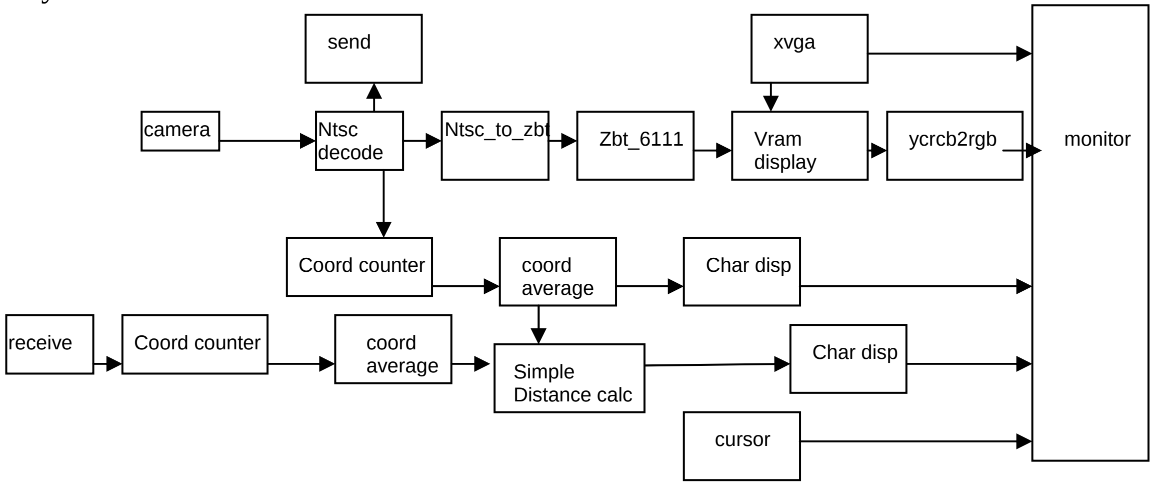
Photogrammetry system

Miles Colman
TA Willie Sanchez

The intent of the project was to create a system that would display the position of an object based on information from two cameras. The actual implementation of the system only advanced as far as calculating the centroid of a collection of pixels in a single image, with crosshairs indicating the centroid of the pixel collection, and a coordinate display that showed the position of a separate cursor, to be used in calibration. I added no FSMs to the design, so the code essentially works on two data streams, the stream of data from the NTSC decoder and the stream of data from the xvga display driver. Transmission and reception of data from a labkit was successfully implemented, but not successfully combined with the camera data stream.



Data processing between NTSC input and display on screen:
The NTSC input is read in at a slow external clock (less than 32 mhz) determined by the camera. The input is then synchronized to the system clock (65 mhz) and decoded to YCrCb data, frame and sync signals, and a data valid signal, by Javier’s module, ntsc decode. The YCrCb data is then sent to the zbts by ntsc_to_zbt . ntsc_to_zbt takes 18 bit ycrcb data and stores two 18 bit pixels per 36 bit memory address. The write enable is the LSB of the horizontal count signal. The zbt stores 36 bits of data per address, so the pixel values for hcount[0] and hcount[1] are stored at the same address. The YCrCb data is read out when the LSB of the horizontal count value of the xvga output is 0. The data stored at a given address is decoded into two 16-bit ycrcb pixel values for hcount[0] and hcount[1], which are sent to a ycrcb-rgb converter module by xilinx. The sync signals are delayed 3 cycles to account for the delay introduced by the colorspace conversion. The output of the rgb converter is then OR’d together with data to be displayed on the screen. Due to problems using the OR, the only data which is displayed on screen is the camera output.

Luminance data from the NTSC decoder is used in the distance calculator. The Y data from the ntsc input stream is sent to a coordinate processing module, which checks if the Y value exceeds a threshold luminance value. The threshold is adjusted based on the total number of pixels exceeding the threshold in order to always have about a thousand pixels from which to calculate the centroid.

The threshold adjustment mechanism is very simple, and decrements the threshold by one hundred (luminance is 10 bit, so the max is 1023) if the tally exceeds 2000 pixels, and increments the threshold a hundred pixels if the tally is less than 2000 pixels. Since the adjustment is very coarse, it is easy to get into a situation in which the threshold alternates between a high value at which no pixels are displayed, and a low value at which more than 2000 pixels are displayed. Some form of PID control could be used to set the threshold based on the tally to get a more robust system.

If the luminance value of the NTSC pixel in the coordinate processing module exceeds the luminance threshold, the pixel is included in the pixels that have their centroid calculated. When the luminance value exceeds the threshold, the pixel's x and y values are added to the x and y totals for the NTSC frame, and the tally of pixels is incremented. When the NTSC v-sync signal goes high, the totals are registered as inputs to a divider (by xilinx). The x-total, y-total, and tally are reset on the next clock cycle. Since the division only happens once every v-sync signal on the input, the division has plenty of time to take place. The average x and y values from the calculation are displayed on the screen and input to the distance calculator module.

Distance Calculation:

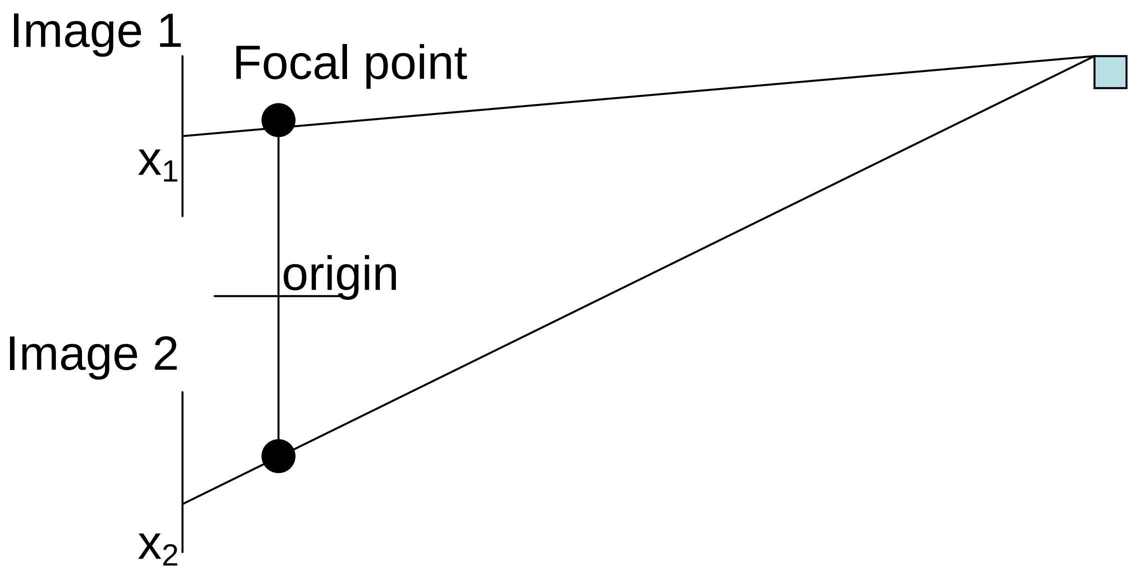


Fig 1: Simple distance calculation geometry [after Horn, *Robot Vision*].

Given the image coordinates from the two images, the difference between the coordinates can be calculated. The difference will be large for close objects, and 0 when the distance between the screens compared to the distance to the object is small. The difference in positions is proportional to the focal length and the distance between the focii (which is equal to the baseline between the cameras if the optical axes are equal). The z distance can be found using similar triangles and knowledge of the baseline and focal length. Since the focal length of the camera was unknown, this coefficient was to be determined after setup, and then hard coded in. The distance calculation module takes in the x and y average positions and calculates the position of the object, using the following formulae:

$X = \text{baseline} * (x1 + x2)/(2*(x1 - x2)$

$Z = \text{baseline} * \text{focal length}/(x1 - x2)$

Y is calculated in the same way as X, by using Y data instead of X coordinates in the equations above. The distance can also be calculated using the Y position information, but the current simple_distance_calc module uses only x. There are more robust ways of calculating position that refer the position to an external reference frame, described in *Robot Vision*, one of which I also coded up. The improved method uses a calibrated affine transform to allow more freedom in camera position, non-square pixels, rotation of camera axes relative to one another, and does not require knowledge of the focal length, but this method would have required determination of 10 calibration coefficients and since the baseline was not made functional, no more work was done on this method.

The coordinate processing module also outputs a pixel which turns pink for pixels that exceed a threshold luminance value and is green when the pixels do not exceed the threshold luminance value. In the labkit, the data written to the zbt for display can be switched to the processed pixel. The display will then show the pixels that exceed the threshold brightness. Since processed pixel generation is done with muxes and comparators, no delay of the signal is needed when the switch is made. Finally, the average pixel value is compared to the video ram address and used to show crosshairs highlighting the centroid of the bright regions of the image.

Since two camera inputs are needed, and the labkit only allows connection of one composite video device at a time, transmit/receive modules were made to allow the processing of two data streams. The transmit module takes a 30 bit input frame at the tv_line_in_clock1 rate and sends out two 15 bit frames at the 65 MHz system clock rate. The connection used is the analyzer port, one of which was redeclared as an input. The 16th bit of the analyzer is reserved for the frame number which is 0 for the first 15 bits of the 30 bit input and 1 for the 2nd 15 bits of the 30 bit input. The receive module synchronizes the 16 bit input to the local system clock (65 mhz) and then checks the LSB of the received data. If the LSB is 0, the 15 bit frame is registered. If the LSB is 1, the current frame is set as the 15 higher hits of the output, while the lower 15 bits come from the register. This system only allows one-way communication, and is only adequate for transmitting display data at 65 MHz. Comparisons of the quality of the transmitted image using a device showed that the transmission introduces some noise, but most of the pixels (>90%) look the same. Since the noise is regularly distributed in the image, the transmitted luminance data could still be used to calculate the centroid of the image for stereo image processing.

Other inputs to the video DAC include the char_disp module, which is provided to display average positions and calculated distance information on the screen. The only other functionality not described above is the cursor, which is a movable cross on the camera image that displays its coordinates on screen as well. The cursor currently has no functionality to the system but would be used by a person to calibrate the system.

Status of the project: crosshairs, distance information, and brightness images were all displayed on screen at high point of the project. The current state of the project is that brightness displays, due probably to some coding mistakes in the top level module. Receiving/transmission of data is very simple, and was working before demonstration, but stopped working, probably due to coding mistakes or unassigned in/outs.

Implementation:

I had planned to implement the zbts first, as this was the hardest part of the project. I thought once I was reading and writing the zbts, the rest would be easy, but I didn't leave enough time at the end, since I just ended up modifying the zbt sample code. The coordinate processing was relatively straightforward, although getting things to display on the screen reliably gave me some trouble.

Debugging:

For a long time, (~2 weeks) the zbts were unusable. Eventually, when code demonstrating working zbts was released, I realized that I had misunderstood the enable bits based on the naming convention used in the labkit and online documentation: I had been setting them low when they should have been high. The logic analyzer became much more useful to me after implementing a gated clock to allow collection of up to 50 ms of data to be collected. Originally the dividers from core-gen seemed to be broken. However, I tested the dividers on the labkit using the switches as inputs, displaying the output on the hex display, and they worked. Using the analyzer with the gated clock, I figured out that the inputs to the dividers were not the registered versions of the total x, total y, and tally, which was causing the dividers to produce meaningless results due to being run too fast to have their inputs switched every clock cycle. The logic analyzers were also used to check the functioning of the ram and write-enable bits, since I had tried to drive the zbts too fast after seeing the code that was released.

The hex displays were very useful for debugging, as the average coordinate results were working on the hex display before they worked on the monitor. I had been using the wrong address bits to compare to the average pixels, which became clear after checking the hex display. The current code shows a working average on the hex display, but not on the screen, due to a failure to save old versions of the code.

Lessons learned:

It became obvious during the presentation that I should maintain old code and bitfiles, since I made a lot of backwards progress in the last few days. A more basic lesson was that using someone else's zbt access modules in the context of their ntsc to ram and ram to display modules is equivalent to a big design change, and that I should make block diagrams for all the things I plan on using, not just the ones I modify myself. The final system block diagram is very different from the one presented at the design review, and updating it in a timely fashion would have helped a lot.

Appendix: Verilog code: highest level module

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.

//modified by miles colman

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
```

```
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

               clock_feedback_out, clock_feedback_in,

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

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

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

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data;

        input [15:0] analyzer4_data;
        output analyzer4_clock;
        output analyzer1_clock, analyzer2_clock;
        input analyzer3_clock;

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

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input

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

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

   // SRAMs

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

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

/* enable RAM pins */

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

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

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

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

   // Logic Analyzer
   //assign analyzer1_data = 16'h0;
   //assign analyzer1_clock = 1'b1;
   //assign analyzer2_data = 16'h0;
   //assign analyzer2_clock = 1'b1;
   //assign analyzer3_data = 16'h0;
   //assign analyzer3_clock = 1'b1;
/*   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;  */
        //ANALYZER4 IS NOW AN INPUT

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

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

   wire clk = clock_65mhz;

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


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

   // display module for debugging

   reg [63:0] dispdata;
        reg [127:0] dispdata_ascii;

        display_string hexdisp2(reset, clock_27mhz, dispdata_ascii,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);

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

   // wire up to ZBT ram

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


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

        //assign main and aux camera streams
        wire cam_pick = ~switch[5];

   // generate pixel value from reading ZBT memory
   wire [17:0]        vr_pixel;
```

```
        wire [18:0]         vram_addr1;

             //use xvga signals to get pixel data from memory
        vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                        vram_addr1,vram_read_data);

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

             //two sets
        wire [29:0] ycrcb;  // video data (luminance, chrominance)
        wire [2:0] fvh, fvh2;
             reg [2:0] fvh_aux, fvh_to_ntsc;       // sync for field, vertical, horizontal
        wire       dv, dv2;
             reg  dv_to_aux, dv_to_ntsc;   // 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));

       //wires for input from this labkit and logic analyzer
             reg [17:0] ntsc_data_in, data_in_aux;
                    wire [17:0] ycrcb2;
              //I was trying to get the labkit to switch between inputs here, but the result just seemed to be that
the display locked up
             always @ (*) begin
              ntsc_data_in = cam_pick?     ycrcb2 : {ycrcb[29:24],ycrcb[19:14],ycrcb[9:4]};
              fvh_to_ntsc = cam_pick? fvh2 : fvh;
              dv_to_ntsc = cam_pick?  dv2 : dv;
              data_in_aux = ~cam_pick? ycrcb2 : {ycrcb[29:24],ycrcb[19:14],ycrcb[9:4]};
              fvh_aux = ~cam_pick? fvh2 : fvh;
              dv_to_aux = ~cam_pick? dv2 : dv;
              end

        // code to write NTSC data to video memory
        wire [18:0] ntsc_addr, ntsc_addr_1; //outputs to zbt to display main stream on screen
        wire [35:0] ntsc_data, ntsc_data_1; //
        wire        ntsc_we, ntsc_we_1, aux_we; //aux stream is not displayed
        wire [9:0] x_,y_, x_2, y_2, x_1, y_1;
        wire v_, v_2, v_1;
             wire [9:0] lum_data, lum_data_1, lum_data2;
        ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh_to_ntsc, dv_to_ntsc, ntsc_data_in,
                        ntsc_addr_1, ntsc_data_1, ntsc_we_1, ~switch[6], x_1, y_1, v_1, lum_data_1);

             //for testing aux input
             wire [18:0]    ntsc_addr_2;
             wire [35:0] ntsc_data_2;
             wire ntsc_we_2;
             wire [9:0] /*x_2, y_2, v_2, */lum_data_2;
      /*     ntsc_to_zbt n2z_2 (clk, tv_in_line_clock1, fvh_aux, dv_to_aux, data_in_aux,
                        ntsc_addr_2, ntsc_data_2, ntsc_we_2, ~switch[6], x_2, y_2, v_2, lum_data_2);
        */

      //another attempt to switch inputs, same results
             wire cam_p2 = ~switch[2];
             assign ntsc_addr = cam_p2? ntsc_addr_1 : ntsc_addr_2;
             assign ntsc_data = cam_p2?  ntsc_data_1 : ntsc_data_2;
             assign ntsc_we =  cam_p2? ntsc_we_1 : ntsc_we_2;
             assign {x_,y_,v_} = cam_p2? {x_1,y_1,v_1} : {x_2,y_2,v_2};
             assign lum_data = cam_p2? lum_data_1 : lum_data_2;


             //calibration cursor display

             //cursor control buttons
             wire up, down, left, right;
             tvdebounce tv0(reset, tv_in_line_clock1, clk, ~button_up, up_);
             tvdebounce tv1(reset, tv_in_line_clock1, clk, ~button_down, down_);
             tvdebounce tv2(reset, tv_in_line_clock1, clk, ~button_right, right_);
             tvdebounce tv3(reset, tv_in_line_clock1, clk, ~button_left, left_);
             reg [3:0] old_dir;
             always @ (posedge tv_in_line_clock1) begin old_dir[3] <= up_; old_dir[2] <= down_; old_dir[1] <=
right_; old_dir[0] <= left_;  end
             assign up = ~old_dir[3] && up_;
             assign down = ~old_dir[2] && down_;
             assign right = ~old_dir[1] && right_;
```

```
		assign left = ~old_dir[0] && left_;

		//cursor
		wire [29:0] curs_pix;
		wire [9:0] curs_x_loc, curs_y_loc;
		cursor curs0(tv_in_line_clock1, reset, x_, y_, up, down, left, right, ~switch[5], curs_pix, curs_x_loc,
curs_y_loc, eo);
		wire sw_curs = ~switch[2];
		wire [17:0] cursor_pix = sw_curs? {curs_pix[29:24],curs_pix[19:14],curs_pix[9:4]} : 0;

		//convert hex to ascii for use with ascii display module
		wire [23:0] x_curs_ascii_addr, y_curs_ascii_addr;
		hex2ascii h2a_1( {2'b0,curs_x_loc}, x_curs_ascii_addr );
		hex2ascii h2a_2( {2'b0,curs_y_loc}, y_curs_ascii_addr );

		//generate character display strings on screen
		wire [2:0] char_pixel0, char_pixel1, char_pixel2;
		reg [2:0] char_pixel;
		char_string_display char0(clk,hcount,vcount,char_pixel0,{"calibration h,c:",
									x_curs_ascii_addr, "," ,
y_curs_ascii_addr},11'd80,10'd600);
	    defparam char0.NCHAR = 23; // number of 8-bit characters in cstring
	    defparam char0.NCHAR_BITS = 5; // number of bits in NCHAR

		//more character display
		wire [31:0] x,y,z;
		wire [30:0] sx,sy,sz;
		wire [63:0] x_ascii, y_ascii, z_ascii, sx_a, sy_a, sz_a;
		hex2ascii2 h2a2_1(x, x_ascii);
		hex2ascii2 h2a2_2(y, y_ascii);
		hex2ascii2 h2a2_3(z, z_ascii);
		hex2ascii2 h2a2_4({1'b0, sx}, sx_a);
		hex2ascii2 h2a2_5({1'b0,sy}, sy_a);
		hex2ascii2 h2a2_6({1'b0, sz}, sz_a);
		char_string_display char1(clk,hcount,vcount,char_pixel1,{"object
x,y,z:",x_ascii,",",y_ascii,",",z_ascii},11'd80,10'd650);
	    defparam char1.NCHAR = 39; // number of 8-bit characters in cstring
	    defparam char1.NCHAR_BITS = 6; // number of bits in NCHAR
		char_string_display char2(clk,hcount,vcount,char_pixel2,{"simple
x,y,z:",sx_a,",",sy_a,",",sz_a},11'd80,10'd725);
	    defparam char2.NCHAR = 39; // number of 8-bit characters in cstring
	    defparam char2.NCHAR_BITS = 6; // number of bits in NCHAR


		//assign incoming signals from slave labkit

	  wire [5:0] lum_2, cr_2, cb_2;
	  wire [15:0] frame_in = analyzer4_data;
	  recieve rcv0(clk, frame_in,lum_2,cr_2,cb_2,fvh2,dv2);		//clk should be ok since it is > 2x
tv_in_line_clock
	  assign       ycrcb2 = {lum_2,cr_2,cb_2};
	  assign analyzer4_clock = tv_in_line_clock1;			//xmit signal to slave labkit to try to
synchronize cameras

				//signal is unused
		//code to test transmitter - should be used on slave labkit
		DCM tx_clk1(.CLKIN(tv_in_line_clock1),.CLKFX(tx_clock_unbuf));
	   // synthesis attribute CLKFX_DIVIDE of tx_clk1 is 2
	   // synthesis attribute CLKFX_MULTIPLY of tx_clk1 is 4
	   // synthesis attribute CLK_FEEDBACK of tx_clk1 is NONE
		// synthesis attribute CLKIN_PERIOD of tx_clk1 is 37;

	   BUFG tx_clk2(.O(tx_clk),.I(tx_clock_unbuf));

		//wire up transmitter
		wire [15:0] frame_out;
		send snd0(tx_clk, ycrcb[29:24],ycrcb[19:14],ycrcb[9:4],fvh,dv,frame_out);
	   assign analyzer3_data = frame_out;

		//switch for display of bright regions
	   wire sw_lum = ~switch[4];

		//calculation of image average coordinates
	   wire [9:0] av_x, av_x2;
		wire [9:0] av_y, av_y2;
	   wire [17:0] lum_pix, aux_lum_pix;
	   coord_proc cor0(clk, reset, ntsc_we, v_, x_,y_, lum_data, av_x, av_y,
			lum_pix, update_avgs);

		wire update_avgs_aux;
```

```
            coord_proc cor_aux(clk, reset, aux_we, v_2, x_2,y_2, lum_data2, av_x2, av_y2,
                                                                aux_lum_pix, update_avgs_aux);

            //calculate distance to object

            distance_calc dist0(clk, update_avgs, update_avgs_aux, av_x, av_y, av_x2, av_y2, x, y, z);
            simple_distance_calc sdist0(clk, update_avgs, update_avgs_aux, av_x, av_y, av_x2, av_y2, sx, sy, sz);

            // code to write pattern to ZBT memory
        reg [31:0]  count;
        always @(posedge clk) count <= reset ? 0 : count + 1;
        wire [18:0]        vram_addr2 = count[0+18:0];
        wire [35:0]        vpat = ( ~switch[1] ? {6{count[3+3:3],2'b0}}
                            : {3{count[3+4:4],8'b0}} );


            //display average positions on screen
        wire        b,hs,vs;
            wire [2:0] av_curs_pix;
            av_cursor acurs0(clk,av_x,av_y,vram_addr1,hs, av_curs_pix);

            //convert positions from bin to ascii address
            wire [23:0] x_av_ascii_addr, y_av_ascii_addr, x_av2_ascii_addr, y_av2_ascii_addr;
            wire [2:0] char_pixel3;
            always @ (posedge clk) dispdata_ascii <= {"av",x_av_ascii_addr,

       y_av_ascii_addr,"c:",x_curs_ascii_addr,y_curs_ascii_addr};

            hex2ascii h2a8( {2'b0,av_x}, x_av_ascii_addr );
            hex2ascii h2a9( {2'b0,av_y}, y_av_ascii_addr );
            hex2ascii h2a10( {2'b0,av_x2}, x_av2_ascii_addr );
            hex2ascii h2a11( {2'b0,av_y2}, y_av2_ascii_addr );

         char_string_display char3(clk,hcount,vcount,char_pixel3,
                                                                                            {"camera
x,y:",x_av_ascii_addr,",",y_av_ascii_addr},11'd80,10'd625);
        defparam char3.NCHAR = 18; // number of 8-bit characters in cstring
        defparam char3.NCHAR_BITS = 5; // number of bits in NCHAR
                    always @ (posedge clk) char_pixel <= char_pixel0 | char_pixel1 | char_pixel2 | char_pixel3;
          // font1224_hex_rom f(font_addr,clk,font_byte);

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

        wire        sw_ntsc = ~switch[7];
        wire        my_we = sw_ntsc ? (hcount[0]==1'd0) : blank;
        wire [18:0]        write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
        reg [17:0] old_lum_pix;
        always      @ (posedge clk) old_lum_pix <= lum_pix;
        wire [35:0]        write_data = sw_ntsc ? (sw_lum? ({18{av_curs_pix}} | {lum_pix,old_lum_pix}) : ntsc_data)
: vpat;

       //trying to get average display working again
       /*      reg [2:0] av_pix;
               always @ (*)
                av_pix <= ( (vram_addr[18:10]=={av_y[8:0]})
                                    || ( vram_addr[8:0] == av_x[9:1])) ?   {18{2'b01}} : 0; //{2
{{6'b101010}},{6{1'b0}}, {6{1'b0}}} } : 0;
         */

         //assigning data to vram
        assign      vram_addr = my_we ? write_addr : vram_addr1;
        assign      vram_we = my_we;
        always @ (*) vram_write_data = (switch[3] && ((~switch[5]? (vram_addr[18:9] == {av_y[9:0]}) :

             (vram_addr[18:10]=={av_y[8:0]})) || ((switch[1]? vram_addr[9:1] :

            vram_addr[8:0]) == (switch[2]? av_x[8:0] : av_x[9:1])) ) )?
                                                                           {12{3'd5}} : write_data;


        //trying to get cross hairs working again
       /*  always @ (*) vram_write_data = (eo? {write_data[35:18], cursor_pix }: {cursor_pix, write_data[17:0]})
                                                                                  || (switch[3] &&
((vram_addr[18:10]=={av_y[8:0]})
                                                                                  || ( vram_addr[8:0] == av_x[9:1]) )
)?
                                                                           {36'd40403} : write_data;
       */
       //      always @ (*) vram_write_data =  /* (eo? {write_data[35:18], cursor_pix }: {cursor_pix,
write_data[17:0]}) ; */
       //                                                                                     write_data;
```

```
          // select output pixel data

          reg [23:0]  pixel;


          delayN dn1(clk,hsync,hs1); // delay by 3 cycles to sync with ZBT read
          delayN dn2(clk,vsync,vs1);
          delayN dn3(clk,blank,bl1);
          delayN dn4(clk,hs1,hs);    // delay by 3 cycles to sync with pipelines
          delayN dn5(clk,vs1,vs);
          delayN dn6(clk,bl1,bl);

          wire [7:0] red, green, blue;
          YCrCb2RGB conv0( .R(red), .G(green), .B(blue), .clk(clk), .rst(reset),
                                     .Y({vr_pixel[17:12],4'b0}), .Cr({vr_pixel[11:6],4'b0}),
.Cb({vr_pixel[5:0],4'b0}) );

               //assign output pixel
               always@ (posedge clk)
               pixel <= ~switch[0] ?  {hcount[8:6],5'b0,hcount[8:6],5'b0,hcount[8:6],5'b0} :
               (vcount >= 10'd600)? {12{char_pixel}} : {red,green,blue};
       /*    pixel <= ~switch[0]? {hcount[8:6],5'b0,hcount[8:6],5'b0,hcount[8:6],5'b0} :
                                                                      ~switch[1]? {red,green,blue}  :
                                                                      (vcount > 10'd600)? char_pixel : pixel_pipe;
                                                                      pixel_pipe <= ~switch[2]? (char_pixel |
{red,green,blue}) : {av_curs_pix,cursor_pix[17:12]};
                                                                      end                    */
                                    //pixel doesn't like these ors?
          // VGA Output.  In order to meet the setup and hold times of the
          // AD7125, we send it ~clock_65mhz.
          assign vga_out_red =  pixel[23:16];
          assign vga_out_green =   pixel[15:8];
          assign vga_out_blue = pixel[7:0];
          assign vga_out_sync_b = 1'b1;    // not used
          assign vga_out_pixel_clock = ~clock_65mhz;
          assign vga_out_blank_b = ~b;
          assign vga_out_hsync = hs;
          assign vga_out_vsync = vs;


         assign led = ~x[7:0];

          always @(posedge clk)
                    dispdata <= {2'b0,curs_x_loc,4'b0,2'b0,curs_y_loc,4'b0,2'b0,curs_pix};

               reg [7:0] counter;
               always @ (posedge clk) counter <= counter + 1;
               assign analyzer1_clock = switch[3]? counter[7]: switch[2]? counter[5] : switch[1]? counter [3] : clk;
                 assign analyzer1_data = ntsc_addr[15:0];
               assign analyzer2_clock = clk;
               assign analyzer2_data =  {av_x,6'b0};

               endmodule
```

```
//module to display '+' at average position of x and y data for a camera

module av_cursor(clk,av_x,av_y,vram_addr,sync, av_curs_pix);
                        input clk;
                        input [9:0] av_x, av_y;
                        input [18:0] vram_addr;
                        input sync;
                        output [2:0] av_curs_pix;

                        reg [2:0] av_curs_pix;
                        always @ (posedge clk)
                                av_curs_pix <= ~sync && ( (vram_addr[18:10]=={av_y[8:0]})
                                                || ( (vram_addr[8:0] + 1) == av_x[9:1]) ) ?   3'b101 : 0;

endmodule





//module to count coordinates based on whether a threshold luminance is exceeded or not
//reset shortly after v-sync (see code)

module coord_counter(clk, threshold, ntsc_we, v_sync, x,y, x_tot, y_tot,lum, tally, lum_pix, update );
   input clk;  //system clock
   input ntsc_we;      //ntsc we on input stream synched to system clock
   input v_sync; //write enable on input stream      synched to system clock
   input [9:0] x;      //x output of ntsc decoder
   input [9:0] y;      //y output of ntsc decoder
   input [9:0] lum ;   //luminance output of ntsc decoder
   input [9:0] threshold; //threhold luminance to accept pixels above

   output reg [27:0] x_tot;     //coordinates totals for pixels exceeding threshold
        output reg [27:0] y_tot;
   output reg [19:0] tally;    //total number of pixels exceeding threshold
   output [17:0] lum_pix ;           //luminance data for pixel based on threshold
        output update;                        //update signal to coord average

        wire inc = (lum >= threshold);
   assign lum_pix = (inc)? {18{1'b1}}: {18{1'b0}}; //makes pink for inc and green for not inc after conversion

   reg          old_v;
        reg      old_pos_v;
   wire         pos_v = v_sync && ~old_v;           //detect positive edge of vsync
   always @ (posedge clk) begin
      old_v <= v_sync;
               old_pos_v <= pos_v;
      if (old_pos_v) begin //reset totals     on positive edge of vsync, delayed one cycle
               x_tot <= 0;
               y_tot <= 0;
               tally <= 0;
               end
      else if (ntsc_we && inc) begin  //update on ntsc write enable and if the threshold is met
         x_tot <= x_tot + x;
         y_tot <= y_tot + y;
         tally <= 1 + tally;
           end
        end
        assign update = pos_v;         //tell coord average to read in data on posedge of v sync
endmodule

//modulte to determine average coordinate based on x, y totals and tally
module coord_average(clk, reset, update, x_tot, y_tot,  tally,
                                                            av_x, av_y,  threshold, update_avgs);
   input clk, reset, update; //sys clock, reset, update signal from coord counter
   input [27:0] x_tot;          //x total from coord counter
        input [27:0] y_tot;            //y total from coord counter
   input [19:0]       tally;   //coordinate tally from coord counter
   output [9:0] av_y;           //calculate average y
        output [9:0] av_x;                    //calculated average y
        output reg update_avgs;                 //update signal to distance calculator
   output reg [9:0] threshold = 256;  //default threshold value
        reg [19:0] tally_reg;                  //latched tally
   reg [31:0]  x_tot_reg;                      //latched x total
        reg [31:0] y_tot_reg;                  //latched y total


   always @ (posedge clk) begin
               if (reset) threshold <= 256;   //reset threshold on reset
         else if (update) begin        //update threshold and totals on a update signal from coord counter
         threshold <= ((threshold < 1022) &&(tally_reg > 5000))? threshold + 1 :
```

```
                                                                                                    ((threshold > 1) &&
(tally_reg < 2000))? threshold - 1:
                                                                                                    threshold; //cheezy
threshold level setter
                x_tot_reg <= x_tot;
                y_tot_reg <= y_tot;
                tally_reg <= tally;
             end
          end

               //wire up dividers to calculate averages
          wire [19:0] quot0, quot1;
          wire [9:0] rem0, rem1;
          fat_divider av0(x_tot_reg,tally_reg,quot0,rem0,clk);
          fat_divider av1(y_tot_reg,tally_reg,quot1,rem1,clk);

               //assign outputs
          assign av_x = quot0[9:0];
          assign av_y = quot1[9:0];

               //assign distance calculator update signal on positive edge of (average gives same value twice)
               reg [9:0] old_av_x, older_av_x,  old_av_y, older_av_y;
               always @ (posedge clk) begin
                       {old_av_x, older_av_x, old_av_y, older_av_y} <= {av_x, old_av_x, av_y, old_av_y};
                       update_avgs <= (((old_av_x == av_x) && (old_av_x != older_av_x))
                                                              && ((old_av_y == av_y) && (old_av_y != older_av_y)))?
1'b1 : 1'b0;
                       end
       endmodule

       //wrapper module for coordinate processing
       module coord_proc(clk, reset, ntsc_we, v_sync, x,y, lum, av_x, av_y, lum_pix, update_avgs);
               input clk,reset; //system clock and reset signal
               input ntsc_we;  //ntsc write enable from ntsc_to_zbt
               input v_sync;  //vsync is actually used to determine when to latch in tallies
               input [9:0] x;   //ntsc x coordinate
               input [9:0] y;   //ntsc y coordinate
               input [9:0] lum;      //luminance for current ntsc pixel
               output [9:0] av_x; //average pixel x
               output [9:0] av_y; //average pixel y
               output update_avgs;   //update signal to distance calculator
               output [17:0] lum_pix; //pink/green pixel based on luminance threshold

               //wire up signals passed from coord counter to coord average
               wire [9:0] threshold;
               wire [27:0] x_tot, y_tot;
               wire [19:0] tally;
               //count coordinates
               coord_counter c0 (clk, threshold, ntsc_we, v_sync, x,y, x_tot, y_tot,lum, tally, lum_pix, update );
               //calculate averages
               wire [19:0] quot0, quot1, rem0, rem1;
               coord_average av0(clk, reset, update, x_tot, y_tot, tally,av_x, av_y, threshold, update_avgs);

       endmodule


       ////module to display cursor on video of camera and give cursor coordinates
       module cursor(tv_clk, reset, xcount, ycount, up, down, left, right, jump, pixel, x_loc, y_loc,eo);
          input tv_clk; //input video clk
          input reset; //reset position
          input [9:0] xcount; //x-coordinate of ntsc
          input [9:0] ycount; //y-coordinate of ntsc
          input       up, down, left, right; //cursor position contr
               input jump;     //big jumps in position instead of one unit
          output [29:0] pixel; //ycrcb data to screen
          output reg [9:0]  x_loc, y_loc; //cursor location
               output eo; //0 for even, 1 for odd
               //helps identify x location for use in 2 pixels/mem location ssytem
          parameter xmax = 1000;
          parameter xmin = 0;
          parameter ymax = 1000;
          parameter ymin = 0;
          parameter width = 1;
               parameter span = 20;
          parameter pixel_color = {10'd512,10'd1023,10'd0}; //blue in 30b ycrcb

          //update coordinate position
               always @ (posedge tv_clk) begin
             if (reset) x_loc <= 10'd100;
         else if (left && (x_loc > 1+  xmin)) x_loc <= x_loc - (1 + jump);
```

```
                else if (right && 1 + x_loc < xmax) x_loc <= x_loc + (1+ jump)<<1;  //testing large jumps on x,
currently broken
           end

           always @ (posedge tv_clk) begin
              if (reset) y_loc <= 10'd100;

            else if (up && (y_loc > ymin + 1)) y_loc <= (y_loc - 1);
                else if (down && (1 + y_loc < ymax)) y_loc <= y_loc + 1;
           end

           //assert T to display pixel
                //assert based on pixel falling within cursor location
           wire assert = (
                         (
                          (
                           (y_loc <= (ycount + width)) &&
                           (y_loc > (ycount - width))) &&
                          ((x_loc > xcount - span) &&
                           (x_loc < xcount + span))) ||
                         (
                          (
                           (x_loc <= (xcount + width)) &&
                           (x_loc > (xcount - width)))       &&
                          (
                           (y_loc > ycount - span)&&
                           (y_loc < ycount + span))));
           assign pixel = assert? pixel_color: 0;
                assign eo = xcount[0];
           //pixel should be OR'ed to screen (first in OR)
           //clocks off of ntsc so that the pixels actually corrsepond to ntsc pixels (more accurate cal)
        endmodule



        //calculate distance based on formula from Robot Vision by Horn
        module distance_calc(clk, update, update2, av_x1, av_y1, av_x2, av_y2, x, y, z);
                       input clk;  //system clk
                       input update, update2; //ntsc sync signal used to clock multiplier
                       input [9:0] av_x1, av_x2;  //av x coord
                       input [9:0] av_y1, av_y2;  //av y coord
                       output [31:0] x, y, z;
                       parameter signed S11 = 10'b10000000; parameter signed S12 = 10'b10000000;
                       parameter signed S13 = 10'b10000000; parameter signed S14 = 10'b10000000; //influence
coefficients
                       parameter signed S21 = 10'b10000000; parameter signed S22 = 10'b10000000;
                        parameter signed S23 = 10'b10000000; parameter signed S24 = 10'b10000000;
                       parameter signed S31= 10'b10000000; parameter signed S32 = 10'b10000000;
                        parameter signed S33 = 10'b10000000; parameter signed S34 = 10'b10000000;


                       //generate signed (2's comp) value
                       reg signed [9:0] x_1;
                       reg signed [9:0] y_1;
                       reg signed [9:0] x_2;
                       reg signed [9:0] y_2;

                       always @ (posedge clk)                 //update on update from coord average module for either
camera's data stream
                                if (update || update2) begin x_1 <= ~av_x1 + 1; y_1 <= ~av_y1 + 1; x_2 <= ~av_x2 + 1;
y_2 <= ~av_y2 + 1; end

                       //set up terms in matrix equation for position
                       wire signed [9:0] A12 = -x_2;
                       reg signed [21:0] A11, A21;
                       wire signed [9:0] A22 = -y_2;
                       reg signed [31:0] divisor;
                       parameter signed [9:0] B1 = -S14;
                       parameter signed [9:0] B2 = -S24;
                       reg signed [31:0] zl_num;
                       reg signed [31:0] zr_num;

                       wire signed [31:0] zl_rem, zr_rem;
                       wire signed [31:0] z_l, z_r;

                       always @ (posedge clk) begin
                               divisor  <= (A11*A22) - (A12*A21);
                               zl_num <= (A22 * B1) - (A21 * B2);
                               zr_num <= (-A12 * B1) + (A11 * B2);
                               A11 <= (S11 * x_1) + (S12 * y_1) + S13;
                               A21 <= (S11 * x_1) + (S22 * y_1) + S23;
                       end

                               //divide to calculate postion in left and right coordinate systems
                               dv det0(zl_num,divisor,z_l,zl_rem,clk);
                               dv det1(zr_num,divisor,z_r,zr_rem,clk);

               //arbitrarily choose left coordinate system
               //x and y distance scale with z if f is taken to be 1 and error is absorbed by coefficients
                       reg signed [31:0] x, y;
                       always @ (posedge clk)
                       begin x <= z_l * x_1; y <= z_l * y_1; end
                       assign z = z_l;
        endmodule

        //simple distance calculator based on idea that optical axes are parallel, focal length and baseline are known
        module simple_distance_calc(clk, update, update2, av_x1, av_y1, av_x2, av_y2, x, y, z);
                       input clk;  //system clk
                       input update, update2; //ntsc sync signal used to clock multiplier
                       input [9:0] av_x1, av_x2;  //av x coord
                       input [9:0] av_y1, av_y2;  //av y coord
                       output [30:0] x, y, z; //x, y, z position outputs
                       //focal length and baseline
                       parameter signed f = 10'b10000000;  parameter signed b = 10'b10000000;
                       //numerator of z equation
                       parameter signed z_num = b*f;


                       //generate signed (2's comp) value
                       reg signed [9:0] x_1;
                       reg signed [9:0] y_1;
                       reg signed [9:0] x_2;
                       reg signed [9:0] y_2;

               //latch in image positions on update signal
                       always @ (posedge clk)
                                if (update || update2) begin x_1 <= ~av_x1 + 1; y_1 <= ~av_y1 + 1; x_2 <= ~av_x2 + 1;
y_2 <= ~av_y2 + 1; end

               //calculate sum and difference of x terms, non-division bits of equation
                reg [9:0] delta_x;
                reg [19:0] sum_x, sum_y;
                reg [31:0] x_num, y_num;
                always @ (posedge clk)       begin
                       delta_x <= x_1 - x_2;
                       sum_x <= x_1 + x_2;
                       sum_y <= y_1 + y_2;
                       x_num <= (b * sum_x)>>1;
                       y_num <= (b * sum_y)>>1;
                 end
                 //do division to get x, y, z
                       dv2 div0(x_num,delta_x,x,x_rem,clk);
                       dv2 div1(y_num,delta_x,y,y_rem,clk);
                       dv2 div2(z_num,delta_x,z,z_rem,clk);

        endmodule



        //convert 12 bit hex number to ascii address
        module hex2ascii(hex, ascii_addr);

           input [11 : 0] hex;
           output [23: 0] ascii_addr;


               assign ascii_addr[7:0] = (hex[3:0] > 9)? hex[3:0] + 55 : hex[3:0] + 48;
               assign ascii_addr[8+7:8+0] =  (hex[4+3:4+0] > 9)? hex[4+3:4+0] + 55 : hex[4+3:4+0] + 48;
               assign ascii_addr[2*8+7:2*8+0] = (hex[8+3:8+0] > 9)? hex[8+3:8+0] + 55 : hex[8+3:8+0] + 48;


        endmodule
        //convert 32 bit hex number to ascii address
        module hex2ascii2(hex, ascii_addr);

           input [31 : 0] hex;
           output [63: 0] ascii_addr;


               assign ascii_addr[7:0] = (hex[3:0] > 9)? hex[3:0] + 55 : hex[3:0] + 48;
               assign ascii_addr[8+7:8+0] =  (hex[4+3:4+0] > 9)? hex[4+3:4+0] + 55 : hex[4+3:4+0] + 48;
```

```
         assign ascii_addr[2*8+7:2*8+0] = (hex[8+3:8+0] > 9)? hex[8+3:8+0] + 55 : hex[8+3:8+0] + 48;

    assign ascii_addr[3*8+7:3*8+0] = (hex[3*4+3:3*4+0] > 9)? hex[3*4+3:3*4+0] + 55 : hex[3*4+3:3*4+0] + 48;
         assign ascii_addr[4*8+7:4*8+0] = (hex[4*4+3:4*4+0] > 9)? hex[4*4+3:4*4+0] + 55 : hex[4*4+3:4*4+0] + 48;
         assign ascii_addr[5*8+7:5*8+0] = (hex[5*4+3:5*4+0] > 9)? hex[5*4+3:5*4+0] + 55 : hex[5*4+3:5*4+0] + 48;
         assign ascii_addr[6*8+7:6*8+0] = (hex[6*4+3:6*4+0] > 9)? hex[6*4+3:6*4+0] + 55 : hex[6*4+3:6*4+0] + 48;
         assign ascii_addr[7*8+7:7*8+0] = (hex[7*4+3:7*4+0] > 9)? hex[7*4+3:7*4+0] + 55 : hex[7*4+3:7*4+0] + 48;
endmodule



//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

/////////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data
//small mods by miles colman

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw, x_, y_,v_,lum_data);

   input        clk;   // system clock
   input        vclk;  // video clock from camera
   input [2:0]         fvh;    //frame, vsync, hsync
   input        dv;    //data valid
   input [17:0]        din;                  //data in
   output [18:0] ntsc_addr;                          //ntsc address to zbt
   output [35:0] ntsc_data;            //data to z bt
   output       ntsc_we;       // write enable for NTSC data
        output reg     [9:0] lum_data;         //output luminance data
   output [9:0] x_, y_; //output coordinate for pixel
   output v_; //output vsync signals, synchronized to clk
   input        sw;            // switch which determines mode (for debugging)

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

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display

   reg [9:0]    col = 0;
   reg [9:0]    row = 0;
   reg [17:0]   vdata = 0;
   reg          vwe;
   reg          old_dv;
   reg          old_frame;    // frames are even / odd interlaced
   reg          even_odd;     // decode interlaced frame to this wire

   wire         frame = fvh[2];
   wire         frame_edge = frame & ~old_frame;

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

        if (!fvh[2])
          begin
             col <= fvh[0] ? COL_START :
                    (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
             row <= fvh[1] ? ROW_START :
                    (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
             vdata <= (dv && !fvh[2]) ? din : vdata;
          end
     end

   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [17:0] data[1:0];
   reg       we[1:0];
   reg      eo[1:0];
   reg v[1:0];   //synchronize vsync too
   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
        {eo[1],eo[0]} <= {eo[0],even_odd};
     {v[1],v[0]} <= {v[0],frame};
        end

   // edge detection on write enable signal

   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];

   // shift each set of sixteen bytes into a register for one address for the ZBT

   reg [35:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= { mydata[17:0], data[1] };

   // compute address to store data in

   wire [18:0] myaddr = { y[1][8:0], eo[1], x[1][9:1]};

   // alternate (256x192) image data and address
   wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};          //data2 unchanged
   wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

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

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][0]==1'b0));
   reg [9:0] x_, y_;
   reg v_;

   always @(posedge clk) begin
     v_ <= v[1];                             //update vsync all the time since ntsc_we doesn't happen during
vsync
          if ( ntsc_we )
       begin
          ntsc_addr <= sw ? myaddr2 : myaddr; // normal and expanded modes
          ntsc_data <= sw ? {4'b0,mydata2} : mydata;
       x_ <= x[1];
           y_ <= y[1];
                lum_data <= {data[1][17:13],4'b0};   //implement wider y pathway later (will require new input)
                end
          end

endmodule // ntsc_to_zbt




//module to send data out through logic analyzer port
module send(clk, y,cr,cb,fvh,dv,frame);
   input clk; //send clk, should be twice tv_in_line_clock1
                                    //would work with faster clocks if inputs were synched
                                    //faster clock would lead to more noise
   input [5:0] y;        //6 bit luminance
   input [5:0] cr;  //6 bit chrominance
   input [5:0] cb;  //more 6 bit chrominance
   input [2:0] fvh;    //frame, vsync, hsync
   input       dv;  //data valid
   output [15:0] frame;        //16 bit output to LA port

        //encode frame number on last bit of frame
        //first 15 bits are input data [0:14] and [15:29]
        //for frame 0 and frame 1 respectively
   reg odd = 0;
   reg [15:0] frame ;
   always @ (posedge clk) begin
      odd <= ~odd;
      frame[0] <= odd;
      if (odd) frame <= {y,cr,cb[5:3],odd};
      else frame <= {cb[2:0],fvh,dv,8'b0,odd};
   end

endmodule

//receive module
module recieve(clk, frame,y,cr,cb,fvh,dv);
   input clk; //clock should be equal to or faster than xmit rate, ie twice tv_in_line_clock1
   input [15:0] frame; //data from LA port
   output [5:0] y;     //6 bit luminance
   output [5:0] cr;    //6 bit chrominance
   output [5:0] cb;    //more 6 bit chromiinance
   output [2:0] fvh;    //frame, vsync,hsync
   output      dv;             //data valid

   reg  [15:0]  frame_buf[2:0];
   //synchronize frame input w/ 3 cycles
   always @ (posedge clk)
     {frame_buf[2],frame_buf[1],frame_buf[0]} <= {frame_buf[1],frame_buf[0],frame};

   reg [14:0]  frame_even;
   reg [29:0]  data_out;
   //extract frame number from frame
        wire    odd = frame_buf[2][0];
        //extract data from frame
        wire [14:0]    frame_data = frame_buf[2][15:1];
   //get data from frame, update data out on odd frame
        always @ (posedge clk) begin
      if (~odd ) frame_even <= frame_data;
      else data_out <= {frame_data,frame_even};
   end
   //assign outputs for application in video transmission
   assign y = data_out[29:24];
   assign cr = data_out[23:18];
   assign cb = data_out[17:12];
   assign fvh = data_out[11:9];
   assign dv = data_out[8];

endmodule


////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// modified from Prof. Chuang's code to deal with 16 bpp and 2 pixels/read

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

   input reset, clk;
   input [10:0] hcount;
   input [9:0]        vcount;
   output [17:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]         vram_addr = {vcount, hcount[9:1]};

   wire [1:0]   hc4 = hcount[1:0];
   wire hc2 = hcount[0];
   reg [17:0]   vr_pixel;
   reg [35:0]   vr_data_latched;
   reg [35:0]   last_vr_data;

        always @(posedge clk)
     last_vr_data <= (hc4==1'd0) ? vr_data_latched : last_vr_data;

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

   always @(*)         // each 36-bit word from RAM is decoded to 18 bytes
     case (hc2)
       2'd0: vr_pixel = last_vr_data[17+18:0+18];
       2'd1: vr_pixel = last_vr_data[17:0];
     endcase

endmodule // vram_display
```