SOC with NIOS II in SystemVerilog

Hongbo Zheng, Yuhao Yuan

Summer 2021

ECE 385

Section AB1

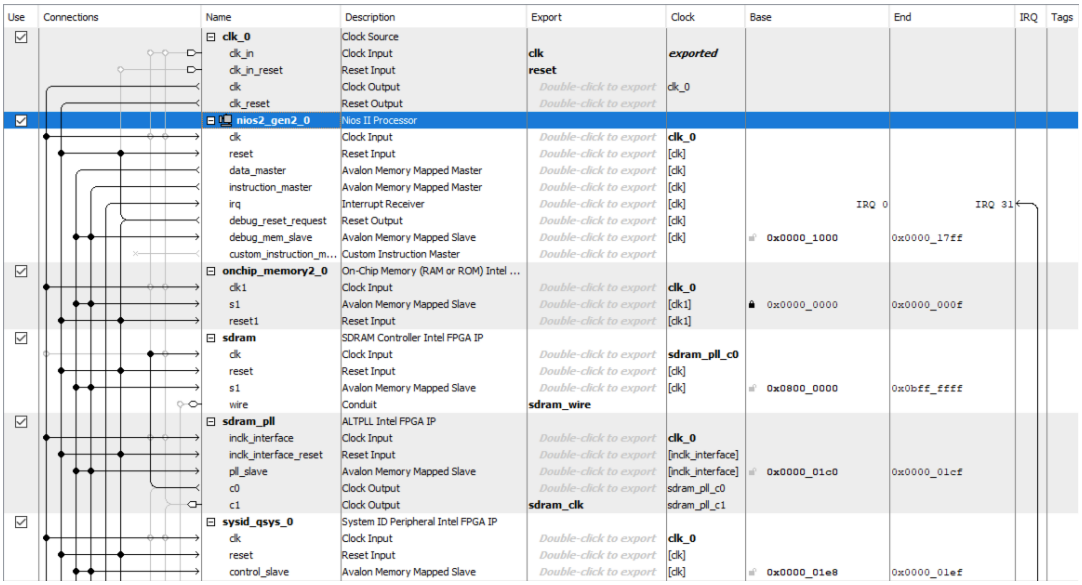
TA: Hanfei Wang

Introduction

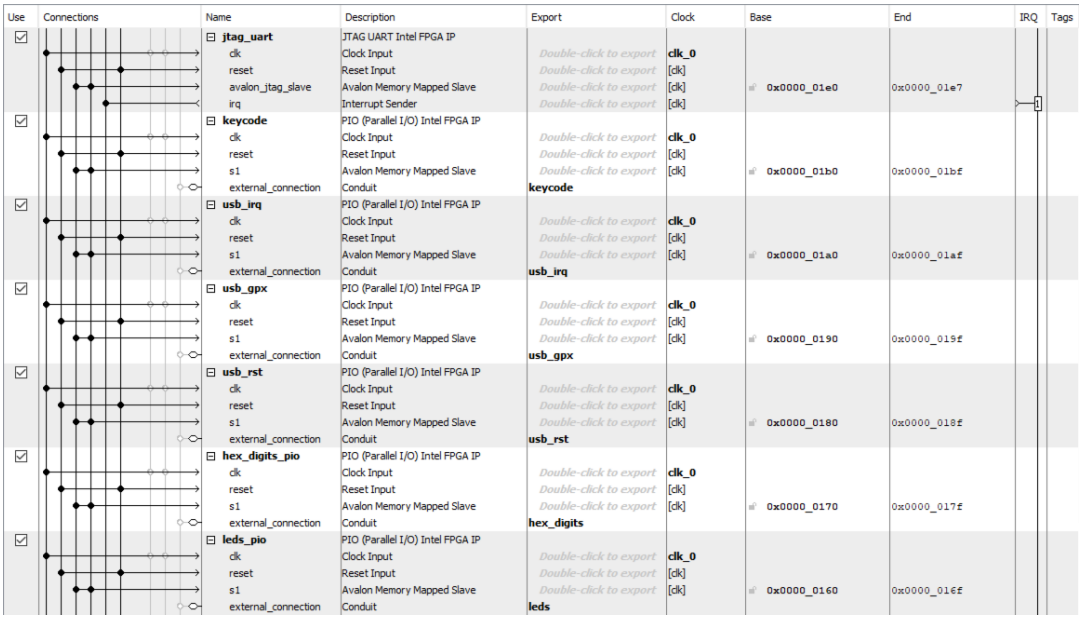
The purpose of this lab is to first learn the fundamentals of memory-mapped I/Os and then implement a simple SOC (System-On-Chip) interfacing with peripherals. The lab requires to configure a basic SOC design with a CPU, memory, and some basic peripherals: LEDs and on-board switches. Then, the system is extended with USB and VGA peripherals in order to display and control the moving direction of a bouncing ball with 4 keys: W, A, S, D. Finally, an C program is written to allow the USB driver running on the Nios II to communicate with the MAX3421E via the SPI peripheral.

Diagrams and Written Description of Nios-II System

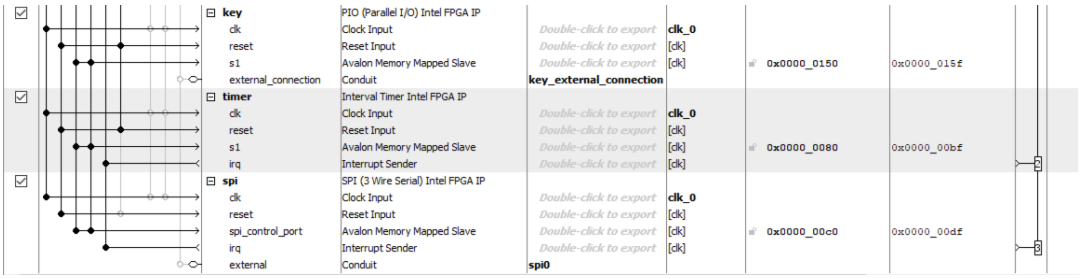
Platform Designer Module (Hardware Component)



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
☑		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	***exported***				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
☑		**nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		irq	Interrupt Receiver	*Double-click to export*	[clk]			IRQ 0	IRQ 31
		debug_reset_request	Reset Output	*Double-click to export*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_17ff		
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*					
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...						
		clk1	Clock Input	*Double-click to export*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f		
		reset1	Reset Input	*Double-click to export*	[clk1]				
☑		**sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**sdram_pll_c0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_0000	0x0bff_ffff		
		wire	Conduit	**sdram_wire**					
☑		**sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interface]				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interface]	0x0000_01c0	0x0000_01cf		
		c0	Clock Output	*Double-click to export*	sdram_pll_c0				
		c1	Clock Output	**sdram_clk**	sdram_pll_c1				
☑		**sysid_qsys_0**	System ID Peripheral Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01e8	0x0000_01ef		



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
☑		**jtag_uart**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01e0	0x0000_01e7		
		irq	Interrupt Sender	*Double-click to export*	[clk]				
☑		**keycode**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01b0	0x0000_01bf		
		external_connection	Conduit	**keycode**					
☑		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01a0	0x0000_01af		
		external_connection	Conduit	**usb_irq**					
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0190	0x0000_019f		
		external_connection	Conduit	**usb_gpx**					
☑		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0180	0x0000_018f		
		external_connection	Conduit	**usb_rst**					
☑		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0170	0x0000_017f		
		external_connection	Conduit	**hex_digits**					
☑		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0160	0x0000_016f		
		external_connection	Conduit	**leds**					



The hardware component is consisted of the Nios II/e CPU, on-chip memory, SDRAM controller, PLL for SDRAM, System ID Peripheral, and several PIOs (Parallel I/O). The Nios II/e CPU is a simple version of Nios II CPUs because it has less features compared to the Nios II/f CPU. However, it is still capable of performing the tasks: flickering LED and bouncing ball. The on-chip memory is instantiated as a placeholder block. The valuable on-chip memory is saved because it is physically closer to the processor which makes the data manipulation faster. Therefore, the Nios II programs are executed from the DRAM instead of the on-chip memory. Since the on-chip memory has limited storage capacity, the off-chip SDRAM is used to store the software program. The SDRAM controller IP core is used to interface the SDRAM to the Avalon bus because the SDRAM cannot be interfaced to the bus directly. The PLL for SDRAM component is added to provide required clock signal for the SDRAM because the SDRAM requires precise clock signal for the SDRAM chip. Moreover, PLL can compensate for clock skew due to the board layout. The System ID Peripheral is added to ensure the compatibility between hardware and software. The module gives a serial number which the software loader checks against when the software starts. This prevents loading software onto an FPGA which has an incompatible Nios II configuration. Finally, several PIOs are added to drive the LEDs (flickering LED lab) or the keyboard (bouncing ball lab).

PIO Name: keycode

Inputs: clk, reset, [15:0] data_master

Output: keycode

Address: x01b0 – x01bf

Purpose: It takes the data from the keyboard and send it to FPGA.

PIO Name: usb_irq

Inputs: clk, reset, [15:0] data_master

Output: usb_irq

Address: x01a0 – x01af

Purpose: It is used to send interrupt request of the keyboard.

PIO Name: usb_gpx

Inputs: clk, reset, [15:0] data_master

Output: usb_gpx

Address: x0190 – x019f

Purpose: It is used to send input data of the keyboard.

PIO Name: usb_rst

Inputs: clk, reset, [15:0] data_master

Output: usb_gpx

Address: x0180 – x018f

Purpose: It is used to reset output of the usb.

PIO Name: hex_digits_pio

Inputs: clk, reset, [15:0] data_master

Output: hex_digits

Address: x0170 – x017f

Purpose: It is used as the hexadecimal output.

PIO Name: leds_pio

Inputs: clk, reset, [15:0] data_master

Output: leds

Address: x0160 – x016f

Purpose: It is used as the output of the LEDs.

PIO Name: key

Inputs: clk, reset, [15:0] data_master

Output: key_external_connection

Address: x0150 – x015f

Purpose: It is used as the input of key information from the keyboard.

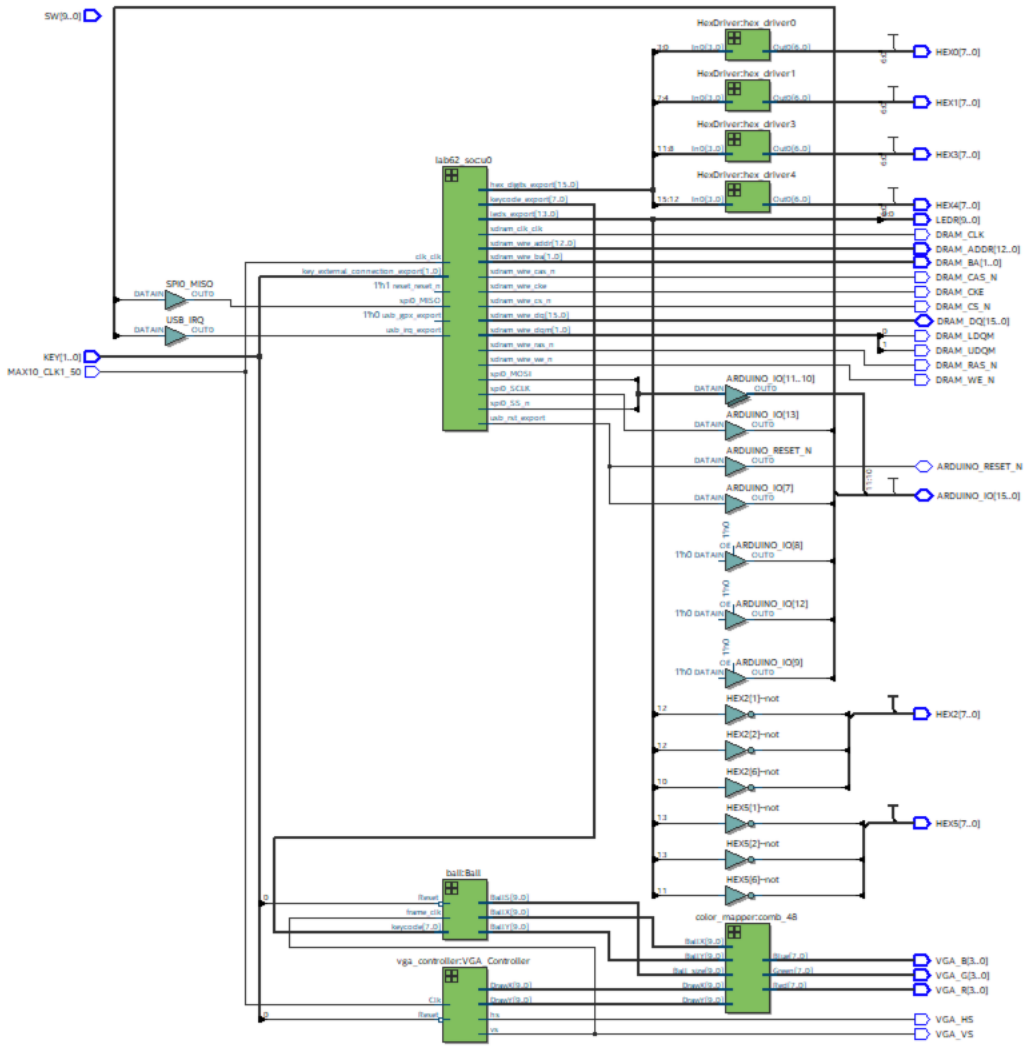
SPI protocol is an interface with synchronous and serial functionality for communication. It acts as a bus that is commonly been used to transport data between a master device(controller) and multiple slave devices(peripherals), such as sensors and shift registers. It has separate transmission lines for both clock

and data. There are 2 sets of data lines, “Controller-Out Peripheral-In” and “Controller-in Peripheral-Out”, so the protocol can both send and receive data. It also has a selecting line (Chip Select) to “wake up” the corresponding peripherals to send and receive data.

The Ball.sv module controls the position and the motion of the ball displayed on the screen. The position information is sent to the Color_Mapper.sv in order to decide what color to draw at what position. In addition, the VGA_controller sends the out the DrawX and DrawY signals to tell the Color_Mapper what positon it is currently drawing on the VGA monitor. It also sends out the vs (vertical sync) signal to Ball.sv in order to update the position of the ball whenever the VGA monitor gets refreshed. Finally, the RGB values of each pixel are sent to the VGA monitor while the VGA_controller is drawing the VGA monitor from the left to right and then from the top to bottom.

Block Diagrams and Module Descriptions

Lab62.sv Block Diagram



Module Description

Module: lab62.sv

Input: MAX10_CLK1_50, [1:0] KEY, [9:0] SW

Output: [9:0] LEDR, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, DRAM_CLK, DRAM_CKE, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [15:0] DRAM_DQ, DRAM_LDQM, 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

Inout: [15:0] ARDUINO_IO, ARDUINO_RESET_N

Description: This component contains all the components connected with each other, such as ball, vga_controller, color_mapper, and lab62_soc.

Purpose: The module is used as the top-level of the bouncing ball project. It is able to display an orange ball bouncing horizontally and vertically on the screen every time when the ball touches the edges of the screen.

lab62_soc.sv High-Level Block Diagram

Module Description

Module: lab62_soc.v

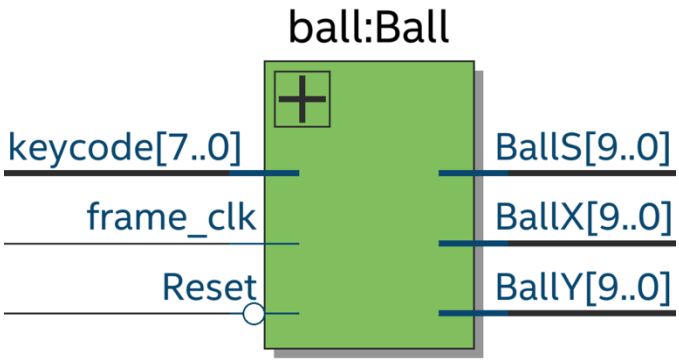
Input: clk_clk, [1:0] key_external_connection_export, reset_reset_n, spi0_MISO, usb_gpx_export, usb_irq_export

Output: [15:0] hex_digits_export, [7:0] keycode_export, [13:0] leds_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [15:0] sdram_wire_dq, [15:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export

Description: This component is generated by the Platform Designer which is the Nios II system. It contains Nios II/e CPU, on-chip memory, SDRAM controller, PLL for SDRAM, System ID Peripheral, and several PIOs (Parallel I/O).

Purpose: This module is the Nios II system which is generated from the Platform Designer. It is used as the main control module of the entire lab.

ball.sv High-Level Block Diagram



Module Description

Module: ball.sv

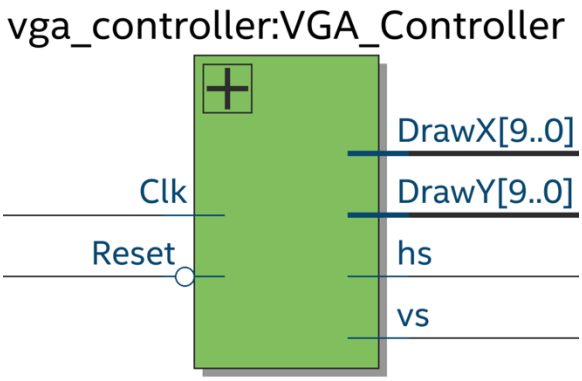
Input: [7:0] keycode, frame_clk, Reset

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

Description: This component controls the position and the motion of the ball. The position of the ball is controlled by the [9:0] BallX and [9:0] BallY. The motion of the ball is controlled by the internal signal [9:0] Ball_X_Motion and [9:0] Ball_Y_Motion. In addition, the motion of the ball can also be affected by the keycode which is the key pressed by the user.

Purpose: This module is used to display the ball on the VGA screen, and it also controls the position and the motion of the ball.

vga_controller.sv High-Level Block Diagram



Module Description

Module: vga_controller.sv

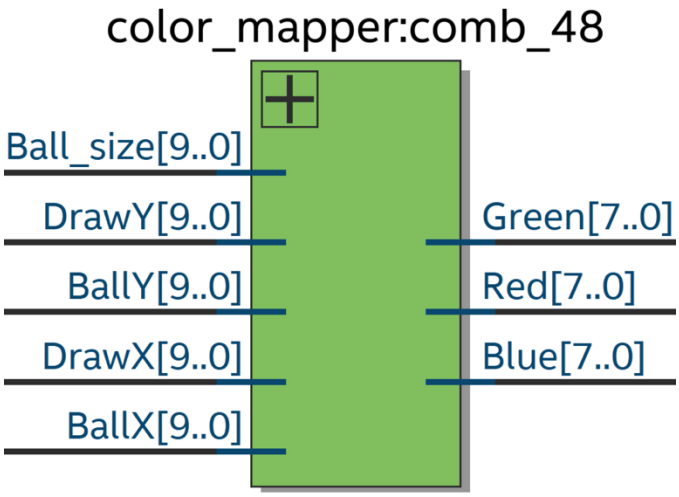
Input: Clk, Reset

Output: [9:0] DrawX, [9:0] DrawY, hs, vs

Description: This component has [9:0] DrawX and [9:0] DrawY to output the current drawing position on the VGA monitor. hs is the horizontal sync pulse, and vs is the vertical sync pulse. The component controls the electron beam inside the VGA monitor which scans from the left to the right end of each row and then scans from the top row to the bottom row.

Purpose: This module is used to control the VGA monitor. It tells the computer when to refresh the screen which draws the entire screen row by row with new content.

Color_Mapper.sv High-Level Block Diagram



Module Description

Module: Color_Mapper.sv

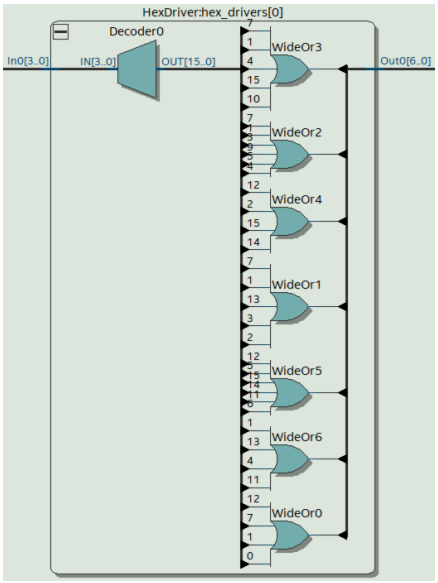
Input: [9:0] Ball_size, [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY

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

Description: This component has [7:0] Red, [7:0] Green, and [7:0] Blue as its outputs which assign the color to each pixel on the VGA monitor.

Purpose: The module is used to draw the entire VGA screen with color.

HexDriver Block Diagram



Module Description

Module: HexDriver.sv

Input: [3:0] In0

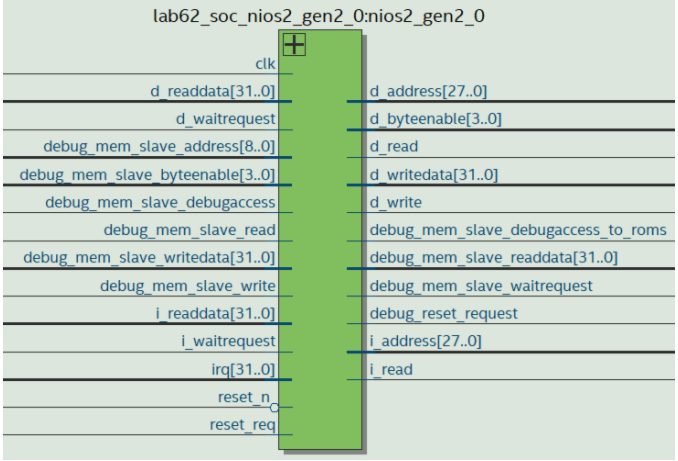
Output: [6:0] Out0

Description: This component contains a decoder which decodes the each 4-bit binary value to its corresponding 7-bit binary control value of LED segments. The input [3:0] In0 is translate to 7-bit control value of LED segments [6:0] Out0 which is capable of displaying the hexadecimal letter (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) on the HEX units.

Purpose: This module is used as 7-Segment Display which can show the 4-bit hexadecimal (2-bit hexadecimal value representing 8-bit binary value in register A and 2-it hexadecimal value representing 8-bit binary value in register B) values to the user on the 4 HEX units.

System Level Block Diagram

lab62_soc_nios2_gen2_0 High-Level Block Diagram



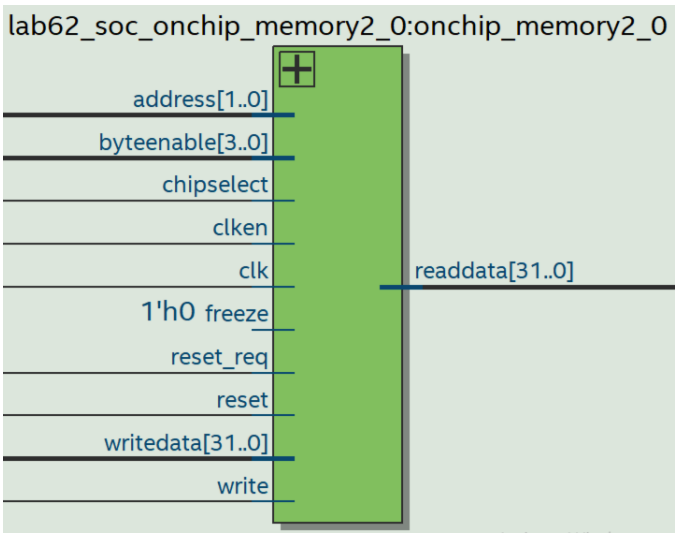
Module Description

Module: lab62_soc_nios2_gen2_0.sv

Description: This component is Nios II CPU.

Purpose: It is used as the center processing unit of the project.

lab62_soc_onchip_memory2_0 High-Level Block Diagram



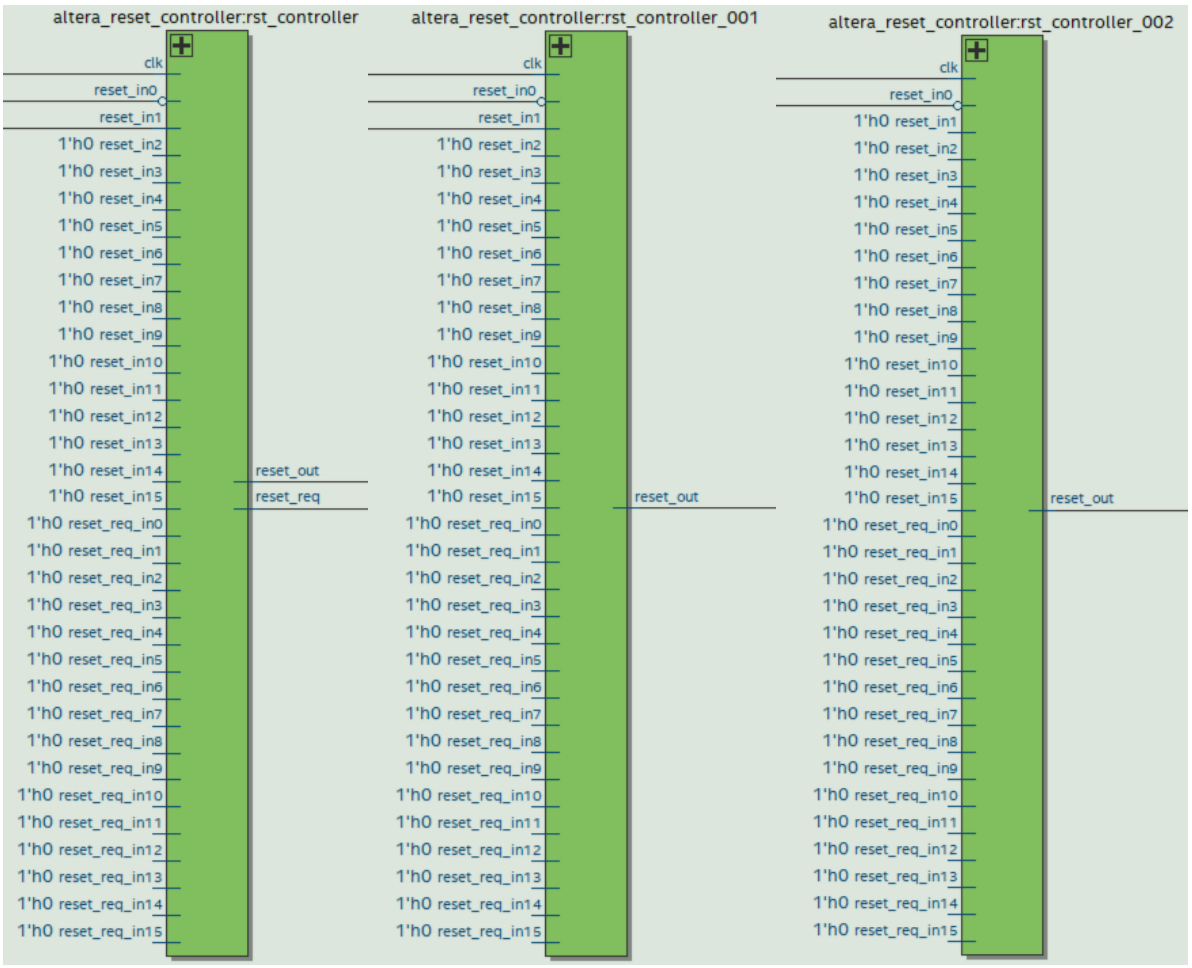
Module Description

Module: lab62_soc_onchip_memory2_0.sv

Description: This component is the on-chip memory of the Nios II system.

Purpose: It is used as a placeholder which is not used in this lab.

altera_reset_conotroller High-Level Block Diagram



Module Description

Module: altera_reset_controller.sv

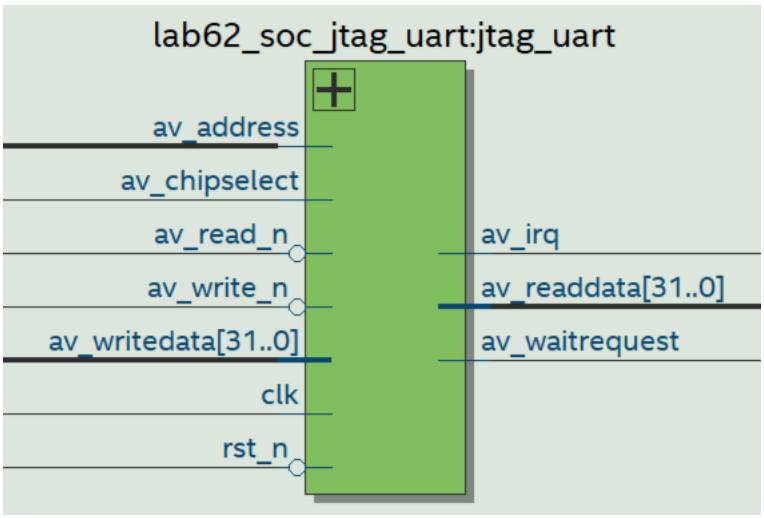
Input: clk, [15:0] reset_req_in

Output: reset_out, reset_req

Description: This component is the reset controller of the Nios II system

Purpose: It is used to reset various component in the Nios II system.

lab62_soc_jtag_uart High-Level Block Diagram



Module Description

Module: lab62_soc_jtag_uart.sv

Description: This component controls the USB interface between the Nios II system and the keyboard.

Purpose: It is used to send interrupt instruction to the Nios II CPU.

C Code of Flickering LED

```
1  // Main.c - makes LEDG0 on DE2-115 board blink if NIOS II is set up correctly
4
5  int main()
6  {
7      int i = 0;
8      volatile unsigned int *LED_PIO = (unsigned int*)0x40; //make a pointer to access the PIO block
9
10     *LED_PIO = 0; //clear all LEDs
11     while ( (1+1) != 3) //infinite loop
12     {
13         for (i = 0; i < 100000; i++); //software delay
14         *LED_PIO |= 0x1; //set LSB
15         for (i = 0; i < 100000; i++); //software delay
16         *LED_PIO &= ~0x1; //clear LSB
17     }
18     return 1; //never gets here
19 }
```

Line 7 declares an integer variable i and sets it to ‘0’. It is used as the counting index for the 2 for loops inside the while loop.

Line 8 ‘Volatile’ keyword tells the compiler to read the data from the memory every time this variant is used. Otherwise, the compiler will only read once and the optimize the running time.

Line 10 assigns value ‘0’ to *LED_PIO which clears all the LEDs.

Line 11 – Line 17 creates an infinite while loop because 1+1 never equals to 3.

Line 13 – Line 14 Inside the while loop, the first for loop sets the least significant bit of *LED_PIO to ‘1’ by doing *LED_PIO OR with 0x1 which the result will always be 0x1. Therefore, LED0 will be turned on.

Line 15 – Line 16 The second for loop sets the least significant bit of *LED_PIO to ‘0’ by performing *LED_PIO AND with NOT 0x1 (0x0) which the result will always be 0x0. Therefore, the LED0 will be turned off.

Line 18 Since there is an infinite while loop above, the C program will never get to Line 18.

C Code for 4 functions in MAX3421E.c

```
37 void MAXreg_wr(BYTE reg, BYTE val) {
38     //psuedocode:
39     //select MAX3421E (may not be necessary if you are using SPI peripheral)
40     //write reg + 2 via SPI
41     //write val via SPI
42     //read return code from SPI peripheral (see Intel documentation)
43     //if return code < 0 print an error
44     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
45
46     BYTE temp_array[2] = {reg + 2, val};
47
48     int return_code = alt_avalon_spi_command(SPI_BASE, 0, 2, temp_array, 0, NULL, 0);
49
50     if(return_code < 0)
51         alt_printf("Error\n");
52 }
```

This function writes register to MAX3421E via SPI.

Write reg + 2 and val via SPI ----- BYTE temp_array[2] = {reg+2,val}

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8 * write_data, alt_u32 read_length, alt_u8 * read_data, alt_u32 flags);

Read return code from SPI peripheral

1. Find the name of base register in system.h as the alt_u32 base
2. alt_u32 slave = 0
3. alt_u32 write length is 2
4. const alt_u8 * write_data is temp_array
5. alt_u32 read_length = 0 since it’s writing
6. alt_u8 * read_data = NULL since it’s writing
7. alt_u32 flags always = 0

If return code < 0, print an Error ----- if(return_code < 0) alt_printf("Error\n");

```
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 temp_array[nbytes +1];

    temp_array[0] = (reg+2);

    for(int i = 1; i < nbytes + 1; i++){
        temp_array[i] = data[i-1];
    }

    int return_code = alt_avalon_spi_command(SPI_BASE, 0, nbytes+1, temp_array, 0, NULL, 0);

    if(return_code < 0)
        alt_printf("Error\n");

    return (data+nbytes);
}
```

This function writes multiple-byte and returns a pointer to a memory position after last written.

Write reg + 2 via SPI

1. BYTE temp_array[nbytes +1];
2. temp_array[0] = (reg+2);

Write data[n] via SPI where n goes from n to nbytes-1

for(int i = 1; i < nbytes + 1; i++){temp_array[i] = data[i-1];}

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8 * write_data, alt_u32 read_length, alt_u8 * read_data, alt_u32 flags);

Read return code from SPI peripheral

1. Find the name of base register in system.h as the alt_u32 base
2. alt_u32 slave = 0
3. alt_u32 write length is nbytes+1
4. const alt_u8 * write_data is temp_array
5. alt_u32 read_length = 0 since it's writing
6. alt_u8 * read_data = NULL since it's writing
7. alt_u32 flags always = 0

If return code < 0, print an Error ----- if(return_code < 0) alt_printf("Error\n");

```
82 BYTE MAXreg_rd(BYTE reg) {
83     //psuedocode:
84     //select MAX3421E (may not be necessary if you are using SPI peripheral)
85     //write reg via SPI
86     //read val via SPI
87     //read return code from SPI peripheral (see Intel documentation)
88     //if return code < 0 print an error
89     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
90     //return val
91 
92     BYTE val;
93 
94     int return_code = alt_avalon_spi_command(SPI_BASE, 0, 1, &reg, 1, &val, 0);
95 
96     if(return_code < 0)
97         alt_printf("Error\n");
98 
99     return(val);
100 }
```

This function reads register from MAX3421E via SPI.

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8 * write_data, alt_u32 read_length, alt_u8 * read_data, alt_u32 flags);

Read return code from SPI peripheral

1. Find the name of base register in system.h as the alt_u32 base
2. alt_u32 slave = 0
3. alt_u32 write length is 1
4. const alt_u8 * write_data is &reg (address of the register)
5. alt_u32 read_length = 1 since it's reading
6. alt_u8 * read_data = &val since it's reading
7. alt_u32 flags always = 0

If return code < 0, print an Error ----- if(return_code < 0) alt_printf("Error\n");

```
103 BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
104     //psuedocode:
105     //select MAX3421E (may not be necessary if you are using SPI peripheral)
106     //write reg via SPI
107     //read data[n] from SPI, where n goes from 0 to nbytes-1
108     //read return code from SPI peripheral (see Intel documentation)
109     //if return code < 0 print an error
110     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
111     //return (data + nbytes);
112 
113     int return_code = alt_avalon_spi_command(SPI_BASE, 0, 1, &reg, nbytes, data, 0);
114 
115     if(return_code < 0)
116         alt_printf("Error\n");
117 
118     return (data+nbytes);
119 }
```

This function writes multiple-byte and returns a pointer to a memory position after last written.

int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8 * write_data, alt_u32 read_length, alt_u8 * read_data, alt_u32 flags);

Read return code from SPI peripheral

1. Find the name of base register in system.h as the alt_u32 base
2. alt_u32 slave = 0

3. alt_u32 write length is 1
4. const alt_u8 * write_data is &reg (address of the register)
5. alt_u32 read_length = nbytes since it's reading
6. alt_u8 * read_data = data since it's reading
7. alt_u32 flags always = 0

If return code < 0, print an Error ----- if(return_code < 0) alt_printf("Error\n");

INQ Questions

1. What are the differences between the Nios II/e and Nios II/f CPUs?
   Nios II/e is an "economy" version which requires the least amount of FPGA resources, but also has the most limited set of user-configurable features.
   Nios II/f is a "fast" version designed for superior performance. It has the widest scope of configuration options that can be used to optimize the processor for performance.

2. What advantage might on-chip memory have for program execution?
   The on-chip memory is a storage on the processor chip which decreases the amount of time (latency) for more frequently accessed data and make it more efficient for program execution because it locates physically closer to the processor.

3. Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?
   The NIOS II is a modified Harvard machine because the memory between the instruction and the data master are shared, and they can be accessed by one another. In addition, the addresses can also be modified in the modified Harvard machine.

4. 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 on-chip memory needs access to both the data and the program bus because it contains actual data that the program can read and write.
   The LED peripheral needs access to the data bus because it is an external output which only requires data like '0' or '1' to display.

5. Why does the SDRAM require constant refreshing?
   The SDRAM is consisted of a transistor and capacitor per bit of information. The information may be lost or incorrect since the capacitor will discharge over time. Therefore, the SDRAM requires constant refreshing to make sure the data is correctly saved.

6. Make sure this is consistent with you above numbers; you will need to justify how you came up with 512Mbit to your TA.

SDRAM Parameter	Short Name	Parameter Value (fill in from datasheet)
Data Width	[width]	16
# of Rwos	[nrows]	13
# of Columns	[ncols]	10
# of Chip Selects	[ncs]	1

# of Banks	[nbanks]	4

The memory of the chip is calculated with the following equation

$$Data\ Width \times \#\ rows \times \#\ columns \times \#\ chip \times \#\ bank$$
$$16 \times 8k \times 1k \times 1 \times 4 = 512000000\ bits = 512Mbits$$

7. What is the maximum theoretical transfer rate to the SDRAM according to the timings given?
   Data Width = 16-bit
   Access Times = 5.4ns

$$\frac{16\ bits}{5.4\ ns} \times \frac{1\ Byte}{8\ bits} = 370\ MB/s$$

8. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?
   The SDRAM cannot be run too slow because there is a window of time that the SDRAM transactions are valid. Therefore, the SDRAM clock must toggle within that period of time to capture the correct values after data stored in the SDRAM is refreshed. If the clock is too slow, it will capture the incorrect data.

9. You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. 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? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.
   The reason why the second clock is needed to connect to the SDRAM chip 1ns behind the controller clock is because the controller takes time for address, data, and control signals to be correct at the SDRAM pins for the chip. Therefore, the window of time of the clock needs to be delayed, so we can make sure the values are correct within that time frame and then capture those correct values.

10. What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?
    The NIOS II starts execution from the start of the SDRAM which has the address x0800 0000. We do this step after assigning the addresses because the processor will know where to start the program. In addition, the processor will know where to return when there is a reset signal. Therefore, memory overlap will not happen.

11. What the volatile keyword does in line 8?
    'Volatile' keyword tells the compiler to read the data from the memory every time this variant is used. In addition, volatile prevents the compiler from performing any optimization to the value. Moreover, it tells the compiler that this object represents hardware which is accessed outside of the main.c program. It assigns the pointer to the appropriate address of the hardware.

12. How the set and clear functions work by working out an example on paper (line 13 and line 16)?
    SET function: set the least significant bit of the first LED to the value that is equivalent to LED_PIO = LED_PIO | 0x1. In other words, the value of LED_PIO is OR with 0x1 which the result will always be LED_PIO = 0x1.

CLEAR function: set the least significant bit of the first LED to the value that is equivalent to LED_PIO = LED_PIO & ~0x1. In other words, the value of LED_PIO is AND with NOT 0x1 (0x0) which the result will always be LED_PIO = 0x0.

13. Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code: const int my_constant[4] = {1, 2, 3, 4}

    .bss ----- It contains global variables without initialization. Specifically, it contains all zero initialized global variables and static variables or do not have explicit initialization in the source code.
    Example: int x;

    .heap ----- region where memory is allocated
    Example: int ptr = (int) malloc (sizeof(int));

    .rodata ----- region of static constants (not variables, read only data)
    Example: const int x = 0;

    .rwdata ----- region of read/write data that can be modified
    Example: int x = 1;

    .stack ----- region of stack where the activation record and function calls are stored
    Example: int function(int x){}

    .text ----- Part of an object file or the corresponding portion of the executable instructions stored in the program's virtual address. It is normally read-only and fixed size.
    Example: executable binary instruction

Bugs encountered and corrective measures taken

The Platform Designer part of this lab is pretty straightforward because the "Introduction to NIOS II and Platform Designer" and "Introduction to USB on the Nios II" provide very detailed instructions on how to create the hardware part of the lab (Nios II system). The most difficult and time-consuming part is to fill in the code for the 4 functions in order to allow the USB driver running on the Nios II to communicate with the MAX3421E via the SPI peripheral. Specifically, the parameters of alt_avalon_spi_command function is very difficult to fill in. Initially, all the names of "ult_u32_base" parameter in the 4 functions are filled with the wrong name, and then the correct name is found in system.h file under "spi configuration". Moreover, some names of alt_u8 * read_data pointer needs to fill with the keyword NULL instead of '0' if they are not used. After fixing these issues and programming the hardware on the FPGA, all the functions are working properly, and the 4 keys on the USB keyboard: W, A, S, D, are able to change the direction of the bouncing ball.

Post-Lab

	Bouncing Ball
LUT	4092
DSP	None
Memory (BRAM)	55296
Flip-Flop	2447
Frqeuency	72.88MHz
Static Power	96.53mW
Dynamic Power	64.05mW
Total Power	182.27mW

Conclusion

The functionality of this lab is to configure a basic SOC design with a CPU, memory and some peripherals: LEDs, switches, USB, and VGA to performing some tasks, such as flickering LED and bouncing ball. The "Introduction to NIOS II and Platform Designer" and "Introduction to USB on the Nios II" documents are very helpful for beginners because they have very detailed instructions on how to create the hardware portion of the lab in Platform Designer. Moreover, the INQ questions allow us to understand deeper about the Nios II system and Platform Designer. In addition, this lab allows us to learn about how the VGA monitor works through the SystemVerilog code, and how the USB keyboard works through the implementation of the 4 functions. These extensions expand the abilities of the FPGA which enable us to create more complicated and more advanced project on the FPGA.