ECE 385

Fall 2022

Experiment #6

Lab 6

SOC with NIOS II in System Verilog

Haocheng Yang((hy38)

Yicheng Zhou(yz69)

Introduction:

a.) In this Lab, we designed and implemented an SOC using the NIOS-II processor which enables us to write and compile C code on the FPGA board. We implemented 2 functions, one to keep the LED blinking and one is an accumulator that keeps adding numbers to a sum until over flow. (6.1)

b.) In this Lab, we designed and implemented a USB ad VGA system to run a bouncing ball. The ball's direction is controlled by the keyboard inputs (WASD), and the ball will bounce back once it reached the boundary of the screen. (6.2)

Written Description and Diagrams of NIOS-II System

1.) The hardware used includes the NIOS-II processor, sdram controller, on chip memory and a clock phase shifter. The sdram controller is used to control and specify elements of the sdram. The on-chip memory is used to store values. The clock phase shifter is used to phase shift the clock signal so that sdram is read at the correct window,

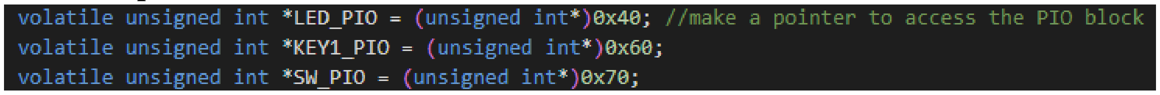
2.) To implement I/O, we defined the address for the LED PIO, where communication with hardware take place. For reset functions, a 0 is written into the memory.
For blinker function, a loop is created to set the memory for LED such that LSB is 1 and then back to 0 on repeat to blink the right most LED.
For accumulator, unsigned integer pointers are declared to accommodate the switches. Then a pause integer is declared to ensure the program only execute once with each button press. Then 'if' statement is used to implement the pause function for button press and release.

3.) To interact with the MAX3421E (usb chip), the FPGA board will declare an address into the chip to allow data transfer. Then according to the write or read signals, we can write data to the chip or read data to the host.
To interact with the VGA, a horizontal sync signal is generated to tell the display to change lines, and according the to pixel clock,

we send RGB data to display. Note the in future a vertical sync signal is also needed to tell the display the 1 frame is completed rendered and start a new frame.

4.) SPI (Avalon-ST) contains 4 ports: MOSI, MISO, sclk, ss_n.
The MOSI is the master out slave in signal to transfer data out of the master and into slaves.
The MISO is the master in slave out signal to transfer data out of the slaves in to the master.
The sclk is the serial clock which drives to slaves by master to synchronize the data bits.
The ss_n is the slave select driven by master to individual slaves to select target slave.
All of these are visible at Avalon memory mapper interface.

5.) The code modified is listed here, for more detail please see part 6 of this report.



```
volatile unsigned int *LED_PIO = (unsigned int*)0x40; //make a pointer to access the PIO block
volatile unsigned int *KEY1_PIO = (unsigned int*)0x60;
volatile unsigned int *SW_PIO = (unsigned int*)0x70;
```

6.) The VGA operation in this lab is ‘hard coded’ on the logic (as oppose to software), thus the content displayed is determined once the Verilog compile and can not be change in runtime. The ball only accepts interrupts form keyboard to change direction. The color mapper is what assigns the RGB values to the background and the ball. The VGA controller has its own clock which run at 50% frequency of the system clock dur to the refresh rate constrain of the monitor.

Top Level Block Diagram:

The RTL diagram is shown below:

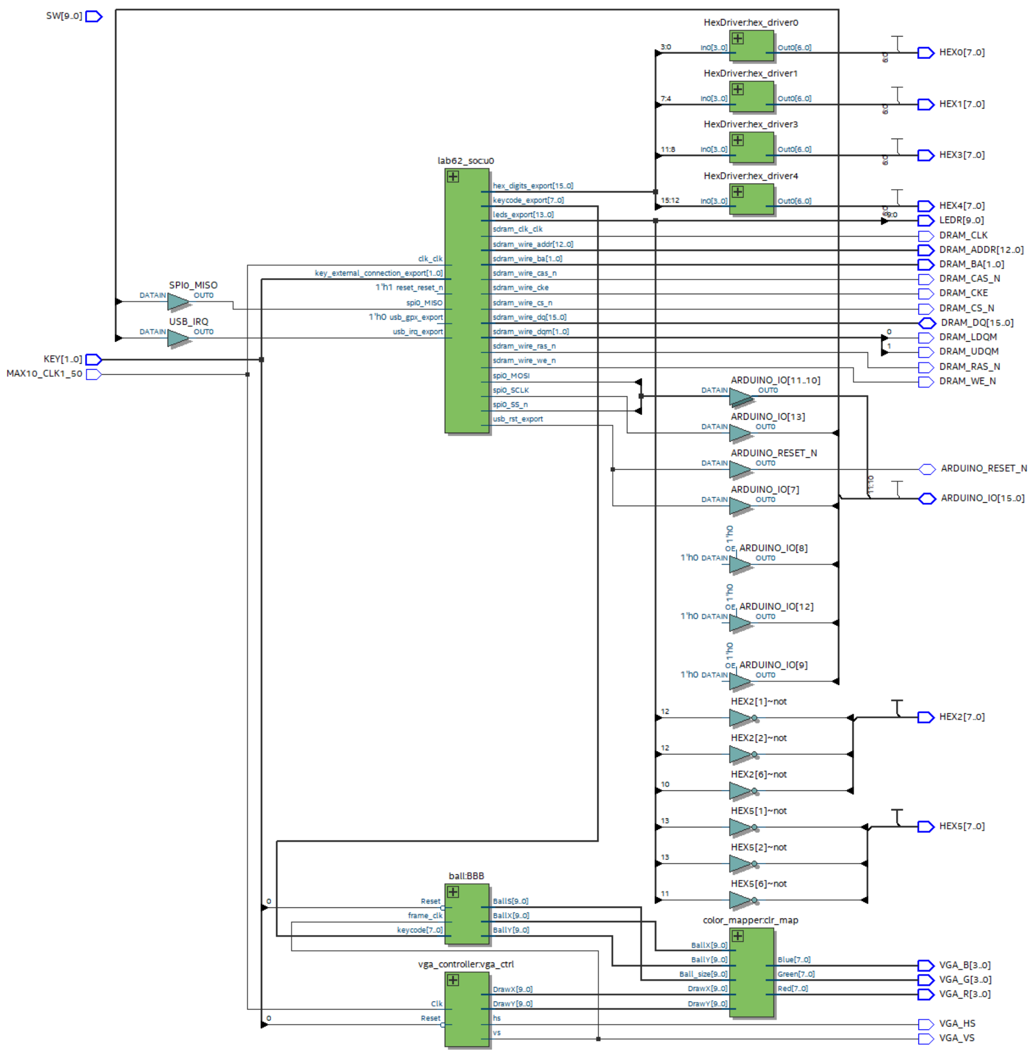


Figure 1: RTL view for the system.

Written Description of all .sv Modules

1.) VGA_controller.sv:

Inputs: clk, Reset

Output: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY

Description: this is the VGA controller which sends signals through VGA port to the monitor.

Purpose: This is the module which determines the boundary of the screen, with it's on board clock, generates the pixel clock. It also generates the horizontal and vertical sync.

2.) Lab62.sv:

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0]SW, [15:0]DRA,_DQ, [15:0] ARDUINO_IO, Arduino_reset_n.

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_LDQMM DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0]VGA_R, [3:0]VGA_G, [3:0] VGA_B

Description: This is the top level entity of lab 6

Purpose: this calls on all other modules for functions and bridge data between other modules.

3.) HexDriver.sv:

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module converts 4bit-binry input and convert it to 7 bit for hex-display.

Purpose: This module converts 4bit binary numbers from registers into hexadecimal so LEDs can display outputs.

4.) Color_Mapper.sv:

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

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

Description: This module generates the RGB values for each pixel for monitor.

Purpose: This modules takes in inputs and determine if the pixel being drawn is on the ball or not.

5.) Ball.sv:

Inputs: Reset, frame_clk [7:0]keycode

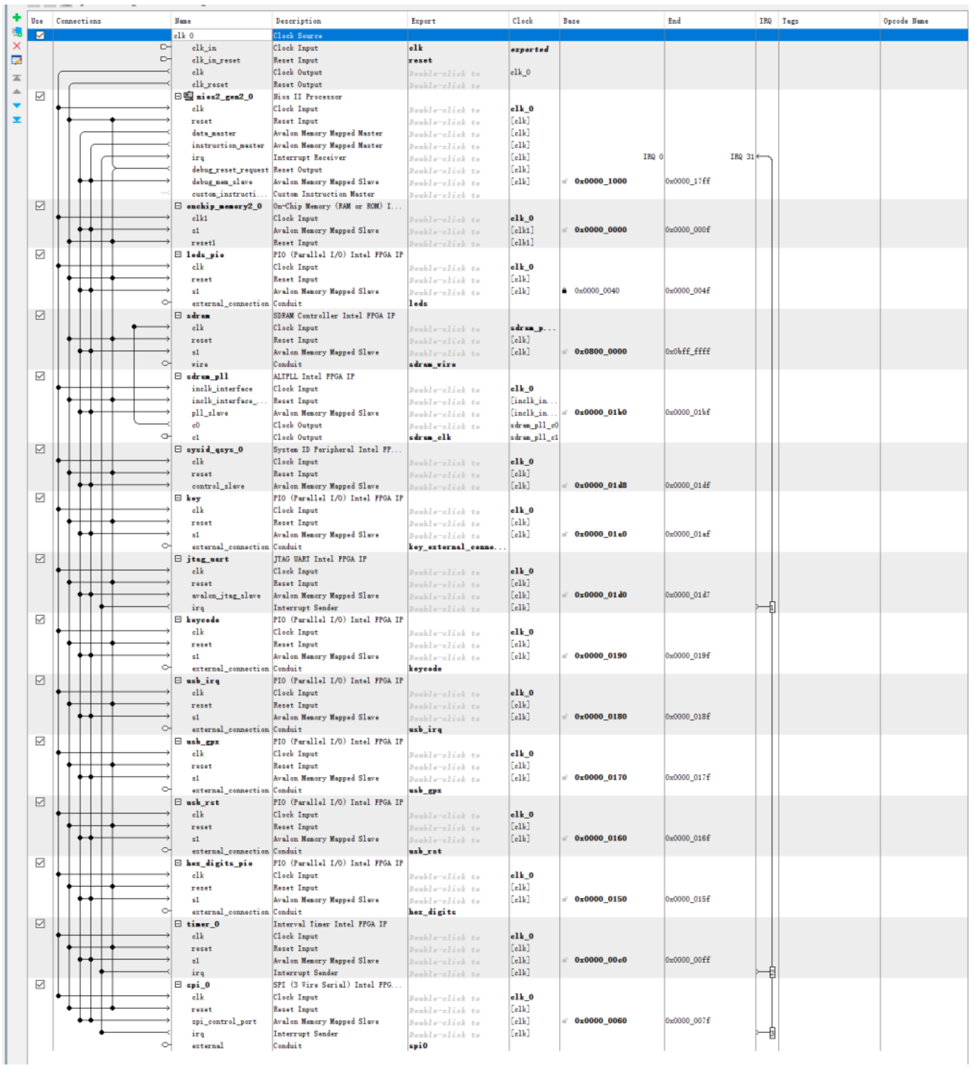
Outputs: [9:0] BallX, BallY, BallS

Description: This is the logic for the ball which takes in keyboard inputs.

Purpose: This module takes in keyboard inputs to change direction of the ball. It also takes in reset signal from the button on FPGA board to reset the ball's motion. This also determines if the ball is at screen boundary and bounce it back (reverse the ball's direction)

System Level Block Diagram

Platform Design View for lab62_soc.qsys:



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		clk 0	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	Double-click to	clk_0					
		clk_reset	Reset Output	Double-click to						
✓		**nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		data_master	Avalon Memory Mapped Master	Double-click to	[clk]					
		instruction_master	Avalon Memory Mapped Master	Double-click to	[clk]					
		irq	Interrupt Receiver	Double-click to	[clk]		IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	Double-click to	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_1000**	0x0000_17ff			
		custom_instructi...	Custom Instruction Master	Double-click to						
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) I...							
		clk1	Clock Input	Double-click to	**clk_0**					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	**0x0000_0000**	0x0000_000f			
		reset1	Reset Input	Double-click to	[clk1]					
✓		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0000_0040	0x0000_004f			
		external_connection	Conduit	**leds**						
✓		**sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	Double-click to	**sdram_p...**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0800_0000**	0x0bff_ffff			
		wire	Conduit	**sdram_wire**						
✓		**sdram_pll**	ALTPLL Intel FPGA IP							
		inclk_interface	Clock Input	Double-click to	**clk_0**					
		inclk_interface_...	Reset Input	Double-click to	[inclk_in...					
		pll_slave	Avalon Memory Mapped Slave	Double-click to	[inclk_in...	**0x0000_01b0**	0x0000_01bf			
		c0	Clock Output	Double-click to	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
✓		**sysid_qsys_0**	System ID Peripheral Intel FP...							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		control_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_01d8**	0x0000_01df			
✓		**key**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_01a0**	0x0000_01af			
		external_connection	Conduit	**key_external_conne...**						
✓		**jtag_uart**	JTAG UART Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_01d0**	0x0000_01d7			
		irq	Interrupt Sender	Double-click to	[clk]			1		
✓		**keycode**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0190**	0x0000_019f			
		external_connection	Conduit	**keycode**						
✓		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0180**	0x0000_018f			
		external_connection	Conduit	**usb_irq**						
✓		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0170**	0x0000_017f			
		external_connection	Conduit	**usb_gpx**						
✓		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0160**	0x0000_016f			
		external_connection	Conduit	**usb_rst**						
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0150**	0x0000_015f			
		external_connection	Conduit	**hex_digits**						
✓		**timer_0**	Interval Timer Intel FPGA IP							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_00c0**	0x0000_00ff			
		irq	Interrupt Sender	Double-click to	[clk]			2		
✓		**spi_0**	SPI (3 Wire Serial) Intel FPG...							
		clk	Clock Input	Double-click to	**clk_0**					
		reset	Reset Input	Double-click to	[clk]					
		spi_control_port	Avalon Memory Mapped Slave	Double-click to	[clk]	**0x0000_0060**	0x0000_007f			
		irq	Interrupt Sender	Double-click to	[clk]			3		
		external	Conduit	**spi0**						

Brief description:

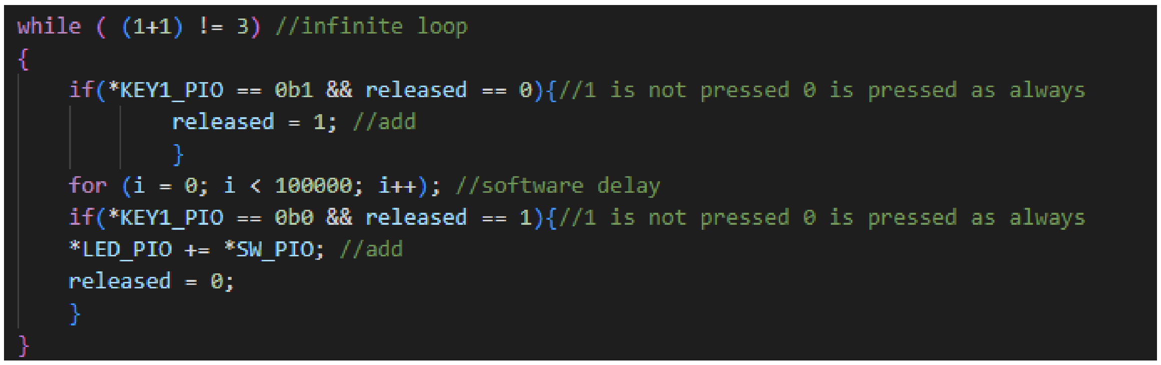
1. Clk_0: global clock signal, connects to most subsystems on this platform, with the exception of the sdram module using the c0 output of sdram_pll with a -1ns phase shift to compensate for the short time window of the sdram when it generates a valid value for read/write. This module also serves as an reset signal for most modules. The module takes input from external connections, where in the top level we connect to buttons.
2. Nios_gen2_0: Nois II processor. Using the /e variant, this is a economical version of the processor. This module supports JTAG Debug and ECC RAM Protection only, as opposed to many other features offered in the /f version. This module is the processor of the system as a modified Harvard machine, providing 2 buses, data_master and instruction_master, where they are the bus for data and instruction. Notice this module also have an Interrupt Receiver where this drives the Keyboard Inputs.
3. Onchip_memory_2_0: On chip RAM/ROM.
4. Leds_pio: An output pio connection, serves to light the LEDs in the accumulator design, was not removed for convenience.
5. Sdram: Memory of whole system, a 16bits, 1 CS, 4 Bank memory, with 13 Row and 10 Column of addresses.
6. Sdram_pll: This module outputs a special, phase shifted clock for sdram, the reason for us to use this feature in this module is described in Clk_0.
7. Sysid_qsys_0: This is a System ID peripheral which generates a System ID per connection.
8. Key: This is a 2-bit width pio input, reading the buttons on the FPGA board, key0, and key1, positioned on the side of the board.
9. Jtag_uart: A block to support the communication with the FPGA with the computer regarding software data.
10. Keycode: This is an 8-bit pio Output where the 7 segment display will display the keyboard input in hexadecimal numbers. Note: this is the number, not the actual display driver.
11. Usb_irq, Usb_gpx, Usb_rst: Interrupt request line: generates a interrupt when there is data to transfer in this serial bus, gpx and rst was not introduced in the lab manual, but they are pio input and output with 1 bit width, respectively.

12. Hex_digits_pio: 7 segment display output pio port. Works with the HexDriver.sv.
13. Timer_0: Interval timer with 1ms timeout.
14. Spi_0: a 3 wire serial communication module.

Describe in words the software component of the lab

1. Code for accumulator:

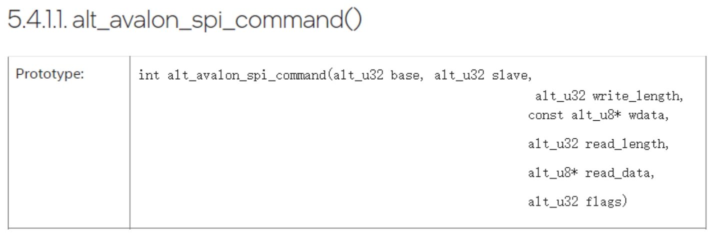
Accumulator adds the input of switch, notice the global variable released, whenever the key is released, released resets to 1, which allows the hardware to continue adding, instead of adding repeatedly while continuously pressing the button. This also helps dealing with oscillation of input key. Also notice that when the key is not pressed, the KEY PIO is 1, this is just why we always invert input in our testbenches.



```
while ( (1+1) != 3) //infinite loop
{
    if(*KEY1_PIO == 0b1 && released == 0){//1 is not pressed 0 is pressed as always
            released = 1; //add
            }
    for (i = 0; i < 100000; i++); //software delay
    if(*KEY1_PIO == 0b0 && released == 1){//1 is not pressed 0 is pressed as always
    *LED_PIO += *SW_PIO; //add
    released = 0;
    }
}
```

2. Code for USB to SPI:

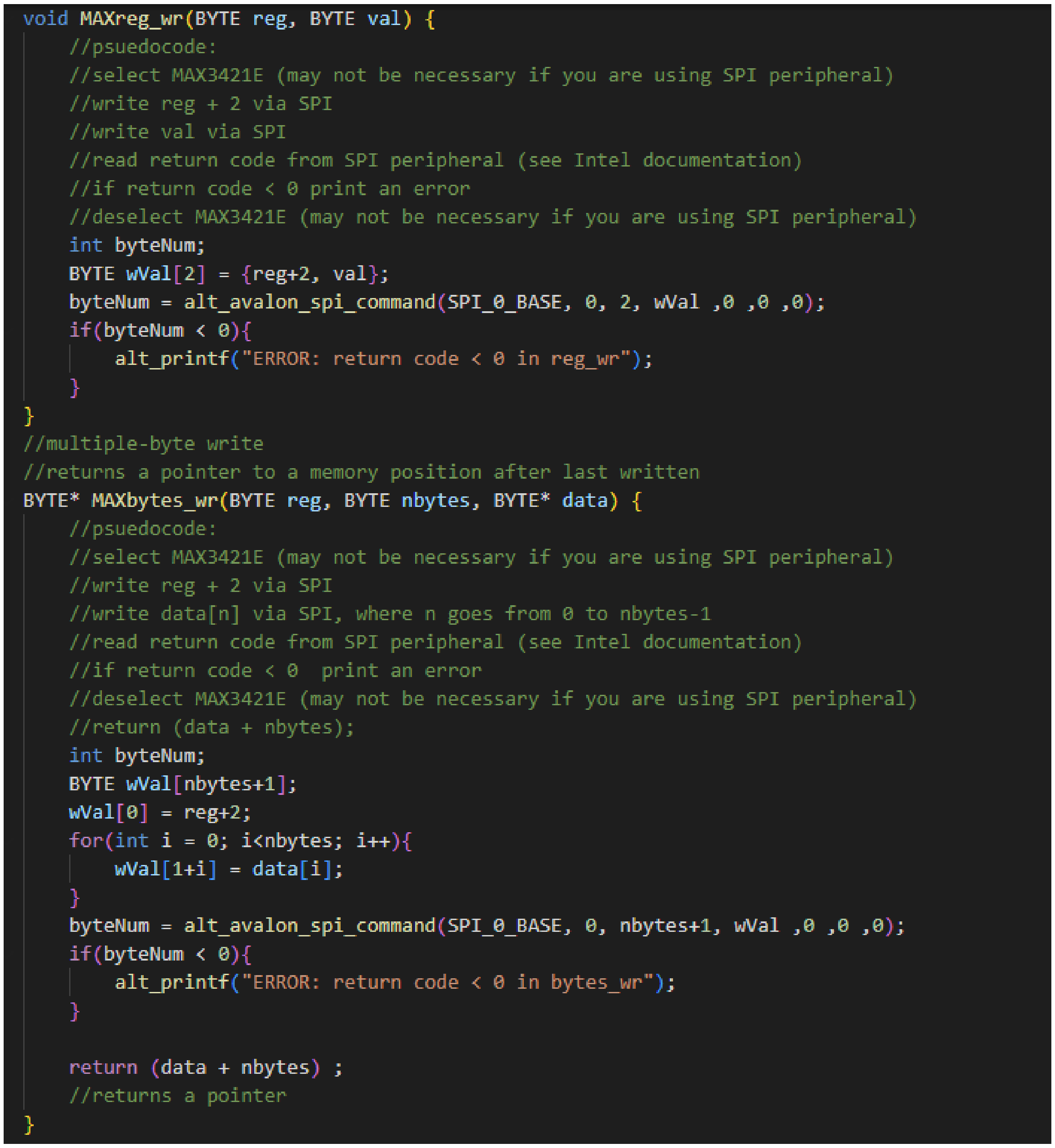
All lister methods writes/reads 1 byte/multiple bytes worth of data to any register/s via SPI using the method alt_avalon_spi_command() documentation was found here to assist development:



5.4.1.1. alt_avalon_spi_command()

Prototype:	`int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8* wdata, alt_u32 read_length, alt_u8* read_data, alt_u32 flags)`

https://www.intel.com/content/www/us/en/docs/programmable/683130/22-1/alt-avalon-spi-command.html



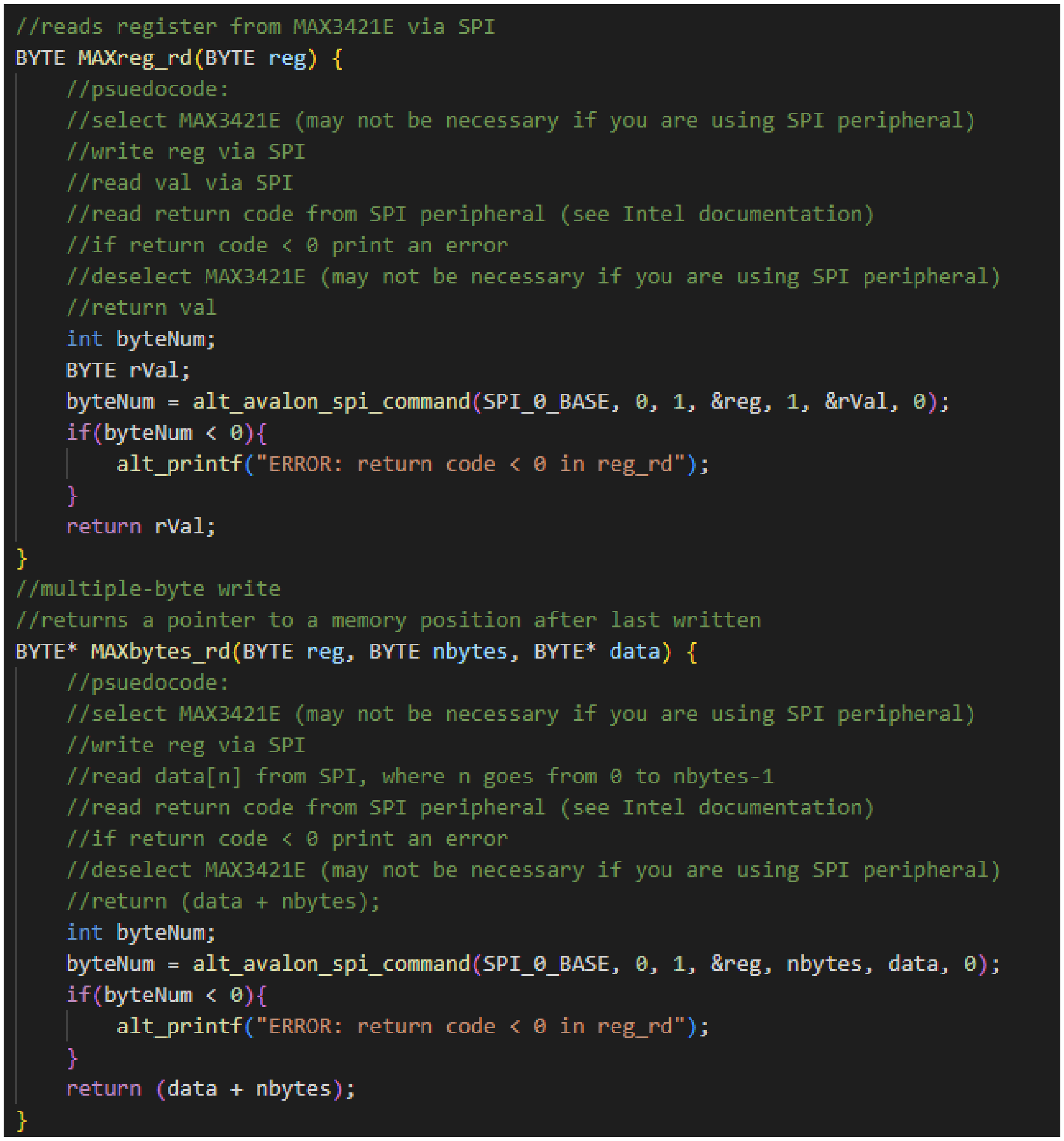
```
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    int byteNum;
    BYTE wVal[2] = {reg+2, val};
    byteNum = alt_avalon_spi_command(SPI_0_BASE, 0, 2, wVal ,0 ,0 ,0);
    if(byteNum < 0){
        alt_printf("ERROR: return code < 0 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);
    int byteNum;
    BYTE wVal[nbytes+1];
    wVal[0] = reg+2;
    for(int i = 0; i<nbytes; i++){
        wVal[1+i] = data[i];
    }
    byteNum = alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, wVal ,0 ,0 ,0);
    if(byteNum < 0){
        alt_printf("ERROR: return code < 0 in bytes_wr");
    }

    return (data + nbytes) ;
    //returns a pointer
}
```

The register write function MAXreg_wr() writes data via the Avalon bus, and detects error with the feedback return value from the variable byteNum.

The multiple byte write function MAXbytes_wr() preprocesses the data into the packet wVal[n+1] and then writes data via the Avalon bus, and detects error with the feedback return value from the variable byteNum. Returns a pointer to the address at the end of the wrote data.

The register read function MAXreg_rd() reads data via the Avalon bus, and detects error with the feedback return value from the variable byteNum.

The multiple byte read function MAXbytes_rd() reads data via the Avalon bus, and detects error with the feedback return value from the variable byteNum. Returns a pointer to the address at the end of the read data.



```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
    int byteNum;
    BYTE rVal;
    byteNum = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, &rVal, 0);
    if(byteNum < 0){
        alt_printf("ERROR: return code < 0 in reg_rd");
    }
    return rVal;
}
//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);
    int byteNum;
    byteNum = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, nbytes, data, 0);
    if(byteNum < 0){
        alt_printf("ERROR: return code < 0 in reg_rd");
    }
    return (data + nbytes);
}
```

Answers to all INQ & Post lab questions

What are the differences between the Nios II/e and Nios II/f CPUs:

The NIOS-II/e is more economical thus uses less resources but is also slower.

What advantage might on-chip memory have for program execution?

On-chip memory benefits from the shorter data path and can transfer data faster.

Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?

Our implementation is a modified Harvard since the instruction memory is accessible as data and the addresses are changeable.

For pure Harvard: data and instructions uses separate bus.

For Von Neumann: data and instructions uses same bus.

For Modified Harvard: the instruction memory is accessible as data and the addresses are changeable.

Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?

The LED is a output only peripheral, however the on-chip memory requires signals to know when to read/write data.

Why does SDRAM require constant refreshing?

The electrons in the ram will decay over time and cause data lose if not refreshed.

What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

1/(5.5)*32=727Mb/s

The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

If the SDRAM is run at too low of clock speed, the electron inside will decay and cause a voltage drop which may lead to incorrect reading.

Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind of the controller clock (clk c0). Why do we need to do this?

A delay is needed for SDRAM due to controller needing time to access data and wait for control signal to reach its steady state.

What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?

It starts from x0200 000 of SDRAM. To avoid encroaching on memory data.

Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean?

segment	meaning	example
.bss	region of uninitialized parameter	int i;
.heap	dynamically allocated parameter	malloc(int);
.rodata	region of read only constant data	const int i =1
.rwdata	region of read & write data	int i = 1;
.stack	allocated data and function calls	int func (int i, int j)
.text	string	char A = "STRING"

Document the Design Resources and Statistics:

LUT #	4369
DSP #	8
BRAM #	11392 bits
FF #	2486
Frequency	125.34MHz
Static Power	96.18mW
Dynamic Power	0.86mW
Total Power	106.35mW (9.31mW from IO)

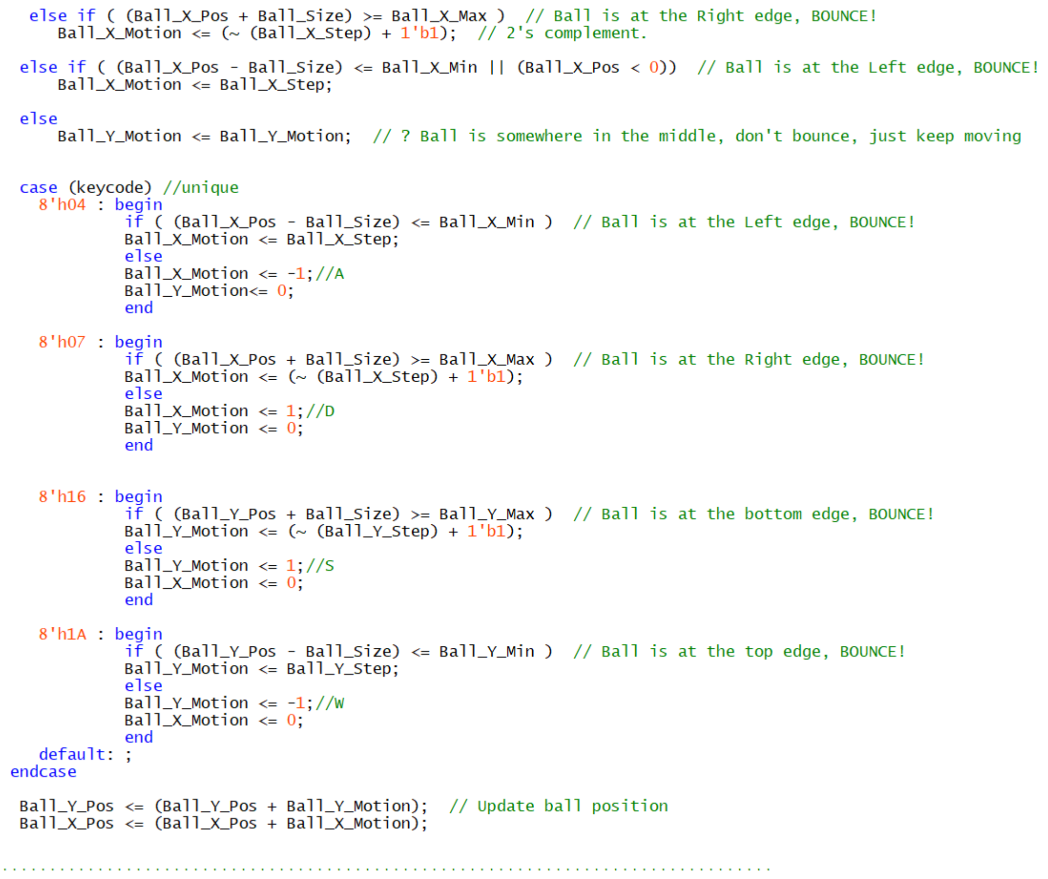
Extra credit:

We noticed that not only the ball will fly out of bounds on the top, it will also do it on the left. It will also fly in diagonal direction (ie x speed and y speed both equal 1)

We initially thought that this is caused the boundary condition contradicting keyboard input ,so we made the keyboard instruction unique case. However, this fails to fix the problem due to it only limiting the keyboard input to 1 which it already was. (green circle)

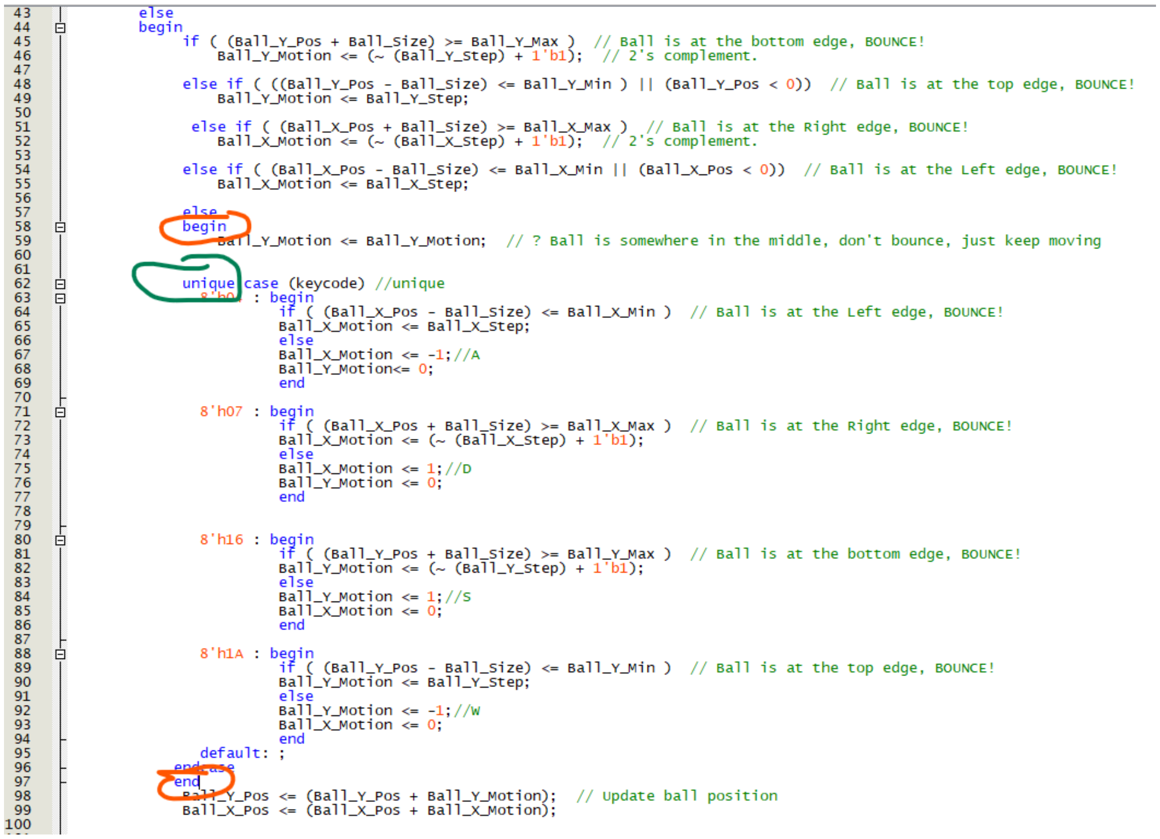
Then we modify the code so that the ball only takes keyboard instruction when it’s not at boundary, which solve the overlapping instruction causing the bugs. (red circle)

The codes are shown below



```
 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_X_Pos < 0))  // 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 keep moving


case (keycode) //unique
  8'h04 : begin
           if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE!
           Ball_X_Motion <= Ball_X_Step;
           else
           Ball_X_Motion <= -1;//A
           Ball_Y_Motion<= 0;
           end

  8'h07 : begin
           if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )  // Ball is at the Right edge, BOUNCE!
           Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);
           else
           Ball_X_Motion <= 1;//D
           Ball_Y_Motion <= 0;
           end


  8'h16 : 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);
           else
           Ball_Y_Motion <= 1;//S
           Ball_X_Motion <= 0;
           end

  8'h1A : begin
           if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
           Ball_Y_Motion <= Ball_Y_Step;
           else
           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);
```

Code before fix



```
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_Y_Pos < 0))  // 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_X_Pos < 0))  // Ball is at the Left edge, BOUNCE!
         Ball_X_Motion <= Ball_X_Step;

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


     unique case (keycode) //unique
       8'h04 : begin
                if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min )  // Ball is at the Left edge, BOUNCE!
                Ball_X_Motion <= Ball_X_Step;
                else
                Ball_X_Motion <= -1;//A
                Ball_Y_Motion<= 0;
                end

       8'h07 : begin
                if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max )  // Ball is at the Right edge, BOUNCE!
                Ball_X_Motion <= (~ (Ball_X_Step) + 1'b1);
                else
                Ball_X_Motion <= 1;//D
                Ball_Y_Motion <= 0;
                end


       8'h16 : 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);
                else
                Ball_Y_Motion <= 1;//S
                Ball_X_Motion <= 0;
                end

       8'h1A : begin
                if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min )  // Ball is at the top edge, BOUNCE!
                Ball_Y_Motion <= Ball_Y_Step;
                else
                Ball_Y_Motion <= -1;//W
                Ball_X_Motion <= 0;
                end
       default: ;
    endcase
    end
     Ball_Y_Pos <= (Ball_Y_Pos + Ball_Y_Motion);  // Update ball position
     Ball_X_Pos <= (Ball_X_Pos + Ball_X_Motion);
```

code after fix.

Conclusion:

We did not encounter significant difficulty besides the ball flying out of bound which is part of extra credit and was described in detail above.

The instructions given were sufficient.