Augmented Reality System: Virtual Postcards

Jessica B. Barber and Andrew J. Meyer
Professor: Chris Terman
TA: Alessandro Yamhure

12 December 2007

Abstract

Augmented reality systems combine real world and computer generated data, outputting a version of reality that is "enhanced" in some way. The Virtual Postcard system takes in a live video image of a postcard that is blank in reality, determines the postcard's position and orientation, and "affixes" a predetermined image to the card. The result of this processing is that in the video output of the system, it appears that the image is printed on the card in reality; that is, the index card can be moved around as desired and the image will appear on the card with the appropriate rotation, skew, and resizing.

Contents

List of Figures

1 Overview

Virtual Postcard is an augmented reality system in which video of a blank index card, being moved and rotated by a user, is processed in order to "print" a virtual image onto the card. The system then outputs video to a screen in which the card appears to have a picture printed on it in real life. Although this problem may appear on the surface to be a simple one, there are actually many factors that must be considered when attempting to solve it.

On a most basic level, the system must somehow be able to identify the current position and orientation of the card from raw video data. Based on that information, it must then appropriately transform an image such that it "fits" on the card. A composite of the virtual image and the real video is generated and output to the screen. There are, of course, many different ways that these tasks could be accomplished. In implementing the Virtual Postcard system, different solutions were evaluated on the basis of ease of implementation, the minimization of computational overhead, and their ability to take advantage of the inherent way the FPGA interacts with video inputs and outputs. Through careful design, the Virtual Postcard system requires no frame buffering and exploits several mathematical tricks to shift computational overhead away from system bottlenecks.

2 Description

Implementation of the Virtual Postcard system involves two main tasks: vertex detection and image transformations. The vertex detection logic receives image data directly from the video camera, and from this data it is able to determine the x and y coordinates of each of the four corners (updating these coordinates once per frame as the index card moves). Once the vertices are located, their coordinates are passed to the image transformation logic, which uses this information to skew, rotate, and resize a preloaded image. Once the transformation is complete, the image data is combined with the video data and output to the screen.

Figure 1 is a block diagram indicating the pertinent modules in the system and the manner in which data flows through them.

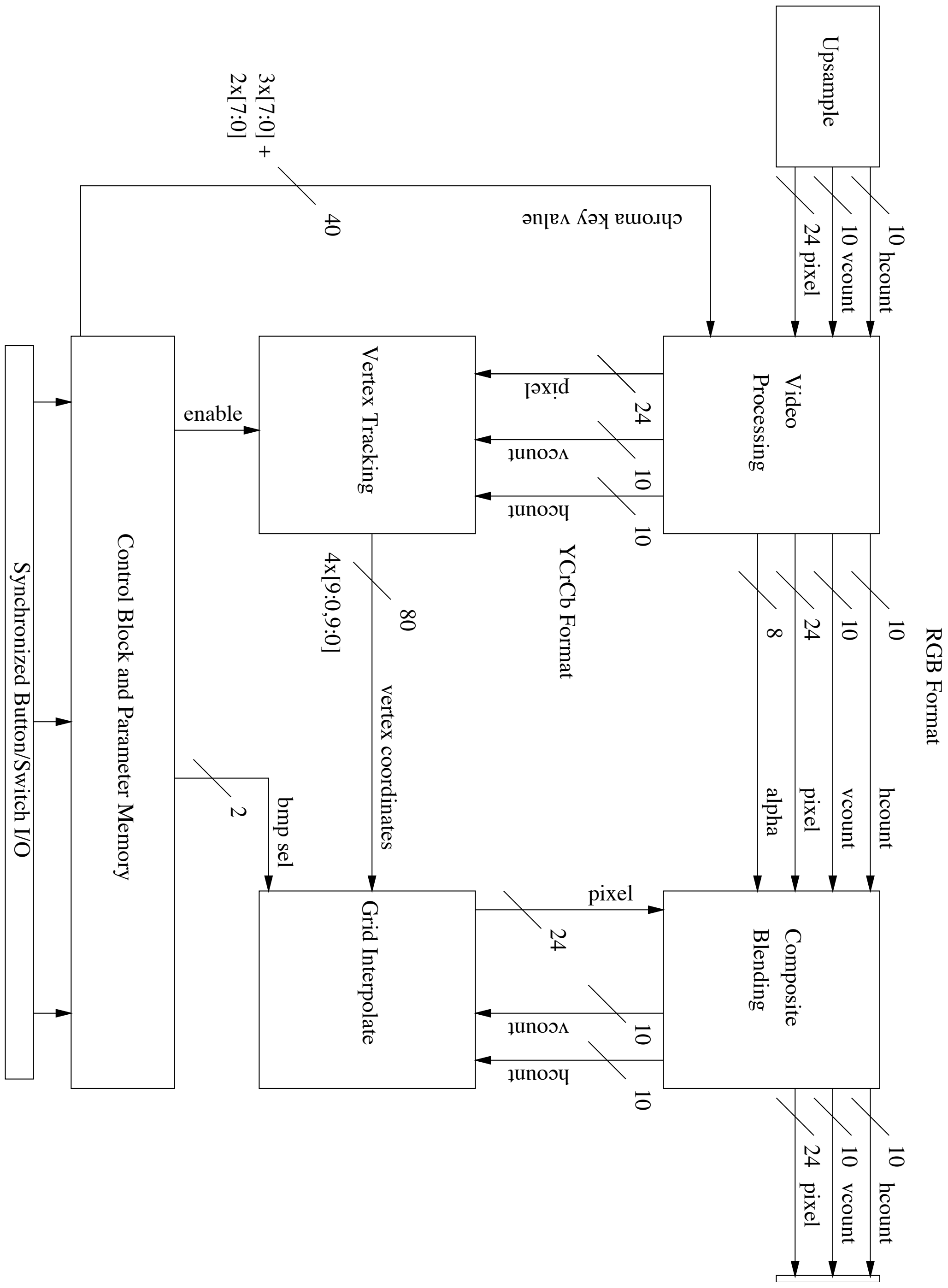

Figure 1: Block diagram of the overall system design.

2.1 The Vertex Output Module

Many augmented reality systems use a two-dimensional marker to help the system determine the relative orientation of the video camera and the surface onto which the image/figure is being projected. The Virtual Postcard system, however, requires only that the postcard be a single solid color. The vertex detection logic can then find the four vertices by locating the boundaries of this block. Once the four corners have been detected, some additional logic is required to keep track of their locations through rotations of the card. This is accomplished by running a continuous check to see if the card has been oriented horizontally or vertically relative to the camera (at which point the boundary conditions break down), and by maintaining a finite state machine that keeps track of the card's possible current position given its previous state.

2.1.1 Color Detection

The current implementation of the Virtual Postcard system requires that the index card in use be a solid red color. The color detection logic essentially has two parts. First, every pixel is checked to determine whether its YCrCb values fall within the determined parameters for "red". These parameters were strictly defined in order to detect only "true red", eliminating most false detection of red-hued items such as skin. If the current pixel's YCrCb values fall within these constraints, the flag *is_red* is set high.

The second part of color detection refines the search for red by setting the flag *yes_red* high only when four consecutive pixels are marked as *is_red*. This filters noise out of the image, improving the probability that a pixel cluster will be declared as red only if it actually falls within the boundaries of the index card.

2.1.2 Vertex Detection

The initial vertex detection assumes that the index card is /emphnot oriented such that it is perfectly horizontal of vertical with respect to the camera. This is important, because the initial positions of the four corners are found by locating the four "boundary" clusters of red pixels; that is, the red pixel cluster with the highest x coordinate, with the lowest x coordinate, the highest y coordinate, and the lowest y coordinate. These four pixel clusters will necessarily represent the locations of each of the four corners.

These pixel clusters are found as follows: image data is fed in from the video camera until *yes_red* goes high for the first time. At this point, the x and y coordinates of the current pixel are set as the definitive location of the top most pixel, and as the reference locations for the left most, right most, and bottom most pixels. For the rest of the frame, every time *yes_red* is high, the current x and y coordinates will be checked against the current values of the left most, right most, and bottom most pixels. If, for example, the x coordinate of the current pixel has a higher value than the stored position of the right most pixel, the location of the right most pixel will be replaced with the current values. Left most and bottom most are similarly updated. Once the end of the frame is reached, the definitive locations of all four corner pixels have been found.

2.1.3 Horizontal/Vertical Position Check

When the index card enters a position that does not have clear top, bottom, left, and right most corners, that is, a position that is horizontal or vertical with respect to the camera, the original method of corner detection breaks down. Being in a horizontal or vertical position means, for example that the the system is equally likely to detect any one of the pixels on the top edge of the card as the top most corner, since all of these pixels will have the same y value. To remedy this, the system performs a check to see if the card is in one of these positions in each frame.

To determine if the card is in a horizontal or vertical position, the system checks to see how many red pixels there are with the same x coordinates as the left most and right most pixels and the same y coordinates as the top most and bottom most pixels found by the vertex detection logic. If the number of pixels found is above a certain threshold, the card is marked as "horizontal" for this frame.

Knowing that the card is in a horizontal/vertical position, it is now possible to assign coordinates for all four corners, based on some assumptions. For example, although the coordinates found by the vertex detection logic for the "top most corner" do not necessarily represent the position of a corner, the y coordinate found does accurately represent the position of the top edge of the card, and so both the top right and top left corners should have this y coordinate. Similarly, the x coordinate found for the

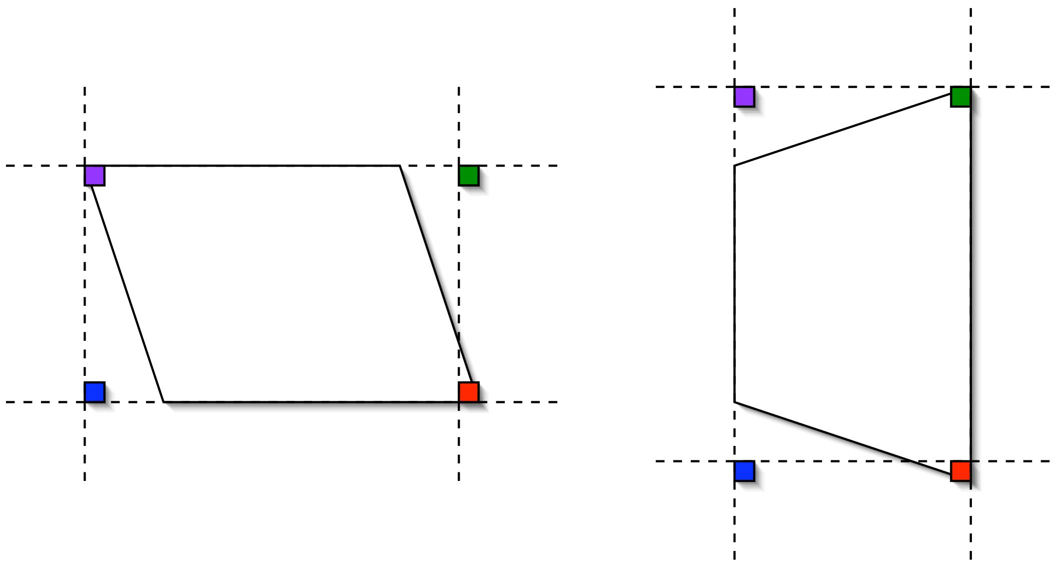

Figure 2: Examples of horizontal/vertical contingency failure. These cards are in positions that will be flagged as horizontal/vertical but are not actually at 90° angles, so the corner fitting is slightly inaccurate. However, the effect on the output should be negligible.

"left most corner" will accurately represent the left edge of the card, so the top left and bottom left corners should have this x coordinate. This logic can be applied to find all four corners of the card.

A note: it is possible that a card will be detected as being in the horizontal/vertical position when it is in a configuration such that the corner location technique as described above does not yield an accurate approximation. Two examples of this possibility are shown in Figure 2. However, since the final output only replaces the original video data with the transformed image data at pixel clusters that are marked as *yes_red*, the image will still appear to "fit" on the card, even though the image transformation will not occur with perfect accuracy.

2.1.4 The Vertex Tracking Finite State Machine

The majority of the work in the vertex output module is done by the vertex tracking finite state machine. This FSM operates under the assumption that, given a particular starting orientation of the card, it can move into two other possible positions, depending on whether the card is rotated right or left. By keeping track of what rotations have happened, and using information from the vertex detection logic and the horizontal/vertical check logic, the vertex tracking finite state machine can keep track of which corner is which throughout arbitrarily many rotations.

Figure 3 shows the eight possible positions through which the index card can cycle. The system assumes that the index card will begin in state/position one; that is, with the bottom right corner as the lowest corner, the top right corner as the right most corner, etc. When the system is in use, the index card will appear on the screen with colored markers affixed to each of the four corners; the user verifies that the card is in the proper position and that the system has accurately located all four corners. Once this has been verified, the user initializes the vertex tracking FSM by pressing a button.

After the user initializes the FSM, it remains in state one until the horizontal/vertical check goes high. Until the horizontal/vertical check goes high, the card can make arbitrary small movements but the positions of the four corners can still be detected by the original vertex detection logic: the top most pixel cluster will still represent the top left corner, the right most pixel cluster will still represent the top right corner, and so on. It is impossible for the card to enter a position where this is no longer true without first passing through a horizontal/vertical state.

Once the horizontal/vertical check *does* go high, the system knows that the card must have entered either state eight or state two. In can determine which state the card has entered by checking the distances between the maximum and minimum x coordinates and between the maximum and minimum y coordinates. If the distance between the y max and y min is smaller, the card has entered state eight; if the distance between x max and x min is smaller, the card has entered state two. The system knows where the four corners should be as described in the horizontal/vertical check logic, and it knows which corner is which according to the current state.

When the card is transitioning out of a horizontal/vertical state, a similar situation occurs. For example, assume the card has been found to be in state two. If the horizontal/vertical check goes low, the system knows the card has either returned to state one by rotating counterclockwise, or proceeded on to state three by rotating clockwise. The check to determine which state it has actually entered is simple. The y coordinates of the leftmost and rightmost pixel clusters are compared: if the left most cluster is higher, the card has reverted to state one, and if the right most pixel cluster is higher, the card is in state three. Corner labels may be assigned accordingly.

2.2 Image Interpolation and Output

This section will describe how a fixed resolution image stored in a Read Only Memory is transformed appropriately to fit the scale, rotation, and skew of the target index card. Most commonly, these types of geometric transformations are performed by large matrix multiplication operations. A two dimensional image can be considered a large matrix. Rotating, scaling, and skewing the image can then be achieved by multiplying this matrix by special transformation matrices to achieve the desired configuration of the final image. Unfortunately, such a large matrix operation is difficult to implement in hardware, requiring a more unique solution to the problem. One pixel by pixel alternative was devised which roughly approximates the effect of standard matrix based image transformations. This operation requires two adds, two multiplies, and one divide per clock cycle, a large improvement over a matrix based approach.

In addition, this section will describe how the system generates a composite video feed to output to the screen. To do this, the transformed image must somehow be combined with the original video input. One simple technique to accomplish this is to isolate the index card using color segmentation and merely replace pixels attributed to the card with the output of the interpolation module. While this method was utilized in the final system, several potential improvements to this final stage in our system will be briefly reviewed as well.

A block diagram of this portion of the system can be seen in Figure 4.

2.2.1 Storing Images on ROM

The ISE Verilog programming environment provided by Xilinx includes many tools to aid in the construction of gate level modules. Using this utility, large on-chip ROM's were generated to store two small color bitmap images. This approach is hardly ideal, however the alternatives would have required massive complications to be added to

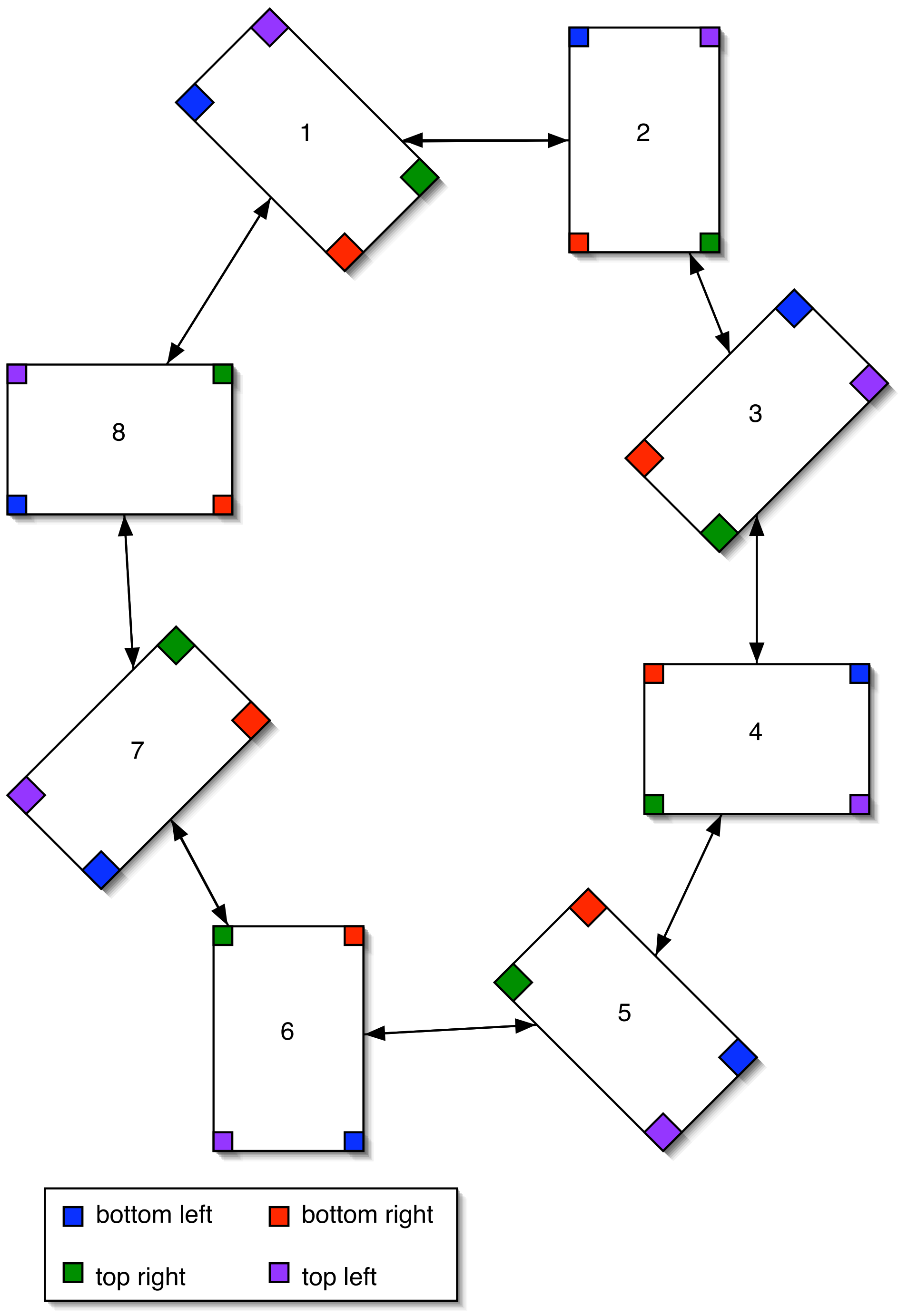

Figure 3: A state transition diagram for the vertex tracking finite state machine. At each state, the card has the possibility of entering into one of two potential states depending on whether it is rotated clockwise or counterclockwise.

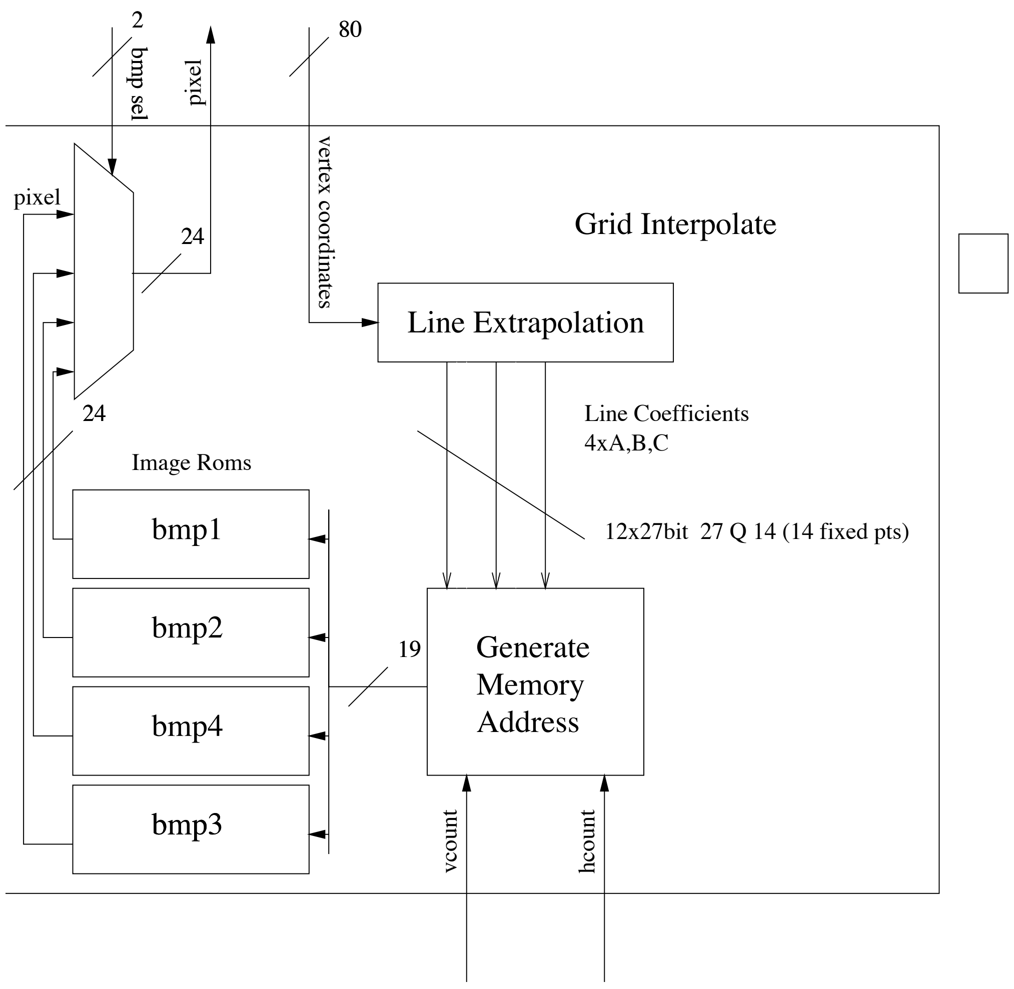

Figure 4: This figure gives a high level description of the image interpolation portion of the system.

the design. For example, a FLASH ROM could have allowed several large resolution bitmaps to be stored at the expense of adding convoluted interface modules to write to the FLASH ROM. The front end of the FLASH ROM chip was designed for access by microprocessors and requires long handshaking protocols to be observed leading up to and following data transfer. These complications strongly encouraged the use of smaller on-chip ROMS.

In order to create an on-chip ROM, image data must first be converted into the appropriate format and stored in a .COE file. This file includes a short header describing the data type described by the file and then a matrix of hexadecimal numbers which represent the RGB pixel values of an image. The .COE file is entirely ASCII based, which means that binary image data must be converted to text numerals in order to utilize the Xilinx ROM generator. To do this, a small matlab script was written to load a color image, write the .COE header, and write a matrix of RGB pixel values to the file in hexadecimal text. Pixel values are represented as 24 bit numbers, which can be divided into three 8 bit red, green, and blue pixel intensity values. The final system utilized only one image ROM which contained a 300x200 pixel image of the actress Zooey Deschanel, as seen in Figure 5. The picture was an attempt to best approximate the girl-hat image found in many modern publications related to image processing.

Once a .COE file is successfully generated from the raw bitmap, Xilinx generates a simple module which accepts a one dimensional address and returns a 24 bit pixel value one cycle later. In order to generate the address of the pixel at location x,y, simply add the x value to $x + xRes * y$ the width of the image times the desired y value). Overflow is avoided by never passing values into the image ROM beyond the size of the image stored in it.

2.2.2 Generating Vector Lines from Corners

A rectangular card can assume an awful lot of two dimensional orientations when viewed from different angles. Perspective can cause one side to appear shorter than the other, rotations and skew can destroy the intrinsic right angles of the card, and in general knowledge that the card is actually a rectangle is of little use. Because of this, an appropriate interpolation scheme must be capable of transforming a rectangular image into the shape of any quadrilateral. The corner detection generates four vertex

Figure 5: Zooey Deschanel, the popular actress, stored in an on-chip ROM.

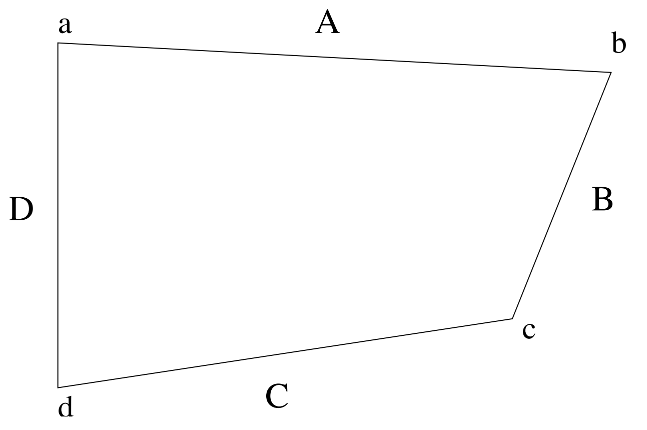

Figure 6: The names of the corners of this quadrilateral are maintained regardless of its orientation.

locations. The interpolation module requires only that they form a convex shape (a line drawn through the shape will only intersect two of the sides). While more restrictive requirements could likely be determined, the general approach to non-rectangular image interpolation taken here gives enough room so that any orientation of the index card should yield an appropriately transformed image.

The first step towards fitting the image onto on arbitrary quadrilateral is describing the quadrilateral by its sides. This is the role of the module combinationalAbc. The original design for this module did not take into consideration the exact method for recombining the transformed image and the video feed, so it was also necessary to devise a method for determining whether a given pixel lies inside or outside of the quadrilateral. These two roles yielded a natural design for combinationalAbc as a simple vector arithmetic unit.

As visible in the figure, the index card is described by four labeled corners, A, B, C, and D. Four vectors can be generated to describe the sides of the card by computing $\overline{BA}$, $\overline{CB}$, $\overline{DC}$, and $\overline{AD}$. It is essential that consistency is maintained in the direction of these vectors (for example, $\overline{BA}$ and $\overline{BC}$ would not work) since we can then call the "inside" of the quadrilateral as the either the negative or positive side of all four vectors. An appropriately labeled quadrilateral can be seen in Figure 6.

To generate a line from a vector the general vector form of a line can be used,

$Ax+By+C=0$. For any line, there are multiple choices for A, B, and C. That is, this form is not a unique descriptor. However, this form is an extremely convenient choice of a representation of a line since the equation $\frac{Ax_0+By_0+C}{\sqrt{A^2+B^2}}$ represents the distance of a point to that line, an operation that will need to be performed once per pixel clock in the pixelInterpolate module. To save computation effort, and since the $Ax+By+C$ line equation is not unique, $\sqrt{A^2+B^2}$ can be chosen to equal one. This means that extra work can be done in normalizing the line equations here in generateAbc to avoid performing an extra divide at every pixel clock. Refer to the verilog for the betterPixelInterpolate module in the appendix for a mathematical demonstration of how the module generates a line equation for a given set of two points.

Vertex detection logic only generates new corner estimates once a frame, thus line equations only need be generated once per frame. This quality makes it desirable to include as much of the computational lifting into this module as possible. Over one million clock cycles pass while displaying a single frame. Therefore there is little reason to pipeline or otherwise attempt to increase the throughput of the generate-Abc since one million clock cycles is far greater than needed to perform the necessary computations. By implementing this module as an entirely combinational block the design can stay streamlined and simple. Once the registered values for the vertices are passed in, a timer begins and the combinational circuit which generates the line equation is allowed to mature naturally. A generous 40 cycles later, a ready signal goes high and the system stores the values A, B, and C for a given line into registers. Rather than attempt to compute the equation for each of the lines bounding the quadrilateral in succession, four instantiations of combinationalAbc allow for each line to be computed in parallel. This further simplifies the design of the overall system.

2.2.3 Pixel Interpolation

Conventionally, geometric image transformations are performed using large matrix multiply operations. Given a parameter for rotation, skew, or scale it is relatively straightforward to take an image matrix and multiply it appropriately to generate a transformed image with the correct geometric properties. However, not only is it logistically undesirable to perform such an operation in hardware but in this case it is also unnecessary. Matrix based image transformations are intended to implement the actual geometry of rotation, scale, etc. In this case, possession of four labeled vertices suffices to perform the transformation much more elegantly. In fact, the geometric transformations are happening in "the real world", as captured by the vertex detection logic. Thus, the use of matrix multiplication methods is not required. This insight was not evident prima facie, rather a careful attempt to derive a simpler interpolation technique proved extremely successful.

Given line equations for the four bounds of a convex quadrilateral the distance from each line is computed. Call these distances, d_a, d_b, d_c, $d_d$where 'd_a' indicates the distance from line A (as shown in the figure) and so on. From these, the x,y coordinates of the desired pixel from the image ROM is determined by the following formulas ($xmax$ and $ymax$ are the resolution of the image stored in ROM):

$$x = \frac{d_d * xmax}{(d_a + d_b)}$$
$$y = d_a * ymax(d_a + d_c)$$

The form of these equations is quite intuitive. Consider the case where the current hcount/vcount of the vga scan lies on the line D of the quadrilateral. Then da equals zero and x also equals zero. This reasonable since line D of the quadrilateral corresponds to the left edge ($x = 0$) of the stored image in ROM. Similarly if the current hcount/vcount lies on line B, then $x = xmax$, which is reasonable since the right edge of the image is desired. Figure 7 demonstrates an example of this kind of interpolation at work. Interestingly, this technique avoids the problem of resolution up or down sampling. If quadrilateral is smaller than the stored image than this equation will automatically skip certain pixel addresses. If this quadrilateral is larger, certain pixel addresses will automatically be repeated. To avoid all interpolation related artifacts it is desirable to store overlay images with resolutions higher than the size of the entire display. This way, no pixels are ever repeated. Unfortunately, storing the images using the on-chip ROM's disallows this technique.

This computation requires two additions and two multiplications for each da-dd and one multiply, one divide, and one addition to compute the actual x or y coordinate to be input into the final ROM. Furthermore, this computation needs to be performed with a throughput of a one result per pixel clock. This requirement makes a combinational approach (as with combinationalAbc) infeasible. Because of this, the entire pixelInterpolation module has been pipelined. By pipelining this module a latency

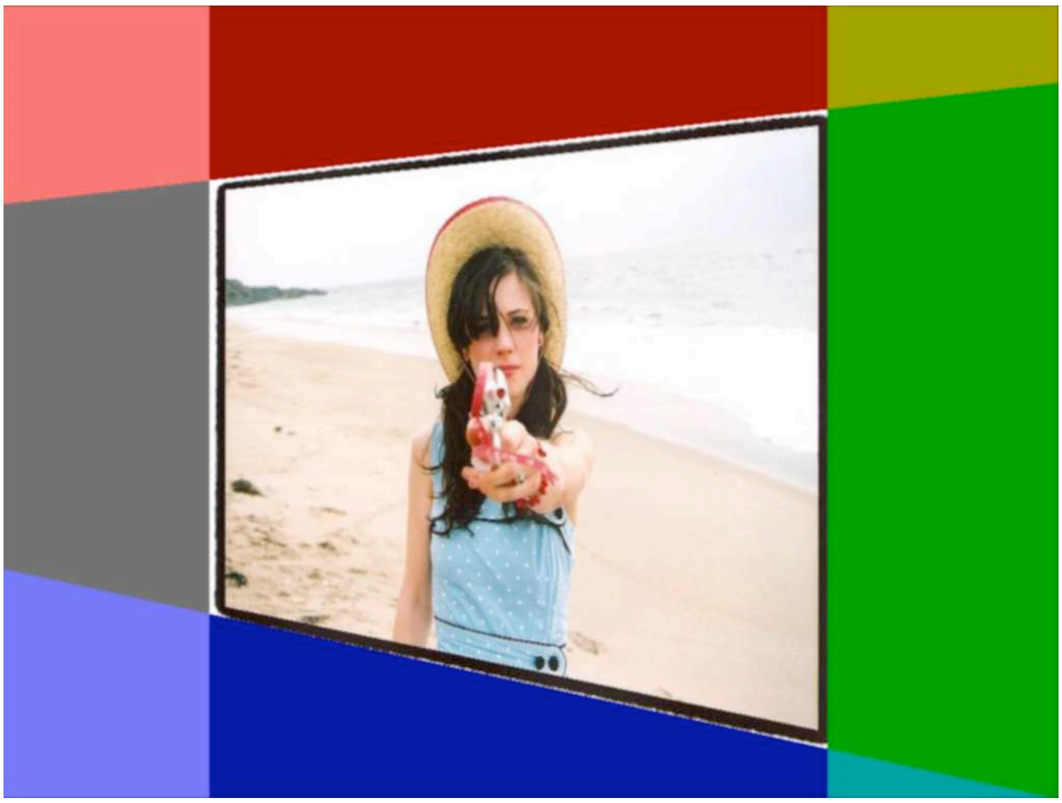

Figure 7: This image, generated by a MATLAB simulation, demonstrates the pixel by pixel geometric interpolation technique.

of three cycles is introduced. While this latency could be corrected by delaying the entire frame by three pixels as it passes through to the display, it is unnecessary. Failing to account for the delay amounts to shifting the interpolated image by three pixels when its overlay ed onto the index card. This is totally unnoticeable.

It is important to note the necessity for a form a decimal notation for the intermediate values used in this processing chain. For example, consider the computation $\frac{d_a * xmax}{(d_a + d_c)}$. This equation can be thought of as scaling the maximum x address of the ROM (xmax) by some number less than one $\frac{d_a}{d_a + d_c}$. We are guaranteed that this scale is some number between zero and one inclusive since da and dc are both positive. To represent numbers that are partially or entirely fractional, a fixed point arithmetic scheme was used. This scheme is described in the next section.

This module is the only point of failure for the system. While this technique works in simulation (the functionality of all of the interpolation techniques were verified in MATLAB) it yielded errors in practice which could not be corrected. After substantial attempts to debug the problem, the finished system fails to interpolate the image properly. Visually, the interpolated image is usually either a gridded version of the image stored on ROM or small region of the image upsampled to consume the entire face of the index card. Furthermore, the system often switches between the two modes of error (gridding or zooming) indicating that the error is likely related to bit overflow of an intermediate value in the processing chain. This error can clearly be seen in Figure 8 of the entire system at work. Encouragingly, the interpolation module does properly scale, rotate, and skew the image appropriately (albeit a gridded or otherwise distorted form of the image). There are several potential sources for the failure and several reasons for why it proved so difficult to track down. These issues will be discussed in the following section.

2.2.4 Fixed Point Arithmetic

In the course of the computations described above, any of the divide operations could yield a number between zero and one or a number with a fractional component. In some cases, these fractions are essential and can not be rounded off. Therefore it was necessary to implement some kind of fixed point arithmetic. In retrospect, it may have been more effective to utilize floating point arithmetic techniques. In floating

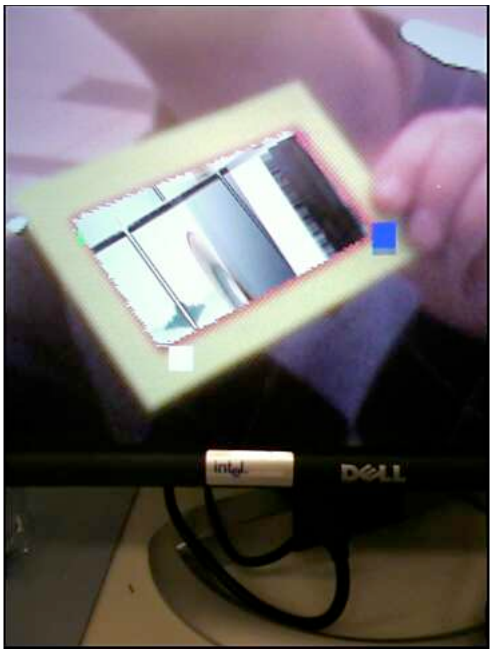

Figure 8: A photo of the final system image showing interpolation error. The grid interpolation module successfully maintains rotations and scaling, but fails to properly fit the image on to the card. This is likely due to an overflow of an intermediate value in the processing chain.

point arithmetic, numbers are defined by a mantissa (integer base) and an exponent (some scaling integer in the form of an exponential). This representation is desirable since the number of bits in any number remains fixed regardless of if the number is a fraction or an integer. Unfortunately, the mantissa exponent format is extremely unintuitive and would introduce significant complications to the debugging process.

Fixed point arithmetic, on the other hand, simply represents all numbers as integers. It is up to the programmer or some fixed point data type to store the location of the decimal point. In decimal notation, the number 10.5 could be represented as a 3Q2 fixed point number. This means that the entire number is 3 digits, and the decimal is after the second digit. So to represent the number .678, simply record the number 678 and store the fact the decimal comes before the first digit. In decimal format, its clear that converting a fraction to and from an integer entails multiplying by 10 to some power (depending on the location of the decimal). In binary, however, a fraction is represented by an integer multiplied by some power of two. Accordingly, numbers BEFORE the fixed point correspond to positive powers of two (regular binary integers) and numbers AFTER the fixed point correspond to negative powers of two: ... 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} ... In this scheme, a 14 bit number could represent a minimum of 2^{-14} and a maximum of 2^{14}. This system also ports easily into twos complement notation. In this notation an integer includes a sign bit as the high order bit. In an N bit number, the sign bit actually represents $-1 * 2^N$. 1001, for example, corresponds to the decimal number -7 $(-8 + 1)$. Twos complement notation works equally well with fixed point notation as long as the fixed point sits at least one bit below the sign bit.

This restriction is likely the source of the overflow error observed in the pixel interpolation procedure. If any of the operations in the arithmetic chain produced a number whose sign bit is to be considered fractional, then that number has overflowed and will yield an error. Although firm attempts were made to isolate which intermediate value had overflowed, the debugging procedure was extremely difficult for reasons to be discussed later on. For more information on the mechanics of fixed point arithmetic, refer to the comments included in the verilog code provided for combinationalAbc and betterPixelInterpolate modules.

2.2.5 Chroma-Key blending and Alternatives

At the output of the system, the result of the image interpolation and the original video input must be combined. To do this, the result of the color detection (used as part of the vertex detection logic) can be used. If the index card is composed of a solid, high contrast color a chroma-key mask can be generated which contains the result of a simple threshold test on each pixel. If the pixel is sufficiently “red”, for example, a binary flag is raised and that pixel is considered to me a member of the card. In the simple scheme utilized by the Virtual Postcard system, this flag acts as the selector for a mux which chooses between pixels from the interpolation module or pixels from the video feed.

To avoid storing this image mask in a frame buffer, pixels from the current frame are combined directly with the current output of the pixel interpolation module. This means that the image being overlayed onto the index card is actually appropriately transformed for the PREVIOUS frame of video, not the current one. This is due to the fact that it takes an entire frame to determine the vertex locations and generate the line equations used by pixel interpolation. This one frame mismatch is unnoticeable by the user since the human eye cannot perceive this discrepancy at sixty frames per second.

Several interesting alternatives to this simple combination technique were proposed although eventually discarded due to time constraints. The first of these were to perform alpha blending instead of binary chroma key combination. In alpha blending either the image stored in ROM or the color detection scheme produces a trasparency value between zero and one. This number could be the euclidian distance between the current pixel value and “red” or it could be a representation of the transparency of each pixel in the image ROM. To generate the composite image, each pixel is $\alpha * pixelA + (1 - \alpha) * pixelB$. This blending would have produced a significantly smoother transition between pixels from the video feed and pixels from the image interpolation. In this scheme, only one two pixel sources (image or video) include alpha values, however. If this were not the case, it would be possible for both pixels to be transparent and the resultant output pixel to be black! To avoid this problem the image ROM could have included binary transparency, while the video pixels included alpha blending values. In the image from ROM is not transparent at a given pixel, alpha blending would have been performed. Otherwise, the video pixel would have passed straight through to the output. This would have produced the effect of images being "printed" onto parts of the card rather than the transformed image covering the entire card as is the case in the final system.

3 Testing and Debugging

3.1 Vertex Output Module

The Vertex Output module was designed to be as straightforward to implement as possible, and so most "debugging" issues centered around tweaking various parameters for maximum efficiency. The two most delicate parameters that had to be set were the threshold values for labeling a pixel as red, and the minimum number of red pixels with the same x or y coordinates that had to be found before the horizontal/vertical check went high. The "red" threshold was determined by displaying the YCrCb value for a predetermined pixel on the 16-bit hex display, and then making sure this pixel was occupied by the index card. In this way the index card's color could be directly "sampled". The horizontal/vertical check parameters were originally made adjustable with the user input switches; once a value that yielded the desired results was found the adjustability was removed and that parameter was set permanently.

Since most of the work in the Vertex Output Module is done by a pair of finite state machines, the remainder of the debugging was accomplished by simply outputting the current state of these two machines to the 16-bit hex display and verifying that they were transitioning between states as desired.

3.2 Pixel Interpolation Module

The betterPixelInterpolate module was the only point of failure for the Virtual Postcard system. Therefore, a significant amount of debugging efforts were devoted to it. In addition, the combinationalAbc module returned three fixed point values which needed to be verified as valid in order to proceed with implementing the system. Unfortunately, combinationalAbc is instantated four times that include two multiplies, a divide, and one square root each. Each of the two betterPixelInterpolate modules include 5 multipliers and 1 divide. Each of these operations are performed with 27 bits of precision and add large amounts of compile time to route and place these large arithmetic blocks. Once added to the system, the compile time for the two modules often exceeded 30 minutes. because of this, it was essential to devise efficient debugging practices. During development each of these modules had multiple switches as input and 64 bits of debug output to be sent to the logic analyzer. In this way, individual values intermediate to the processing pipeline could be verified. While the logic analyzer supported binary, decimal, hexidecimal, and signed hexidecimal values it did not include fixed point capabilities. To verify fixed point values, the integer result printed on the logic analyzer was multiplied by the appropriate exponent of two to generate the decimal value. A MATLAB simulation of both modules was seeded with some fixed set of vertices and used to verify the values for each of the intermediate values in either combinationalAbc or pixelInterpolate.

The logic analyzer was not particularly helpful in isolating the source of the error in betterPixelInterpolate. In addition, the large reliance on Xilinx Coregen arithmetic blocks precluded the use of ModelSim simulation which does not support certain mathematical modules. A careful review of the code and the warnings provided by the Xilinx compilers yielded a few potential problems though. Xilinx reported that certain signed numbers with leading zeros would never use the high order bits based on the operations being performed. In response, the compiler appeared to set these high order digits to a constant zero. This optimization is undesirable since these numbers could take negative values, which would require sign extending ones into the higher order bits. It is also likely that a manually maintaining the locations of the fixed point was ineffective. While the operations were traced out carefully in terms of how many decimals points were being applied to each number, it is very possible that time constraints prevented a small mistake to be overlooked. Several experiments were devised in order to test various assumptions about the location of the fixed point in the final computation (interpolated x, y addresses to access the ROM) but these proved unsuccessful.

Debugging combinationalAbc was far more fruitful, but still not entirely complete. The module drawLine was written in order to graphically represent the line equations generated by the module. Strangely, the positive and negative sides of these lines (since they are actually vector based) were not consistent. Many anomalies were observed where positive and negative sides of a line would switch periodically while being displayed to the screen. This is indicative of a sign bit overflow. Widening each of the intermediate values in the drawLine module appeared to fix the problem, but time constraints precluded this from being throughly explored.

4 Conclusion

The Virtual Postcard system is designed to maximize the use of an FPGA based implementation. This means that care has been taken to parallelize computations, favor a pixel by pixel approach, and generate a high resolution, high framerate output. While the system did not operate 100% as designed, many of the original design goals were attained. Corner detection and tracking is elegantly implemented using pixel by pixel logic and a nine state FSM. The system also successfully demonstrated an image transformation method which does not require matrix algebra and performs rotations, scales, and skew at 60 frames per second.

Despite these successful elements, certain technical challenges reamain for a more advanced instantiation of the system. Color and vertex detection logic would benefit from some kind of filtering to reduce glitches and generate smoother corner approximations. The fixed point arithmetic used in the interpolation modules could have avoided bugs by using a fixed point data type that includes the location of the decimal, rather than implicit decimal tracking by the programmer. A later generation of the Virtual Postcard system would likely benefit from a floating point arithmetic unit which inherently avoids the problems encountered like sign bit overflow and poor decimal tracking.

The system described herein is a substantial first step towards an advanced augmented reality program implemented in FPGA hardware. Unfortunately time constraints and unforeseen challenges meant that only the base design was implemented. However, two of the three core modules (vertex tracking and line extrapolation) were remarkably effective and the third module showed significant promise for a novel image interpolation scheme.

A Virtual Postcard Verilog Code

A.1 Toplevel Labkit Module

```
///////////////////////////////////////////////////////////////////////////////
// 6.111 Final Project, Fall 2007
// labkit.v code modified from its original versions
// (credited below) by Jessica Barber and AJ Meyer
//////////////////////////////////////////////////////////////////////////////
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
```

```
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
```

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

module labkit(beep, audio_reset_b,
       ac97_sdata_out, ac97_sdata_in, ac97_synch,
       ac97_bit_clock,

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

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

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

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

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

       clock_feedback_out, clock_feedback_in,

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

       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

       mouse_clock, mouse_data, keyboard_clock, keyboard_data,
       clock_27mhz, clock1, clock2,

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

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

       switch,

       led,

       user1, user2, user3, user4,

       daughtercard,

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

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

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

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

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

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

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

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

 input  clock_feedback_in;
 output clock_feedback_out;

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

 output rs232_txd, rs232_rts;
 input  rs232_rxd, rs232_cts;

 input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

 input  clock_27mhz, clock1, clock2;

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

```
   output  disp_data_out;

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

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

   inout [43:0] daughtercard;

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

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

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

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input
   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

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

   // SRAMs

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

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

```
/* enable RAM pins */

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

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

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

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

   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

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

   ////////////////////////////////////////////////////////////////////////////
   // Demonstration of ZBT RAM as video memory
   ///////////////////////////////////////////////////////////////////////////

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

   wire clk = clock_65mhz;

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

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

 // DEBOUNCE other signals
 wire up,down,left,right;

 debounce db2(reset, clock_65mhz, ~button_left,  left );
 debounce db3(reset, clock_65mhz, ~button_right, right );
 debounce db4(reset, clock_65mhz, ~button_up,  up );
 debounce db5(reset, clock_65mhz, ~button_down,  down );

 wire db_button0, db_button1, db_button2, db_button3;

 debounce b0db(power_on_reset, clock_65mhz, ~button0, db_button0);
 debounce b1db(power_on_reset, clock_65mhz, ~button1, db_button1);
 debounce b2db(power_on_reset, clock_65mhz, ~button2, db_button2);
 debounce b3db(power_on_reset, clock_65mhz, ~button3, db_button3);

 // display module for debugging

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

 // generate basic XVGA video signals
 wire [10:0] hcount;
 wire [9:0]  vcount;
 wire hsync,vsync,blank;
 xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

 // wire up to ZBT ram
```

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

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 [17:0] vr_pixel;
wire [18:0] vram_addr1;

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

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

wire [29:0] ycrcb; // video data (luminance, chrominance)
wire [2:0] fvh; // sync for field, vertical, horizontal
wire       dv; // data valid

ntsc_decode decode(.clk(tv_in_line_clock1), .reset(reset),
    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
    .ycrcb(ycrcb), .f(fvh[2]),
    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

// code to write NTSC data to video memory

wire [18:0] ntsc_addr;
```

```
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb[29:0],
    ntsc_addr, ntsc_data, ntsc_we, db_button2);

   // code to write pattern to ZBT memory
   reg [31:0]  count;
   always @(posedge clk) count <= reset ? 0 : count + 1;

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

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

   wire  sw_ntsc = ~db_button3;
   wire  my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
   wire [18:0]  write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]  write_data = sw_ntsc ? ntsc_data : vpat;

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

   // select output pixel data

   reg [17:0]  pixel;
   wire  b,hs,vs;

   delayN dn1(clk,hsync,hs); // delay by 3 cycles to sync with ZBT read
   delayN dn2(clk,vsync,vs);
   delayN dn3(clk,blank,b);

   always @(posedge clk)
   begin
pixel <= vr_pixel;
```

```
   end

  // vga output, clocked at ~65mhz to meet timing contsraints
  wire horiz,vert,red_yes;
  wire [11:0] tlh_pos,tlv_pos,trh_pos,trv_pos,blh_pos,blv_pos,brh_pos,brv_pos;
  reg [7:0] red,green,blue;
  wire [7:0] r,gr,bl;
  wire [3:0] state;

  assign vga_out_pixel_clock = ~clock_65mhz;

  vertex_output vo(clock_65mhz,reset,db_button0,switch[3],
1'b1,~b,hcount,vcount,hs,vs,
{pixel[17:12],2'b00},{pixel[11:6],2'b00},{pixel[5:0],2'b00},
r,gr,bl,
vga_out_sync_b,vga_out_blank_b,vga_out_hsync,vga_out_vsync,
tlh_pos,tlv_pos,trh_pos,trv_pos,blh_pos,blv_pos,brh_pos,brv_pos,
state,horiz,vert,red_yes);


  // Instantiate Application Modules

  // combinationalAbc, take corners, generate Line equations
reg [11:0] A1,B1,C1,D1,A2,B2,C2,D2;
always @ (posedge clk) begin
A1 <= tlh_pos;
A2 <= tlv_pos;
B1 <= trh_pos;
B2 <= trv_pos;
C1 <= brh_pos;
C2 <= brv_pos;
D1 <= blh_pos;
D2 <= blv_pos;
end
```

```
wire [15:0] analyzerOut1a, analyzerOut2a;
wire [15:0] analyzerOut1b, analyzerOut2b;
wire [15:0] analyzerOut1c, analyzerOut2c;
wire [15:0] analyzerOut1d, analyzerOut2d;

wire [7:0] counta,countb,countc,countd;

wire signed [15:0] Aa,Ba,Ca,Ab,Bb,Cb,Ac,Bc,Cc,Ad,Bd,Cd;
wire rdya,rdyb,rdyc,rdyd;
wire [12:0] sqva;
wire [15:0] nva1,nva1_frac;

// handshake with start and ready (for lining in multiple
// ABC computations
combinationalAbc Abca(clock_65mhz,switch[1],up,
A1,A2,B1,B2, analyzerOut1a,
analyzerOut2a,Aa,Ba,Ca,
start,rdya,counta);
combinationalAbc Abcb(clock_65mhz,switch[1],up,
B1,B2,C1,C2, analyzerOut1b,
analyzerOut2b,Ab,Bb,Cb,
start,rdyb,countb);
combinationalAbc Abcc(clock_65mhz,switch[1],up,
C1,C2,D1,D2, analyzerOut1c,
analyzerOut2c,Ac,Bc,Cc,
start,rdyc,countc);
combinationalAbc Abcd(clock_65mhz,switch[1],up,
D1,D2,A1,A2, analyzerOut1d,
analyzerOut2d,Ad,Bd,Cd,
start,rdyd,countd);
// ///////////////////////////////////////////////////////

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

// set the resolutions of our two test images
// Im being pretty generous with the widths in case
// we use the flash rom for large images in the future
//wire [10:0] flowersResX, flowersResY;
wire [10:0] zooeyResX, zooeyResY;

//assign flowersResX = 11'd320;
//assign flowersResY = 11'd240;

assign zooeyResX   = 11'd300;
assign zooeyResY   = 11'd201;

/////////////////////////////////////////////////////////////////////
// draw the lines

wire [23:0] lpixela;
wire [23:0] lpixelb, lpixelc, lpixeld;
wire signA,signB,signC,signD;

drawLine drawlineA(clock_65mhz,hcount,vcount,Aa,Ba,Ca,{2'b00},lpixela,signA,switch
drawLine drawlineB(clock_65mhz,hcount,vcount,Ab,Bb,Cb,{2'b01},lpixelb,signB,switch
drawLine drawlineC(clock_65mhz,hcount,vcount,Ac,Bc,Cc,{2'b10},lpixelc,signC,switch
drawLine drawlineD(clock_65mhz,hcount,vcount,Ad,Bd,Cd,{2'b11},lpixeld,signD,switch

////////////////////////////////////////////////////
// generate the interpolation address

wire [25:0] outAddrX1, outAddrY1, outAddrX2, outAddrY2;
wire [15:0] outAddrX, outAddrY;
wire [26:0] sumAxBy1a, sumAxBy2a, sumAxBy1b, sumAxBy2b;
wire [10:0] Xmax, Ymax;

// set static lines for debugging
// assign Aa = 16'd0;
// assign Ba = 16'd16384;
// assign Ca = ~{16'd800}+1;
// assign Ab = ~{16'd16384}+1;
// assign Bb = 16'd0;
// assign Cb = 16'd8000;
// assign Ac = 16'd0;
// assign Bc = ~{16'd16384}+1;
// assign Cc = 16'd8000;
// assign Ad = 16'd16384;
// assign Bd = 16'd0;
// assign Cd = ~{16'd800}+1;

// switch between the images
//assign Xmax = switch[3] ? flowersResX : zooeyResX;
//assign Ymax = switch[3] ? flowersResY : zooeyResY;


wire [26:0] numeratorY, numeratorX;
wire [26:0] denominatorY, denominatorX;
wire [26:0] scaleQuotientY, scaleQuotientX;

betterInterpolate  better1(clock_65mhz,hcount,vcount,
   Aa,Ac,Ba,Bc,Ca,Cc,
   zooeyResY,outAddrY2,numeratorY,denominatorY,
   scaleQuotientY);

betterInterpolate  better2(clock_65mhz,hcount,vcount,
   Ad,Ab,Bd,Bb,Cd,Cb,
   zooeyResX,outAddrX2,numeratorX,denominatorX,
   scaleQuotientX);

// debug area, switch between interpolation modes
//reg interpMode = 1'b0;
//reg rightDn;

//always @ (posedge clock_65mhz)
//begin
```

```
// rightDn <= right;

// if (~reset) interpMode <= (rightDn && ~right) ? interpMode + 1 : interpMode;
// else interpMode <= 1'b0;
//end

//assign outAddrX = switch[5] ? outAddrX1 : outAddrX2;
//assign outAddrY = switch[5] ? outAddrY1 : outAddrY2;

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

// read image in from rom starting at x0,y0
wire [10:0] x0 = 11'b01000000000;
wire [9:0] y0  = 10'b0100000000;

// are we INSIDE the quadrilateral bounded by the corners?
wire bound = (signA && signB && signC && signD);

wire  [7:0]  imgPixelt1;
wire  [23:0]  imgPixel1, imgPixel2, imgPixelt2, imgPixel;
reg  [8:0]  x;
reg  [7:0]  y;

//vga_flowersRom flowersRom(clock_65mhz,outAddrX2[8:0],outAddrY2[7:0],
//      ~clock_65mhz,imgPixelt1);

vga_zooeyRom zooeyRom(clock_65mhz,outAddrX2[23:15],outAddrY2[20:13],
   ~clock_65mhz,imgPixelt2);

assign imgPixel2 = bound ? imgPixelt2 : 24'b0;


always @ (posedge clk)
  begin
  if (red_yes) begin
```

```
    red <= imgPixelt2[23:16];
 green <= imgPixelt2[15:8];
 blue <= imgPixelt2[7:0];
 end
  else begin
 red <= r;
 blue <= bl;
 case (switch[7:5])
   3'b000: green <= gr;
3'b001 : green <= |lpixela[15:8] ? lpixela[15:8] : gr;
3'b010 : green <= |lpixelb[15:8] ? lpixela[15:8] : gr;
3'b011 : green <= |lpixelc[15:8] ? lpixela[15:8] : gr;
3'b100 : green <= |lpixeld[15:8] ? lpixela[15:8] : gr;
 endcase
 end
     end

   assign vga_out_red = red;
   assign vga_out_green = green;
   assign vga_out_blue = blue;

   assign led = ~{1,0,vert,vert,vert,horiz,horiz,horiz};

   always @(posedge clk)
   dispdata <= {state,60'b0};

endmodule
```

A.2 Vertex Output Module

```
//////////////////////////////////////
// Vertex detection logic by Jessica Barber
//////////////////////////////////////

module vertex_output(clock_65mhz,reset,initialize,wantred,
   sync_in,blank_in,hcount_in,vcount_in,hsync_in,vsync_in,
   y,u,v,
   vga_red,vga_green,vga_blue,
   sync_out,blank_out,hsync_out,vsync_out,
   tlh_pos,tlv_pos,trh_pos,trv_pos,blh_pos,blv_pos,brh_pos,brv_pos,
   state_is,horizontal,vertical,red);

input clock_65mhz;
input reset;
input initialize;
input wantred;
input sync_in,blank_in;
input [10:0] hcount_in;
input [9:0] vcount_in;
input hsync_in;
input vsync_in;
input [7:0] y,u,v;
output [7:0] vga_red,vga_green,vga_blue;
output sync_out,blank_out;   // delayed signals
output hsync_out,vsync_out;  //
output [11:0] tlh_pos,tlv_pos,trh_pos,trv_pos,blh_pos,blv_pos,brh_pos,brv_pos;
output [3:0] state_is;
output horizontal,vertical,red;

reg s_out,b_out,h_out,v_out;
reg [7:0] v_r,v_g,v_b;
reg hcorner_pixel;
reg vcorner_pixel;
reg found_corner;

reg is_red = 0;
reg [3:0] yes_red;
reg yes_red_2;
reg hhoriz,vhoriz;
reg [10:0] rh1_pos,rh2_pos,rh3_pos;
reg [10:0] h1_pos,h2_pos,h3_pos,h4_pos;
reg [10:0] ah1_pos,ah2_pos,ah3_pos,ah4_pos;
reg [10:0] h1_pos_temp,h2_pos_temp,h3_pos_temp,h4_pos_temp;
reg [10:0] tlh,blh,trh,brh,tlha,blha,trha,brha,tlhb,blhb,trhb,brhb,tlhc,blhc,trhc,
reg [12:0] tlhd,blhd,trhd,brhd;
reg [9:0] rv1_pos,rv2_pos,rv3_pos;
reg [9:0] v1_pos,v2_pos,v3_pos,v4_pos;
reg [9:0] av1_pos,av2_pos,av3_pos,av4_pos;
reg [9:0] v1_pos_temp,v2_pos_temp,v3_pos_temp,v4_pos_temp;
reg [9:0] tlv,blv,trv,brv,tlva,blva,trva,brva,tlvb,blvb,trvb,brvb,tlvc,blvc,trvc,b
reg [12:0] tlvd,blvd,trvd,brvd;
reg [9:0] hhoriz_check;
reg [10:0] vhoriz_check;
reg [3:0] state,second_state,third_state = 4'b0000;

parameter WIDTH   = 32;      // default width: 64 pixels
         parameter HEIGHT  = 32;      // default height: 64 pixels

         parameter BLCOLOR = {8'b11111111,8'b11111111,8'b11111111};  // bottom lef
parameter TLCOLOR = {8'b0,8'b11111111,8'b0}; // top left color: green
parameter BRCOLOR = {8'b0,8'b0,8'b11111111}; // bottom right color: blue
parameter TRCOLOR = {8'b11111111,8'b0,8'b11111111}; // top right color: purple

parameter RED_U_MIN = 8'b10111000;
parameter RED_Y_MIN = 8'b00111100;
parameter RED_Y_MAX = 8'b11111111;

wire [7:0] r,g,b;
wire [10:0] tl_h;
wire [9:0] tl_v;
reg [23:0] pixel;

YCrCb2RGB rgb(clock_65mhz,reset,{y,2'b0},{u,2'b0},{v,2'b0},r,g,b);

always @ (posedge clock_65mhz) begin
```

```
is_red <= ((RED_Y_MIN < y) & (y < RED_Y_MAX) & (u > RED_U_MIN));
// searches for a block of 9 red pixels before labelling a corner

yes_red <= {yes_red[2:0],is_red};

if (yes_red == 4'b1111 && hcount_in < 1023 && hcount_in > 25 && vcount_in < 767 &&
rh1_pos <= hcount_in;
rv1_pos <= vcount_in;
yes_red_2 <= 1;
end

case (second_state)
4'b0000:
if (yes_red_2)
begin
h1_pos_temp <= rh1_pos; // bottom corner
h2_pos_temp <= rh1_pos; // rightmost corner
h3_pos_temp <= rh1_pos; // leftmost corner
h4_pos_temp <= rh1_pos; // top corner
v1_pos_temp <= rv1_pos;
v2_pos_temp <= rv1_pos;
v3_pos_temp <= rv1_pos;
v4_pos_temp <= rv1_pos;
second_state <= 4'b0001;
end
4'b0001:
begin
if (hcount_in == 1343 && vcount_in == 805) second_state <= 4'b0010;
else if (yes_red_2) begin
if (rh1_pos > h2_pos_temp) begin
h2_pos_temp <= rh1_pos;
v2_pos_temp <= rv1_pos;
second_state <= 4'b0001;
end
```

```
if (rh1_pos < h3_pos_temp) begin
h3_pos_temp <= rh1_pos;
v3_pos_temp <= rv1_pos;
second_state <= 4'b0001;
end
if (rv1_pos < v4_pos_temp) begin
h4_pos_temp <= rh1_pos;
v4_pos_temp <= rv1_pos;
second_state <= 4'b0001;
end
end
else second_state <= 4'b0001;
end

4'b0010:
begin
ah1_pos <= h1_pos_temp;
av1_pos <= v1_pos_temp;
ah2_pos <= h2_pos_temp;
av2_pos <= v2_pos_temp;
ah3_pos <= h3_pos_temp;
av3_pos <= v3_pos_temp;
ah4_pos <= h4_pos_temp;
av4_pos <= v4_pos_temp;
second_state <= 4'b0000;
end
endcase

if ((hcount_in == ah2_pos || hcount_in == ah3_pos) && is_red) hhoriz_check <= hhor
if ((vcount_in == av4_pos || vcount_in == av1_pos) && is_red) vhoriz_check <= vhor

if (hcount_in == 1343 && vcount_in == 805) begin
if (hhoriz_check > 10'b0000110000) hhoriz <= 1;
else if (hhoriz_check < 10'b0000010000 && hhoriz == 1) hhoriz <= 0;
```

```
if (vhoriz_check > 11'b00111000000) vhoriz <= 1;
else if (vhoriz_check < 11'b00011110000 && vhoriz == 1) vhoriz <= 0;

hhoriz_check <= 10'b0;
   vhoriz_check <= 11'b0;
end

case (third_state)
     4'b0000: if (initialize) third_state <= 4'b0001;

4'b0001: begin // must start with bottom left corner pointing down
if (reset) third_state <= 4'b0000;
blh <= ah1_pos;
blv <= av1_pos;
brh <= ah2_pos;
brv <= av2_pos;
tlh <= ah3_pos;
tlv <= av3_pos;
trh <= ah4_pos;
    trv <= av4_pos;
third_state <= 4'b0010;
end
4'b0010: begin
if (reset) third_state <= 4'b0000;
if (hhoriz || vhoriz) begin
if (ah2_pos - ah3_pos < {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b0011;
else if (ah2_pos - ah3_pos > {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b01
end
else begin
blh <= ah1_pos;
blv <= av1_pos;
brh <= ah2_pos;
brv <= av2_pos;
tlh <= ah3_pos;
tlv <= av3_pos;
```

```
trh <= ah4_pos;
trv <= av4_pos;
end
end

4'b0011: begin
if (reset) third_state <= 4'b0000;
begin // if rotated vertically
blh <= ah2_pos;
blv <= av1_pos;
brh <= ah2_pos;
brv <= av4_pos;
tlh <= ah3_pos;
tlv <= av1_pos;
trh <= ah3_pos;
trv <= av4_pos;
end
if (av3_pos > av2_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4'b0010;
else if (av3_pos < av2_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4'b0101;
end

4'b0100: begin
if (reset) third_state <= 4'b0000;
begin // if rotated horizontally
blh <= ah3_pos;
blv <= av1_pos;
brh <= ah2_pos;
brv <= av1_pos;
tlh <= ah3_pos;
tlv <= av4_pos;
trh <= ah2_pos;
trv <= av4_pos;
end
```

```
if (av2_pos > av3_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4'b0010;
else if (av2_pos < av3_pos && vhoriz == 0 && hhoriz == 0)
       third_state <= 4'b1000;
end

4'b0101: begin
if (reset) third_state <= 4'b0000;
if (hhoriz || vhoriz) begin
if (ah2_pos - ah3_pos < {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b0011;
else if (ah2_pos - ah3_pos > {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b01
end
begin
blh <= ah2_pos;
blv <= av2_pos;
brh <= ah4_pos;
brv <= av4_pos;
tlh <= ah1_pos;
tlv <= av1_pos;
trh <= ah3_pos;
trv <= av3_pos;
end
end

4'b0110: begin
if (reset) third_state <= 4'b0000;
begin // if rotated horizontally
blh <= ah2_pos;
blv <= av4_pos;
brh <= ah3_pos;
brv <= av4_pos;
tlh <= ah2_pos;
tlv <= av1_pos;
trh <= ah3_pos;
trv <= av1_pos;
```

```
end
if (av2_pos > av3_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4'b0111;
else if (av2_pos < av3_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4'b0101;
end

4'b0111: begin
if (reset) third_state <= 4'b0000;
if (hhoriz || vhoriz) begin
if (ah2_pos - ah3_pos < {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b1001;
else if (ah2_pos - ah3_pos > {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b01
end
begin
blh <= ah4_pos;
blv <= av4_pos;
brh <= ah3_pos;
brv <= av3_pos;
tlh <= ah2_pos;
tlv <= av2_pos;
trh <= ah1_pos;
trv <= av1_pos;
end
end

4'b1000: begin
if (reset) third_state <= 4'b0000;
if (hhoriz || vhoriz) begin
if (ah2_pos - ah3_pos < {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b1001;
else if (ah2_pos - ah3_pos > {1'b0,av1_pos} - {1'b0,av4_pos}) third_state <= 4'b01
end
begin
blh <= ah3_pos;
blv <= av3_pos;
brh <= ah1_pos;
```

```
brv <= av1_pos;
tlh <= ah4_pos;
tlv <= av4_pos;
trh <= ah2_pos;
trv <= av2_pos;
end
end

4’b1001: begin
if (reset) third_state <= 4’b0000;
begin // if rotated vertically
blh <= ah3_pos;
blv <= av4_pos;
brh <= ah3_pos;
brv <= av1_pos;
tlh <= ah2_pos;
tlv <= av4_pos;
trh <= ah2_pos;
trv <= av1_pos;
end
if (av3_pos > av2_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4’b0111;
else if (av3_pos < av2_pos && vhoriz == 0 && hhoriz == 0)
third_state <= 4’b1000;
end

endcase


if (third_state == 4’b0000) begin
if ((hcount_in >= ah1_pos && hcount_in < (ah1_pos+WIDTH)) &&
   (vcount_in >= av1_pos && vcount_in < (av1_pos+HEIGHT)))
pixel <= BLCOLOR;
else if ((hcount_in >= ah2_pos && hcount_in < (ah2_pos+WIDTH)) &&
        (vcount_in >= av2_pos && vcount_in < (av2_pos+HEIGHT)))
```

```
pixel <= BRCOLOR;
else if ((hcount_in >= ah3_pos && hcount_in < (ah3_pos+WIDTH)) &&
        (vcount_in >= av3_pos && vcount_in < (av3_pos+HEIGHT)))
pixel <= TLCOLOR;
else if ((hcount_in >= ah4_pos && hcount_in < (ah4_pos+WIDTH)) &&
        (vcount_in >= av4_pos && vcount_in < (av4_pos+HEIGHT)))
pixel <= TRCOLOR;
else if (wantred && is_red) pixel <= {8'b11111111,8'b0,8'b0};
else pixel <= {r,g,b};
end
else begin
if ((hcount_in >= blh && hcount_in < (blh+WIDTH)) &&
   (vcount_in >= blv && vcount_in < (blv+HEIGHT)))
   pixel <= BLCOLOR;
else if ((hcount_in >= trh && hcount_in < (trh+WIDTH)) &&
        (vcount_in >= trv && vcount_in < (trv+HEIGHT)))
pixel <= TRCOLOR;
else if ((hcount_in >= brh && hcount_in < (brh+WIDTH)) &&
  (vcount_in >= brv && vcount_in < (brv+HEIGHT)))
pixel <= BRCOLOR;
else if ((hcount_in >= tlh && hcount_in < (tlh+WIDTH)) &&
  (vcount_in >= tlv && vcount_in < (tlv+HEIGHT)))
pixel <= TLCOLOR;
else if (wantred && is_red) pixel <= {8'b11111111,8'b0,8'b0};
else pixel <= {r,g,b};
end

{s_out,b_out,h_out,v_out} <= {sync_in,blank_in,hsync_in,vsync_in};
{v_r,v_g,v_b} <= pixel;

   end // end the always block

   assign sync_out = s_out;
   assign blank_out = b_out;
   assign hsync_out = h_out;
   assign vsync_out = v_out;
   assign vga_red = v_r;
   assign vga_green = v_g;
   assign vga_blue = v_b;
   assign tlh_pos = {1'b0,tlh};
   assign tlv_pos = {2'b0,tlv};
   assign trh_pos = {1'b0,trh};
   assign trv_pos = {2'b0,trv};
   assign blh_pos = {1'b0,blh};
   assign blv_pos = {2'b0,blv};
   assign brh_pos = {1'b0,brh};
   assign brv_pos = {2'b0,brv};
   assign state_is = third_state;
   assign horizontal = hhoriz;
   assign veritcal = vhoriz;
   assign red = is_red;

endmodule
```

A.3 combinationalAbc Module

```
module combinationalAbc(clk,switch,btn,inA1,inA2,inB1,inB2,
analyzerOut1, analyzerOut2,
regA,regB,regC,start,rdy,count);

// this operation is only performed once every FRAME.

// Generate a line in the form Ax + By + C = 0
// from two points. Since we use vectors to do this
// the line HAS a positive and negative side.
// In addition, normalize the components A, B, and C
// to avoid doing a square root at every pixel to find the
// the distance from a point to a line.

// Generate A,B,C for a vector
```

```
input clk;
input switch;
input btn, start;

input [10:0] inA1,inA2,inB1,inB2;
wire  [11:0] A1,A2,B1,B2;

// pad the inputs
assign A1 = {1'b0,inA1};
assign A2 = {1'b0,inA2};
assign B1 = {1'b0,inB1};
assign B2 = {1'b0,inB2};



// since all the logic is combinational, implement
// a simple timer that is generous enough to ensure
// the computation is complete.
output reg [6:0] count = 7'b0;
output reg rdy = 1'b0;

output reg signed [15:0] regA,regB, regC = 16'b0;

// configurable debug
output [15:0] analyzerOut1, analyzerOut2;

// intermediate values in the computation
wire       [25:0] va1sq, va2sq; = 26'b0;
reg  signed [12:0] va1,va2 = 13'b0;
reg  [24:0] vasq = 25'b0;

wire   [12:0] sqva;
wire [15:0] nva1, nva1_frac;
wire [15:0] nva2, nva2_frac;
```

```
reg btnDown;

// register our outputs
reg signed [17:0] regAt,regBt;
reg signed [29:0] regCt1, regCt2, regCt;

wire signed [13:0] wireA,wireB,wireC;

wire [25:0] q1,q2;
wire sqrt_rdy;

// instantiate our hardware arithmetic
multiply_signed13 mult131 (clk,va1,va1,va1sq,q1);
multiply_signed13 mult132 (clk,va2,va2,va2sq,q2);
square_root25 sqrt251 (vasq,clk,sqva,sqrt_rdy);

wire rfd1,rfd2;
wire [15:0] divider1, divisor1, divider2, divisor2;

// take the absolute value to ensure a non signed division
// we will resign the quotient later
assign divider1 = va1[12] ? ~{3'b111,va1} + 1 : {3'b000,va1};
assign divisor1 = {3'b000,sqva};

assign divider2 = va2[12] ? ~{3'b111,va2} + 1 : {3'b000,va2};
assign divisor2 = {3'b000,sqva};

divide16 divide161(clk,divider1,divisor1,nva1,nva1_frac,rfd1);
divide16 divide162(clk,divider2,divisor2,nva2,nva2_frac,rfd2);

always @ (posedge clk)
begin
count <= (start) ? 7'b0 : count+1;
rdy <= sqrt_rdy;
if (count == 40) // generous alotment of cycles for the calculation
```

```
begin
//rdy = 1'b1;
regA <= regAt[17:2];  // makes reg a 2_14 signed fixed pt
regB <= regBt[17:2];  // makes reg a 2_14 signed fixed pt
regC <= regCt[27:12]; // makes it a  11_4 signed fixed point
end
end

// take two paths, one combinational path which runs on A,B,C,D
// and one procedural path which just waits for the combinational
// logic to propagate and then reads the output
always @ ( A1,B1,A2,B2,btn,va1,va2,nva1,nva1_frac, regAt, regBt,
va1sq, va2sq, nva2, nva2_frac)
begin
va1  = B1 - A1;
va2  = B2 - A2;
vasq  = va1sq+va2sq;

// check these lines in debug, unsure of them!
regAt = va2[12] ? {1'b0,nva2[0],nva2_frac} : ~{1'b0,nva2[0],nva2_frac}+1;
regBt = va1[12] ? ~{1'b0,nva1[0],nva1_frac}+1 : {1'b0,nva1[0],nva1_frac};
regCt1 = -A1*regAt; // lower 16 bits are fractional
regCt2   = -A2*regBt; // lower 16 bits are fractional
regCt = -1*A1*regAt -1*A2*regBt; // lower 16 bits are fractional

end

// both 14_2 fixed point numbers
assign  analyzerOut2  = regCt1[27:12];
assign analyzerOut1  = regCt2[27:12];

endmodule
```

subsectionInterpolation Module

```
module betterInterpolate(clk,hcount,vcount,
 A1,A2,B1,B2,C1,C2,
dimMax,addrReg,absDistanceAReg,denominatorReg,
scaleQuotient, remainderFrac);

// take two lines in the form Ax+By+C and a point. Determine
// the distance of the point to each line (Ax0+By0+C) and then interpolate
// by: da*dimMax/(da+dc)  where da and dc are the distance
// of our target points to the the given lines and dimMax is the
// maximum resolution of our source image along the axis we are interpolating
// over (dimMaxX for a 640x480 image is 640...)

// this is a 3 cycle pipelined version of pixelInterpolate with a latency
//of 3 cycles and a throughput of 1 result per cycle.


input clk;
input [10:0] hcount, dimMax;
input [9:0]  vcount;
input signed [15:0] A1,B1,C1,A2,B2,C2;

output reg [25:0] addrReg;

reg [10:0]  regHcount1, regHcount2, regHcount3;
reg [9:0] regVcount1, regVcount2, regVcount3;

// notice none of summations get a widened bit output from
// the inputs, so there is a chance (unlikely in the problem)
// of overflow
wire signed [26:0] Ax1, Ax2;
wire signed [26:0] By1, By2;
reg signed  [26:0] regAx1, regAx2;
reg signed  [26:0] regBy1, regBy2;
output    [28:0] scaleQuotient;

// 29 bits for these is larger than possible, but when i left it at
// 27 it seemed i was getting overflow (which is possible, even likely)!
reg signed  [28:0] distanceA, distanceC;
wire  [28:0] absDistanceA, absDistanceC;

assign absDistanceA = distanceA[28]? ~distanceA + 1 : distanceA;
assign absDistanceC = distanceC[28]? ~distanceC + 1 : distanceC;

wire signed  [37:0] numerator, addrOut;
reg         [37:0] absNumeratorReg;
output reg  [28:0] denominatorReg;
output reg [28:0] absDistanceAReg;

wire [31:0] addrQuotient;
// in order to pipeline, well need to just propagate values through
// the registers at each tick. Pray that the modules do what
// theyre supposed to and register their output.
wire [25:0] addrFullReg;
output wire [15:0] remainderFrac;

always @ (posedge clk)
begin

regHcount1 <= hcount;
regHcount2 <= regHcount1;
regHcount3 <= regHcount2;

regVcount1 <= vcount;
regVcount2 <= regVcount1;
regVcount3 <= regVcount2;

regAx1 <= Ax1;
regAx2 <= Ax2;
regBy1 <= By1;
regBy2 <= By2;
```

```
distanceA <= Ax1 + By1 + {C1,10'b0};
distanceC <= Ax2 + By2 + {C2,10'b0};

absDistanceAReg <= absDistanceA;
denominatorReg  <= absDistanceA + absDistanceC;
//addrFullReg  <= remainderFrac*dimMax;
addrReg  <= addrFullReg;
end

assign addrFullReg = remainderFrac*dimMax;

// isntantiation of all our black box arithmetic modules
mult2_16x11 mult162_111(clk,A1,regHcount3,gnd,Ax1);
mult2_16x11 mult162_112(clk,A2,regHcount3,gnd,Ax2);
mult2_16x11 mult162_113(clk,B1,{1'b0,regVcount3},gnd,By1);
mult2_16x11 mult162_114(clk,B2,{1'b0,regVcount3},gnd,By2);

//mult_signed27x11 mult27_111(clk,scaleQuotient,dimMax,addrOut,gnd);

// notice we do the same clipping as we did in the original version,
// divide a 23_12 by a 12_14 and get a number scaled by 2^2 too hight!
// so we scale the answer back
divide_fraction_29x29 df29x291(clk,absDistanceAReg,denominatorReg,scaleQuotient,re



endmodule
```

A.4 Original Pixel Interpolation

(Included for comparison/posterity; not actually used.)

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

```
module pixelInterpolate(clk,hcount,vcount,
inA1,inB1,inC1,inA2,inB2,inC2,
Amax,outAddr,sumAxBy1,sumAxBy2);

// this function is now depricated. It was an attempt to perform
// the interpolation operation combinationally.

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

// A and B should be signed 2bit_14bit (2Q) numbers
// C should be a signed 12bit_4bit (12Q) number
input signed [15:0] inA1,inB1,inC1;
input signed [15:0] inA2,inB2,inC2;

input [10:0] Amax; // never need bigger than hcount resolution!

output reg [11:0] outAddr;

output reg signed [26:0] sumAxBy1, sumAxBy2;
reg signed [26:0] sumD;


// if Ax + By + C is > -2 and < -2 then draw the line
// or if were not drawing the line, just check the sign of the number

// Ax is 13_14, By is 12_14 C is 12_4, all are signed
wire signed [25:0] Ax1, Ax2;
wire signed [25:0] By1, By2;
wire signed [25:0] q1,q2,q3,q4;
wire signed [37:0] numerator, regNumerator;
```

```
// here’s where more fixed point magic has to happen
// absD is 13_14 unsigned, absNumerator is 24_14
// unless we have d=1024 and xmax=1024 (upper bound), absNumerator
// should be less than 20 bits. In addition, verilog has required us
// do build a divider no wider than 32 bits. Thus, we cap 4 bits off
// the front (integer part) and 2 bits of the back (fractional) and
// run the divide with a 20_12 numerator and a 13_14 denominator.
// we could also shave some more bits since we are absolute valuing
// the sign bits, but ill leave them in case i want to do a signed
// divide later to check direction

// after all this, we snip the remainder and end up with with two trailing
// bits which scale greater than we need and a bunch of leading bits
// greater than we need (cant generate more than Amax!)

// this divide has a one cycle latency, which worries me that we might start
// putting more numbers into it before its done with the ones its got, so lets
// register its inputs

// well take 2 full cycles for the front end of computation, and a full cycle
// for the division, then output the result a hair late!



// compute da combinationally
mult_signed16x11 mult16_111(clk,inA1,hcount,Ax1,q1);
mult_signed16x11 mult16_112(clk,inB1,{1’b0,vcount},By1,q2);
//assign sumAxBy1 = Ax1 + By1 + {inC1,10’b0};

// compute dc combinationally
mult_signed16x11 mult16_113(clk,inA2,hcount,Ax2,q3);
mult_signed16x11 mult16_114(clk,inB2,{1’b0,vcount},By2,q4);

//assign sumAxBy2 = Ax2 + By2 + {inC2,10’b0};
//assign sumD = sumAxBy2 + sumAxBy1;
```

```
mult_signed27x11 mult27_111(clk,sumAxBy1,Amax,numerator,regNumerator);



wire [31:0] addrQuotient;
reg [26:0] absSumD;
reg [37:0] absNumerator;
reg [26:0] divider;
reg [31:0] divisor;
wire rfd;

divide_38x27 divide38x271(clk,divisor,divider,addrQuotient,gnd,rfd);

always @ (Ax1,By1,Ax2,By2,inC1,inC2,sumAxBy1,sumAxBy2, regNumerator, sumD)
begin
sumAxBy1  = Ax1 + By1 + {inC1,10'b0};
sumAxBy2  = Ax2 + By2 + {inC2,10'b0};
sumD  = sumAxBy2 + sumAxBy1;
absSumD       = sumD[26] ? ~{sumD[26:0]} + 1: sumD;
absNumerator  = regNumerator[37] ? ~{regNumerator[37:0]} + 1 : regNumerator;
end

always @ (rfd)
begin
divider <= absSumD;
divisor <= absNumerator[33:2];
end


always @ (posedge clk)
begin
//absSumD       <= sumD[26] ? ~sumD + 1: sumD; // not relevant for interpolation
//absNumerator  <= regNumerator[37] ? ~regNumerator + 1 : regNumerator;
```

```
outAddr  <= addrQuotient[17:2];
end


endmodule
```

A.5 Line Drawing Module

```
/////////////////////////////////////////////////////////////////////
// draws the positive and negative fields on either side of a line
/////////////////////////////////////////////////////////////////////
module drawLine(clk,hcount,vcount,A,B,C,mode,pixel, sign,switch);
input clk;
input [10:0]  hcount;
input [9:0]  vcount;
input [1:0] mode, switch; // controls the color of the seperators

// A and B should be signed 2bit_14bit (2Q) numbers
// C should be a signed 12bit_4bit (12Q) number
input signed [15:0]  A,B,C;


output reg [23:0] pixel;
output reg sign;
reg signed [28:0] sumAxBy;


// if Ax + By + C is > -2 and < -2 then draw the line
// or if were not drawing the line, just check the sign of the number

// Ax is 13_14, By is 12_14 C is 12_4, all are signed
wire signed [26:0] Ax;
wire signed [26:0] By;
wire signed [26:0] q1,q2;
```

```
// run the summation and multiplies completely combinationally
mult_signed16x11 mult16_111(clk,A,hcount,Ax,q1);
mult_signed16x11 mult16_112(clk,B,{1'b0,vcount},By,q2);
//sumAxBy = Ax + By + {C,10'b0};

// take whatever the current value is of sumAxBy and dump it into the
// registered output.
always @ (posedge clk)
begin
//Ax  = A*hcount;
//By  = B*vcount;

//sumAxBy  = A*hcount + B*vcount + {C,10'b0};  // scale C to match Q12
sign  <= switch ? sumAxBy[28] : ~sumAxBy[28]; // 0 is positive, 1 is negative (duh
sumAxBy  <= Ax + By + {C,10'b0};

// no default needed
if (switch)
case (mode)
2'b00: pixel   <= (sumAxBy < 0) ?  24'h000000 : 24'h00FF00;
2'b01: pixel   <= (sumAxBy < 0) ?  24'h000000 : 24'h00FF00;
2'b10: pixel   <= (sumAxBy < 0) ?  24'h000000 : 24'h00FF00;
2'b11: pixel   <= (sumAxBy < 0) ?  24'h000000 : 24'h00FF00;
endcase
else
case(mode)
2'b00: pixel   <= (sumAxBy > 0) ?  24'h000000 : 24'h00FF00;
2'b01: pixel   <= (sumAxBy > 0) ?  24'h000000 : 24'h00FF00;
2'b10: pixel   <= (sumAxBy > 0) ?  24'h000000 : 24'h00FF00;
2'b11: pixel   <= (sumAxBy > 0) ?  24'h000000 : 24'h00FF00;
endcase

end

endmodule
```

A.6 Debouncing Module

```
///////////////////////////////////////////////////////////////////////////////
//
// debounce: Clean up switch and button signals over a 65MHz clock
//
///////////////////////////////////////////////////////////////////////////////

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

///////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
// by 6.111 staff, taken from Fall 2005 Handouts page
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

A.7 ZBT Driver Module

```
///////////////////////////////////////////////////////////////////////////
// File: zbt_6111.v
// Date: 27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user. The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//
///////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.
///////////////////////////////////////////////////////////////////////////

module zbt_6111(clk, cen, we, addr, write_data, read_data,
ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);
input clk; // system clock
input cen; // clock enable for gating ZBT cycles
input we; // write enable (active HIGH)
input [18:0] addr; // memory address
input [35:0] write_data; // data to write
```

```
output [35:0] read_data; // data read from memory
output ram_clk; // physical line to ram clock
output ram_we_b; // physical line to ram we_b
output [18:0] ram_address; // physical line to ram address
inout [35:0] ram_data; // physical line to ram data
output ram_cen_b; // physical line to ram clock enable

// clock enable (should be synchronous and one cycle high at a time)
wire ram_cen_b = ~cen;

// create delayed ram_we signal: note the delay is by two cycles!
// ie we present the data to be written two cycles after we is raised
// this means the bus is tri-stated two cycles after we is raised.

reg [1:0] we_delay;

always @(posedge clk)
we_delay <= cen ? {we_delay[0],we} : we_delay;

// create two-stage pipeline for write data

reg [35:0] write_data_old1;
reg [35:0] write_data_old2;

always @(posedge clk)
if (cen)
{write_data_old2, write_data_old1} <= {write_data_old1, write_data};

// wire to ZBT RAM signals
assign ram_we_b = ~we;
assign ram_clk = ~clk; // RAM is not happy with our data hold
   // times if its clk edges equal FPGA's
   // so we clock it on the falling edges
   // and thus let data stabilize longer
```

```
assign ram_address = addr;

assign ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
assign read_data = ram_data;

endmodule
```

A.8 ZBT Read Module

```
//////////////////////////////////////////////////////////////////////////////
// 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; 2 blocks of 18-bit 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 [17:0] vr_pixel;
    output [18:0] vram_addr;
    input [35:0] vram_read_data;

    parameter HMID = 9'd367; // The horizontal center of the image in MEMORY
    parameter HSTART = HMID-9'd256; // The horizontal counter decrements!!! // use
    parameter VMID = 9'd287; // The vertical center of the image in MEMORY
    parameter VSTART = VMID-9'd192;

    //wire [18:0] vram_addr = {1'b0,vcount[9:1]+VSTART, ~hcount[10:2]-9'd180}; //
    wire [18:0] vram_addr = {1'b0,vcount[9:1]+VSTART, hcount[10:2]+HSTART};
    wire [1:0] hc4 = hcount[1:0];
    reg [17:0] vr_pixel;
```

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

   always @(posedge clk)
     last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;

   always @(*) // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4)
       2'd3: vr_pixel = last_vr_data[17:0];//last_vr_data[8:0];
       2'd2: vr_pixel = last_vr_data[17:0];//last_vr_data[8+9:0+9];
       2'd1: vr_pixel = last_vr_data[35:18];//last_vr_data[8+18:0+18];
       2'd0: vr_pixel = last_vr_data[35:18];//last_vr_data[8+27:0+27];
     endcase

endmodule // vram_display
```

A.9 NTSC to ZBT Module

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

/////////////////////////////////////////////////////////////////////////////
// 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.
```

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

// Prepare data and address values to fill ZBT memory with NTSC data
module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);
    input clk; // system clock
    input vclk; // video clock from camera
    input [2:0] fvh;
    input dv;
    input [29:0] din;
    output [18:0] ntsc_addr;
    output [35:0] ntsc_data;
    output ntsc_we; // write enable for NTSC data
    input sw; // switch which determines mode (for debugging)

    parameter COL_START = 10'd30;
    parameter ROW_START = 10'd30;

    // here put the luminance data from the ntsc decoder into the ram
    // this is for 1024 x 768 XGA display
    reg [9:0] col = 0;
    reg [9:0] row = 0;
    reg [17:0] vdata = 0;
    reg vwe;
    reg old_dv;
    reg old_frame; // frames are even / odd interlaced
    reg even_odd; // decode interlaced frame to this wire
    wire frame = fvh[2];
    wire frame_edge = frame & ~old_frame;

    always @ (posedge vclk) //LLC1 is reference
        begin
            old_dv <= dv;
            vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
            old_frame <= frame;
            even_odd = frame_edge ? ~even_odd : even_odd;
            if (!fvh[2])
                begin
                    col <= fvh[0] ? COL_START :
                    (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;

                    row <= fvh[1] ? ROW_START :
                    (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;

                    // 8 bits Y, 8 bits Cr, 2 bits Cb.
                    //vdata <= (dv && !fvh[2]) ? {din[29:22],din[19:12],din[9:8]}
                    // 6 bits each of Y, Cr, Cb
                    vdata <= (dv && !fvh[2]) ? {din[29:24],din[19:14],din[9:4]} :
                end
        end

    // synchronize with system clock

    reg [9:0] x[1:0],y[1:0];
    reg [17:0] 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

    // edge detection on write enable signal
    reg old_we;
    wire we_edge = we[1] & ~old_we;
```

```
    always @(posedge clk) old_we <= we[1];
        // shift each set of four bytes into a large register for the ZBT
        reg [35:0] mydata;

    always @(posedge clk)
        if (we_edge)
            mydata <= { mydata[17:0], data[1] };

    // compute address to store data in
    wire [18:0] myaddr = {y[1][8:0], eo[1], x[1][9:1]};
    // alternate (256x192) image data and address
    wire [35:0] mydata2 = {data[1],data[1],data[1],data[1]};
    wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

    // update the output address and data every 2. 2 pixels stored per line
    reg [18:0] ntsc_addr;
    reg [35:0] ntsc_data;
    wire ntsc_we = sw ? we_edge : (we_edge & (x[1][0]==1'b0));

    always @(posedge clk)
        if ( ntsc_we )
            begin
                ntsc_addr <= sw ? myaddr2 : myaddr; // normal and expanded modes
                ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
            end

endmodule
///////////////////////////////////////////////////////////////////////////////
```

A.10 Video Decoder Module

```
///////////////////////////////////////////////////////////////////////////////
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
```

```
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
/////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.
// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.
////////////////////////////////////////////////////////////////////////////

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output  f;
   output  v;
   output  h;
   output  data_valid;
   // output [4:0] state;

   parameter  SYNC_1 = 0;
   parameter  SYNC_2 = 1;
   parameter  SYNC_3 = 2;
   parameter  SAV_f1_cb0 = 3;
   parameter  SAV_f1_y0 = 4;
   parameter  SAV_f1_cr1 = 5;
   parameter  SAV_f1_y1 = 6;
   parameter  EAV_f1 = 7;
   parameter  SAV_VBI_f1 = 8;
   parameter  EAV_VBI_f1 = 9;
   parameter  SAV_f2_cb0 = 10;
   parameter  SAV_f2_y0 = 11;
   parameter  SAV_f2_cr1 = 12;
   parameter  SAV_f2_y1 = 13;
   parameter  EAV_f2 = 14;
   parameter  SAV_VBI_f2 = 15;
   parameter  EAV_VBI_f2 = 16;




   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.

   // Once we determine where to start, the FSM goes through a normal
   // sequence of SAV process_YCrCb EAV... repeat

   // The data stream looks as follows
   // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | E
   // There are two things we need to do:
   //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
   //   2. Decode the subsequent data
```

```
   reg [4:0]  current_state = 5'h00;
   reg [9:0]  y = 10'h000;  // luminance
   reg [9:0]  cr = 10'h000; // chrominance
   reg [9:0]  cb = 10'h000; // more chrominance

   assign  state = current_state;

   always @ (posedge clk)
     begin
if (reset)
  begin

  end
else
  begin
     // these states don't do much except allow us to know where we are in the str
     // whenever the synchronization code is seen, go back to the sync_state befor
     // transitioning to the new state
     case (current_state)
       SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
       SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
       SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
(tv_in_ycrcb == 10'h274) ? EAV_f1 :
(tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
(tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
(tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
(tv_in_ycrcb == 10'h368) ? EAV_f2 :
(tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
(tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

       SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
       SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
       SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
       SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;
```

```
      SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
      SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
      SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
      SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

      // These states are here in the event that we want to cover these signals
      // in the future. For now, they just send the state machine back to SYNC_1
      EAV_f1: current_state <= SYNC_1;
      SAV_VBI_f1: current_state <= SYNC_1;
      EAV_VBI_f1: current_state <= SYNC_1;
      EAV_f2: current_state <= SYNC_1;
      SAV_VBI_f2: current_state <= SYNC_1;
      EAV_VBI_f2: current_state <= SYNC_1;

    endcase
end
    end // always @ (posedge clk)

 // implement our decoding mechanism

 wire y_enable;
 wire cr_enable;
 wire cb_enable;

 // if y is coming in, enable the register
 // likewise for cr and cb
 assign y_enable = (current_state == SAV_f1_y0) ||
           (current_state == SAV_f1_y1) ||
           (current_state == SAV_f2_y0) ||
           (current_state == SAV_f2_y1);
 assign cr_enable = (current_state == SAV_f1_cr1) ||
            (current_state == SAV_f2_cr1);
 assign cb_enable = (current_state == SAV_f1_cb0) ||
            (current_state == SAV_f2_cb0);
```

```
   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg    f = 0;

   always @ (posedge clk)
     begin
y <= y_enable ? tv_in_ycrcb : y;
cr <= cr_enable ? tv_in_ycrcb : cr;
cb <= cb_enable ? tv_in_ycrcb : cb;
f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////
```

```
`define INPUT_SELECT                            4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                              4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
///////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                           2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                    1'b0
  // 0: Normal mode
  // 1: Square pixel mode
```

```
‘define DIFFERENTIAL_INPUT                      1’b0
  // 0: Single-ended inputs
  // 1: Differential inputs
‘define FOUR_TIMES_SAMPLING                     1’b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
‘define BETACAM                                 1’b0
  // 0: Standard video input
  // 1: Betacam video input
‘define AUTOMATIC_STARTUP_ENABLE                1’b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

‘define ADV7185_REGISTER_1 {‘AUTOMATIC_STARTUP_ENABLE, 1’b0, ‘BETACAM, ‘FOUR_TIMES

////////////////////////////////////////////////////////////////////////////////
// Register 2
////////////////////////////////////////////////////////////////////////////////

‘define Y_PEAKING_FILTER                        3’h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
‘define CORING                                  2’h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

‘define ADV7185_REGISTER_2 {3’b000, ‘CORING, ‘Y_PEAKING_FILTER}
```

```
///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                        2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                           4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                 1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                              1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT,

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                       1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
```

```
‘define BT656_TYPE                              1’b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

‘define ADV7185_REGISTER_4 {‘BT656_TYPE, 3’b000, 3’b110, ‘OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


‘define GENERAL_PURPOSE_OUTPUTS                 4’b0000
‘define GPO_0_1_ENABLE                          1’b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
‘define GPO_2_3_ENABLE                          1’b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
‘define BLANK_CHROMA_IN_VBI                     1’b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
‘define HLOCK_ENABLE                            1’b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

‘define ADV7185_REGISTER_5 {‘HLOCK_ENABLE, ‘BLANK_CHROMA_IN_VBI, ‘GPO_2_3_ENABLE,

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////

‘define FIFO_FLAG_MARGIN                        5’h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
‘define FIFO_RESET                              1’b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                    1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                     1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RES

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                         8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                       8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                       8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

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

```
// Register B
////////////////////////////////////////////////////////////////////////////////

‘define INPUT_HUE_ADJUST                              8’h00

‘define ADV7185_REGISTER_B {‘INPUT_HUE_ADJUST}

////////////////////////////////////////////////////////////////////////////////
// Register C
////////////////////////////////////////////////////////////////////////////////

‘define DEFAULT_VALUE_ENABLE                    1’b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
‘define DEFAULT_VALUE_AUTOMATIC_ENABLE          1’b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
‘define DEFAULT_Y_VALUE                         6’h0C
  // Default Y value

‘define ADV7185_REGISTER_C {‘DEFAULT_Y_VALUE, ‘DEFAULT_VALUE_AUTOMATIC_ENABLE, ‘DE

////////////////////////////////////////////////////////////////////////////////
// Register D
////////////////////////////////////////////////////////////////////////////////

‘define DEFAULT_CR_VALUE                        4’h8
  // Most-significant four bits of default Cr value
‘define DEFAULT_CB_VALUE                        4’h8
  // Most-significant four bits of default Cb value

‘define ADV7185_REGISTER_D {‘DEFAULT_CB_VALUE, ‘DEFAULT_CR_VALUE}

////////////////////////////////////////////////////////////////////////////////
// Register E
```

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

`define TEMPORAL_DECIMATION_ENABLE              1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL             2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE                4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_

////////////////////////////////////////////////////////////////////////////////
// Register F
////////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                      2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY              1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                    1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR                1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                         1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                        1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                              1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP, `POW

///////////////////////////////////////////////////////////////////////////////
// Register 33
///////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                       1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES               1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                             3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                              1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE, `AVERAGE_BIRIG

`define ADV7185_REGISTER_10 8'h00
```

```
‘define ADV7185_REGISTER_11 8’h00
‘define ADV7185_REGISTER_12 8’h00
‘define ADV7185_REGISTER_13 8’h45
‘define ADV7185_REGISTER_14 8’h18
‘define ADV7185_REGISTER_15 8’h60
‘define ADV7185_REGISTER_16 8’h00
‘define ADV7185_REGISTER_17 8’h01
‘define ADV7185_REGISTER_18 8’h00
‘define ADV7185_REGISTER_19 8’h10
‘define ADV7185_REGISTER_1A 8’h10
‘define ADV7185_REGISTER_1B 8’hF0
‘define ADV7185_REGISTER_1C 8’h16
‘define ADV7185_REGISTER_1D 8’h01
‘define ADV7185_REGISTER_1E 8’h00
‘define ADV7185_REGISTER_1F 8’h3D
‘define ADV7185_REGISTER_20 8’hD0
‘define ADV7185_REGISTER_21 8’h09
‘define ADV7185_REGISTER_22 8’h8C
‘define ADV7185_REGISTER_23 8’hE2
‘define ADV7185_REGISTER_24 8’h1F
‘define ADV7185_REGISTER_25 8’h07
‘define ADV7185_REGISTER_26 8’hC2
‘define ADV7185_REGISTER_27 8’h58
‘define ADV7185_REGISTER_28 8’h3C
‘define ADV7185_REGISTER_29 8’h00
‘define ADV7185_REGISTER_2A 8’h00
‘define ADV7185_REGISTER_2B 8’hA0
‘define ADV7185_REGISTER_2C 8’hCE
‘define ADV7185_REGISTER_2D 8’hF0
‘define ADV7185_REGISTER_2E 8’h00
‘define ADV7185_REGISTER_2F 8’hF0
‘define ADV7185_REGISTER_30 8’h00
‘define ADV7185_REGISTER_31 8’h70
‘define ADV7185_REGISTER_32 8’h00
‘define ADV7185_REGISTER_34 8’h0F
```

```
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
    tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
   end

   //
   // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
   //

   reg [7:0] clk_div_count, reset_count;
   reg clock_slow;
   wire reset_slow;

   initial
     begin
clk_div_count <= 8'h00;
// synthesis attribute init of clk_div_count is "00";
clock_slow <= 1'b0;
// synthesis attribute init of clock_slow is "0";
     end

   always @(posedge clock_27mhz)
     if (clk_div_count == 26)
       begin
  clock_slow <= ~clock_slow;
  clk_div_count <= 0;
       end
     else
       clk_div_count <= clk_div_count+1;
   always @(posedge clock_27mhz)
     if (reset)
       reset_count <= 100;
     else
       reset_count <= (reset_count==0) ? 0 : reset_count-1;

   assign reset_slow = reset_count != 0;

   //
   // I2C driver
   //

   reg load;
   reg [7:0] data;
   wire ack, idle;

   i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
   .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
   .sda(tv_in_i2c_data));

   //
   // State machine
   //

   reg [7:0] state;
   reg tv_in_reset_b;
   reg old_source;

   always @(posedge clock_slow)
      if (reset_slow)
begin
   state <= 0;
   load <= 0;
   tv_in_reset_b <= 0;
   old_source <= 0;
```

```
end
      else
case (state)
  8'h00:
    begin
       // Assert reset
       load <= 1'b0;
       tv_in_reset_b <= 1'b0;
       if (!ack)
 state <= state+1;
    end
  8'h01:
    state <= state+1;
  8'h02:
    begin
       // Release reset
       tv_in_reset_b <= 1'b1;
       state <= state+1;
           end
  8'h03:
    begin
       // Send ADV7185 address
       data <= 8'h8A;
       load <= 1'b1;
       if (ack)
 state <= state+1;
    end
  8'h04:
    begin
       // Send subaddress of first register
       data <= 8'h00;
       if (ack)
 state <= state+1;
    end
  8'h05:
    begin
       // Write to register 0
       data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
       if (ack)
 state <= state+1;
    end
  8'h06:
    begin
       // Write to register 1
       data <= `ADV7185_REGISTER_1;
       if (ack)
 state <= state+1;
    end
  8'h07:
    begin
       // Write to register 2
       data <= `ADV7185_REGISTER_2;
       if (ack)
 state <= state+1;
    end
  8'h08:
    begin
       // Write to register 3
       data <= `ADV7185_REGISTER_3;
       if (ack)
 state <= state+1;
    end
  8'h09:
    begin
       // Write to register 4
       data <= `ADV7185_REGISTER_4;
       if (ack)
 state <= state+1;
    end
  8'h0A:
    begin
       // Write to register 5
       data <= ‘ADV7185_REGISTER_5;
       if (ack)
 state <= state+1;
    end
  8’h0B:
    begin
       // Write to register 6
       data <= 8’h00; // Reserved register, write all zeros
       if (ack)
 state <= state+1;
    end
  8’h0C:
    begin
       // Write to register 7
       data <= ‘ADV7185_REGISTER_7;
       if (ack)
 state <= state+1;
    end
  8’h0D:
    begin
       // Write to register 8
       data <= ‘ADV7185_REGISTER_8;
       if (ack)
 state <= state+1;
    end
  8’h0E:
    begin
       // Write to register 9
       data <= ‘ADV7185_REGISTER_9;
       if (ack)
 state <= state+1;
    end
  8’h0F: begin
```

```
    // Write to register A
    data <= ‘ADV7185_REGISTER_A;
  if (ack)
    state <= state+1;
 end
 8’h10:
   begin
      // Write to register B
      data <= ‘ADV7185_REGISTER_B;
      if (ack)
state <= state+1;
   end
 8’h11:
   begin
      // Write to register C
      data <= ‘ADV7185_REGISTER_C;
      if (ack)
state <= state+1;
   end
 8’h12:
   begin
      // Write to register D
      data <= ‘ADV7185_REGISTER_D;
      if (ack)
state <= state+1;
   end
 8’h13:
   begin
      // Write to register E
      data <= ‘ADV7185_REGISTER_E;
      if (ack)
state <= state+1;
   end
 8’h14:
   begin
```

```
      // Write to register F
      data <= ‘ADV7185_REGISTER_F;
      if (ack)
state <= state+1;
   end
 8’h15:
   begin
      // Wait for I2C transmitter to finish
      load <= 1’b0;
      if (idle)
state <= state+1;
   end
 8’h16:
   begin
      // Write address
      data <= 8’h8A;
      load <= 1’b1;
      if (ack)
state <= state+1;
   end
 8’h17:
   begin
      data <= 8’h33;
      if (ack)
state <= state+1;
   end
 8’h18:
   begin
      data <= ‘ADV7185_REGISTER_33;
      if (ack)
state <= state+1;
   end
 8’h19:
   begin
      load <= 1’b0;
      if (idle)
state <= state+1;
   end

 8'h1A: begin
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
      state <= state+1;
 end
 8'h1B:
   begin
      data <= 8'h33;
      if (ack)
state <= state+1;
   end
 8'h1C:
   begin
      load <= 1'b0;
      if (idle)
state <= state+1;
   end
 8'h1D:
   begin
      load <= 1'b1;
      data <= 8'h8B;
      if (ack)
state <= state+1;
   end
 8'h1E:
   begin
      data <= 8'hFF;
      if (ack)
state <= state+1;
   end
 8'h1F:
   begin
      load <= 1'b0;
      if (idle)
state <= state+1;
   end
 8'h20:
   begin
      // Idle
      if (old_source != source) state <= state+1;
      old_source <= source;
   end
 8'h21: begin
    // Send ADV7185 address
    data <= 8'h8A;
    load <= 1'b1;
    if (ack) state <= state+1;
 end
 8'h22: begin
    // Send subaddress of register 0
    data <= 8'h00;
    if (ack) state <= state+1;
 end
 8'h23: begin
    // Write to register 0
    data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
    if (ack) state <= state+1;
 end
 8'h24: begin
    // Wait for I2C transmitter to finish
    load <= 1'b0;
    if (idle) state <= 8'h20;
 end
      endcase
```

```
endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
  state <= 0;
  ack <= 0;
       end
     else
       case (state)
 8'h00: // idle
   begin
      scl <= 1'b1;
      sdai <= 1'b1;
      ack <= 1'b0;
      idle <= 1'b1;
      if (load)
begin
   ldata <= data;
   ack <= 1'b1;
   state <= state+1;
end
   end
 8'h01: // Start
   begin
      ack <= 1'b0;
      idle <= 1'b0;
      sdai <= 1'b0;
      state <= state+1;
   end
 8'h02:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h03: // Send bit 7
   begin
      ack <= 1'b0;
      sdai <= ldata[7];
      state <= state+1;
   end
 8'h04:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h05:
   begin
```

```
      state <= state+1;
   end
 8'h06:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h07:
   begin
      sdai <= ldata[6];
      state <= state+1;
   end
 8'h08:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
 8'h09:
   begin
      state <= state+1;
   end
 8'h0A:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
 8'h0B:
   begin
      sdai <= ldata[5];
      state <= state+1;
   end
 8'h0C:
   begin
      scl <= 1'b1;
      state <= state+1;
```

```
  end
8'h0D:
  begin
     state <= state+1;
  end
8'h0E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0F:
  begin
     sdai <= ldata[4];
     state <= state+1;
  end
8'h10:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h11:
  begin
     state <= state+1;
  end
8'h12:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h13:
  begin
     sdai <= ldata[3];
     state <= state+1;
  end
8'h14:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h15:
  begin
     state <= state+1;
  end
8'h16:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h17:
  begin
     sdai <= ldata[2];
     state <= state+1;
  end
8'h18:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h19:
  begin
     state <= state+1;
  end
8'h1A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
```

```
      state <= state+1;
   end
8'h1C:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
8'h1D:
   begin
      state <= state+1;
   end
8'h1E:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
8'h1F:
   begin
      sdai <= ldata[0];
      state <= state+1;
   end
8'h20:
   begin
      scl <= 1'b1;
      state <= state+1;
   end
8'h21:
   begin
      state <= state+1;
   end
8'h22:
   begin
      scl <= 1'b0;
      state <= state+1;
   end
```

```
 8’h23: // Acknowledge bit
   begin
      state <= state+1;
   end
 8’h24:
   begin
      scl <= 1’b1;
      state <= state+1;
   end
 8’h25:
   begin
      state <= state+1;
   end
 8’h26:
   begin
      scl <= 1’b0;
      if (load)
begin
   ldata <= data;
   ack <= 1’b1;
   state <= 3;
end
      else
state <= state+1;
   end
 8’h27:
   begin
      sdai <= 1’b0;
      state <= state+1;
   end
 8’h28:
   begin
      scl <= 1’b1;
      state <= state+1;
   end
 8'h29:
   begin
      sdai <= 1'b1;
      state <= 0;
   end
       endcase

endmodule
///////////////////////////////////////////////////////////////////////////////
```

A.11 16-bit hex display driver

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
disp_blank, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data; // 16 hex nibbles to display

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Display Clock
   //
   // Generate a 500kHz clock for driving the displays.
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [4:0] count;
   reg [7:0] reset_count;
   reg clock;
   wire dreset;

   always @(posedge clock_27mhz)
     begin
if (reset)
  begin
     count = 0;
     clock = 0;
  end
else if (count == 26)
  begin
     clock = ~clock;
     count = 5'h00;
  end
else
  count = count+1;
     end
```

```
 always @(posedge clock_27mhz)
   if (reset)
     reset_count <= 100;
   else
     reset_count <= (reset_count==0) ? 0 : reset_count-1;

 assign dreset = (reset_count != 0);

 assign disp_clock = ~clock;

 ////////////////////////////////////////////////////////////////////////////
 //
 // Display State Machine
 //
 ////////////////////////////////////////////////////////////////////////////

 reg [7:0] state; // FSM state
 reg [9:0] dot_index; // index to current dot being clocked out
 reg [31:0] control; // control register
 reg [3:0] char_index; // index of current character
 reg [39:0] dots; // dots for a single digit
 reg [3:0] nibble; // hex nibble of current character

 assign disp_blank = 1'b0; // low <= not blanked

 always @(posedge clock)
   if (dreset)
     begin
state <= 0;
dot_index <= 0;
control <= 32'h7F7F7F7F;
     end
   else
     casex (state)
 8'h00:
   begin
      // Reset displays
      disp_data_out <= 1'b0;
      disp_rs <= 1'b0; // dot register
      disp_ce_b <= 1'b1;
      disp_reset_b <= 1'b0;
      dot_index <= 0;
      state <= state+1;
   end

 8'h01:
   begin
      // End reset
      disp_reset_b <= 1'b1;
      state <= state+1;
   end

 8'h02:
   begin
      // Initialize dot register (set all dots to zero)
      disp_ce_b <= 1'b0;
      disp_data_out <= 1'b0; // dot_index[0];
      if (dot_index == 639)
state <= state+1;
      else
dot_index <= dot_index+1;
   end

 8'h03:
   begin
      // Latch dot data
      disp_ce_b <= 1'b1;
      dot_index <= 31; // re-purpose to init ctrl reg
      disp_rs <= 1'b1; // Select the control register
      state <= state+1;
   end

 8'h04:
   begin
      // Setup the control register
      disp_ce_b <= 1'b0;
      disp_data_out <= control[31];
      control <= {control[30:0], 1'b0}; // shift left
      if (dot_index == 0)
state <= state+1;
      else
dot_index <= dot_index-1;
   end

 8'h05:
   begin
      // Latch the control register data / dot data
      disp_ce_b <= 1'b1;
      dot_index <= 39; // init for single char
      char_index <= 15; // start with MS char
      state <= state+1;
      disp_rs <= 1'b0;   // Select the dot register
   end

 8'h06:
   begin
      // Load the user's dot data into the dot reg, char by char
      disp_ce_b <= 1'b0;
      disp_data_out <= dots[dot_index]; // dot data from msb
      if (dot_index == 0)
        if (char_index == 0)
          state <= 5; // all done, latch data
else
begin
```

```
  char_index <= char_index - 1; // goto next char
  dot_index <= 39;
end
      else
dot_index <= dot_index-1; // else loop thru all dots
   end

       endcase

   always @ (data or char_index)
     case (char_index)
       4'h0:    nibble <= data[3:0];
       4'h1:    nibble <= data[7:4];
       4'h2:    nibble <= data[11:8];
       4'h3:    nibble <= data[15:12];
       4'h4:    nibble <= data[19:16];
       4'h5:    nibble <= data[23:20];
       4'h6:    nibble <= data[27:24];
       4'h7:    nibble <= data[31:28];
       4'h8:    nibble <= data[35:32];
       4'h9:    nibble <= data[39:36];
       4'hA:    nibble <= data[43:40];
       4'hB:    nibble <= data[47:44];
       4'hC:    nibble <= data[51:48];
       4'hD:    nibble <= data[55:52];
       4'hE:    nibble <= data[59:56];
       4'hF:    nibble <= data[63:60];
     endcase

   always @(nibble)
     case (nibble)
       4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
       4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
       4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
       4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
       4’h4: dots <= 40’b00011000_00010100_00010010_01111111_00010000;
       4’h5: dots <= 40’b00100111_01000101_01000101_01000101_00111001;
       4’h6: dots <= 40’b00111100_01001010_01001001_01001001_00110000;
       4’h7: dots <= 40’b00000001_01110001_00001001_00000101_00000011;
       4’h8: dots <= 40’b00110110_01001001_01001001_01001001_00110110;
       4’h9: dots <= 40’b00000110_01001001_01001001_00101001_00011110;
       4’hA: dots <= 40’b01111110_00001001_00001001_00001001_01111110;
       4’hB: dots <= 40’b01111111_01001001_01001001_01001001_00110110;
       4’hC: dots <= 40’b00111110_01000001_01000001_01000001_00100010;
       4’hD: dots <= 40’b01111111_01000001_01000001_01000001_00111110;
       4’hE: dots <= 40’b01111111_01001001_01001001_01001001_01000001;
       4’hF: dots <= 40’b01111111_00001001_00001001_00001001_00000001;
     endcase

endmodule
```

References

[1] C. Terman and I. Chuang, “6.111 Handouts,” [Online Document], Sep 2005, [cited 2007 Dec 12], Available HTTP: http://web.mit.edu/6.111/www/f2005/handouts.html