CHROMA KEY COMPOSITING WITH THE FPGA

6.111 FINAL PROJECT REPORT – FALL 2014

DANIEL MOON & THIPOK (BEN) RAK-AMNOUYKIT

Abstract

We have implemented a real-time Chroma Key Compositing system that overlays a static image onto area of the video covered by the "green screen". The system receives video data from the NTSC camera, process the pixel signal and store all pixels in Zero Bus Turnaround memory (ZBT). The signal processing highlights any pixel within chroma key range with a specific constant value before storing it with other pixels in the ZBT. Meanwhile, a static background picture is stored in FPGA's Block RAM memory (BRAM). Given a point on the VGA monitor, a zoom factor, and a focus point for zooming, the picture blob module accesses the BRAM and selects the background picture's pixel that corresponded to the given location on the monitor. In the final stage, we mux the video pixels and the background picture pixels, replacing the video pixel with the picture pixel if it carries the pre-processed chroma key highlighted value. Next, we generate a VGA output signal and display the video feed with the picture composited over chroma key pixels.

TABLE OF CONTENTS

1 Overview

Our final project's report documents and details the design and architecture of our implementation of Chroma Key Compositing. Chroma key compositing is a special effects technique commonplace in fields such as newscasting, video production and movies. Chroma Key Compositing essentially identifies a certain color – referred to as the chroma key – in a video feed, replaces any pixel with the color with corresponding pixel from an image or a video. In the processed video streaming, it is as If we have layered (composited) an image or a video in lieu of the chroma key color. Most modern day's chroma key compositing occurs after the video feed has been taken and post-processed using video editing software, such as Adobe After Effects. For our final project, we have replicated this technique on an FPGA – filtering the chroma key pixels out from the NTSC video feed, storing the processed video data in the ZBT and overlaying the background image onto the chroma key pixel as we mux the image from BRAM and the video from ZBT for the VGA monitor display.

In this report, we introduce our approach with a high-level block diagram. This diagram explains our top-level implementation and general design decisions, which will be further emphasized in greater details in each individual module. For each individual module, we describe the calculations and steps we have taken to arrive at our design decisions. Lastly, we will revisit the overall design process, and explain any issues that we have experienced and resolved.

2 High Level System Block Diagram

The diagram in Figure 1 below shows an overall logical flow of our project. It depicts how data coming in from the NTSC camera and the background image stored in BRAM move through the modules. These blocks represents the storage of the video and the image, the zoom effect for the image display, and the compositing of the image on to the chroma key in the video feed, which is finally outputted onto the VGA display. The selection for chroma key compositing happens in the final stage before a video feed is displayed on the VGA. Additionally, the diagrams show that the first filtering occurs before we store the video in ZBT memory. A possible extension to the project that we could have implemented was to reduce this extra stage of filtering into one final step. We will describe in the comprehensive breakdown of modules why we implemented our modules in such manner.

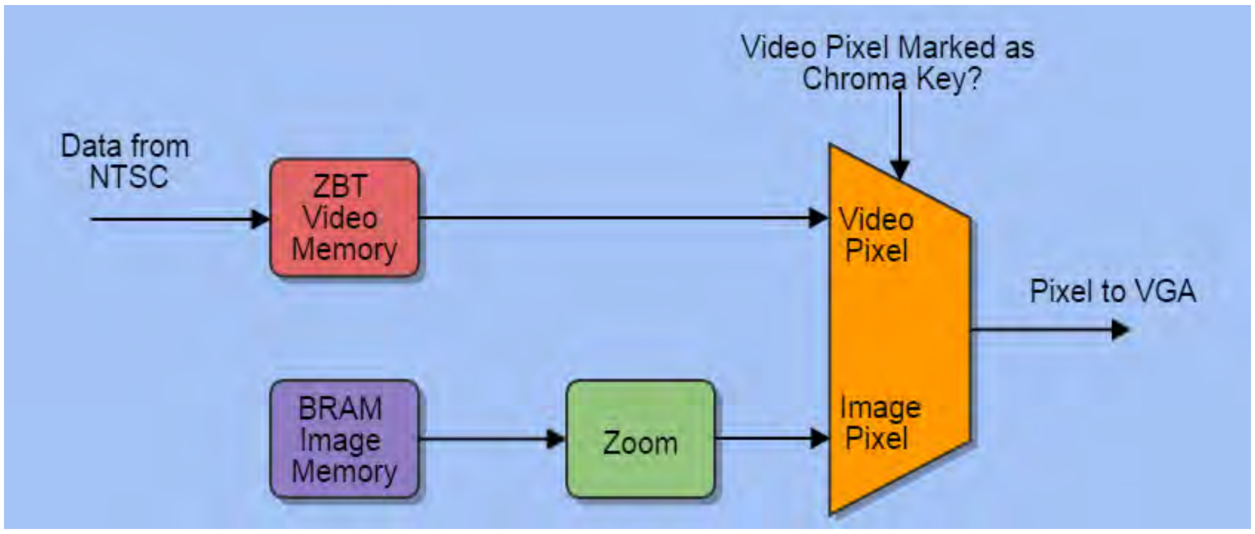


Figure 1: System Block Diagram

The system block diagram depicts the Chroma Key processing on the Video Feed. In the orange muxing block, video pixels are checked whether they were marked as chroma key pixel. If so, that pixel is replaced with the image pixel before outputting the final signal to the VGA display. The compositing image is also influenced by a zoom module, which enlarges the background image before it is displayed over the Chroma Key.

3 Module Descriptions

The proposed Chroma Key Compositing is divided into four parts: memory storage for the video feed, memory storage for the static background image, a zoom module to handle the display of the image, and the chroma keying that occurs on the read out before the video is displayed on the VGA.

Section 3-A discusses how the memory module, the ZBT Video Memory, transforms the video feed into the proper color scheme to be stored after initial checking for the Chroma Key. Section 3-B and section 3-C discuss the storage, readout and processing of the image memory, as well as the thought process behind how the image is zoomed in the zoom module. The ZBT and the BRAM are accessed in parallel, resulting in two asynchronous signals entering the final mux. In section 3-D, the final stage of the Chroma Key Compositing system combines the static image and the video feed to produce the Chroma Key Compositing special effect and output the signal to the VGA screen.

3-A ZBT Video Memory – Daniel

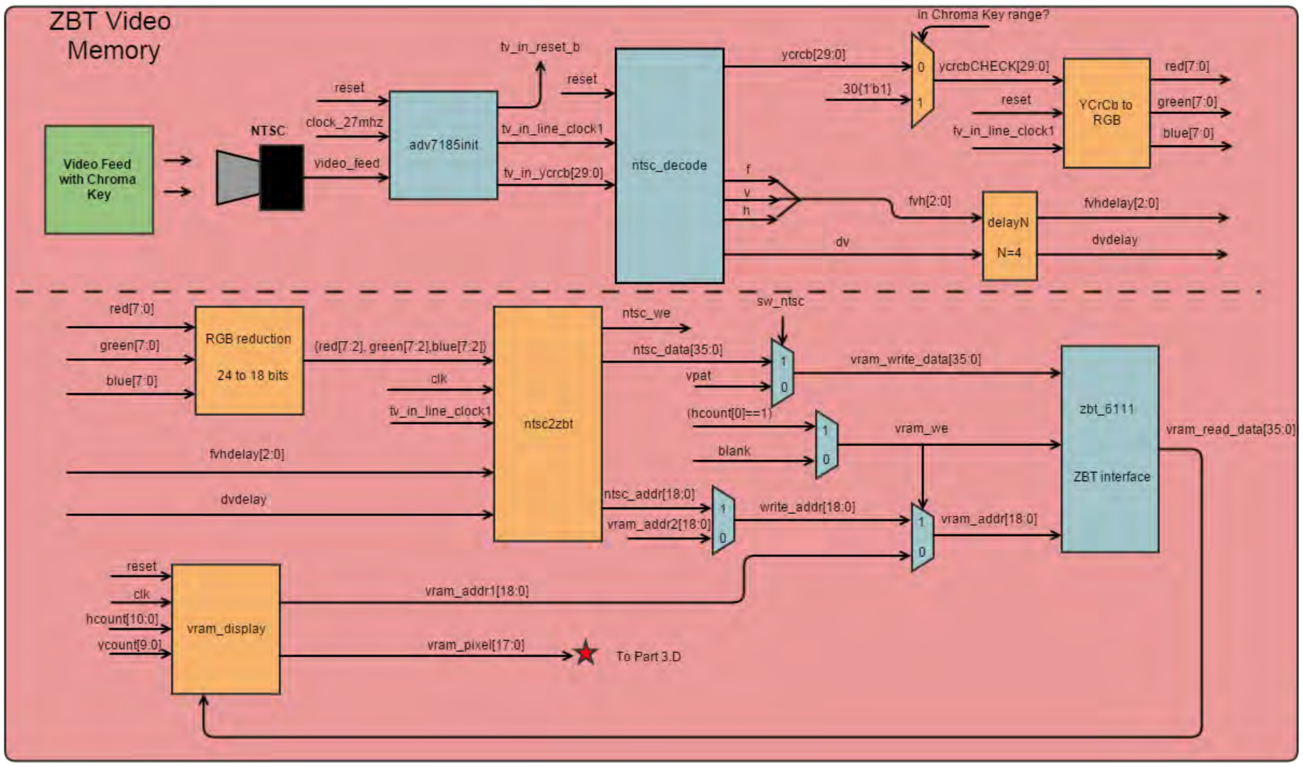


Figure 2: ZBT Video Memory Block Diagram

The ZBT Video Memory block diagram depicts the flow of data from the NTSC camera to the ZBT memory and to the final mux stage of the Chroma Key Compositing system. Note that the blue-grey blocks indicate that nothing has occurred in these modules. Orange blocks indicate changes or new appendages to the new 6.111_sample.v code. Additionally, clk refers to the 65MHz clock. The input for the mulitplexers without an input reference is sw_ntsc. The dotted line indicates the final outputs from the top half of the diagram are the inputs to the second half of the diagram.

The ZBT Memory module has 3 main goals:

(i) Filter video pixels that fall into a YCrCb range and replace the pixel with a 'highlight', then output the signal YCrCbCHECK[29:0].

(ii) Convert the YCrCb data coming from NTSC into RGB color data, using the signal {red[7:2],green[7:2],blue[7:2]} as input.

(iii) Store the video pixel in ZBT where it can be read out for the final muxing in part 3-D as vram_pixel[17:0].

Fortunately, the 6.111 staff has given us considerable amount of starting codes. The code zbt_6111_sample.v has been used as a skeleton on which we further appended our features. The NTSC signal decoder, zbt_6111.v, vram_display.v, as well as the adv7185init chip module were also provided to obtain video data that could be practically used for VGA display.

Initially, the video feed was displayed in black and white because only the leading 8 bits of luminesce (Y of YCrCb) was used for the display of the image. This meant the default version of zbt_6111_sample.v must be altered to correctly display the video in color. Luckily, the 6.111 staff gave a suggestion on modifying the code to convert YCrCb into RGB. A YCrCb to RGB module, provided by the staff, was used to allow this conversion between color spaces. The YCrCb to RGB module returns 8 bits of R, G, and B each – the standard 24bit pixel display and color field for a VGA monitor. However, a total of 18 bits will be stored into memory such that we can conveniently store two 18 bits of video pixel per 36-bit address location. This means that we will only take the most significant 6 bits of R, G, and B since we can hardly discern this visual loss. We then also have to make necessary changes to fvh[2:0] and dv. Since the conversion of YCrCb to RGB takes 3 cycles, we must delay fvh[2:0] and dv by 3 cycles. These signals interface with the ntsc2zbt.v module, which coordinates inputting video data into the memory. This pipelining is done by the delayN module, which delays the input signal by 3 cycles.

Since we are now storing two 18-bit pixel data per memory location, we may access addresses in the new configuration twice as fast as when we were storing four 8-bits of luminance data per address. This also means we need twice the number of address spaces, which is resolved by modifying the ntsc2zbt.v code. Additionally, we also need to change vram_display.v which reads out the values in the ZBT that holds the video information, since we will now be outputting 18 bits of pixel data rather than 8 bits, and indexing twice as fast as described in the modifications for ntsc2zbt.v code. With these modifications, the code can finally display a color display on VGA as shown in Figure 3.

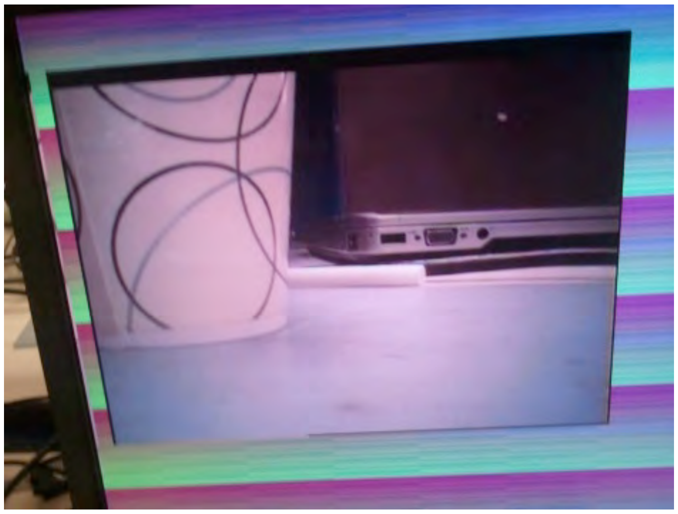


Figure 3: VGA Color Display

A proper color display from the NTSC camera feed using the 6 most significant bits of R,G, and B.

The next implementation that we executed was pre-filtering a video pixel in the Chroma Key range with a highlight pixel. We initially understood that filtering could occur in part 3-D, in the final muxing of the image pixel in replace of the video pixel that was the color of the Chroma Key. However, we approached this filtering before memory storage as a proof of understanding that filtering can occur both before and after video data is stored in memory.

With José's recommendation, the image filtering was done in the YCrCb domain. We understood that an HSV color space might have been a better color field for the color filtering, because in this space, color is separated into hue (H), saturation (S), and value (V). A specific color was only mapped primarily by hue and brightness was determined by value. Therefore, a tighter and more accurate range could check for hue and a broader range for value when we try to determine if a video pixel is green. However, we decided to go for simplicity and timing robustness, following José suggestion. Using past code from previous terms for color range values as well as Gim's colorcursor.bit file that allowed us to determine ranges for Y, Cr, and Cb, we were able to create a range of values that worked reasonably well in detecting a large majority of the Chroma Key.

A mux was used to check if the video pixel coming from the NTSC camera was in the range of YCrCb. If so, a highlight pixel with value equivalent to 30 bits of ‘1’ was sent into the YCrCB to RGB module instead of the video pixel. This multiplexer created another cycle of delay in addition to the color space conversion, so a total delay of 4 cycles was required for fvh[2:0] and dv. Figure 4 shows the initial pre-filtering without image compositing.

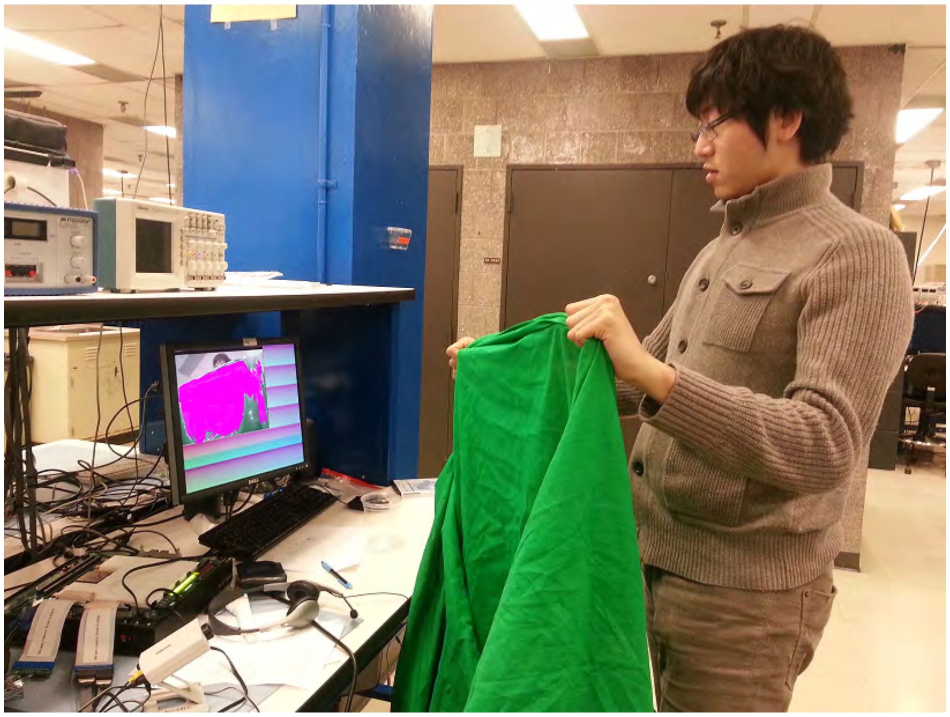


Figure 4: Pre-filtering Chroma Key before storing into memory.

When a video pixel within the range of specified YCrCb range is passed into the ZBT memory module, 30 bits of ‘1’ is sent into YCrCb to RGB module and then into memory. The resulting RGB is pink, which will be described more thoroughly in section 3-D.

3-B-i Compact Flash (not implemented) – Thipok

The compact flash module is part of the original plan to store the background picture. Since the FPGA does not have enough built-in memory – in BRAM format – to store an image of the dimension 1280 x 960, the background picture must be initially loaded into the labkit's compact flash. At the first reset signal, the picture in compact flash will then be loaded into a ZBT memory. During the operation, FPGA will read this background picture from the ZBT only.

Data, in bytes, can be loaded into a compact flash from a computer's serial port using a hardware module UM 245-R Parallel FIFO. The computer sends one byte of data via its serial port to the hardware module. When the data is ready to be read by the FPGA, the signal RXF# will go low. The FPGA then pulls the output signal RD# down and the data will be available at D[7:0].

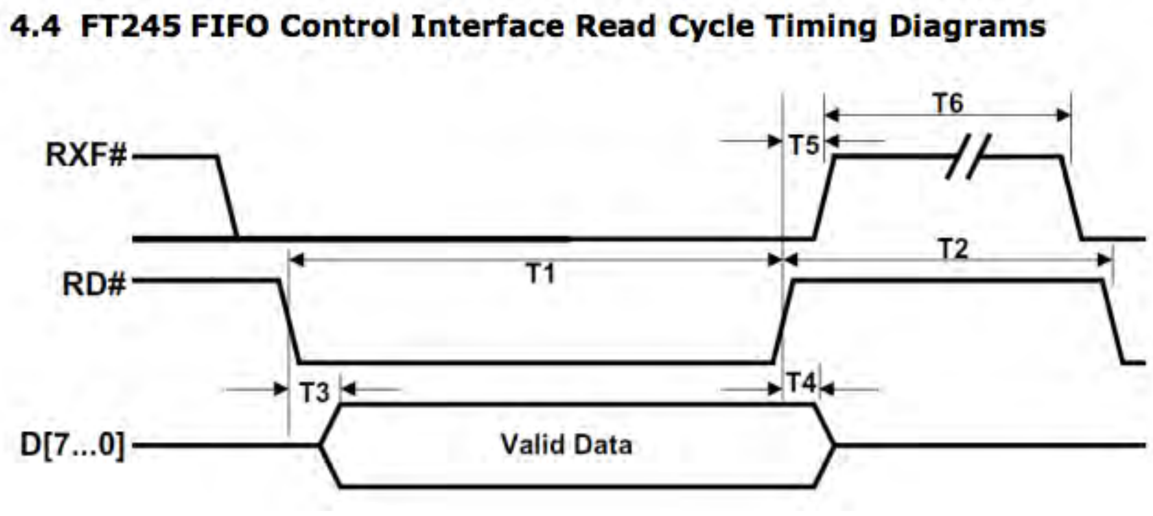


Figure5: FIFO Control Interface Diasgram
from FTDI Chip's UM245R USB – Parallel FIFO Development Module Datasheet

We used the module usb_input as an interface between the FGPA and the hardware FIFO and use the module flash_manager as an interface between the FGPA and the compact flash memory. Simply speaking, usb_input module reads bytes of data from the FIFO, then passes these data to flash_manager. The flash_manager increments memory address and stores the input data in incoming order.

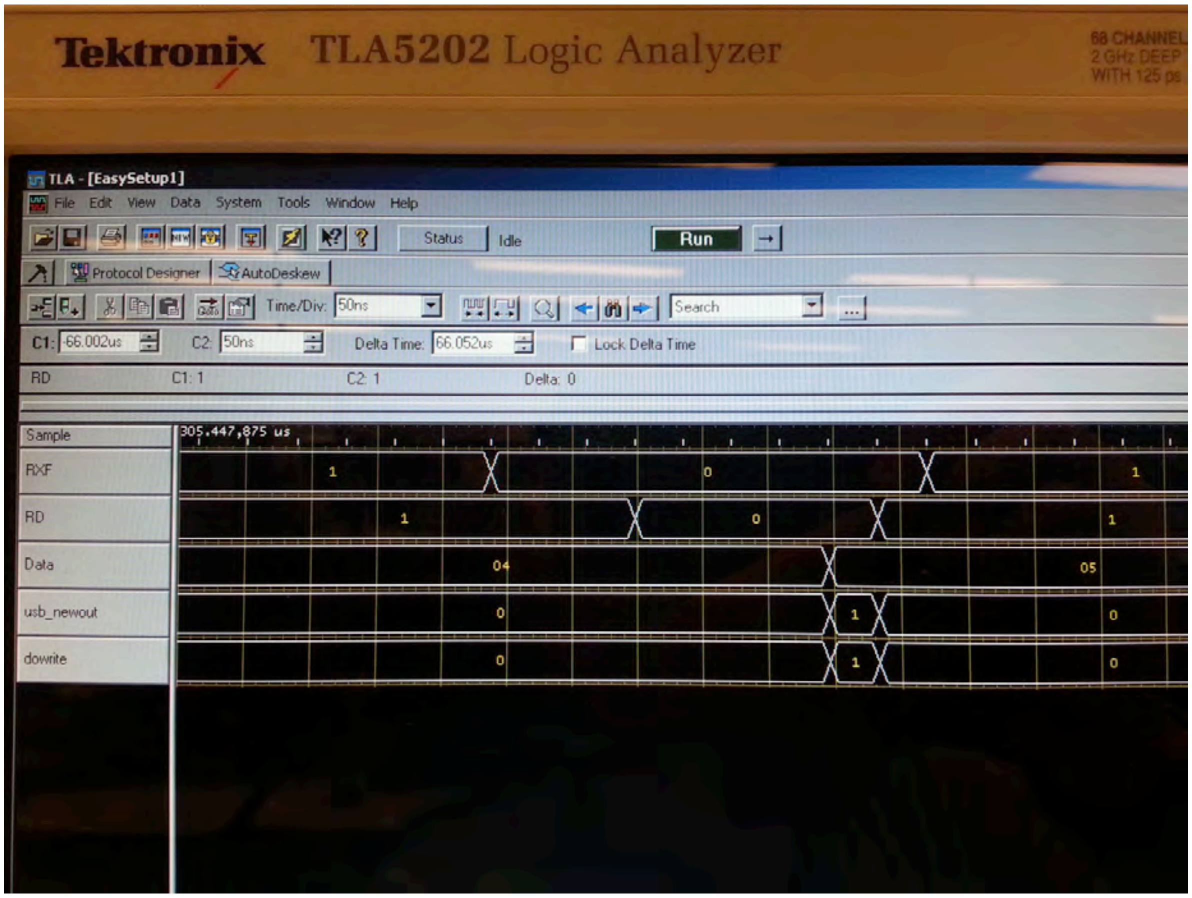


Figure 6: Logic analyzer displaying output signals of "usb_input" module

The testing of this data went well up until the flash_manager part. Using a logic analyzer to check the output data from usb_input, we verified that all byte data sent from the computer was received by the FPGA, in correct order. However, we encountered a problem where the compact flash memory stores only about half of the data it received. The error was non-deterministic, as different transmission of the same set of byte data resulted in different sets of byte data stored in the compact flash. Every time, some data would be missing.

After one week of unsuccessful debugging, we decided to resort to the FPGA's built-in BRAMs, at the cost of less memory capacity.

3-B-ii BRAM Image Memory – Thipok

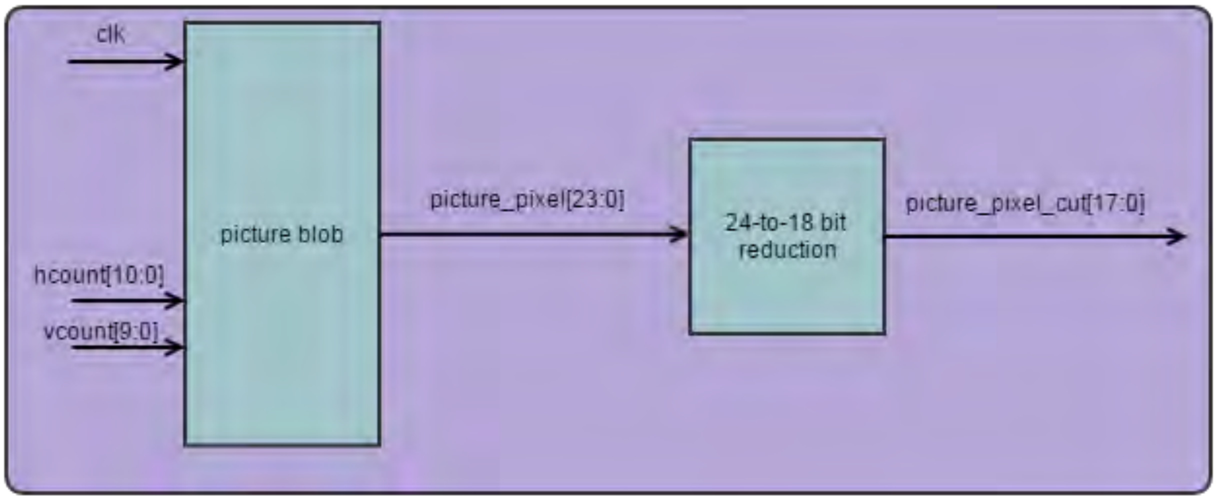


Figure 7: Background Picture Memory Block Diagram
The BRAMs are within "picture blob" module

The BRAM module for background picture storage was implemented as a back-up plan to replace the compact flash and ZBT combination. According to the 6.111's website, the FPGA contains 144 of small 18kbit memory blocks. The blocks are synchronous and dual-port memory. For simplicity, we decided to use the block memories (BRAMs) as a single-port read-only memory, because we will not alter the background picture at any point throughout the program's operation.

As suggested by "Display JPG images" section on the 6.111 website, it is most efficient to store each image in 256-color format. Each pixel in the images will have a corresponding color value between 0 and 255, which can be interpreted into Red, Green, and Blue values for RGB display by looking up the Red, the Green, and the Blue color maps.

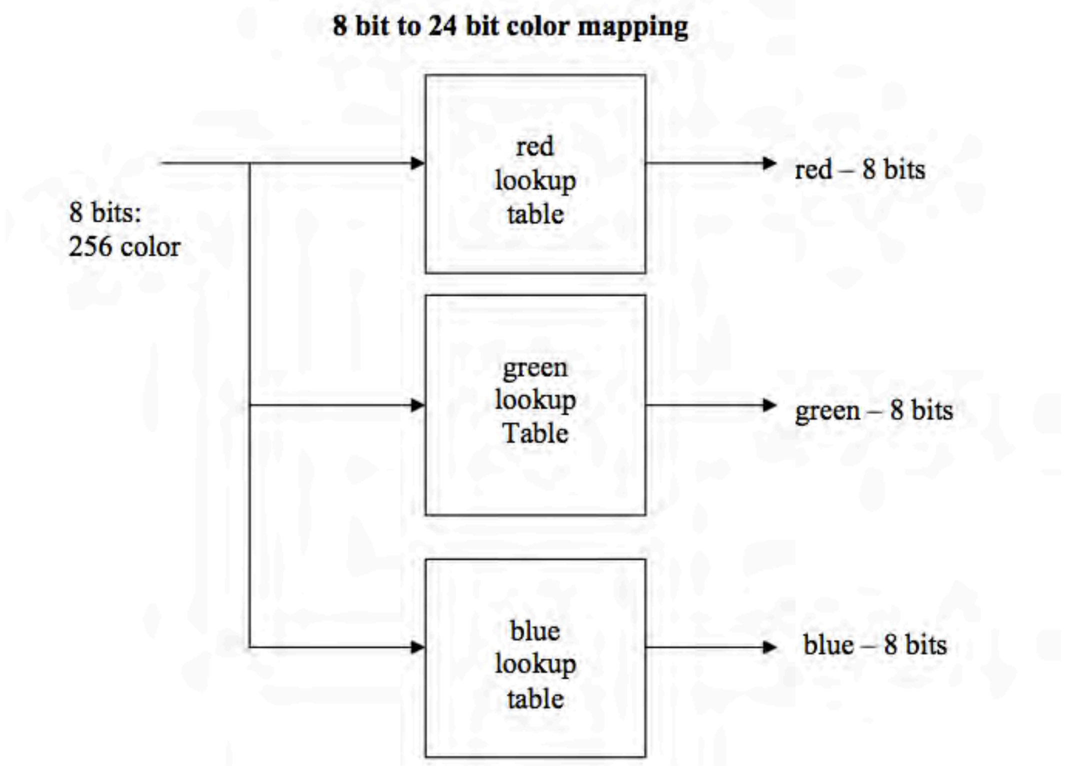


Figure 8: 8-bit to 24-bit color mapping
Using an 8bit value of pixel to recover 24 bits of RGB.

Note that the standard VGA monitor can display 2^{24} colors, corresponded to 8 bits of Red, Green, and Blue. However, compressing pictures into 2^8-color format still gives acceptable color resolution and helps reduce the memory storage per pixel by three times.

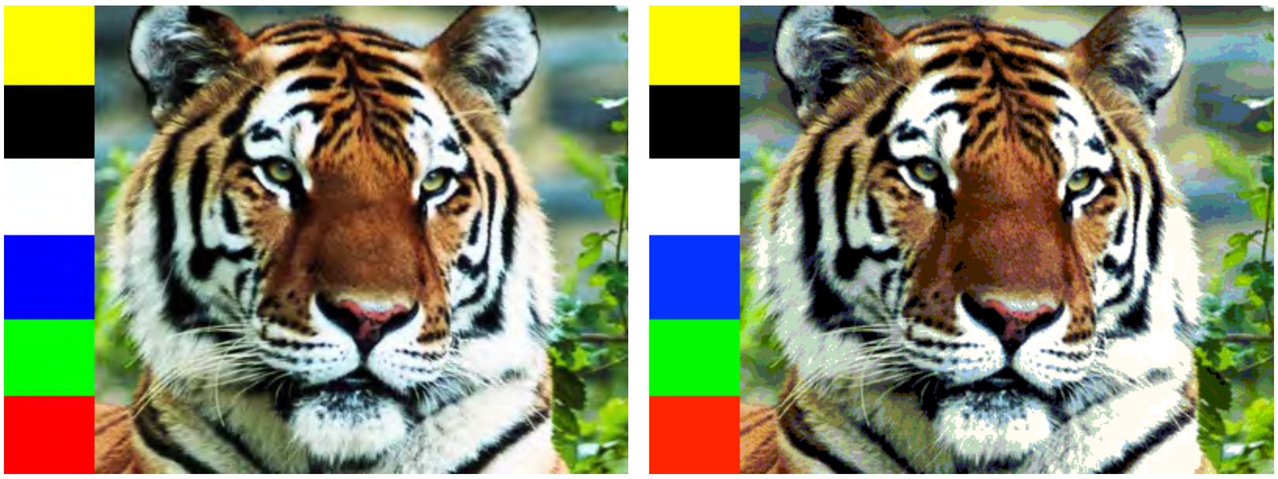


Figure 9: Sample bitmap pictures in 24-bit (left) and 8-bit format (right)

The total memory capacity of BRAMs is 144 x 18kbits = **2.592Mbits**.

Therefore, we may not store a background picture of dimension 1280 x 960 pixels, which would contains 1280 x 960 x 8bit = **9.83Mbits**. Scaling down to the NTSC camera's output dimension of 640 x 480 pixels, we are just below the limit, at 640 x 480 x 8bit = **2.458Mbits**

Our first design decision is to store one byte (one pixel) per BRAM address, so that we may keep the design as simple as possible. However, after generating the BRAM with CoreGen and trying to implement the Verilog code, we discovered that this configuration requires 152 BRAMs. A possible explanation is that 144 BRAMs are distributed throughout the FPGA, so the one-byte-per-address design exceeds the block memory's physical limit.

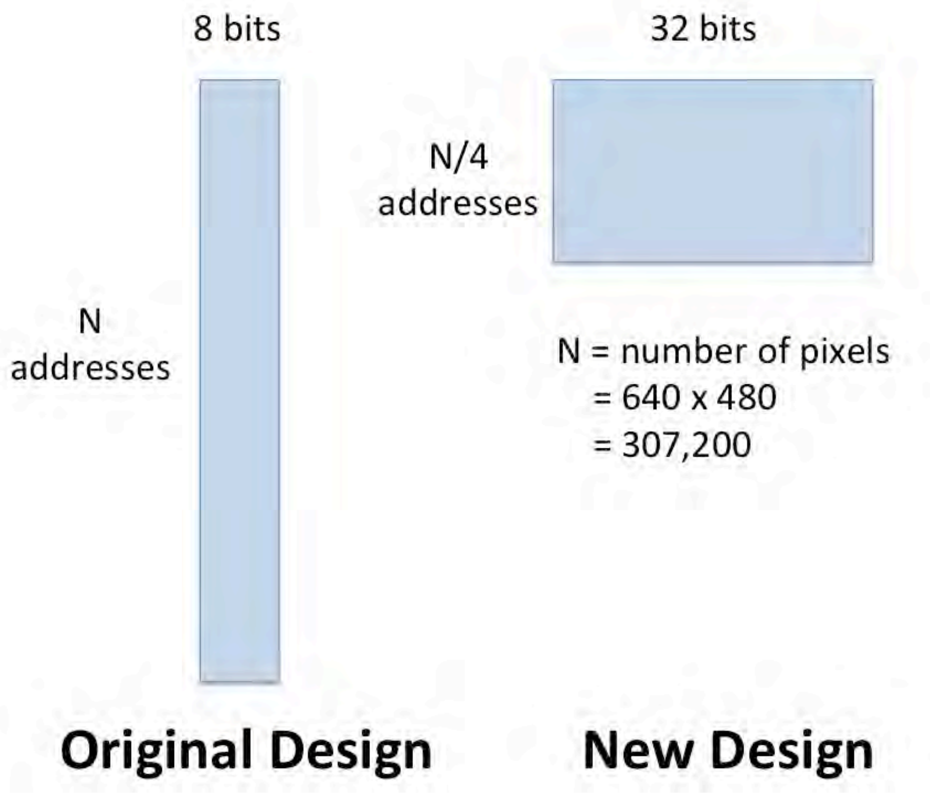


Figure 10: Memory management of the image
Since we were unable to use an 8 pixel by N layout, we resorted to a 32 pixel by N formatting to successfully store the image into BRAM

We then try store four bytes (four pixels) of data per BRAM address and found that the configuration requires only 140 BRAMs. Note that, with this design decision, we must carefully index the storage address to output the correct pixel's color value.

For example, in the original design, if we convert a 2-dimensional picture with height = HEIGHT and width = WIDTH into a 1-dimensional array, a pixel at (x, y) will be at the location:

$$location = x + y(WIDTH)$$

and the address of this pixel is

$$address = location$$

However, in the new design, assuming that location is n-bit long, the new address will be

$$address = location[n{:}2]$$

and the pixel is stored at the of this address's index

$$index = location[2{:}0]$$

3-C Zooming – Thipok and Daniel

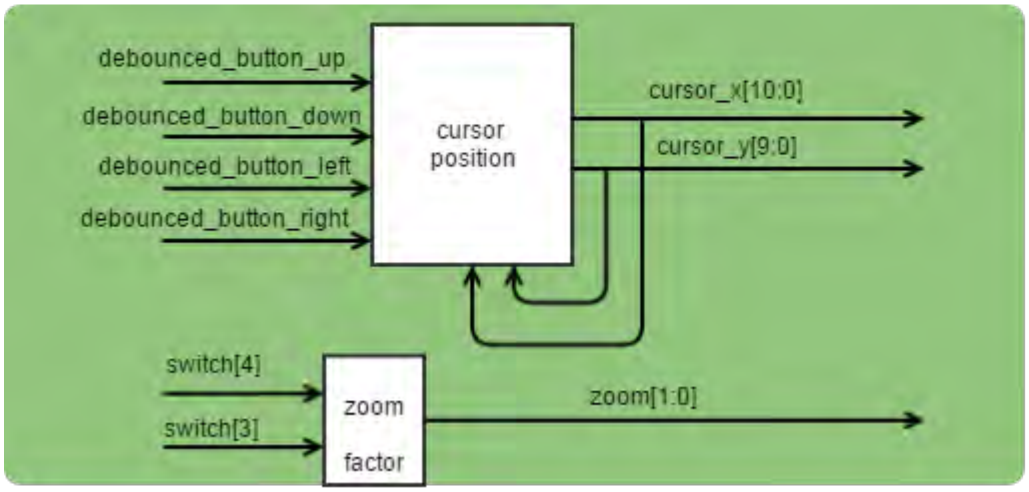
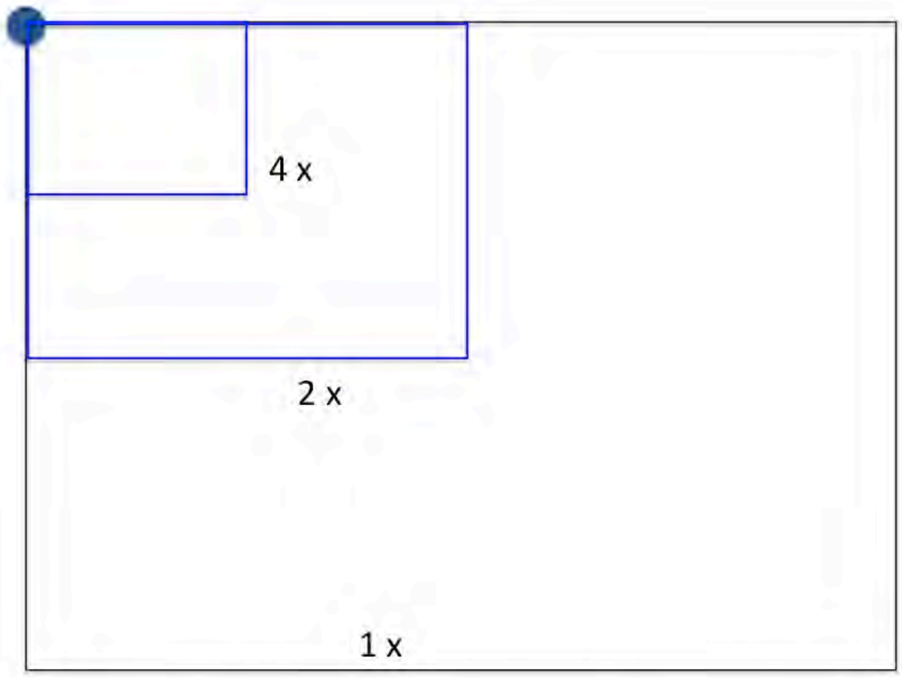


Figure 11: Cursor and Zoom Factor Block Diagram

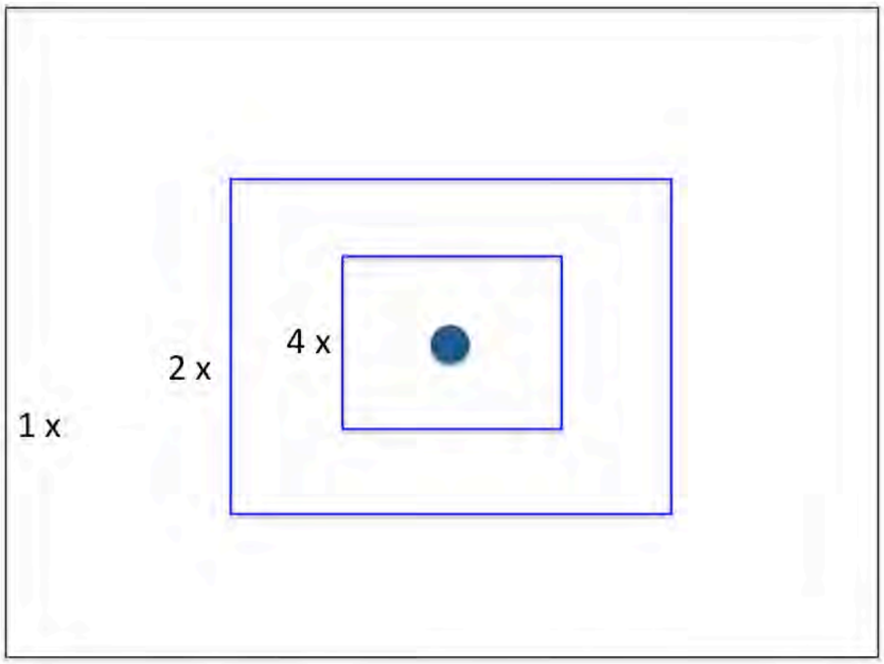
The zoom module takes in the pixel location on the screen, the cursor location, and the zoom factor. For different zoom factors, the module calculates the pixel from the background picture that corresponds to the given pixel on the monitor, and output the RBG color values of this background picture pixel.

To avoid calculation-intensive division, we select the zooming factor to be 1x, 2x, 4x, and 8x, such that we may perform any necessary division with bit-shifting.

There are many possible zooming types. For example, without a cursor, we may choose to zoom in and out with respect to the top left corner of the background picture. Similarly, we might as well choose to zoom with respect to the center of the picture. These two designs are examples of the fixed-point zoom. In our design, the zoom's reference point is movable, such that the users can choose to zoom in on different locations of the background picture. We mark the zoom's reference point with a blue crosshair (the cursor).



Fixed-point Zoom, from the top left corner



Fixed-point Zoom, from the center

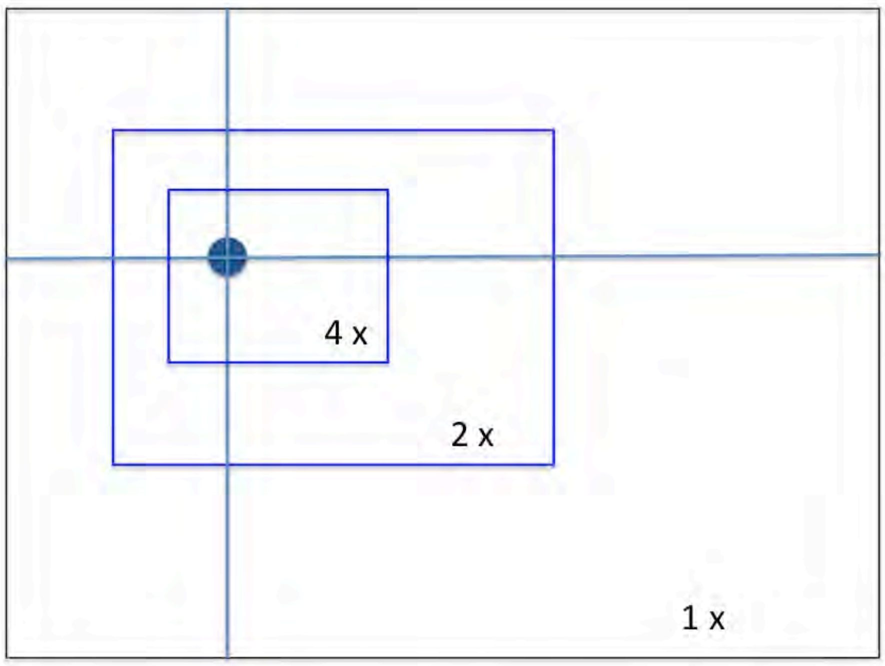


Figure 12: Different versions of zoom

We initially began our approach with corner zoom, with simply address holding. We then expanded the approach to centering where we zoomed in on the center of the image by manipulating the bounds of our addressing. Our final approach was using a cursor as a reference point as it anchored the point of view. Note that the bigger zoom factors are a smaller area that expands to the total view of the VGA.

Note that both fixed-point zooms are actually special cases of the cursor zoom.

For the cursor zoom, we can calculate the (x, y) coordinate for the picture pixel that corresponds to a given (hcount, vcount) coordinate of a point on the VGA monitor

Let the zooming factor be "n", and the cursor is at (cursorx, cursory)

$$x = cursorx + \frac{hcount - cursorx}{n} = cursorx - \frac{cursorx}{n} + \frac{hcount}{n}$$
$$y = cursory + \frac{vcount - cursory}{n} = cursory - \frac{cursory}{n} + \frac{vcount}{n}$$

For n = 1, 2, 4, and 8,

$$\frac{s}{n} = s \gg (\log_2 n)$$

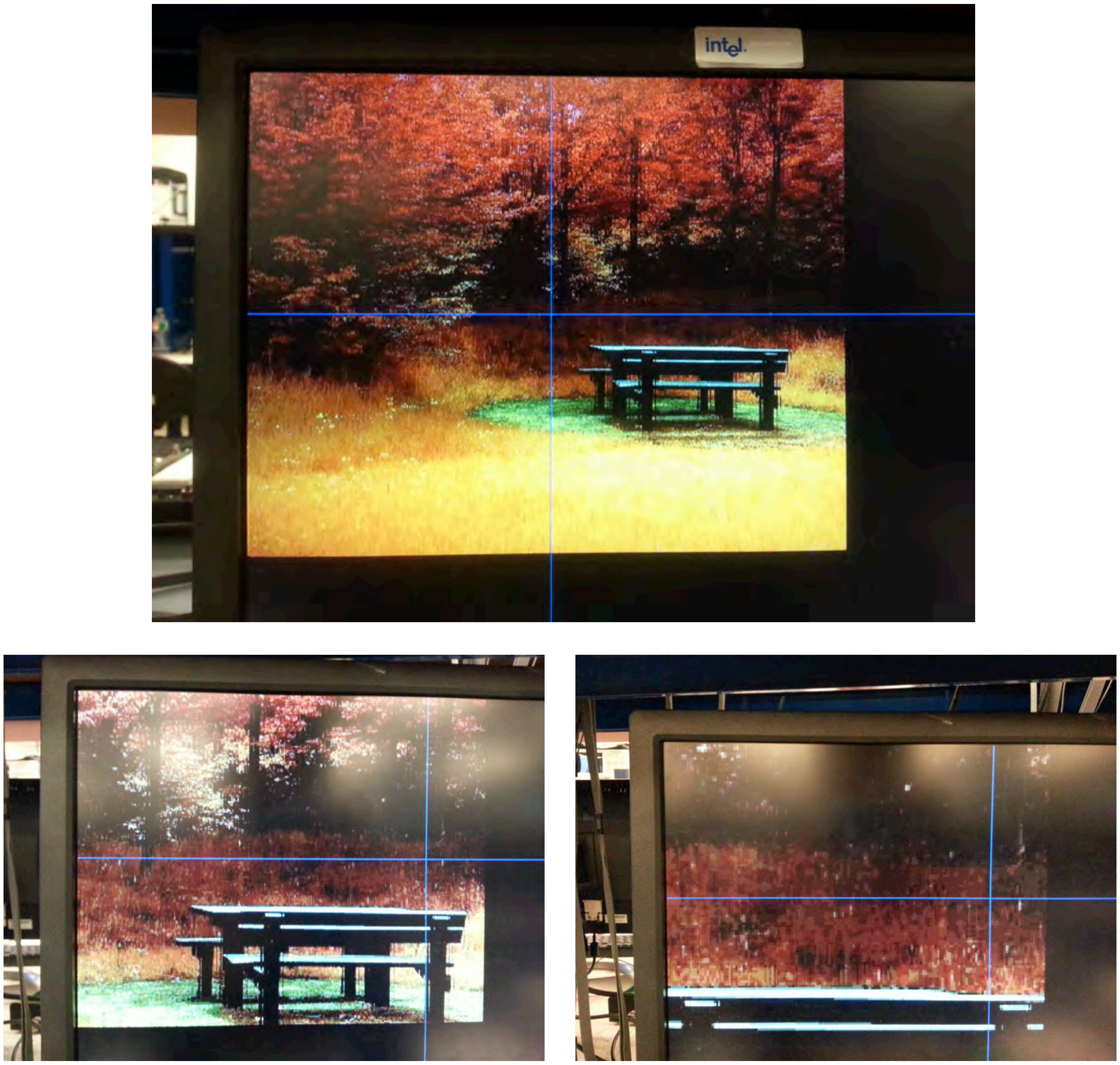


Figure 13: Implementation of cursor zoom on image

We apply a cursor zoom on the image we will be compositing later on. The last image shows the cursor being the reference point for the rectangles we will be zooming into.

3-D Chroma Key Compositing – Thipok and Daniel

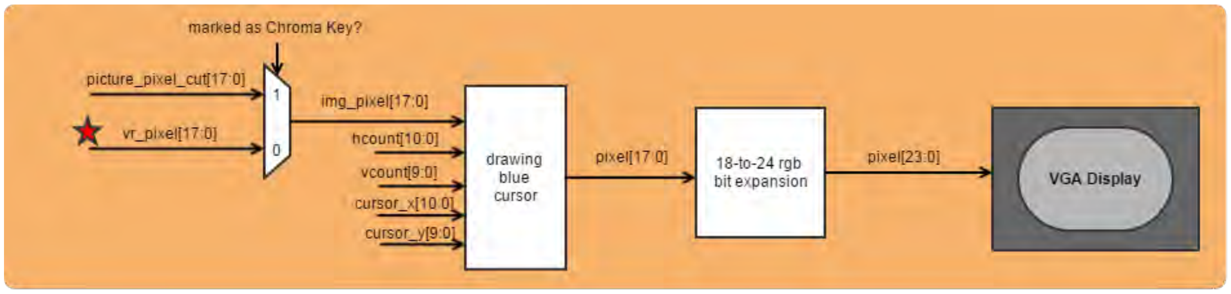


Figure 14: System Block Diagram of Chroma Key Compositing

Final step in the Chroma Key Compositing. We mux video data and the background image by checking whether the incoming pixel is pink and send it into the VGA. The blue cursor lines are identified by checking (hcount, vcount) with the current cursor location.

We may view the previous modules involving the video feed storage and the background picture as two main pipelines of the project.

- The first pipeline is the NTSC video feed stored and accessed from the ZBT.
- The second pipeline is the background stored and accessed from the BRAM.

Now that we have correctly stored the video and the picture, the final step is to composite them using Chroma Key range as the criteria. Notice that the two data pipelines are completely independent, except for sharing the 65MHz clock. Prior to this step, we have marked any pixel of the video output that falls within the Chroma Key range by changing its value to a specific “flag” value.

In the final compositing, we check the Red, Green, and Blue values of a pixel from the video frame. If the pixel value matches the flag value, we know that this pixel used to be a Chroma Key pixel and replace it with a corresponding pixel from the background picture. Otherwise, we output that video pixel as it is.

A very important concept for the final muxing is to use proper registers to synchronize signals from different pipelines. Recall that the video-ZBT pipeline and the picture-BRAM pipeline are independent, their output signals might not change at the same time as we input the new sets of (hcount, vcount). Adding registers solve this problem by assigning new values to variables only at either positive or negative edges of the clock. Again, recall that the two pipelines still share a common clock. Failure to register certain signals and synchronize the two lines may result in image distortion on the VGA monitor.

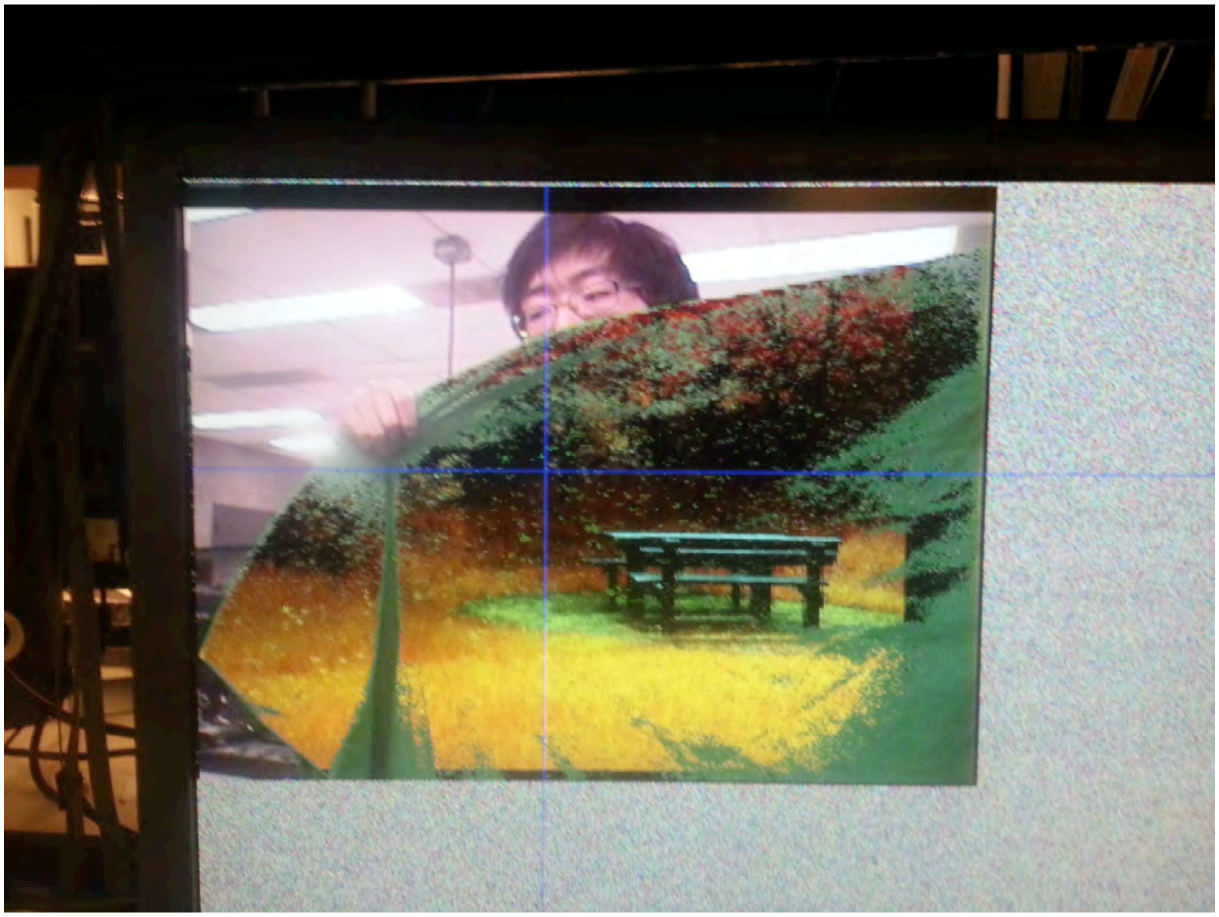


Figure 15: Demonstration of the Project

Picture of our final project presentation, with a proper compositing of the background image.

5. Reviews and Recommendation

We hope that this final project report is useful for anyone who would like to tinker with an FPGA. A newer model of FPGA would prove to be more useful as it would have lower memory constraints for displaying larger images and more advance image processing mentioned in our presentation and proposal. Even at the final stage of the project, we noticed that noises from NTSC camera prevent perfect chroma key compositing. Morphological processing has the potentials to reduce this noises and increase compositing accuracy. The algorithm requires buffer storage for every frame of the NTSC video. – much more than those available in the FPGA's BRAM or the ZBT.

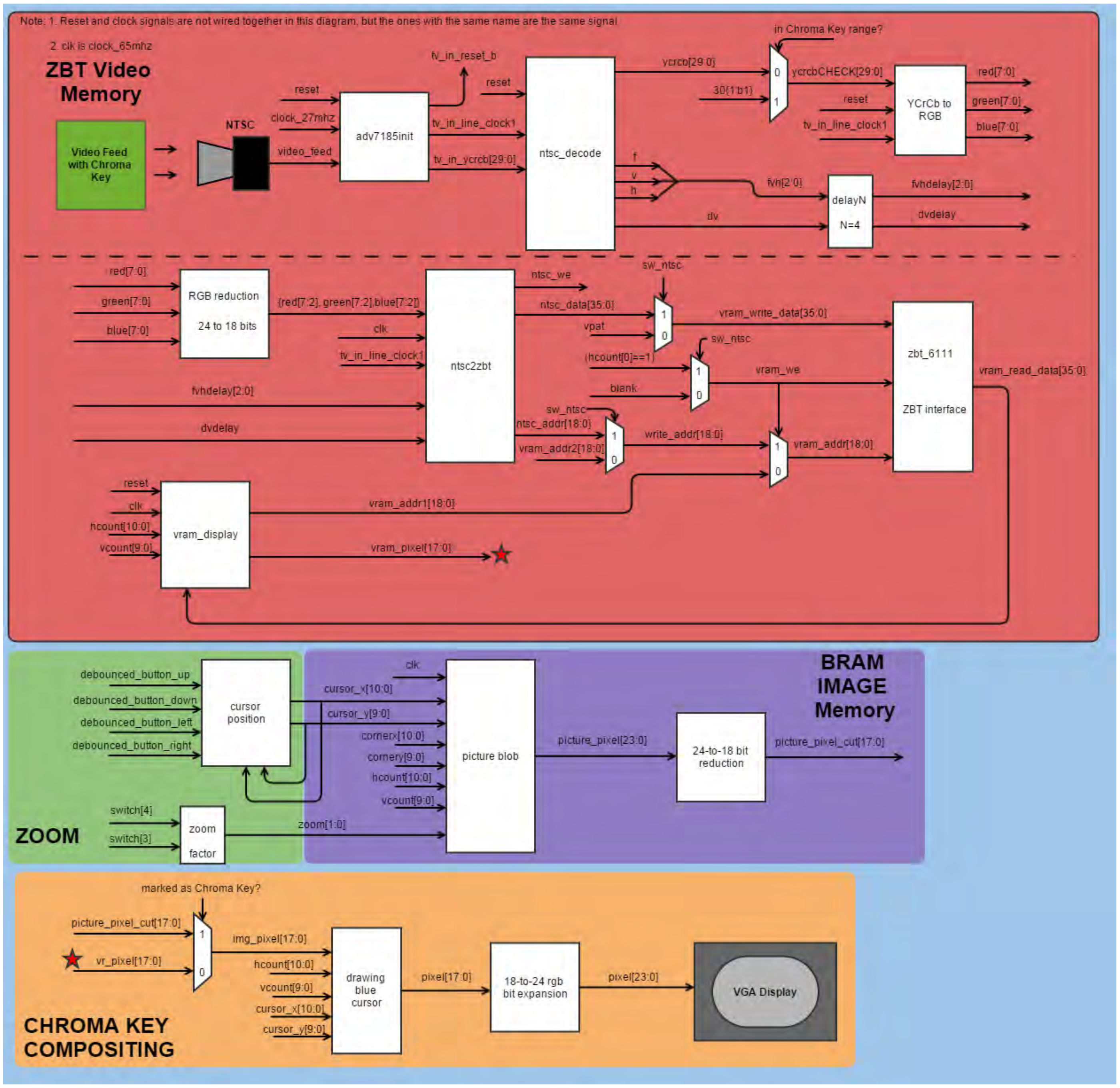
From this project, we have learned a very important lesson about registers. Pipeline and synchronizing modules proved to be a headache. Signals from unrelated modules will be updated at different moments. We must register them to synchronize the update and prevent glitches. We have had a problem with image distortion that resolves with only two lines of registers added (and two days of debugging).

6. Conclusion

In conclusion, we completed our project's goals and were able to add on some extensions to improve the zoom functionality and the VGA display presentation. However, we were unable to complete our stretch goal due to memory constraints. We demonstrated that a successful composited image can be produced with only hardware and all calculations can be done in real time. The keys to our success in this project are good time management and having an early start.

APPENDIX:

A: Comprehensive Chroma Key Compositing System's Block Diagram



B: Verilog's Source Code

Top Module: zbt_6111_sample

Auxiliary Modules:

xvga	vram_display
adv7185init	delayN
ntsc_zbt	ramclock
ntsc_decode	picture_blob
YCrCb2RGB	debounce
zbt_6111	display_16hex

```
////////////////////////////////////////////////////////////////////////////////
// THIS IS THE PROJECT'S TOP LEVEL MODULE
////////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

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

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,

             clock_27mhz, clock1, clock2,
             disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
             disp_reset_b, disp_data_in,

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

             switch,

             led,

             user1, user2, user3, user4,

             daughtercard,

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

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

```
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;

// ac97_sdata_in is an input

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

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

   // SRAMs
/* change lines below to enable ZBT RAM bank0 */
/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;  // clock enable
*/
/* enable RAM pins */
   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

//   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b1;
//   assign ram1_clk = 1'b0;

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

   // clock_feedback_out will be assigned by ramclock
   // assign clock_feedback_out = 1'b0;  //2011-Nov-10
   // clock_feedback_in is an input

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

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

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

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

//   wire clk = clock_65mhz;  // gph 2011-Nov-10

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

   ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
                              .ram0_clock(ram0_clk),
                              .ram1_clock(ram1_clk),   //uncomment if ram1 is
used
                              .clock_feedback_in(clock_feedback_in),
                              .clock_feedback_out(clock_feedback_out),
.locked(locked));
//
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// reset generation and button debouncing
//
   // 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, debounced_button;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
      debounce db2(power_on_reset, clk, ~button3, debounced_button);

   // UP DOWN LEFT RIGHT buttons for zoom cursor control
   wire debounced_button_up, debounced_button_down;
   wire debounced_button_left, debounced_button_right;
      debounce db3(power_on_reset, clk, ~button_up, debounced_button_up);
      debounce db4(power_on_reset, clk, ~button_down, debounced_button_down);
      debounce db5(power_on_reset, clk, ~button_left, debounced_button_left);
      debounce db6(power_on_reset, clk, ~button_right, debounced_button_right);
   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);

   // 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);

////////////////////////////////////////////////////////////////////////////////
// Initiate ZBT for video data
```

```
//
   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;
   wire ram0_clk_not_used;
   //to get good timing, don't connect ram_clk to zbt_6111
   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
         vram_write_data, vram_read_data,
         ram0_clk_not_used,
         ram0_we_b, ram0_address, ram0_data, ram0_cen_b);
//
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// NTSC video decoding, Chroma Key marking, and YCrCb to RGB conversion
// and write video data to ZBT
//
   // 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));

   // ntsc_decode module
   wire [29:0] ycrcb;   // video data (luminance, red chrom, blue chrom)
   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));

   // mark any ycrcb pixels that fall within Chroma Key range
   // such that they can be replaced by picture pixels later
   reg [29:0] ycrcbCHECK;
   always @(posedge tv_in_line_clock1)
      ycrcbCHECK <= ((ycrcb[29:20]<10'h2e9) && (ycrcb[29:20]>10'h154)
                  && (ycrcb[19:10]<10'h1fc) && (ycrcb[19:10]>10'h174)
                  && (ycrcb[9:0]<10'h204) && (ycrcb[9:0]>10'h178))
                  ? {30{1'b1}} : ycrcb;

   // delay fvh and dv to correspond with muxing delay above
   wire fvhdelay2, fvhdelay1, fvhdelay0, dvdelay;
   delayN #(.NDELAY(4))
fvhbit2(.clk(tv_in_line_clock1),.in(fvh[2]),.out(fvhdelay2));
   delayN #(.NDELAY(4))
fvhbit1(.clk(tv_in_line_clock1),.in(fvh[1]),.out(fvhdelay1));
   delayN #(.NDELAY(4))
fvhbit0(.clk(tv_in_line_clock1),.in(fvh[0]),.out(fvhdelay0));
   delayN #(.NDELAY(4)) dvbit(.clk(tv_in_line_clock1),.in(dv),.out(dvdelay));

   // convert the marked YCrCb to RGB
   wire [7:0] red, green, blue;
   YCrCb2RGB rgb1( .R(red), .G(green), .B(blue),
      .clk(tv_in_line_clock1), .rst(reset),
      .Y(ycrcbCHECK[29:20]), .Cr(ycrcbCHECK[19:10]), .Cb(ycrcbCHECK[9:0]));

   // 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,{fvhdelay2,fvhdelay1,fvhdelay0},
dvdelay,
      {red[7:2],green[7:2],blue[7:2]}, ntsc_addr, ntsc_data, ntsc_we,
switch[6]);
//
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// read ZBT to retrieve video pixel corresponded to a given (hcount, vcount)
//
   wire [17:0]    vr_pixel; //now holds 18 bits
   wire [18:0]    vram_addr1;
   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
          vram_addr1,vram_read_data);
//
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Alternating read and write to the video ZBT
//
   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 = 36'd0;       // to make pixels outside ntsc frame
black

   // 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'd1) : blank; //accessing every
other count
   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;
//
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Cursor movement
//
   reg [10:0] cursor_x = 11'd320;
   reg [9:0] cursor_y = 10'd240;
   reg [19:0] move = 20'd0;
   always @(posedge clk) begin
```

```
      if (move==20'hFFFFF) begin
         if ((debounced_button_left) && (cursor_x>=11'd1))
            cursor_x <= cursor_x - 1;
         if ((debounced_button_right) && (cursor_x<=11'd639))
            cursor_x <= cursor_x + 1;
         if ((debounced_button_up) && (cursor_y>=10'd1))
            cursor_y <= cursor_y - 1;
         if ((debounced_button_down) && (cursor_y<=10'd479))
            cursor_y <= cursor_y + 1;
         move <= move + 1;
      end
      else move <= move + 1;
   end
//
//
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// Adding the picture blob
//
      wire [23:0] picture_pixel;
      reg [17:0] picture_pixel_cut;
      wire [1:0] zoom;
      wire [10:0] cornerx;
      wire [9:0] cornery;

      assign zoom = switch[4:3];
      assign cornerx = 11'd0;
      assign cornery = 10'd20;

      picture_blob #(.WIDTH(640),.HEIGHT(480))

      sample(.pixel_clk(clk),.zoom(zoom),.cursx(cursor_x),.cursy(cursor_y),
                  .x(cornerx),.hcount(hcount),.y(cornery),.vcount(vcount),
         .pixel(picture_pixel));

   // use a register for picture_pixel_cut
   // to handle delay dislay discrepancy between vram_dispay and picture_blob
   always @(posedge clk)
            picture_pixel_cut <=
         {picture_pixel[23:18],picture_pixel[15:10],picture_pixel[7:2]};
//
//
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// select output pixel data
// We use 18-bit pixels and extrapolate them to 24-bit rgb pixels
   reg [17:0]  pixel;
   reg b, hs, vs;
   reg [17:0] img_pixel;

   always @(posedge clk) begin
      // look for the marked Chroma pixel
      // if Chroma pixel, replace with picture pixel
            if (vr_pixel == 18'b111111011111111111)
                  img_pixel <= picture_pixel_cut;
      // if not, output the video pixel
            else
                  img_pixel <= vr_pixel;

      // mark cursor lines with color blue
      pixel <= (((hcount-cornerx) == (cursor_x+11'd4)) || ((vcount-cornery) ==
cursor_y))
               ? 18'b000000_000000_111111 : (switch[0] ? img_pixel :
picture_pixel_cut);
      b <= blank;
      hs <= hsync;
      vs <= vsync;
      end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clk.
   assign vga_out_red = {pixel[17:12],2'b00};
   assign vga_out_green = {pixel[11:6],2'b00};
   assign vga_out_blue = {pixel[5:0],2'b00};
   assign vga_out_sync_b = 1'b1;    // not used
      assign vga_out_pixel_clock = ~clk;
      assign vga_out_blank_b = ~b;
      assign vga_out_hsync = hs;
      assign vga_out_vsync = vs;

   // debugging
   assign led = ~{vram_addr[18:13],reset,switch[0]};
//
//
/////////////////////////////////////////////////////////////////////////////
endmodule   // zbt_6111_sample

/////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output   vsync;
   output   hsync;
   output   blank;

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

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

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

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

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

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

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

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

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

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

   input reset, clk;
   input [10:0] hcount;
   input [9:0]    vcount;
   output [17:0] vr_pixel; //now it is 18 pixels of data
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

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

   wire [18:0]     vram_addr = {vcount_f, (~hcount_f[9:1] -9'd150)};
//accessing every other time
   wire hc2 = hcount[0]; //now we are accessing every other time
   reg [17:0]      vr_pixel;
   reg [35:0]      vr_data_latched;
   reg [35:0]      last_vr_data;

   always @(posedge clk)
      last_vr_data <= (hc2==1'd1) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
      vr_data_latched <= (hc2== 1'd0) ? vram_read_data : vr_data_latched;

   always @(*)    begin // each 36-bit word from RAM is decoded two 18 bit
values
     case (hc2)
       1'd1: vr_pixel = last_vr_data[17:0];
       1'd0: vr_pixel = last_vr_data[35:18];
     endcase
   end

endmodule // vram_display
//
//
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
// delay module for wire
//
module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;

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

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

endmodule // delayN
//
//
///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////
// ramclock module
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

      assign ref_clk = ref_clock;

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

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

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

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

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

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


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

   assign locked = lock1 && lock2;

endmodule // ramclock


////////////////////////////////////////////////////
//
// picture_blob: display a picture
//
```

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

module picture_blob
      #(parameter       WIDTH = 640,            // default picture width
                        HEIGHT = 480)           // default picture height
      (input pixel_clk,
      input [1:0] zoom,
      input [10:0] x, hcount, cursx,
      input [9:0] y, vcount, cursy,
      output reg [23:0] pixel);

      reg [18:0] image_addr1, image_addr2;      // for 680x480 image
      wire [31:0] four_pixel_bits;              // stored in one BRAM location
      reg [7:0] image_bits;                     // 8-bit color value of a pixel
      wire [7:0] red_mapped, green_mapped,blue_mapped;      // R G B values

      reg [10:0] newx;        // adjusted with zoom factor
      reg [9:0] newy;               // adjusted with zoom factor

      // note the one clock cycle delay in pixel!
      always @ (posedge pixel_clk) begin
            // calculate rom address and read the location
            newx <= (zoom == 2'b00) ? (hcount-x)
: (cursx-(cursx>>zoom)+((hcount-x)>>zoom));
            newy <= (zoom == 2'b00) ? (vcount-y)
: (cursy-(cursy>>zoom)+((vcount-y)>>zoom));
            image_addr1 <= newx + newy*WIDTH;
            image_addr2 <= image_addr1;

            // assign color value, chosen from 32-bit value in corresponding
address
            case(image_addr2[1:0])
                  2'b00: image_bits <= four_pixel_bits[7:0];
                  2'b01: image_bits <= four_pixel_bits[15:8];
                  2'b10: image_bits <= four_pixel_bits[23:16];
                  2'b11: image_bits <= four_pixel_bits[31:24];
            endcase

            // map R G B values to pixel
            if ((hcount >= x && hcount < (x+WIDTH)) &&
                  (vcount >= y && vcount < (y+HEIGHT))) begin
                  pixel <= {red_mapped, green_mapped, blue_mapped};
            end
            else pixel <= 24'd0;          // black pixel
      end

      // A 32 x (n/4) ROM
      // Each location stores 32 bits, equals to 4 pixels
      bench_image_rom4
image_rom(.addra(image_addr1[18:2]),.clka(pixel_clk),.douta(four_pixel_bits));

      // use color map to create 8bits Red, 8bits Green, 8 bits Blue;
      red_color_table_bench rcm
(.addra(image_bits),.clka(pixel_clk),.douta(red_mapped));
      green_color_table_bench gcm
(.addra(image_bits),.clka(pixel_clk),.douta(green_mapped));
```

```
      blue_color_table_bench bcm
(.addra(image_bits),.clka(pixel_clk),.douta(blue_mapped));

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clk, noisy, clean);
   input reset, clk, noisy;
   output clean;

   parameter NDELAY = 650000;
   parameter NBITS = 20;

   reg [NBITS-1:0] count;
   reg xnew, clean;

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

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 02-Nov-05 Ike: updated to make it completely synchronous
//
// Inputs:
//
//   reset       - active high
//   clock_27mhz - the synchronous clock
//   data        - 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
```

```
//    disp_*     - display lines used in the 6.111 labkit (rev 003 & 004)
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in,
            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_in;            // 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 [5:0] count;
   reg [7:0] reset_count;
//   reg         old_clock;
   wire      dreset;
   wire      clock = (count<27) ? 0 : 1;

   always @(posedge clock_27mhz)
     begin
      count <= reset ? 0 : (count==53 ? 0 : count+1);
      reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
//    old_clock <= clock;
     end

   assign dreset = (reset_count != 0);
   assign disp_clock = ~clock;
   wire   clock_tick = ((count==27) ? 1 : 0);
//   wire   clock_tick = clock & ~old_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
   reg [63:0] data;

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

```
always @(posedge clock_27mhz)
  if (clock_tick)
    begin
     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
            state <= state+1;
         end

         8'h04:
         begin
            // Setup the control register
            disp_rs <= 1'b1; // Select 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
            data <= data_in;
            state <= state+1;
         end

         8'h06:
         begin
            // Load the user's dot data into the dot reg, char by char
            disp_rs <= 1'b0;                 // Select the dot register
            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
                 data <= data_in;
                 dot_index <= 39;
                end
            else
              dot_index <= dot_index-1;      // else loop thru all dots
         end

       endcase // casex(state)
    end

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


//
// 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.
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09  // gph mod 11/3/2011
//
//
// Bug due to memory management will be fixed. It happens because
// the memory addressing protocol is off between ntsc2zbt.v and
// vram_display.v. There are 2 solutions:
// -. Fix the memory addressing in this module (neat addressing protocol)
//    and do memory forecast in vram_display module.
// -. Do nothing in this module and do memory forecast in vram_display
//    module (different forecast count) while cutting off reading from
//    address(0,0,0).
//
// Bug in this module causes 4 pixel on the rightmost side of the camera
// to be stored in the address that belongs to the leftmost side of the
// screen.
//
// In this example, the second method is used. NOTICE will be provided
// on the crucial source of the bug.
//
/////////////////////////////////////////////////////////////////////////////
```

```
// 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 [7:0]   din;
      input [17:0] din; //we are now storing 18 bits of y,cr,cb
   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'd0;
   parameter       ROW_START = 10'd0;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 * 788 XGA display

   reg [9:0]       col = 0;
   reg [9:0]       row = 0;
      reg [17:0] vdata = 0; //now we store 18 bits
   //reg [7:0]     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;
           vdata <= (dv && !fvh[2]) ? din : vdata;
        end
     end

   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
      reg [17:0] data[1:0]; //handling 18 bits now
//   reg [7:0] 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; //mydata now howlds two 18bit addresses
//   reg [31:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= { mydata[17:0], data[1] };

   // NOTICE : Here we have put 4 pixel delay on mydata. For example, when:
   // (x[1], y[1]) = (60, 80) and eo[1] = 0, then:
   // mydata[31:0] = ( pixel(56,160), pixel(57,160), pixel(58,160),
pixel(59,160) )
   // This is the root of the original addressing bug.


   // NOTICE : Notice that we have decided to store mydata, which
   //          contains pixel(56,160) to pixel(59,160) in address
   //          (0, 160 (10 bits), 60 >> 2 = 15 (8 bits)).
   //
   //          This protocol is dangerous, because it means
   //          pixel(0,0) to pixel(3,0) is NOT stored in address
   //          (0, 0 (10 bits), 0 (8 bits)) but is rather stored
   //          in address (0, 0 (10 bits), 4 >> 2 = 1 (8 bits)). This
   //          calculation ignores COL_START & ROW_START.
   //
   //          4 pixels from the right side of the camera input will
   //          be stored in address corresponding to x = 0.
   //
   //          To fix, delay col & row by 4 clock cycles.
   //          Delay other signals as well.

   reg [39:0] x_delay;
   reg [39:0] y_delay;
   reg [3:0] we_delay;
   reg [3:0] eo_delay;

   always @ (posedge clk)
   begin
     x_delay <= {x_delay[29:0], x[1]};
     y_delay <= {y_delay[29:0], y[1]};
     we_delay <= {we_delay[2:0], we[1]};
```

```
      eo_delay <= {eo_delay[2:0], eo[1]};
    end

    // compute address to store data in
    wire [8:0] y_addr = y_delay[38:30];
       wire [9:0] x_addr = x_delay[39:30];

    wire [18:0] myaddr = {y_addr[8:0], eo_delay[3], x_addr[9:1]};

    // Now address (0,0,0) contains pixel data(0,0) etc.


    // alternate (256x192) image data and address
    wire [35:0] mydata2 = {data[1],data[1]};
    wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};

    // update the output address and data only when four bytes ready

    reg [18:0] ntsc_addr;
    reg [35:0] ntsc_data;
    wire       ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0));

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

endmodule // ntsc_to_zbt


//
// 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 | ...
| EAV sequence
   // 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 stream.
           // whenever the synchronization code is seen, go back to the
sync_state before
           // 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_SAMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE,
`VIDEO_QUALITY}

///////////////////////////////////////////////////////////////////////////////
// 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, `INTERFACE_SELECT}

////////////////////////////////////////////////////////////////////////////////
// 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, `GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// 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_RESET, `FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// 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,
`DEFAULT_VALUE_ENABLE}

////////////////////////////////////////////////////////////////////////////////
// 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_CONTROL, `TEMPORAL_DECIMATION_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// 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,
`POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY,
`POWER_SAVE_CONTROL}

///////////////////////////////////////////////////////////////////////////////
// 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_BIRIGHTNESS_LINES, `PEAK_WHITE_UPDATE}

`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


/***************************************************************************
 **
 ** Module: ycrcb2rgb
 **
 ** Generic Equations:
 ***************************************************************************/

module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb );

output [7:0]  R, G, B;

input clk,rst;
input[9:0] Y, Cr, Cb;

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

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
        Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

always @ (posedge clk or posedge rst)
```

```
   if (rst)
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64));
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end



/*always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     R_int <= X_int + (const2 * (Cr_reg - 'd512));
     G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg -
'd512));
     B_int <= X_int + (const5 * (Cb_reg - 'd512));
     end

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

endmodule


//
// 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 [19:0] 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 [19:0] 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 = 1'b0;  // gph 2011-Nov-10
                                    // set to zero as place holder

//   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 // zbt_6111
```