds to a single pixel of the bitmap. All RGB values are encoded in the 8 bit truecolor format, in which the high order 3 bits correspond to red, the next 3 bits correspond to green, and the low order 2 bits correspond to blue. Since the ADV7125 expects 24 bit color (8 bits for each of red, green and blue), an appropriate number of zeroes is added as low order bits to turn the 2 or 3 bit values into 8 bit values. For example, if a given pixel had the 8 bit RGB value 10111011, then 10100000, 11000000, and 11000000 would be the red, green, and blue values sent to the ADV7125. In many modules, the bitmaps stored in the ROMS were compressed relative to their display size. This was done to save memory. In order to display the image at the appropriate size, the same address in memory is accessed for several different hCount, vCount pairs.

In order to facilitate the creation of the .coe files needed by the ROMS, a Matlab script was written to convert .jpg, .gif, and .bmp files to .coe format. This made the graphics creation far easier since it eliminated the otherwise absurdly tedious task of generating .coe files.

The modules that make up the graphics pipeline are: Duck, Crosshair, TreeGrass, Background, ScoreOverlay, RoundOverOverlay, and TitleScreen.

**Duck Module**

This module is responsible for displaying the animated duck. It possesses 6 ROMs, each of which corresponds to one of the 6 possible duck images. 3 of these images form the flying animation, 2 form the falling animation, and the last is displayed when the duck is first shot. The Game Module sends a state signal to this module to specify which image should be displayed. For example, when the duck is in its standard flying state, the state signal repeatedly cycles through the 3 flying images. Thus, it appears as though the duck is flapping its wings while flying. The Game Module also provides the current coordinates of the top left corner of the duck. This allows the Duck Module know where the duck is, and provide RGB information for a given hCount and vCount.

**Crosshair Module**

This module is responsible for displaying a crosshair. There are two instances of this module in the device, one for the player’s crosshair, and another for the AI’s
crosshair. Since the bitmap for a crosshair is dichromatic (the “X” portion of the
crosshair is one color, and the background is another color), the ROM that stores the
crosshair bitmap is only 1 bit wide, for the reason discussed above. This module has
a color parameter, which specifies the color to be returned whenever a portion of the
“X” is encountered in memory. This parameter is set to a different value in each
instance of the module, thereby allowing the two crosshairs to be distinguished. No
color information is returned when the background is encountered. This is done to
allow the image of whatever the crosshair is over to still be visible, while still
accurately displaying the “X” portion. As was the case for the Duck Module, this
module needs the coordinates of the object that it represents. In this case, the Game
Module transmits the x and y coordinates of the center of the “X”. Since the
crosshair is not animated, no state information is needed.

Tree and Grass Modules

This module stores the color information for the tree, grass, and dirt. Since all
these objects are static, neither state information nor coordinates are required. For
convenience, the bitmap of the tree is stored in a different ROM than the bitmap of
the grass and dirt.

Background Module

Unlike the other modules that make up the graphics pipeline, this module has no
need for a ROM. This module simply returns a blue pixel for any hCount, vCount
pair in the sky, and a brown pixel otherwise. If the device is working properly, this
brown pixel will never be displayed since the grass and dirt are displayed over it. It
was included to assure that something reasonable is displayed even if there is a glitch
in the TreeGrass Module.

Score Overlay Module

This module is responsible for displaying all score related information, as well as
the number of shots that the player has left. The score information includes the
player’s score and AI’s score, which are computed by the Game Module. Additionally,
there is a display of the 10 ducks that appear in a round. The color of
the duck corresponds to its state. White indicates that the duck has not been killed,
red indicates that it was killed by the AI, and green indicates that it was killed by the
player. This data is provided by the Game Module.

Round-Over Overlay Module

This module is responsible for displaying a graphic popup whenever the round
ends. The graphic popup is either a fake image of Dick Cheney brandishing a rifle or
a fake image of Dick Cheney holding a dead duck. The Game Module sends signals
to this module indicating the player that killed the duck, or that the duck was not
killed, as well location of the top left corner of the Dick Cheney image (which is
needed since the image slowly rises from behind the grass). If the human player
killed the duck, the popup will show Cheney holding a dead duck. If the AI player
shot the duck, or no player shot the duck, the popup will show Cheney holding a
rifle.
**Title Screen Module**  
Authorship: Zachary Remscrim

This module displays the title screen. It receives a signal from the Game Module which specifies whether or not the title screen should be displayed at any point in time.

**Round Timer Module**  
Authorship: Zachary Remscrim

The purpose of this module is to keep track of the passage of time. Upon receiving a start signal, the module will increment a counter variable at each clock edge until that variable reaches a threshold, at which point it will assert a time expired signal. The time expired signal will remain asserted until this module is reset. This threshold is determined by multiplying the clock frequency by the length of the desired timing interval. The length of the timing interval is a parameter that can be set on instantiation. This allows different instances of the module to measure different lengths of time. Several instances of this module are present throughout the device. For example, the Game Module uses an instance of this module to determine when the time limit of a round has expired.

**TESTING AND DEBUGGING**

**Video Decoding and Processing**

When the project was first conceived, the video filtering was simply a single hue-pass region. Testing, however, showed this was a dramatic oversimplification. The first problem that became apparent during testing was how insufficient a hue-pass filter region was by itself. As described in the Filtering Overview section, it was quickly learned that regions of near-white pixels tended to display as pixels of all hues; when the camera was focused on a white area, there would always be noise from there regardless of which hue was being passed through the filter. A second filter – the Saturation-Pass Filter – improved this problem significantly.

It was also realized early on that the customization of the filter parameters was a necessity rather than a convenience. Depending on the particular lab bench chosen for the day and the position of the sun through the outside windows, the lighting conditions would change enough that if the camera was calibrated for a certain color at one bench, at another bench or at a different time of day the perceived color would change enough to make the calibration outdated.

A further complication in calibrating to perceived colors came from the camera itself. The camera that the group was assigned had built-in gain and color-shift circuitry. These are desirable features for a consumer electronics device since they ensure the best picture based on changing lighting conditions, but for us, this
unnecessarily complicated the project. We would find that we could calibrate the camera’s detection to work well for one region, but when we would move our hand closer to it or to another position (such as blocking a ceiling light from the camera’s view), the camera’s auto-gain would kick in and shift perceived colors out of our calibration region. We experimented with other groups’ cameras, and we found that other groups’ cameras lacked this auto-gain function and so performed significantly better.

Another grievance that was had with the assigned camera was the lack of picture quality. One issue that was noticed right away was for the camera to shift the picture towards green. Because it lacked significant color depth, any gray or dark pixels would often be shifted into the green hue region, and so if the filter was calibrated for green, there would be a significant amount of noise. Again, other groups’ cameras, although older, proved to have a much greater color depth and noise tended to be inside the color’s actual hue region rather than shifted into the green region.

Both types of cameras, however, displayed significant trouble with sharp edges. When they would see a sharp edge, they would actually register pixels colored with colors that we not actually in the edge but just showed up as noise. Dealing with the noise encouraged the development of the noise filter, but an ultimately more effective and low-tech solution was found in simply blurring the camera’s focus— all we cared about was regions of color, and blurring the camera’s focus effectively eliminated edge-noise.

**Game Engine Debugging**

In order to facilitate easy debugging of the Game Module, AI Module, and graphics modules, careful attention was paid to assure their modularity. This greatly simplified testing since each module could be tested independently. During the early stages of the project, work was focused on the creation of the Game Module and the framework of the graphics modules. Initially, many of the bitmaps used by the graphic modules were simply single color rectangles that were used purely for testing purposes to verify that the module correctly displayed a simple image at the times dictated by the Game Module. Additionally, the input that would normally come from the Image Processing Module was replaced by a simple input through the labkit’s buttons. This allowed the basic functionality of the Game Module and graphics modules to be tested. Once this stage of the project was fully debugged, actual bitmaps were used by the graphics modules, the AI was introduced, and the Game Module had all desired functionality implemented. After the debugging, this stage, the final phase of the project began. This included integration of these modules with the Image Processing Module, as well as improving the general look and performance of the game.

While the initial Game Module and basic graphic modules were mostly bug free, many bugs were encountered once the AI was introduced. The initial behavior of the AI was problematic. It had a tendency to wander, not accurately track the duck, and
fire at strange times. After a closer examination of the AI module, including extensive use of the logic analyzer to examine the critical signals of this module, it was discovered that the AI’s calculation for the duck’s position had more error than intended (a small amount of error is purposely present, in order to make the AI’s play more realistic). Once this error was discovered, it was quickly corrected, and many of the bugs present disappeared.

CONCLUSIONS

The goal of this project was to create a device that enabled a user to play a game of Duck Hunt, with several added features. These features included the addition of a “Realistic Dick Cheney AI” which competes against the player as well as the replacement of the traditional “Duck Hunt Gun” with an input system based on moving one’s hand in front of a camera.

After extensive testing of all modules, it is clear that these goals have been satisfied. With the exception of a few minor, and rare, glitches, as well as a slight jitter presence in the input system, all aspects of the device behave as desired. All game rules are accurately followed, all graphics are displayed properly, the AI functions well, and visual inputs are successfully delivered to the game.

The design and implementation of this device demonstrated the value of modularity in digital systems (or any other system, for that matter). By separating the functionality of the device into several modules, each module could be tested and debugged independently. This greatly sped up the debugging process.

Furthermore, the modular nature of the device makes the system easier to understand due to the layers of abstraction it provides. The system as a whole can be analyzed from a global perspective without considering the specific implementation details of a given module. This has the further advantage of allowing a module to be revised for the purpose of improving efficiency. Any such internal change will not affect the overall behavior of the device if the utilized abstractions are used.
APPENDIX 1: VERILOG CODE

Base Labkit File

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

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

// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
// "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
// output of the FPGA, and "in" is an input.

// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.

// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
// the data bus, and the byte write enables have been combined into the
// 4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
// hardwired on the PCB to the oscillator.

///////////////////////////////////////////////////////////////////////////////
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
// "disp_data_out", "analyzer[2-3]_clock" and
// "analyzer[2-3]_data".

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

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

// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
// value. (Previous versions of this file declared this port to
// be an input.)

// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
// actually populated on the boards. (The boards support up to
// 72Mb devices, with 21 address lines.)
module DuckHunt   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
ac97_bit_clock,

vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
vga_out_vsync,
tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
tv_out_ycrcb, tv_in_data_valid, tv_in_line_clock1,
tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,
ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,
clock_feedback_out, clock_feedback_in,
flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b,
rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
mouse_clock, mouse_data, keyboard_clock, keyboard_data,
clock_27mhz, clock1, clock2,
disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_in,
button0, button1, button2, button3, button_enter, button_right,
button_left, button_down, button_up,
switch,
led,
user1, user2, user3, user4,
daughtercard,
systemace_data, systemace_address, systemace_ce_b,
systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,
analyzer1_data, analyzer1_clock,
analyzer2_data, analyzer2_clock,
analyzer3_data, analyzer3_clock,
analyzer4_data, analyzer4_clock);
output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input ac97_bit_clock, ac97_sdata_in;
output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
    vga_out_hsync, vga_out_vsync;
output [9:0] tv_out_yrcrb;
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;
in [19:0] tv_in_yrcrb;
in 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] raml_data;
output [18:0] raml_address;
output raml_adv_ld, raml_clk, raml_cen_b, raml_ce_b, raml_oe_b, raml_we_b;
output [3:0] raml_bwe_b;
in 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;
in mouse_clock, mouse_data, keyboard_clock, keyboard_data;
in 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;
in [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_synch = 1'b0;
assign ac97_sdata_out = 1'b0;

// ac97_sdata_in is an input

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

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

SRAMs
/* change lines below to enable ZBT RAM bank0 */
/*
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_clk = 1'b0;
assign ram0_we_b = 1'b1;
assign ram0_cen_b = 1'b0; // clock enable
*/
/* enable RAM pins */
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;
assign ram0_bwe_b = 4'h0;
**********/
assign raml_data = 36'hZ;
assign raml_address = 19'h0;
assign raml_adv_ld = 1'b0;
assign raml_clk = 1'b0;
assign raml_cen_b = 1'b1;
assign raml_ce_b = 1'b1;
assign raml_oe_b = 1'b1;
assign raml_we_b = 1'b1;
assign raml_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
// Lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

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

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

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

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

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset, user_reset;
debounce db1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;
reg [7:0] rgb;

// generate basic XVGA video signals
wire [10:0] hcount, hcount_inverse;
assign hcount_inverse = hcount <= 720 ? 720 - hcount : 0;
wire [9:0] vcount;
wire hsync, vsync, blank;
xvga xvga1(clk, hcount, vcount, hsync, vsync, blank);

 volcanolux

// HUE LIMITS CONTROLLER
// This part makes the hue-pass controlled by the labkit buttons
wire [8:0] hue1_lowMark, hue1_highMark, hue2_lowMark, hue2_highMark;
// hue values between 0 and 360 to pass
wire [6:0] sat1_lowMark, sat1_highMark, sat2_lowMark, sat2_highMark;
wire [16*8-1:0] string_data_out;
reg [16*8-1:0] string_data[1:0];
limits_interface li(clk, 1'b0, ~button_up, ~button_down, ~button_left,
    ~button_right, button_enter, // don't invert enter_button so that stuff happens
    hsync, vsync, blank, button_right, button_left, // buttonUp rather than on buttonDown
    button_up, ~button_down, ~button_right, button_enter, // don't invert enter_button so that stuff happens
    button_right, button_left, // buttonUp rather than on buttonDown
    button_up, ~button_down, ~button_right, button_enter, // don't invert enter_button so that stuff happens
    button_right, button_left, // buttonUp rather than on buttonDown
    hlowMark, hue2_lowMark, hue2_highMark,
    sat1_lowMark, sat2_lowMark, sat2_highMark,
    string_data_out);
//syncronize data to 27mhz clock
always @ (posedge clock_27mhz) begin
  (string_data[1], string_data[0]) <= {string_data[0], string_data_out};
end

assign led = hue_lowMark;

// END HUE LIMITS

// display_string ds(reset, clock_27mhz, string_data[1], disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

// 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(clk, 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
wire [23:0] vr_pixel; //Full RGB info
wire [18:0] vram_addr1;
vram_display vd1(reset,clk,hcount_inverse,vcount,vr_pixel, vram_addr1,vram_read_data);

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

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

// code to write NTSC data to video memory
wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire ntsc_we;
wire [9:0] x1_avg, y1_avg, x2_avg, y2_avg;
wire filterWinner; //Which filter is winning, 0=filter1, 1=filter2
/*DEBUGS:
wire [26:0] x_sum;
wire [26:0] new_x_avg;
wire [18:0] pixelCount;
wire startFinalAvgDividing, endFinalAvgDividing;
wire [9:0] x;
wire [7:0] y; */

wire [18:0] pixelCount1, pixelCount2;

ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb,
  ntsc_addr, ntsc_data, ntsc_we,
  1'b0, switch[5],
  hue1_lowMark, hue1_highMark, hue2_lowMark, hue2_highMark,
  sat1_lowMark, sat1_highMark, sat2_lowMark, sat2_highMark,
  switch[4], switch[3], switch[2], switch[1],
  x1_avg, x2_avg, y1_avg, y2_avg, filterWinner, 1,
  switch[0],
  pixelCount1, pixelCount2 );
//x_sum, new_x_avg, pixelCount, startFinalAvgDividing,
endFinalAvgDividing, x, y);

// code to write pattern to ZBT memory
reg [31:0] count;
always @(posedge clk) count <= reset ? 0 : count + 1;
wire [18:0] vram_addr2 = count[0+18:0];
wire [35:0] vpat = ( switch[1] ? {4{count[3+3:3],4'b0}} : {4{count[3+4:4],4'b0}} );

wire sw_ntsc = 1'b1;
wire [18:0] write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
wire [35:0] write_data = sw_ntsc ? ntsc_data : vpat;

//SWITCH BETWEEN READING AND WRITING MODES:

/*Method 1:
  Give vram priority to xvga
Here we write new data only when the current xvga pixel
outside of the NTSC area.
Result is you get "waves" of old frames in the display
since we can't write new video data
constantly. We take advantage of the fact that XVGA clock
and video clock are not multiples,
so because of non-constant interferance, we do write new
video data to all pixels eventually,
just not in the same frame.
wire read_select = (hcount <= 720) && (vcount <= 480);
//when this wire is 1, we want to reader.
wire is 0, we want to write.
assign vram_addr = (read_select) ? vram_addr1 : write_addr;
assign vram_write_data = write_data;*/
assign vram_we = ~read_select;

//NTSC Mask (during write mode, display b/w bars)
//wire [23:0] pixel = vr_pixel;
reg [23:0] pixel;
always @(posedge clk)
begin
  //pixel <= switch[5] ? vr_pixel : {hcount[8:6],5'b0, hcount[8:6],5'b0, hcount[8:6],5'b0};
  pixel <= (hcount <= 723 && vcount <= 483) ? vr_pixel : 24'b0;
end

/*Method 2:
Give vram priority to video
Here we write new data whenever it arrives. The problem is that certain xvga pixels will not be able to be reader. The advantage here is that we don't get "waves of old data" as in method 1.
The disadvantage is we get bits of random snow scattered throughout (ie at those pixels where new data is being written). We'll just display the previous pixel again. Thus the displayed image won't be true, but it shouldn't be noticable since we're repeating only single pixels. */
assign vram_addr = (ntsc_we) ? write_addr : vram_addr1;
assign vram_write_data = write_data;
assign vram_we = ntsc_we; //*/

// Delay the we by ZBT read + color conversion so we know when we hit a "snow pixel"
wire we_delay;
wire theDelayedWE;
//defparam we_delayer.NDELAY = 2; //2 for ZBT pipeline
reg delayed_we[1:0];
always @(posedge clk)
{delayed_we[1], delayed_we[0]} <= {delayed_we[0], vram_we};
wire theDelayedWE = delayed_we[1]; //*/

//NTSC Mask (during write mode, display b/w bars)
//wire [23:0] pixel = vr_pixel;
reg [23:0] pixel;
wire [23:0] pixel_out[2:0];
reg [23:0] oldPixel;
always @(posedge clk)
begin
  //pixel <= switch[5] ? vr_pixel : {hcount[8:6],5'b0, hcount[8:6],5'b0, hcount[8:6],5'b0};
  oldPixel <= vr_pixel;
  pixel <= (hcount <= 722 && vcount <= 482) ?
    (theDelayedWE ? oldPixel : vr_pixel) :
    //When the pixel was being written to (ie couldn't read it), interpolate by putting in the previous pixel
    24'b0;
  //else (when we're outside the NTSC box),
display blackness
//DRAWING FILTER TRACKERS
wire [8*3-1:0] filter1_tracker, filter2_tracker;
//filterWinner is which color filter has more pixels. ==0 -> filter1, ==1 -> filter2
assign filter1_tracker = (~filterWinner) ?
{8'd0, 8'd255, 8'b0} :
{8'd255, 8'd255, 8'b0};
assign filter2_tracker = (filterWinner) ?
{8'd0, 8'd255, 8'b0} :
{8'd255, 8'd255, 8'b0};

assign pixel_out[0] = (hcount_inverse == {1'b0, x1_avg} || vcount == y1_avg) ? filter1_tracker : pixel;
assign pixel_out[1] = (hcount_inverse == {1'b0, x2_avg} || vcount == y2_avg) ? filter2_tracker : pixel_out[0];

//GENERATING 1024x768-TRANSLATED X AND Y AVERAGES (only winning x & y)
wire [9:0] x720, y720, x1024, y1024;
assign x720 = filterWinner ?
(720 - x2_avg) : (720 - x1_avg);
assign y720 = filterWinner ?
y2_avg : y1_avg;

// feed XVGA and AI signals to game
wire [7:0] aPixel;
wire phsync, pvsync, pblank, AIFire;
wire [10:0] AICrosshairX, duckX, x720Prime;
wire [9:0] AICrosshairY, duckY, y720Prime;
wire [1:0] playerShotsLeft, AIShotsLeft, dir;
wire abutton_left, abutton_right, abutton_up, abutton_down, aButton2, aButton3, calibrate, startPlaying;

assign pixel_out[2] = (hcount == x720Prime || vcount == y720Prime) ?
{8'b0, 8'b0, 8'd255} : pixel_out[1];

wire hsPrime, vsPrime, bPrime;
delayN dn1(clk, hsync, hsPrime); // delay by 3+2+1 cycles to sync with ZBT read + YCrCh2RGB + RGB2Hue + NTSC mask
delayN dn2(clk, vsync, vsPrime);
delayN dn3(clk, blank, bPrime);
defparam dn1.NDELAY = 4;
defparam dn2.NDELAY = 4;
defparam dn3.NDELAY = 4;

// debugging
/*wire [26:0] x_sum,
wire [9:0] x_avg;
wire [26:0] new_x_avg;
wire [18:0] pixelCount;*/
assign analyzer1_data = {pixelCount1[18:3]}; // {x_avg, pixelCount[18:16],
  ntsc_we, startFinalAvgDividing, endFinalAvgDividing};
assign analyzer1_clock = clk;
assign analyzer2_data = {x1024, x1_avg[9:4]}; // {x, y[7:2]};
// vram_addr1[15:0];
assign analyzer2_clock = clk;
assign analyzer3_data = {pixelCount2[18:3]}; // {new_x_avg[9:0], y[1:0],
  4'b0}; // {hcount[10:0], vcount[4:0]};
assign analyzer3_clock = clk;
assign analyzer4_data = {x720, x1_avg[3:0], 2'b0}; // {hcount[10:0], 5'b0};
// pixelCount[15:0];
assign analyzer4_clock = clk;

assign x720Prime=(x720>100) ? (((x720<512)? ((2*x720)-200):1024)) : 0;
assign y720Prime=y720;

Game
  aGame(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,AICrosshairX,AICrosshairY,AIFire,x720Prime,y720Prime,filterWinner,~button3,~button2,~button1,
    phsync,pvsync,pblank,aPixel,duckX,duckY,duckAlive,AIShotsLeft,playerShotsLeft,roundOver,roundStart,dir,inCalibrate);

AI
  aAI(clock_65mhz,reset,duckX,duckY,duckAlive,AIShotsLeft,roundStart,roundOver,dir,vsync,switch[7:6],
    AICrosshairX,AICrosshairY,AIFire);

debounce aDebounce(reset,clock_65mhz,button_left,abutton_left);
debounce bDebounce(reset,clock_65mhz,button_right,abutton_right);
debounce cDebounce(reset,clock_65mhz,button_up,abutton_up);
debounce dDebounce(reset,clock_65mhz,button_down,abutton_down);
debounce eDebounce(reset,clock_65mhz,button3,aButton3);
debounce fDebounce(reset,clock_65mhz,button2,aButton2);

reg b,hs,vs;
// playerInput
  aPlayerInput(clock_65mhz,reset,~abutton_left,~abutton_right,~abutton_up,~abutton_down,~button0,playerShotsLeft,playerCrosshairX,playerCrosshairY,playerFire);
always @ (posedge clock_65mhz) begin
  hs <= phsync;
  vs <= pvsync;
  b <= pblank;
  rgb <= aPixel;
end

// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = inCalibrate ? pixel_out[2][23:16]:{rgb[7:5],5'b00000};
assign vga_out_green = inCalibrate ? pixel_out[2][15:8]:{rgb[4:2],5'b00000};
assign vga_out_blue = inCalibrate ? pixel_out[2][7:0]:{rgb[1:0],6'b000000};
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = inCalibrate ? ~b : ~bPrime;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = inCalibrate ? hsPrime : hs;
assign vga_out_vsync = inCalibrate ? vsPrime : vs;
**XVGA Signal Generator**

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

module xvga(vclock, hcount, vcount, hsync, vsync, blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output  vsync;
  output  hsync;
  output  blank;

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

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  wire        hsyncon, hsync0ff, hreset, hblankon;
  assign hblankon = (hcount == 1023);
  assign hsyncon = (hcount == 1047);
  assign hsync0ff = (hcount == 1183);
  assign hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire        vsyncon, vsync0ff, vreset, vblankon;
  assign vblankon = hreset & (vcount == 767);
  assign vsyncon = hreset & (vcount == 776);
  assign vsync0ff = 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 <= hsync0ff ? 0 : hsync0ff ? 1 : hsync;  // active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsync0ff ? 1 : vsync;  // active low

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

**VRAM Display**

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

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

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

wire [18:0] vram_addr;
assign vram_addr = (vcount > 480 || hcount > 720) ?
  19'b0 :
  {vcount[8:0], hcount[9:0]};

wire [1:0] hc4 = hcount[1:0];

assign vr_pixel = vram_read_data[23:0]; //Filtered RGB info!
/*assign vr_pixel[1] = (hueFilterSwitch) ?
  (satpassHigh > satpassLow ?
    vr_pixel[0] :
    hueFilterFailColor) :
  (sat >
    satpassLow && sat < satpassHigh ?
    vr_pixel[0] :
    hueFilterFailColor) :
  satpassLow || sat < satpassHigh ?
    vr_pixel[0] :
    hueFilterFailColor) ;
*/

endmodule // vram_display
module limits_interface(clk, reset, up_button, down_button, left_button, right_button, enter_button, hue1_lowMark, hue1_highMark, hue2_lowMark, hue2_highMark, sat1_lowMark, sat1_highMark, sat2_lowMark, sat2_highMark, vfd_data);

input clk, reset;
input up_button, down_button, left_button, right_button, enter_button;
//raw button lines; assumes 1=pressed, 0=open
output [8:0] hue1_lowMark, hue1_highMark, hue2_lowMark, hue2_highMark;
//Min and max values of the hue-pass, valued from 0 to 360
output [6:0] sat1_lowMark, sat1_highMark, sat2_lowMark, sat2_highMark;
//Min and max values of the sat-pass, valued from 0 to 100 (0 == white, 100 == full color)
output [16*8-1:0] vfd_data; //ascii output to vfd data (ie 16 8-bit bytes, encoded as ASCII)

//First lets syncronize and debounce the buttons
wire up_button_s, down_button_s, left_button_s, right_button_s, enter_button_s;
wire up_button_d, down_button_d, left_button_d, right_button_d, enter_button_d;
synchronize sync1(clk, up_button, up_button_s);
synchronize sync2(clk, down_button, down_button_s);
synchronize sync3(clk, left_button, left_button_s);
synchronize sync4(clk, right_button, right_button_s);
synchronize sync5(clk, enter_button, enter_button_s);
debounce deb1(reset, clk, up_button_s, up_button_d);
debounce deb2(reset, clk, down_button_s, down_button_d);
debounce deb3(reset, clk, left_button_s, left_button_d);
debounce deb4(reset, clk, right_button_s, right_button_d);
debounce deb5(reset, clk, enter_button_s, enter_button_d);

//Wire up the marker fsm's
wire doIndividualMove, doJointMove; //These wires go high for every clock cycle where we move either

//and individual marker (selected by mode fsm) or both markers at once
slow_fast_fsm individual_fsm(reset, clk, (left_button_d ||
right_button_d), doIndividualMove);
    //ie the left or right button control what speed we're moving the individual marker

slow_fast_fsm joint_fsm(reset, clk, (up_button_d || down_button_d),
doJointMove);
    //ie the up or down buttons control what speed we're moving both markers at once

//SELECTION: move between 0:MODE, 1:LOW, 2:HIGH, or 3:BOTH on left/right buttons
parameter SELOBJ_MODE = 0;
parameter SELOBJ_LOW = 1;
parameter SELOBJ_HI  = 2;
parameter SELOBJ_BOTH = 3;
reg [1:0] objectSelection = 0;
reg prev_right=0, prev_left = 0;
wire right_pulse = ~prev_right && right_button;
wire left_pulse  = ~prev_left  && left_button;
always @(posedge clk) begin
    prev_right <= right_button;
    prev_left <=  left_button;
    if (right_pulse)
        objectSelection <= objectSelection + 1;
    else if (left_pulse)
        objectSelection <= objectSelection - 1;
end

//Wire up the limits controller
wire [8:0] hue1_lowMark, hue1_highMark, hue2_lowMark, hue2_highMark;
wire [6:0] sat1_lowMark, sat1_highMark, sat2_lowMark, sat2_highMark;
wire [1:0] modeSelection; //0:HUE1, 1:SAT1, 2:HUE2, 3:SAT2
limits_controller lc(reset, clk,
down_button_d, left_button_d, right_button_d,
doIndividualMove, objectSelection, modeSelection,
hue1_highMark, sat1_lowMark, sat1_highMark,
hue2_highMark, sat2_lowMark, sat2_highMark);

//Wire up the VFD controller
vfd_control vfdc(clk, reset,
objectSelection, modeSelection,
module limits_controller(reset, clk, up_button, down_button, left_button, right_button, doIndepMove, doJointMove, objectSelection, modeSelection, hue1_lowMark, hue1_hiMark, sat1_lowMark, sat1_hiMark, hue2_lowMark, hue2_hiMark, sat2_lowMark, sat2_hiMark);

    input reset, clk, up_button, down_button, left_button, right_button;
    input doIndepMove, doJointMove; // Move the currently selected marker by itself or with its partner (ie high AND low together) -- merely tells which marker-control fsm the button came from

    input [1:0] objectSelection;
    parameter SELOBJ_MODE = 0;
    parameter SELOBJ_LOW  = 1;
    parameter SELOBJ_HI   = 2;
    parameter SELOBJ_BOTH = 3;

    output [1:0] modeSelection;
    reg   [1:0] modeSelection = 0;
    parameter SELMODE_HUE1 = 0;
    parameter SELMODE_SAT1 = 1;
    parameter SELMODE_HUE2 = 2;
    parameter SELMODE_SAT2 = 3;

    output [8:0] hue1_lowMark, hue1_hiMark, hue2_lowMark, hue2_hiMark;
    // These get sent to the VFD display and to the ultimate output of the limits interface
    output [6:0] sat1_lowMark, sat1_hiMark, sat2_lowMark, sat2_hiMark;

    // Keep the individual marks as separate modules
    mark the_hue1_hiMark( reset, clk, hue1_hiMark_incr, hue1_hiMark_decr, hue1_hiMark);
    mark the_hue1_lowMark(reset, clk, hue1_lowMark_incr, hue1_lowMark_decr, hue1_lowMark);
    defparam the_hue1_hiMark.START_VALUE = 35; // Add some arbitrary default values
    defparam the_hue1_lowMark.START_VALUE = 20;

    mark the_hue2_hiMark( reset, clk, hue2_hiMark_incr, hue2_hiMark_decr, hue2_hiMark);
    mark the_hue2_lowMark(reset, clk, hue2_lowMark_incr, hue2_lowMark_decr, hue2_lowMark);
defparam the_hue2_hiMark.START_VALUE = 35;  //Add some arbitrary default values
defparam the_hue2_lowMark.START_VALUE = 20;
wire [1:0] junk[3:0];
mark the_sat1_hiMark(reset, clk, sat1_hiMark_incr, sat1_hiMark_decr, {junk[0], sat1_hiMark});
mark the_sat1_lowMark(reset, clk, sat1_lowMark_incr, sat1_lowMark_decr, {junk[1], sat1_lowMark});
defparam the_sat1_hiMark.MAX_VALUE = 100;
defparam the_sat1_lowMark.MAX_VALUE = 100;
defparam the_sat1_hiMark.START_VALUE = 100;  //Note: 0% saturation == white, 100% saturation == pure color
defparam the_sat1_lowMark.START_VALUE = 20;
mark the_sat2_hiMark(reset, clk, sat2_hiMark_incr, sat2_hiMark_decr, {junk[2], sat2_hiMark});
mark the_sat2_lowMark(reset, clk, sat2_lowMark_incr, sat2_lowMark_decr, {junk[3], sat2_lowMark});
defparam the_sat2_hiMark.MAX_VALUE = 100;
defparam the_sat2_lowMark.MAX_VALUE = 100;
defparam the_sat2_hiMark.START_VALUE = 100;  //Note: 0% saturation == white, 100% saturation == pure color
defparam the_sat2_lowMark.START_VALUE = 20;

//MODE TRANSITIONING: when SELECTION == 0 (selecting MODE), change modeSelection between 0:HUE1, 1:HUE2, 2:SAT1, and 3:SAT2
reg up_button_prev = 0, down_button_prev = 0;
wire up_button_pulse = ~up_button_prev && up_button;
wire down_button_pulse = ~down_button_prev && down_button;
always @(posedge clk) begin
  up_button_prev <= up_button;
down_button_prev <= down_button;
  if (objectSelection == SELOBJ_MODE) begin //ie if changing MODE
    if (up_button_pulse)
      modeSelection <= modeSelection + 1;
    if (down_button_pulse)
      modeSelection <= modeSelection - 1;
  end
end

//DO THE CONTROLLER LOGIC
//hue1
assign hue1_hiMark_incr = (modeSelection == SELMODE_HUE1 &&
//ie, increment the hi-mark when: we're adjusting hue1, AND
(objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
// we're adjusting either the HI mark OR BOTH marks, AND
up_button &&
doJointMove) ? // we pressed the up button AND the movementFSM tells us to go
  1 : 0;
assign hue1_lowMark_incr = (modeSelection == SELMODE_HUE1 &&
  // ie,
  (objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
  // we're adjusting either the LOW mark OR BOTH marks, AND
  up_button &&
  doJointMove) ?
  // we pressed the up button AND the movementFSM
tells us to go
  1 : 0;

assign hue1_hiMark_decr = (modeSelection == SELMODE_HUE1 &&
  (objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
  down_button
  && doJointMove) ?
  1 : 0;

assign hue1_lowMark_decr = (modeSelection == SELMODE_HUE1 &&
  (objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
  down_button
  && doJointMove) ?
  1 : 0;

//hue2
assign hue2_hiMark_incr = (modeSelection == SELMODE_HUE2 &&
  (objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
  up_button &&
  doJointMove) ?
  1 : 0;

assign hue2_lowMark_incr = (modeSelection == SELMODE_HUE2 &&
  (objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
  up_button &&
  doJointMove) ?
  1 : 0;

assign hue2_hiMark_decr = (modeSelection == SELMODE_HUE2 &&
  (objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
  down_button
  && doJointMove) ?
  1 : 0;

assign hue2_lowMark_decr = (modeSelection == SELMODE_HUE2 &&
  (objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
  down_button
  && doJointMove) ?
  1 : 0;
down_button

&& doJointMove) ?

1 : 0;

//saturation1
assign sat1_hiMark_incr = (modeSelection == SELMODE_SAT1 &&
(objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
up_button &&
doJointMove) ?

1 : 0;
assign sat1_lowMark_incr= (modeSelection == SELMODE_SAT1 &&
(objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
up_button &&
doJointMove) ?

1 : 0;
assign sat1_hiMark_decr = (modeSelection == SELMODE_SAT1 &&
(objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
down_button
&& doJointMove) ?

1 : 0;
assign sat1_lowMark_decr= (modeSelection == SELMODE_SAT1 &&
(objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
down_button
&& doJointMove) ?

1 : 0;

//hue2
assign sat2_hiMark_incr = (modeSelection == SELMODE_SAT2 &&
(objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
up_button &&
doJointMove) ?

1 : 0;
assign sat2_lowMark_incr= (modeSelection == SELMODE_SAT2 &&
(objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
up_button &&
doJointMove) ?

1 : 0;
assign sat2_hiMark_decr = (modeSelection == SELMODE_SAT2 &&
  (objectSelection == SELOBJ_HI || objectSelection == SELOBJ_BOTH) &&
  down_button && doJointMove) ?
    1 : 0;

assign sat2_lowMark_decr = (modeSelection == SELMODE_SAT2 &&
  (objectSelection == SELOBJ_LOW || objectSelection == SELOBJ_BOTH) &&
  down_button && doJointMove) ?
    1 : 0;
endmodule

module blinker(clk, onOff);
  input clk;
  output onOff;
  reg onOff = 0;
  parameter MAXCOUNT = 25000000; // 1/2 second blinking
  reg [25:0] count = 0;
  always @(posedge clk) begin
    if (count == MAXCOUNT) begin
      count <= 0;
      onOff <= !onOff;
    end else
      count <= count + 1;
  end
endmodule

module vfd_control(clk, reset,
  objectSelection, modeSelection,
  hue1_low, hue1_high, sat1_low_in,
  sat1_high_in,
  hue2_low, hue2_high, sat2_low_in,
  sat2_high_in,
  vfd_data);
  // Purpose: Output text "HUE LOWxxx HIxxx" or "SAT LOWxxx HIxxx" to VFD
  module, blinking the mode thats appropriate
  input clk, reset;
  input [1:0] objectSelection, modeSelection;
parameter SELOBJ_MODE = 0;
parameter SELOBJ_LOW = 1;
parameter SELOBJ_HI = 2;
parameter SELOBJ_BOTH = 3;
parameter SELMODE_HUE1 = 0;
parameter SELMODE_SAT1 = 1;
parameter SELMODE_HUE2 = 2;
parameter SELMODE_SAT2 = 3;

input [8:0] hue1_low, hue1_high, hue2_low, hue2_high;
input [6:0] sat1_low_in, sat1_high_in, sat2_low_in, sat2_high_in;
wire [8:0] sat1_low = {2'b0, sat1_low_in}, //sign-extend saturation to
           fit division format
           sat1_high = {2'b0, sat1_high_in},
           sat2_low = {2'b0, sat2_low_in},
           sat2_high = {2'b0, sat2_high_in};

output [8*16-1:0] vfd_data;

//Blink generator
wire blink;
blinker bl(clk, blink);

wire [8*4-1:0] textMode[1:0];  //partial string "H1 ",
    "H2 ", "S1 ", "S2 ": to be determined by mode and blinker
wire [8*3-1:0] textLow, textHi;  //partial strings "LOW" and " HI"

assign textMode[0] = (modeSelection == SELMODE_HUE1) ?
    (objectSelection == SELOBJ_MODE ? "H1 " : "h1 ") :
    (modeSelection == SELMODE_SAT1 ?
        (objectSelection == SELOBJ_MODE ? "S1 " : "s1 ") :
        (modeSelection == SELMODE_HUE2 ?
            (objectSelection == SELOBJ_MODE ? "H2 " : "h2 ") :
            (objectSelection == SELOBJ_MODE ? "S2 " : "s2 ")
        )
    )

assign textMode[1] = (objectSelection == SELOBJ_MODE) ?
    (blink ?
        textMode[0] : "    ") :
    textMode[0];

assign textLow = (objectSelection == SELOBJ_LOW || objectSelection ==
    SELOBJ_BOTH) ? /*LOW*/ : "low";
    (blink ? "LOW" : "  "):
    "low";

assign textHi  = (objectSelection == SELOBJ_HI  || objectSelection ==
    SELOBJ_BOTH) ? /* HI*/ : " hi";
    (blink ? " HI" : "  "):
// EXTRACTION OF 100's, 10's, and 1's DIGITS
// ***ASSUMES max number will be 360***
wire [8:0] high, low; //hue high-mark and low-mark or sat high-mark and
low-mark, depending on the mode
assign high = modeSelection == SELMODE_HUE1 ?
    hue1_high :
    (modeSelection == SELMODE_SAT1 ?
        sat1_high :
        (modeSelection == SELMODE_HUE2 ?
            hue2_high :
            sat2_high));
assign low = modeSelection == SELMODE_HUE1 ?
    hue1_low :
    (modeSelection == SELMODE_SAT1 ?
        sat1_low :
        (modeSelection ==
            SELMODE_HUE2 ?
                hue2_low :
                sat2_low));
reg [8:0] dividend;
reg [6:0] divisor; //will be either 100 or 10
wire [8:0] quot;
wire [6:0] remainder;
reg [6:0] lastRemainder;
wire ready;
divider3 div9x7(dividend, divisor, quot, remainder, clk, ready, 1'b0,
1'b0, 1'b0); //Divider with dividend size 9 (511-0), divisor size 7 (127-0),
delay of 9+2 + 1=12 clock cycles
reg [7:0] highs_hundreds, lows_hundreds;
reg [7:0] highs_tens, highs_ones, lows_tens, lows_ones;
reg [1:0] hi_low = 0; //multiplex hi-val and low-val division
always @(posedge clk) begin
    lastRemainder <= remainder;
    hi_low <= hi_low + 1;
    /* Divider has 12-cycle pipeline (ie get answer out 12
    cycles later)*/
    when =0, send in hi / 100,
    get out hi/100
    from 12 cycles ago
    -1, send in rem(hi/100) / 10, get
out rem(hi/100) /10 from 12 cycles ago
    -2, send in low / 100,
    get out low/100
    from 12 cycles ago
    -3, send in rem(low/100)/ 10,
    get out rem(low/100)/10 from 12 cycles ago
*/
case (hi_low)
    0: begin
        dividend <= high;
        divisor <= 100;
highs_hundreds <= quot + 48;
end

1: begin
  dividend <= lastRemainder; // remainder (hi/100)
divisor <= 10;
highs_tens <= quot + 48;
highs_ones <= remainder + 48;
end

2: begin
  dividend <= low;
divisor <= 100;
lowsHundreds <= quot + 48;
end

3: begin
  dividend <= lastRemainder;
divisor <= 10;
lows_tens <= quot + 48;
lows_ones <= remainder + 48;
end

endcase
end

wire [16*8-1:0] vfd_data = {textMode[1], textLow,
lows_hundreds, lows_tens, lows_ones,
textHi,
highs_hundreds, highs_tens, highs_ones };

/*
({6'b0, lows_hundreds} + 48), //+48 to get the number into the ASCII code
(ie ASCII "0" == 0d48)
({4'b0, lows_tens} + 48),     // hopefully these all come out to be 8-
bits wide
({4'b0, lows_ones} + 48),
textHi,
({6'b0, highs_hundreds} + 48), // +48 to get the number into the ASCII
code (ie ASCII "0" == 0d48)
({4'b0, highs_tens} + 48),     // hopefully these all come out to
be 8-bits wide
({4'b0, highs_ones} + 48) };*/
endmodule

module mark(reset, clk, increment, decrement, value);
//This module just keeps track of incrementing/decrementing a mark-value and doing the wrap around.
//Mark value is 8-bits [0 and 360]

parameter MIN_VALUE = 0;
parameter MAX_VALUE = 360;

parameter START_VALUE = 5;

input reset, clk, increment, decrement; //We do +/- 1 each clock cycle that increment and decrement are high (increment takes priority)
output [8:0] value;

reg [8:0] value = START_VALUE;

always @ (posedge clk) begin
  if (reset)
    value <= START_VALUE;
  else begin
    if (increment)
      if (value == MAX_VALUE)
        value <= MIN_VALUE;
    else
      value <= value + 1;
    else if (decrement)
      if (value == MIN_VALUE)
        value <= MAX_VALUE;
    else
      value <= value - 1;
  end
end
endmodule

/*module selection_fsm(reset, next_button, prev_button, mode_select);

  input reset, next_button, prev_button;
  output [1:0] mode_select; //Says which parameter we’re controlling...
0=hue, 1=low mark, 2=hi mark

  parameter MODE_HUE = 0;
  parameter MODE_LOWMARK = 1;
  parameter MODE_HIMARK = 2;

  reg [1:0] mode_select = MODE_HUE;

  always @ (posedge enter_button) begin //move only on posedge of the enter_button
    if (reset)
      mode_select <= MODE_HUE;
    else begin
      if (mode_select == MODE_HIMARK)
        mode_select <= 0;
      if (mode_select == MODE_LOWMARK)
        mode_select <= 0;
      if (mode_select == MODE_HUE)
        mode_select <= 1;
    end
  end
endmodule /*
module slow_fast_fsm(reset, clk, button, doMovePulse);

    input reset, clk, button;  //button is the slow/fast button of interest
    output doMovePulse;  //High for every clock cycle we want to increment the
    value of interest

    parameter SLOW_LENGTH = 130000000;  //Num clock cycles to stay in
    slow mode while button is depressed (max 256)
    parameter SLOW_TICK = 30000000;  //Num clock cycles between slow
doMovePulses, should be a factor of SLOW_INTERVAL (max 256)
    parameter FAST_TICK = 01000000;  //Num clock cycles between fast
doMovePulses (max 256)

    reg [26:0] count = 0, tick = 0;
    reg mode=0, doMovePulse;

    always @ (posedge clk) begin
        if (reset) begin
            count <= 0;
            mode <= 0;
        end else begin
            if (mode == 0) begin //SLOW MODE
                if (button) begin
                    if (count < SLOW_LENGTH) begin
                        count <= count + 1;
                    
                    if (tick >= SLOW_TICK)
                        tick <= 0;
                    else
                        tick <= tick + 1;
                    
                    if (tick == 0)
                        doMovePulse <= 1;
                    else begin
                        doMovePulse <= 0;
                    end
                end else begin //ie, if no button is pressed
                    mode <= 1; //Switch to fast mode
                end
            end else begin //ie, if no button is pressed
                doMovePulse <= 0;
            end
        end
    end
*/
module slow_mode(clk, vclk, fvh, dv, din, count, tick, button, mode);

begin

  if (mode == SLOW_MODE)
    begin //SLOW MODE
      count <= 0;
      tick <= 0;
    end

  end //end slow mode

else begin //FAST MODE

  if (button) begin
    if (count >= FAST_TICK) begin
      count <= 0;
      doMovePulse <= 1;
    end else begin
      doMovePulse <= 0;
      count <= count + 1;
    end

  end else begin //ie, no button pressed
    doMovePulse <= 0;
    count <= 0;
    mode <= 0; //back to SLOW MODE;

  end

end
endmodule

---

**NTSC TO ZBT And Filtering**

//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we,
  filterFail_bw_switch,
  filterFail_color_switch,
  hue2_passLow, hue2_passHigh,
  sat2_passLow, sat2_passHigh,
  sat1FilterSwitch, hue2FilterSwitch, sat2FilterSwitch,
  weighted_y1_avg, weighted_y2_avg, winningFilter,
  weighted_x1_avg, weighted_x2_avg,
  noiseFilterSwitch,
  pixelCount1, pixelCount2);

//DEBUGS:
//x_sum, new_x_avg, pixelCount,
//startFinalAvgDividing, endFinalAvgDividing, x_out, y_out);

input clk; // system clock
input vclk; // video clock from camera
input [2:0] fvh;
input dv;
input [29:0] din; // full ycrcb data
output [18:0] ntsc_addr;
output [35:0] ntsc_data; //in RGB!!
output ntsc_we; // write enable for NTSC data
output [9:0] weighted_x1_avg, weighted_x2_avg; //x-average of all passing pixels in filter1
output [9:0] weighted_y1_avg, weighted_y2_avg; //...in filter2
output winningFilter; // (Debounced) Whether there is more in filter1 (==0) or filter2 (==1)
output [18:0] pixelCount1, pixelCount2;

input filterFail_bw_switch, filterFail_color_switch;
input [8:0] hue1_passLow, hue1_passHigh, hue2_passLow, hue2_passHigh;
input [6:0] sat1_passLow, sat1_passHigh, sat2_passLow, sat2_passHigh;
input hue1FilterSwitch, hue2FilterSwitch;
input sat1FilterSwitch, sat2FilterSwitch;
input noiseFilterSwitch;
input filterDebounceSw;

//DEBUGS:
//output [26:0] x_sum;
//output [26:0] new_x_avg;
output [18:0] pixelCount;
output startFinalAvgDividing, endFinalAvgDividing;
output [9:0] x_out;
output [7:0] y_out; /*
wire [23:0] vr_pixel; //This is the output pixel line.
wire [23:0] vr_pixel_hue1pass, vr_pixel_hue2pass;
wire [23:0] vr_pixel_sat1pass, vr_pixel_sat2pass;
reg [9:0] col = 0;
reg [7:0] row = 0;
reg [29:0] 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;

/**********************/
// START COLOR-SPACE CONVERTERS
// all color space converters use 65mhz clock
// YCrCb->RGB Color Space Converter (with Y Cr Cb as 10-bit inputs, R G B as 8-bit outputs, 3 clock cycle delay */
wire [9:0] Ylum, Cr, Cb;
wires [7:0] R, G, B;
YCrCb2RGB ycrcb_csc( R, G, B, clk, reset, Ylum, Cr, Cb );

//RGB->Hue COLOR SPACE CONVERTER (24 cycle delay)
wire [8:0] hue; //between 0 and 360
wire [6:0] sat;
RGB2Hue rgb_csc(clk, reset, R, G, B, hue, sat);

//Still want to use RGB later, so need to delay it by 24 cycles to match it with the hue output
wire [7:0] Rd, Gd, Bd;
parameter RGB_delay = 24;
delayNxM Rder(clk, R, Rd);
delayNxM Gder(clk, G, Gd);
delayNxM Bder(clk, B, Bd);
defparam Rder.MSIZE = 8;
defparam Rder.dn[7].NDELAY = RGB_delay;
defparam Rder.dn[6].NDELAY = RGB_delay;
defparam Rder.dn[5].NDELAY = RGB_delay;
defparam Rder.dn[4].NDELAY = RGB_delay;
defparam Rder.dn[3].NDELAY = RGB_delay;
defparam Rder.dn[2].NDELAY = RGB_delay;
defparam Rder.dn[1].NDELAY = RGB_delay;
defparam Rder.dn[0].NDELAY = RGB_delay;
defparam Gder.MSIZE = 8;
defparam Gder.dn[7].NDELAY = RGB_delay;
defparam Gder.dn[6].NDELAY = RGB_delay;
defparam Gder.dn[5].NDELAY = RGB_delay;
defparam Gder.dn[4].NDELAY = RGB_delay;
defparam Gder.dn[3].NDELAY = RGB_delay;
defparam Gder.dn[2].NDELAY = RGB_delay;
defparam Gder.dn[1].NDELAY = RGB_delay;
defparam Gder.dn[0].NDELAY = RGB_delay;
defparam Bder.MSIZE = 8;
defparam Bder.dn[7].NDELAY = RGB_delay;
defparam Bder.dn[6].NDELAY = RGB_delay;
defparam Bder.dn[5].NDELAY = RGB_delay;
defparam Bder.dn[4].NDELAY = RGB_delay;
defparam Bder.dn[3].NDELAY = RGB_delay;
defparam Bder.dn[2].NDELAY = RGB_delay;
defparam Bder.dn[1].NDELAY = RGB_delay;
defparam Bder.dn[0].NDELAY = RGB_delay;

//If displaying just B&W data, want to delay it by 27 cycles so it still stays sync'd with RGB mode
wire [7:0] BW_delayed;
parameter BW_delay = 27; //3 for RGB + 24 for hue
delayNxM BWder(clk, Ylum, BW_delayed);
defparam BWder.MSIZE = 8;
defparam BWder.dn[7].NDELAY = BW_delay;
defparam BWder.dn[6].NDELAY = BW_delay;
defparam BWder.dn[5].NDELAY = BW_delay;
defparam BWder.dn[4].NDELAY = BW_delay;
defparam BWder.dn[3].NDELAY = BW_delay;
defparam BWder.dn[2].NDELAY = BW_delay;
defparam BWder.dn[1].NDELAY = BW_delay;
defparam BWder.dn[0].NDELAY = BW_delay;
defparam BWder.dn[0].NDELAY = BW_delay;

//END COLOR-SPACE CONVERTERS

//Address Delay -- Synchronize generated address with the results of hue and sat filtering
wire [18:0] myaddr;
wire [18:0] ntsc_addr;
parameter addr_delay = BW_delay; //3 for RGB + 24 for hue
delayNxM addr_der(clk, myaddr, ntsc_addr);
defparam addr_der.MSIZE = 19;
defparam addr_der.dn[18].NDELAY = addr_delay;
defparam addr_der.dn[17].NDELAY = addr_delay;
defparam addr_der.dn[16].NDELAY = addr_delay;
defparam addr_der.dn[15].NDELAY = addr_delay;
defparam addr_der.dn[14].NDELAY = addr_delay;
defparam addr_der.dn[13].NDELAY = addr_delay;
defparam addr_der.dn[12].NDELAY = addr_delay;
defparam addr_der.dn[11].NDELAY = addr_delay;
defparam addr_der.dn[10].NDELAY = addr_delay;
defparam addr_der.dn[9].NDELAY = addr_delay;
defparam addr_der.dn[8].NDELAY = addr_delay;
defparam addr_der.dn[7].NDELAY = addr_delay;
defparam addr_der.dn[6].NDELAY = addr_delay;
defparam addr_der.dn[5].NDELAY = addr_delay;
defparam addr_der.dn[4].NDELAY = addr_delay;
defparam addr_der.dn[3].NDELAY = addr_delay;
defparam addr_der.dn[2].NDELAY = addr_delay;
defparam addr_der.dn[1].NDELAY = addr_delay;
defparam addr_der.dn[0].NDELAY = addr_delay;


///////////////////////////////////////
//VIDEO READING STUFF
always @ (posedge vclk) //LLC1 is reference
begin
old_dv <= dv;
vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
old_frame <= frame;
even_odd = frame_edge ? ~even_odd : even_odd;
if (!fvh[2])
begin
col <= fvh[0] ? 0 : //on a horizontal-sync, reset column
row <= fvh[1] ? 0 : //on a vertical-sync, reset row
//should effectively go up to 480/2 (interlacing)
vdata <= (dv && !fvh[2]) ? din : vdata;
end
end

// synchronize with system clock (above was sync'd with tv_in_line_clock1)
reg [9:0] x[1:0]; //two-register synchronizers
reg [7:0] y[1:0]; //''
reg [29:0] data[1:0]; //''
reg     we[1:0];    //'
reg     eo[1:0];    //'
always @(posedge clk)
begin
{x[1],x[0]} <= {x[0],col};
{y[1],y[0]} <= {y[0],row};
{data[1],data[0]} <= {data[0],vdata};
{we[1],we[0]} <= {we[0],vwe};
{eo[1],eo[0]} <= {eo[0],even_odd};
end

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

//STORAGE ADDRESS COMPUTATION
//Address to store in ZBT-ram is basically the y-location plus the x-location
assign myaddr = {y[1][7:0], eo[1], x[1][9:0]};
//wire ntsc_addr (output) is my_addr delayed by 3+24 @ 65mhz (ie its the
myaddr of the output of the hue and sat filters)
delayN we_der(clk, we_edge, ntsc_we);
defparam we_der.NDELAY = addr_delay; //Delay the generated we by 3+24 to
syncronize it with outputs of filters

//COLOR SPACE CONVERSION
// YCrCb->RGB Color Space Converter (with Y Cr Cb as 10-bit inputs, R G B
as 8-bit outputs, 3 clock cycle delay */
assign Ylum =  data[1][29:20]; //luminance data
assign Cr = data[1][19:10];  //Cr data
assign Cb = data[1][ 9: 0];  //Cb data

/*/WIRE LIST AT THIS POINT:
--------------
R, G, B are Ylum, Cr, and Cb converted and delayed by 3  @ 65mhz
hue and sat are R, G, B converted and delayed by 24  @
65mhz
Rd, Gd, Bd are R, G, B
delayed by 24  @ 65mh
BW_delayed is Ylum
delayed by 3+24 @ 65mhz */
/*OUTPUT PIXEL GENERATION:*
Output to ZBT ram will be in RGB format.
To create output pixel, we want to send it first through two filters.

First we send it through a hue-pass filter. If the pixel's hue is inside the specified range of allowable hues, we pass the RGB information on to the next filter.

The next filter is a saturation-pass filter. If the pixel's saturation is inside the specified range of allowable saturations, we pass the RGB information out to the memory.

If the pixel fails any of these filters, we pass through a "fail-color" to be written to ZBT memory for that particular location. 

/*Fail-Color:
If we fail the hue and sat limits filter, we have 3 options:
if bw_switch: display failed pixels as b&w
if colorFail_switch: display failed pixels as pure blue
if neither: display failed pixels as black */

wire [24:0] hueFilterFailColor = filterFail_bw_switch ?
{BW_delayed, BW_delayed, BW_delayed} :
(filterFail_color_switch ? {8'b0, 8'b0, 8'd255} : 24'b0);
wire pixel_failed_huesatFilter1, pixel_failed_huesatFilter2;
hueSatFiltering huesat1_filter( hue1FilterSwitch, sat1FilterSwitch, hue, sat,
hue1_passHigh, hue1_passLow, sat1_passHigh, sat1_passLow,
hue1_Failed, sat1_Failed, pixel_failed_huesatFilter1);
hueSatFiltering huesat2_filter( hue2FilterSwitch, sat2FilterSwitch, hue, sat,
hue2_passHigh, hue2_passLow, sat2_passHigh, sat2_passLow,
hue2_Failed, sat2_Failed, pixel_failed_huesatFilter2);
wire pixel_failed_huesat = pixel_failed_huesatFilter1 &&
pixel_failed_huesatFilter2;  //Did the pixel fail both enabled filters (ie passes either filter)?

/*NOISE FILTER:
For a pixel to be displayed and count, the last number_of_past_passing_pixels pixels have to have passed the hue and sat filters*/
reg [6:0] past_passingFilter1_pixel_buffer = 0;
reg [6:0] past_passingFilter2_pixel_buffer = 0;
parameter LAST_7_PIXELS = 7’b1111111;

always @ (posedge clk) begin
    if (x[1] < 2) begin  //Reset pixel buffer at the start of a row
        past_passingFilter1_pixel_buffer <= 0;
        past_passingFilter2_pixel_buffer <= 0;
    end else begin
        past_passingFilter1_pixel_buffer <=
        (past_passingFilter1_pixel_buffer[5:0], ¬pixel_failed_huesatFilter1);
        past_passingFilter2_pixel_buffer <=
        (past_passingFilter2_pixel_buffer[5:0], ¬pixel_failed_huesatFilter2);
    end

    wire pixel_passed_noiseFilter1, pixel_passed_noiseFilter2;
    assign pixel_passed_noiseFilter1 = past_passingFilter1_pixel_buffer[6:0] == LAST_7_PIXELS;
    assign pixel_passed_noiseFilter2 = past_passingFilter2_pixel_buffer[6:0] == LAST_7_PIXELS;

    //Output Pixel:
    wire pixelPassedFilters1 = noiseFilterSwitch ?
        pixel_passed_noiseFilter1 : ¬pixel_failed_huesatFilter1;
    wire pixelPassedFilters2 = noiseFilterSwitch ?
        pixel_passed_noiseFilter2 :
        ¬pixel_failed_huesatFilter2;
    //Note: If filter1 (ie hue1 and sat1) or filter2 (ie hue2 and sat2) is disabled,
    // the pixel "fails" the appropriate filter
    assign vr_pixel = (pixelPassedFilters1 || pixelPassedFilters2) ? //If the pixel passes either noise filter...
        (Rd,Gd,Bd) :
        //display it as rgb
        hueFilterFailColor;
    //Else, kill it

    wire [35:0] ntsc_data;
    assign ntsc_data = {12’b0, vr_pixel};

    //CENTROID CALCULATOR
wire [9:0] x1_avg, x2_avg, y1_avg, y2_avg;
wire [18:0] pixelCount1, pixelCount2;

centroid_calculator filter1_centroid(clk, x[1], y[1], eo[1], ntsc_we,
~pixelPassedFilters1, x1_avg, y1_avg, pixelCount1);
centroid_calculator filter2_centroid(clk, x[1], y[1], eo[1], ntsc_we,
~pixelPassedFilters2, x2_avg, y2_avg, pixelCount2);

weighted_averager w_avger1(clk, x1_avg, (x[1]==0 && y[1]==0),
weighted_x1_avg);
weighted_averager w_avger2(clk, x2_avg, (x[1]==0 && y[1]==0),
weighted_x2_avg);
weighted_averager w_avger3(clk, y1_avg, (x[1]==0 && y[1]==0),
weighted_y1_avg);
weighted_averager w_avger4(clk, y2_avg, (x[1]==0 && y[1]==0),
weighted_y2_avg);

//OUTPUT DEBOUNCING
wire noisyWinner; //The filter that has most pixels detected at the
moment (0 == filter1, 1==filter2)
assign noisyWinner = (pixelCount2 > pixelCount1) ? 1:0;

reg newWinner = 0; //Which filter we are suspecting to be the winner (0
== filter1, 1==filter2)
reg cleanFilterWinner = 0; //Debounced winning filter
reg [4:0] debounceCount=0; //Number of NTSC frames the current winning
filter has been the winning filter
parameter FILTER_DEBOUNCE_TIME = 5; //Number of NTSC frames to a filter
needs to be winning before its considered the winner (note: 1 frame = 37ms)

always @(posedge clk) begin
    if (x[1] == 0 && y[1] == 0) begin //ie check every new frame
        if (noisyWinner != newWinner) begin
            newWinner <= noisyWinner;
            debounceCount <= 0;
        end else if (debounceCount == FILTER_DEBOUNCE_TIME)
            cleanFilterWinner <= newWinner;
        else
            debounceCount <= debounceCount+1;
    end
end

assign winningFilter = filterDebounceSw ?
    cleanFilterWinner :
    noisyWinner;

endmodule // ntsc_to_zbt

module hueSatFiltering(hueFilterSwitch, satFilterSwitch,
    hue, sat,
    huepassHigh, huepassLow,
    satpassHigh, satpassLow,
    hueFailed, satFailed,
    pixelFailed);

    //Hue/Sat-Pass Filter:
hue1_passLow, hue1_passHigh, hue2_passLow, hue2_passHigh,
sat1_passLow, sat1_passHigh, sat2_passLow, sat2_passHigh,

If huepassHigh > huepassLow, pass any pixel between low and high
If huepassHigh < huepassLow, pass any pixel not between low and high */

input hueFilterSwitch, satFilterSwitch;
input [8:0] hue, huepassHigh, huepassLow;
input [6:0] sat, satpassHigh, satpassLow;
output hueFailed, satFailed, pixelFailed;

assign hueFailed = (hueFilterSwitch) ? (huepassHigh > huepassLow

&& hue < huepassHigh ? 0 : 1) : (hue > huepassLow

|| hue < huepassHigh ? 0 : 1) ;

assign satFailed = (satFilterSwitch) ? (satpassHigh >

satpassLow ? (sat >
satpassLow && sat < satpassHigh ? 0 : 1) : (sat >
satpassLow || sat < satpassHigh ? 0 : 1) ) ;

/*For pixelFailed, we want:
If both filters are enabled, fail (=1) if it fails either filter.
If one filter is enabled, fail (=1) if it fails just that
filter
If both filters are disabled, fail (=1) always. */
assign pixelFailed = (hueFilterSwitch || satFilterSwitch) ? ((hueFilterSwitch

&& hueFailed) || (satFilterSwitch && satFailed)) :

1;

//Thus if both hue and sat filters are disabled, the pixel "fails" the filter.

/*(hueFilterSwitch &&
satFilterSwitch) ?

(hueFailed ||
satFailed) :

(hueFilterSwitch ?

hueFailed :

(satFilterSwitch ?

satFailed) :

1) ;

assign pixelFailed = (hueFilterSwitch || satFilterSwitch) ? ((hueFilterSwitch

&& hueFailed) || (satFilterSwitch && satFailed)) :

1;
module centroid_calculator(clk, x, y, eo, ntsc_we, pixelFailed, x_avg, y_avg, pixelCount_out);

    input    clk;
    input [9:0]  x;
    input [7:0]  y;
    input    eo;
    input    ntsc_we;
    input    pixelFailed;
    output [9:0] x_avg, y_avg;
    output [18:0] pixelCount_out;

    //Averages/Centroid Calculation
    Centroid calculation is done by simply averaging all passing pixels. But what size should
    the sum register be?
    Worst-case scenario: every pixel in a row passes. The max value
    of the row's x-sum
    is then:
    $720/2 \times 720 + 720/2 = 259,560$ (proving this is left as an
    excercise to the reader).
    Assuming every pixel in every row passes, the max value of the
    total x-sum is:
    $259,560 \times 480 = 124,588,800$
    $2^{27}$ is: $134,217,128$
    so let's make the x-sum variable 27-bits long.

    What size should the count register be? Worst case scenario is:
    $480 \times 720 = 345,600$
    $2^{19}$ is: $524,288$

    /*
    reg [26:0] x_sum = 0, y_sum = 0;
    wire [26:0] new_x_sum, new_y_sum;
    wire [9:0] new_pixel_x, new_pixel_y;
    reg latch_x_sum, latch_y_sum;

    reg [9:0] x_avg = 0, y_avg = 0;
    reg [18:0] pixelCount = 0;
    reg [18:0] pixelCount_out = 0;

    xy_adder2 x_summer(x_sum, new_pixel_x, new_x_sum, clk);
    xy_adder2 y_summer(y_sum, new_pixel_y, new_y_sum, clk);

    wire [26:0] new_x_avg, new_y_avg;
    wire startFinalAvgDividing, endFinalAvgDividing;
    delayN validAvgDivision(clk, startFinalAvgDividing, endFinalAvgDividing);
    defparam validAvgDivision.NDELAY = 30; //Divider's delay is 29 clocks.
    start is high when the final values go in, so end will be high when final avg
    comes out

    xyavg_divider x_avger(new_x_sum, pixelCount, new_x_avg, junk1, clk,
    junk2, junk3, junk4, junk5);
assign new_pixel_x = pixelFailed ? 0 : x; //Add the current pixel's x-value to the sum only when it passes the filters
assign new_pixel_y = pixelFailed ? 0 : {1'b0, y, eo};

always @ (posedge clk) begin
    latch_x_sum <= ntsc_we; //ntsc_we is valid when the output pixel and address info is correct.
    //Since the adder is adding passing pixels and has a latency of 1, the cycle after ntsc_we is high is when we have a new valid sum.
    if (x < 1 && y < 1) begin
        //If we're starting a new frame, reset all the averages
        x_avg <= 0;
        y_avg <= 0;
        x_sum <= 0;
        y_sum <= 0;
        pixelCount <= 0;
    end else begin
        //Update sums
        if (latch_x_sum) begin
            x_sum <= new_x_sum;
            y_sum <= new_y_sum;
        end
        //Update the pixel count
        if (ntsc_we && !pixelFailed)
            pixelCount <= pixelCount + 1;
        if (startFinalAvgDividing)
            pixelCount_out <= pixelCount;
    end
    //Latch the output average to the calculated average from the (pipelined) divider
    else if (endFinalAvgDividing) begin //Output of divider right now is the final x_avg
        x_avg <= new_x_avg[9:0];
        y_avg <= new_y_avg[9:0];
    end
    end
endmodule
if \( x_4 \) is oldest input and \( x_0 \) is current input, output is basically
\[
x_0 + x_1 + x_2 + x_3 + x_4 \ldots
\]
\[
/2 \hspace{2em} /4 \hspace{2em} /6 \hspace{2em} /8 \hspace{2em} /10 \hspace{2em} */
\]

```
input clk;
input [10:0] new_input;
input new_data_enable;
output [10:0] prev_average;

reg [11:0] prev_average = 0;
reg [10:0] new_input_latched = 0;
wire [10:0] weighted_average = prev_average[10:0];
wire [11:0] new_weighted_average;
weighted_average_divider w_avg_diver(prev_average + new_input_latched,
3'd2, new_weighted_average,
    junk1, clk, junk2, junk3, junk4, junk5);

always @(posedge clk) begin
    if (new_data_enable) begin
        new_input_latched <= new_input;
        prev_average <= new_weighted_average;
    end
end
endmodule
```

**RGB To HSV Color Converter**

```
module delayNxM(clk,in,out);

    //parameter NDELAY = 3; //Number of cycles to delay, min=2
    parameter MSIZE = 1; //Number of bits to delay (ie 2 == input is [1:0]),
    min=1

    input clk;
    input [MSIZE-1:0] in;
    output [MSIZE-1:0] out;

    delayN dn[MSIZE-1:0] ( {MSIZE{clk}}, in, out);
    //defparam dn.NDELAY = NDELAY;

endmodule // delayN
```

```
module RGB2Hue(clk, reset, Rin, Gin, Bin, hueOut, satBuffered);

    input [7:0] Rin, Gin, Bin;
    input clk,reset;

    wire [8:0] R, G, B; //We need to do arithmitic with signed variables,
    so here
    assign R = {1'b0, Rin}; //we're just sign-extending RGB
    assign G = {1'b0, Gin};
    assign B = {1'b0, Bin};
```
output [8:0] hueOut; //hue will be value between 0 and 360
output [6:0] satBuffered; //saturation will be value between 0 and 100

//HUE PIPELINE REGISTERS AND WIRES
///////////

//STAGE 1:
reg signed [8:0] Rd=0, Gd=0, Bd=0;  //signed RGB delayed by 1
reg signed [8:0] max=0, min=0;  //Max and min RGB values
reg [2:0] mode = 0; //notes which value was the maximum
parameter MAXeqMIN = 0;
parameter MAXisR_GgB = 1;
parameter MAXisR_GlB = 2;
parameter MAXisG = 3;
parameter MAXisB = 4;
wire [8:0] num_a, num_b; //These go into numerator subtractor. Combinational logic selects which RGB values go into these.

//STAGE 2:
wire [8:0] denominator; //Denom[0] is used in stage 2, then delayed 2 cycles while the numerator is being multiplied
wire [8:0] numerator;
wire [8:0] denominatorD;

//STAGE 3:
wire [2:0] modeD; //mode is used in stage 1, then gets delayed 22 cycles to middle of stage 3
wire [14:0] numeratorX60; //Numerator from STAGE 2 multiplied by 6
wire hueUndefined;

//STAGE 4:
wire [14:0] quotient;
wire [8:0] addTerm;

//STAGE 5:
wire [16:0] hue; //Hue before checking for undefined (gray) condition
wire [8:0] hueOut;
reg hueUndefinedD; //hueUndefined delayed by 2 cycles

//STAGE 6:
reg [8:0] hueBuffered = 0; //adder takes too long to be accurate. Make the output a buffered register.

//SAT PIPELINE REGISTERS AND WIRES
///////////

wire [14:0] minX100;
wire [7:0] maxD;
wire [14:0] satDivisionResult;
wire [6:0] satSubtractionResult;
wire [6:0] satChecked;
wire [6:0] satOut;
reg [6:0] satBuffered;

//PIPELINE DELAYS
///////////////
//HUE
////////
//\delayN d1[2:0];
delayNxM d1(clk, mode, modeD); //Which formula we're using (to select which number we add in stage 3
  defparam d1.MSIZE = 3;
  defparam d1.dn[2].NDELAY = 22;
  defparam d1.dn[1].NDELAY = 22;
  defparam d1.dn[0].NDELAY = 22;

delayNxM d2(clk, denominator, denominatorD);
  defparam d2.MSIZE = 9;
  defparam d2.dn[8].NDELAY = 2;
  defparam d2.dn[7].NDELAY = 2;
  defparam d2.dn[6].NDELAY = 2;
  defparam d2.dn[5].NDELAY = 2;
  defparam d2.dn[4].NDELAY = 2;
  defparam d2.dn[3].NDELAY = 2;
  defparam d2.dn[2].NDELAY = 2;
  defparam d2.dn[1].NDELAY = 2;
  defparam d2.dn[0].NDELAY = 2;

//\delayN d3(clk, hueUndefined, hueUndefinedD);
//defparam d3.NDELAY = 1;
//Note: can't delay by 1 using NDELAY. So just doing it in sequential area.

//SAT
////////
delayNxM d4(clk, max[7:0], maxD);
  defparam d4.MSIZE = 8;
  defparam d4.dn[7].NDELAY = 2;
  defparam d4.dn[6].NDELAY = 2;
  defparam d4.dn[5].NDELAY = 2;
  defparam d4.dn[4].NDELAY = 2;
  defparam d4.dn[3].NDELAY = 2;
  defparam d4.dn[2].NDELAY = 2;
  defparam d4.dn[1].NDELAY = 2;
  defparam d4.dn[0].NDELAY = 2;

delayNxM d5(clk, satChecked, satOut);
  defparam d5.MSIZE = 7;
  defparam d5.dn[6].NDELAY = 3;
  defparam d5.dn[5].NDELAY = 3;
  defparam d5.dn[4].NDELAY = 3;
  defparam d5.dn[3].NDELAY = 3;
  defparam d5.dn[2].NDELAY = 3;
  defparam d5.dn[1].NDELAY = 3;
  defparam d5.dn[0].NDELAY = 3;

//ANY COMBINATIONAL LOGIC
/////////////////////////
assign num_a = (mode == MAXeqMIN ? 8'b0 :
  (mode == MAXisR_GgB || mode == MAXisR_GlB ?
    Gd :
    (mode == MAXisG ?

MAXisR_GlB ?
```verilog
assign num_b = (mode == MAXeqMIN ? 8'b0 : (mode == MAXisR_GgB || mode == MAXisR_GlB ? Bd : (mode == MAXisG ? Rd : Gd) ) );

assign addTerm = (modeD == MAXisR_GlB ? 9'd360 : (modeD == MAXisG ? 9'd120 : (modeD == MAXisB ? 9'd240 : 9'b0) ) );

assign hueUndefined = (modeD == MAXeqMIN);
assign hueOut = hueUndefinedD ? //If the hue is undefined, we want to ultimately output 0.
9'b0 : hue[8:0];  //Otherwise, we want to output the low 8 bits of the adder

assign satChecked = (satSubtractionResult > 100) ? //Rounding bugs in subtractor sometimes give us values more than 100. Don’t let that.
7'd100 :
satSubtractionResult;

//WIREF UP ALL THE MATH-ERS
 harassed
//HUE
hsv_subtractor sub1(max, min, denominator, clk); //Denominator <= max-min
hsv_subtractor sub2(num_a, num_b, numerator, clk);

hsv_multiplier mult1(clk, numerator, numeratorX60);

hsv_divider div1(numeratorX60, denominatorD, quotient, junk1, clk, junk2, junk3, junk4, junk5);

hsv_adder add1(quotient, addTerm, hue, clk);

//SAT
hsv_s_multiplier mult2(clk, min[7:0], minX100); //min[7:0] b/c min was designed to be signed, so it's actually 8. Its always positive, though, so drop the sign

hsv_s_divider div2(minX100, maxD, satDivisionResult, junk6, clk, junk7, junk8, junk9, junk10);

wire [6:0] hsv_subtractor_a = 7'd100;
```
hsv_s_subtractor  sub3(hsv_subtractor_a, satDivisionResult[6:0], satSubtractionResult, clk); //We set it up so division result is no more than 100, so use only [6:0]

always @(posedge clk) begin

FLICTC

//STAGE 1: MIN/MAX SELECTOR and RGB DELAY
//Length: 1 clock cycle

//First step: sort out the min and max values
if (R >= G && R >= B) begin
    max   <= R;
    if (G > B) begin
        min  <= B;
        if (R == B)
            mode  <= MAXeqMIN;
        else
            mode  <= MAXisR_GgB; //...and green is greater than blue
    end
    else begin
        min  <= G;
        if (R == G)
            mode  <= MAXeqMIN;
        else
            mode  <= MAXisR_Glb; //...and green is less than blue
    end

//Green is max
end else if (G > R && G > B) begin
    mode  <= MAXisG;
    max   <= G;
    if (R > B)
        min  <= B;
    else
        min  <= R;

//Blue is max
end else begin
    mode  <= MAXisB;
    max   <= B;
    if (R > G)
        min  <= G;
    else
        min  <= R;
end
// Next delay R, G, B to fit with pipeline and get ready for signed arithmetic

// hopefully \( R = 100 \) will be \( Rd = +100 \)
Rd <= R;
Gd <= G;
Bd <= B;

// STAGE 2: NUMERATOR AND DENOMINATOR GENERATION
// Latency: 1 (a 9-bit signed subtractor w/ 9-bit signed output)

// STAGE 3: MULTIPLIER
// Multiply numerator by 0d60
// Latency: 2 (9-bit signed multiplier by constant 0d60, output is 15-bit signed)

// STAGE 4: DIVIDER and ADDING SELECTOR
// Divide the numerator*60 by denominator
// Latency: 15+4=19 (15-bit-signed-numerator divider)

hueUndefinedD <= hueUndefined;
hueBuffered <= hueOut;
satBuffered <= satOut;

end

dendumodule

// Pushbutton Debounce Module (video version)
module debounce (reset, clock_65mhz, noisy, clean);
    input reset, clock_65mhz, noisy;
    output clean;
    reg [19:0] count;

reg new, clean;

always @(posedge clock_65mhz)
  if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
  else if (noisy != new) begin new <= noisy; count <= 0; end
  else if (count == 650000) clean <= new;
  else count <= count+1;
endmodule

Game Engine

module Game
  input clock, reset, hCount, vCount, hsync, vsync, blank, AICrosshairX, AICrosshairY, AIFire, playerCrosshairX, playerCrosshairY, playerFire, calibrate, startPlaying, leaveCalibrate, phsync, pvsync, pblank, pixel, duckX, duckY, duckAlive, AIShotsLeft, playerShotsLeft, roundOver, roundStart, dir, inCalibrate;
  input [7:0] pixel; //rgb coloring of pixel
  input [10:0] duckX; //duck's x coordinate
  input [9:0] duckY; //duck's y coordinate
  input duckAlive; //1 if duck is alive
  input [1:0] AIShotsLeft; //number of remaining shots for AI
  input [1:0] playerShotsLeft; //number of remaining shots for player
  input roundOver; //asserted high for one clock cycle if round expired due to time elapsing
  output roundStart; //high for one clock cycle when round starts
  output [1:0] dir; //direction that duck is travelling
  output inCalibrate; //high when device is in calibration mode

  reg [7:0] pixel;
  reg oldvsync; //stores value of vsync from one clock cycle ago, used for edge detection
reg [10:0] duckX;       //duck's x coordinate
reg [9:0] duckY;       //duck's y coordinate
reg duckAlive;         //1 if duck is alive
reg [2:0] duckState;   //specifies which image of the duck
                      //should be displayed
reg [1:0] dir;         //duck's current heading
reg [3:0] duckSpeed;   //duck's speed
reg [4:0] playerScore; //player's score
reg [4:0] AIScore;     //AI's score
reg [3:0] roundNumber; //round number
reg startTimer;        //asserted to start timer running

reg signed [4:0] duckXAdj, duckYAdj; //equal to the change in x and y per
frame, signed since x and y could increase or decrease
                      //these are equal to +/- duckSpeed,
with the sign determined by current direction

reg [4:0] animCount;   //used to make each frame of duck animation
                      //last for several frames of video output
reg duckHit;           //1 when duck is shot
reg [1:0] AIShotsLeft; //number of remaining shots for AI
reg [1:0] playerShotsLeft; //number of remaining shots for player
reg [7:0] respawnCount; //used to count frames between duck respawn
reg [19:0] ducksHit;   //specifies state of all ducks in round (is alive or
                      //which player killed)
reg [10:0] popupX;     //x coordinate of popup
reg [9:0] popupY;      //y coordinate of popup
reg justRespawned;     //high for one cycle when the duck just respawned
reg [10:0] roundX;     //x coordinate of top left corner of round display
reg [9:0] roundY;      //y coordinate of top left corner of round display
reg [23:0] roundString; //string to display in round display
reg resetTimer;        //asserted to reset timer
reg playerWin;         //1 if player killed duck in last round
reg oldPlayerFire1;    //player fire delayed by one update cycle
reg oldPlayerFire2;
reg oldPlayerFire3;
reg oldPlayerFire4;
reg oldPlayerFire5;
reg oldPlayerFire6;

parameter hDim=1024;    //horizontal dimension of screen
parameter vDim=768;     //vertical dimension of screen
parameter horiz=450;    //distance of horizon from top of
                      //screen
parameter duckWidth=64; //width of duck
parameter duckHeight=64; //height of duck
parameter animPeriod=5'b01010; //duration in frames of duck animation
parameter fallSpeed=4'b0110; //speed of fall

//enum of directions, north east, north west, and so on
parameter dirNE = 2'b00;
parameter dirNW = 2'b01;
parameter dirSE = 2'b10;
parameter dirSW = 2'b11;

//enum of all possible states that the duck could be in, each state
//corresponds to a different image
parameter fly1=3'b0000;
parameter fly2=3'b0011;
parameter fly3=3'b010;
parameter fly4=3'b011;
parameter hit=3'b100;
parameter fall1=3'b101;
parameter fall2=3'b110;
parameter dead=3'b111;

// number of frames before duck respawns
parameter respawnDelay=120; // 1 second at 60 Hz refresh rate

// used to specify entries of ducksHit, which holds state of ducks for score overlay
parameter playerKilledDuck=2'b10;
parameter AKilledDuck=2'b01;

parameter popupHide=1000;
parameter popupDisplay=338;
parameter borderPixels=30; // number of pixels in border

// connections to game objects
wire [10:0] hCount;
wire [9:0] vCount;
wire [7:0] backgroundPixel, duckPixel, treePixel, crosshairPixel, AICrosshairPixel, roundPixel, scorePixel, roundOverPixel, titlePixel;
wire duckHasPixel, treeHasPixel, crosshairHasPixel, AICrosshairHasPixel, roundOver, score1, score2, scoreHasPixel, roundOverHasPixel, titleHasPixel, doneTitle, calibrateMode;

// outputted syncs and blank are same as input
assign phsync=hsync;
assign pvsync=vsync;
assign pblank=blank;

assign roundStart=startTimer; // round starts when timer starts
assign inCalibrate=calibrateMode;

always @ (posedge clock) begin
  oldvsync<=vsync;
  
  if ((reset) || (roundNumber>=10)) begin
    duckX<=hDim/2;
    duckY<=horiz+1;
    dir<=dirNE;
    duckSpeed<=4'b0110;
    duckState<=fly1;
    animCount<=5'b00000;
    duckHit<=0;
    AIShotsLeft<=2'b11;
    playerShotsLeft<=2'b11;
    AIScore<=5'b00000;
    playerScore<=5'b00000;
    duckAlive<=1'b1;
    roundNumber<=4'b0000;
    startTimer<=1'b1;
    respawnCount<=8'b00000000;
    resetTimer<=1'b0;
    ducksHit<=20'b0;
    popupX<=hDim/2;
    popupY<=popupHide; // offscreen
    playerWin<=1'b0;
    oldPlayerFire1<=1'b0;
    oldPlayerFire2<=1'b0;
    oldPlayerFire3<=1'b0;
  end
end


oldPlayerFire4<=1'b0;
oldPlayerFire5<=1'b0;
oldPlayerFire6<=1'b0;
end

//if vsync just stepped low, and title screen is done, update state
else if (((oldvsync == 1) && (vsync==0)) && (doneTitle==1)) begin
oldPlayerFire1<=playerFire;
oldPlayerFire2<=oldPlayerFire1;
oldPlayerFire3<=oldPlayerFire2;
oldPlayerFire4<=oldPlayerFire3;
oldPlayerFire5<=oldPlayerFire4;
oldPlayerFire6<=oldPlayerFire5;
//increment animCount each frame
if (animCount==animPeriod-1)
animCount<='5'b00000;
else
animCount<=animCount+1;

//increment animation state
if (animCount == 5'b00000) begin
    case (duckState)
        fly1: duckState<=duckHit ? hit : (roundOver ?
            dead : fly2);
        fly2: duckState<=duckHit ? hit : (roundOver ?
            dead : fly3);
        fly3: duckState<=duckHit ? hit : (roundOver ?
            dead : fly4);
        fly4: duckState<=duckHit ? hit : (roundOver ?
            dead : fly1);
        hit: duckState<=fall1;
        fall1:
            duckState<=(duckY+fallSpeed+duckHeight>=horiz) ? dead : fall2;
        fall2:
            duckState<=(duckY+fallSpeed+duckHeight>=horiz) ? dead : fall1;
        dead: duckState<=(respawnCount>=respawnDelay)
            ? fly1 : dead;
        default: duckState<=fly1;
    endcase
end

//duck respawning, start of new round
if(duckState==dead) begin
    duckX<=hDim/2;
    duckY<=horiz-10;
    if(respawnCount>=respawnDelay) begin
        respawnCount<=0;
        AIShotsLeft<='2'b11;
        playerShotsLeft<='2'b11;
        dir<=(roundNumber==((roundNumber/2)*2)) ?
            dirNE : dirNW;
        duckAlive<=1'b1;
        duckHit<=0;
        duckState<=fly1; //just in case, already set above, but done again to avoid any chance of glitch
        startTimer<=1'b1;
        popupY<=popupHide;
    end
    else begin
        popupY<=popupDisplay;
respawnCount<=respawnCount+1;
startTimer<=1'b0;
end
else begin
  startTimer<=1'b0;
end

// determines the Adjustment to duck's x and y coordinates
// if flying, move in given direction
if(duckState==fly1 || duckState==fly2 || duckState==fly3 || duckState==fly4) begin
  case (dir)
    dirNE: begin
      duckXAdj<=duckSpeed-2*(roundNumber-((roundNumber/2)*2));
      duckYAdj<=-duckSpeed;
    end
    dirNW: begin
      duckXAdj<=(-duckSpeed)+2*(roundNumber-((roundNumber/2)*2));
      duckYAdj<=-duckSpeed;
    end
    dirSE: begin
      duckXAdj<=duckSpeed-2*(roundNumber-((roundNumber/2)*2));
      duckYAdj<=duckSpeed;
    end
    dirSW: begin
      duckXAdj<=(-duckSpeed)+2*(roundNumber-((roundNumber/2)*2));
      duckYAdj<=duckSpeed;
    end
  endcase
end
else begin
// after being hit
  if dead, don't move
  if(duckState==dead) begin
    duckXAdj<=0;
    duckYAdj<=0;
  end
  else begin
    // when first hit don't move
    if(duckState==hit) begin
      duckXAdj<=0;
      duckYAdj<=0;
    end
    else begin
      // then fall down
      duckXAdj<=0;
      duckYAdj<=fallSpeed;
    end
  end
end
if adjustments to x and y coordinate keep the duck in the sky or behind the grass, just move the duck
if ((duckX+duckXAdj>=borderPixels) && (duckX+duckXAdj+duckWidth<=hDim-borderPixels) && (duckY+duckYAdj>=horiz) && ((duckState==dead)) begin
  duckX<=duckX+duckXAdj;
  duckY<=duckY+duckYAdj;
end
else begin
  if(~(duckState==dead)) begin
    case (dir)
      dirNE: dir <= (duckX+duckXAdj+duckWidth < hDim-borderPixels) ? dirSE : (((duckX+duckXAdj+duckWidth == hDim-borderPixels) && (duckY+duckYAdj == borderPixels)) ? dirSW : dirNW);
      dirNW: dir <= (duckX+duckXAdj > borderPixels) ? dirSE : (((duckX+duckXAdj == borderPixels) && (duckY+duckYAdj == borderPixels)) ? dirNE : dirSW);
      dirSE: dir <= (duckX+duckXAdj+duckWidth < hDim-borderPixels) ? dirNE : (((duckX+duckXAdj+duckWidth == hDim-borderPixels) && (duckY+duckYAdj == horiz)) ? dirNW : dirSW);
      dirSW: dir <= (duckX+duckXAdj > borderPixels) ? dirNE : (((duckX+duckXAdj == borderPixels) && (duckY+duckYAdj == horiz)) ? dirSE : dirNE);
    endcase
    //moves the duck during reflection
    duckX<=((duckX+duckXAdj+duckWidth < hDim-borderPixels) && (duckX+duckXAdj > borderPixels)) ? duckX+duckXAdj : duckX-duckXAdj;
    duckY<=((duckY+duckYAdj < horiz) && (duckY+duckYAdj > borderPixels)) ? duckY+duckYAdj : duckY-duckYAdj;
  end
end
//handles player shooting
if(oldPlayerFire5 && (~(oldPlayerFire6)) && (playerShotsLeft>0)) begin
  playerShotsLeft<=playerShotsLeft-1;
  if((playerCrosshairX==duckX) && (playerCrosshairY<duckY) && (playerCrosshairY<duckY+duckHeight)) begin
    duckHit<=1;
    duckAlive<=1'b0;
    roundNumber<=roundNumber+1;
    resetTimer<=1'b1;
    playerWin<=1'b1;
    if(score2) begin
      playerScore<=playerScore+1;
    end
    else begin
      if(score1) playerScore<=playerScore+2;
      else playerScore<=playerScore+3;
    end
    case (roundNumber)
      0:ducksHit[1:0]<=playerKilledDuck;
      1:ducksHit[3:2]<=playerKilledDuck;
      4:ducksHit[9:8]<=playerKilledDuck;
      7:ducksHit[15:14]<=playerKilledDuck;
      8:ducksHit[17:16]<=playerKilledDuck;
    endcase
  end
end
9:ducksHit[19:18]<=playerKilledDuck;
default:ducksHit<=ducksHit;
endcase
end

derelse begin
   //handles AI shooting
   if(AIFire && (AIShotsLeft>0)) begin
      AIShotsLeft<=AIShotsLeft-1;
      if((AICrosshairX>=duckX)&&(AICrosshairX<duckX+duckWidth)&&(AICrosshairY>=
duckY)&&(AICrosshairY<duckY+duckHeight)) begin
         duckHit<=1;
         duckAlive<=1'b0;
         roundNumber<=roundNumber+1;
         resetTimer<=1'b1;
         playerWin<=1'b0;
         if(score2) begin
            AIScore<=AIScore+1;
         end
         else begin
            if(score1) AIScore<=AIScore+2;
            else AIScore<=AIScore+3;
         end
       case (roundNumber)
         0:ducksHit[1:0]<=AIKilledDuck;
         1:ducksHit[3:2]<=AIKilledDuck;
         4:ducksHit[9:8]<=AIKilledDuck;
         7:ducksHit[15:14]<=AIKilledDuck;
         8:ducksHit[17:16]<=AIKilledDuck;
         default:ducksHit<=ducksHit;
       endcase
     end
     else begin
       resetTimer<=1'b0;
     end
   end
end

Background aBackground(hCount,vCount,backgroundPixel);
defparam aBackground.horizon = horiz;

Duck aDuck(clock, duckX, duckY, hCount, vCount, duckState,
duckHasPixel, duckPixel);
defparam aDuck.flyWidth = duckWidth;
defparam aDuck.fallWidth = duckWidth/2;
defparam aDuck.height= duckHeight;
TreeGrass aTree(clock, hCount, vCount, treeHasPixel, treePixel);

// player's crosshair
Crosshair aCrosshair(clock, playerCrosshairX, playerCrosshairY, hCount, vCount, crosshairHasPixel, crosshairPixel);
defparam aCrosshair.color = 8'b00011100;

// ai's crosshair
Crosshair bCrosshair(clock, AICrosshairX, AICrosshairY, hCount, vCount, AICrosshairHasPixel, AICrosshairPixel);
defparam bCrosshair.color = 8'b11100000;

// round timer
RoundTimer aRoundTimer(clock, reset, startTimer, resetTimer, roundOver);
defparam aRoundTimer.numSeconds = 10;

// one second score timer
RoundTimer scoreTimer1(clock, reset, startTimer, resetTimer, score1);
defparam scoreTimer1.numSeconds = 1;

// three second score timer
RoundTimer scoreTimer2(clock, reset, startTimer, resetTimer, score2);
defparam scoreTimer2.numSeconds = 3;

// score overlay
ScoreOverlay aScoreOverlay(clock, reset, hCount, vCount, playerScore, playerShotsLeft, ducksHit, scoreHasPixel, scorePixel);

// round over overlay
RoundOverOverlay aRoundOverOverlay(clock, reset, roundOver, hCount, vCount, popupX, popupY, playerWin, roundOverHasPixel, roundOverPixel);

wire compoundReset;
assign compoundReset=(roundNumber>=10);

// title screen
TitleScreen aTitleScreen(clock, reset, compoundReset, hCount, vCount, calibrate, startPlaying, leaveCalibrate, titleHasPixel, titlePixel, doneTitle, calibrateMode);

// round display font rom, still need to implement roundX, roundY, and roundString calculations
aChar_string_display(clock, hcount, vcount, roundString, roundX, roundY, roundPixel);
defparam aChar_string_display.NCHAR=4;
defparam aChar_string_display.NCHAR_BITS=2;

// update pixel whenever a new location is requested, will be cleaned up to avoid all this if/else
always @ * begin
  if(doneTitle) begin
    if (scoreHasPixel) pixel<=scorePixel;
    else begin
      if (crosshairHasPixel) pixel<=crosshairPixel;
      else begin
        if (AICrosshairHasPixel) pixel<=AICrosshairPixel;
        else begin
          if (treeHasPixel) pixel<=treePixel;
        end
      end
    end
  end
  else begin
    if (AICrosshairHasPixel) pixel<=AICrosshairPixel;
    else begin
      if (treeHasPixel) pixel<=treePixel;
    end
  end
end
if (roundOverHasPixel)
    pixel<=roundOverPixel;
else begin
    if (duckHasPixel)
        pixel<=duckPixel;
    else
        pixel<=backgroundPixel;
end
endmodule

---

**Graphics Pipeline**

**Background Sky**

```
// Background: blue sky

module Background (hCount, vCount, pixel);
    input [10:0] hCount; // hCount is x coordinate of current pixel being requested
    input [9:0] vCount; // vCount is y coordinate of the pixel being requested
    output [7:0] pixel; // rgb value of pixel

    parameter hDim=1024; // hDim is horizontal size of screen
    parameter vDim=768; // vDim is vertical size of screen
    parameter horizon=450; // location of horizon as measured from top of screen
    parameter topColor=8'b00000111; // color of sky
    parameter bottomColor=8'b11111010; // color of dirt, never actually displayed, grass and dirt display over it, there just in case
    parameter blackColor=8'b00000000; // border
    parameter dBlack=30; // 30 pixel black border

    assign pixel = ((hCount<=dBlack) || (hCount>hDim-dBlack) || (vCount<=dBlack)) ? blackColor : (((vCount >= 0) && (vCount <= horizon + 50)) ? topColor : bottomColor);

endmodule
```

**Duck**

```
// Duck: animated duck
```

```
```
module Duck (clk, x, y, hCount, vCount, state, hasPixel, pixel);
  input clk;      //clock
  input [10:0] x, hCount;  //x is the duck's x position, hCount is the x coordinate of the pixel being requested
  input [9:0] y, vCount;  //y is the duck's y position, vCount is the y coordinate of the pixel being requested
  input [2:0] state;  //specifies which of the possible images of the duck to display, like fly or falling

  output hasPixel;  //asserted high if the duck has a pixel in that region
  output [7:0] pixel;  //rgb value of the pixel

  reg hasPixel;   //asserted high if the duck has a pixel in that region
  reg [7:0] pixel;  //rgb value of the pixel

  reg [11:0] addr; //addr to read from ram

  wire [7:0] dout1,dout2,dout3,dout4,dout5,dout6; //enum of all possible states that the duck could be in, each state corresponds to a different image
  parameter fly1=3'b000;
  parameter fly2=3'b001;
  parameter fly3=3'b010;
  parameter fly4=3'b011; //same as fly2 in terms of image
  parameter hit=3'b100;
  parameter fall1=3'b101;
  parameter fall2=3'b110;
  parameter dead=3'b111;

  parameter flyWidth=64;
  parameter fallWidth=32;
  parameter height=64;

  wire [6:0] width;

  //these roms contain the bitmaps of the duck in various positions
  fly1rom aFly1Rom(addr, clk, dout1);
  fly2rom aFly2Rom(addr, clk, dout2);
  fly3rom aFly3Rom(addr, clk, dout3);
  hitrom aHitRom(addr, clk, dout4);
  fall1rom aFall1Rom(addr, clk, dout5);
  fall2rom aFall2Rom(addr, clk, dout6);

  //width of duck sprite different when flying or falling
  assign width=(state==fly1 || state==fly2 || state==fly3 || state==fly4 || state==hit) ? flyWidth : fallWidth;

  always @(posedge clk) begin
    //check to see if requested pixel is within boundary
    if ((hCount >= x) && (hCount < x+width) && (vCount >= y) && (vCount < y+height)) begin
      hasPixel<=1;
      //determine which address to read
      addr<=width*(vCount-y)+(hCount-x);

      //determine which output to use
      case (state)
        fly1:pixel<=dout1;
        fly2:pixel<=dout2;
        fly3:pixel<=dout3;
        fly4:pixel<=dout2;
null
//check if pixel is within boundary of grass, divisions by 4 are due to compression of grass bitmap
if (vCount > (screenHeight - grassDirtHeight)) && (vCount < screenHeight) begin
    if (hCount > hBlack) || (hCount > (screenWidth - hBlack)) begin
        pixel <= 8'b00000000;
        hasPixel <= 1'b1;
    end
    else begin
        addr2 <= (screenWidth/4)*(vCountPrime - (screenHeight - grassDirtHeight)/4) + hCountPrime;
        //bitmap for grass and dirt is compressed by 4
        if (dout2 == 8'b0000111) begin //if pixel is blue,
            don't display so duck will be on top of blue sky but behind grass
        //check to see if requested pixel is within boundary of tree, division by 4 due to compression of tree bitmap
        if (hCount >= x) && (hCount < x+width) && (vCount >= y) && (vCount < y+height) begin
            //determine which address to read
            addr1 <= width/4*(vCount/4-y/4) + (hCount/4-x/4);
            pixel <= dout1;
            hasPixel <= 1'b1;
        end
        else begin
            hasPixel <= 1'b0;
        end
        else begin
            pixel <= dout2;
            hasPixel <= 1'b1;
        end
    end
end
else begin
    //check to see if requested pixel is within boundary of tree, division by 4 due to compression of tree bitmap
    if (hCount >= x) && (hCount < x+width) && (vCount >= y) && (vCount < y+height) begin
        //determine which address to read
        addr1 <= width/4*(vCount/4-y/4) + (hCount/4-x/4);
        pixel <= dout1;
        hasPixel <= ~(dout1 == 8'b0000111);    //if pixel is blue, don't display so duck will be on top of blue sky but behind tree
    end
    else begin
        hasPixel <= 1'b0;
    end
end

endmodule

Crosshair

//------------------------------
module Crosshair (clk, x, y, hCount, vCount, hasPixel, pixel);

input clk;      //clock
input [10:0] x;     //x coordinate of crosshair(center)
input [9:0] y;     //y coordinate of crosshair(center)
input [10:0] hCount;    //horizontal location of the pixel being requested
input [9:0] vCount;    //vertical location of the pixel being requested

output hasPixel;    //asserted high if this object has a pixel at hCount, vCount
output [7:0] pixel;   //rgb value of pixel

reg hasPixel;
reg [7:0] pixel;
reg [7:0] addr;
wire dout;

parameter height=16;    //height of crosshair
parameter width=16;    //width of crosshair
parameter color=8'b11111111; //color of crosshair
parameter hOff=0; //memory offset to deal with delay

crosshairrom aCrosshairRom(addr,clk,dout);

always @(posedge clk) begin
    //check to see if requested pixel is within boundary, horiz offset of 1 for timing reasons
    if ((hCount+hOff >= x-(width/2)) && (hCount+hOff < x+(width/2)) && (vCount >= y-(height/2)) && (vCount < (y+height/2))) begin
        //determine which address to read
        addr<=width*(vCount-(y-height/2))+(hCount+hOff-(x-width/2));
        hasPixel<=dout;
        pixel<=color;
    end
    else begin
        hasPixel<=0;
    end
end
endmodule

module AI(clk,reset,curDuckX,curDuckY,duckAlive,shotsLeft,roundStart,roundTimeExpired, dir,vsync,difficulty,x,y,fire);

input clk; //clock
input reset; //global reset, active high
input [10:0] curDuckX; //duck's x coordinate
input [9:0] curDuckY; //duck's y coordinate
input duckAlive; //1 if duck is alive
input [1:0] shotsLeft; //number of shots left this round
input roundStart; //high for one clock cycle at start of round
input roundTimeExpired; //high if round timer expires
input [1:0] dir; //direction that duck is travelling
input vsync; //vertical sync signal
input [1:0] difficulty; //AI difficult, 00 is very hard, 01 is medium, 10 is easy, 11 is very easy

output [10:0] x; //x coordinate of crosshair
output [9:0] y; //y coordinate of crosshair
output fire; //high for one cycle when firing

reg [10:0] x; //x coordinate of crosshair
reg [9:0] y; //y coordinate of crosshair
reg fire; //high for one cycle when firing
reg justFired; //high for ten cycles after firing
reg [3:0] fireTimeCount; //used to count time since last firing
reg freeToShoot; //1 when AI is allowed to shoot
reg roundOver; //1 if round is over
reg resetTimer; //asserted to reset timer
reg [10:0] oldDuckX; //value of curDuckX from last clock cycle
reg [9:0] oldDuckY; //value of curDuckY from last clock cycle
reg [10:0] duckX; //estimated value of new curDuckX
reg [9:0] duckY; //estimated value of new curDuckY

parameter hDim=1024; //horizontal dimension of screen
parameter vDim=768; //vertical dimension of screen
parameter duckWidth=64; //width of duck
parameter duckHeight=64; //height of duck

parameter maxMove=40; //largest distance that the crosshair can move per update, in each of x or y
parameter shotAdj=8+(4*difficulty); //min distance from border to take shot
parameter shotDelay=7; //number of seconds between start of round before attempting shooting

enum of directions, north east, north west, and so on
parameter dirNE = 2'b00;
parameter dirNW = 2'b01;
parameter dirSE = 2'b10;
parameter dirSW = 2'b11;

wire delayOver;
//round timer
RoundTimer shotDelayTimer(clk,reset,roundStart,resetTimer,delayOver);
defparam shotDelayTimer.numSeconds = shotDelay+difficulty;

always @(posedge clk) begin
roundOver<=!duckAlive || roundTimeExpired;
oldDuckX<=curDuckX;
oldDuckY<=curDuckY;
duckX<=(2*curDuckX)-oldDuckX;
duckY<=(2*curDuckY)-oldDuckY;
resetTimer<=1'b0;
if (reset) begin
    x<=hDim/2;
y<=vDim/4;
    justFired<=0;
    fireTimeCount<=4'b0000;
    fire<=1'b0;
    freeToShoot<=1'b0;
    oldDuckX<=curDuckX;
    oldDuckY<=curDuckY;
end
else begin

    //determines when AI is free to shoot
    if(roundOver) begin
        freeToShoot<=1'b0;
    end
else begin
    if(delayOver) begin
        freeToShoot<=1'b1;
    end
end

    //if duck just moved
    if(~((curDuckX==oldDuckX) && (curDuckY==oldDuckY)) begin
        //if x coordinate near duck x center, only move a little
        if ((x>=duckX+duckWidth/2-maxMove) && (x<duckX+duckWidth/2+maxMove)) begin
            x<=duckX+duckWidth/2;
        end
        //otherwise move a lot
    else begin
        if (x<duckX+duckWidth/2-maxMove) x<=x+maxMove;
        else x<=x-maxMove;
    end

    //if y coordinate near duck y center, only move a little
    if ((y>=duckY+duckHeight/2-maxMove) && (y<duckY+duckHeight/2+maxMove)) begin
        y<=duckY+duckHeight/2;
    end
    //otherwise move a lot
else begin
    if (y<duckY+duckHeight/2-maxMove) y<=y+maxMove;
    else y<=y-maxMove;
end

    //handles firing
    if ((x>=duckX+shotAdj) && (x<duckX+duckWidth-shotAdj) && (y>=duckY+shotAdj) && (y<duckY+duckHeight-shotAdj) && (shotsLeft>0) && (freeToShoot==1'b1) && duckAlive && ~justFired) begin
        justFired<=1;
        fireTimeCount<=0;
        fire<=1'b1;
    end
else fire<=1'b0;
    //increment fireTimeCount and reset justFired to 0
when fireTimeCount reaches 10
    if (justFired) begin

if(fireTimeCount<10) 
    fireTimeCount=fireTimeCount+1; 
    end 
else 
    justFired<=0; 
end 
end 
endmodule 

/************************************ 
// playerInput (temporary, for test purposes only) 
// ************************************
module playerInput(clk,reset,moveLeft,moveRight,moveUp,moveDown,fireIn,shotsLeft,x,y,fireOut); 
    input clk,reset,moveLeft,moveRight,moveUp,moveDown,fireIn; 
    input [1:0] shotsLeft; 
    output [10:0] x; 
    output [9:0] y; 
    output fireOut; 
    reg [10:0] x; 
    reg [9:0] y; 
    parameter screenWidth=1024; 
    parameter screenHeight=768; 
    parameter posAdj=20; 
    reg oldMoveLeft,oldMoveRight,oldMoveUp,oldMoveDown,oldFireIn,fireOut; 
    reg [1:0] oldShotsLeft;
always @ (posedge clk) begin 
    oldMoveLeft<=moveLeft; 
    oldMoveRight<=moveRight; 
    oldMoveUp<=moveUp; 
    oldMoveDown<=moveDown; 
    oldFireIn<=fireIn; 
    oldShotsLeft<=shotsLeft; 
    if(reset) begin 
        x<=screenWidth/2; 
        y<=screenHeight/2; 
    end 
    else begin 
        if(moveLeft && !oldMoveLeft)begin 
            x=x-posAdj; 
        end 
        else begin 
            if (moveRight && !oldMoveRight) begin 
                x= x+posAdj; 
        end 
        end 
        if(moveUp && !oldMoveUp)begin 
            y=y-posAdj; 
        end 
        else begin 
            if (moveDown && !oldMoveDown) begin 

```verilog
module RoundTimer(clk, reset, startTimer, resetTimer, timeElapsed);
  input clk; //clock
  input reset; //global reset
  input startTimer; //high for 1 cycle when the timer should be started
  input resetTimer; //if asserted, resets timer to 0;
  output timeElapsed; //high for 1 cycle when time elapsed
  reg timeElapsed; //high when time elapsed
  reg currentlyTiming; //1 when timing
  reg [0:29] count; //used to count clock cycles, excessively large to allow larger
  //timing intervals to be implemented without danger of overflow
  parameter numSeconds=5; //number of seconds before round over
  parameter clockFreq=6500000; //clock frequency

  always @ (posedge clk) begin
    if(reset || resetTimer) begin
      count<=30'b0;
      currentlyTiming<=1'b0;
      timeElapsed<=1'b0;
    end
    else begin
      if(startTimer && !currentlyTiming) begin
        currentlyTiming<=1'b1;
        timeElapsed<=1'b0;
      end
      else begin
        if(currentlyTiming && (count>=numSeconds*clockFreq))
          begin
            timeElapsed<=1'b1;
            count<=30'b0;
            currentlyTiming<=1'b0;
          end
        else begin
          if(currentlyTiming && (count>=numSeconds*clockFreq))
            begin
              timeElapsed<=1'b1;
              count<=30'b0;
              currentlyTiming<=1'b0;
            end
          else begin
            count<=count+1;
            //timeElapsed<=1'b0;
          end
        end
      end
    end
  end
endmodule
```

**Round Timer**

// RoundTimer: timer to determine when round ends

//

//---------------------------------------------
// RoundTimer: timer to determine when round ends
//
//---------------------------------------------
```
module ScoreOverlay(clk, reset, hCount, vCount, playerScore, playerShotsLeft, ducksHit, hasPixel, pixel);
  input clk; //system clock
  input reset; //global reset
  input [10:0] hCount; //horizontal location of the pixel being requested
  input [9:0] vCount; //vertical location of the pixel being requested
  input [4:0] playerScore; //players score in hundreds
  input [1:0] playerShotsLeft; //number of shots remaining
  input [19:0] ducksHit; //which ducks were hit by whom, 10 pairs, 00 means not hit, 10 means hit by player, 01 means hit by AI
  output hasPixel; //asserted high if this object has a pixel at hCount, vCount
  output [7:0] pixel; //rgb value of pixel

  reg [7:0] addr1, addr2, addr3;
  reg [3:0] selectedDuck; //duck whose pixel is currently requested
  reg [1:0] selectedDuckState; //state of selected duck
  reg [1:0] selectedBullet;

  reg hasPixel;
  reg [7:0] pixel;

  reg hasPixel1, hasPixel2, hasPixel3, pixel1;
  reg [7:0] fpixel1, pixel2, pixel3;
  reg [3:0] digit1, digit2; //first and second digit of score

  wire [7:0] doutB;
  wire doutA, dout0, dout1, dout2, dout3, dout4, dout5, dout6, dout7, dout8, dout9;

  parameter deadDuckX=360; //x coordinate of top left corner of first duck
  parameter deadDuckY=628; //y coordinate of top left corner of first duck
  parameter duckHeight=16; //height of duck
  parameter duckWidth=16; //width of duck
  parameter numDucks=10; //number of ducks per round
  parameter bulletX=112; //x coordinate of top left corner of first bullet
  parameter bulletY=628; //y coordinate of top left corner of first bullet
  parameter bulletWidth=16; //width of bullet
  parameter bulletHeight=15; //height of bullet
parameter scoreX=812; //x coordinate of top left corner of first score digit
parameter scoreY=612;  //y coordinate of top left corner of first score digit
parameter scoreWidth=16;  //width of score digit
parameter scoreHeight=16; //height of score digit
parameter scoreDigits=4;  //number of digits in score

//enum of duck state and color
parameter duckAlive=2'b00;
parameter duckDeadPlayer=2'b10;
parameter duckDeadAI=2'b01;
parameter aliveColor=8'b11111111;
parameter playerColor=8'b00011100;
parameter AIColor=8'b11100000;

scoreduckrom    aScoreDuckRom(addr2,clk,doutA);
scorebulletrom  aScoreBulletRom(addr3,clk,doutB);
score0rom       aScore0rom(addr1,clk,dout0);
score1rom       aScore1rom(addr1,clk,dout1);
score2rom       aScore2rom(addr1,clk,dout2);
score3rom       aScore3rom(addr1,clk,dout3);
score4rom       aScore4rom(addr1,clk,dout4);
score5rom       aScore5rom(addr1,clk,dout5);
score6rom       aScore6rom(addr1,clk,dout6);
score7rom       aScore7rom(addr1,clk,dout7);
score8rom       aScore8rom(addr1,clk,dout8);
score9rom       aScore9rom(addr1,clk,dout9);

always @(posedge clk) begin
  if(reset) begin
    addr2<=8'b0;
    selectedDuck<=4'b0;
  end
  else begin
    //compute score digits
    if(playerScore<10)
      digit1<=0;
    else begin
      if(playerScore<20)
        digit1<=1;
      else begin
        if(playerScore<30)
          digit1<=2;
        else
          digit1<=3;
      end
    end
    digit2<=playerScore-digit1*10;
    //check if requested pixel is within boundary of any duck
    if((hCount>=deadDuckX) && (hCount<deadDuckX+numDucks*duckWidth) &&
      (vCount>=deadDuckY) && (vCount<deadDuckY+duckHeight)) begin
      selectedDuck<=((hCount-deadDuckX)/duckWidth);
      case (selectedDuck)
        0: selectedDuckState<=ducksHit[1:0];
        1: selectedDuckState<=ducksHit[3:2];
        2: selectedDuckState<=ducksHit[5:4];
        3: selectedDuckState<=ducksHit[7:6];
      default:
        //default case
      endcase
    end
  end
end
4: selectedDuckState<=ducksHit[9:8];
5: selectedDuckState<=ducksHit[11:10];
6: selectedDuckState<=ducksHit[13:12];
7: selectedDuckState<=ducksHit[15:14];
8: selectedDuckState<=ducksHit[17:16];
9: selectedDuckState<=ducksHit[19:18];
default: selectedDuckState<=ducksHit[1:0];
endcase
addr2<=hCount-deadDuckX-(selectedDuck*duckWidth)+(vCount-deadDuckY)*duckHeight;
hasPixel2<=doutA;
case (selectedDuckState)
  duckAlive: pixel2<=aliveColor;
  duckDeadPlayer: pixel2<=playerColor;
  duckDeadAI: pixel2<=AIColor;
  default: pixel2<=aliveColor;
endcase
end
else begin
  pixel2<=8'b0;
  hasPixel2<=1'b0;
end

//check if requested pixel is within boundary of bullets
if((hCount>=bulletX) && (hCount<bulletX+bulletWidth*playerShotsLeft) && (vCount>=bulletY) && (vCount<bulletY+bulletHeight)) begin
  hasPixel3<=1'b1;
  pixel3<=doutB;
  selectedBullet<=(hCount-bulletX)/(bulletWidth);
  addr3<=hCount-bulletX-(selectedBullet*bulletWidth)+(vCount-bulletY)*bulletHeight;
end
else begin
  pixel3<=8'b0;
  hasPixel3<=1'b0;
end

//check if requested pixel is within boundary of score
if((hCount>=scoreX) && (hCount<scoreX+scoreWidth*2*scoreDigits) && (vCount>=scoreY) && (vCount<scoreY+scoreHeight*2)) begin
  //first digit
  if(hCount<scoreX+scoreWidth*2) begin
    hasPixel1<=1'b1;
    addr1<=(hCount-scoreX)/2+((vCount-scoreY)/2)*(scoreHeight);
    case (digit1)
      0:pixel1<=dout0;
      1:pixel1<=dout1;
      2:pixel1<=dout2;
      3:pixel1<=dout3;
      4:pixel1<=dout4;
      5:pixel1<=dout5;
      6:pixel1<=dout6;
      7:pixel1<=dout7;
      8:pixel1<=dout8;
      9:pixel1<=dout9;
      default:pixel1<=dout0;
    endcase
  end
  else begin
    //second digit
    hasPixel1<=0;
if(hCount<scoreX+scoreWidth*4) begin
  hasPixel1<=1'b1;
  addr1<=(hCount-scoreX-scoreWidth*2)/2+((vCount-scoreY)/2)*(scoreHeight);
  case (digit2)
    0:pixel1<=dout0;
    1:pixel1<=dout1;
    2:pixel1<=dout2;
    3:pixel1<=dout3;
    4:pixel1<=dout4;
    5:pixel1<=dout5;
    6:pixel1<=dout6;
    7:pixel1<=dout7;
    8:pixel1<=dout8;
    9:pixel1<=dout9;
    default:pixel1<=dout0;
  endcase
end
else begin
  //third digit
  if(hCount<scoreX+scoreWidth*6) begin
    hasPixel1<=1'b1;
    pixel1<=dout0;
    addr1<=(hCount-scoreX-scoreWidth*4)/2+((vCount-scoreY)/2)*(scoreHeight);
  end
  //fourth digit
  else begin
    hasPixel1<=1'b1;
    pixel1<=dout0;
    addr1<=(hCount-scoreX-scoreWidth*6)/2+((vCount-scoreY)/2)*(scoreHeight);
  end
end
else begin
  pixel1<=8'b0;
  hasPixel1<=1'b0;
end

end
endmodule

---

**Round Over Overlay**

//Round over overlay

```verilog
module RoundOverOverlay(clk, reset, roundOver, hCount, vCount, x, y, win, hasPixel, pixel);
  input clk; //system clock
  input reset; //global reset
  input roundOver; //asserted high when round over
```
input [10:0] hCount;    //horizontal location of the pixel being requested
input [9:0] vCount;    //vertical location of the pixel being requested
input [10:0] x; //x coordinate of top left corner of cheney popup
input [9:0] y; //y coordinate of top left corner of cheney popup
input win; //1 if player killed duck this round

output hasPixel;    //asserted high if this object has a pixel at hCount, vCount
output [7:0] pixel;    //rgb value of pixel

reg hasPixel;
reg [7:0] pixel;

parameter popWidth=128;
parameter popHeight=96;

reg [13:0] addr;
wire [7:0] dout,dout2;

popuprom aPopupRom(addr,clk,dout);
popuprom2 aPopupRom2(addr,clk,dout2);
always @(posedge clk) begin
    if((hCount>=x) && (hCount<x+popWidth) && (vCount>=y) && (vCount<y+popHeight)) begin
        hasPixel<=1'b1;
        pixel<=(win ? dout2: dout);
        addr<=hCount-x+(vCount-y)*popWidth;
    end
    else begin
        hasPixel<=1'b0;
    end
end
endmodule
output done; //high when module finished
output calibrateMode; //high when in calibration mode

reg hasPixel, done;
reg [7:0] pixel;

reg [13:0] addr;
wire [7:0] dout;

reg calibrateMode, startPlayingMode;
titlerom aTitleRom(addr, clk, dout);

parameter screenWidth=1024;
parameter screenHeight=768;
parameter hBlack=8; //first and last 16 pixels are black

always @(posedge clk) begin
  if(reset || compoundReset) begin
    calibrateMode<=1'b0;
    startPlayingMode<=1'b0;
    done<=1'b0;
  end
  //latch calibrate and startPlaying
  if(calibrate) calibrateMode<=1'b1;
  if(startPlaying) startPlayingMode<=1'b1;

  if(calibrateMode) begin
    done<=1'b0;
    calibrateMode<=leaveCalibrate ? 1'b0 :1'b1;
  end
  else begin
    if(startPlayingMode) begin
      done<=1'b1;
    end
    else begin
      hasPixel<=1'b1;
      pixel<=((hCount<=hBlack)||(hCount>=screenWidth-
      hBlack))? 8'b0:dout; //divisions by 8 are due to compression of title
      addr<=hCount/8+((vCount/8)*(screenWidth/8));
    end
  end
endmodule
module delayN(clk, in, out);
    input clk;
    input in;
    output out;

    parameter NDELAY = 3;

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

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

endmodule // delayN