# ECE 385

Fall 2020

Experiment #8

# SoC with USB and VGA Interface

Alex Wen (acwen2), Manav Agrawal (manava3)

Section ABF

Xinbo Wu

## Introduction

This lab branches out the understanding of NIOS II with the application of VGA and USB interface using System-on-Chip (SoC) design. The USB interface consists of a port with 4 pins. It is powered by Vdd at pin 1 and grounded at pin 4. However, the bulk of data transfer occurs at pins 2 and 3, allocated as D- and D+ respectively. The data transfer to Max3421E will be done through the SPI peripheral by using the master-out-slave-in (MOSI) or master-in-slave-out (MISO).

The Video Graphics Array (VGA) is the standard used for pixelated displays. This is characterized by the screen, which is organized via a matrix of pixels, in the form of RGB (or Red, Green, Blue) characterization. In the case of this lab, this is specified as 640 horizontal pixels by 480 vertical lines. When prompting display, an electron beam will indicate the pixels from left to right, top to bottom determined with a certain framerate.

## System Description

This lab focuses on the implementation of the linking of a VGA monitor and USB keyboard to the FPGA using both hardware and software design. For the USB in terms of introducing data, this is incorporated by a SPI peripheral that dictates the clock cycle, MOSI, MISO, and Slave Select signal. In other words, the SPI Peripheral will interact with the MAX3421E where MAX3421E is the logic and the circuitry to fully implement a functional USB. SPI peripheral will be used to direct the flow of information such that the master can transmit the data while the slave reads it - the selection of master/slave would depend on the slave select signal as allocated. The NIOS II and the platform did need some initial modules such as usb_rst, usb_irq, usb_gpx, keycode, etc to correctly give inputs and outputs to MAX3421E. To communicate with the MAX3421E as said, the SPI peripheral became important as it is the 4 wire serial interface that will help us to read some data. The VGA components are then exported the keycode read by the keyboard and processed by NIOS II.

To talk a little bit more about the SPI, SPI is a Serial Peripheral Interface. The SPI core uses the two data lines (MOSI, MISO), a control line and a clock. These are the 4 main lines which it relies upon to communicate. The SS_n is the slave select signal which is active low and when it turns to 0 then it can be used to communicate and also interface with MAX3421E. The Serial Clock is driven by the master. The MOSI (Master Output Slave Input) is what the processor or the master outputted and it was given to the slave which in this case will be the MAX3421E and it allowed us to write on the 32 registers. The MISO is Master In Slave Output which was used for reading purposes. So if we look at the internal registers for SPI, there is the rxdata register which is used for reading, txdata which is used for writing, status which indicates the conditions of SPI core and whether the receive and transmission is fine, control register which is used for interrupts, ready etc and finally the slave select. These are some of the main register maps for SPI master devices. So we can either use the addresses to communicate or there is a software

command called alt_avalon_spi_command which when called performs a sequence on the SPI bus to write the data buffer to MOSI or read from MISO port. So when we want to read 1 byte of data, we set the read_length to 1, and then it stores the data pointed by the buffer. During this read of 1 byte, the data from MISO is ignored. A transaction usually takes around 8 clock cycles. For writing 1 byte, the write_length is set to 1 and then it is stored on the MOSI port where MISO is discarded. The entire C program which codes the read and write operations will be described in the appendix along with other descriptions of the modules.

Separately, the system further elaborates on the interface with the VGA. Specifically for this lab, the VGA sets control for the initial display - that is, the control will determine the size and shape of the ball as well as the RGB implementation of ball and background via the color mapper (which is further implemented by the horizontal and vertical position counter). Also, the system will set ball routine to instantiate the behavior of the ball displayed on the monitor. The behavior in this case is such that, interfacing with the USB keyboard, the keys W, A, S, D will set and reflect movements of the ball on the screen. The pixelated matrix will also set the boundary for which the ball can move, such that if it hits the boundaries, or 'the edge of the screen', it will alter the ball movement in the opposite direction from original.

If we see, the entire interaction between the FPGA, balls.sv, vga-controller.sv, color-mapper.sv, the platform design and NIOS processor passed the keycode to the balls.sv which determined where the ball should move. Based on that, it gave the output of where the location of the ball is. The VGA_controller also worked synchronously with the balls.sv as it took Clock_50 and gave the frame clock to balls.sv for it to paint a new frame. The vga_controller gave the coordinates to draw, gave the sync signals etc. This all was taken by color_mapper who used to map it to RGB and output it which was then used by the FPGA to VGA to the monitor.

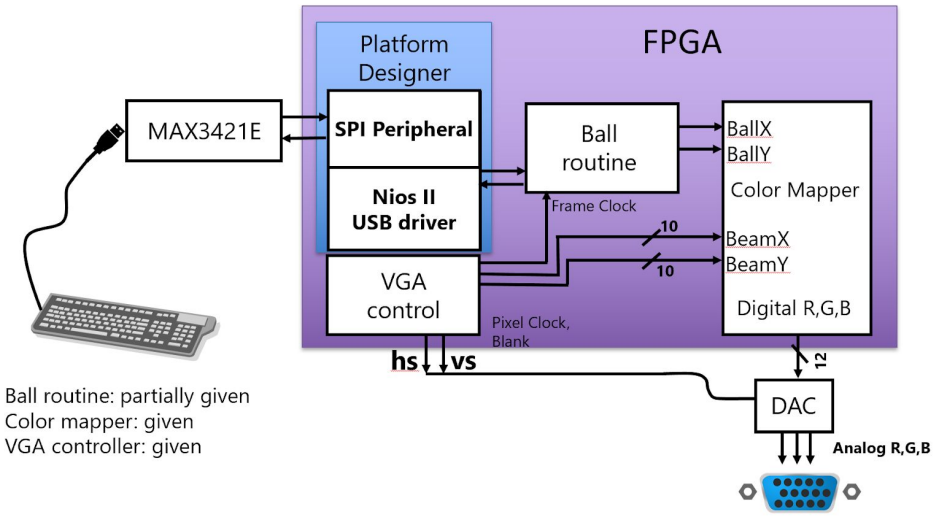



Figure 1: General Block Diagram of Lab SoC with USB/VGA Interface

This is a graphical representation that shows the lab. We see that the keyboard connects with FPGA through a MAX3421E chip. The platform design and NIOS II device interfaces through the SPI peripheral. The VGA control which is the top level (using vga_controller) provides the HS and VS (horizontal and vertical sync) to the VGA Digital to Analog converter. In addition, the vga_controller and top-level passes inputs to ball.sv and then later to color-mapper.sv which provides the RGB and using a cable, the monitor is being used for display. The C functions are better explained in the appendix where each module is described.

**Block Diagram**

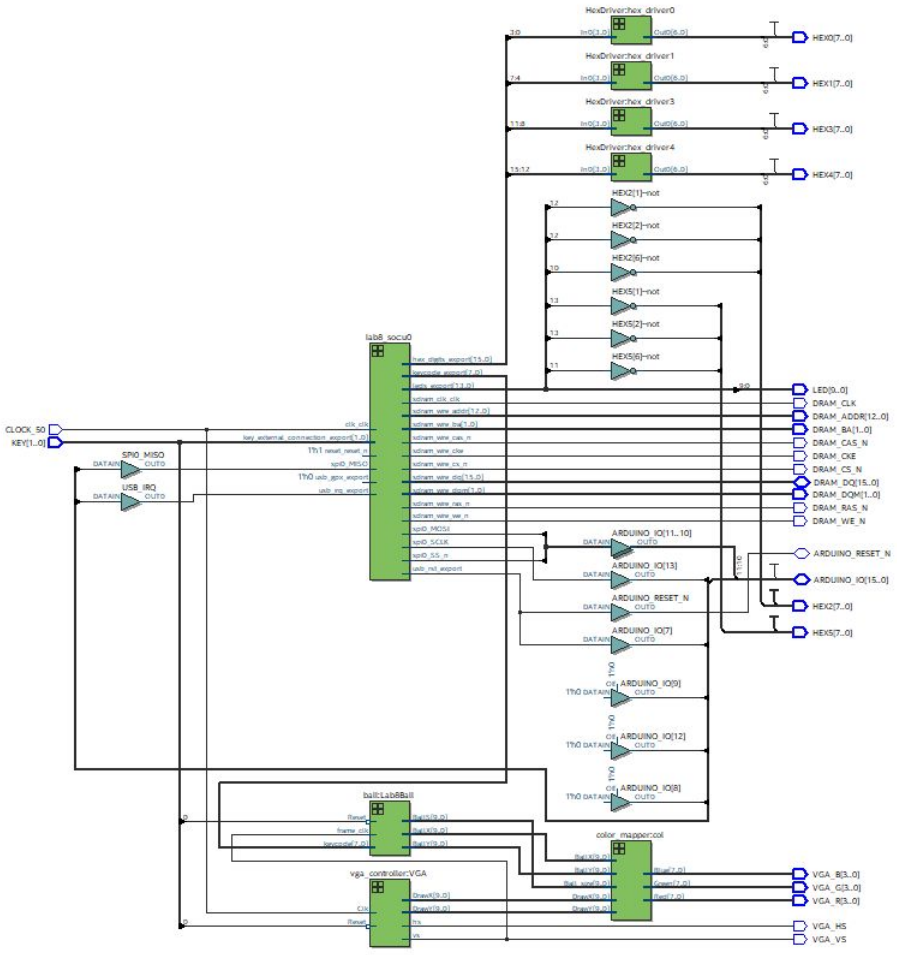



**Figure 2**: Block Diagram of Lab 8

This is the general block diagram of the entire Lab 8. The top-level is with the processor and is connected to balls.sv, vga_controller and color_mapper which provide all the VGA inputs. The hex drivers drive the value of the keycode from the processor. Since the USB connected through

a shield which was plugged into the Arduino I/O ports, this can also be seen in the overall diagram.

## Platform Module Descriptions

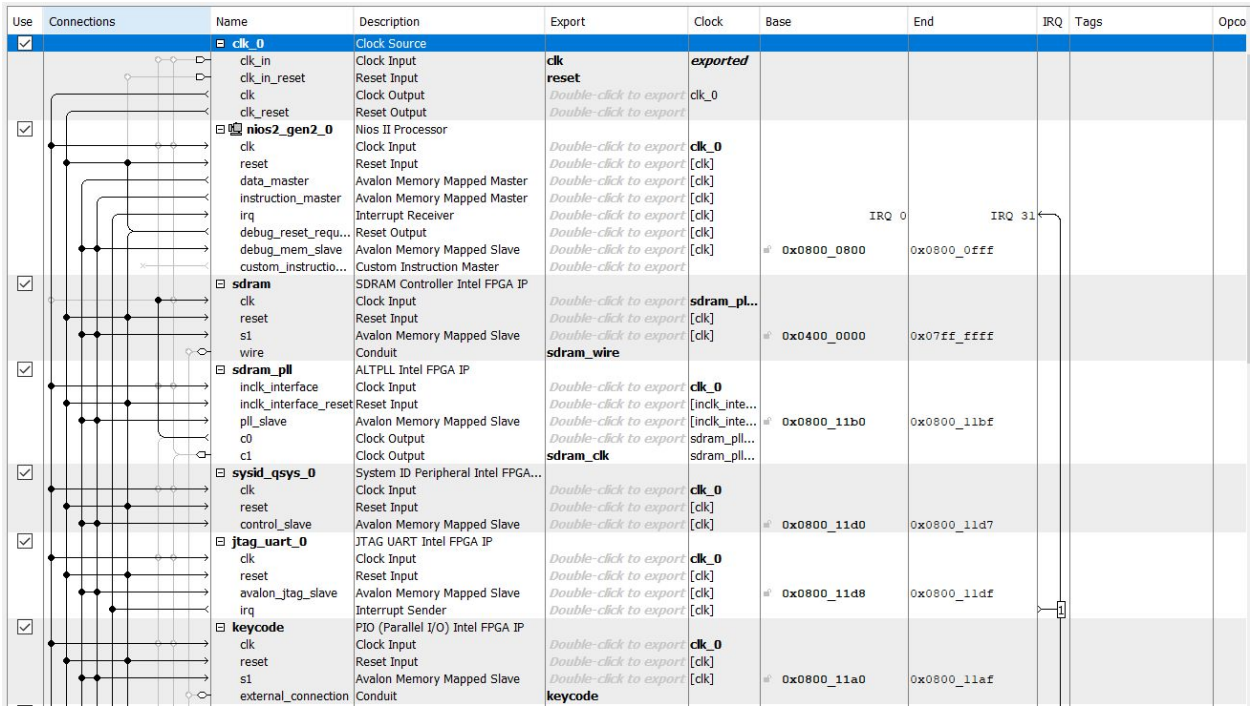
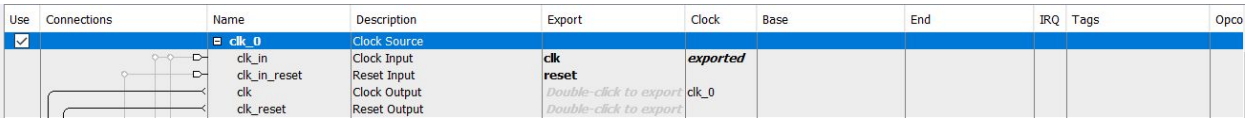



**Figure 3: General Platform Design of the SoC with USB and VGA Interface**

The figure 4 shows the figure of the platform design which shows different modules utilized for input, output, timer, peripherals, memory, processor, etc.



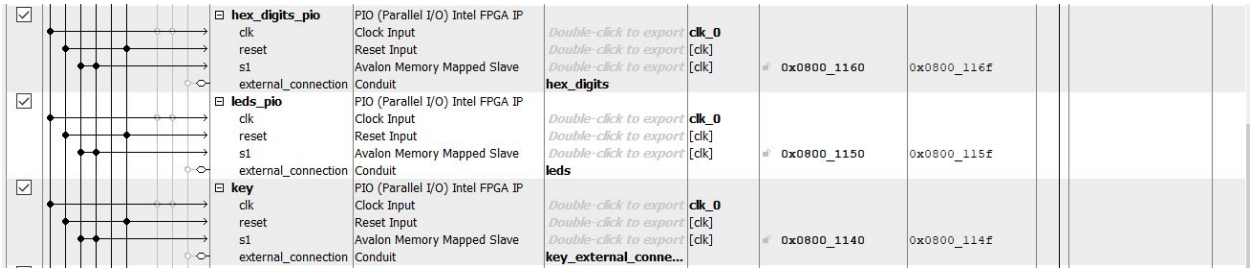

This module is providing clock inputs which are utilized by other components except SDRAM. SDRAM needs a separated phase shifted clock which will be discussed later but other components are synchronous to 50 MHz clock.


This module is the NIOS II/e processor. This provides the data_master and instruction_master bus also. This is a modified 32 bit Harvard Risc architecture.


These components refer to the various SDRAM and its related components. The SDRAM Controller is the main SDRAM memory which has a clock input, reset input, avalon memory mapped slave and exports data through sdram_wire. SDRAM_pll (phase locked loop) is that interface which provides a phase shifted clock and the clock used by SDRAM.


The first part is the System ID peripheral which is a read-only that provides the SOPC. Creates an identification and identity for each of the modules. The second two are the parallel Input/Output Intel FPGA IP. The second is a jtag which allows the processor to read/write on the console and can help debug functionalities. It is an important component.


This is the major USB peripherals, keycode, usb_irq, usp_gpx and usb_rst. The keycode will output what key was pressed and it will be used by other modules to determine and which value to display?The usb_irq and usb_gpx are 1-bit data inputs. USB_irq is the usb interrupt request. The USB may interrupt the transaction and this can be observed when USB is abruptly removed or added that it shows IRQ69. USB_rst is the 1-bit output which sends the reset signal. The usb_gpx is a general push-pull multiplexer which indicates the default operating state or whether the USB bus has traffic or something else.




This has the hardware input/output commands. The first one is the hex value which will display the keyboard keycode value on the hex digits. The second PIO module will take the input from the memory, or the NIOS processor. The clock is synchronous and reset functionality. This is done to provide the Input/Output functionality. The PIO modules that we chose had the data width of 8 bits since 8 LEDs. The third PIO module is the keys which is for reset functionality.


The interval timer was used for a lot of purposes, usually polling is a primary function which is used to monitor the transactions between the USB.

Max3421e is a usb peripheral and the SPI peripheral was used to interact with the inputs provided by the USB through a keyboard. SPI is a serial peripheral interface and consists of 4 wires just like a USB but handles the transactions efficiently. (SPI - Serial Peripheral Interface)

They have all been generated addresses by the platform design as it was better for the designer.

**Design Resources and Statistics**

| | NIOS Processor |
|---|---|
| LUT | 2447 |
| DSP | 10 DSP Blocks used |
| Memory (BRam) | 11,264 / 1,677,312 ( < 1 % ) - Total Block Memory Bits<br>46,080 / 1,677,312 ( 3 % ) - Total Block Memory Bit Implementation |
| Flip-Flop | 2447/49,760 (5%) |
| Frequency | 73.48 MHz |
| Static Power | 96.79 mW |
| Dynamic Power | 58.78 mW |
| Total Power | 237.11 mW |

Hidden Question #2/2:
Note that Ball_Y_Motion in the above statement may have been changed at the same clock edge that is causing the assignment of Ball_Y_pos. Will the new value of Ball_Y_Motion be used, or the old? How will this impact behavior of the ball during a bounce, and how might that interact with a response to a keypress? Can you fix it? Give an answer in your Post-Lab.

If seen by the statements, they are being assigned at the same positive clock edge. So it is a possibility that the ball will use the old value and it will not bounce as it might take the older value. A conditional statement to decide when Ball_Y_Pos should be assigned which value next which in hardware will translate to a multiplexer can alleviate the problem. (This issue was also touched upon later on in extra credit after the description of modules in appendix)

**Conclusion**

This lab essentially went well in terms of its functionality for the most part. The FPGA receives correctly the keyboard code for which it is entered. This correct code output leads to a successful manipulation of the ball movement in both the X and Y directions as requested. There is a slight

glitch in the system implementation, however, and that is indicated by the corner cases in which sometimes the design does not work as intended. This condition is exempted from overall design consideration, however, so no alterations were made in that regard at that point; however, the fix is applied in Extra Credit in the Appendix.

While debugging the lab, the main problem that came along was the avalon command and understanding what it returned and how its parameters worked. Many people were clueless about it and I received help from a great undergraduate assistant.

In terms of the guidances of this lab, the USB and VGA interface with the FPGA and NIOS II is fairly well explained. As designers, we have developed a fair understanding of how a USB device and a VGA component will interact with a computer. A point of ambiguity in the lab guidance that could be further elaborated is the implementation of the Avalon Command. In other words, there is little reference to help us understand the general purpose of the Avalon Command as well as how it uses the SPI to interact with the MAX3421E chip. The I/O learning was very fun and learning more about the display and the USB was interesting. This can help us design some ideas in the final project.

# APPENDIX

## I. SV Module Description

```
module lab8 (

      ///////// Clocks /////////
      input              CLOCK_50,

      ///////// KEY ////////
      input    [ 1: 0]   KEY,

      ///////// SW /////////
      input    [ 9: 0]   SW,

      ///////// LEDR /////////
      output   [ 9: 0]   LED,

      ///////// HEX ////////
      output   [ 7: 0]   HEX0,
      output   [ 7: 0]   HEX1,
      output   [ 7: 0]   HEX2,
      output   [ 7: 0]   HEX3,
      output   [ 7: 0]   HEX4,
      output   [ 7: 0]   HEX5,

      ///////// SDRAM /////////
      output             DRAM_CLK,
      output             DRAM_CKE,
      output   [12: 0]   DRAM_ADDR,
      output   [ 1: 0]   DRAM_BA,
      inout    [15: 0]   DRAM_DQ,
      output   [1:0]     DRAM_DQM,
      output             DRAM_CS_N,
      output             DRAM_WE_N,
      output             DRAM_CAS_N,
      output             DRAM_RAS_N,

      ///////// VGA /////////
      output             VGA_HS,
      output             VGA_VS,
      output   [ 3: 0]   VGA_R,
      output   [ 3: 0]   VGA_G,
      output   [ 3: 0]   VGA_B,


      ///////// ARDUINO /////////
      inout    [15: 0]   ARDUINO_IO,
      inout              ARDUINO_RESET_N
);


logic Reset_h, vssig, blank, sync, VGA_Clk;


//=======================================================
//  REG/WIRE declarations
//=======================================================
   logic SPIO_CS_N, SPIO_SCLK, SPIO_MISO, SPIO_MOSI, USB_GPX, USB_IRQ, USB_RST;
   logic [3:0] hex_num_4, hex_num_3, hex_num_1, hex_num_0; //4 bit input hex digits
   logic [1:0] signs;
   logic [1:0] hundreds;
   logic [9:0] drawxsig, drawysig, ballxsig, ballysig, ballsizesig;
   logic [7:0] Red, Blue, Green;
   logic [7:0] keycode;

//=======================================================
//  Structural coding
//=======================================================
   assign ARDUINO_IO[10] = SPIO_CS_N;
   assign ARDUINO_IO[13] = SPIO_SCLK;
   assign ARDUINO_IO[11] = SPIO_MOSI;
   assign ARDUINO_IO[12] = 1'bZ;
```

```
//=======================================================
//  Structural coding
//=======================================================
  assign ARDUINO_IO[10] = SPIO_CS_N;
  assign ARDUINO_IO[13] = SPIO_SCLK;
  assign ARDUINO_IO[11] = SPIO_MOSI;
  assign ARDUINO_IO[12] = 1'bZ;
  assign SPIO_MISO = ARDUINO_IO[12];

  assign ARDUINO_IO[9] = 1'bZ;
  assign USB_IRQ = ARDUINO_IO[9];

  //Assignments specific to Circuits At Home UHS_20
  assign ARDUINO_RESET_N = USB_RST;
  assign ARDUINO_IO[7] = USB_RST;//USB reset
  assign ARDUINO_IO[8] = 1'bZ; //this is GPX (set to input)
  assign USB_GPX = 1'b0;//GPX is not needed for standard USB host - set to 0 to prevent interrupt

  //Assign uSD CS to '1' to prevent uSD card from interfering with USB Host (if uSD card is plugged in)
  assign ARDUINO_IO[6] = 1'b1;

  //HEX drivers to convert numbers to HEX output
  HexDriver hex_driver4 (hex_num_4, HEX4[6:0]);
  assign HEX4[7] = 1'b1;

  HexDriver hex_driver3 (hex_num_3, HEX3[6:0]);
  assign HEX3[7] = 1'b1;

  HexDriver hex_driver1 (hex_num_1, HEX1[6:0]);
  assign HEX1[7] = 1'b1;

  HexDriver hex_driver0 (hex_num_0, HEX0[6:0]);
  assign HEX0[7] = 1'b1;

  //fill in the hundreds digit as well as the negative sign
  assign HEX5 = {1'b1, ~signs[1], 3'b111, ~hundreds[1], ~hundreds[1], 1'b1};
  assign HEX2 = {1'b1, ~signs[0], 3'b111, ~hundreds[0], ~hundreds[0], 1'b1};
```

```
  //Assign one button to reset
  assign {Reset_h}=~ (KEY[0]);

  //Our A/D converter is only 12 bit
  assign VGA_R = Red[7:4];
  assign VGA_B = Blue[7:4];
  assign VGA_G = Green[7:4];


  lab8_soc u0 (
     .clk_clk                       (CLOCK_50),            //clk.clk
     .reset_reset_n                 (1'b1),           //reset.reset_n
     //.altpll_0_locked_conduit_export    (),               //altpll_0_locked_conduit.export
     //.altpll_0_phasedone_conduit_export (),               //altpll_0_phasedone_conduit.export
     //.altpll_0_areset_conduit_export    (),               //altpll_0_areset_conduit.export
     .key_external_connection_export    (KEY),            //key_external_connection.export

     //SDRAM
     .sdram_clk_clk(DRAM_CLK),                            //clk_sdram.clk
     .sdram_wire_addr(DRAM_ADDR),                         //sdram_wire.addr
     .sdram_wire_ba(DRAM_BA),                             //.ba
     .sdram_wire_cas_n(DRAM_CAS_N),                       //.cas_n
     .sdram_wire_cke(DRAM_CKE),                           //.cke
     .sdram_wire_cs_n(DRAM_CS_N),                         //.cs_n
     .sdram_wire_dq(DRAM_DQ),                             //.dq
     .sdram_wire_dqm(DRAM_DQM),             //.dqm
     .sdram_wire_ras_n(DRAM_RAS_N),                       //.ras_n
     .sdram_wire_we_n(DRAM_WE_N),                         //.we_n

     //USB SPI
     .spi0_SS_n(SPIO_CS_N),
     .spi0_MOSI(SPIO_MOSI),
     .spi0_MISO(SPIO_MISO),
     .spi0_SCLK(SPIO_SCLK),

     //USB GPIO
     .usb_rst_export(USB_RST),
     .usb_irq_export(USB_IRQ),
```

Module: Lab8.sv (Top level)

Inputs: CLOCK_50, KEY[1:0] - Clock used for synchronous operation, key[0] for reset

Outputs: LED[9:0], HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM outputs (DRAM_CLK, DRAM_CKE, DRAM_BA, DRAM_DQ*, DRAM_DQM*, DRAM_CS, DRAM_WE, DRAM_CAS/RAS) Vga_HS and VS (horizontal and vertical sync), VGA_R, VGA_B, VGA_G

There are some bidirectional ports such as for ARDUINO_IO and some of the DRAM_DQ (inout) which is important to correctly implement the USB shield and the DRAM memory

Description: This is the top-level module which combines all of the components together, including VGA, Ball, HexDriver, SDRAM, NIOS II etc.

Purpose: The purpose of the top-level is that it is the interface between the FPGA and the actual processor and SDRAM. This HDL file combines and makes sure everything is properly assigned/placed.

```
module HexDriver (input  [3:0]  In0,
                  output logic [6:0]  Out0);

    always_comb
    begin
        unique case (In0)
            4'b0000   : Out0 = 7'b1000000; // '0'
            4'b0001   : Out0 = 7'b1111001; // '1'
            4'b0010   : Out0 = 7'b0100100; // '2'
            4'b0011   : Out0 = 7'b0110000; // '3'
            4'b0100   : Out0 = 7'b0011001; // '4'
            4'b0101   : Out0 = 7'b0010010; // '5'
            4'b0110   : Out0 = 7'b0000010; // '6'
            4'b0111   : Out0 = 7'b1111000; // '7'
            4'b1000   : Out0 = 7'b0000000; // '8'
            4'b1001   : Out0 = 7'b0010000; // '9'
            4'b1010   : Out0 = 7'b0001000; // 'A'
            4'b1011   : Out0 = 7'b0000011; // 'b'
            4'b1100   : Out0 = 7'b1000110; // 'C'
            4'b1101   : Out0 = 7'b0100001; // 'd'
            4'b1110   : Out0 = 7'b0000110; // 'E'
            4'b1111   : Out0 = 7'b0001110; // 'F'
            default   : Out0 = 7'bX;
        endcase
    end

endmodule
```

Module: HexDriver.sv

nputs: In0 [3:0]

Outputs: Out0 [6:0]

Description: This is the Hex Board on the FPGA where the input needs to be converted to proper binary so correct hexadecimal is shown on the Hex Board on the FPGA.

Purpose: This module shows how to display LEDs on the Hex Board.

```
module  vga_controller ( input        Clk,       // 50 MHz clock
                         Reset,     // reset signal
                output logic hs,        // Horizontal sync pulse.  Active low
                         vs,        // Vertical sync pulse.  Active low
                         pixel_clk, // 25 MHz pixel clock output
                         blank,     // Blanking interval indicator.  Active low.
                         sync,      // Composite Sync signal.  Active low.  We don't use it
                                    //   but the video DAC on the DE2 board requires an inpu
                output [9:0] DrawX,     // horizontal coordinate
                         DrawY );   // vertical coordinate

    // 800 horizontal pixels indexed 0 to 799
    // 525 vertical pixels indexed 0 to 524
    parameter [9:0] hpixels = 10'b1100011111;
    parameter [9:0] vlines = 10'b1000001100;

    // horizontal pixel and vertical line counters
    logic [9:0] hc, vc;
    logic clkdiv;

    // signal indicates if ok to display color for a pixel
    logic display;

    //Disable Composite Sync
    assign sync = 1'b0;

   //This cuts the 50 Mhz clock in half to generate a 25 MHz pixel clock
    always_ff @ (posedge Clk or posedge Reset )
    begin
        if (Reset)
            clkdiv <= 1'b0;
        else
            clkdiv <= ~ (clkdiv);
    end

   //Runs the horizontal counter  when it resets vertical counter is incremented
   always_ff @ (posedge clkdiv or posedge Reset )
   begin: counter_proc
```

```
    //Runs the horizontal counter  when it resets vertical counter is incremente
    always_ff @ (posedge clkdiv or posedge Reset )
    begin: counter_proc
        if ( Reset )
        begin
            hc <= 10'b0000000000;
            vc <= 10'b0000000000;
        end

        else
          if ( hc == hpixels )  //If hc has reached the end of pixel count
          begin
              hc <= 10'b0000000000;
              if ( vc == vlines )   //if vc has reached end of line count
                  vc <= 10'b0000000000;
              else
                  vc <= (vc + 1);
          end
          else
              hc <= (hc + 1);  //no statement about vc, implied vc <= vc;
    end

    assign DrawX = hc;
    assign DrawY = vc;

    //horizontal sync pulse is 96 pixels long at pixels 656-752
    //(signal is registered to ensure clean output waveform)
    always_ff @ (posedge Reset or posedge clkdiv )
    begin : hsync_proc
        if ( Reset )
            hs <= 1'b0;
        else
            if ((((hc + 1) >= 10'b1010010000) & ((hc + 1) < 10'b1011110000)))
                hs <= 1'b0;
            else
                hs <= 1'b1;
    end
```

```
    //vertical sync pulse is 2 lines(800 pixels) long at line 490-491
    //(signal is registered to ensure clean output waveform)
    always_ff @ (posedge Reset or posedge clkdiv )
    begin : vsync_proc
        if ( Reset )
            vs <= 1'b0;
        else
            if ( ((vc + 1) == 9'b111101010) | ((vc + 1) == 9'b111101011) )
                vs <= 1'b0;
            else
                vs <= 1'b1;
    end

    //only display pixels between horizontal 0-639 and vertical 0-479 (640x480)
    //(This signal is registered within the DAC chip, so we can leave it as pure combinational logic her
    always_comb
    begin
        if ( (hc >= 10'b1010000000) | (vc >= 10'b0111100000) )
            display = 1'b0;
        else
            display = 1'b1;
    end

    assign blank = display;
    assign pixel_clk = clkdiv;


endmodule
```

Module: VGA_Controller.sv

Inputs: Clk, Reset

Outputs: HS, VS, Pixel_clk, Blank, Sync, DrawX[9:0], DrawY[9:0]

Description: This creates the clock for the pixel frequency, it also provides the vertical sync which is the frame clock as it indicates when to paint the new frame and also it tells how the ball is bouncing as an output from FPGA.

Purpose: The VGA_Controller is (Video graphics array) creates a 25 MHz clock, and provides when to paint a new frame or when to draw on the display monitor.

```
module  color_mapper ( input        [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
                       output logic [7:0]  Red, Green, Blue );

    logic ball_on;

 /* Old Ball: Generated square box by checking if the current pixel is within a square of length
    2*Ball_Size, centered at (BallX, BallY).  Note that this requires unsigned comparisons.

    if ((DrawX >= BallX - Ball_size) &&
       (DrawX <= BallX + Ball_size) &&
       (DrawY >= BallY - Ball_size) &&
       (DrawY <= BallY + Ball_size))

     New Ball: Generates (pixelated) circle by using the standard circle formula.  Note that while
     this single line is quite powerful descriptively, it causes the synthesis tool to use up three
     of the 12 available multipliers on the chip!  Since the multiplicants are required to be signed,
     we have to first cast them from logic to int (signed by default) before they are multiplied). */

    int DistX, DistY, Size;
    assign DistX = DrawX - BallX;
    assign DistY = DrawY - BallY;
    assign Size = Ball_size;

    always_comb
    begin:Ball_on_proc
        if ( ( DistX*DistX + DistY*DistY) <= (Size * Size) )
            ball_on = 1'b1;
        else
            ball_on = 1'b0;
     end

    always_comb
    begin:RGB_Display
        if ((ball_on == 1'b1))
        begin
            Red = 8'hff;
            Green = 8'h55;
            Blue = 8'h00;
        end

        else
        begin
            Red = 8'h00;
            Green = 8'h00;
            Blue = 8'h7f - DrawX[9:3];
        end
    end

endmodule
```

Module: Color_Mapper.sv

Inputs: BallX[9:0], BallY[9:0], DrawX[9:0], DrawY[9:0], Ball_Size[9:0]

Outputs: Red[7:0], Green[7:0], Blue[7:0]

Description: This module generates a pixelated circle using the math formula

Purpose: This is important as it helps us about how we can draw different objects which can really ameliorate our future projects.

```
module  ball ( input Reset, frame_clk,
               input [7:0] keycode,
               output [9:0]  BallX, BallY, BallS );

    logic [9:0] Ball_X_Pos, Ball_X_Motion, Ball_Y_Pos, Ball_Y_Motion, Ball_Size;

    parameter [9:0] Ball_X_Center=320;  // Center position on the X axis
    parameter [9:0] Ball_Y_Center=240;  // Center position on the Y axis
    parameter [9:0] Ball_X_Min=0;       // Leftmost point on the X axis
    parameter [9:0] Ball_X_Max=639;     // Rightmost point on the X axis
    parameter [9:0] Ball_Y_Min=0;       // Topmost point on the Y axis
    parameter [9:0] Ball_Y_Max=479;     // Bottommost point on the Y axis
    parameter [9:0] Ball_X_Step=1;      // Step size on the X axis
    parameter [9:0] Ball_Y_Step=1;      // Step size on the Y axis

    assign Ball_Size = 4;  // assigns the value 4 as a 10-digit binary number, ie "0000000100"

    always_ff @ (posedge Reset or posedge frame_clk )
    begin: Move_Ball
        if (Reset)  // Asynchronous Reset
        begin
            Ball_Y_Motion <= 10'd0; //Ball_Y_Step;
            Ball_X_Motion <= 10'd0; //Ball_X_Step;
            Ball_Y_Pos <= Ball_Y_Center;
            Ball_X_Pos <= Ball_X_Center;
        end

        else
        begin
             if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max )  // Ball is at the bottom edge, BOUNCE!
                Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);  // 2's complement.

             else if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
                Ball_Y_Motion <= Ball_Y_Step;

              else if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )  // Ball is at the Right edge, BOUNCE!
                Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);  // 2's complement.

             else if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE!
                Ball_X_Motion <= Ball_X_Step;

             else
                Ball_Y_Motion <= Ball_Y_Motion;  // Ball is somewhere in the middle, don't bounce, just

             case (keycode)
               8'h04 : begin

                        Ball_X_Motion <= -1;//A
                        Ball_Y_Motion<= 0;
                       end

               8'h07 : begin

                        Ball_X_Motion <= 1;//D
                        Ball_Y_Motion <= 0;
                        end


               8'h16 : begin

                        Ball_Y_Motion <= 1;//S
                        Ball_X_Motion <= 0;
                       end

               8'h1A : begin
                        Ball_Y_Motion <= -1;//W
                        Ball_X_Motion <= 0;
                       end
                 default: ;
             endcase


              Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion);  // Update ball position
              Ball_X_Pos <= (Ball_X_Pos + Ball_X_Motion);


      /**************************************************************************************
        ATTENTION! Please answer the following quesiton in your lab report! Points
        Hidden Question #2/2:
           Note that Ball_Y_Motion in the above statement may have been changed at
           that is causing the assignment of Ball_Y_pos.  Will the new value of Ba
           or the old?  How will this impact behavior of the ball during a bounce,
           interact with a response to a keypress?  Can you fix it?  Give an answe
       **************************************************************************************/


      end
    end

    assign BallX = Ball_X_Pos;

    assign BallY = Ball_Y_Pos;

    assign BallS = Ball_Size;

endmodule
```

Module: Ball.sv

Inputs: Reset, Frame_Clk, Keycode[7:0]

Outputs: BallX[9:0], BallY[9:0], BallS[9:0]

Description: This controls the movement of the ball and how it moves left, right, up, down and bounce etc.

Purpose: This is done using the keycode as the Ball.sv helps the ball to move, bounce and interact with the VGA

```
#**************************************************************
# Create Clock
#**************************************************************
create_clock -period "10.0 MHz" [get_ports ADC_CLK_10]
create_clock -period "50.0 MHz" [get_ports CLOCK_50]
create_clock -period "50.0 MHz" [get_ports MAX10_CLK2_50]



# SDRAM CLK
#create_generated_clock -source [get_nets { lab_7_soc_sdram_pll_altpll_vg92:sd1:wire_pll7_clk[1] }] \
                       -name sdram_clk
                       #[get_ports {lab_7_soc_sdram_pll_altpll_vg92:sd1:wire_pll7_clk[1]}]

create_generated_clock -name sdram_clk [get_ports {DRAM_CLK}] -source [get_nets {u0|sdram_pll|sd1|wire_p

#lab_7_soc_sdram_pll_altpll_vg92:sd1:wire_pll7_clk[1]
#**************************************************************
# Create Generated Clock
#**************************************************************
derive_pll_clocks
```

```
#**************************************************************
# Set Input Delay
#**************************************************************
# suppose +- 100 ps skew
# Board Delay (Data) + Propagation Delay - Board Delay (Clock)
# max 5.4(max) +0.4(trace delay) +0.1 = 5.9
# min 2.7(min) +0.4(trace delay) -0.1 = 3.0
set_input_delay -max -clock sdram_clk 5.9 [get_ports DRAM_DQ*]
set_input_delay -min -clock sdram_clk 3.0 [get_ports DRAM_DQ*]
#set_input_delay -max -clock sdram_clk 5.9 [get_ports DRAM_DQM*]
#set_input_delay -min -clock sdram_clk 3.0 [get_ports DRAM_DQM*]



#shift-window
set_multicycle_path -from [get_clocks {sdram_clk}] -to  [get_clocks {u0|sdram_pll|sd1|pll7|clk[0]}]  -se

#**************************************************************
# Set Output Delay
#**************************************************************
# suppose +- 100 ps skew
# max : Board Delay (Data) - Board Delay (Clock) + tsu (External Device)
# min : Board Delay (Data) - Board Delay (Clock) - th (External Device)
# max 1.5+0.1 =1.6
# min -0.8-0.1 = 0.9
set_output_delay -max -clock sdram_clk 1.6  [get_ports {DRAM_DQ* DRAM_DQM*}]
set_output_delay -min -clock sdram_clk -0.9 [get_ports {DRAM_DQ* DRAM_DQM*}]
set_output_delay -max -clock sdram_clk 1.6  [get_ports {DRAM_ADDR* DRAM_BA* DRAM_RAS_N DRAM_CAS_N DRAM_WE
set_output_delay -min -clock sdram_clk -0.9 [get_ports {DRAM_ADDR* DRAM_BA* DRAM_RAS_N DRAM_CAS_N DRAM_WE
```

Module: lab8.sdc

Description: This provides constraints for the clock to be 50MHz, sets the input and output delay for SDRAM and sets a false path to exclude LED, KEYs, Switches etc.

```
// lab8_soc.v

// Generated using ACDS version 18.1 625

`timescale 1 ps / 1 ps
module lab8_soc (
		input  wire        clk_clk,                       //                    clk.clk
		output wire [15:0] hex_digits_export,             //            hex_digits.export
		input  wire [1:0]  key_external_connection_export, // key_external_connection.export
		output wire [7:0]  keycode_export,                //             keycode.export
		output wire [13:0] leds_export,                   //                leds.export
		input  wire        reset_reset_n,                 //               reset.reset_n
		output wire        sdram_clk_clk,                 //           sdram_clk.clk
		output wire [12:0] sdram_wire_addr,               //          sdram_wire.addr
		output wire [1:0]  sdram_wire_ba,                 //                    .ba
		output wire        sdram_wire_cas_n,              //                    .cas_n
		output wire        sdram_wire_cke,                //                    .cke
		output wire        sdram_wire_cs_n,               //                    .cs_n
		inout  wire [15:0] sdram_wire_dq,                 //                    .dq
		output wire [1:0]  sdram_wire_dqm,                //                    .dqm
		output wire        sdram_wire_ras_n,              //                    .ras_n
		output wire        sdram_wire_we_n,               //                    .we_n
		input  wire        spi0_MISO,                     //                spi0.MISO
		output wire        spi0_MOSI,                     //                    .MOSI
		output wire        spi0_SCLK,                     //                    .SCLK
		output wire        spi0_SS_n,                     //                    .SS_n
		input  wire        usb_gpx_export,                //             usb_gpx.export
		input  wire        usb_irq_export,                //             usb_irq.export
		output wire        usb_rst_export                 //             usb_rst.export
	);

	wire        sdram_pll_c0_clk;                                   // sdram_pll:c0 -> [mm_inte
	wire  [31:0] nios2_gen2_0_data_master_readdata;                 // mm_interconnect_0:nios2_
	wire        nios2_gen2_0_data_master_waitrequest;               // mm_interconnect_0:nios2_
	wire        nios2_gen2_0_data_master_debugaccess;               // nios2_gen2_0:debug_mem_s
	wire  [27:0] nios2_gen2_0_data_master_address;                  // nios2_gen2_0:d_address ->
	wire   [3:0] nios2_gen2_0_data_master_byteenable;               // nios2_gen2_0:d_byteenabl
	wire        nios2_gen2_0_data_master_read;                      // nios2_gen2_0:d_read -> mm
	wire        nios2_gen2_0_data_master_write;                     // nios2_gen2_0:d_write ->
```

```
	lab8_soc_hex_digits_pio hex_digits_pio (
		.clk        (clk_clk),                            //               clk.clk
		.reset_n    (~rst_controller_reset_out_reset),    //             reset.reset_n
		.address    (mm_interconnect_0_hex_digits_pio_s1_address),    //                .address
		.write_n    (~mm_interconnect_0_hex_digits_pio_s1_write),     //                .write_n
		.writedata  (mm_interconnect_0_hex_digits_pio_s1_writedata),  //                .writedata
		.chipselect (mm_interconnect_0_hex_digits_pio_s1_chipselect), //                .chipselect
		.readdata   (mm_interconnect_0_hex_digits_pio_s1_readdata),   //                .readdata
		.out_port   (hex_digits_export)                   // external_connection.export
	);

	lab8_soc_jtag_uart_0 jtag_uart_0 (
		.clk            (clk_clk),                            //               clk
		.rst_n          (~rst_controller_reset_out_reset),    //             reset
		.av_chipselect  (mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_chipselect),  // avalon_jtag_slave
		.av_address     (mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_address),     //
		.av_read_n      (~mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_read),      //
		.av_readdata    (mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_readdata),    //
		.av_write_n     (~mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_write),     //
		.av_writedata   (mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_writedata),   //
		.av_waitrequest (mm_interconnect_0_jtag_uart_0_avalon_jtag_slave_waitrequest), //
		.av_irq         (irq_mapper_receiver0_irq)            //              irq
	);

	lab8_soc_key key (
		.clk      (clk_clk),                            //               clk.clk
		.reset_n  (~rst_controller_reset_out_reset),    //             reset.reset_n
		.address  (mm_interconnect_0_key_s1_address),   //                .address
		.readdata (mm_interconnect_0_key_s1_readdata), //                .readdata
		.in_port  (key_external_connection_export)      // external_connection.export
	);

	lab8_soc_keycode keycode (
		.clk        (clk_clk),                            //               clk.clk
		.reset_n    (~rst_controller_reset_out_reset),    //             reset.reset_n
		.address    (mm_interconnect_0_keycode_s1_address),    //                .address
		.write_n    (~mm_interconnect_0_keycode_s1_write),     //                .write_n
		.writedata  (mm_interconnect_0_keycode_s1_writedata),  //                .writedata
```

Module: lab8_soc.v

Description: This is the verilog file which is generated by the platform design to get the HDL and to combine the NIOS II processor and all the modules in Platform design seen in the earlier section.

```c
BYTE SPI_wr(BYTE data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write data on SPI, simultanously read status register (byte) from MAX3421E
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return the status register
    //volatile unsigned int status[8] = (unsigned int*) 0x0008;
    //BYTE * statusRegister = (BYTE*) 0x0008;
    //BYTE baseAddress = 0;
    BYTE readData;
    int returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 1, &data, 1, &readData, 0);
    if(returnCode < 0) printf("Error in SPI_WR");
    return (readData);
}
```

```c
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //unsigned int baseAddress = 0;
    BYTE readData;
    BYTE registerVal = (reg)+2;
    BYTE com[2];
    com[0] = registerVal;
    com[1] = val;
    printf("%x", registerVal);
    int returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 2, com, 0, &readData, 0);
    //returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 1, &val, 1, &readData, ALT_AVALON_SPI_COMMAND_MERGE);
    //printf("%x \n", readData);
    if (returnCode<0) printf("Error in reg_wr");
}
//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0  print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
    BYTE regAddr = reg + 2;
    BYTE junk;
    unsigned char sh[nbytes+1];//+1
    for(int i = 0; i<nbytes+1; i++){
        if(i == 0){
            sh[0] = regAddr;
            continue;
        }
        else{
            sh[i] = data[i-1];
        }
```

```c
//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val
    //unsigned long baseAddress = 0;
    BYTE val;
    unsigned char read_data;
    BYTE regAddr = reg;
    int returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 1, &regAddr, 1, &read_data, 0);
    //returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 0, &regAddr, 1, &val, 0);
    //printf("%x \n", readData);
    if (returnCode<0) printf("Error in reg_rd");
    return read_data;
}
//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
    BYTE read_data[nbytes];
    BYTE regAddr = reg;
    int returnCode = alt_avalon_spi_command(SPI_MASTER, 0, 1, &regAddr, nbytes, data, 0);
    return data+nbytes;

}
```

Description of MAX3421E.c

This has the 5 functions where we used the alt_avalon_spi_command() to read and write from the SPI peripheral to communicate with Max3421e peripheral. We could have used an address to directly access it but the alt_avalon_spi_command() controls it better as the SPI peripheral has 32 registers, MOSI, MISO etc and does communicate better.

Functions implemented in MAX3421E.c

1) SPI_Wr: This function had just taken a single byte and it read the status register which comes in the MISO (Master Input Slave Output). So we first gave 1 byte of data to write and then read the 1 byte which will be status register. This command was the main one: alt_avalon_spi_command(SPI_MASTER, 0, 1, &data, 1, &readData, 0);

2) MAXreg_wr: This is when we had to write a value to a specific register. Here the write length because the first 8 bit will be the register and the second 8 bit will be the actual value. alt_avalon_spi_command(SPI_MASTER, 0, 2, com, 0, &readData, 0); This was the major command.
3) BYTE* MAXbytes_wr: This is where we write multiple bytes into a register. So the length is just n+1 bytes and the first byte we write on the MOSI is the register so that they can get the register and the subsequent inputs are the n bytes we want to write.
4) BYTE MAXreg_rd(BYTE reg): This is where we read the value and we just write 1 byte on the MOSI and get 1 byte from MISO.
5) Byte* MAXBytes_rd: This is multiple byte read where we read nbytes of data so we first give write the reg value and read nbytes of data provided.

**II) Extra Credit:**

Problems:

1) As mentioned in the hidden question: we saw how the <= assignment causes in an always_ff block to take the old value
2) In addition, if we look at the display, when we move to the corner, there is a possibility that it comes out of frame and there might be glitches

Causes:

1) Not enough conditional loops to make sure that we are checking for all the conditions
2) Parallel assignments in an always_ff block causes it to take old values
3) We should only change the position if we are sure that we are not in the edge cases

Solutions:

1) We need to implement some extra combinational logic, and we need to use if/else loops to better look for those edge cases that we are missing
2) We can look at varying some assignments so we can make sure that BallY_Pos and BallX_Pos takes the current and the updated location

```
if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max )  // B
    Ball_Y_Motion <= (~ (Ball_Y_Step) + 1'b1);  //

else if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )
    Ball_Y_Motion <= Ball_Y_Step;

 else if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )
    Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);  //

else if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )
    Ball_X_Motion <= Ball_X_Step;

else
    Ball_Y_Motion <= Ball_Y_Motion;  // Ball is som
```

They were made some edits in the if, else statements to make it more effective.