A Real-Time Video Effects Processor

Ben Gelb and Venkat Chandar

December 14, 2005

Abstract

This report describes the design and implementation of a real-time video effects processor. The processor has several capabilities useful for a TV broadcast station. First, the processor allows the user to overlay two text objects and a stored still image (in fullscreen or picture-in-picture modes). Also, the user can draw on the TV screen using a mouse. The processor also lets the user capture one frame of video, and supports a digital zoom around a selectable point by 2x, 3x, or 4x. A bluescreen feature allows the user to filter out a background and replace it with a stored image. A graphical user interface on a VGA monitor controls how the various effects are applied. After discussing the design in detail, we explain our testing procedure and the main problems we had to debug. The final result is a processor with almost all of the functionality we originally designed.

Contents

1 Overview **3**

2 Description **7**
2.1 PS/2 Mouse and Keyboard Modules . . . 7
2.2 Control Module (Ben Gelb) . . . 8
2.3 Video Input Module . . . 13
2.4 Framegrab Module (Ben Gelb) . . . 13
2.5 Bluescreen Module (Ben Gelb) . . . 15
2.6 Zoom Module (Venkat Chandar) . . . 17
2.7 Overlay Module (Venkat Chandar) . . . 19
2.8 Output Module (Venkat Chandar) . . . 23

3 Testing and Debugging **24**

4 Conclusion **28**

A Verilog Code **29**
A.1 Bluescreen Code - bluescreen.v . . . 29
A.2 Framegrab Module Code - framegrab.v . . . 32
A.3 GUI Interface Objects - gui_objects.v . . . 35
A.4 Overlay Text Pixel Generation Code - cstringdisp.v . . . 38
A.5 VGA Text Pixel Generation Code - prefetch_cstring.v . . . 39
A.6 Control Module Code - control.v . . . 42
A.7 Zoom Module - zoom.v . . . 59
A.8 Overlay Module - overlay.v . . . 68
A.9 Video Output Module - video.v . . . 78
A.10 Video Output Module - video_encode.v . . . 80
A.11 Labkit Module - avtest.v . . . 82

List of Figures

1 Video Effects Processor in Operation . . . 3
2 System Block Diagram . . . 6
3 Control Module Block Diagram . . . 9
4 Font FSM State Transition Diagram . . . 12
5 Framegrab Module Block Diagram . . . 14
6 Framegrab FSM State Transition Diagram . . . 15
7 Bluescreen Module Block Diagram . . . 16
8 Bluescreen FSM State Transition Diagram . . . 17
9 Zoom Module Block Diagram . . . 18
10 Overlay Module Block Diagram . . . 20

1 Overview

The Real-Time Video Effects Processor is a system designed to aid in the production of live television broadcasts. It allows the overlay of graphics and text to the video stream, as well as drawing capabilities. It also is capable of performing a digital zoom on a selectable portion of the frame, and finally, can use bluescreen photography techniques (known as chroma keying) to replace the background with a still image while retaining the live subject in the foreground. A photograph of the working system in operation is shown in Figure 1.

Figure 1: Video Effects Processor in Operation - The flat panel monitor shows the user interface used to control the system. The television on the left shows a sample output - note the picture-in-picture window in the corner of the screen.

From the user's perspective, the system is operated through the use of a graphical user interface.

This interface is driven by input from a standard PS/2 mouse and keyboard, and its output is displayed on a standard VGA monitor. The GUI consists of an array of buttons and check boxes, as well as a large rectangular drawing surface, which is used to position objects on the television screen, as well as to allow the user to draw freehand on top of the video.

To operate the overlay functionality, a user uses the group of buttons and checkboxes near the bottom of the screen labeled “overlay”. To draw freehand on the video frame, for example, the user would click the enable checkbox next to the draw button. Then the user would click the “draw” button, then move the mouse cursor on top of the drawing surface, and click and hold. As the user moves the mouse, trails will be left behind on the screen.

If the user wishes to overlay text, he would first click the “enter text” button corresponding to one of the two text buffers, and enter the desired text on the PS/2 keyboard. When the user hits the enter key, the text is rendered into the correct overlay buffer. The user may then turn on the overlay by clicking the correct enable checkbox, and position it by clicking the correct set position button, and then clicking to the desired location on the drawing surface. To aid the user’s placing, a mouse cursor is also displayed on the TV screen.

The user may also overlay a still image by storing it in the framegrab buffer. First, the user should point a camera at the desired still image, and then click the framegrab trigger button. This will store the image in a framegrab memory. Then the user can enable the overlay of the grabbed frame by clicking on the correct enable button. If the user desires, he can click picture-in-picture enable button, which will reduce the still image to one quarter of the screen, and can then choose its position on the screen in a manner similar to the text overlays.

In addition to the overlay functionality the user may also enable a digital zoom filter, which can magnify either 2, 3, or 4 times. The center point may be changed arbitrarily as well, by clicking

the set position button and then choosing a new centerpoint on the drawing surface.

Finally, the user may use the bluescreen functionality to replace the background of the video (supposing the background is a suitable solid-color surface). The user first points the camera at the background, with no other objects in frame, and clicks the calibrate button. This runs a short routine which determines the minimum and maximum chrominance and luminance values inside of a rectangle in the center of the video frame. Then the user may enable the bluescreen by clicking the enable checkbox. Once this happens, any pixel in the video stream that falls within the values determined in the calibrate step is replaced by a pixel from the image in the framegrabber. Note that when the bluescreen functionality is enabled, the framegrabber overlay functionality is forced to be turned off, since both modules cannot read from the framegrabber at the same time.

The internals of the system are organized as seen in the system block diagram in Figure 2.

Video decode and video encode modules make up the beginning and end of the data path in our project. The video decode module decodes a 10-bit CCIR656 stream from the ADV7195 video digitizer chip and outputs a 30-bit YCrCb color value for each pixel of video, along with field, h-sync, and v-sync signals, denoted fvh. The video encode module takes the data generated by the video decode module and transforms it back into the 10-bit CCIR656 format used by the 6.111 labkit's video output chip, the ADV7194. Each of three video editing modules - bluescreen, zoom, and overlay are inserted in series in the data path. The framegrab module is connected to the video data path as well, but doesn't output modified pixel values. It only reads what's being output by the decode module.

A control module generates all of the signals necessary to disable, enable, and adjust the operation of the modules that make up the data path. These signals include an enable signal for each of the various bits of functionality, as well as a trigger signal for the framegrabber, a calibrate signal

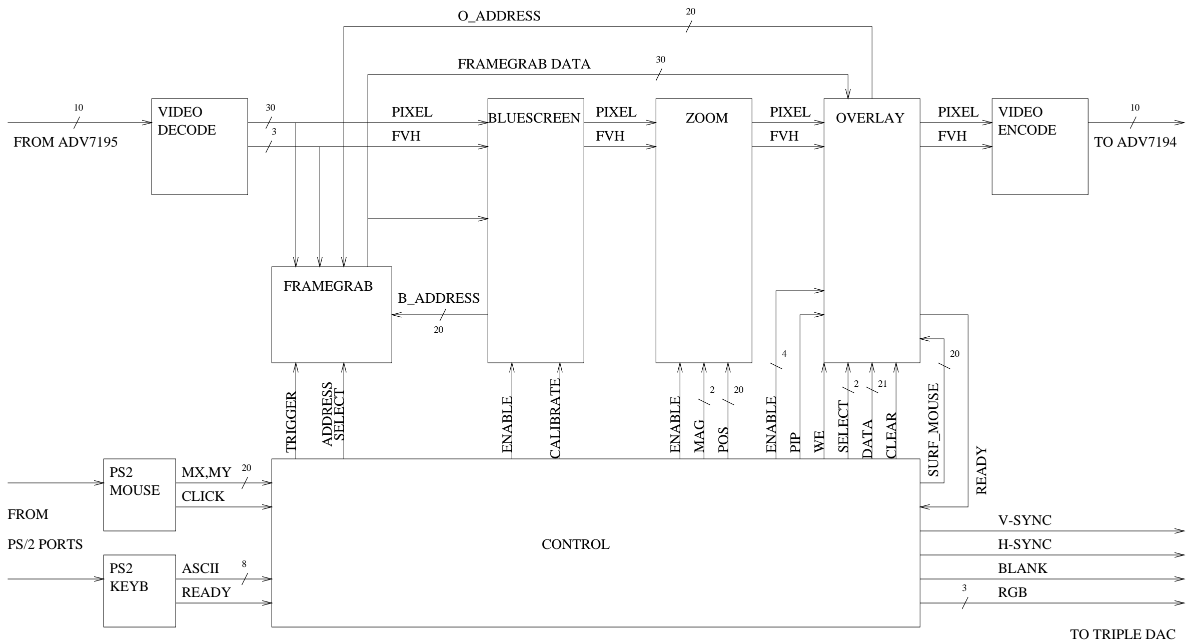

Figure 2: System Block Diagram - The video encode and decode modules define the endpoints of the data path in the system. The bluescreen, zoom, and overlay modules make up the middle of the data path, and apply transformations to the video. The framegrab module attaches to the data path so that it can capture a single frame. The control module orchestrates the operation of all of the components in the data path.

for the bluescreen module, a 20-bit center position and a 2-bit magnification value for the zoom module. There are also several signals output to the overlay module. These include a write enable line, which indicates to the overlay module to perform a write, a 3-bit select signal, which indicates what memory object should be written to by the overlay module, and a 21-bit data bus which carries addressing and pixel values. The control module also exports a clear signal for clearing the freehand drawing memory, and a 20-bit mouse position line which indicates where on the TV screen a mouse cursor should be displayed.

The control module generates the GUI spelled out in detail above and outputs the VGA h-sync, v-sync, and blanking values, along with a 3-bit RGB value. These signals are connected to the ADV7125 triple DAC which is used for VGA output. The control module receives input from the PS/2 mouse and PS/2 keyboard modules as well. The mouse sends two signals - a button click signal, which indicates when the left mouse button is pressed, and a 20-bit position value, which gives the position of the mouse on a 1024x768 pixel screen (the dimensions of the GUI). The keyboard module outputs an 8-bit ASCII code for any keys pressed, as well as a ready signal that indicates a new character is waiting.

2 Description

2.1 PS/2 Mouse and Keyboard Modules

The PS/2 mouse and keyboard modules provide user input to the control module. The PS/2 mouse module outputs a 20-bit screen position for a 1024x768 screen, as well as a button signal that provides the state of the mouse pushbuttons. The PS/2 keyboard module delivers an 8-bit ASCII code for the last character typed on a keyboard as well as a ready signal that goes high for

one cycle after a key press to indicate the presence of a new character. Both of these modules were provided as part of the 6.111 labkit code, and we didn't modify them significantly for use in this project.

2.2 Control Module (Ben Gelb)

The control module is responsible for accepting user input via PS/2 keyboard and mouse modules, and controlling the data path accordingly via an array of output signals. It's functionality falls into two categories - user interface and output logic. The user interface portion of the control subsystem is responsible for rendering a graphical interface on a VGA monitor, and responding to input (keystrokes, button clicks, etc.). The graphical interface consists of several buttons labeled with text, as well as a large rectangular drawing surface, that corresponds to the pixel locations on the television. The output logic develops signals used to control the data path based on the state of the user interface submodules. A block diagram of the control module is shown in Figure 3.

VGA output is generated at 1024x768 resolution. A 65MHz pixel clock is used to generate h-sync, v-sync, along with h-count and v-count signals. A DCM module is used to generate the 65MHz clock.

Four types of objects are displayed on the VGA screen - check boxes, option buttons, push-buttons and text labels. Each one of these items displayed on-screen is an instantiation of the appropriate module. Each instance of a module acts as a sprite, and accepts an h-count and v-count signal as input, and outputs a corresponding RGB value. Additionally, all of the interactive objects take as input a mouse position, and a input that indicates a mouse button press. This allows the objects to change state or output level if a mouse click is registered.

Check boxes contain their own internal state register to track whether they are checked or

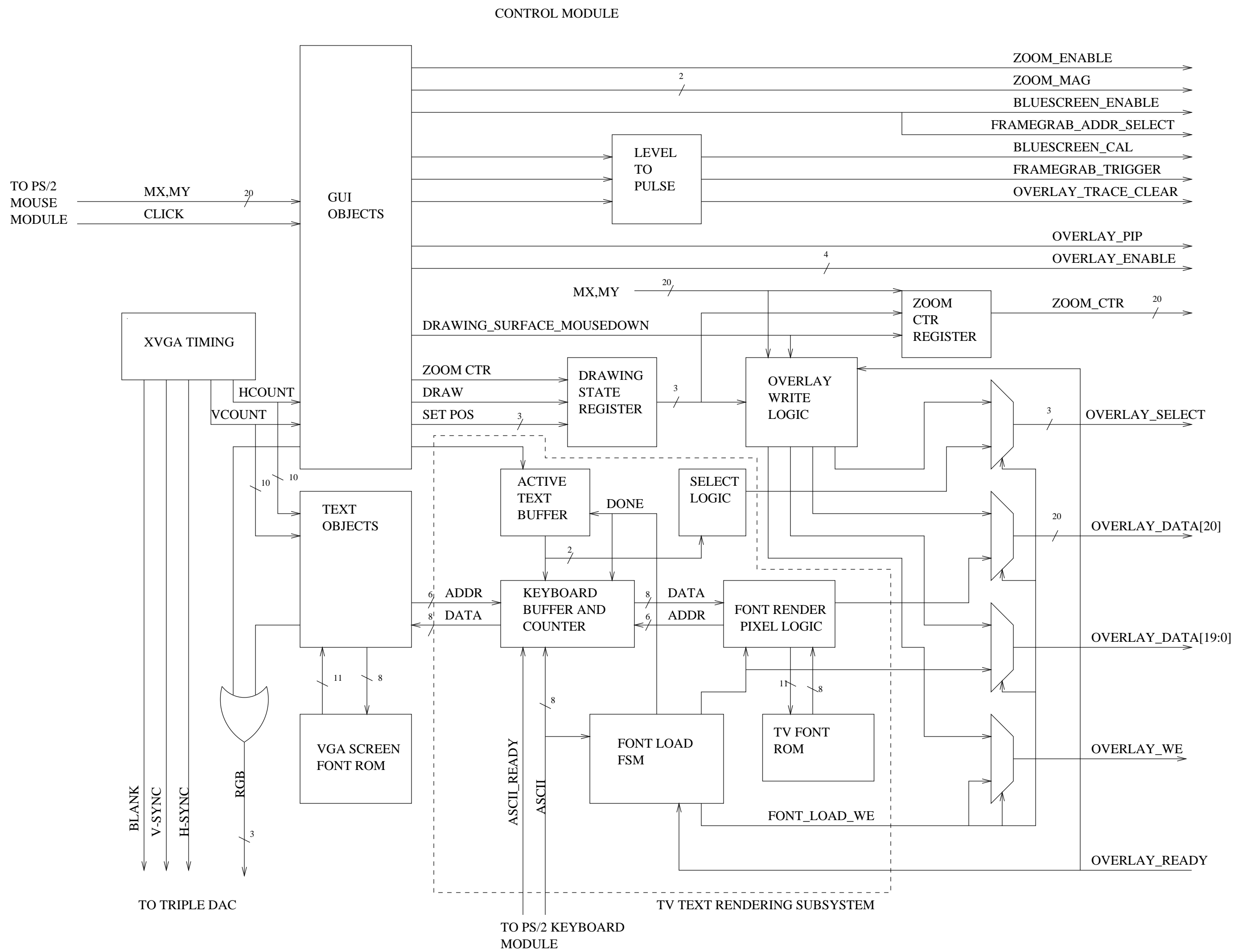

Figure 3: Control Module Block Diagram - The GUI functionality interfaces directly with the user. The logic blocks attached to it use the state of the UI objects to generate control signals to drive the other modules.

unchecked. The state of this register also affects the value of the sprite - if the internal state register is 0, the sprite is red in color, if it is 1, then the output color is green.

Option buttons are similar to check boxes in appearance, but are grouped together so that only one may be active at a time. As such, they are wired to use a single external state register.

Pushbuttons are stateless, and simply output a high level when the mouse button is depressed and mouse position is within bounds of the pushbutton.

Text blocks are used to label all of the various controls. Each text block uses an instantiation of a cstring_display module. Similar to the various button objects, the text blocks take an h-count and v-count signal and output a corresponding pixel value. The pixel value is generated through the use of a font ROM. At each h-count and v-count value, the cstring_display module outputs an address to be sent to a font ROM module, and receives a byte of data that represents 8 pixels of a character. The addressing lines from each cstring_display module are ored together in order to share a single font ROM. In order to meet timing requirements for the VGA display, the cstring_display module must pre-fetch data from the font rom beginning 16 pixels (2 character spaces) prior to the first character of a text block.

One of the text blocks uses a slightly modified version of the cstring_display module that reads from a register file containing any keyboard input. The keyboard register file contains 48 locations (one byte each) and writes in ASCII data received from the PS/2 keyboard module (this data is later used by the output logic to generate text overlays) whenever the PS/2 keyboard module indicates that a new character has been typed. The keyboard input buffer uses a counter to keep track of the next location to store a new character. All locations above the value in the counter are blanked out when fed to the cstring_display module. This way, the keyboard input buffer may be cleared simply by resetting the counter value to 0.

The RGB outputs from each instance of a GUI module are ored together to form a single composite RGB signal, which is latched and fed, along with the h-sync, v-sync and blanking signals to the triple DAC hardware used for VGA. In order to deal with excessive combination delay introduced by oring together such a large number of signals, a two-stage pipeline was used to spread out the delay across two cycles. This had the effect of removing any on-screen glitches in the VGA output caused by excessive latency.

The output logic segment of the control module uses the signals from the user interface portion to generate output signals to send to the various modules that form the data path (the framegrab, bluescreen, zoom and overlay modules). In some cases this is nothing more than simply assigning the output of a control object to an output signal. Such examples are the various enable signals used to switch various modules on and off in the data path, as well as the picture-in-picture enable signal for the overlay module, the address_select signal for the framegrab module, and the zoom magnitude signal for the zoom module.

A handful of other signals, such as the framegrab trigger, bluescreen calibrate, and overlay clear signals are simply the output of pushbuttons fed through a level-to-pulse converter that outputs a single high pulse after the rising edge of the pushbutton signal (the instant when the button is clicked).

When any of the set position buttons, along with the draw button are clicked, a register inside of the control module is set to indicate which function will be executed upon any input to the drawing surface.

One of these states is to set the zoom center position. If this state is active and a location on the drawing surface is clicked, the corresponding position value for the TV screen is stored into the zoom center register. The value of this register is used as the zoom position output which is fed to

the zoom module.

Each of the other set position states, along with the draw state, generate output for the overlay module. Depending on which function is active corresponding overlay_select and overlay_data signals are generated, according to Table 1, and the overlay_we signal is asserted high.

Table 1: Overlay Programming Specification

Programming Function	Select Bus Value	Data Bus Format
Text1 Video Buffer	000	Bits 19:0 for address, bit 20 for value
Text1 Position	001	Bits 19:0 for value
Text2 Video Buffer	010	Bits 19:0 for address, bit 20 for value
Text2 Position	011	Bits 19:0 for value
Framegrab PIP Position	100	Bits 19:0 for value
Trace Video Buffer	110	Bits 19:0 for address, bit 20 for value

The last part of the output logic is the overlay text rendering subsystem. This part makes use of a small FSM shown in Figure 4.

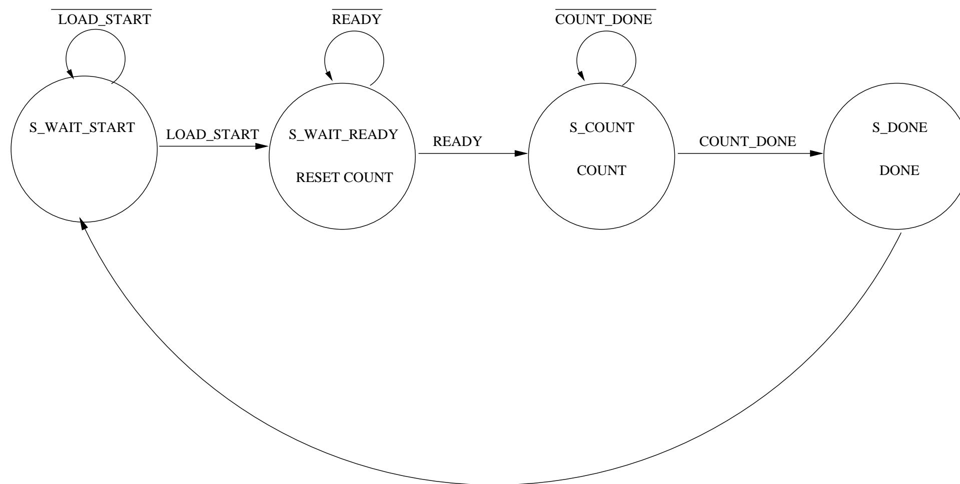

Figure 4: Font FSM State Transition Diagram - The FSM first waits for a start signal (given by the user hitting the enter key on the keyboard). Then it waits for the overlay module to assert a high ready signal. Finally it enters the count state, which initiates a counter to step through all of the pixel addresses needed to fully render the text. Then it asserts a done signal which resets the keyboard buffer before returning to the first wait state.

The FSM first waits for a return character to be entered from the keyboard. This is the signal to start rendering. It then waits for the ready signal from the overlay module to be high, and then outputs a count signal which causes a counter to step though each pixel address in the text overlay buffer in the overlay module (720x24 pixels). When the counter reaches its maximum count, it asserts the count_done signal. The FSM then transitions into the done state for a single cycle, which outputs a done signal to reset the keyboard buffer, before returning to the initial wait state.

When the FSM is active (in one of its write states), the text rendering subsystem takes over the overlay_data, overlay_select and overlay_we lines. The text rendering system generates the appropriate select signal, as listed in Table 1, according to which text buffer is active. The address on the data output is generated by the font loading FSM, and the pixel value is generated through the use of a second font ROM.

2.3 Video Input Module

The video input module takes in a video stream in 10-bit CCIR656 format from the ADV7195 video digitization chip on the labkit (connected to our video camera). It generates a 30-bit pixel value in YCrCb format, as well as field, vertical, and horizontal sync signals. This code was provided as part of the 6.111 labkit code, and we did not modify it significantly for use in our project.

2.4 Framegrab Module (Ben Gelb)

The framegrab modules captures a single frame of video and stores it into one of the ZBT RAMs on the 6.111 labkit. Each pixel of video is stored into a 36-bit data word on the ZBT (6 bits are unused). A small FSM is used to control the storing operation. In the read mode, a multiplexer selects whether addresses will come from the overlay or the bluescreen module, depending on an

address select input from the control module. During the write phase, addressing is provided by an internal row and column counter, and writing is controlled by a small FSM. A block diagram of the framegrab module is shown in Figure 5.

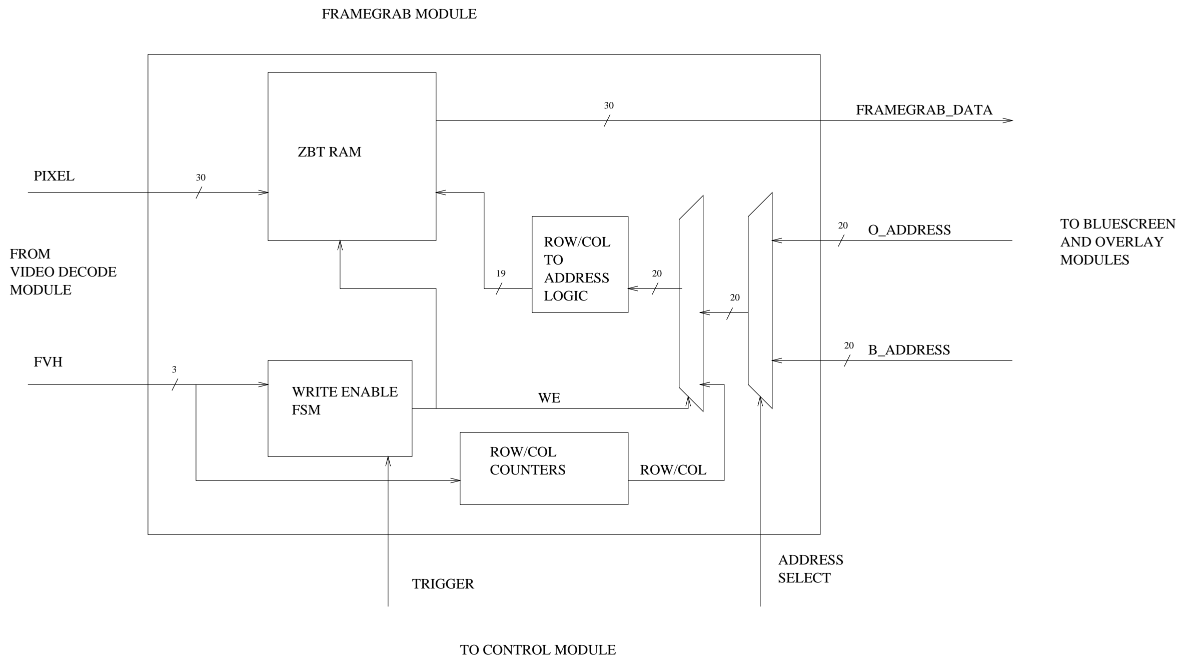

Figure 5: Framegrab Module Block Diagram - A multiplexer selects which module gets to control the read address of the ZBT. When a trigger signal is received from the control module, the write enable FSM generates a write enable signal for synchronized with a frame of video and row and column counters provide addressing.

Addresses provided to the framegrab module are 20-bits and consist of 10-bits row and 10-bits column data. A simple piece of logic maps the allowable row and column values into a 19-bit address used directly with the ZBT.

A state transition diagram from the write enable FSM is shown in Figure 6.

The FSM waits for a trigger signal from the control module, then waits for a high-to-low transition of the field signal (indicating the start of the first field). It then waits for both fields to pass before ending in the first wait state again.

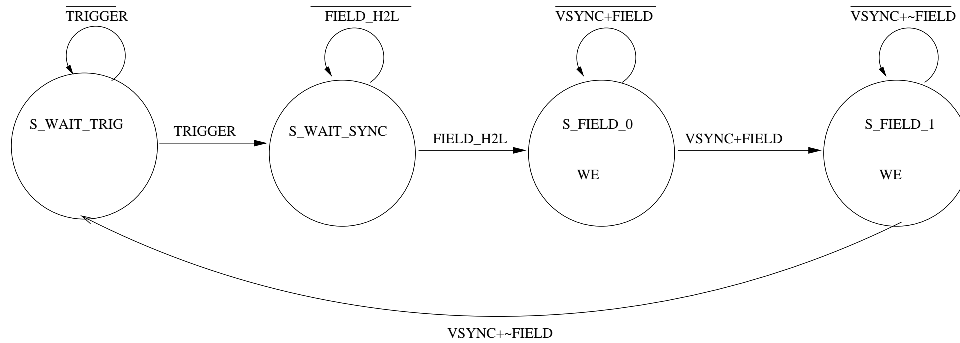

Figure 6: Framegrab FSM State Transition Diagram - The FSM waits for a trigger signal from the control module, then for a low-going edge of the field signal to make sure it starts storing data at the beginning of the first field. Then it asserts a write enable signal for the first and second field of video before returning to the wait state.

2.5 Bluescreen Module (Ben Gelb)

The bluescreen module performs chroma keying. This is the process of replacing a solid color background by a still image. Such techniques are very commonly used in TV weather forecasts, as well as special effects in movies. The bluescreen module uses a simple calibrate routine to determine the minimum and maximum chrominance and luminance values inside a rectangular sample of the video picture. When the module is enabled, these values are used to determine whether or not a pixel is "blue" and should be replaced. A block diagram of the bluescreen module is shown in Figure 7.

The bluescreen module takes in a 30-bit video pixel along with field, v-sync, and h-sync signals, it also outputs a 30-bit video pixel, along with v-sync, h-sync and field signals. When enabled via an enable signal from the control module, a multiplexer is used to switch between passing through the video stream, or replacing it with a pixel at the correct screen position from the framegrab memory. The data from the framegrab memory is fetched regardless of whether or not it is used, so that it is available at the correct time.

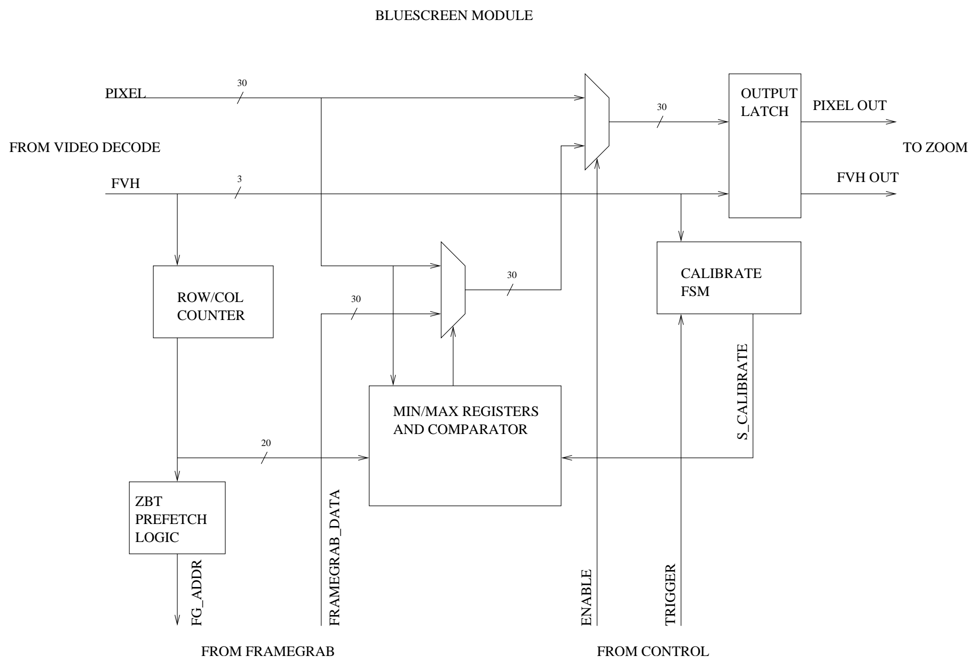

Figure 7: Bluescreen Module Block Diagram - When enabled minimum and maximum color values are compared to the incoming video to determine if pixels should be replaced by pixels from a still image in the framegrab module. When a calibrate signal is received, an FSM drives a calibrate routine to set the values of the min and max registers.

When a trigger signal is received from the control module, a simple FSM is activated to set find the minimum and maximum chrominance and luminance values in a small rectangle in the middle of the screen. A state transition diagram is shown in Figure 8.

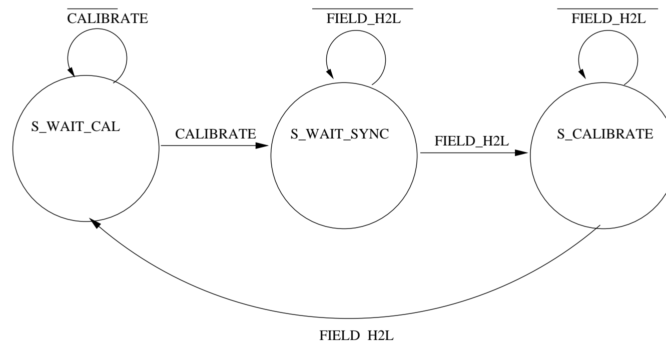

Figure 8: Bluescreen FSM State Transition Diagram - The FSM waits for a calibrate signal, synchs with the video field transition, and then enters the calibrate state until the next frame. While in the calibrate state, registers are latched when the screen position is in a rectangle in the middle of the screen to record min and max values.

When the FSM is in the calibrate state, a bank of six registers record the min and max values of Cr, Cb, and Y in a 20x40 rectangle in the center of the screen. When the line and column counters are in the correct range, the registers are latched on each clock edge with either their current value, or a new value if it is larger than the current value (or smaller than the current value if it is a minimum value register).

2.6 Zoom Module (Venkat Chandar)

The zoom module allows the user to select a position of the frame to magnify using the mouse, and zooms in by a variable scale factor of either 2x, 3x, or 4x. The inputs to the module are the field, vertical, and horizontal sync signals, and a 30-bit pixel value. Also, a 20-bit position input specifies the centerpoint to zoom around, and a 2-bit magnification input specifies the scale factor. The zoom enable signal enables the zoom functionality when asserted. The zoom module outputs

updated field, vertical and horizontal sync signals, and a modified pixel value. Figure 9 shows the block diagram for the zoom module.

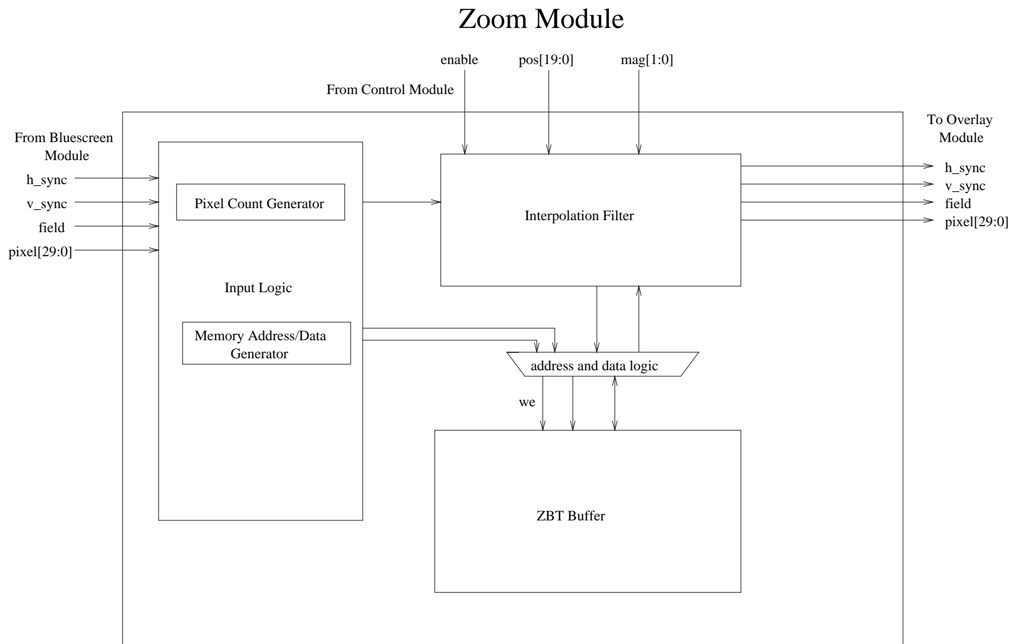

Figure 9: Zoom Module Block Diagram - ZBT RAM stores previous field for use by the Interpolation Filter.

Our design for the zoom module consists of two main blocks - an interpolation filter, and a ZBT RAM. The ZBT RAM stores previous pixel values, which can then be read by the interpolation filter to compute the output pixel value. Our scheme uses the ZBT RAM as a double buffer. This means that we treat one part of the memory as the current write buffer, i.e. we write the incoming pixels to this buffer. The other part of the memory is the read buffer, which the interpolation filter can access to find the previous field's pixel values. Every time the field transitions, we swap the read and write buffers. In our implementation, swapping the to buffers is accomplished by adding a suitable field-dependent offset to the memory read and write addresses.

We designed two interpolation filters. The first is a sample/hold filter. Essentially, to zoom in, this filter simply copies each pixel in the read buffer the appropriate number of times. For 2x and

4x, the math involved can be accomplished using shifts to divide by 2 or 4. For 3x, we use two lookup tables so that we can divide the horizontal and vertical position counters by 3. The code in the appendix uses a BRAM for this, but a sensible implementation would use a ROM. We indicate the small changes to the code that would allow us to use a ROM instead of a BRAM. We used a BRAM simply because we did not have time to make a .coe file to initialize the memory.

The second filter we designed is a bilinear filter. This filter takes a weighted average of the four nearest pixels. This filter was not completed, but we describe the design we used because we almost finished implementing this filter, and in the testing section we will describe what is left to do to complete the implementation. The design for the bilinear filter uses 3 BRAM buffers. Each of these BRAMs stores one line of video from the ZBT read buffer. One BRAM is designated as the write buffer, while the other two are read buffers. By reading from the read buffers (which are dual port), we can access the four pixels needed for bilinear interpolation. Simultaneously, the write buffer gets the next line of video from the ZBT read buffer. At the end of the line, we rotate which BRAM is the write buffer, so that the BRAMs are used in a kind of "ring" structure. This allows us to perform bilinear interpolation for the entire field.

2.7 Overlay Module (Venkat Chandar)

The overlay module implements the overlay functionality described in the overview. This module takes in the field, as well as vertical and horizontal sync signals. These signals are used to produce row and column counters which determine the current pixel's position. The module also takes in a 30-bit pixel input, which has already been processed by several earlier modules in the video data path. Figure 10 shows the block diagram for the overlay module.

The overlay module takes in several inputs from the control module. There are enable signals

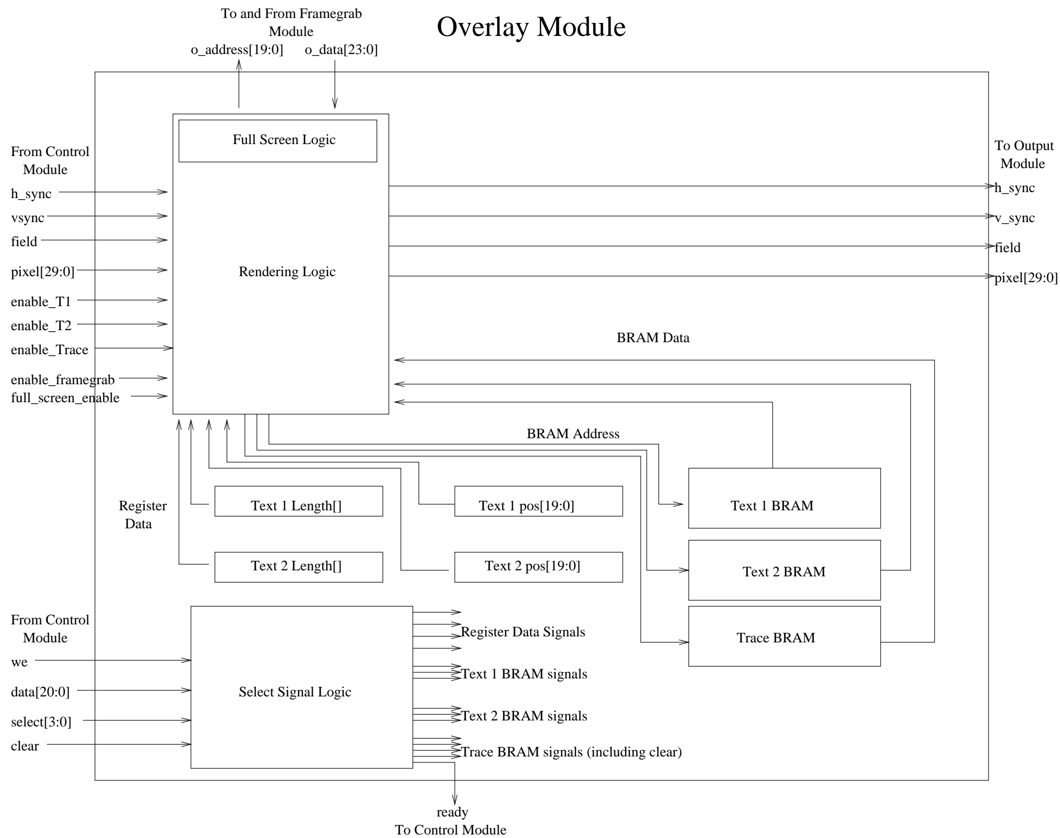

Figure 10: Overlay Module Block Diagram - Data flows from the select logic to the BRAMs to the rendering logic.

controlling whether or not to apply the trace, text, or framegrab overlay objects to the input video stream. The 21-bit data bus allows the control module to write the correct data into various memories inside of the overlay module. The 3-bit select input tells the overlay module how to parse the data on the bus, thus controlling what gets stored in the memories. The control module also outputs a write-enable signal. Data on the bus is only written to memory when the write-enable signal is asserted. The overlay module also receives a 20-bit mouse position, which is used to render a cursor so that the user knows where the mouse is located on the TV monitor. The last input from the control module is a clear signal. When clear is asserted, the overlay module clears the trace buffer. This takes many clock cycles, so when the overlay module has finished clearing the buffer and is ready to! accept data again, the overlay module asserts the ready signal so that the control module knows that data can be sent for processing.

Finally, the overlay module interacts with the framegrab module by outputting a 20-bit address and receiving a 30-bit value from the ZBT buffer inside of the framegrab module.

The outputs of the overlay module include updated field, vertical and horizontal sync signals. Also, the module outputs modified pixel data. These signals are sent to the video output module. The ready signal and the framegrab address were descried above.

The overlay module has two main pieces. There is rendering logic, which looks at the current pixel position and the trace, text, and framegrab enable signals, and modifies the pixel value appropriately. The other piece of the overlay module is the select logic, which processes the select and data bus inputs and updates the memories properly. Information passes from the select logic to the rendering logic through several memories, which we describe next.

The overlay module has 3 BRAM buffers to store the trace object and two text objects. The trace buffer stores 1 bit per pixel, so it has 240*720 = 172800 locations (the reason for 240 instead

of 480 is that we did not want the trace object to flicker; to accomplish this, the even and odd fields are assigned the same value by the trace object, so only half as many locations are needed). The text is rendered to have a height of 24 pixels. Thus, the text buffers have only 24*720 = 17280 locations. The rendering logic and the select logic could end up simultaneously trying to access the same memory. For this reason, we used dual-port BRAMs with one read port and one write port (the rendering logic only needs to read, and the select logic only needs to write). The addressing scheme used for the buffers is to store the pixel at (row, col) at position row*720 + col. For the trace buffer, row is actually row truncated to cutoff the least significant bit, since both fields get the s! ame value. For the text buffers, row and column are not actually the row and column of the current pixel, but rather the row and column relative to the top left corner of the text, i.e. the row is a number between 0 and 23.

The rendering logic computes addresses to read from at every cycle using the addressing formula given above. The logic also sends an address to the framegrab module. Technically, the logic should compute the address for the next pixel, so that the data coming out of the buffers is correct at the next cycle. For our addressing, this just means that we should add 1 to the address, and we need to wrap around when we reach the largest address. Our implementation does not prefetch like this, so we actually output pixels a cycle later than they should be output. It is easy to add the prefetching logic if one is worried about edge effects or the shift by one pixel that this causes. However, we note that the TV monitor we used does not even show the entire frame, so the edges of the frame are not displayed and no practical improvement would be observed by including this additional logic. The one exception to this is the picture-in-picture functionality. In this case, the edge of the! frame becomes visible, so we implement a crude form of prefetching so that the edges show actual data instead of blanking data or random (unwritten) memory data. The code in the

appendix illustrates this prefetching method and should give the interested reader a good idea of how to implement a correct prefetching strategy if this is necessary.

The rendering logic also reads the data coming out of the buffers and the data from the framegrab module. Combinational logic is used to determine whether the incoming pixel value should be changed or not. Essentially, if a certain overlay object is enabled and the data from the buffer is 1, we change the current pixel value to white. Otherwise, we pass the incoming pixel value through unmodified. If the framegrab overlay is enabled, we change the current pixel value to the data value coming in from the framegrab module.

The select logic is essentially a large case statement that generates the write addresses and write data for the buffers and several position registers. The code in Table 1 is used to parse the data bus accordingly.

2.8 Output Module (Venkat Chandar)

The output module takes in the modified video stream and produces an NTSC output to send to the TV monitor. Using the field, vertical, and horizontal sync signals, the module determines whether to look at the incoming pixel data or to insert the appropriate timecodes and blanking data. During periods in which the pixel data is valid (i.e. outside the blanking intervals), the output module splits the 30-bit pixel values into 10-bit luminance and chrominance values so that the outgoing stream is in the correct format, alternating between luminance and the two different chrominance values. Finally, the encoded stream is sent to the labkit's adv7194 chip so that it can be output to the TV.

3 Testing and Debugging

Over the course of implementing our system, several challenges arose which we had to deal in order to achieve the desired functionality. In order to make debugging manageable, each module was first tested individually. This definitely helped us save time, because as the project got larger, it took a long time for the Xilinx tools to synthesize our project. Synthesizing a single module was significantly faster.

The first step in our implementation was to build a working video output module so that video could come in from the video encoder module, pass through our output module, and display properly on the TV monitor. This step actually turned out to be one of the more time-consuming parts of the project. By modifying the test code supplied with the labkit (in the video.v file), we were able to pass video through the input and output modules and get the correct image on the TV. However, the original code was quite messy, and we did not fully understand the intricacies of the NTSC encoding process. After examining several datasheets, we finally understood how NTSC encoding works, and a cleaner encoder was written. By this point in time, the overlay and zoom modules were being written, so unfortunately the code used by these two modules to generate row and column position counters is not very clean. We believe that the cleaner code used in the bluescreen and framegrab modules to gene! rate the position counters can replace the code used in the zoom and overlay modules, but we did not have time to test this.

The overlay module did not require much debugging. A dummy control module was written to verify the operation of the overlay module. No major problems were found, and the dummy module was able to overlay stripes using the trace functionality and the text functionality. The only debugging needed for the overlay module came at the very end, when the entire system was integrated. We realized that the picture-in-picture functionality would display the edge of the

frame, so we implemented a crude form of prefetching to fix this.

While the overlay module was being tested, the control module was also tested. There were never any major problems with the control module logic, however there were problems that cropped up in regards to the rendering of the GUI. Once all of the text labels were in place that were supposed to be part of the GUI, much of the text became noisy or distorted. It was pretty obvious that this was due to latency introduced by the large combinational circuitry used to combine all of the addressing requests sent to the font ROM. It took me a while to arrive at it, but the eventual (and in retrospect, very correct) solution is to prefetch pixel values from the font ROM. Since a single 8-bit word of data corresponds to a whole line of a character (8 pixels), a fetch only need be done every 8 cycles. By initiating the fetch well in advance of each character, the combinational logic that combined all of the addressing requests from the various text blocks could be allowed to settle for 8 cycles instead of just one. This made all the difference, and cleaned up the text display dramatically.

There was also a bit of noise (though less severe) on the RGB signal due to a similar problem. Since all of the text blocks and other graphical objects are sprites, their RGB values must all be ored together before they can be fed to the triple DAC. In order to eliminate the noise caused by the combinational delay exceeding the cycle time, the oring step was broken into a two-stage pipeline, so that the operation could be spread out over two cycles.

At this point, we decided to perform some system integration, so we setup the control-overlay interface and tested the two modules together. After some minor changes, the interface worked and we were able to draw on the TV monitor using the trace functionality. We decided that it would be useful to have a cursor indicating the mouse position on the TV monitor, so we added this to the overlay module.

After integrating the control and overlay modules, we began work on the zoom module. Over the course of implementing this module we needed to change our original design. The first step in implementing the zoom module was to get the ZBT working properly. When we designed the zoom module, our plan was to clock the ZBT using a faster clock. Recall from section 2.6 that the zoom module uses a double buffer memory scheme. By clocking the ZBT at a faster rate, we wanted to read and write in one cycle of the slower (27mhz)tv line-locked clock. We were able to implement a zoom module that clocked the ZBT using the 65mhz VGA clock. However, even with synchronizers, this method introduced an unacceptable amount of noise at the output, so that the image on the TV had many wrong pixels. To solve this problem, we developed a new design taking advantage of how the video input module works. The NTSC video stream alternates between luminance and chrominance. Thus, the video input module only actually outputs a new pixel value every 2 clock cycles. So, to implement double buffering, we interleave the reads and writes so that we read every other clock cycle. This means that the ZBT can be clocked at the same rate as the tv line-locked clock. Using this new design, we eliminated the noise and generated a much cleaner image at the output.

The next problem we encountered with the zoom module was that it had a poor response when the camera was moved around. This was caused by the fact that our module de-interlaced the stream by storing an entire frame (2 fields) in each of the buffers. When the camera was moved, the odd field would be at a slightly different position than the even field, resulting in horizontal bars that appeared on the screen when the camera was moved to quickly. To solve this problem, we changed the memory scheme so that only 1 field as stored in each buffer. This changes the math for the zoom module slightly, but after minor changes we implemented this scheme and produced a zoom module that responds much better to camera movement (in particular, the horizontal bars

disappeared).

The last part of the zoom module to implement was the bilinear interpolation filter. Due to time constraints, this addition to the zoom module was not completed. However, we did make significant progress on this part, and we feel that it would take at most a day or two to complete the implementation. Using the logic analyzer, we were able to verify that the memory scheme using 3 line buffers, as described in section 2.6, was working properly. Also, horizontal interpolation worked properly. However, we did not have enough time to debug the vertical interpolation. We know that at least one reason that the vertical interpolation failed is that the formula we used was written assuming the old design, where an entire frame was stored in each buffer instead of one field. The main challenge in implementing the bilinear zoom filter was that we did not know enough about the details of Verilog arithmetic. Secondarily we also had concerns about the details of the YCrCb color spa! ce. Specifically, we discovered that YCrCb is not a linear function of the RGB color space, so we were worried that a bilinear interpolation might not produce the correct color outputs.

While implementing and testing the zoom module, the framegrab and bluescreen modules were also tested. The framegrab module was tested by using the overlay module to display the contents of the ZBT memory written to by the framegrabber. The framegrabber at first had a bug where sometimes it would only capture a single field of a frame, instead of a complete frame. It turned out that due to the way h-sync and v-sync pulses were generated by the video decode module, that it was possible to witness v-sync pulses with both even and odd field values in the same blanking interval. Since the FSM in the framegrab module originally transitioned at a v-sync pulse with a particular field value, it could sometimes essentially skip the writing of one of the fields. The solution was to trigger the state transitions on the rising and falling edges of the field signal. This resulted in a full capture of both frames.

The bluescreen module required no debugging. It worked the first time it was written, and we were able to run a nice demonstration using a weather map.

Integrating all the modules into one system went quite smoothly. The complex modules were tested individually, so there were no major surprises when we put the modules together.

4 Conclusion

This paper has described the design and implementation of a real-time video effects processor. This processor is capable of overlaying text and stored image, and also allows the user to draw on the TV screen using a mouse. In addition, the processor supports 2x, 3x, and 4x zoom functionality. The processor can store one frame of video in memory, and can filter out a background using the bluescreen functionality.

The final implementation of our system achieves almost all of the functionality of our original design. The control module functions properly, and the user can to overlay text or draw with the mouse. Also, the picture-in-picture and fullscreen overlay functionality works as planned. The zoom module can scale by variable zoom factors, and the user can set the centerpoint of the zoom. The bilinear interpolation filter was not finished due to time constraints, but the design of the filter appears to be sound and only a small amount of debugging would be necessary to finish the implementation of this feature. Finally, the framegrab and bluescreen modules are fully functional.

There are several improvements that could be made to this project in the future. If we had more memory, a useful feature to add is instant-replay. We could implement this easily by extending the framegrab module. The zoom module would be easier to use if we added a picture-in-picture feature so that the user knows which portion of the frame he or she is zooming in on. Implementing this would just require some changes to the math in the zoom module.

This project taught us many things about digital system design. After a substantial amount of time and effort, we learned how the NTSC encoding process works. Also, we learned how to think about aspects of digital systems which do not arise in standard programming. For example, we had think carefully about the timing requirements for the ZBT RAM when implementing several modules. As always, spending time to develop a modular design helped us in testing the system, and made it easy to add features incrementally.

We would like to thank the 6.111 staff for all of their help with this project.

A Verilog Code

A.1 Bluescreen Code - bluescreen.v

```
// bluescreen.v
// detects and replaces 'blue' pixels with data from
// framegrab buffer

module bluescreen (reset, tv_in_line_clock1, enable, cal_start,
        ycrcb_in, fvh_in, ycrcb_out, fvh_out, framegrab_addr, framegrab_data);

input reset, tv_in_line_clock1;
input [29:0] ycrcb_in;
input [2:0] fvh_in;
input enable;
input cal_start;

output [2:0] fvh_out;
output [29:0] ycrcb_out;
reg [29:0] ycrcb_out;

output [19:0] framegrab_addr;
input [29:0] framegrab_data;

reg [10:0] sample_count;
reg [9:0] line_count;

reg [2:0] fvh_out;
wire [2:0] fvh_in;
```

```
wire real_vsync;
assign real_vsync = fvh_in[1]&~fvh_in[0]; //beginning of line

reg old_field;
always @ (negedge tv_in_line_clock1) old_field <= fvh_in[2];

wire field_h2l;
assign field_h2l = old_field&~fvh_in[2];

wire active_video;
assign active_video = (sample_count < 1440);

wire [29:0] ycrcb_internal, ycrcb_in;
wire is_blue;

// register for storing min and max color values
reg [9:0] y_max, y_min, cr_max, cr_min, cb_max, cb_min;

// line count

    always @ (negedge tv_in_line_clock1) begin
        if (reset|real_vsync) begin //vsync
        sample_count <= 0;
        line_count <= 0;
        end
        else if(sample_count==1715) begin //hsync
            line_count <= line_count + 1;
            sample_count <= 0;
        end
        else sample_count <= sample_count+1;
    end

// framegrab address generation
// this needs to be two cycles ahead, due to ZBT lag
reg [19:0] framegrab_addr;
always @ (sample_count) begin
    if(sample_count < 1398)
        framegrab_addr =
            {line_count[8:0], fvh_in[2], sample_count[10:1]} + 2;
    else if (sample_count == 1714)
        framegrab_addr =
            {line_count[8:0], fvh_in[2], 10'd0};
    else if (sample_count == 1714)
        framegrab_addr = {line_count[8:0], fvh_in[2], 10'd1};
```

```
    else framegrab_addr = 20'bxxxxxxxxxxxxxxxxxxxx; //dont care
end

// output

    // latch the output so we don't propagate any combinational delays
    always @ (negedge tv_in_line_clock1) begin
        fvh_out <= fvh_in;
        ycrcb_out <= ycrcb_internal;
    end

    assign is_blue = (ycrcb_in[29:20] >= y_min & ycrcb_in[29:20] <= y_max)
        & (ycrcb_in[19:10] >= cr_min & ycrcb_in[19:10] <= cr_max)
        & (ycrcb_in[9:0] >= cb_min & ycrcb_in[9:0] <= cb_max);

    assign ycrcb_internal = (active_video&enable&is_blue ?
                        framegrab_data : ycrcb_in);

// calibrate FSM

reg [1:0] state;
parameter S_WAIT_START = 0;
parameter S_WAIT_FIELD = 1;
parameter S_CALIBRATE = 2;

    always @ (negedge tv_in_line_clock1) begin
        if(reset) state <= S_WAIT_START;
        else case (state)
            S_WAIT_START: state <=
                (cal_start ? S_WAIT_FIELD : S_WAIT_START);
            S_WAIT_FIELD: state <=
                (field_h2l ? S_CALIBRATE : S_WAIT_FIELD);
            S_CALIBRATE: state <=
                (field_h2l ? S_WAIT_START : S_CALIBRATE);
            default: state <= S_WAIT_START;
        endcase
    end

// when in the CALIBRATE state, find min and max pixel
// values for small rectangle in center of screen

    always @ (negedge tv_in_line_clock1) begin
        if(state == S_CALIBRATE) begin

            if(line_count==101 //initialize the min/max values
```

```
            & sample_count == 621
            & fvh_in[2]==0) begin

            y_max <= ycrcb_in[29:20];
            y_min <= ycrcb_in[29:20];
            cr_max <= ycrcb_in[19:10];
            cr_max <= ycrcb_in[19:10];
            cb_max <= ycrcb_in[9:0];
            cb_max <= ycrcb_in[9:0];
        end
        else if ((line_count > 100 & line_count < 140) &
            (sample_count > 620 & sample_count < 720)) begin

        y_max <= (ycrcb_in[29:20] > y_max ?
                ycrcb_in[29:20] : y_max);
        y_min <= (ycrcb_in[29:20] < y_min ?
                ycrcb_in[29:20] : y_min);

        cr_max <= (ycrcb_in[19:10] > cr_max ?
                ycrcb_in[19:10] : cr_max);
        cr_min <= (ycrcb_in[19:10] < cr_min ?
                ycrcb_in[19:10] : cr_min);

        cb_max <= (ycrcb_in[9:0] > cb_max ?
                ycrcb_in[9:0] : cb_max);
        cb_min <= (ycrcb_in[9:0] < cb_min ?
                ycrcb_in[9:0] : cb_min);

        end
    end
end

endmodule
```

A.2 Framegrab Module Code - framegrab.v

```
// framegrab.v
// After a trigger signal, write a frame (both fields) to ZBT memory.
// When not writing, fetch a value from ZBT at address given by one of the
// addressing ports, from either the overlay or bluescreen module

module framegrab (reset, tv_in_line_clock1, ycrcb_in, fvh_in, vram_read_data,
    vram_write_data, vram_addr, vram_we, fg_trig, bs_read_addr,
    overlay_read_addr, read_data, addr_select, state);

input reset, tv_in_line_clock1;
```

```
input [29:0] ycrcb_in;
input [2:0] fvh_in;
input fg_trig;
output [35:0] vram_write_data;
input [35:0] vram_read_data;
output [18:0] vram_addr;
output vram_we;
input [19:0] overlay_read_addr;
input [19:0] bs_read_addr;
input addr_select;

output [29:0] read_data;
output [1:0] state;

reg [10:0] sample_count;
reg [9:0] line_count;
reg old_field;

// find transition of field signals

always @ (negedge tv_in_line_clock1) old_field <= fvh_in[2];

wire field_l2h;
wire field_h2l;

assign field_l2h = ~old_field&fvh_in[2];
assign field_h2l = old_field&~fvh_in[2];

// vsync occurring at beginning of line

wire real_vsync;
assign real_vsync = fvh_in[1]&~fvh_in[0];

// line/column counter

    always @ (negedge tv_in_line_clock1) begin
        if (reset|real_vsync) begin //vsync
        sample_count <= 0;
        line_count <= 0;
        end
        else if(sample_count==1715) begin //hsync
            line_count <= line_count + 1;
            sample_count <= 0;
        end
        else sample_count <= sample_count+1;
```

```
    end
// compute an address for the ZBT ram from the row/col values
// use field signal as lowest order bit for row so that lines
// are deinterlaced in memory

wire [18:0] vram_write_addr;
assign vram_write_addr = ((line_count << 1) + fvh_in[2])*720 + (sample_count >> 1);

wire [18:0] vram_read_addr;
wire [29:0] read_data;
wire [19:0] bs_read_addr;
wire [19:0] overlay_read_addr;
wire [19:0] read_addr;

assign read_addr = (addr_select ?  bs_read_addr : overlay_read_addr);
assign vram_read_addr = (read_addr[19:10] * 720) + (read_addr[9:0]);
assign read_data = vram_read_data[29:0];

// FSM for grabbing a frame

parameter S_WAIT_TRIG = 0;
parameter S_WAIT_SYNC = 1;
parameter S_WRITE_EVEN = 2;
parameter S_WRITE_ODD = 3;

reg [1:0] state;

    always @ (negedge tv_in_line_clock1) begin
        if(reset) state <= S_WAIT_TRIG;
        else case(state)
            S_WAIT_TRIG: state <=
                (fg_trig ? S_WAIT_SYNC : S_WAIT_TRIG);
            S_WAIT_SYNC: state <=
                (field_h2l ? S_WRITE_EVEN : S_WAIT_SYNC);
            S_WRITE_EVEN: state <=
                (real_vsync&fvh_in[2] ? S_WRITE_ODD : S_WRITE_EVEN);
            S_WRITE_ODD: state <=
                (real_vsync&~fvh_in[2] ? S_WAIT_TRIG : S_WRITE_ODD);
            default: state <= S_WAIT_TRIG;  //must have gotten here by
                                            //some horrible error, so reset
        endcase
    end

// wire up outputs to ZBT
```

```
    reg [35:0] vram_write_data;
    reg vram_we;
    reg [18:0] vram_addr;
    always @ (negedge tv_in_line_clock1) begin
        // if FSM is in a write state and we're on the screen (not blanking)
        // then write to memory
        vram_we <= (sample_count < 1440) & (line_count < 263)
                & ((state == S_WRITE_EVEN) | (state == S_WRITE_ODD));
        vram_addr <= (vram_we ? vram_write_addr : vram_read_addr);
        vram_write_data <= {6'b0, ycrcb_in};
    end
endmodule
```

A.3 GUI Interface Objects - gui_objects.v

```
// gui_objects.v
// module definitions for the user interface objects used in the
// control module.

// checkbox module
module checkbox(reset,mx,my,click,vcount,hcount,pixel,clock,en,vclock);
parameter WIDTH = 16;
parameter HEIGHT = 16;
parameter x = 0;
parameter y = 0;

input reset;
input [9:0] mx,my;
input click;
input [10:0] hcount;
input [9:0] vcount;
input clock;
output [2:0] pixel;
output en;
input vclock;

reg state;
assign en = state;

// synchronize the state used to calculate pixel to vclock

reg state_sync1, state_sync2;
always @ (posedge vclock) begin
    state_sync1 <= state;
    state_sync2 <= state_sync1;
end
```

```
reg lastclick;
always @ (posedge clock) lastclick <= click;
assign newclick = ~lastclick & click;

assign mouseover = (mx >= x) & (mx < WIDTH+x) & (my >= y) & (my <= y+HEIGHT);

always @ (posedge clock) begin
    if(reset) state <= 0;
    else if(newclick & mouseover) state <= ~state;
end

assign drawnow = (hcount >= x) & (hcount < WIDTH+x)
        & (vcount >= y) & (vcount < y+HEIGHT);
assign border = drawnow & ((hcount == x)
    || (hcount+1 == x+WIDTH) || (vcount == y) || (vcount+1 == y+HEIGHT));

assign pixel = (drawnow ? (border ? 7 : (state_sync2 ? 2 : 4)) : 0);
endmodule


// option module
module option(mx,my,click,vcount,hcount,pixel,clock,status,clicked,vclock);
parameter WIDTH = 16;
parameter HEIGHT = 16;
parameter x = 0;
parameter y = 0;

input [9:0] mx,my;
input click;
input [10:0] hcount;
input [9:0] vcount;
input clock;
input status;
output [2:0] pixel;
output clicked;
input vclock;

// synchronize the status used to calculate pixel to vclock

reg status_sync1, status_sync2;
always @ (posedge vclock) begin
    status_sync1 <= status;
    status_sync2 <= status_sync1;
end
```

```
reg lastclick;
always @ (posedge clock) lastclick <= click;
assign newclick = ~lastclick & click;

assign mouseover = (mx >= x) & (mx < WIDTH+x) & (my >= y) & (my <= y+HEIGHT);

assign clicked = newclick&mouseover;

assign drawnow = (hcount >= x) & (hcount < WIDTH+x)
            & (vcount >= y) & (vcount < y+HEIGHT);
assign border = drawnow & ((hcount == x)
    || (hcount+1 == x+WIDTH) || (vcount == y) || (vcount+1 == y+HEIGHT));

assign pixel = (drawnow ? (border ? 7 : (status_sync2 ? 2 : 4)) : 0);
endmodule

// button module
module button(mx,my,click,hcount,vcount,pixel,down,vclock);
parameter WIDTH = 100;
parameter HEIGHT = 16;
parameter COLOR = 7;
parameter FILLCOLOR = 0;
parameter INVERT_ON_CLICK = 1;
parameter x = 0;
parameter y = 0;

input [9:0] mx,my;
input click;
input [10:0] hcount;
input [9:0] vcount;
output [2:0] pixel;
output down;
input vclock;

// synchronize the down signal used to calculate pixel to vclock

reg down_sync1, down_sync2;
always @ (posedge vclock) begin
    down_sync1 <= down;
    down_sync2 <= down_sync1;
end

assign mouseover = (mx >= x) & (mx < WIDTH+x) & (my >= y) & (my < y+HEIGHT);
assign down = mouseover&click;
```

```
assign drawnow = (hcount >= x) & (hcount < WIDTH+x)
    & (vcount >= y) & (vcount < y+HEIGHT);
assign border = drawnow & ((hcount == x)
    || (hcount+1 == x+WIDTH) || (vcount == y) || (vcount+1 == y+HEIGHT));

assign pixel = (drawnow ? (border ? COLOR :
    (INVERT_ON_CLICK & down_sync2 ? ~FILLCOLOR : FILLCOLOR)) : 0);

endmodule
```

A.4 Overlay Text Pixel Generation Code - cstringdisp.v

```
// cstringdisp.v
//
// A modified version the original cstringdisp.v by I. Chuang and C. Terman.
//
// This module for rendering text into the overlay buffers, not for
// rendering on VGA screen. The cx and cy signals are removed from the
// module definition, since all text written into an overlay buffer is
// positioned at 0,0 within the buffer. This module is also modified to
// read from a register file instead of a wire. Pixel output is also a
// single bit, since the overlay buffer is 1bpp.

module overlay_font_render (vclock,hcount,vcount,pixel,ascii_in,buffer_addr);

   parameter NCHAR = 8; // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // number of bits in NCHAR

   input vclock;    // 65MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   output pixel;    // char display's pixel
   input [7:0] ascii_in;    // character string to display
   output [NCHAR_BITS-1:0] buffer_addr;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]  hoff = hcount-1;
   wire [9:0]   voff = vcount;
   wire [NCHAR_BITS-1:0] column = hoff[NCHAR_BITS-1+4:4];  // < NCHAR
   wire [2:0]   h = hoff[3:1];            // 0 .. 7
   wire [3:0]   v = voff[4:1];        // 0 .. 11

   // look up character to display (from character string)
   wire [7:0]  char;
```

```
   assign buffer_addr = column;
   assign char = ascii_in;

   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire cpixel = (font_byte[7 - h] ^ reverse) ? 1 : 0;
   wire dispflag = ((hcount > 0) & (vcount >= 0) & (hcount <= NCHAR*16)
          & (vcount < 24));
   wire pixel = dispflag ? cpixel : 0;

endmodule
```

A.5 VGA Text Pixel Generation Code - prefetch_cstring.v

```
// prefetch_cstring.v
//
// New versions of cstring display functions that prefetch data
// from the font ROM to eliminate latency problems.
// There are two modules, char_string_display, which is used for
// static text blocks, and read_array_text_display which is used to
// read out text stored in a register file.

// used for display of static text blocks on the VGA screen. Can
// be used as a drop-in replacement for the original char_string_display
// module by I. Chuang and C. Terman.

module char_string_display (vclock,hcount,vcount,pixel,cstring,
                                        cx,cy,font_byte,font_addr);

    parameter NCHAR = 8;    // number of 8-bit characters in cstring
    parameter NCHAR_BITS = 3; // number of bits in NCHAR

    parameter PREFETCH_LEN = 16;

    input vclock;   // 65MHz clock
    input [10:0] hcount;    // horizontal index of current pixel (0..1023)
    input [9:0]     vcount; // vertical index of current pixel (0..767)
    output [2:0] pixel; // char display's pixel
    input [NCHAR*8-1:0] cstring;    // character string to display
    input [10:0] cx;
    input [9:0]     cy;
```

```
input [7:0] font_byte;
output [10:0] font_addr;

wire dispflag;
wire fetchflag;

assign dispflag = ((hcount > cx) & (vcount >= cy)
                        & (hcount <= cx+NCHAR*8) & (vcount < cy + 12));

assign fetchflag =  ((hcount+PREFETCH_LEN > cx) & (vcount >= cy)
            & (hcount+PREFETCH_LEN <= cx+NCHAR*8) & (vcount < cy + 12));

wire [NCHAR_BITS-1:0] fetch_col;
wire [10:0] fetch_pos, real_hpos;
wire [9:0] real_vpos;

assign real_hpos = (hcount-cx);
assign real_vpos = (vcount-cy);
assign fetch_pos = (real_hpos+PREFETCH_LEN);
assign fetch_col = fetch_pos >> 3;

wire [2:0] h;
wire [3:0] v;

assign h = real_hpos[2:0]; // 0 to 7
assign v = real_vpos[3:0]; // 0 to 11

// look up character to display (from character string)
reg [7:0]  char;
integer  n;
always @(*) begin
        for (n=0 ; n<8 ; n = n+1)       // 8 bits per character (ASCII)
        char[n] <= cstring[(NCHAR-1-fetch_col)*8+n];
    end

wire [10:0] font_addr_comb;
assign font_addr_comb = char[6:0]*12 + v;

reg [10:0] font_addr;
reg [7:0] font_chunk;

always @ (posedge vclock) begin
    if(fetch_pos[2:0] == 3'b111) begin
        font_chunk <= font_byte;
        font_addr <= (fetchflag ? font_addr_comb : 0);
```

```
        end
    end

    wire [2:0] cpixel = (font_chunk[7 - h] ? 7 : 0);
    wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule


// used to render contents of keyboard buffer onto VGA display
module read_array_text_display (vclock,hcount,vcount,pixel,
                    ascii_in,buffer_addr,cx,cy,font_byte,font_addr);

    parameter NCHAR = 8;    // number of 8-bit characters in cstring
    parameter NCHAR_BITS = 3; // number of bits in NCHAR

    parameter PREFETCH_LEN = 16;

    input vclock;   // 65MHz clock
    input [10:0] hcount;    // horizontal index of current pixel (0..1023)
    input [9:0]     vcount; // vertical index of current pixel (0..767)
    output [2:0] pixel; // char display's pixel
    input [7:0] ascii_in;   // character string to display
    output [NCHAR_BITS-1:0] buffer_addr;
    input [10:0] cx;
    input [9:0] cy;
    input [7:0] font_byte;
    output [10:0] font_addr;

    wire dispflag;
    wire fetchflag;

    assign dispflag = ((hcount > cx) & (vcount >= cy)
                & (hcount <= cx+NCHAR*8) & (vcount < cy + 12));

    assign fetchflag =  ((hcount+PREFETCH_LEN > cx) & (vcount >= cy)
                & (hcount+PREFETCH_LEN <= cx+NCHAR*8) & (vcount < cy + 12));

    wire [NCHAR_BITS-1:0] fetch_col;
    wire [10:0] fetch_pos, real_hpos;
    wire [9:0] real_vpos;

    assign real_hpos = (hcount-cx);
    assign real_vpos = (vcount-cy);
    assign fetch_pos = (real_hpos+PREFETCH_LEN);
```

```
    assign fetch_col = fetch_pos >> 3;

    wire [2:0] h;
    wire [3:0] v;

    assign h = real_hpos[2:0]; // 0 to 7
    assign v = real_vpos[3:0]; // 0 to 11

    wire [7:0] char;
    assign char = ascii_in;
    assign buffer_addr = fetch_col;

    wire [10:0] font_addr_comb;
    assign font_addr_comb = char[6:0]*12 + v;

    reg [10:0] font_addr;
    reg [7:0] font_chunk;

    always @ (posedge vclock) begin
        if(fetch_pos[2:0] == 3'b111) begin
            font_chunk <= font_byte;
            font_addr <= (fetchflag ? font_addr_comb : 0);

        end
    end

    wire [2:0] cpixel = (font_chunk[7 - h] ? 7 : 0);
    wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

A.6 Control Module Code - control.v

```
// control.v
//
// Implements the control module, in all its glory.

module control(reset,clock_27mhz,hs,vs,rgb,b,mx,my,click,
            clock_65mhz,keyb_ascii, keyb_ready,
            framegrab_trig, framegrab_addr_select,
            bluescreen_en, bluescreen_cal,
            zoom_en, zoom_pos, zoom_mag,
            overlay_ready, overlay_en, overlay_pip, overlay_data,
            overlay_select, overlay_clear, overlay_we,
            surface_mouse_pos);
```

```
    input reset;
    input clock_27mhz;
    output hs;
    output vs;
    output [2:0] rgb;
    output b;
    input [9:0] mx;
    input [9:0] my;
    input clock_65mhz;
    input click;
    input [7:0] keyb_ascii;
    input keyb_ready;

    // control I/O to datapath modules
    output framegrab_trig;
    output framegrab_addr_select;
    output bluescreen_en;
    output bluescreen_cal;
    output zoom_en;
    output [19:0] zoom_pos;
    output [1:0] zoom_mag;
    input overlay_ready;
    output [3:0] overlay_en;
    output overlay_pip;
    output [20:0] overlay_data;
    output [2:0] overlay_select;
    output overlay_clear;
    output overlay_we;
    output [19:0] surface_mouse_pos;

//////////////////////////////////BEGIN GUI CODE/////////////////////////////////

    reg hs,vs,b;
    reg [2:0] rgb;

    wire [10:0] hcount;
    wire [9:0]  vcount;
    wire hsync, vsync, blank;

    xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

// button for framegrab
    wire [2:0] fg_trig_rgb;
    wire framegrab_trig;
```

```
    wire fg_trig_down;
    button fgtrig(mx,my,click,hcount,vcount,
            fg_trig_rgb,fg_trig_down,clock_65mhz);
    defparam fgtrig.x = 50;
    defparam fgtrig.y = 128;
    l2p l2p_1(reset, clock_27mhz, fg_trig_down, framegrab_trig);

// checkbox for bluescreen enable
    wire [2:0] bs_check_rgb;
    wire bluescreen_en;
    wire framegrab_addr_select;
    checkbox bsen(reset,mx,my,click,vcount,hcount,
            bs_check_rgb,clock_27mhz,bluescreen_en,clock_65mhz);
    defparam bsen.x = 50;
    defparam bsen.y = 256;

    assign framegrab_addr_select = bluescreen_en;

// button for bluescreen calibrate
    wire [2:0] bs_cal_rgb;
    wire bluescreen_cal_down;
    wire bluescreen_cal;
    button bscal(mx,my,click,hcount,vcount,
            bs_cal_rgb,bluescreen_cal_down,clock_65mhz);
    defparam bscal.x = 50;
    defparam bscal.y = 288;
    l2p l2p_2(reset, clock_27mhz, bluescreen_cal_down, bluescreen_cal);

// checkbox for zoom enable
    wire [2:0] zoom_check_rgb;
    wire zoom_en;
    checkbox zoomen(reset,mx,my,click,vcount,hcount,
            zoom_check_rgb,clock_27mhz,zoom_en,clock_65mhz);
    defparam zoomen.x = 50;
    defparam zoomen.y = 512;

// zoom depth option buttons
    reg [1:0] zoom_level;
    assign zoom_mag = zoom_level;

    wire x2_status,x3_status,x4_status;
    wire x2_clicked,x3_clicked,x4_clicked;
    wire [2:0] x2_rgb, x3_rgb, x4_rgb;

    assign x2_status = (zoom_level == 0);
```

```
    assign x3_status = (zoom_level == 1);
    assign x4_status = (zoom_level == 2);

    option x2(mx,my,click,vcount,hcount,
        x2_rgb,clock_27mhz,x2_status,x2_clicked,clock_65mhz);
    defparam x2.x = 50;
    defparam x2.y = 544;

    option x3(mx,my,click,vcount,hcount,
        x3_rgb,clock_27mhz,x3_status,x3_clicked,clock_65mhz);
    defparam x3.x = 50;
    defparam x3.y = 564;

    option x4(mx,my,click,vcount,hcount,
        x4_rgb,clock_27mhz,x4_status,x4_clicked,clock_65mhz);
    defparam x4.x = 50;
    defparam x4.y = 584;

    always @ (posedge clock_27mhz) begin
        if(reset) zoom_level <= 0;
        else if(x2_clicked) zoom_level <= 0;
        else if(x3_clicked) zoom_level <= 1;
        else if(x4_clicked) zoom_level <= 2;
    end

// button for zoom position
    wire [2:0] zoom_pos_rgb;
    wire zoom_pos_down;
    button zoompos(mx,my,click,hcount,vcount,
        zoom_pos_rgb,zoom_pos_down,clock_65mhz);
    defparam zoompos.x = 50;
    defparam zoompos.y = 616;

// checkboxes for enabling parts of overlay
    wire [3:0] overlay_en;
    wire [2:0] text1_check_rgb;
    checkbox text1en(reset,mx,my,click,vcount,hcount,
        text1_check_rgb,clock_27mhz,overlay_en[3],clock_65mhz);
    defparam text1en.x = 512;
    defparam text1en.y = 600;

    wire [2:0] text2_check_rgb;
    checkbox text2en(reset,mx,my,click,vcount,hcount,
        text2_check_rgb,clock_27mhz,overlay_en[2],clock_65mhz);
    defparam text2en.x = 512;
```

```
    defparam text2en.y = 632;

    wire overlay_fg_en;

    wire [2:0] fg_check_rgb;
    checkbox fgen(reset,mx,my,click,vcount,hcount,
        fg_check_rgb,clock_27mhz,overlay_fg_en,clock_65mhz);
    defparam fgen.x = 512;
    defparam fgen.y = 664;

    assign overlay_en[1] = (bluescreen_en ? 0 : overlay_fg_en);

    wire [2:0] trace_check_rgb;
    checkbox traceen(reset,mx,my,click,vcount,hcount,
        trace_check_rgb,clock_27mhz,overlay_en[0],clock_65mhz);
    defparam traceen.x = 512;
    defparam traceen.y = 696;

// checkbox for enabling PIP mode of framegrab overlay

    wire overlay_pip;
    wire [2:0] fg_pip_check_rgb;
    checkbox fgpip(reset,mx,my,click,vcount,hcount,
        fg_pip_check_rgb,clock_27mhz,overlay_pip,clock_65mhz);
    defparam fgpip.x = 768;
    defparam fgpip.y = 664;

// buttons for overlay positions
    wire [2:0] text1_pos_rgb;
    wire text1_pos_down;
    button text1pos(mx,my,click,hcount,vcount,
        text1_pos_rgb,text1_pos_down,clock_65mhz);
    defparam text1pos.x = 640;
    defparam text1pos.y = 600;

    wire [2:0] text2_pos_rgb;
    wire text2_pos_down;
    button text2pos(mx,my,click,hcount,vcount,
        text2_pos_rgb,text2_pos_down,clock_65mhz);
    defparam text2pos.x = 640;
    defparam text2pos.y = 632;

    wire [2:0] fg_pos_rgb;
    wire fg_pos_down;
    button fgpos(mx,my,click,hcount,vcount,
```

```
        fg_pos_rgb,fg_pos_down,clock_65mhz);
    defparam fgpos.x = 640;
    defparam fgpos.y = 664;

// text enter buttons
    wire [2:0] text1_enter_rgb;
    wire text1_enter_down;
    button text1enter(mx,my,click,hcount,vcount,
        text1_enter_rgb,text1_enter_down,clock_65mhz);
    defparam text1enter.x = 768;
    defparam text1enter.y = 600;

    wire [2:0] text2_enter_rgb;
    wire text2_enter_down;
    button text2enter(mx,my,click,hcount,vcount,
        text2_enter_rgb,text2_enter_down,clock_65mhz);
    defparam text2enter.x = 768;
    defparam text2enter.y = 632;

// trace clear button
    wire [2:0] trace_clear_rgb;
    wire trace_clear_down;
    wire overlay_clear;
    button trace_clear(mx,my,click,hcount,vcount,
        trace_clear_rgb,trace_clear_down,clock_65mhz);

    l2p l2p_3(reset, clock_27mhz, trace_clear_down, overlay_clear);
    defparam trace_clear.x = 640;
    defparam trace_clear.y = 696;

// trace draw button
    wire [2:0] trace_draw_rgb;
    wire trace_draw_down;
    button trace_draw(mx,my,click,hcount,vcount,
        trace_draw_rgb,trace_draw_down,clock_65mhz);
    defparam trace_draw.x = 768;
    defparam trace_draw.y = 696;

// drawing surface blob
    wire [2:0] surface;
    wire surface_mousedown;
    wire [9:0] surf_x, surf_y;

    parameter SURF_X = 304;
    parameter SURF_Y = 0;
```

```
    parameter SURF_WIDTH = 720;
    parameter SURF_HEIGHT = 525;

    button surf(mx,my,click,hcount,vcount,
        surface,surface_mousedown,clock_65mhz);
    defparam surf.WIDTH=SURF_WIDTH;
    defparam surf.HEIGHT=SURF_HEIGHT;
    defparam surf.COLOR=1;
    defparam surf.FILLCOLOR=1;
    defparam surf.INVERT_ON_CLICK=0;
    defparam surf.x = SURF_X;
    defparam surf.y = SURF_Y;

    assign surf_x = ((mx < SURF_X)|(mx > SURF_X + SURF_WIDTH) ? 0 : mx - SURF_X);
    assign surf_y = ((my > SURF_HEIGHT + SURF_Y)|(my < SURF_Y) ? 0 : my - SURF_Y);
    assign surface_mouse_pos = {surf_y, surf_x};

// enable text blocks
    wire [7:0]  font_byte;
    wire [10:0] font_addr;

    wire [63:0] enable_cstring = "ENABLE";

    wire [10:0] bs_en_text_addr;
    wire [2:0]  bs_en_text_rgb;
    char_string_display bs_en_text(clock_65mhz,hcount,vcount,bs_en_text_rgb,
        enable_cstring,11'd74,10'd258,font_byte,bs_en_text_addr);
    defparam bs_en_text.NCHAR = 6;
    defparam bs_en_text.NCHAR_BITS = 3;

    wire [10:0] zoom_en_text_addr;
    wire [2:0]  zoom_en_text_rgb;
    char_string_display zoom_en_text(clock_65mhz,hcount,vcount,zoom_en_text_rgb,
        enable_cstring,11'd74,10'd514,font_byte,zoom_en_text_addr);
    defparam zoom_en_text.NCHAR = 6;
    defparam zoom_en_text.NCHAR_BITS = 3;

    wire [10:0] text1_en_text_addr;
    wire [2:0]  text1_en_text_rgb;
    char_string_display text1_en_text(clock_65mhz,hcount,vcount,text1_en_text_rgb,
        enable_cstring,11'd536,10'd602,font_byte,text1_en_text_addr);
    defparam text1_en_text.NCHAR = 6;
    defparam text1_en_text.NCHAR_BITS = 3;

    wire [10:0] text2_en_text_addr;
```

```
    wire [2:0]  text2_en_text_rgb;
    char_string_display text2_en_text(clock_65mhz,hcount,vcount,text2_en_text_rgb,
        enable_cstring,11'd536,10'd634,font_byte,text2_en_text_addr);
    defparam text2_en_text.NCHAR = 6;
    defparam text2_en_text.NCHAR_BITS = 3;

    wire [10:0] fg_en_text_addr;
    wire [2:0]  fg_en_text_rgb;
    char_string_display fg_en_text(clock_65mhz,hcount,vcount,fg_en_text_rgb,
        enable_cstring,11'd536,10'd666,font_byte,fg_en_text_addr);
    defparam fg_en_text.NCHAR = 6;
    defparam fg_en_text.NCHAR_BITS = 3;

    wire [10:0] trace_en_text_addr;
    wire [2:0]  trace_en_text_rgb;
    char_string_display trace_en_text(clock_65mhz,hcount,vcount,trace_en_text_rgb,
        enable_cstring,11'd536,10'd698,font_byte,trace_en_text_addr);
    defparam trace_en_text.NCHAR = 6;
    defparam trace_en_text.NCHAR_BITS = 3;

// text block for PIP button
    wire [23:0] pip_cstring = "PIP";
    wire [10:0] pip_en_text_addr;
    wire [2:0]  pip_en_text_rgb;
    char_string_display pip_en_text(clock_65mhz,hcount,vcount,pip_en_text_rgb,
        pip_cstring,11'd792,10'd666,font_byte,pip_en_text_addr);
    defparam pip_en_text.NCHAR = 3;
    defparam pip_en_text.NCHAR_BITS = 2;

// title text blocks
    wire [95:0] fg_cstring = "FRAMEGRABBER";

    wire [10:0] fg_text_addr;
    wire [2:0]  fg_text_rgb;
    char_string_display fg_text(clock_65mhz,hcount,vcount,fg_text_rgb,
        fg_cstring,11'd50,10'd112,font_byte,fg_text_addr);
    defparam fg_text.NCHAR = 12;
    defparam fg_text.NCHAR_BITS = 4;

    wire [79:0] bs_cstring = "BLUESCREEN";

    wire [10:0] bs_text_addr;
    wire [2:0]  bs_text_rgb;
    char_string_display bs_text(clock_65mhz,hcount,vcount,bs_text_rgb,
        bs_cstring,11'd50,10'd240,font_byte,bs_text_addr);
```

```
    defparam bs_text.NCHAR = 10;
    defparam bs_text.NCHAR_BITS = 4;

    wire [31:0] zoom_cstring = "ZOOM";

    wire [10:0] zoom_text_addr;
    wire [2:0]  zoom_text_rgb;
    char_string_display zoom_text(clock_65mhz,hcount,vcount,zoom_text_rgb,
        zoom_cstring,11'd50,10'd496,font_byte,zoom_text_addr);
    defparam zoom_text.NCHAR = 4;
    defparam zoom_text.NCHAR_BITS = 2;

    wire [63:0] overlay_cstring = "OVERLAY";

    wire [10:0] overlay_text_addr;
    wire [2:0]  overlay_text_rgb;
    char_string_display overlay_text(clock_65mhz,hcount,vcount,overlay_text_rgb,
        overlay_cstring,11'd640,10'd568,font_byte,overlay_text_addr);
    defparam overlay_text.NCHAR = 7;
    defparam overlay_text.NCHAR_BITS = 3;

// zoom level option labels
    wire [15:0] x2_zoom_cstring = "2x";

    wire [10:0] x2_zoom_text_addr;
    wire [2:0]  x2_zoom_text_rgb;
    char_string_display x2_zoom_text(clock_65mhz,hcount,vcount,x2_zoom_text_rgb,
        x2_zoom_cstring,11'd74,10'd546,font_byte,x2_zoom_text_addr);
    defparam x2_zoom_text.NCHAR = 2;
    defparam x2_zoom_text.NCHAR_BITS = 1;

    wire [15:0] x3_zoom_cstring = "3x";

    wire [10:0] x3_zoom_text_addr;
    wire [2:0]  x3_zoom_text_rgb;
    char_string_display x3_zoom_text(clock_65mhz,hcount,vcount,x3_zoom_text_rgb,
        x3_zoom_cstring,11'd74,10'd566,font_byte,x3_zoom_text_addr);
    defparam x3_zoom_text.NCHAR = 2;
    defparam x3_zoom_text.NCHAR_BITS = 1;

    wire [15:0] x4_zoom_cstring = "4x";

    wire [10:0] x4_zoom_text_addr;
    wire [2:0]  x4_zoom_text_rgb;
    char_string_display x4_zoom_text(clock_65mhz,hcount,vcount,x4_zoom_text_rgb,
```

```
        x4_zoom_cstring,11'd74,10'd586,font_byte,x4_zoom_text_addr);
    defparam x4_zoom_text.NCHAR = 2;
    defparam x4_zoom_text.NCHAR_BITS = 1;

//button text

    wire [95:0] bs_button_text_cstring = "CALIBRATE";

    wire [10:0] bs_button_text_addr;
    wire [2:0]  bs_button_text_rgb;
    char_string_display bs_button_text(clock_65mhz,hcount,vcount,bs_button_text_rgb,
        bs_button_text_cstring,11'd52,10'd290,font_byte,bs_button_text_addr);
    defparam bs_button_text.NCHAR = 9;
    defparam bs_button_text.NCHAR_BITS = 4;

    wire [95:0] fg_button_text_cstring = "TRIGGER";

    wire [10:0] fg_button_text_addr;
    wire [2:0]  fg_button_text_rgb;
    char_string_display fg_button_text(clock_65mhz,hcount,vcount,fg_button_text_rgb,
        fg_button_text_cstring,11'd52,10'd130,font_byte,fg_button_text_addr);
    defparam fg_button_text.NCHAR = 7;
    defparam fg_button_text.NCHAR_BITS = 3;

    wire [143:0] set_pos_cstring = "SET POSITION";

    wire [10:0] zoom_center_button_text_addr;
    wire [2:0]  zoom_center_button_text_rgb;
    char_string_display zoom_center_button_text(clock_65mhz,hcount,vcount,
        zoom_center_button_text_rgb,set_pos_cstring,11'd51,10'd618,font_byte,
    zoom_center_button_text_addr);
    defparam zoom_center_button_text.NCHAR = 12;
    defparam zoom_center_button_text.NCHAR_BITS = 4;

    wire [10:0] text1_pos_button_text_addr;
    wire [2:0]  text1_pos_button_text_rgb;
    char_string_display text1_pos_button_text(clock_65mhz,hcount,vcount,
        text1_pos_button_text_rgb,set_pos_cstring,11'd641,10'd602,
    font_byte,text1_pos_button_text_addr);
    defparam text1_pos_button_text.NCHAR = 12;
    defparam text1_pos_button_text.NCHAR_BITS = 4;

    wire [10:0] text2_pos_button_text_addr;
    wire [2:0]  text2_pos_button_text_rgb;
    char_string_display text2_pos_button_text(clock_65mhz,hcount,vcount,
```

```
    text2_pos_button_text_rgb,set_pos_cstring,11'd641,10'd634,
font_byte,text2_pos_button_text_addr);
defparam text2_pos_button_text.NCHAR = 12;
defparam text2_pos_button_text.NCHAR_BITS = 4;

wire [10:0] fg_pos_button_text_addr;
wire [2:0]  fg_pos_button_text_rgb;
char_string_display fg_pos_button_text(clock_65mhz,hcount,vcount,fg_pos_button_text_rgb,
    set_pos_cstring,11'd641,10'd666,font_byte,fg_pos_button_text_addr);
defparam fg_pos_button_text.NCHAR = 12;
defparam fg_pos_button_text.NCHAR_BITS = 4;

wire [31:0] draw_cstring = "DRAW";

wire [10:0] draw_button_text_addr;
wire [2:0]  draw_button_text_rgb;
char_string_display draw_button_text(clock_65mhz,hcount,vcount,draw_button_text_rgb,
    draw_cstring,11'd770,10'd698,font_byte,draw_button_text_addr);
defparam draw_button_text.NCHAR = 4;
defparam draw_button_text.NCHAR_BITS = 2;

wire [79:0] enter_cstring = "ENTER TEXT";

wire [10:0] text1_enter_button_text_addr;
wire [2:0]  text1_enter_button_text_rgb;
char_string_display text1_enter_button_text(clock_65mhz,hcount,vcount,
    text1_enter_button_text_rgb,enter_cstring,11'd770,10'd602,font_byte,
text1_enter_button_text_addr);
defparam text1_enter_button_text.NCHAR = 10;
defparam text1_enter_button_text.NCHAR_BITS = 4;

wire [10:0] text2_enter_button_text_addr;
wire [2:0]  text2_enter_button_text_rgb;
char_string_display text2_enter_button_text(clock_65mhz,hcount,vcount,
    text2_enter_button_text_rgb,enter_cstring,11'd770,10'd634,font_byte,
text2_enter_button_text_addr);
defparam text2_enter_button_text.NCHAR = 10;
defparam text2_enter_button_text.NCHAR_BITS = 4;

wire [39:0] clear_cstring = "CLEAR";

wire [10:0] trace_clear_button_text_addr;
wire [2:0]  trace_clear_button_text_rgb;
char_string_display trace_clear_button_text(clock_65mhz,hcount,vcount,
    trace_clear_button_text_rgb,clear_cstring,11'd642,10'd698,font_byte,
```

```
    trace_clear_button_text_addr);
    defparam trace_clear_button_text.NCHAR = 5;
    defparam trace_clear_button_text.NCHAR_BITS = 3;

// render keyboard buffer contents
parameter KEYB_BUF_LEN = 48;
parameter KEYB_BUF_LEN_BITS = 6;
reg [7:0] keyb_buffer[KEYB_BUF_LEN-1:0];
reg [KEYB_BUF_LEN_BITS-1:0] keyb_buffer_count;
wire [KEYB_BUF_LEN_BITS-1:0] keyb_buffer_addr;

wire [10:0] keyb_text_addr;
wire [2:0] keyb_text_rgb;

// this will replace any characters beyond keyb_buffer_count with spaces
wire [7:0] keyb_buffer_blanked;

    //synchronizer for ascii values in keyb_buffer
    //(synchronized to 27mhz tv line clock)
    reg [7:0] ascii_sync1, ascii_sync2;
    reg [KEYB_BUF_LEN_BITS-1:0] keyb_buffer_count_sync1, keyb_buffer_count_sync2;


    always @ (posedge clock_65mhz) begin
        ascii_sync1 <= keyb_buffer[keyb_buffer_addr];
        ascii_sync2 <= ascii_sync1;
        keyb_buffer_count_sync1 <= keyb_buffer_count;
        keyb_buffer_count_sync2 <= keyb_buffer_count_sync1;
    end

assign keyb_buffer_blanked = (keyb_buffer_addr < keyb_buffer_count_sync2 ? ascii_sync2 : 32);

read_array_text_display keyb_text(clock_65mhz,hcount,vcount,keyb_text_rgb,
        keyb_buffer_blanked,keyb_buffer_addr,11'd256,10'd730,font_byte,keyb_text_addr);
defparam keyb_text.NCHAR = KEYB_BUF_LEN;
defparam keyb_text.NCHAR_BITS = KEYB_BUF_LEN_BITS;

// font rom address

font_rom f(font_addr,clock_65mhz,font_byte);
     reg [10:0] font_addr_1, font_addr_2;
     assign font_addr = font_addr_1 | font_addr_2 | keyb_text_addr;

     always @ (posedge clock_65mhz) begin
        font_addr_1 <= bs_en_text_addr | zoom_en_text_addr | text2_en_text_addr
```

```
        | fg_en_text_addr | trace_en_text_addr | zoom_text_addr
        | zoom_center_button_text_addr | fg_button_text_addr
        | text2_enter_button_text_addr | trace_clear_button_text_addr
        | text1_enter_button_text_addr | text1_pos_button_text_addr;

        font_addr_2 <= draw_button_text_addr | text2_pos_button_text_addr |
        fg_pos_button_text_addr | bs_button_text_addr | text1_en_text_addr |
        overlay_text_addr | x2_zoom_text_addr | x3_zoom_text_addr |
        x4_zoom_text_addr | fg_text_addr | bs_text_addr | pip_en_text_addr;
    end

// rendering
// pipeline used to or together sprite outputs across two cycles
// to combat effects of latency

    reg cursor, border;
    reg [9:0] mx_retrace;
    reg [9:0] my_retrace;
    wire [2:0] video;

    reg [2:0] video_1, video_2;
    reg hsync_delay, vsync_delay, blank_delay;

    assign video = ((cursor|border) ? 7 : (video_1 | video_2));

    always @(posedge clock_65mhz) begin
        // sync mouse movement to vertical retrace
        mx_retrace <= (vsync ? mx : mx_retrace);
        my_retrace <= (vsync ? my : my_retrace);

        cursor <= ((hcount==mx_retrace | vcount==my_retrace) ? 1 : 0);
        border <= ((hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 1 : 0);

        video_1 <= bs_check_rgb | zoom_check_rgb | text1_check_rgb
                | text2_check_rgb | fg_check_rgb | trace_check_rgb | surface |
                fg_trig_rgb | text1_pos_rgb | text2_pos_rgb | fg_pos_rgb |
                text1_enter_rgb | text2_enter_rgb | trace_clear_rgb | trace_draw_rgb |
                zoom_pos_rgb | x2_rgb | x3_rgb | x4_rgb | fg_pip_check_rgb |
                pip_en_text_rgb;

        video_2 <= bs_en_text_rgb | zoom_en_text_rgb | text1_en_text_rgb |
            text2_en_text_rgb | fg_en_text_rgb | trace_en_text_rgb |
            fg_text_rgb | bs_text_rgb | zoom_text_rgb | overlay_text_rgb |
            x2_zoom_text_rgb | x3_zoom_text_rgb | x4_zoom_text_rgb |
            bs_button_text_rgb | bs_cal_rgb | fg_button_text_rgb |
```

```
            zoom_center_button_text_rgb | text1_pos_button_text_rgb |
            text2_pos_button_text_rgb | fg_pos_button_text_rgb |
            draw_button_text_rgb | text1_enter_button_text_rgb |
            text2_enter_button_text_rgb | trace_clear_button_text_rgb |
            keyb_text_rgb;

     hsync_delay <= hsync;
     vsync_delay <= vsync;
     blank_delay <= blank;

     hs <= hsync_delay;
     vs <= vsync_delay;
     b <= blank_delay;
     rgb <= video;
    end

//////////////////////////////////END OF GUI CODE////////////////////////////////

//active text buffer select
//determines which buffer in overlay should be fed keyboard input
wire [7:0] keyb_ascii;
wire keyb_ready;
wire text_load_done;
wire text_load_start;
wire [19:0] text_load_addr;
wire text_load_we;
wire text_load_pixel;
wire [KEYB_BUF_LEN_BITS-1:0] text_load_buffer_addr;

parameter TEXT_BUFFER_OFF = 0;
parameter TEXT1_ACTIVE = 1;
parameter TEXT2_ACTIVE = 2;

reg [1:0] active_text_buffer;

always @ (posedge clock_27mhz) begin
    if(reset|text_load_done) active_text_buffer <= TEXT_BUFFER_OFF;
    else if(text1_enter_down) active_text_buffer <= TEXT1_ACTIVE;
    else if(text2_enter_down) active_text_buffer <= TEXT2_ACTIVE;
end

// code to render contents of keyboard buffer into overlay memory
assign text_load_start = (keyb_ascii==13)&keyb_ready;

font_load_fsm flf(reset, clock_27mhz, text_load_start, overlay_ready, text_load_done,
```

```
                text_load_addr, text_load_we);

//blank out any chars beyond the end of the buffer counter
wire [7:0] text_load_buffer_blanked;
assign text_load_buffer_blanked = (text_load_buffer_addr < keyb_buffer_count ?
                    keyb_buffer[text_load_buffer_addr] : 32);

overlay_font_render ovf(clock_27mhz,text_load_addr[9:0],text_load_addr[19:10],
                    text_load_pixel,text_load_buffer_blanked,text_load_buffer_addr);
defparam ovf.NCHAR = KEYB_BUF_LEN;
defparam ovf.NCHAR_BITS = KEYB_BUF_LEN_BITS;

//accepting keyboard input
always @ (posedge clock_27mhz) begin
    if (reset||text_load_done) keyb_buffer_count <=0;
    else if (keyb_ready
            & (active_text_buffer==TEXT1_ACTIVE
                    | active_text_buffer==TEXT2_ACTIVE)) begin
        if(keyb_ascii==8) begin //backspace
            keyb_buffer[keyb_buffer_count-1] <= 0;
            keyb_buffer_count <= (keyb_buffer_count==0 ? 0 : keyb_buffer_count - 1);
        end
        else if (keyb_ascii==13) begin //enter
            //do nothing, we don't want to put enter in the buffer
            keyb_buffer[keyb_buffer_count-1] <= keyb_buffer[keyb_buffer_count-1];
        end
        else if ((keyb_buffer_count < KEYB_BUF_LEN-1)&~text_load_we) begin
            keyb_buffer[keyb_buffer_count] <= keyb_ascii;
            keyb_buffer_count <= keyb_buffer_count + 1;
        end
    end
end

//drawing surface state
parameter SURF_OFF = 0;
parameter SURF_ZOOM = 1;
parameter SURF_TRACE = 2;
parameter SURF_TEXT1_POS = 3;
parameter SURF_TEXT2_POS = 4;
parameter SURF_FG_POS = 5;

reg [2:0] surf_state;

always @ (posedge clock_27mhz) begin
    if(reset) surf_state <= SURF_OFF;
```

```
    else if(zoom_pos_down) surf_state <= SURF_ZOOM;
    else if(text1_pos_down) surf_state <= SURF_TEXT1_POS;
    else if(text2_pos_down) surf_state <= SURF_TEXT2_POS;
    else if(fg_pos_down) surf_state <= SURF_FG_POS;
    else if(trace_draw_down) surf_state <= SURF_TRACE;
end

// zoom center point select logic
reg [19:0] zoom_pos;

always @ (posedge clock_27mhz) begin
    if(reset) zoom_pos <= 0;
    else if(surface_mousedown&surf_state==SURF_ZOOM) zoom_pos <= {surf_y, surf_x};
end

//overlay write logic
parameter OVERLAY_TEXT1_BUF = 3'b000;
parameter OVERLAY_TEXT2_BUF = 3'b010;
parameter OVERLAY_TRACE_BUF = 3'b110;
parameter OVERLAY_TEXT1_POS = 3'b001;
parameter OVERLAY_TEXT2_POS = 3'b011;
parameter OVERLAY_FG_POS = 3'b100;

reg [2:0] overlay_select;
wire overlay_we;
wire [20:0] overlay_data;

always @ (surf_state or text_load_we) begin //COMBINATIONAL always block
    if(text_load_we) overlay_select = (active_text_buffer == TEXT1_ACTIVE ?
                                    OVERLAY_TEXT1_BUF : OVERLAY_TEXT2_BUF);
    else    case (surf_state)
        SURF_TRACE: overlay_select = OVERLAY_TRACE_BUF;
        SURF_TEXT1_POS: overlay_select = OVERLAY_TEXT1_POS;
        SURF_TEXT2_POS: overlay_select = OVERLAY_TEXT2_POS;
        SURF_FG_POS: overlay_select = OVERLAY_FG_POS;
        default: overlay_select = 3'bxxx; //it really, really doesn't matter
    endcase
end

assign overlay_data[19:0] = (text_load_we ? text_load_addr : {surf_y, surf_x});
assign overlay_data[20] = (text_load_we ? text_load_pixel : 1);
assign overlay_we = (overlay_ready&(
                            (overlay_select == OVERLAY_TRACE_BUF)|
                            (overlay_select == OVERLAY_TEXT1_POS)|
                            (overlay_select == OVERLAY_TEXT2_POS)|
```

```
                             (overlay_select == OVERLAY_FG_POS))
                             & surface_mousedown) | text_load_we;
endmodule

// simple level to pulse converter

module l2p (reset, clk, in, out);
input reset;
input clk;
input in;
output out;

reg last;
wire out;

assign out = ~last & in; //positive transition of in

always @ (posedge clk) begin
    if(reset) last <= 0;
    else last <= in;
end

endmodule

// fsm to oversee font rendering into overlay buffer

module font_load_fsm (reset, clk, begin_load, ready, done, addr, we);
input reset;
input clk;
input begin_load;
input ready;
output done;
output [19:0] addr;
output we;

reg [1:0] state;
wire count_done;
wire we;
wire done;
reg [9:0] row, col;

parameter S_WAIT_START = 0;
parameter S_WAIT_READY = 1;
parameter S_COUNT = 2;
parameter S_DONE = 3;
```

```
always @ (posedge clk) begin
    if(reset) state <= S_WAIT_START;
    else case (state)
        S_WAIT_START: state <= (begin_load ? S_WAIT_READY : S_WAIT_START);
        S_WAIT_READY: state <= (ready ? S_COUNT : S_WAIT_READY);
        S_COUNT: state <= (count_done ? S_DONE : S_COUNT);
        S_DONE: state <= S_WAIT_START;
    endcase
end

assign done = (state == S_DONE);
assign we = (state == S_COUNT);
assign addr = {row, col};
assign count_done = (col == 719)&(row == 23);

// position counter which is incremented when FSM is in the S_COUNT state

always @ (posedge clk) begin
    if(state == S_WAIT_READY) begin
        row <= 0;
        col <= 0;
    end
    else if (state == S_COUNT) begin
        if (col == 719) begin
            col <= 0;
            row <= row + 1;
        end
        else col <= col + 1;
    end
end
endmodule
```

A.7 Zoom Module - zoom.v

```
//Zoom Module (zoom.v)

module zoom (reset, tv_in_line_clock1, ycrcb_in, ycrcb_out, fvh_in, fvh_out,
        vram_read_data, vram_write_data, vram_addr, vram_we,
        zoom_enable, zoom_pos, zoom_mag);

input reset;            //system reset
input tv_in_line_clock1;    //tv line-locked clock
input [29:0] ycrcb_in;      //pixel data coming in from bluescreen module
input [2:0] fvh_in;     //field, v_sync, and h_sync from bluescreen module
```

```
input zoom_enable;      //enables zoom feature (active HIGH) - otherwise just pass
                //ycrcb_in through unmodified
input [19:0] zoom_pos;      //centerpoint to zoom around
                //bits [19:10] are the row
                //bits [9:0] ar the column

reg [19:0] real_zoom_pos;   //real_zoom_pos moves zoom_pos to the closest valid value
                //a valid value cannot be too close to the edges of the frame
                //since we need to be able to read enough values in a window
                //around the centerpoint


input [1:0] zoom_mag;       //Scale factor - 0 means zoom by 2x
                //               1 means zoom by 3x
                //               2 means zoom by 4x
                //               3 should never be used

output [29:0] ycrcb_out;    //updated pixel value after possible applying a zoom
output [2:0] fvh_out;       //updated field, v_sync, and h_sync
reg [29:0] ycrcb_out;
reg [2:0] fvh_out;


output [35:0] vram_write_data;  //write data for ZBT
reg [35:0] vram_write_data=0;
input [35:0] vram_read_data;    //read data from ZBT
output [18:0] vram_addr;    //address to ZBT
reg [18:0] vram_addr=0;
output vram_we;         //write-enable for ZBT
reg vram_we=0;


reg write_offset=0;     //register for double buffering; see comments later


reg old_field;  //value of field at previous clock cycle
always @ (negedge tv_in_line_clock1) begin
    old_field <= fvh_in[2];
    if (~fvh_in[2] && old_field) begin
        //update zoom position only on a negative edge of the field
        if (zoom_mag == 0) begin
            //calculate real_zoom_pos based on a scale factor of 2
            real_zoom_pos[19:10] <= (zoom_pos[19:10] > 10'd119
                        && zoom_pos[19:10] < 10'd360) ? zoom_pos[19:10]
                        : (zoom_pos[19:10] <= 10'd119 ? 10'd120 : 10'd359);
```

```
            real_zoom_pos[9:0] <= (zoom_pos[9:0] > 10'd179 && zoom_pos[9:0] < 10'd540) ?
                     zoom_pos[9:0] : (zoom_pos[9:0] <= 10'd179 ? 10'd180 : 10'd539);

        end
        else if (zoom_mag == 1) begin
            //calculate real_zoom_pos based on a scale factor of 3
            real_zoom_pos[19:10] <= (zoom_pos[19:10] > 10'd79
                        && zoom_pos[19:10] < 10'd400) ?
                        zoom_pos[19:10] : (zoom_pos[19:10] <= 10'd79 ?
                         10'd80 : 10'd399);
            real_zoom_pos[9:0] <= (zoom_pos[9:0] > 10'd119
                        && zoom_pos[9:0] < 10'd600) ? zoom_pos[9:0]
                        : (zoom_pos[9:0] <= 10'd119 ? 10'd120 : 10'd600);
        end
        else if (zoom_mag == 2) begin
            //calculate real_zoom_pos based on a scale factor of 4
            real_zoom_pos[19:10] <= (zoom_pos[19:10] > 10'd59
                        && zoom_pos[19:10] < 10'd420) ?
                        zoom_pos[19:10] :
                        (zoom_pos[19:10] <= 10'd59 ? 10'd60 : 10'd419);
            real_zoom_pos[9:0] <= (zoom_pos[9:0] > 10'd89
                        && zoom_pos[9:0] < 10'd630) ? zoom_pos[9:0]
                        : (zoom_pos[9:0] <= 10'd89 ? 10'd90 : 10'd629);
        end
        else real_zoom_pos <= 0;
    end
    else real_zoom_pos <= real_zoom_pos;
end



//just pass along field, v_sync, and h_sync
//technically we should  delay these by the right amount to avoid a horizontal shift,
//but practically there isn't much benefit to doing this
always @ (negedge tv_in_line_clock1) begin
    fvh_out[2:0] <= fvh_in[2:0];
end



//Generate vertical and horizontal position counters
//Note: this code is not very clean. A cleaner version of code to generate these counters
//can be found in the camerae or bluescreen modules. We believe that the cleaner code
//can be used with the zoom, module, but we did not change the code since we have not
//tested the module with the new code.
```

```
reg [10:0] sample_count;    //these are intermediate counters used to derive the
reg [9:0] line_count;       //horizontal and vertical position

reg [9:0] h_position, v_position;   //horizontal position and vertical position
reg [29:0] zoom_value;          //zoom pixel value computed by combinational logic
                    //We latch zoom_value to produce ycrcb_out

   always @(negedge tv_in_line_clock1)
     if (reset)
       begin
      sample_count <= 0;
      line_count <= 0;
       end
     else
       begin
      if (sample_count < 1440)
        ycrcb_out <= zoom_value;    //latch zoom_value

      if (sample_count == 1715)
        //this is the end of the current line
        begin
        //reset sample_count and increment line_count
           sample_count <= 0;
           if (line_count == 524)
         line_count <= 0;
           else
         line_count <= line_count+1;
        end
          else if (fvh_in[1]) begin
        //if v_sync is high, we are in the blanking period between fields
        sample_count<=0;
        line_count<=0;
        end
          else
        //in this case, increment sample_count
        sample_count <= sample_count+1;
       end

    //compute position based on sample_count, line_count
    //Note: this can probably be combinational, but we did not test this
    always @ (negedge tv_in_line_clock1)
     if (line_count < 19)
       begin
      h_position <= 0;
```

```
      v_position <= 0;
       end
     else if (line_count < 262)
       begin
      h_position <= (sample_count[10:1] < 719) ?
               sample_count[10:1]+sample_count[0] : 0;
      v_position <= {line_count-19, fvh_in[2]};
       end

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


reg old_field_2;    //this is actually the same as old_field defined above

//write_offset is used as part of the double-buffering scheme for zooming in
//write_offset controls which buffer is the read buffer and which buffer is the write
//buffer. This is not really necessary for our new design since we zoom in on a field
//instead of an entire frame. However, we did not test the code without write_offset,
//so we kept it in. Also, the logic for the read and write addresses given later in
//the code still uses the write_offset when calculating the addresses, even though
//this isn't really necessary anymore.

always @ (negedge tv_in_line_clock1) begin
    old_field_2 <= fvh_in[2];
    //on a rising edge of the field, flip write_offset
    //write_offset flips every frame
    if (fvh_in[2] && ~old_field_2) write_offset <= ~write_offset;
end




//Setup read and write for ZBT RAM
reg [29:0] stored_read_data;        //data from ZBT
reg [29:0] write_data;          //data to write to ZBT
reg [18:0] read_addr, write_addr;   //read and write addresses
reg we;                 //ZBT write-enable


//Our strategy for using the ZBT is to read every other cycle and write every other cycle
//this works because the pixel value only actually changes every two clock cycles
//see video_encoder.v for more details

//latch ZBT read data every other clock cycle
```

```
always @ (negedge tv_in_line_clock1) begin
    stored_read_data <= sample_count[0] ? vram_read_data[29:0] : stored_read_data;
end

always @ (sample_count[0]) begin
    if (sample_count[0] == 1) begin
        //read from ZBT RAM
        vram_we = 0;
        vram_addr = read_addr;      //read_addr is calculated later in the code
        end
    else begin
        //write to ZBT RAM
        //we, write_data, and write_addr are calculated later in the code
        vram_we = we;
        vram_write_data = {6'b0,write_data};
        vram_addr = write_addr;
    end
end




//Calculate stuff for ZBT write cycles
always @ (h_position or v_position or zoom_mag or sample_count[0]) begin
    if (zoom_mag==0) begin
    //zoom in by 2x
        if (h_position >= real_zoom_pos[9:0]-180 && h_position <real_zoom_pos[9:0]+180
            && v_position >= real_zoom_pos[19:10]-120
            && v_position < real_zoom_pos[19:10]+120) begin
            //write data the ZBT if we are in a window around real_zoom_pos
            //addressing for ZBT is calculated by row*width + column, where row and
            //column are relative to the top left corner of the zoom window, and
            //width is a function of the scale factor
            we = 1;
            write_addr = (v_position+120-real_zoom_pos[19:10])*360
                     +(h_position-real_zoom_pos[9:0]+180)
                     +(write_offset ? 0 : 250*360);
            write_data = ycrcb_in;
        end
        else begin
            we = 1;     //if it is important, we should be able to de-assert
                    //write-enable in this case, but in our tests this
                    //caused problems, so we always write
            write_addr = 180000;    //this is an unused memory location
```

```
            write_data = 0;
        end
    end

    else if (zoom_mag ==1) begin
    //zoom in by 3x
        if (h_position >= real_zoom_pos[9:0]-120 && h_position <real_zoom_pos[9:0]+120
            && v_position >= real_zoom_pos[19:10]-80
            && v_position < real_zoom_pos[19:10]+80) begin
            we = 1;
            write_addr = (v_position+80-real_zoom_pos[19:10])*240
                     +(h_position-real_zoom_pos[9:0]+120)
                     +(write_offset ? 0 : 250*360);
            write_data = ycrcb_in;
        end
        else begin
            we = 1;
            write_addr = 180000;
            write_data = 0;
        end
    end

    else if (zoom_mag==2) begin
    //zoom in by 4x
        if (h_position >= real_zoom_pos[9:0]-90 && h_position <real_zoom_pos[9:0]+90
            && v_position >= real_zoom_pos[19:10]-60
            && v_position < real_zoom_pos[19:10]+60) begin
            we = 1;
            write_addr = (v_position+60-real_zoom_pos[19:10])*180
                     +(h_position-real_zoom_pos[9:0]+90)
                     +(write_offset ? 0 : 250*360);
            write_data = ycrcb_in;
        end
        else begin
            we = 1;
            write_addr = 180000;
            write_data = 0;
        end
    end
    else we = 0;
end
```

```
//Divide-by-three lookup tables
//Note: this implementation uses RAMs because we did not make a .coe file to initialize the
//memory. If we made a .coe file, it makes more sense to use a ROM instead
//We use two memories so that v_position and h_position can both be divided at the same time
//We could also just use one dual-port memory


//v_position divide-by-three memory
reg [9 : 0] vbram_addr;     //address
reg [7 : 0] vbram_din;      //write data
wire [7 : 0] vbram_dout;    //read data
reg vbram_we;           //write-enable
ram_div_by_3  ram1(vbram_addr,~tv_in_line_clock1,vbram_din,vbram_dout,vbram_we);


//h_position divide-by-three memory
reg [9 : 0] hbram_addr;     //address
reg [7 : 0] hbram_din;      //write data
wire [7 : 0] hbram_dout;    //read data
reg hbram_we;           //write-enable
ram_div_by_3  ram2(hbram_addr,~tv_in_line_clock1,hbram_din,hbram_dout,hbram_we);


//the following code initializes the BRAMs using hardware
//If we replaced the BRAMs with ROMs, this code could be eliminated
reg [9:0] mem_count;    //address for both BRAMs
reg [8:0] div;      //mem_count/3
reg [1:0] count_mod_3;  //counter mod 3

always @ (negedge tv_in_line_clock1) begin
    if (reset) begin
        //reset all counters
        mem_count <= 0;
        count_mod_3 <= 0;
        div <= 0;
    end
    else if (mem_count < 725) begin
        //we never need to divide part 720 - to be safe, go up to 724

        //increment mem_count every cycle
        mem_count <= mem_count+1;

        //every 3 clock cycles, reset count_mod_3 and increment div
```

```
        if (count_mod_3 == 2) begin
            count_mod_3 <= 0;
            div <= div+1;
        end
        else count_mod_3 <= count_mod_3+1;
    end
    else
        //if mem_count is past 724, stop running the initialization logic
        mem_count <= 730;
end

always @ (mem_count) begin
    if (mem_count < 730) begin
        //write the memories with the correct divide-by-three values
        vbram_we = 1;
        hbram_we = 1;
        vbram_addr = mem_count;
        hbram_addr = mem_count;
        vbram_din = div;
        hbram_din <= div;
    end
    else begin
        //we are done with initialization, so stop writing memory
        vbram_we =0;
        hbram_we = 0;
        vbram_addr = v_position;
        hbram_addr = h_position;
    end
end



//combinational logic to generate zoom_value
always @ (h_position or v_position or stored_read_data or zoom_mag) begin
    zoom_value = zoom_enable ? stored_read_data : ycrcb_in; //only zoom if zoom_enable is
                                //asserted

    //calculate the read address to zoom in properly. This code implements a simple sample/
    //hold filter, i.e. just copy each pixel the correct number of times in both directions

    //Note: we should calculate the read address for the pixel we want to read two cycles
    //early since the ZBT has a two cycle latency. However, practically there is no real
    //benefit to doing this since our TV monitor does not show the edges of the frame anyway
    if (zoom_mag == 0) begin
        //calculate read address assuming scale factor of 2
```

```
        //the funny stuff with 2*v_position[0] is a hack to stop the zoomed image
        //from flickering. Basically this offsets one field. The reason we need this
        //may have something to do with the position counter code and the fact that
        //one field has an extra blanking line than the other field
        read_addr = ({v_position[9:2],~v_position[0]}+2*v_position[0])*360
                +h_position[9:1]+(write_offset ? 250*360 : 0);
    end
    else if (zoom_mag==1) begin
        //calculate read address assuming scale factor of 3
        read_addr = ({vbram_dout[7:1],~v_position[0]}+2*v_position[0])*240
                +hbram_dout+(write_offset ? 250*360 : 0);
    end

    else if (zoom_mag == 2) begin
        //calculate read address assuming scale factor of 4
        read_addr = ({v_position[9:3],~v_position[0]}+2*v_position[0])*180
                +h_position[9:2]+(write_offset ? 250*360 : 0);
    end
    else
        read_addr = 0;
end
endmodule
```

A.8 Overlay Module - overlay.v

```
////////////////////////////////////////////////////////////////////////////////
//
// Overlay Module (overlay.v)
//
////////////////////////////////////////////////////////////////////////////////

module overlay (reset, tv_in_line_clock1, ycrcb_in, ycrcb_out, fvh_in, fvh_out,
            trace_enable, text1_enable, text2_enable, framegrab_enable, select,
            data_bus, write_enable, clear, ready, framegrab_data,
            framegrab_addr, surface_mouse_pos, full_screen_enable);
   input reset;             //reset the module
   input tv_in_line_clock1;     //tv line-locked clock
   input [29:0] ycrcb_in;       //pixel data from previous module
   input [2:0] fvh_in;          //field, v_sync, and h_sync signals from zoom
                  //module
   input trace_enable;          //enable signal for trace overlay functionality
                  //(active HIGH)
   input text1_enable;          //enable signal for first text buffer overlay
                  //(active HIGH)
   input text2_enable;          //enable signal for second text buffer overlay
                  //(active HIGH)
```

```
input framegrab_enable;      //enable signal for framegrab overlay
                 //(active HIGH)
input full_screen_enable;        //switch between full-screen and picture-in-picture
                 //modes (HIGH enables full-screen mode; LOW enables
                 //picture-in-picture mode)
input [2:0] select;          //signal from control module indicating what the
                 //content on the data bus means
input [20:0] data_bus;       //general purpose data bus used by control module
                 //to write the overlay module's BRAMs/registers
input write_enable;          //data is written to memory only when write_enable
                 //is asserted by control module (active HIGH)
input clear;             //one cycle pulse from control module; when asserted,
                 //the trace buffer should be cleared (active HIGH)
input [29:0] framegrab_data;     //data from the framegrab ZBT buffer
wire [29:0] framegrab_data;
input [19:0] surface_mouse_pos;      //mouse position from control module - this is used
                 //to render a cursor on the video output
                 //bits [19:10] are the vertical position
                 //bits [9:0] are the horizontal position


output [29:0] ycrcb_out;     //output pixel value after overlay processing
output [2:0] fvh_out;        //output field, v_sync, and h_sync signals
output ready;            //output to control module - if asserted (active
                 //HIGH), overlay is ready to accept data from the
                 //control module

output [19:0] framegrab_addr;    //memory address for ZBT RAM in framegrab module
                 //bits [19:10] specify the row of the pixel to read
                 //bits [9:0] specify the column of the pixel to read
reg [29:0] ycrcb_out=0;
reg [2:0] fvh_out=0;
reg [19:0] framegrab_addr=0;
reg ready=0;

parameter WHITE = {10'h3AC,10'd464,10'd464}; //pixel value for approximately white color
parameter TEXT_HEIGHT = 10'd24;          //Height of text characters in  pixels



reg [19:0] text1_pos, text2_pos; //stores the upper left corner of the two text
                 //buffers
                 //bits [19:10] are the row (v_position)
                 //bits [9:0] are the column (h_position)
```

```
reg [19:0] framegrab_pos;        //stores upper left corner for framegrab position
                 //in picture-in-picture mode
                 //bits [19:10] are the row (v_position)
                 //bits [9:0] are the column (h_position)

reg [29:0] overlay_value;        //output containing the modified pixel value after
                 //applying the appropriate overlay objects
                 //this is produced by combinational logic, so we
                 //latch it to produce ycrcb_out

reg [19:0] framegrab_addr_1;     //framegrab address produced by combinational logic;
                 //we latch it to produce framegrab_addr

//latch framegrab address
always @ (negedge tv_in_line_clock1) begin
 framegrab_addr <= framegrab_addr_1;
end


//just pass the field, v_sync and h_sync to the next module
//technically, we should delay so that there is no horizontal shift between input and
//output, but this has a negligible effect
always @ (negedge tv_in_line_clock1) begin
     fvh_out <= fvh_in;
end




//Generate position counters based on field, v_sync, and h_sync
//NOTE : the code to generate the position counters is not very clean. A cleaner version
//is available in the framegrab or bluescreen modules
//We did not change this code because we did not have time to test the overlay module
//with the cleaner code

reg [10:0] sample_count;         //sample_count and line_count are
                     //intermediate counters used to derive the
                     //the current pixel's position

reg [9:0] line_count;            //h_position is the current pixel's column
reg [9:0] h_position, v_position;        //v_position is the current pixel's row


always @(negedge tv_in_line_clock1)
  if (reset)
```

```
    begin
   //set the counters to 0
   sample_count <= 0;
   line_count <= 0;
    end
  else
    begin
   //sample_count < 1440 corresponds to active video, so latch overlay_value to get
       //ycrcb_out
   if (sample_count < 1440)
     ycrcb_out <= overlay_value;

   if (sample_count == 1715)
   //sample_count == 1715 corresponds to the end of the current line, so we reset
       //sample_count and increment line count
     begin
        sample_count <= 0;
        if (line_count == 524)
      line_count <= 0;
        else
      line_count <= line_count+1;
     end
       else if (fvh_in[1]) begin
   //if v_sync is asserted, reset sample_count and line_count because we are
   //in a blanking period between frames
     sample_count<=0;
     line_count<=0;
     end
   else
   //otherwise we just increment sample_count
     sample_count <= sample_count+1;
    end



//produce h_position and v_position from the sample_count and line_count
//h_position is essentially sample_count/2 because NTSC works by interleaving Y with Cr
//and Cb, effectively producing a pixel every 2 cycles
//v_position's least significant bit depends on the field because of even-odd interlacing

//Note: this can probably be replaced with a combinational always block, but we did not
//test this
always @ (negedge tv_in_line_clock1)
  if (line_count < 19)
    begin
```

```
   h_position <= 0;
   v_position <= 0;
    end
  else if (line_count < 262)
    begin
   h_position <= (sample_count[10:1] < 719) ?
            sample_count[10:1]+sample_count[0] : 0;
   v_position <= {line_count-19, fvh_in[2]};
    end
/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

//Some notes about the memories we use:
//The trace BRAM has 240*720=172800 locations. We don't use 480*720 locations because we
//chose to have the same data output to both fields. Otherwise, the trace overlay object
//could flicker since the interlacing means each field is refreshed at a lower rate
//(about 30Hz).
//The text BRAMs both have 24*720=17280 locations. The text rendering is done by a font
//ROM inside the control module.
//All the BRAMs store 1 bit per location - a 1 indicates that the pixel should be changed
//to white (i.e. apply the overlay). A 0 means the pixel should be transparent, i.e.
//just pass along ycrcb_in. Of course, the actual assignment of overlay_value also
//depends on the enable signals.
//All the BRAMs are dual port. One port is a read port, and the other is a write port.
//The idea is that the control module only needs to write, and the logic to calculate the
//overlay output only needs to read

//Setup the trace BRAM
reg [18:0] trace_read_addr;      //trace BRAM read address
reg [18:0] trace_write_addr;     //trace BRAM write address
reg trace_write_din;         //trace BRAM write data
wire trace_read_dout;        //trace BRAM read data
reg trace_write_we;          //trace BRAM write-enable (active HIGH)

trace_ram trace1 (trace_read_addr, trace_write_addr, ~tv_in_line_clock1,
          ~tv_in_line_clock1, trace_write_din, trace_read_dout, trace_write_we);

//Note: this can probably be combinational, but we did not test this
always @ (negedge tv_in_line_clock1) begin
 trace_read_addr <= v_position[9:1]*720+h_position;   //technically, we should
                              //actually read the next pixel

                              //this code causes a horizontal
                                                      //shift, but practically this
                              //is not a problem.
```

```
                              //this comment applies to the
                              //other read addresses as well
end


//Setup text1 BRAM
reg [14:0] text1_read_addr;
reg [14:0] text1_write_addr;
reg text1_write_din;
wire text1_read_dout;
reg text1_write_we;

text_ram text1 (text1_read_addr, text1_write_addr, ~tv_in_line_clock1,
         ~tv_in_line_clock1, text1_write_din, text1_read_dout,
         text1_write_we);

//Note: this can probably be combinational, but we did not test this
//we only compute the address if we are at a position where the pixel is in the range of
//the text1 buffer
always @ (negedge tv_in_line_clock1) begin
 if (v_position>=text1_pos[19:10] && v_position<text1_pos[19:10]+TEXT_HEIGHT) begin
     if (h_position > text1_pos[9:0]) begin
     text1_read_addr <= (v_position-text1_pos[19:10])*720
                +(h_position-text1_pos[9:0]);
     end
 end
end


//Setup text2 BRAM
reg [14:0] text2_read_addr;
reg [14:0] text2_write_addr;
reg text2_write_din;
wire text2_read_dout;
reg text2_write_we;

text_ram text2 (text2_read_addr, text2_write_addr, ~tv_in_line_clock1,
         ~tv_in_line_clock1, text2_write_din, text2_read_dout,
         text2_write_we);

//Note: this can probably be combinational, but we did not test this
//we only compute the address if we are at a position where the pixel is in the range of
//the text2 buffer
always @ (negedge tv_in_line_clock1) begin
 if (v_position>=text2_pos[19:10] && v_position<text2_pos[19:10]+TEXT_HEIGHT) begin
```

```
     if (h_position > text2_pos[9:0]) begin
     text2_read_addr <= (v_position-text2_pos[19:10])*720
                +(h_position-text2_pos[9:0]);
     end
 end
end
```

```
//Select Logic for writing memory
//Note: A lot of this code can probably be combinational, but we did not test this

reg [17:0] memory_count = 0;

//Signal Code For Control Module
parameter TEXT1_DATA = 0;           //Data bus has text1 data and address
parameter TEXT1_POSITION = 1;       //Data bus has text1 position
parameter TEXT2_DATA = 2;           //Data bus has text2 data and address
parameter TEXT2_POSITION = 3;       //Data bus has text2 position
parameter FRAMEGRAB_POSITION = 4;       //Data bus has framegrab position
parameter TRACE_DATA = 6;           //Data bus has trace address and data


always @ (negedge tv_in_line_clock1) begin
 if (clear) begin
    //Start clearing the trace buffer, and de-assert ready
    ready <= 0;
    memory_count <= 0;   //counter to keep track of how many memory locations have
             //been cleared
 end
 else if (~ready) begin
     //we can think of this as a really simple FSM
     //If we are in the state where ready is not asserted, increment memory_count
     //and clear the next address in the trace buffer
     trace_write_din <= 0;
     trace_write_we <= 1;
     trace_write_addr <= memory_count;
     memory_count <= memory_count+1;

     //there are 172800 locations in the trace buffer, so clear after
     //memory_count gets this high
     //We can assert ready at this point since the trace buffer has been cleared
     if (memory_count == 172800) begin
```

```
            memory_count <= 0;
            ready <= 1;
        end
    end
    else
    //look at the select signal and decode it properly using the signal code
    case (select)
        TEXT1_DATA: begin
            text1_write_din <= data_bus[20];
            text1_write_addr <= data_bus[19:10]*720+data_bus[9:0];//calculate address
            text1_write_we <= write_enable;
            end
        TEXT1_POSITION:
            text1_pos <= write_enable ? data_bus[19:0] : text1_pos;
        TEXT2_DATA: begin
            text2_write_din <= data_bus[20];
            text2_write_addr <= data_bus[19:10]*720+data_bus[9:0];//calculate address
            text2_write_we <= write_enable;
            end
        TEXT2_POSITION:
            text2_pos <= write_enable ? data_bus[19:0] : text2_pos;
        FRAMEGRAB_POSITION:
            framegrab_pos <= write_enable ? data_bus[19:0] : framegrab_pos;
        3'b101:
            trace_write_we <= 0;    //do nothing in this case
        TRACE_DATA: begin
            trace_write_din <= data_bus[20];
            trace_write_addr <= data_bus[19:11]*720+data_bus[9:0]; //calculate address
            trace_write_we <= write_enable;
            end
        3'b111:
            trace_write_we <= 0;    //do nothing in this case
    endcase
end




//latch the data coming from the framegrab module
//otherwise the overlay looks noisy
reg [29:0] real_frame_data;
```

```
always @ (negedge tv_in_line_clock1) begin
 real_frame_data <= framegrab_data;
end



//parameters which control cropping during picture-in-picture mode
parameter CROP_LEFT = 10;     //controls amount to crop from left
parameter CROP_RIGHT = 6;     //controls amount to crop from right
parameter CROP_TOP = 19;  //controls amount to crop from top; choose at least 19 to
              //eliminate blanking lines
              //Note: this could be caused because of the different code
              //used by the framegrab and overlay modules to generate
              //h_position and v_position

parameter CROP_BOTTOM = 20;   //controls amount to crop from bottom - because of the way
              //the code is written, CROP_BOTTOM should be bigger than
              //CROP_TOP or invalid addresses may be generated



//combinational logic to assign the output based on what overlay objects have been
//enabled

always @(h_position or v_position or trace_enable or text1_enable or text2_enable or
     framegrab_enable or full_screen_enable) begin
 if (v_position == surface_mouse_pos[19:10] && h_position > surface_mouse_pos[9:0]-5
     && h_position < surface_mouse_pos[9:0]+5) begin
         //mouse cursor condition - create horizontal line 9 pixels wide
         overlay_value = WHITE;
         framegrab_addr_1 = 20'bxxxxxxxxxxxxxxxxxxxx;    //we specify an assignment
                                //in all cases so that Verilog
                                //does not produce a latch, etc.
 end
 else if (h_position == surface_mouse_pos[9:0] && v_position > surface_mouse_pos[19:10]-5
     && v_position < surface_mouse_pos[19:10]+5) begin
         //mouse cursor condition - create vertical line 9 pixels high
         overlay_value = WHITE;
         framegrab_addr_1 = 20'bxxxxxxxxxxxxxxxxxxxx;
 end
     else if (trace_enable && trace_read_dout) begin
         //apply trace overlay
         overlay_value = WHITE;
         framegrab_addr1 = 20'bxxxxxxxxxxxxxxxxxxxx;
 end
```

```
    else if (text1_enable && v_position>=text1_pos[19:10]
        && v_position < (text1_pos[19:10]+TEXT_HEIGHT)
        && text1_read_dout && h_position>text1_pos[9:0]) begin
            //apply text1 overlay
            overlay_value = WHITE;
            framegrab_addr_1 = 20'bxxxxxxxxxxxxxxxxxxxx;
    end
    else if (text2_enable && v_position>=text2_pos[19:10]
        && v_position < (text2_pos[19:10]+TEXT_HEIGHT)
        && text2_read_dout && h_position>text2_pos[9:0]) begin
            //apply text2 overlay
            overlay_value = WHITE;
            framegrab_addr_1 = 20'bxxxxxxxxxxxxxxxxxxxx;
    end
    else if (framegrab_enable) begin
        //apply framegrab overlay
        if (full_screen_enable) begin
            //calculate framegrab address for a fullscreen overlay
            framegrab_addr_1[19:10] = v_position+19;    //add 19 to cut off the blanking lines
            framegrab_addr_1[9:0] = h_position;
            overlay_value = real_frame_data;
        end
        else begin
            //we are in picture-in-picture mode, so calculate framegrab_addr_1
            //by subsampling by a factor of 2 in both directions. This makes the
            //overlay take up about a quarter of the screen


            //the following code is a hack to implement a crude form of prefetching
            //ZBT RAM has a 2 cycle latency, so start sending valid addresses to
            //memory 2 cycles before you want to display data
            //just to be safe, we start sending addresses 3 cycles early


            //the conditions below also crop the frame horizontally and vertically
            if (v_position > framegrab_pos[19:10]
     && v_position <= framegrab_pos[19:10]+240-CROP_BOTTOM
                && h_position > framegrab_pos[9:0]+CROP_LEFT
&& h_position < framegrab_pos[9:0]+CROP_LEFT+4) begin
                //prefetching stage; data from ZBT isn't valid yet, so still
                //assign output to be ycrcb_in

                //we multiply by 2; this probably gets optimized out, but to
                //be safe we could use a shift instead
                framegrab_addr_1[19:10] = 2*(v_position+CROP_TOP)-2*framegrab_pos[19:10];
```

```
                framegrab_addr_1[9:0] = 2*h_position-2*framegrab_pos[9:0];
                overlay_value = ycrcb_in;
            end
            else if (v_position > framegrab_pos[19:10]
     && v_position <= framegrab_pos[19:10]+240-CROP_BOTTOM
                && h_position >= framegrab_pos[9:0]+CROP_LEFT+4
&& h_position <= framegrab_pos[9:0]+360-CROP_RIGHT)
            begin
                //overlay stage - now we can assign the latched ZBT data to overlay_value

                //we multiply by 2; this probably gets optimized out, but to
                //be safe we could use a shift instead
                framegrab_addr_1[19:10] = 2*(v_position+CROP_TOP)-2*framegrab_pos[19:10];
                framegrab_addr_1[9:0] = 2*h_position-2*framegrab_pos[9:0];
                overlay_value = real_frame_data;
            end
            else begin
                // the current pixel is not in the overlay area,
// so overlay_value should just be ycrcb_in
                overlay_value = ycrcb_in;
//address does not matter in this case
                frame_grab_addr_1 = 20'bxxxxxxxxxxxxxxxxxxxx;
            end
        end
    end
    else
        overlay_value = ycrcb_in;
   end
endmodule
```

A.9 Video Output Module - video.v

This is essentially identical to the video.v file provided with the labkit. We just added a few inputs so that video.v can process the stream we produce.

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Video (TV) Test Code
//
// For Labkit Revision 004
//
//
// Created: November 3, 2004
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
```

```
`include "i2c.v"
`include "adv7194init.v"
`include "adv7185init.v"

module video (reset, clock_27mhz, 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, mode, ycrcb_in, fvh_in);

      //We added ycrcb_in, fvh_in, and use video_encoder instead of videologo

    input [29:0] ycrcb_in;
    input [2:0] fvh_in;

   input reset;
   input clock_27mhz;

   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 [9:0] 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;
   input [1:0] mode;

   // Mode Decoding
   //
   // 0 = colorbars, 1 = MIT logo, 2 = composite passthrough,
   // 3 = s-video passthrough

   wire colorbars, logo, svideo;
   assign colorbars = (mode == 0);
   assign logo = (mode == 1);
   assign svideo = (mode == 3);

   //
   // Logo generator
   //

   wire [9:0] logo_ycrcb;
```

```
   video_encoder enc1 (reset, tv_in_line_clock1, logo_ycrcb, ycrcb_in, fvh_in);

   //
   // ADV7194 (Output)
   //

   assign tv_out_clock = (colorbars) ? clock_27mhz : tv_in_line_clock1;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_vsync_b = 0;
   assign tv_out_hsync_b = 0;
   assign tv_out_subcar_reset = 0;
   assign tv_out_blank_b = 0;

   adv7194init init7194 (reset, clock_27mhz, colorbars,
             tv_out_reset_b, tv_out_i2c_clock, tv_out_i2c_data);

   assign tv_out_ycrcb = logo ? logo_ycrcb : tv_in_ycrcb;
endmodule
```

A.10 Video Output Module - video_encode.v

```
// CCIR656 Video Encoder
// Ben Gelb
//
// This module is based on the videologo module from the 6.111 labkit A/V test code.
//
// This module is intended to be used with a data stream such as the one
// generated by Javier Castro's CCIR656 decoder module for the ADV7195. It
// outputs CCIR656 encoded video data in a 10-bit format for use by the ADV7194
// video encoder chip.
//
// It expects video data in 30-bit words with Y Cr and Cb values, with
// every two successive samples the same. The field, v-sync and h-sync
// signals must be timed such that they occur when EAV and SAV timecodes
// would be present in a CCIR656 stream. Details below:
//
// h-sync - must be 0 immediately prior to the first sample of video data
//          on a given line. Then 1 on the 4th sample after the end of the
//          line.
//
// v-sync - must be 1 immediately prior to the first sample of a vertical
//          blanking line, and 1 again on the 4th sample afterwards.
//
// field -  must transition at the end of a line (h-sync and v-sync both 1) somewhere
//          in the vertical blanking period between retraces.
```

```
module video_encoder (reset, tv_clock, ycrcb, ycrcb_in, fvh_in);

    input reset, tv_clock;
    input [29:0] ycrcb_in;
    input [2:0] fvh_in;
    output [9:0] ycrcb;

    wire [9:0] y,cr,cb;
    assign y = ycrcb_in[29:20];
    assign cr = ycrcb_in[19:10];
    assign cb = ycrcb_in[9:0];

    reg [9:0] ycrcb;
    reg [10:0] sample_count;
    wire f, v, h;
    wire [7:0] xy;
    assign f = fvh_in[2];
    assign v = fvh_in[1];
    assign h = fvh_in[0];

    // calculate timecode and parity bits for SAV and EAV
    assign xy = {1'b1, f, v, h, v^h, f^h, f^v, f^v^h};

    wire leading_vsync;
    assign leading_vsync = v&~h; //the vsync at the beginning of a line

    // sample counter, runs from 0 to 1715 on each line
    // active video from 0 to 1439
    always @ (negedge tv_clock) begin
        if (reset|leading_vsync|(sample_count == 1715)) sample_count <= 0;
        else sample_count <= sample_count + 1;
    end

    always @ (negedge tv_clock) begin

        // samples 0 to 1439 are for active video
        // 720 pixels per line,  two samples (Y and C) per position
        // Sample order is Cb0, Y0, Cr1, Y1, Cb2, Y2, . . .
        if (sample_count < 1440)
            case (sample_count[1:0])
                2'b00: ycrcb <= cb;
                2'b01: ycrcb <= y;
                2'b10: ycrcb <= cr;
                2'b11: ycrcb <= y;
            endcase
```

```
        // Start header for the EAV timecode
        else if (sample_count == 1440) ycrcb <= 10'h3FC;
        else if (sample_count == 1441) ycrcb <= 10'h000;
        else if (sample_count == 1442) ycrcb <= 10'h000;
        else if (sample_count == 1443) ycrcb <= {xy, 2'b00}; //EAV timecode

        // samples in between active video lines, output blanking levels
        else if (sample_count < 1712 ) ycrcb <= sample_count[0] ? 10'h040 : 10'h200;

        // Begin header for SAV timecode
        // This is the timecode for the start of the NEXT line
        else if (sample_count == 1712) ycrcb <= 10'h3fC;
        else if (sample_count == 1713) ycrcb <= 10'h000;
        else if (sample_count == 1714) ycrcb <= 10'h000;
        else if (sample_count == 1715) ycrcb <= {xy, 2'b00}; // SAV timecode
    end

endmodule
```

A.11 Labkit Module - avtest.v

```
//The labkit module contains instantiations of all the modules we wrote,
//and connects them together to form the complete system.
//There is also a switch debounce module at the end, which is useful
//for debugging purposes. We copied the switch debounce module from
//the code on the labkit code on the website.

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

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

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

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

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

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

        clock_feedback_out, clock_feedback_in,

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

        rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

        mouse_clock, mouse_data, keyboard_clock, keyboard_data,

        clock_27mhz, clock1, clock2,

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

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

        switch,

        led,

        user1, user2, user3, user4,

        daughtercard,

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

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

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

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

output [9:0] tv_out_ycrcb;
```

```
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
   tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
   tv_out_subcar_reset;

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

inout  [35:0] ram0_data;
output [20:0] 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 [20:0] 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 [24:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

inout  mouse_clock, mouse_data;
input  keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//Reset Generation
wire reset;
SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(reset),
        .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

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

////////////////////////////////////////////////////////////////////////////
//
// VGA Output
//
////////////////////////////////////////////////////////////////////////////

 // 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 [2:0] btn_click;
wire [11:0] mx,my;

wire ps2_clk, ps2_data;

ps2_mouse_xy ms(tv_in_line_clock1, reset, mouse_clock, mouse_data, mx, my, btn_click);

wire hs,vs,b;
wire [2:0] rgb;

// keyboard input
wire [7:0] ascii;
wire char_rdy;
ps2_ascii_input kbd(tv_in_line_clock1, reset, keyboard_clock, keyboard_data, ascii,
           char_rdy);

//instantiate control module
wire framegrab_trig,bluescreen_en,bluescreen_cal,zoom_en,overlay_ready,overlay_clear;
wire [3:0] overlay_en;
wire [19:0] zoom_pos;
wire [1:0] zoom_mag;
wire [20:0] overlay_data;
wire [2:0] overlay_select;
wire overlay_we;
wire [19:0] surface_mouse_pos;
wire framegrab_addr_select;
wire overlay_pip;


control control1(reset,tv_in_line_clock1,hs,vs,rgb,b,mx[9:0],my[9:0],btn_click[2],
            clock_65mhz,ascii,char_rdy,
            framegrab_trig, framegrab_addr_select,
            bluescreen_en, bluescreen_cal,
            zoom_en, zoom_pos, zoom_mag,
            overlay_ready, overlay_en, overlay_pip, overlay_data,
            overlay_select, overlay_clear, overlay_we,
            surface_mouse_pos);

assign led[5:2] = ~overlay_en;
assign led[6] = ~overlay_clear;
```

```
 // VGA Output.  In order to meet the setup and hold times of the
 // AD7125, we send it ~clock_65mhz.
     assign vga_out_red = {8{rgb[2]}};
 assign vga_out_green = {8{rgb[1]}};
 assign vga_out_blue = {8{rgb[0]}};
 assign vga_out_sync_b = 1'b1;    // not used
 assign vga_out_blank_b = ~b;
 assign vga_out_pixel_clock = ~clock_65mhz;
 assign vga_out_hsync = hs;
 assign vga_out_vsync = vs;


// wire up ZBT ram 1 for use with the zoom module

wire [35:0] vram_write_data_2;
wire [35:0] vram_read_data_2;
wire [18:0] vram_addr_2;
wire        vram_we_2;

zbt_6111 zbt2(~tv_in_line_clock1, 1'b1, vram_we_2, vram_addr_2,
        vram_write_data_2, vram_read_data_2,
        ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

////////////////////////////////////////////////////////////////////////////
//
// Video Input
//
////////////////////////////////////////////////////////////////////////////


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

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

ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
            .tv_in_ycrcb(tv_in_ycrcb[19:10]),
            .ycrcb(ycrcb), .f(fvh[2]),
```

```
            .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

wire [29:0] ycrcb_out;
wire [2:0] fvh_out;


//Instantiate bluescreen module
wire [19:0] bluescreen_framegrab_addr;
wire [29:0] overlay_read_data;

bluescreen blue1(reset, tv_in_line_clock1, bluescreen_en, bluescreen_cal, ycrcb, fvh,
         ycrcb_out, fvh_out, bluescreen_framegrab_addr, overlay_read_data);




//Instantiate zoom module
wire [2:0] fvh_zoom;
wire [29:0] ycrcb_zoom;
zoom zoom1 (reset, tv_in_line_clock1, ycrcb_out, ycrcb_zoom, fvh_out,
        fvh_zoom, vram_read_data_2, vram_write_data_2, vram_addr_2, vram_we_2,
        zoom_en, zoom_pos, zoom_mag);




//Instantiate overlay_module
wire [29:0] ycrcb_overlay_out;
wire [2:0] fvh_overlay_out;
wire [19:0] overlay_read_addr;


overlay over1 (reset, tv_in_line_clock1, ycrcb_zoom, ycrcb_overlay_out, fvh_zoom,
                fvh_overlay_out, overlay_en[0], overlay_en[3], overlay_en[2],
            overlay_en[1], overlay_select, overlay_data, overlay_we,
            overlay_clear, overlay_ready, overlay_read_data, overlay_read_addr,
            surface_mouse_pos, ~overlay_pip);




//Instantiate framegrab module and wire up to ZBT
wire [35:0] vram_write_data;
wire [35:0] vram_read_data;
wire [18:0] vram_addr;
```

```
 wire        vram_we;
 wire [1:0] fg_state;
 assign led[1:0] = fg_state;

 zbt_6111 zbt1(~tv_in_line_clock1, 1'b1, vram_we, vram_addr,
        vram_write_data, vram_read_data,
        ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

 framegrab fg1(reset, tv_in_line_clock1, ycrcb, fvh, vram_read_data,
  vram_write_data, vram_addr, vram_we, framegrab_trig, bluescreen_framegrab_addr,
  overlay_read_addr, overlay_read_data,  framegrab_addr_select, fg_state);

//////////////////////////////////////////////////////////////////////////////
//Video Output
//////////////////////////////////////////////////////////////////////////////

 wire [1:0] videomode;    //videomode controls which mode the output module operates
             //in. See video.v for more details

 debounce vmode0 (reset, clock_27mhz, switch[3], videomode[0]);
 debounce vmode1 (reset, clock_27mhz, switch[4], videomode[1]);

 video video_test (reset, clock_27mhz, 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[19:10],
          tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
          tv_in_aef, tv_in_hff, tv_in_aff, videomode, ycrcb_overlay_out,
          fvh_overlay_out);


 //Default Video assignments
 assign tv_in_fifo_read = 1'b1;
 assign tv_in_fifo_clock = 1'b0;
 assign tv_in_iso = 1'b1;
 assign tv_in_clock = clock_27mhz;

 PULLUP pu_in (.O(tv_in_i2c_data));
 PULLUP pu_out (.O(tv_out_i2c_data));

 assign user1 = {6'b000000,
        tv_in_line_clock1,  // 25
        tv_in_i2c_clock,    // 24
        tv_in_i2c_data,     // 23
        tv_in_ycrcb[19:10], // 22-13
```

```
        clock_27mhz,        // 12
        tv_out_i2c_clock,   // 11
        tv_out_i2c_data,    // 10
        tv_out_ycrcb};      // 9-0

assign analyzer1_clock = clock_27mhz;
assign analyzer1_data = vram_addr[15:0];

assign analyzer2_clock = tv_in_line_clock1;
assign analyzer2_data = vram_read_data[15:0];
assign analyzer3_data = vram_read_data[31:16];


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

// 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 = 21'h0;
   assign ram1_clk = 1'b0;
   assign ram1_we_b = 1'b1;
   assign ram1_cen_b = 1'b1;
   */
```

```
assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_adv_ld = 1'b0;
assign ram1_bwe_b = 4'h0;
assign clock_feedback_out = 1'b0;

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 15'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;

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;

// Buttons, Switches, and Individual LEDs
/*reg [7:0] led;
always @ (posedge tv_in_line_clock1) begin
led[7:0] <= 8'h0f;
end*/


// User I/Os
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;

// Logic Analyzer
//assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
```

```
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;
endmodule

///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

   always @(posedge clock)
     if (reset)
       begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
       end
     else if (noisy != new)
       begin
      new <= noisy;
      count <= 0;
       end
     else if (count == 270000)
       clean <= new;
     else
       count <= count+1;

endmodule
```