Do-It-Yourself newscast

Obinna Anyanwu and Jyotsna Venkataramanan

Abstract

This report describes the design and implementation of a do-it-yourself newscast. It allows the user to use a camera to take snapshots, add captions to the pictures, and change the background of the newscast and more. The chroma keying module allows the newscaster to be separated from a blue background. A snapshot button allows the user to store a smaller version of a camera frame to the zbt. This can then be expanded for a "fullscreen" version. The final version of our project allows the user to combine all these functions to create their own newscast.

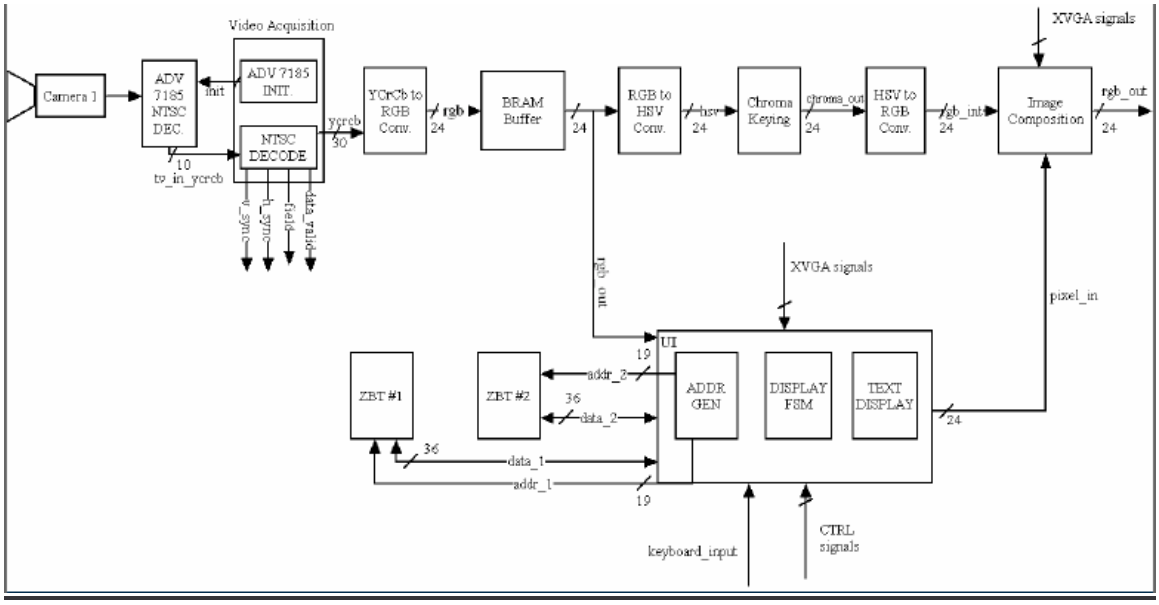
Table of Contents

1. Overview

The goal of this project is to create a virtual news studio. In modern news production we have a reporter, an artificial news background, some image in a smaller box relating to the story and a corresponding caption. To do so we want to take a live video feed of a newscaster in front of a green screen and superimpose it upon a image of the news studio. The newscaster will also have a live audio feed. The picture and caption are loaded into the FPGA beforehand and can be switched during operation to represent the flow of news during program. As the reporter segways into the story, the producer can choose to take the boxed image full screen. This would either be just an image or another live feed. The reporter provides live commentary.

This project can be broken up into a few major parts, input, processing, control, and output. The input part relates to the main live feed, the possible other live feed and sound input. The processing involves recognizing the reporter from in front of the blue screen and overlaying the reporter, small picture and caption in front of the studio background. Control involves the entire image switching as well as the fullscreen mode. The output is the final product of our project.

2. Block Diagram



3. Description of Modules

3.1 Obinna's modules

3.1.1 Video Acquisition (Obinna)

Video entered the system using a standard NTSC camera. According to the NTSC standard, 720x480 active video (858x486 including blanking) is transmitted using a 27MHz clock. This composite video data was processed by the ADV7185 NTSC video decoder chip on the labkit. External chips on the labkit often have to be initialized, and this usually involves a complex pattern of steps that must be followed to ensure correct operation. Fortunately, code from the Fall 2005 website written by Nathan Ickes deals with this process. After initialization, the subsequent data from the chip was YCrCb data, but it was sent 10 bits at a time. The ntsc_decode module, also from the Fall 2005 website and written by Javier Castro, took care of this issue. By using a state machine to determine the flow of data from the chip, the module extracts the Y, Cr, and Cb values to generate the 30-bit YCrCb data necessary to perform the video format conversions later on.

3.1.2 Video Buffering (Obinna)

As mentioned previously, NTSC video runs at 27MHz, but the goal of the project was to display the final product onto a computer screen at XVGA resolution. This raised the issue of different clock domains, since XVGA video runs at 65MHz. In situations involving varying clock domains, it is necessary to use a buffer. The most straightforward and common method is to buffer an entire frame of video using both ZBT RAM chips. However, our project was constrained by the fact that one ZBT would be used for the implementation of the snapshot feature. As a result, it became necessary to

modify code from the Fall 2005 website to suit our needs. The code (ntsc_to_zbt and vram_display modules) originally displayed a black and white image using a single ZBT RAM as a buffer, and this was accomplished by discarding the Cr and Cb components and storing four pixels of data into a single memory location. To fit our purposes, the code was modified to store two pixels (with all YCrCb component data) into a single memory location at the cost of 12 bits of data per pixel (since the YCrCb data from the ntsc_decode module was 30 bits).

3.1.3 YCrCb to RGB Conversion (Obinna)

In addition to running at a different clock rate, XVGA video requires RGB data for display. To this end, a YCrCb to RGB converter written by Xilinx was utilized (taken from the xapp283 datasheet and related files). Using a binary decimal representation, the converter outputs 24-bit RGB data three cycles after input according to the following set of equations:

$$R = Y + 1.402*Cr$$
$$G = Y - 0.344*Cb - 0.714*Cr$$
$$B = Y + 1.772*Cb$$

Following the conversion, the RGB data was output to the image composition module, the RGB to HSV conversion module, and the control module (to allow for snapshots).

3.1.4 RGB to HSV Conversion (Obinna)

To perform chroma keying, we decided to make use of the HSV color space to do our color recognition. HSV stands for hue, saturation, and value:

- Hue is the absolute color
- Saturation is the intensity of the color
- Value represents the brightness

By using HSV, we were able to separate our pixel data into an absolute color and two secondary values, making the filtering process extremely simple. The RGB to HSV module was written using two pipelined dividers created by the Xilinx Coregen to perform the divisions to calculate the hue and saturation. The HSV data output was 25 bits (9 for hue, 8 for saturation and value). Saturation and value usually range from zero to one, but for the purposes of calculation, they were rescaled from 0 to 255. I decided not to rescale the hue, since it normally ranges from 0 to 360. The delay from input to output was 22 cycles, and this is mostly due to the algorithm used to perform the division. Using the Xilinx division module, a 16-bit division takes 18 cycles, and accounts for almost all of the delay.

3.1.5 Chroma Keying (Obinna)

The output from the RGB to HSV conversion module was fed into the chroma keying module. For the purpose of this project, the only data necessary is the hue data. A baseline hue value is set for the background value, along with an allowable range. If the color seen by the camera is within the range, it is recognized as part of the background and is replaced by video data from the control system. If not, the data from the camera is kept, giving the desired effect (similar to the weatherman effect). This effect is accomplished with a simple keep_pixel flag, which is sent to the image composition module along with RGB data from the camera (delayed by 22 cycles to account for the delay in converting to HSV).

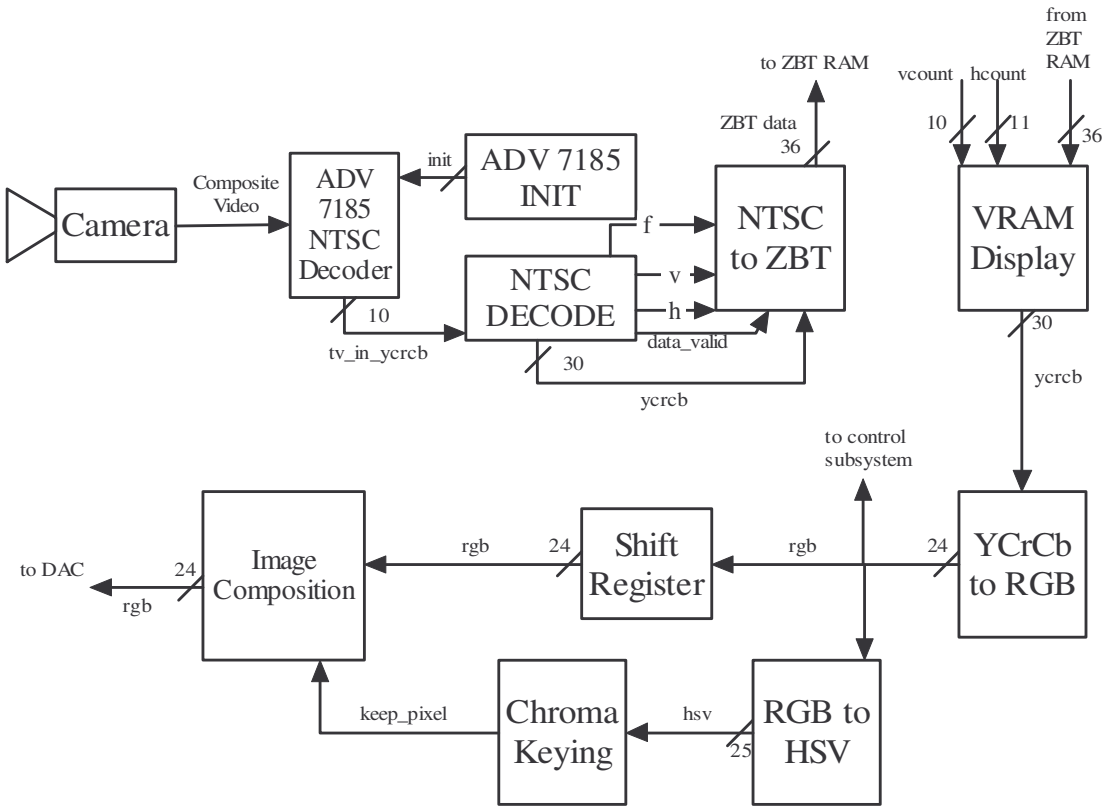
3.1.6 Image Composition (Obinna)

The way the system is designed, the image composition module is nothing more than a simple or statement. If the keep_pixel signal is high, the pixel displayed comes from the camera; if not, it comes from the control module, which provides the background image as well as the snapshots and text to be displayed. The output of this module is sent to the ADV7125 DAC for display on the screen.

3.1.7 Testing and Debugging

The testing and debugging of the modules relating to the processing and display of video was straightforward once the operation of the RGB to HSV converter had been verified. Video in and of itself is an extremely useful debugging tool; if it looks wrong, there's probably something wrong with the code. Both the image composition and chroma keying modules were tested at the same time. Using an RGB to HSV calculator, I calibrated the system to filter out the blue pillars in the lab. To confirm correct operation, I had the image composition module replace the pillars with color bars. The test was successful, but during testing we noticed that the camera was not always perceiving colors consistently, which reduced the effectiveness of the filter.

The RGB to HSV converter was tested by writing a testbench and running simulations in Modelsim. Before I was made aware of the Xilinx Coregen dividers, I used a simple divide module and wrote a converter that provided pixel data on every clock cycle to verify that the algorithm was correct. By generating known RGB values, I could view the waveforms in Modelsim and confirm the HSV values by calculating them by hand. Without Modelsim as a tool, it would have been extremely time consuming to debug this module correctly, as loading a new file into the FPGA numerous times and using the logic analyzer is not the best use of time. Whenever possible, it is most effective to test modules individually, and we tried to keep this rule in mind in designing the project.



Video Acquisition and Processing Subsystem

3.2 Jyotsna's modules

3.2.1 UI module

The UI module was used to let the user control what was being displayed onto the screen. The screen was a XVGA screen meaning that the size of the screen was 1024x767. This meant that all of the components that were being shown on the screen had to match this resolution. The concept of the UI was very easy, it allowed the user to move around by dragging and dropping the picture/snapshot box as well as the text box. This meant that there was a mouse and there were two colored boxes, one for the snapshot which had a size of 360x243 and one for the text box which had a size of 512x48. The user could click on the left corner of any of the boxes with the mouse and move the boxes around. This was simply done by saying that when the box corner was being clicked on allow the position of the box to equal the position of the mouse. It was relatively simple to implement.

3.2.2 Mouse module

The code for the mouse module was previously provided from Fall2005. However there were several changes that had to be made to this code to make it compatible with our system. The original code had a 10x10 block that could be used as a mouse and allowed the user to move the mouse around and see the box moving with it as if it were a cursor.

However this code was only compatible with a 50MHz clock and our system was running a 65MHz clock. The problem with running a faster clock was of course that the mouse would have to be able to support the speed of the clock. The delays internally had to be changed to match the clock. Since I had very little knowledge of how this system worked it involved time on my part to understand the code and try out different values. In the end the mouse was made compatible with the 65MHz clock and worked functionally. The original mouse code had certain issues because it only worked for sometime before it started displaying a very jumpy, jerky and non-responsive mouse. I was able to improve the code by adding delays in certain places and such and was able to make the clock last longer. The block was changed to represent an arrow but I was not able to integrate this into the project in the last minute.

3.2.3 Text module

The text module code was easy enough to implement. Some sample code was provided as part of the Fall 2005 handout page and I used this code to implement this module. Certain changes had to of course be made to the code. The code had to be made compatible with our system. The code had to be able to be on top of every other module so that had to be taken into account. It was made a function of x and y so we could move it around in the UI module. The size of the text was also made a lot bigger. There were some problems in implementing this code, the original code was difficult to understand and therefore using it was quite tedious. In the end it worked quite well.

3.2.4 Snapshot module

This module allowed the user to take a frame of the camera display shrink it down and store it in ZBT memory. Up to four snapshots could be stored in memory and the user could decide which picture to show using the switches on the labkit. The snapshot button was the down button. It was debounced and fed into the module. To make sure that the signal lasted at least one cycle, we waited until the person had lifted their finger off the button and then waited for one cycle to complete. This was a simple piece of code and gave us the signal ‘shot’. This meant that if a person kept holding down the button, only the latest frame would be grabbed. The signal ‘shot’ determined when the picture was going to be taken. The frame was then stored into the zbt, the addresses were increasing sequentially. Half the horizontal pixels and half the vertical pixels were removed so that meant that the picture was one quarter the original screen. There was a variable ‘n’ which decided where the picture was going to be stored i.e. what the starting address was.

The display of the snaphot module was done within the snapshot module itself. It allowed for easy control of the address module. The display part simply looked at what the starting address was depending on the switches and then displayed the snapshot on the screen wherever the UI placed it. The interfacing caused a lot of issues. The fullscreen module simply doubled all the pixels and then placed them on the screen to cover the full size of 720x486. The fullscreen part was controlled by another switch.

3.2.5 Display

The display that was used was an XVGA display so the resolution was 1024x768. The initial video display code only displayed a box of 720x480 and this was the code that I

used. Therefore the live feed was only displayed in a 720x480 box, however the user could still move the snapshot to outside this box and the text could also be moved as such. This display allowed the user to have more flexibility and choice in where to place the snapshot and text yet was not intentional.

3.2.6 Testing and debugging

This project was very interesting because I had the chance to work on video which was my initial aim. However since both teammates were working on video, there was a large part of code we both required and therefore integration was very complicated. This meant that a lot of time was spent in integrating and a lot of the project was neglected because of that. Some other things that I had added but did not have time to integrate due to time constraints was showing more that one picture on the screen. This was a complicated module because I had to redesign my snapshot module. It allowed the user to move around multiple snapshots and be shown on the screen at the same time. I was also working on resizing module that allowed the user to multiply the size of the picture n times but time did not permit me to finish.

Our presentation was plagued by the demo curse and at the moment the cameras turned on, nothing worked. The parts that we weren't able to show Prof.Terman were the fullscreen function although we did manage to demonstrate it to our TA. If I were to do this project again I would begin the integration of the different modules a lot earlier. This might have given me time to finish some of the ambitious projects I was working towards.

4. Verilog Code

4.1 Chroma Keying

```
module chroma_key(clk, rst, hcount, vcount, h, s, v, keep_pixel);
input clk, rst;
// HSV is only necessary for hue detection, so this data is the hue
data generated
// in the HSV to RGB converter
input[8:0] h;    // range: (0,360]
input[7:0] s, v; // range: [0,255]
input[10:0] hcount;
input[9:0] vcount;

// 0 if pixel is in the background, 1 otherwise; image composer will
replace
// any pixel in the background with that of the stored image
output keep_pixel;

reg keep_pixel;

parameter DELTA = 35;  // Tolerance allowable for picture filtering
parameter HUE_FILTER = 216; // Background hue value  (216,200,120)

always @ (posedge clk)
begin
      if (rst == 1) begin
            keep_pixel <= 1;
            end
      else begin
            if ((hcount < 720) & (vcount < 486)) begin
            // Only apply chroma keying to the displayed video area,
since the video does not
            // occupy the entire screen
                  if (((HUE_FILTER - DELTA) < h) & ((HUE_FILTER + DELTA)
> h)) keep_pixel <= 0;
                  else keep_pixel <= 1;
                  end
            else keep_pixel <= 0;
            end
end

endmodule
```

4.2 Image Composition

```
module image_composition(keep_pixel, camera_pixel, ctrl_pixel,
composite_pixel_out);
// The image_composition module combines the keep_pixel signal from the
chroma keying module with the
// background images provided by the control module.  When the
keep_pixel signal equals zero, it represents
// a pixel location that can be replaced by the background image.
// ** Tested by overlaying a checkerboard pattern over a series of
```

```
color bars; only the black color bar regions
// ** should be replaced by the checkerboard pattern.
//input clk, rst;
input keep_pixel;                  // 1 if camera pixel should be kept,
0 otherwise
input[23:0] camera_pixel;                          // RGB pixels from
the camera
input[23:0] ctrl_pixel;                        // RGB pixels from the
control subsystem
//input[10:0] hcount;                             // Horizontal pixel
count signals from the VGA sync module
//input[9:0] vcount;                               // Vertical pixel
count signals from the VGA sync module
output[23:0] composite_pixel_out;  // Composited RGB pixel output (to
VGA DAC)

//reg[23:0] composite_pixel_out;

assign composite_pixel_out = (keep_pixel == 0) ? ctrl_pixel :
camera_pixel;

endmodule
```

4.3 UI Module

```
module user_int1(clk, reset, hcount, vcount, mx, my, max_x, max_y,
btn_click, pic_p);

      input clk, reset;
      input [10:0] hcount;
      input [9:0] vcount;
      input [11:0] mx, my;
      input [2:0] btn_click;

      output [2:0] pic_p;
      output [10:0] max_x;
      output [9:0] max_y;

      wire [10:0] mouse_x = mx[10:0];
      wire [9:0] mouse_y = my[9:0];
      wire btn = btn_click[2];

      reg [10:0] pic_x, text_x, max_x;
      reg [9:0] pic_y, text_y, max_y;
      always @(btn or mouse_x or mouse_y) begin
            if (reset) begin
                  text_x <= 11'd50;
                  text_y <= 10'd300;
                  pic_x <= 11'd30;
                  pic_y <= 10'd30;
                  max_x <= 1023 - 10;
                  max_y <= 767 - 10;
            end
            else if (btn && (text_x >= mouse_x) && (text_x <= mouse_x +
10) && (text_y >= mouse_y) && (text_y <= mouse_y + 10)) begin
                  text_x <= mouse_x;
```

```
                text_y <= mouse_y;
                max_x <= 1023 - 512;
                max_y <= 767 - 48;
            end
            else if (btn && (pic_x >= mouse_x) && (pic_x <= mouse_x +
10) && (pic_y >= mouse_y) && (pic_y <= mouse_y + 10)) begin
                pic_x <= mouse_x;
                pic_y <= mouse_y;
                max_x <= 1023 - 360;
                max_y <= 767 - 240;
            end
            else begin
                pic_x <= pic_x;
                pic_y <= pic_y;
                max_x <= 1023 - 10;
                max_y <= 767 - 10;
            end
      end

      wire [2:0] mouse_pixel, picture_pixel, text_pixels;
      blob mouse(mouse_x, mouse_y, hcount, vcount, mouse_pixel);
            defparam mouse.COLOR = 3'b001;

      blob picture(pic_x, pic_y, hcount, vcount, picture_pixel);
            defparam picture.WIDTH = 11'd360;
            defparam picture.HEIGHT = 10'd240;
            defparam picture.COLOR = 3'b100;

      blob texts(text_x, text_y, hcount, vcount, text_pixels);
            defparam texts.WIDTH = 11'd512;
            defparam texts.HEIGHT = 10'd48;
            defparam texts.COLOR = 3'b011;

      reg [2:0] pic_p;
      always @(posedge clk) begin
            pic_p <= mouse_pixel[0] ? mouse_pixel : text_pixels[0] ?
text_pixels : picture_pixel;
      end

endmodule
```

4.4 Snaphot module

```
module snapshot(vclock, reset, snap, ram_data_in, xhcount, xvcount, hsync, vsync,
blank,
                xhsync, xvsync, xblank, vram_we, vram_addr, disp_data_out, x_disp,
y_disp, switch, fullscreen);
      input vclock, reset;
      input snap;
      input [10:0] xhcount;
      input [9:0] xvcount;
      input hsync, vsync, blank;
      input [35:0] ram_data_in;
      input [10:0] x_disp;
```

```
input [10:0] y_disp;
input [2:0] switch;
input fullscreen;

output xhsync, xvsync, xblank;
output vram_we;
output [18:0] vram_addr;
output [35:0] disp_data_out;

assign xhsync = hsync;
assign xvsync = vsync;
assign xblank = blank;

wire [10:0] hcount;
wire [9:0] vcount;
assign hcount = xhcount - 5;
assign vcount = xvcount - 5;

reg vram_we;
reg [18:0] vram_addr;
reg [35:0] disp_data_out;

reg shot;

always @ (posedge vclock)
	begin
		if ((snap == 1) & (vcount == 767) & (hcount == 1024)) begin
				shot <= snap;
			end
		else if ((snap == 0) & (vcount == 767) & (hcount == 1024)) begin
				shot <= 0;
			end
		else shot <= shot;
	end

reg hc, vc;
reg [1:0] n;
reg hc1, vc1;

//720 x 486 pixels total
//display box = 360x243

always @(negedge shot) n <= reset ? 0 : (n == 3) ? 0 : n + 1;

always @ (posedge vclock) begin
	if (shot) begin
	//	snapshot
		if ((vcount == 9) & (hcount == 9)) begin
```

```
        //wait till at beginning of screen and then start storing
                                case (n)
                                        2'd0: vram_addr <= 0;
                                        2'd1: vram_addr <= 87500;
                                        2'd2: vram_addr <= 172000;
                                        2'd3: vram_addr <= 262500;
                                endcase
                                vram_we <= 1;
                                hc <= 1;                        //hc stores every other pixel
                                vc <= 1;
                        end
                        else begin
                                hc <= ~hc;
                                vc <= (hcount == 719) ? ~vc : vc;
                                vram_we <= (hc & vc & (hcount <= 719) & (vcount <=
485)) ? 1 : 0;
                                vram_addr <= (hc & vc & (hcount <= 719) & (vcount <=
485)) ? vram_addr + 1 : vram_addr;
                        end
                end

                else begin                      //display picture
                        if (~fullscreen) begin
                                if ((vcount == y_disp) & (hcount == x_disp)) begin
                                        case (switch)
                                                3'd1: vram_addr <= 0;
                                                3'd2: vram_addr <= 87500;
                                                3'd3: vram_addr <= 172000;
                                                3'd4: vram_addr <= 262500;
                                                default: vram_addr <= 0;
                                        endcase
                                        vram_we <= 0;
                                        disp_data_out <= ram_data_in;
                                end
                                else begin  // replace 36'b0 with color bar generator
                                        vram_we <= 0;
                                        vram_addr <= ((hcount >= x_disp) & (vcount >=
y_disp) & (hcount <= 359 + x_disp) & (vcount <= 242 + y_disp)) ? vram_addr + 1 :
vram_addr;
                                        disp_data_out <= ((hcount >= x_disp) & (vcount >=
y_disp) & (hcount <= 359 + x_disp) & (vcount <= 242 + y_disp)  & (hcount <= 359 +
x_disp) & (vcount <= 242 + y_disp))
                                                                                ? (switch == 0)
? 36'd0 : ram_data_in : 36'd0;
                                end
                        end

                        else begin
```

```
                        if ((vcount == 0) & (hcount == 0)) begin
                        //n <= reset ? 0 : (n == 3) ? 0 : n + 1;
                                case (switch)
                                        3'd1: vram_addr <= 0;
                                        3'd2: vram_addr <= 87500;
                                        3'd3: vram_addr <= 172000;
                                        3'd4: vram_addr <= 262500;
                                        default: vram_addr <= 0;
                                endcase
                                vram_we <= 0;
                                disp_data_out <= ram_data_in;
                                hc1 <= 0;
                                vc1 <= 0;
                        end
                        else begin
                                hc1 <= ~hc1;
                                vc1 <= (hcount == 719) ? ~vc1 : vc1;
                                if ((hcount == 719) & (vc1 == 1)) begin
                                        vram_addr <= vram_addr - 359;
                                        disp_data_out <= ram_data_in;
                                end
                                else begin
                                        vram_addr <= ((hc1 == 0) & (hcount <=
719) & (vcount <= 483)) ? vram_addr + 1 : vram_addr;
                                        disp_data_out <= ram_data_in;
                                end
                        end
                end
        end
    end

endmodule
```

4.5 RGB to HSV

```
module rgb_to_hsv_sub2(clk, rst, r, g, b, h, s, v);
// h, s, and v are output 22 cycles after an input
input clk, rst;
input[7:0] r, g, b;
output[7:0] s, v;
output[8:0] h;
// h ranges from (0, 360] degrees
// s and v ranges from [0,255]

reg[7:0] s, v, s_del, v_del;
reg[8:0] h;

reg[7:0] v_del_final[19:0];  // Delays v so that h, s, v data lines up
correctly

reg[7:0] min;
```

```
wire[7:0] delta;
reg signed [9:0] h_int;

reg[15:0] sat_dividend, sat_divisor;
reg[15:0] hue_dividend, hue_divisor;
wire[15:0] sat_remainder, hue_remainder;
wire[15:0] sat_quotient, hue_quotient;
wire hue_rfd, sat_rfd;  // Signal from divider (ready for data)
reg[2:0] hue_select; // Used to correctly modify the output of the hue
divider
reg[2:0] hue_select_delay[17:0];  // Accounts for divider delay

pipelined_divider sat_divider(clk, sat_dividend, sat_divisor,
sat_quotient, sat_remainder, sat_rfd);
pipelined_divider hue_divider(clk, hue_dividend, hue_divisor,
hue_quotient, hue_remainder, hue_rfd);

// Calculating the maximum value
always @ (posedge clk)
begin
      if ((r >= g) & (r >= b))
            v_del <= r;
      else if ((g >= r) & (g >= b))
            v_del <= g;
      else if ((b >= r) & (b >= g))
            v_del <= b;
end


// Calculating the minimum value
always @ (posedge clk)
begin
      if ((r <= g) & (r <= b))
            min <= r;
      else if ((g <= r) & (g <= b))
            min <= g;
      else if ((b <= r) & (b <= g))
            min <= b;
end

assign delta = v_del - min;
// Calculating the saturation division parameters
always @ (posedge clk)
begin
      sat_divisor <= v_del;
      sat_dividend <= 16'd255*delta;
end

always @ (posedge clk)
begin
            s_del <= sat_quotient;
end

// Calculating the hue division parameters
always @ (posedge clk)
begin
```

```
            hue_divisor <= delta;
            if (r == v_del) begin // between yellow and magenta
                if (g >= b) begin
                  hue_dividend <= 16'd60*(g - b);
                  hue_select <= 0;
                  end
                else begin
                  hue_dividend <= 16'd60*(b - g);
                  hue_select <= 1;
                  end
                end
            else if (g == v_del) begin  // between cyan and yellow
                if (b >= r) begin
                  hue_dividend <= 16'd60*(b - r);
                  hue_select <= 2;
                  end
                else begin
                  hue_dividend <= 16'd60*(r - b);
                  hue_select <= 3;
                  end
                end
            else begin // between magenta and cyan
                if (r >= g) begin
                  hue_dividend <= 16'd60*(r - g);
                  hue_select <= 4;
                  end
                else begin
                  hue_dividend <= 16'd60*(g - r);
                  hue_select <= 5;
                  end
                end
end

// Since the divider accepts data every cycle, we must keep track of
which operation was being performed so that
// the correct addition/subraction can be performed (division takes 18
cycles)
always @ (posedge clk)
begin
            hue_select_delay[0] <= hue_select;
            hue_select_delay[1] <= hue_select_delay[0];
            hue_select_delay[2] <= hue_select_delay[1];
            hue_select_delay[3] <= hue_select_delay[2];
            hue_select_delay[4] <= hue_select_delay[3];
            hue_select_delay[5] <= hue_select_delay[4];
            hue_select_delay[6] <= hue_select_delay[5];
            hue_select_delay[7] <= hue_select_delay[6];
            hue_select_delay[8] <= hue_select_delay[7];
            hue_select_delay[9] <= hue_select_delay[8];
            hue_select_delay[10] <= hue_select_delay[9];
            hue_select_delay[11] <= hue_select_delay[10];
            hue_select_delay[12] <= hue_select_delay[11];
            hue_select_delay[13] <= hue_select_delay[12];
            hue_select_delay[14] <= hue_select_delay[13];
            hue_select_delay[15] <= hue_select_delay[14];
            hue_select_delay[16] <= hue_select_delay[15];
```

```
            hue_select_delay[17] <= hue_select_delay[16];
end

// Calculating the hue divider (assigning correct outputs based on the
operation selected)
always @ (posedge clk)
begin
            case (hue_select_delay[17])
            3'b000 : h_int <= {2'b0, hue_quotient};
            3'b001 : h_int <= -hue_quotient;
            3'b010 : h_int <= 8'd120 + hue_quotient;
            3'b011 : h_int <= 8'd120 - hue_quotient;
            3'b100 : h_int <= 8'd240 + hue_quotient;
            3'b101 : h_int <= 8'd240 - hue_quotient;
            default : h_int <= 0;
            endcase
end

wire signed [9:0] test = - hue_quotient;

always @ (posedge clk)
begin
    if (h_int < 0) h <= h_int + 'd360;
    else h <= h_int[8:0];
end

// Delay s one cycle to sync up with h and v
always @ (posedge clk)
begin
    s <= s_del;
end

// Delay v for 21 cycles to sync up with other values
always @ (posedge clk)
begin
   v_del_final[0] <= v_del;
   v_del_final[1] <= v_del_final[0];
   v_del_final[2] <= v_del_final[1];
   v_del_final[3] <= v_del_final[2];
   v_del_final[4] <= v_del_final[3];
   v_del_final[5] <= v_del_final[4];
   v_del_final[6] <= v_del_final[5];
   v_del_final[7] <= v_del_final[6];
   v_del_final[8] <= v_del_final[7];
   v_del_final[9] <= v_del_final[8];
   v_del_final[10] <= v_del_final[9];
   v_del_final[11] <= v_del_final[10];
   v_del_final[12] <= v_del_final[11];
   v_del_final[13] <= v_del_final[12];
   v_del_final[14] <= v_del_final[13];
   v_del_final[15] <= v_del_final[14];
   v_del_final[16] <= v_del_final[15];
   v_del_final[17] <= v_del_final[16];
   v_del_final[18] <= v_del_final[17];
   v_del_final[19] <= v_del_final[18];
   v <= v_del_final[19];
```

```
   end

endmodule
```

4.6 Labkit

```
//////////////////////////////////////////////////////////////////////////
////////
//
// 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 camera_chain_labkit(beep, audio_reset_b,
                  ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

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

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

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

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

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

             clock_feedback_out, clock_feedback_in,

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

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
```

```
              mouse_clock, mouse_data, keyboard_clock, keyboard_data,

              clock_27mhz, clock1, clock2,

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

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

              switch,

              led,

              user1, user2, user3, user4,

              daughtercard,

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

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   inout  mouse_clock, mouse_data;
      input  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'b0;
   assign ram1_oe_b = 1'b0;
//   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'h0;

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;
```

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

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


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

   wire clk = clock_65mhz;

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(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;


      // 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);

   ////////////////////////////
      //
      // XVGA signal generator
      //
      ////////////////////////////
   wire [10:0] hcount;
```

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


   ////////////////////////////
      //
   // ZBT RAM driver (ram0)
   //
      ////////////////////////////
   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);

   ////////////////////////////
      //
      // Pixel value generation from ZBT RAM (ram0) contents
      //
      ////////////////////////////
   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 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));


  ////////////////////////////
  //
  //  ADV7185 NTSC decoder interface code
  //
  ////////////////////////////
   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));

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

```
      //
   // Writing NTSC data (YCrCb) to ZBT RAM (ram0)
      //
      ////////////////////////////////
   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[29:24],ycrcb[19:14],ycrcb[9:4]}, ntsc_addr, ntsc_data,
ntsc_we,1'b0);

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

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

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

   wire     sw_ntsc = ~switch[7];
   wire     my_we = sw_ntsc ? (hcount[0]==1'd0) : blank;
   wire [18:0]    write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]    write_data = sw_ntsc ? ntsc_data : vpat;

//   wire   write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign       vram_addr = write_enable ? write_addr : vram_addr1;
//   assign       vram_we = write_enable;

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

   // select output pixel data

wire[7:0] r_pixel, g_pixel, b_pixel;


////////////////////////////////////////////////////////////////////////
//////////////////////
      //
      //    Mouse
      //
      ////////////////////////////////////////////////////////////////////
////////////////////////////

      // mouse
   wire [11:0] mx,my;
   wire [2:0]  btn_click;
   ps2_mouse_xy m1(clk, reset, mouse_clock, mouse_data, mx, my,
btn_click);
   defparam    m1.MAX_X = 1023-10;  // max - blob size
   defparam    m1.MAX_Y =    767-10;
```

```
      //gui
      wire [2:0] pic_p;
      wire [10:0] box_x, text_x;
      wire [9:0] box_y, text_y;
      user_int2 mod1(clk, reset, hcount, vcount, mx, my, btn_click,
box_x, box_y, text_x, text_y, pic_p);

      // video output
      wire [2:0] pic_pix;
   assign   pic_pix = pic_p & ~blank;

      ////////////////////////////////////////////////////////////////
//////////////////////////
      //
      //Instantiation of snapshot: takes a picture
      //
      ////////////////////////////////////////////////////////////////
//////////////////////////

      wire xhsync, xvsync, xblank;
      wire snap;
      wire vram1_we;
      wire [18:0] vram1_addr;
      wire shot;

      wire [35:0] ram1_data, pix_data_out;

      //button to take snapshot
      debounce db4(reset, clk, ~button_down, snap);

      wire [2:0] picture;

      assign picture = switch[3] ? 1 : switch[4] ? 2 : switch[5] ? 3 :
(switch[6] ? 4 : 0);

      snapshot sp1(clk, reset, snap, ram1_data, hcount, vcount, hsync,
vsync, blank,
              xhsync, xvsync, xblank, vram1_we, vram1_addr,
pix_data_out, box_x, box_y, picture);

      zbt_6111 zbt2(clk, 1'b1, vram1_we, vram1_addr, {12'b0, r_pixel,
g_pixel, b_pixel}, vram1_read_data,
      ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

      ////////////////////////////////////////////////////////////////
//////////////////////////
      //
      // Text on the screen
      //
      ////////////////////////////////////////////////////////////////
//////////////////////////

      // character display module
   reg [16*8-1:0] cstring;
   wire [2:0]  cdpixel;
   char_string_display cd(clk,hcount,vcount,
```

```
                    cdpixel,cstring,text_x,text_y);
   defparam    cd.NCHAR = 16;
   defparam    cd.NCHAR_BITS = 7;

      reg[7:0] pic_pixel_r, pic_pixel_g, pic_pixel_b;

      // Display of text output
      always @(posedge clk) begin
      pic_pixel_r <= cdpixel[2] ? {8{cdpixel[2]}} : pix_data_out[23:16];
      pic_pixel_g <= cdpixel[1] ? {8{cdpixel[1]}} : pix_data_out[15:8];
      pic_pixel_b <= cdpixel[0] ? {8{cdpixel[0]}} : pix_data_out[7:0];
      end

   // keyboard input
   wire [7:0] ascii;
   wire       char_rdy;
   ps2_ascii_input kbd(clk, reset, keyboard_clock,
                   keyboard_data, ascii, char_rdy);

   reg [3:0] count_kbd = 15;
   reg [7:0] last_ascii;

   always @(posedge clk)
     begin
      count_kbd <= reset ? 15 : (char_rdy ? count_kbd-1 : count_kbd);
      last_ascii <= char_rdy ? ascii : last_ascii;
     end

//   assign cstring = {8{last_ascii}};

    always @(posedge clk) begin
       cstring[7:0] <= (count_kbd==0) ? last_ascii : cstring[7:0];
       cstring[7+'o10:'o10] <= (count_kbd==1) ? last_ascii:
cstring[7+'o10:'o10];
       cstring[7+'o20:'o20] <= (count_kbd==2) ? last_ascii:
cstring[7+'o20:'o20];
       cstring[7+'o30:'o30] <= (count_kbd==3) ? last_ascii:
cstring[7+'o30:'o30];
       cstring[7+'o40:'o40] <= (count_kbd==4) ? last_ascii:
cstring[7+'o40:'o40];
       cstring[7+'o50:'o50] <= (count_kbd==5) ? last_ascii:
cstring[7+'o50:'o50];
       cstring[7+'o60:'o60] <= (count_kbd==6) ? last_ascii:
cstring[7+'o60:'o60];
       cstring[7+'o70:'o70] <= (count_kbd==7) ? last_ascii:
cstring[7+'o70:'o70];

       cstring[7+'o100:'o100] <= (count_kbd==8) ? last_ascii:
cstring[7+'o100:'o100];
       cstring[7+'o110:'o110] <= (count_kbd==9) ? last_ascii:
cstring[7+'o110:'o110];
       cstring[7+'o120:'o120] <= (count_kbd==10) ? last_ascii:
cstring[7+'o120:'o120];
       cstring[7+'o130:'o130] <= (count_kbd==11) ? last_ascii:
cstring[7+'o130:'o130];
       cstring[7+'o140:'o140] <= (count_kbd==12) ? last_ascii:
```

```
cstring[7+'o140:'o140];
       cstring[7+'o150:'o150] <= (count_kbd==13) ? last_ascii:
cstring[7+'o150:'o150];
       cstring[7+'o160:'o160] <= (count_kbd==14) ? last_ascii:
cstring[7+'o160:'o160];
       cstring[7+'o170:'o170] <= (count_kbd==15) ? last_ascii:
cstring[7+'o170:'o170];
    end

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


   //reg [7:0]    pixel;
   wire     b1,hs1,vs1;
      wire b, hs, vs;

   delayN dn1(clk,hsync,hs);  // delay by 3 cycles to sync with ZBT read
   delayN dn2(clk,vsync,vs); // this will probably need to be changed
to account
   delayN dn3(clk,blank,b);  // for the delays introduced throughout
the system

      delayNew dNew1(clk, hs, hs1);
      delayNew dNew2(clk, vs, vs1);
      delayNew dNew3(clk, b, b1);

   //always @(posedge clk)
   //  begin
      //pixel <= {hcount[8:6], 5'b0};
   //  end

   // debugging

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

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


//////////////////////
//
// Video processing modules
//
//////////////////////
wire[8:0] h;
wire[7:0] s, v;
wire[23:0] delay_rgb_out, pixel_out;
wire[7:0] r_delay, g_delay, b_delay;
wire keep_pixel;
wire[7:0] r_output, g_output, b_output;  // RGB data to be sent to the
RGB DAC
assign {r_output, g_output, b_output} = {pixel_out[23:16],
pixel_out[15:8], pixel_out[7:0]};
assign {r_delay, g_delay, b_delay} = {delay_rgb_out[23:16],
```

```
delay_rgb_out[15:8], delay_rgb_out[7:0]};

  //////////////////////////
  //
  // YCrCb to RGB conversion
  //
  //////////////////////////
  ycrcb_to_rgb rgb_conv(r_pixel, g_pixel, b_pixel, clk, reset,
{vr_pixel[17:12], 4'b0}, {vr_pixel[11:6], 4'b0}, {vr_pixel[5:0],
4'b0});

  //////////////////////////
  //
  // RGB to HSV conversion
  //
  //////////////////////////
  rgb_to_hsv_sub2 rgb_to_hsv(.clk(clock_65mhz), .rst(reset),
.r(r_pixel), .g(g_pixel), .b(b_pixel), .h(h), .s(s), .v(v));

  //////////////////////////
  //
  // Chroma keying module
  //
  //////////////////////////
  chroma_key chroma_key_module(.clk(clock_65mhz), .rst(reset),
.hcount(hcount), .vcount(vcount), .h(h), .s(s), .v(v),
.keep_pixel(keep_pixel));

  //////////////////////////
  //
  // Delaying RGB signals
  //
  //////////////////////////
  delayNew RGB_delay(.clk(clock_65mhz),.in({r_pixel, g_pixel,
b_pixel}),.out(delay_rgb_out));

  //////////////////////////
  //
  // Image composition module
  //
  //////////////////////////
  image_composition  image_comp(.keep_pixel(keep_pixel),
.camera_pixel({r_delay, g_delay, b_delay}), .ctrl_pixel({pic_pixel_r,
pic_pixel_g, pic_pixel_b}), .composite_pixel_out(pixel_out));

  // analyzer debug
   assign analyzer1_data = {s,v};
   assign analyzer1_clock = clock_65mhz;
   assign analyzer2_data = {7'd0, h};

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = switch[1] ? {8{pic_pix[0]}} : switch[0] ?
r_output : r_delay;
   assign vga_out_green = switch[1] ? {8{pic_pix[1]}} : switch[0] ?
g_output : g_delay;
```

```
   assign vga_out_blue = switch[1] ? {8{pic_pix[2]}} : switch[0] ?
b_output : b_delay;
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_blank_b = ~b1;
   assign vga_out_hsync = hs1;
   assign vga_out_vsync = vs1;

endmodule  // end of labkit module

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

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

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

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

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

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

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

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

```
   end
endmodule

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

module vram_display(reset,clk,xhcount,xvcount,vr_pixel,
                vram_addr,vram_read_data);
// Modified for use as a color video buffer
   input reset, clk;
   input [10:0] xhcount;
   input [9:0] xvcount;
   output [17:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

      wire [10:0] hcount;
      wire [9:0] vcount;
      assign hcount = xhcount + 2;
      assign vcount = xvcount + 2;
   wire [18:0]     vram_addr = {vcount, hcount[9:1]};

   wire                 hc4 = hcount[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==1'd0) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==1'd1) ? vram_read_data : vr_data_latched;

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

endmodule // vram_display
// display is 720x486
//////////////////////////////////////////////////////////////////////
//////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
```

```
   output out;

   parameter NDELAY = 3;

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

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

endmodule // delayN

module delayNew(clk,in,out);
   input clk;
   input[23:0] in;
   output[23:0] out;
// 21 cycle delay
   reg[23:0] delay_line[19:0];
   reg[23:0] out;

   always @ (posedge clk)
   begin
   delay_line[0] <= in;
   delay_line[1] <= delay_line[0];
   delay_line[2] <= delay_line[1];
   delay_line[3] <= delay_line[2];
   delay_line[4] <= delay_line[3];
   delay_line[5] <= delay_line[4];
   delay_line[6] <= delay_line[5];
   delay_line[7] <= delay_line[6];
   delay_line[8] <= delay_line[7];
   delay_line[9] <= delay_line[8];
   delay_line[10] <= delay_line[9];
   delay_line[11] <= delay_line[10];
   delay_line[12] <= delay_line[11];
   delay_line[13] <= delay_line[12];
   delay_line[14] <= delay_line[13];
   delay_line[15] <= delay_line[14];
   delay_line[16] <= delay_line[15];
   delay_line[17] <= delay_line[16];
   delay_line[18] <= delay_line[17];
   delay_line[19] <= delay_line[18];
   out <= delay_line[19];
   end

      endmodule // delayNew
```