Image Browser and Manipulator

By Mary Knapp, Prannay Budhraja

Abstract

This device allows a user to browse and transform images. To browse we have describes a filmstrip interface that lets the user scroll left or right to view images. Once an image is selected, it can be scaled or rotated according to the user's needs.

Table of Contents

Overview

The purpose of this project is to create a hardware-only device for browsing and manipulating images. Such a device would allow a user to view and interact with images without the overhead of an operating system. The concept is something like digital photo frames, but allows the user to manipulate the images rather than just look at them.

This system has three main functions: browse, rotate, and scale (or zoom). In order to deliver the required functionality, the system will have several modes of operation. In browse mode, all available images will be displayed in a filmstrip format. The user will be able to scroll through this film strip to see all the images. The user can then select an image to manipulate. Once an image is selected, it jumps to the center of the screen and all other images disappear. Then the user can scale or rotate the image as desired.

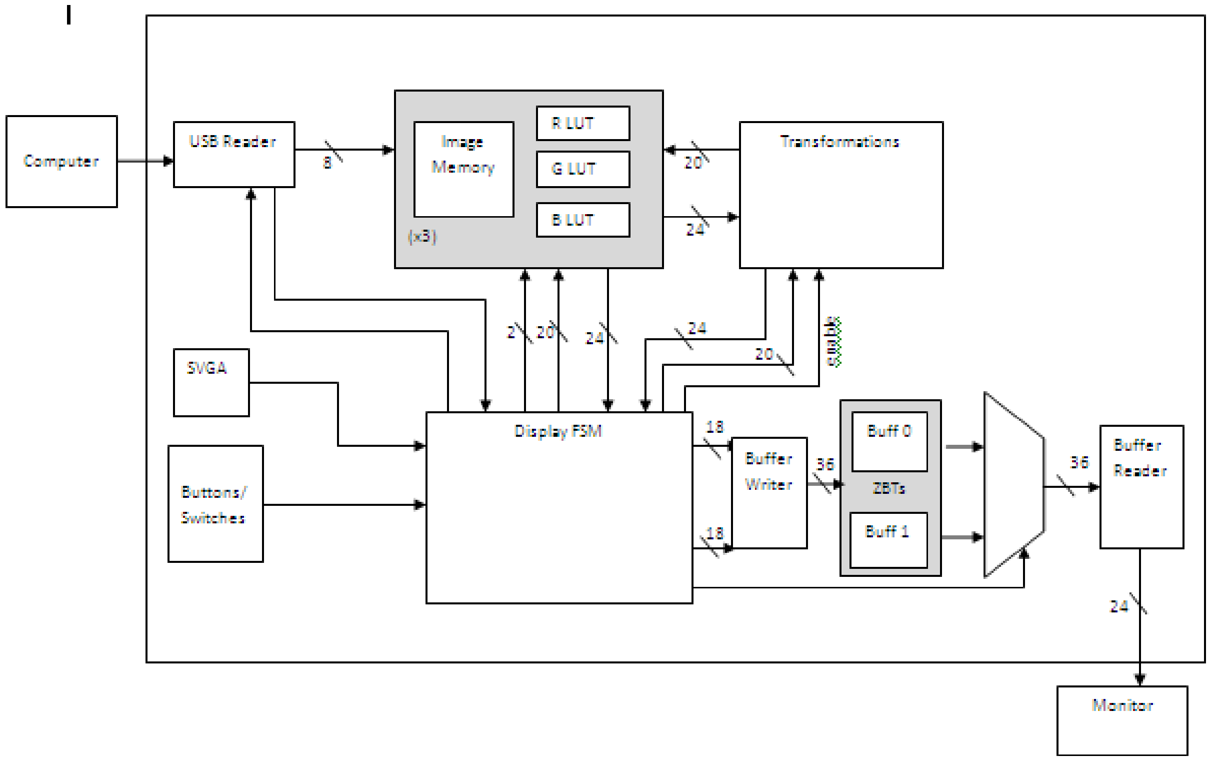
This project can be broken up into a UI display system and a graphics engine that performs the image manipulations. Mary Knapp was responsible for the display portion and Prannay Budhraja was responsible for the graphics engine.

Description

Modules

Block Diagram:

I



All modules take clock and reset signals. The reset signal is high either at system reset or when the enter button is pressed. One 40MHz clock is used for the entire system.

Display

The display module is a master module that controls the location of images on the screen, their motion, and mode selection. It contains a number of module instantiations within it. At the highest level, the display module's purpose is to produce appropriate pixels and pass them to the buffer writer. It also handles the generation of the hsync, vsync, and blank signals.

This module also controls switching between display modes. At reset, the default mode is browse, in which all of the images are visible. Images move when the location of the top left corner is changed. Memory addresses for all four images are generated on every clock cycle, but they are only valid within the "box" that the image occupies. Both the x-coordinate of the image's top left corner and hcount are signed numbers, which allows smooth scrolling both left and right. Pressing a button (button0) changes the mode to select. A pulse is generated on button depression which flips a bit in the display module to signal a change of mode. The system enters select mode by checking which picture is near the center of the screen and then moving that image to the center and hiding the others. Once in this mode, image manipulation would kick in. Unfortunately, time constraints and integration problems prevented full implementation of image manipulation within the display system. A second button press returns the display to the original film strip. Table 1 shows the functions of various lab kit buttons and switches.

Table 1 - User input devices

Input device	Function
Enter Button	Reset
Button0	Switch display modes (browse -> select -> browse)
Left Button	Scroll left
Right Button	Scroll right
Switch[3:2]	Scroll speed
Switch[7:4]	Select image (debugging only)
Switch[1:0]	Switch display between color bars and real output (debugging only)

Memory Considerations and Screen Resolution (M. Knapp)

In order to implement the design, it was necessary to store a number of images in memory. The 6.111 lab kit has limited memory resources (~4Mb ZBT RAM, ~4Megabits distributed). The most memory intensive components of the project are the frame buffers. Each buffer must be able to store every pixel on the screen. Since a double buffer video system requires two full buffers, the memory requirement climbs sharply with screen resolution. This project incorporates rotation and scaling operations, so the frame buffers must contain 24-bit RGB pixels that can be read by the monitor. Storing 8-bit values is not sufficient even though our images are 256 color since interpolation between pixels can only be performed properly on RGB values.

Initial calculations for a 1024x768 resolution (XVGA) showed that the frame buffers alone would consume 4.3MB, which exceeded the resources of the lab kit. In order to leave room for storing the images themselves, a resolution of 800x600 (SVGA) was selected. While the frame buffers should ideally store 24-bit values, this leads to inefficient use of the ZBT RAM. The ZBTs are intended to store 4 bytes per address along with 4 parity bits, giving a total of 36 bits per line. Using 24-bit pixels wastes 12 bits per line. Much better storage can be achieved by storing two 18-bit pixels per line. In order to meet the monitor's 24-bit RGB pixel requirement, only the 6 most significant bits for each color are stored. Two zeros are added as the LSBs of each color as the buffer is being read out. With this scheme, each buffer takes up 1.08MB.

Image Selection and Preprocessing (M. Knapp)

Each image that will be displayed in the filmstrip is stored in a BRAM. Originally, these images were going to be stored in ZBT along with the frame buffers. The ZBT RAMS are serial devices, so only one memory address can be read on a clock cycle. Storing images in ZBT along with the frame buffers would require reading image pixels form ZBT, then processing them and writing them back to the ZBT. This requires extra time since the ZBT cannot read and write at the same time and would need to pause a frame buffer write in order to read out an image pixel. Storing the images in BRAM separates the memory sources and eliminates this problem.

Since storage space in BRAM is even more limited than in ZBT RAM, image size was strictly constrained in two ways. First, the physical size of each image was constrained to 200x150 pixels. Though this is a fairly small image, its size is reasonable given the reduced resolution. Second, images were limited to

256 colors. Each pixel is stored as an 8-bit value representing one of these 256 colors. This significantly reduces memory consumption. Each image is 30kb.

In order to produce 24-bit color, each image requires a set of three 256 entry look-up tables: one for red, green and blue. In order the generate the best possible image quality with 256 colors, Photoshop was used to convert full 16 million color images to 256 colors by choosing the "best" 256 colors for each image. This means that the LUTs for each image are unique. The concatenation of the output of all three LUTs is a full 24-bit RBG pixel. As stated above, only the 6 MSBs of each color are actually stored in the frame buffer.

After each image is reduced to 256 colors in Photoshop, the image is processed into a set of 4 (one image and 3 LUTs) COE files in MATLAB. These COE files are then preloaded into BROMS through Xilinx's CoreGen utility. With small modifications, the images could be loaded directly from a computer using a USB cable and adapter. Unfortunately, there was not enough time to implement this function. Once the BROMS are created, each one is instantiated inside the display module. Addresses for each image ROM are generated at each clock cycle. Since the COE file for the image transposes each row of pixels to a column and then stacks these columns (1st row of n length becomes 1st n values in column), the address formula is

$$\text{Address} = (\text{hcount} - x)+(\text{vcount} - y)*\text{width}$$

where (x,y) represents the top left corner of the image. The ROM and LUT reads are pipelined to take into account the two clock cycle memory read delay.

Frame Buffers (M. Knapp)

Frame buffer graphics implementation allows for smooth video with no glitches since the buffer being displayed is static – it is fully written before it is displayed and is not modified until the next buffer is ready to be displayed. The video clock is selected to produce a refresh rate of 60Hz, so for SVGA the clock runs at 40MHz. For complex rotation and scaling operations, the 60Hz refresh rate does not allow enough clock cycles for calculation. The frame rate was reduced to 30Hz to provide extra time. This does not lead to any reduction in video quality since the standard frame rate for film is 24Hz.

There are two banks of ZBT RAM on the lab kit and one frame buffer is stored on each. Two ZBT driver modules are used to switch between the buffers. The two modules are identical except that one sends all of its outputs to bank0 and the other to bank1. Two write_enable signals are controlled by a switching unit which uses hcount, vcount, and a 1-bit counter to invert both values every two frames (two cycles of hcount and vcount). Each ZBT driver module also has a unique read_data line, which is switched at the same rate to ensure that the buffer being written is not being read simultaneously.

To ensure clean video output, each frame buffer is completely rewritten every cycle. Pixels that are not inside the "box" occupied by an image are painted a uniform background color (red in this case). One line (2 pixels) is written every other clock cycle to achieve a write rate of 1 pixel/clock. When reading directly from the BROMs, the frame buffer can be completely written between two vsync signals (at 60Hz).

SVGA (M. Knapp)

This module controls the video signals that are sent to the monitor. It generates hcount and vcount, which are fed into almost all other modules in the system to provide synchronization. Hsync, vsync, and blank are also generated by this module. These signals are then sent to the display module where they are delayed by three clock cycles to account for memory read delays. The sync signals used by the monitor are the output of the display module, not the SVGA module. Internal values in this module are set to produce a resolution of 800x600 with the appropriate back porches and sync pulses. These values are calculated using the chart in the lab kit documentation under the VGA Video Output section.

Buffer Reader (M. Knapp)

This module performs two functions: It parses incoming 36-bit data from the memory into two 18-bit pixels, then adds two zeros as the LSBs for each color. Its output is a 24-bit pixel that is sent to the monitor. Since the ZBT requires 3 clock cycles to output a memory location, this module receives undelayed hcount and vcount values so that it is effectively three clock cycles ahead of the other modules.

Buffer Writer (M. Knapp)

This module is responsible for concatenating two 18-bit pixels to produce one 36-bit word that is sent to the frame buffer. It receives one pixel on every clock cycle. It stores the first pixel in a register, then concatenates it with the second incoming pixel and assigns the result to output. The LSB of hcount is used to determine whether to save the incoming pixel to a register (odd values) or assign it to output (even values).

ZBT Driver (M. Knapp)

This module was borrowed from the Fall 2005 class website. It streamlines communication with the ZBT memory includes pipelining for write mode to ensure efficient memory writes.

Debounce (M. Knapp)

The debounce module takes asynchronous inputs from buttons on the labkit and returns a clean, synchronous signal. This module was used for all buttons used in the design.

Delay (parameterized) (M. Knapp)

This module delays a signal by some number of clock cycles. It was especially useful for solving timing issues created by memory reads.

Transform

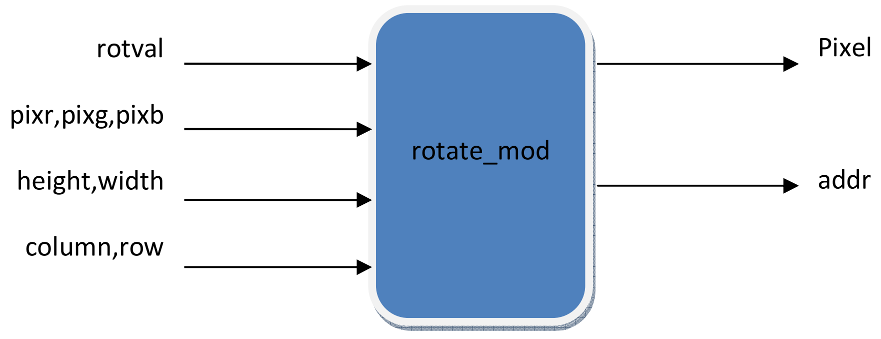
The image manipulation works in the select mode. Once an image is selected it can be rotated or scaled by amounts specified by the user. This module does two operations on the image, first it rotated the image and then it scales. This is because when a user wants to scale the image bigger, it leads to more pixels than in the source, this would take the rotate module more time to process the entire image. Thus it is optimal to rotate first and then scale the rotated image.

Image Manipulation

The image manipulation works in the select mode. Once an image is selected it can be rotated or scaled by amounts specified by the user. This module does two operations on the image, first it rotated the image and then it scales. This is because when a user wants to scale the image bigger, it leads to more pixels than in the source, this would take the rotate module more time to process the entire image. Thus it is optimal to rotate first, and scale the rotated image.

Rotation Module

The rotation module is responsible for spinning the image about its center by an amount specified by the user. For the purpose of simplicity, this module only takes values between 0 and 359.



rotval specifies the amount of rotation between 0-359

height,width are the height and width of the image

column and row is the position of the destination pixel

addr is the address generated by the module to get the value of a pixel. This goes to the memory module which sends back the pix(rbg) values.

Pixel is the value of this (column,row) pixel after rotation

The Algorithm

In order to create this in hardware, I first implemented it in software to determine the constraints and straighten out the algorithm for rotation.

1. Nearest Neighbor rotation: This is the basic rotation algorithm that determines the value of the destination image pixel by counter rotation and rounding. If you give it the coordinates of the pixel, it counter rotates these pixels about the center of the image and gives that pixel as output.

   X' = x*cos(rotval) – y*sin(rotval);

Y’ = x*sin(rotval) + y*cos(rotval);

In order to do the rotation we require the sine and cosine of the angle specified. Although verilog allows real floating point number, but it doesn’t allow you to do floating point multiplications for values stored in registers. This could be solved by creating a floating point multiplication module, but I chose lab4’s approach of storing 20 bit values instead. That is, sine and cosine are implemented as lookup tables with each value multiplied by 2^20. After multiplication to get actual value we just perform a 20 signed bit shift right.

2. Sub-pixel Area Average: The issue with nearest neighbor is that pixels are square things and when counter rotated the don’t exactly overlap with other pixels, and rounding of the value produces image artifacts like jagged lines and fuzziness. To prevent this we need to average subpixel area. Basically, we counter rotate a destination pixel and determine its color based on the colors of the 4 pixels it overlaps. The resultant pixel color is the weighted average of those 4 pixels based on the area they cover of this pixel.

   If the first pixel p1 covers a fraction w1 of the destination pixel, and so on, then:
   Pixel = w1*p1 + w2*p2 + w3*p3 + w4*p4

We see that algorithm required to fetch 4 pixels to determine the value of each pixel. Further it requires multiple 32bit signed multiplications, right shifts and other averaging steps. All this cannot be done in one clock cycle. Thus there was a need to pipeline the calculation. In order to determine the complexity of the calculation and the algorithm, I wrote a java program that separated out each calculation and I used a separate timing utility that determine the time it took for each pixel to be generated on my computer, using that I was able to determine that I can do the calculation for each pixel in 13 clock cycles. This includes fetching 4 pixels as well.

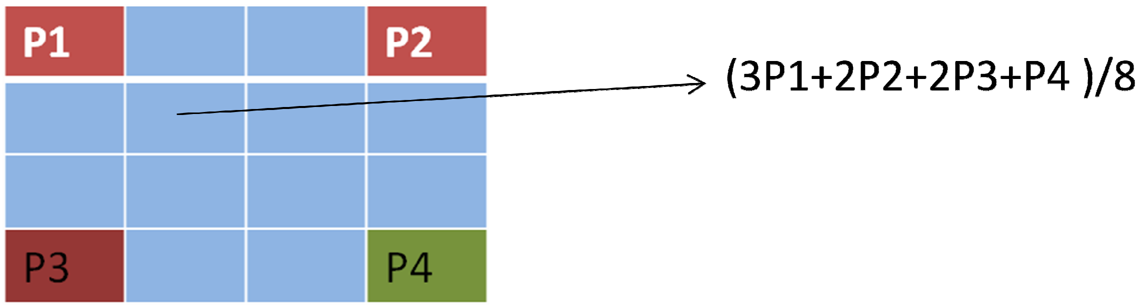
Scale Module

This module is responsible for doing scaling of the image. In order to enlarge the image we need to interpolate it and to shrink it we need anti-aliasing. This module is different in the way that it doesn’t produce pixels sequentially; instead it used a 2x2 block of pixels and produces a 2sx2s block, where s is the scale factor.

In order to verify the algorithm, again I implemented it in software first and you can see that in the java code. Due to integration difficulties with the display module I had to abandon the interpolation implementation and only had a scale module that would stretch out an image without filling in the empty pixels in between. But let’s take a look at the interpolation algorithm

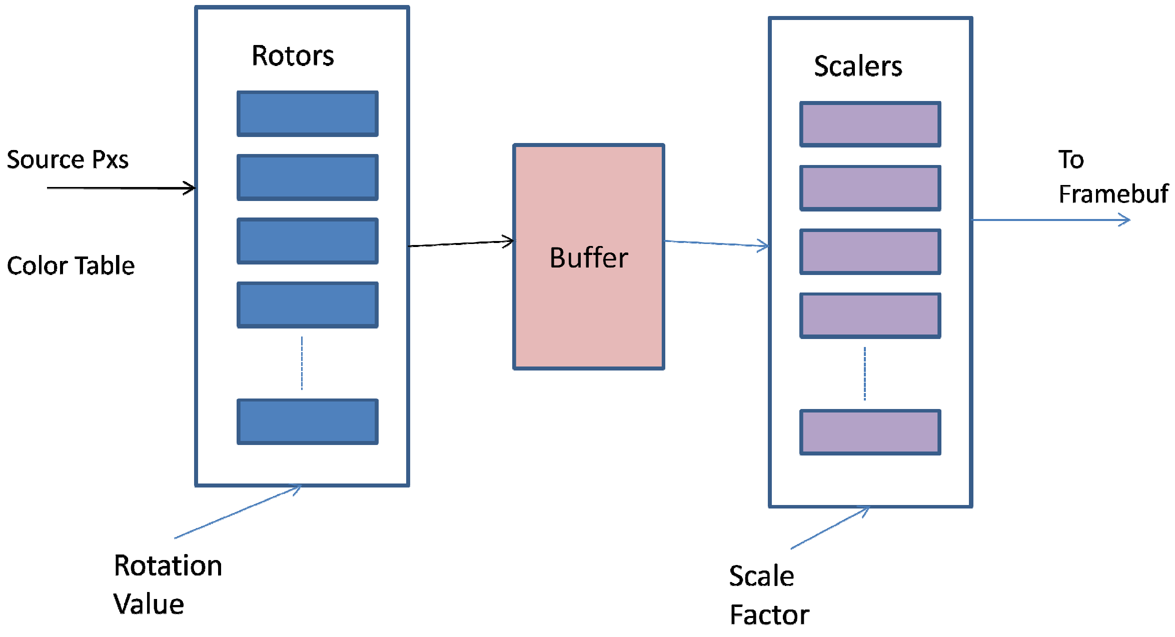
Algorithm

I used bilinear interpolation to scale each 2x2 pixel block. This means that middle pixels are the 2D average of the source pixels. In order to determine the color we compute the weighted sum of the 4 pixels in the source 2x2 block. The weights are the inverse 2D distances of the pixel from each of these 4 pixels.



Pipelining

Both modules need to be linked together and together they take (11+6) = 17 clock cycles. Both fetch 4 pixels each and, the scale module needs to start only after the rotate module had specified the first 2 rows as it requires a 2x2 pixel block to start. Here the pipelining structure proposed:



However, due to integration issue faced, we didn’t get around to including the scaling module, so we didn’t construct the scale pipeline or the buffer. In order to get the rotation module to work, I had to make it a fsm that only did one calculation per clock cycle. This made the rotation module return one pixel per 13 clock cycles. So we had to construct a 13 register pipeline that supplied the hcount and vcount delayed by 13 to 13 different rotate modules in sequence. That is only moduleN receives a signal on the Nth clock cycle. And N cycles from 0-12.

Debugging

Display System

The general approach to debugging in this project was to take small steps and solve problems as they turned up in these small steps. The functionality of the display system was implemented in the following order:

1. Process images in MATLAB
2. Write COE files to BRAM
3. Ensure proper operation of SVGA
4. Display image from BRAM
5. Move image stored in BRAM
6. Write to ZBT memory, display from ZBT memory
7. Write image from BRAM to ZBT
8. Display glitch-free image from ZBT
9. Display multiple images from ZBT
10. Move images using frame buffers
11. Select image

The most common issue throughout the process was timing. Memory reads take time, and they must be synchronized with the rest of the system. This first occurred when trying to display an image stored in BRAM. A 2 clock cycle display was added within that segment of the display module to correct the problem.

Working with the ZBTs was the most challenging part of the debugging process. Writing single colors or color bar patterns to the ZBT was the first step. At first, nothing would write to the ZBT at all. That problem was solved by changing incorrect assignments to the control pins of the RAM.

Once that problem was corrected, it was fairly straightforward to write images from BRAM to the ZBT. Adding multiple images was also not difficult. Scrolling images to the right was also easy, but scrolling to the left was a challenge. After a number of different approaches, the best one was to make x (top left corner horizontal position) and hcount signed numbers so that they did not roll over when they became less than 0.

To debug problems with alternating frame buffers, buttons and switches were used to manually read or write a specific buffer. This allowed us to see what each frame buffer was displaying without having the buffers switch at 60Hz. This turned up issues with the ZBT RAM clock.

To implement the mode switching functionality, switches were used initially to change between images. Then, vertical bars were drawn on the screen (using the display module) to show the threshold values for selecting an image (left side between hcount = 300 and hcount = 400 and right side between hcount = 400 and hcount = 500.

Rotation Module

To debug this module two approaches were used:

1. I used a 3x3 pixel matrix as source image and outputted the rotated image as well as 7 other inner variables to the 16 hex display. The variables displayed were in successive sections of the calculation so that I could see where the calculation would fail and hence correct it.
2. The second method used was a coe file with a 100x100 image that I rotated and printed to another file. Then I compared this file to the image rotated by photoshop and converted to coe.

In order to debug the scale module I just used the COE file approach as the hex display does not have enough values to show. Also the logic analyzer output would have been very difficult to visually verify

However, I did not save my project on another disk and successive versions overwrote the previous ones. Which was a big mistake.

Conclusion

Both subsystems in this project were able to function independently. The display system was able to scroll through images and change from browse to select modes. The graphics engine was able to rotate a matrix of numbers correctly and scale a matrix as well (right? at some point?). Several features of the project could not be implemented due to time constraints. First, capability to download images from the computer directly into labkit memory was not added because there was not enough time to integrate the USB adapter. To add that functionality, the ROMs that stored the images would have needed to be changed to RAMs with no preloaded values. These RAMs would have been populated by incoming data from the USB. There were no technical hurdles to implementing this function, just not enough time.

Integrating the graphics engine with the display also fell victim to deadline constraints. Initial attempts to incorporate rotation modules into the display module failed. The reason is not immediately clear, but was likely either a timing issue or a simple coding error. Again, the problem was a scheduling one rather than a technical one. With a bit more time for debugging, there is every reason to assume that these modules could have worked together to generate the interactive environment that was envisioned at the beginning of the project.

If these integration issues had been worked out, the addition of gesture recognition would have been the next step. Gesture recognition has been used in several previous 6.111 projects, so it is likely that this step also would not have posed any insurmountable technical challenges. Any continued work on this system will smooth out the integration bugs and add this extra functionality to create a user-friendly image viewer.

Verilog

Certain files are missing in the transformations part as I did not save them when I started integrating. Rotate_mod1 is the most recent working version I had. However, we saw it was not compiling in Xilinx due to the fatal error.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.
```

module ZBTdriver

```
(clk, cen, we, addr, write_data, read_data,
                 /*ram_clk,*/ ram_we_b, ram_address, ram_data, ram_cen_b);

  input clk;                     // system clock
  input cen;                     // clock enable for gating ZBT cycles
  input we;                      // write enable (active HIGH)
  input [18:0] addr;             // memory address
  input [35:0] write_data;       // data to write
  output [35:0] read_data;       // data read from memory
  //output        ram_clk;       // physical line to ram clock
  output        ram_we_b;        // physical line to ram we_b
  output [18:0] ram_address;     // physical line to ram address
  inout [35:0]  ram_data;        // physical line to ram data
  output        ram_cen_b;       // physical line to ram clock enable

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

  // create delayed ram_we signal: note the delay is by two cycles!
```

```
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

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

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

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

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

endmodule // zbt_6111


/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used            *
*     solely for design, simulation, implementation and creation of           *
*     design files limited to Xilinx devices or technologies. Use             *
*     with non-Xilinx devices or technologies is expressly prohibited         *
*     and immediately terminates your license.                                *
*                                                                             *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"           *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                 *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION         *
*    AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION             *
*    OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS               *
*    IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                 *
*    AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE        *
*    FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                *
*    WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                 *
*    IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR          *
*    REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF         *
*    INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS         *
*    FOR A PARTICULAR PURPOSE.                                               *
*                                                                            *
*    Xilinx products are not intended for use in life support                *
*    appliances, devices, or systems. Use in such applications are           *
*    expressly prohibited.                                                   *
*                                                                            *
*    (c) Copyright 1995-2006 Xilinx, Inc.                                    *
*    All rights reserved.                                                    *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file image.v when simulating
// the core, image. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module image(
        addr,
        clk,
        dout);


input [16 : 0] addr;
input clk;
output [7 : 0] dout;

// synopsys translate_off
```

```
BLKMEMSP_V6_2 #(
		17,	// c_addr_width
		"0",	// c_default_data
		120000,		// c_depth
		0,	// c_enable_rlocs
		0,	// c_has_default_data
		0,	// c_has_din
		0,	// c_has_en
		0,	// c_has_limit_data_pitch
		0,	// c_has_nd
		0,	// c_has_rdy
		0,	// c_has_rfd
		0,	// c_has_sinit
		0,	// c_has_we
		18,	// c_limit_data_pitch
		"image.mif",	// c_mem_init_file
		0,	// c_pipe_stages
		0,	// c_reg_inputs
		"0",	// c_sinit_value
		8,	// c_width
		0,	// c_write_mode
		"0",	// c_ybottom_addr
		1,	// c_yclk_is_rising
		1,	// c_yen_is_high
		"hierarchy1",	// c_yhierarchy
		0,	// c_ymake_bmm
		"16kx1",		// c_yprimitive_type
		1,	// c_ysinit_is_high
		"1024",// c_ytop_addr
		0,	// c_yuse_single_primitive
		1,	// c_ywe_is_high
		1)	// c_yydisable_warnings
	inst (
		.ADDR(addr),
		.CLK(clk),
		.DOUT(dout),
		.DIN(),
		.EN(),
		.ND(),
		.RFD(),
                .RDY(),
                .SINIT(),
                .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of image is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of image is "black_box"

endmodule


`timescale 1ns / 1ps
```

module im_write (input vclock,

```
                                        reset,
                                        enable,
                                        input [7:0] pixel,   //Pixel from BRAM
                                        output reg [31:0] pix4x, //4 pixels concatenated together (1st
is MSBs, 4th is LSBs)
                                        output reg [18:0] addr); //Memory address, generated
sequentially

        reg [1:0] pixnum = 0;
        reg [7:0] pix1, pix2, pix3;
        always @(posedge vclock) begin
                if (reset) begin
                        pixnum <= 0;
                        addr <= 0;
                end
                else if (enable) begin
                        pixnum <= pixnum+1;
                        case (pixnum)
                                2'b00: pix1 <= pixel;
                                2'b01: pix2 <= pixel;
                        2'b10: pix3 <= pixel;
                        2'b11: begin
                            addr <= addr+1;
        //Increment address
                            pix4x <= {pix1, pix2, pix3, pixel}; //Concatenate pixels
                        end
                    endcase
                end
        end

endmodule


`timescale 1ns / 1ps

//Image display module
```

module display (

```
        input vclock,       // 40MHz clock
  input reset,              // 1 to initialize module
  input left,               // 1 when image should move left
  input right,      // 1 when image should move right
        input mode,     //Either scroll mode (0) or edit mode (1)
  input [3:0] sel, //Select image for rotation/scaling
        input [8:0] scalval,
        input [8:0] rotval,
        input [1:0] speed,  // scroll rate (how many pixels the image moves per frame)
  input [10:0] hcount_unsigned,// horizontal index of current pixel (0..799)
  input [9:0] vcount, // vertical index of current pixel (0..599)
        input [10:0] hc13_unsigned,     // horizontal index of current pixel (0..799)
  input [9:0] vc13_unsigned, // vertical index of current pixel (0..599)
  input hsync,              // XVGA horizontal sync signal (active low)
  input vsync,              // XVGA vertical sync signal (active low)
  input blank,              // XVGA blanking (1 means output black pixel)

  //output write,  //Enables write to ZBT (high is enabled)
        output reg disp_mode,  //0 for browsing, 1 for select
        output reg [1:0] im_sel, //Selected image
        output reg phsync,          //horizontal sync
  output reg pvsync,        //vertical sync
  output reg pblank,        // blanking
```

```
output [17:0] pixel);    // 24-bit RGB pixel truncated to 18);

wire signed [11:0] hcount = hcount_unsigned;
        wire signed [11:0] hc13 = hc13_unsigned;
        wire signed [10:0] vc13 = vc13_unsigned;
        //wire signed [11:0] hcount = hcount_undel - 3;
        //Registers to hold address and pixel values
        //Image 1
        reg [19:0] addr1;
        wire [7:0] pix1;
        wire [7:0] pix_red1;
        wire [7:0] pix_green1;
        wire [7:0] pix_blue1;
        //Image 2
        reg [19:0] addr2;
        wire [7:0] pix2;
        wire [7:0] pix_red2;
        wire [7:0] pix_green2;
        wire [7:0] pix_blue2;
        //Image 3
        reg [19:0] addr3;
        wire [7:0] pix3;
        wire [7:0] pix_red3;
        wire [7:0] pix_green3;
        wire [7:0] pix_blue3;
        //Image 4
        reg [19:0] addr4;
        wire [7:0] pix4;
        wire [7:0] pix_red4;
        wire [7:0] pix_green4;
        wire [7:0] pix_blue4;
        wire [23:0] pix24;

        //Registers to hold location of top left corner of image
        reg signed [11:0] x1;// = 25;
        reg signed [10:0] y1;// = 250;
        reg [10:0] width1 = 200;
        reg [9:0] height1 = 150;
        reg signed [11:0] x2;// = 250;
        reg signed [10:0] y2;// = 250;
        reg [10:0] width2 = 200;
```

```
	reg [9:0] height2 = 150;
	reg signed [11:0] x3;// = 475;
	reg signed [10:0] y3;// = 250;
	reg [10:0] width3 = 200;
	reg [9:0] height3 = 150;
	reg signed [11:0] x4;// = 700;
	reg signed [10:0] y4;// = 250;
	reg [10:0] width4 = 200;
	reg [9:0] height4 = 150;

	reg [1:0] im_disp;
	reg repaint;

	reg mode_del;
	always @(posedge vclock) begin
		mode_del <= mode;
	end

	wire mode_pulse;
	assign mode_pulse = mode_del && ~mode;

	//Generates a pulse which changes on button press, so that pressing the button once
	//will put the system in "select" mode and a second button press returns it to scroll (browse)
mode.
	reg switch = 0;
	always @(posedge vclock) begin
		if (mode == 1 && mode_del == 0) switch <= ~switch;
	end


	//parameter space = 8'd250;
	always @(posedge vclock) begin
		if(reset || repaint)begin
			repaint <= 0;
			x1 <= 0;
			y1 <= 250;
			x2 <= 250; //x1 + space;
			y2 <= 250;
			x3 <= 500; //x1 + space + space;
			y3 <= 250;
			x4 <= 750; //x1 + space + space + space;
```

```
        y4 <= 250;
end

else if (~switch) begin
        if (mode_pulse) repaint <= 1;
        else repaint <= 0;
        if (hcount == 1055 && vcount == 600) begin
                if (left) begin
                        if (x4 >= 300) begin
                                x1 <= x1 - speed;
                                x2 <= x2 - speed;
                                x3 <= x3 - speed;
                                x4 <= x4 - speed;
                        end
                        else begin
                                //{x1, x2, x3, x4} <= {x1, x2, x3, x4};
                                x1 <= x1;
                                x2 <= x2;
                                x3 <= x3;
                                x4 <= x4;
                        end

                end
                else if (right) begin
                        if (x1 <= 300) begin
                                x1 <= x1 + speed;
                                x2 <= x2 + speed;
                                x3 <= x3 + speed;
                                x4 <= x4 + speed;
                        end
                        else begin
                                x1 <= x1;
                                x2 <= x2;
                                x3 <= x3;
                                x4 <= x4;
                        end
                end
                else begin
                        x1 <= x1;
                        x2 <= x2;
                        x3 <= x3;
```

```
					x4 <= x4;
				end
			end
		end
		//Here we change mode to "select" so that only one image is displayed in the center of
the screen
		else if (switch) begin
			repaint <= 0;
			if ((x1 >= 300 && x1 <= 400) || (x1+200 >= 400 && x1+200 <= 500)) begin
				im_disp <= 2'b00;
				x1 <= 300;
			end
			else if ((x2 >= 300 && x2 <= 400) || (x2+200 >= 400 && x2+200 <= 500)) begin
				im_disp <= 2'b01;
				x2 <= 300;
			end
			else if ((x3 >= 300 && x3 <= 400) || (x3+200 >= 400 && x3+200 <= 500)) begin
				im_disp <= 2'b10;
				x3 <= 300;
			end
			else if ((x4 >= 300 && x4 <= 400) || (x4+200 >= 400 && x4+200 <= 500)) begin
				im_disp <= 2'b11;
				x4 <= 300;
			end
		end
	end


	//BRAM that stores image of the adorable panda
	panda cute_panda (.addr(addr1),.clk(vclock),.dout(pix1));

	//Red LUT for panda
	panda_red red1panda (.addr(pix1),.clk(vclock),.dout(pix_red1));

	//Green LUT for panda
	panda_green green1panda (.addr(pix1),.clk(vclock),.dout(pix_green1));

	//Blue LUT for panda
	panda_blue blue1panda (.addr(pix1),.clk(vclock),.dout(pix_blue1));

	//BRAM that stores image of the delicious chocolate cake
	cake nom (.addr(addr2),.clk(vclock),.dout(pix2));

	//Red LUT for cake
	cake_red red1cake (.addr(pix2),.clk(vclock),.dout(pix_red2));

	//Green LUT for cake
	cake_green green1cake (.addr(pix2),.clk(vclock),.dout(pix_green2));

	//Blue LUT for cake
	cake_blue blue1cake (.addr(pix2),.clk(vclock),.dout(pix_blue2));

	//BRAM that stores image of the cute kitteh
	kitten cute_kitten (.addr(addr3),.clk(vclock),.dout(pix3));

	//Red LUT for kitten
	kitten_red red1kitten (.addr(pix3),.clk(vclock),.dout(pix_red3));

	//Green LUT for kitten
	kitten_green green1kitten (.addr(pix3),.clk(vclock),.dout(pix_green3));

	//Blue LUT for kitten
	kitten_blue blue1kitten (.addr(pix3),.clk(vclock),.dout(pix_blue3));

	//BRAM that stores image of the beautiful sunset
	sunset sunset1 (.addr(addr4),.clk(vclock),.dout(pix4));

	//Red LUT for sunset
	sunset_red red1sunset (.addr(pix4),.clk(vclock),.dout(pix_red4));

	//Green LUT for sunset
	sunset_green green1sunset (.addr(pix4),.clk(vclock),.dout(pix_green4));

	//Blue LUT for sunset
	sunset_blue blue1sunset (.addr(pix4),.clk(vclock),.dout(pix_blue4));

	reg show1,show2,show3,show4,show5,show6,show7,show8,ph,pv,pb;
	reg hsync1, vsync1, blank1, hsync2, vsync2, blank2;
	always @(posedge vclock) begin
//			show1 <= ~reset && (hcount >= x && vcount >= y) && (hcount < x+width && vcount <
y+height);
```

```
			show1 <= ~reset && (hcount >= x1 && vcount >= y1) && (hcount < x1+width1 &&
vcount < y1+height1); //&& (~switch || (im_disp == 2'b00));
			show2 <= show1;
			show3 <= ~reset && (hcount >= x2 && vcount >= y2) && (hcount < x2+width2 &&
vcount < y2+height2); //&& (~switch || (im_disp == 2'b01));
			show4 <= show3;
			show5 <= ~reset && (hcount >= x3 && vcount >= y3) && (hcount < x3+width3 &&
vcount < y3+height3); //&& (~switch || (im_disp == 2'b10));
			show6 <= show5;
			show7 <= ~reset && (hcount >= x4 && vcount >= y4) && (hcount < x4+width4 &&
vcount < y4+height4); //&& (~switch || (im_disp == 2'b11));
			show8 <= show7;
			disp_mode <= switch;
			im_sel <= im_disp;
			{hsync1,ph} <= {ph,hsync};
			{vsync1,pv} <= {pv,vsync};
			{blank1,pb} <= {pb,blank};
			{hsync2,ph} <= {ph,hsync1};
			{vsync2,pv} <= {pv,vsync1};
			{blank2,pb} <= {pb,blank1};
			{phsync,ph} <= {ph,hsync2};
			{pvsync,pv} <= {pv,vsync2};
			{pblank,pb} <= {pb,blank2};
		end

//	module scale_mod #(parameter NPIX=9,
//	        parameter WIDTH=3,
//	        parameter HEIGHT=3)
//	       (input reset, clock,
//	 input [8:0] scaleval,
//	 input [5:0] pix_r,
//	 input [5:0] pix_g,
//	 input [5:0] pix_b,
//	 input [12:0] c,r,
//			input [12:0] hcount,vcount,
//	 output [19:0] addr,
//	 output [17:0] pixel);
//addr = (hcount-c)+(vcount-r)*WIDTH;
//pixel <= {pix_r,pix_g,pix_b};
	reg [3:0] pipecount=0;
	reg [11:0] hin0,hin1,hin2,hin3,hin4,hin5,hin6,hin7,hin8,hin9,hin10,hin11,hin12;
reg [10:0] vin0,vin1,vin2,vin3,vin4,vin5,vin6,vin7,vin8,vin9,vin10,vin11,vin12;
always @(posedge vclock) begin
        pipecount <= (pipecount == 12)?0:pipecount + 1;
        case(pipecount)
                0: begin
                        hin0 <= hc13;
                        vin0 <= vc13;
                end
                1: begin
                        hin1 <= hc13;
                        vin1 <= vc13;
                end
                2: begin
                        hin2 <= hc13;
                        vin2 <= vc13;
                end
                3: begin
                        hin3 <= hc13;
                        vin3 <= vc13;
                end
                4: begin
                        hin4 <= hc13;
                        vin4 <= vc13;
                end
                5: begin
                        hin5 <= hc13;
                        vin5 <= vc13;
                end
                6: begin
                        hin6 <= hc13;
                        vin6 <= vc13;
                end
                7: begin
                        hin7 <= hc13;
                        vin7 <= vc13;
                end
                8: begin
                        hin8 <= hc13;
                        vin8 <= vc13;
                end
                9: begin
```

```
                        hin9 <= hc13;
                        vin9 <= vc13;
                    end
                    10: begin
                        hin10 <= hc13;
                        vin10 <= vc13;
                    end
                    11: begin
                        hin11 <= hc13;
                        vin11 <= vc13;
                    end
                    12: begin
                        hin12 <= hc13;
                        vin12 <= vc13;
                    end
                endcase
            end

            reg [11:0] x;
            reg [9:0] y;
            reg [5:0] pix_r, pix_g, pix_b;
            reg [19:0] addr;
            wire [19:0]
addr_0,addr_1,addr_2,addr_3,addr_4,addr_5,addr_6,addr_7,addr_8,addr_9,addr_10,addr_11,addr_12;
            wire [17:0] pix_out0, pix_out1, pix_out2, pix_out3, pix_out4, pix_out5, pix_out6, pix_out7,
pix_out8, pix_out9, pix_out10, pix_out11, pix_out12;


            always @(posedge vclock) begin
                if (switch) begin
                    case (im_disp)
                        2'b00: begin
                            x <= x1;
                            y <= y1;
                            pix_r <= pix_red1[7:2];
                            pix_g <= pix_green1[7:2];
                            pix_b <= pix_blue1[7:2];
                            addr1 <= addr;
                        end
                        2'b01: begin
                            x <= x2;
```

```
                                        y <= y2;
                                        pix_r <= pix_red2[7:2];
                                        pix_g <= pix_green2[7:2];
                                        pix_b <= pix_blue2[7:2];
                                        addr2 <= addr;
                                end
                                2'b10: begin
                                        x <= x3;
                                        y <= y3;
                                        pix_r <= pix_red3[7:2];
                                        pix_g <= pix_green3[7:2];
                                        pix_b <= pix_blue3[7:2];
                                        addr3 <= addr;
                                end
                                2'b11: begin
                                        x <= x4;
                                        y <= y4;
                                        pix_r <= pix_red4[7:2];
                                        pix_g <= pix_green4[7:2];
                                        pix_b <= pix_blue4[7:2];
                                        addr4 <= addr;
                                end
                        endcase
                end
                else begin
                        addr1 <= (hcount-x1)+(vcount-y1)*width1;
                        addr2 <= (hcount-x2)+(vcount-y2)*width2;
                        addr3 <= (hcount-x3)+(vcount-y3)*width3;
                        addr4 <= (hcount-x4)+(vcount-y4)*width4;
                end
        end

        wire d1;
        rotate_mod rot1 (.reset(reset),


        .clock(vclock),


        .rotval(rotval),
.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin0),

.vcount(vin0),

.addr(addr_0),

.pixel(pix_out0));

rotate_mod rot2 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),
.c(x),

.r(y),

.hcount(hin1),

.vcount(vin1),

.addr(addr_1),

.pixel(pix_out1));
rotate_mod rot3 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin2),
.vcount(vin2),

.addr(addr_2),

.pixel(pix_out2));
rotate_mod rot4 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin3),

.vcount(vin3),

.addr(addr_3),

.pixel(pix_out3));
```

```
rotate_mod rot5 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin4),

.vcount(vin4),

.addr(addr_4),

.pixel(pix_out4));

rotate_mod rot6 (.reset(reset),

.clock(vclock),

.rotval(rotval),
.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin5),

.vcount(vin5),

.addr(addr_5),

.pixel(pix_out5));

rotate_mod rot7 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),
.c(x),

.r(y),

.hcount(hin6),

.vcount(vin6),

.addr(addr_6),

.pixel(pix_out6));

rotate_mod rot8 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin7),
.vcount(vin7),

.addr(addr_7),

.pixel(pix_out7));

rotate_mod rot9 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin8),

.vcount(vin8),

.addr(addr_8),

.pixel(pix_out8));
```

```
rotate_mod rot10 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin9),

.vcount(vin9),

.addr(addr_9),

.pixel(pix_out9));

rotate_mod rot11 (.reset(reset),

.clock(vclock),

.rotval(rotval),
.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin10),

.vcount(vin10),

.addr(addr_10),

.pixel(pix_out10));

rotate_mod rot12 (.reset(reset),

.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),
.c(x),

.r(y),

.hcount(hin11),

.vcount(vin11),

.addr(addr_11),

.pixel(pix_out11));

rotate_mod rot13 (.reset(reset)
.clock(vclock),

.rotval(rotval),

.pix_r(pix_r),

.pix_g(pix_g),

.pix_b(pix_b),

.c(x),

.r(y),

.hcount(hin12),
		.vcount(vin12),

		.addr(addr_12),

		.pixel(pix_out12));

//		assign addr1 = (hcount-x1)+(vcount-y1)*width1;
//		assign addr2 = (hcount-x2)+(vcount-y2)*width2;
//		assign addr3 = (hcount-x3)+(vcount-y3)*width3;
//		assign addr4 = (hcount-x4)+(vcount-y4)*width4;
		//Memory output is displayed if within box, black otherwise
//		assign pixel = show2 ? 24'b1111_0000_0000_0000_0000_0000 :
24'b1111_1111_1111_1111_1111_1111;
//
//
//{pix_red[7:2],2'b00,pix_green[7:2],2'b00,pix_blue[7:2],2'b00} :
//						//24'b1111_0000_0000_0000_0000_0000;

		reg [17:0] pix;
		always @(posedge vclock) begin
			if (~switch) begin
				if (show2) pix <= {pix_red1[7:2],pix_green1[7:2],pix_blue1[7:2]};
				else if (show4) pix <= {pix_red2[7:2],pix_green2[7:2],pix_blue2[7:2]};
				else if (show6) pix <= {pix_red3[7:2],pix_green3[7:2],pix_blue3[7:2]};
				else if (show8) pix <= {pix_red4[7:2],pix_green4[7:2],pix_blue4[7:2]};
				else if (hcount == 300 || hcount == 500) pix <=
18'b111_111_111_111_111_111;
				else pix <= 18'b011_000_000_000_000_000;
			end
			else begin
				case(pipecount)
					0: begin
						pix <= pix_out0;
						addr <= addr_0;
					end
					1: begin
						pix <= pix_out1;
						addr <= addr_1;
```

```
end
2: begin
        pix <= pix_out2;
        addr <= addr_2;
end
3: begin
        pix <= pix_out3;
        addr <= addr_3;
end
4: begin
        pix <= pix_out4;
        addr <= addr_4;
end
5: begin
        pix <= pix_out5;
        addr <= addr_5;
end
6: begin
        pix <= pix_out6;
        addr <= addr_6;
end
7: begin
        pix <= pix_out7;
        addr <= addr_7;
end
8: begin
        pix <= pix_out8;
        addr <= addr_8;
end
9: begin
        pix <= pix_out9;
        addr <= addr_9;
end
10: begin
        pix <= pix_out10;
        addr <= addr_10;
end
11: begin
        pix <= pix_out11;
        addr <= addr_11;
end
```

```
                                        12: begin
                                                  pix <= pix_out12;
                                                  addr <= addr_12;
                                        end
                              endcase
                    end
          end

          assign pixel = pix;

          //assign pixel = show2 ? {pix_red[7:2],pix_green[7:2],pix_blue[7:2]} :
18'b011_000_000_000_000_000;
Endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:09:37 11/30/2009
// Design Name:
// Module Name:    delayN
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
```

module delayN(clk,in,out);

```
  input clk;
  input in;
  output out;

  parameter NDELAY = 3;
```

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

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

endmodule // delayN


`timescale 1ns / 1ps
```

module buff_write (input vclock,

```
                                                    reset,
                                                    //enable,
                                                    input [10:0] hcount,
                                                    input [9:0] vcount,
                                                    input [17:0] pixel,  //Pixel from BRAM
                                                    output reg [35:0] pix2x, //2 pixels concatenated together (1st
is MSBs, 2nd is LSBs)
                                                    output reg [18:0] addr); //Memory address, generated
sequentially


          reg [17:0] pix1;
          always @(posedge vclock) begin
                    if (reset || (hcount == 1055 && vcount == 627)) begin
                              addr <= 0;
                    end
                    else if ((hcount >= 0 && hcount <= 799 && vcount >= 0 && vcount <= 599)) begin
                              if (hcount[0]== 0) begin
                                        pix1 <= pixel; //{hcount[10:5],hcount[10:5],hcount[10:5]};
          //Store 1st incoming pixel in register, wait for next
                                        end
                              else begin                                                  //When two pixels are
available, generate address and output
                                        addr <= addr+1;                    //Increment address
                                        pix2x <= {pix1, pixel}; //hcount[10:5],hcount[10:5],hcount[10:5]};
//36'b111_111_000_000_000_000_111_111_000_000_000_000; //{pix1, pixel}; //Concatenate pixels
                              end
                    end
```

```
            end

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
```

```
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//          "disp_data_out", "analyzer[2-3]_clock" and
//          "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//          actually populated on the boards. (The boards support up to
//          256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//          value. (Previous versions of this file declared this port to
//          be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//          actually populated on the boards. (The boards support up to
//          72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////


//this file is to test rotate and scale modules
```

module project_lab

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

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

            tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
            tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
            tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
            tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
            tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
            tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
            tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

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

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

            clock_feedback_out, clock_feedback_in,

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

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

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

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

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

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

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

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

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

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

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

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

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

               input  clock_feedback_in;
               output clock_feedback_out;

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

               output rs232_txd, rs232_rts;
```

```
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

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

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

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
```

```
   assign ram1_bwe_b = 4'hF;
   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

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

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

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

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

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

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

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

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

////////////////////////////////////////////////////////////////////////////
//
// lab5 : a simple pong game
//
////////////////////////////////////////////////////////////////////////////

// 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));

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

  // ENTER button is user reset
  wire reset,user_reset;
  debounce db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
  assign reset = user_reset | power_on_reset;

 wire up,down,left,right,mode;
 debounce db6(.reset(reset), .clock(clock_65mhz), .noisy(~button_right), .clean(right));
 debounce db2(.reset(reset), .clock(clock_65mhz), .noisy(~button_left), .clean(left));
 debounce db3(.reset(reset), .clock(clock_65mhz), .noisy(~button_down), .clean(down));
 debounce db4(.reset(reset), .clock(clock_65mhz), .noisy(~button_up), .clean(up));
 debounce #(.DELAY(650000)) db5(.reset(reset), .clock(clock_65mhz), .noisy(~button0), .clean(mode));

 wire [7:0] s; //syncd switches
 synchronize sync0(.clk(clock_27mhz),.in(switch[0]), .out(s[0]));
 synchronize sync1(.clk(clock_27mhz),.in(switch[1]), .out(s[1]));
 synchronize sync2(.clk(clock_27mhz),.in(switch[2]), .out(s[2]));
 synchronize sync3(.clk(clock_27mhz),.in(switch[3]), .out(s[3]));
 //synchronize sync4(.clk(clock_27mhz),.in(switch[4]), .out(s[4]));
 //synchronize sync5(.clk(clock_27mhz),.in(switch[5]), .out(s[5]));
 //synchronize sync6(.clk(clock_27mhz),.in(switch[6]), .out(s[6]));
 //synchronize sync7(.clk(clock_27mhz),.in(switch[7]), .out(s[7]));

 //rotation value in degrees, scale value in percentage
 wire [8:0] rotval, scaleval;
 param_values #(.NCLOCKS(10000000)) pval(.reset(reset), .clock(clock_65mhz), .up(up), .down(down),
.left(left), .right(right),
                 .rotate(rotval), .scale(scaleval));

 wire [19:0] addr;
 wire [17:0] pix;
 wire done;
 wire [35:0] debug;
 rotate_mod rotimpl
                   (.reset(reset), .clock(clock_65mhz),
        .rotval(rotval),
        .pix_r(1),
        .pix_g(1),
        .pix_b(1),
          .height(3),.width(3),
          .c({s[1],s[0]}),.r({s[3],s[2]}),
          .addr(addr),
          .pixel(pix),
          .calc_done(done),
          .debug(debug));


  display_16hex hex_disp(.reset(reset), .clock_27mhz(clock_65mhz), .data(debug),
                        .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
.disp_ce_b(disp_ce_b),
                        .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));


   // 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;

   assign led = ~{2'b00,reset,done,up,down,left,right};

endmodule
```

module rotate_mod

```
                   (input reset, clock,
          input [8:0] rotval,
          input [5:0] pix_r,
          input [5:0] pix_g,
          input [5:0] pix_b,
          input [10:0] height,width,
          input [10:0] c,r,
        output reg [19:0] addr,
        output reg [17:0] pixel,
        output reg calc_done,
        output reg [35:0] debug);

reg signed [9:0] p,q;
reg [3:0] sourcepix, destpix;
reg signed[10:0] x0,y0,x1,y1;


reg signed [9:0] center_x;
reg signed [9:0] center_y;

wire signed[32:0] cosval,sinval;
sine90 sinimpl(.angle(rotval), .sval(sinval));
cos90 cosimpl(.angle(rotval), .cval(cosval));

//temporary regs used to pipeline
reg signed[43:0] t1,t11,t2,t21,t3,t4;

reg [6:0] calc_count;
reg [8:0] prev_rotval;
reg [10:0] prev_c,prev_r;
reg pix_exists;

reg [3:0] ctr;
assign count = ctr;
always @(posedge clock) begin
  prev_rotval <= rotval;
  prev_r <= r;
  prev_c <= c;
  if (reset) begin
    calc_done <= 0;
    calc_count <= 0;
  end
  else begin
    if((rotval!=prev_rotval)||(r!=prev_r)||(c!=prev_c)) begin
      calc_done <= 0;
      calc_count <= 0;
    end
```

```
//1st clk cycle
x0 <= c- width/2;
y0 <= r- height/2;
debug[7:0] <= {c[3:0],r[3:0]};

//2st clk cycle
t1 <= (x0*cosval);
t2 <= (y0*sinval);
x1 <= x0;
y1 <= y0;


//3nd clk
t3 <= (x1*sinval);
t4 <= (y1*cosval);
t11 <= t1;
t21 <= t2;
debug[15:8] <= {sinval[19:16],cosval[19:16]};

//4nd clk
p <= (t11 - t21)>>>20;
q <= (t3 + t4)>>>20;

//5rd clk
p <= p + width/2;
q <= q + height/2;

//6th clock
//ensure p,q is withing picture
if(p<0 || p>=width || q<0 || q>=height)
  pix_exists <= 0;
else begin
  addr <= q*width + p;
  pix_exists <= 1;
end
debug[23:16] <= {p[3:0],q[3:0]};


//7th clock
if(pix_exists)
  pixel <= {pix_r,pix_g,pix_b};
```

```
      else
        pixel <= 18'b000000_111111_000000; //send green for bgcolor

      calc_count <= calc_count + 1;
      calc_done <= (calc_count>=6);
      debug[27:24] <= {3'b000, calc_count};
    end

  end
endmodule
```

```
`timescale 1ns / 1ps
//20bit sin table, that is 2^20 * sin(val)
```

module sine90

```
(input wire [8:0] angle, output reg signed[32:0] sval);
  reg [8:0] temp1,temp2;
  reg sign;
  reg signed[32:0] val;
  always @ (angle) begin
    sign = 0;
    if(angle > 180) begin
      temp1 = angle - 9'd180;
      sign = 1;
    end
    else temp1 = angle;
    if(temp1 > 90) begin
      temp2 = 9'd180 - temp1;
    end
    else
      temp2 = temp1;
    case (temp2)
      9'd0: val=32'd0;
      9'd1: val=32'd18300;
      9'd2: val=32'd36595;
      9'd3: val=32'd54878;
      9'd4: val=32'd73145;
      9'd5: val=32'd91389;
```

```
9'd6: val=32'd109606;
9'd7: val=32'd127789;
9'd8: val=32'd145934;
9'd9: val=32'd164033;
9'd10: val=32'd182083;
9'd11: val=32'd200078;
9'd12: val=32'd218011;
9'd13: val=32'd235878;
9'd14: val=32'd253673;
9'd15: val=32'd271391;
9'd16: val=32'd289027;
9'd17: val=32'd306574;
9'd18: val=32'd324028;
9'd19: val=32'd341383;
9'd20: val=32'd358634;
9'd21: val=32'd375776;
9'd22: val=32'd392803;
9'd23: val=32'd409711;
9'd24: val=32'd426494;
9'd25: val=32'd443147;
9'd26: val=32'd459665;
9'd27: val=32'd476044;
9'd28: val=32'd492277;
9'd29: val=32'd508360;
9'd30: val=32'd524288;
9'd31: val=32'd540057;
9'd32: val=32'd555661;
9'd33: val=32'd571095;
9'd34: val=32'd586356;
9'd35: val=32'd601438;
9'd36: val=32'd616338;
9'd37: val=32'd631049;
9'd38: val=32'd645568;
9'd39: val=32'd659890;
9'd40: val=32'd674012;
9'd41: val=32'd687928;
9'd42: val=32'd701634;
9'd43: val=32'd715127;
9'd44: val=32'd728402;
9'd45: val=32'd741455;
9'd46: val=32'd754282;
```

```
9'd47: val=32'd766880;
9'd48: val=32'd779244;
9'd49: val=32'd791370;
9'd50: val=32'd803256;
9'd51: val=32'd814897;
9'd52: val=32'd826289;
9'd53: val=32'd837430;
9'd54: val=32'd848316;
9'd55: val=32'd858943;
9'd56: val=32'd869309;
9'd57: val=32'd879410;
9'd58: val=32'd889243;
9'd59: val=32'd898805;
9'd60: val=32'd908093;
9'd61: val=32'd917105;
9'd62: val=32'd925838;
9'd63: val=32'd934288;
9'd64: val=32'd942454;
9'd65: val=32'd950333;
9'd66: val=32'd957922;
9'd67: val=32'd965219;
9'd68: val=32'd972223;
9'd69: val=32'd978930;
9'd70: val=32'd985339;
9'd71: val=32'd991448;
9'd72: val=32'd997255;
9'd73: val=32'd1002758;
9'd74: val=32'd1007956;
9'd75: val=32'd1012847;
9'd76: val=32'd1017429;
9'd77: val=32'd1021701;
9'd78: val=32'd1025662;
9'd79: val=32'd1029311;
9'd80: val=32'd1032646;
9'd81: val=32'd1035666;
9'd82: val=32'd1038371;
9'd83: val=32'd1040760;
9'd84: val=32'd1042832;
9'd85: val=32'd1044586;
9'd86: val=32'd1046022;
9'd87: val=32'd1047139;
```

```
   9'd88: val=32'd1047937;
   9'd89: val=32'd1048416;
   9'd90: val=32'd1048576;
   default: val = 0;
  endcase
  sval = val * (sign?-1:1);
 end
endmodule
```

```
`timescale 1ns / 1ps
//20 bit cosine table
```

module cos90

```
(input wire [8:0] angle, output reg signed[32:0] cval);
 reg [8:0] temp1,temp2;
 reg sign;
 reg signed[32:0] val;
 always @ (angle) begin
  sign = 0;
  if(angle > 180) begin
   temp1 = angle - 9'd180;
   sign = 1;
  end
  else temp1 = angle;
  if(temp1 > 90) begin
   temp2 = 9'd180 - temp1;
   sign = ~sign;
  end
  else
   temp2 = temp1;
  case (temp2)
   9'd0: val=32'd1048576;
   9'd1: val=32'd1048416;
   9'd2: val=32'd1047937;
   9'd3: val=32'd1047139;
   9'd4: val=32'd1046022;
   9'd5: val=32'd1044586;
   9'd6: val=32'd1042832;
   9'd7: val=32'd1040760;
   9'd8: val=32'd1038371;
```

```
9'd9: val=32'd1035666;
9'd10: val=32'd1032646;
9'd11: val=32'd1029311;
9'd12: val=32'd1025662;
9'd13: val=32'd1021701;
9'd14: val=32'd1017429;
9'd15: val=32'd1012847;
9'd16: val=32'd1007956;
9'd17: val=32'd1002758;
9'd18: val=32'd997255;
9'd19: val=32'd991448;
9'd20: val=32'd985339;
9'd21: val=32'd978930;
9'd22: val=32'd972223;
9'd23: val=32'd965219;
9'd24: val=32'd957922;
9'd25: val=32'd950333;
9'd26: val=32'd942454;
9'd27: val=32'd934288;
9'd28: val=32'd925838;
9'd29: val=32'd917105;
9'd30: val=32'd908093;
9'd31: val=32'd898805;
9'd32: val=32'd889243;
9'd33: val=32'd879410;
9'd34: val=32'd869309;
9'd35: val=32'd858943;
9'd36: val=32'd848316;
9'd37: val=32'd837430;
9'd38: val=32'd826289;
9'd39: val=32'd814897;
9'd40: val=32'd803256;
9'd41: val=32'd791370;
9'd42: val=32'd779244;
9'd43: val=32'd766880;
9'd44: val=32'd754282;
9'd45: val=32'd741455;
9'd46: val=32'd728402;
9'd47: val=32'd715127;
9'd48: val=32'd701634;
9'd49: val=32'd687928;
```

```
9'd50: val=32'd674012;
9'd51: val=32'd659890;
9'd52: val=32'd645568;
9'd53: val=32'd631049;
9'd54: val=32'd616338;
9'd55: val=32'd601438;
9'd56: val=32'd586356;
9'd57: val=32'd571095;
9'd58: val=32'd555661;
9'd59: val=32'd540057;
9'd60: val=32'd524288;
9'd61: val=32'd508360;
9'd62: val=32'd492277;
9'd63: val=32'd476044;
9'd64: val=32'd459665;
9'd65: val=32'd443147;
9'd66: val=32'd426494;
9'd67: val=32'd409711;
9'd68: val=32'd392803;
9'd69: val=32'd375776;
9'd70: val=32'd358634;
9'd71: val=32'd341383;
9'd72: val=32'd324028;
9'd73: val=32'd306574;
9'd74: val=32'd289027;
9'd75: val=32'd271391;
9'd76: val=32'd253673;
9'd77: val=32'd235878;
9'd78: val=32'd218011;
9'd79: val=32'd200078;
9'd80: val=32'd182083;
9'd81: val=32'd164033;
9'd82: val=32'd145934;
9'd83: val=32'd127789;
9'd84: val=32'd109606;
9'd85: val=32'd91389;
9'd86: val=32'd73145;
9'd87: val=32'd54878;
9'd88: val=32'd36595;
9'd89: val=32'd18300;
9'd90: val=32'd0;
```

```
      default: val=0;
    endcase
    cval = val * (sign?-1:1);
  end
endmodule
```

module scale_mod

```
#(parameter NPIX=9,
                  parameter WIDTH=3,
                  parameter HEIGHT=3)
                    (input reset, clock,
           input [8:0] scaleval,
           input [5:0] pix_r,
           input [5:0] pix_g,
           input [5:0] pix_b,
           input [10:0] c,r,
           output [19:0] addr,
           output [19:0] dest_addr,
           output [17:0] pixel);

  reg signed[11:0] x1,x2,y1,y2;
  reg signed[11:0] c_x,c_y;

  reg [17:0] pix1,pix2,pix3,pix4;

  c_x = WIDTH/2;
  c_y = HEIGHT/2;

  reg done;
  always @(posedge clock) begin
    prev_scaleval <= scaleval;
    if (reset || (scaleval!=prev_scaleval)) begin
      calc_done <= 0;
      calc_count <= 0;
      done <= 0;
    end
    else if(!done) begin

      //pipeline calculations for real numbers
      calc_count <= calc_count+1;
      done <= (calc_count==7);

      if (calc_count==0) begin
        x1<= ((scaleval)*(c-c_x))+c_x;
        y1<= ((scaleval)*(r-c_y))+c_y;
        x2<= ((scaleval)*(c+1-c_x))+c_x;
        y2<= ((scaleval)*(r+1-c_1))+c_y;
      end

      if (calc_count==1) begin
        addr <= (WIDTH*r)+c;
      end

      if (calc_count==2) begin
        pix1 <= {pix_r,pix_g,pix_b};
        addr <= (WIDTH*r)+c+1;
      end

      if (calc_count==3) begin
        pix2 <= {pix_r,pix_g,pix_b};
        addr <= (WIDTH*r+1)+c;
      end

      if (calc_count==4) begin
        pix3 <= {pix_r,pix_g,pix_b};
        addr <= (WIDTH*r+1)+c+1;
      end

      if (calc_count==5) begin
        pix4 <= {pix_r,pix_g,pix_b};
      end

      if(calc_count==6) begin
        dest_addr <= (y1*WIDTH)+x1;
        pixel <= pix1;
      end
    end
  end
endmodule
```

//generates rotation and scale values

module param_values

```
#(parameter NCLOCKS=50000000)   // .01 sec with a 65Mhz clock
                    (input reset, clock, up, down, left, right,
                     output reg [8:0] rotate,
          output reg [8:0] scale);

  reg [26:0] count;

  always @(posedge clock)
    if (reset) begin
      rotate <= 90;
      scale <= 100;
      count <= 0;
    end
    //when count reaches spec # of clock cycles
    else if (count >= NCLOCKS) begin

      //make rotation value between 0,359
      //use left, right to change in 1 degree steps
      //if rot=360, set rot=0
      //if rot goes negative cycle back to 359
      if(rotate == 359)
        rotate <= left? 0: (right? 358: rotate);
      else if(rotate == 0)
        rotate <= left? 1: (right? 359: rotate);
      else
        rotate <= left? (rotate+1): (right? (rotate-1):rotate);

      //make scale value between 10% and 300%
      //use up/down to change in 10% steps
      if(scale == 300)
        scale <= up? 300: (down? 290: scale);
      else if(scale == 10)
        scale <= up? 20: (down? 10: scale);
      else
        scale <= up? (scale+10): (down? (scale-10):scale);

      //set count back to zero to restart counting
      count <= 0;
    end
    else
      count <= count+1;

endmodule
```

module rotate_mod

```
#(parameter NPIX=9,
                parameter WIDTH=3,
                parameter HEIGHT=3)
                  (input reset, clock,
         input [8:0] rotval,
         input [35:0] image,
         output reg [35:0] outimage,
         output alldone);

  reg fetched;
  reg [3:0] pix1,pix2,pix3,pix4;
  reg signed [9:0] r,c,p,q;
  reg [3:0] sourcepix, destpix;
  reg signed[32:0] x,y;


  reg signed [9:0] center_x = WIDTH/2;
  reg signed [9:0] center_y = HEIGHT/2;

  reg signed[32:0] cosval,sinval;
  sin sinimpl(.angle(rotval), .val(sinv));
  cos cosimpl(.angle(rotval), .val(cosv));

  //temporary regs used to pipeline
  reg signed[32:0] t1,t2;

  reg calc_done,done;
  reg [3:0] calc_count;
  reg [8:0] prev_rotval;
  assign alldone = done;

  always @(posedge clock) begin
    prev_rotval <= rotval;
```

```
if (reset || rotval!=prev_rotval) begin
  r <= 0;
  c <= 0;
  x <= -center_x;
  y <= -center_y;
  calc_done <= 0;
  calc_count <= 0;
  outimage <= {36{1'b1}};
  done <= 0;
end
else if(!done)
  if(calc_done) begin
    outimage[7:4] <= calc_count;
    calc_done <= 0;
    calc_count <= 0;
    if(c==WIDTH-1) begin
      c <= 0;
      x <= -center_x;
      if(r == HEIGHT-1)
        done <= 1;
      else begin
        r <= r+1;
        y <= r+1 - center_y;
      end
    end
    else begin
      c <= c+1;
      x <= c+1 - center_x;
    end
  end
  //pipeline calculations for real numbers
  else case(calc_count)
  4'd0: begin
    t1 <= (x*cosval);
    calc_count <= calc_count + 1;
    outimage[7:4] <= calc_count;
  end
  4'd1: begin
    t2 <= (y*sinval);
    calc_count <= calc_count + 1;
    outimage[7:4] <= calc_count;
```

```
end
4'd2: begin
  if((t1-t2)==0)
    p <= 0;
  else
    p <= (t1 - t2)>>>20;
  calc_count <= calc_count + 1;
  outimage[7:4] <= calc_count;
end
4'd3: begin
  t1 <= (x*sinval);
  calc_count <= calc_count + 1;
  outimage[7:4] <= calc_count;
end
4'd4: begin
  t2 <= (y*cosval);
  calc_count <= calc_count + 1;
  outimage[7:4] <= calc_count;
end
4'd5: begin
  if((t1+t2)==0)
    q <= 0;
  else
    q <= (t1 + t2)>>>20;
  calc_count <= calc_count + 1;
  outimage[7:4] <= calc_count;
end
4'd6: begin
  p <= p + center_x;
  q <= q + center_y;
  destpix <= $unsigned(r*WIDTH + c);
  calc_count <= calc_count + 1;
  outimage[7:4] <= calc_count;
end
4'd7: begin
  if(p<0 || p>=WIDTH || q<0 || q>=HEIGHT) begin
    calc_done <= 1; //skip further calculation
  end
  else begin
    sourcepix <= $unsigned(q*WIDTH + p);
    calc_count <= calc_count + 1;
```

```
        end
        outimage[7:4] <= calc_count;
      end
      4'd8: begin
        if(sourcepix<NPIX && destpix<NPIX) begin
          outimage[3:0]<= sourcepix;
        end
        calc_count <= calc_count + 1;
        outimage[7:4] <= calc_count;
      end
      4'd9: begin
        calc_done <= 1;
        outimage[7:4] <= calc_count;
        //outimage <= outimage << 1;
      end
      default: calc_done <= 1;
      endcase
  end
endmodule
```