ECE 385

Spring 2023

Experiment 6

SOC with NIOS II in SystemVerilog

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

In Lab 6, we utilized the NIOS-II processor and the platform designer to implement a system of memory and peripherals and learn about its interaction with FPGA on-board switches and buttons as well as the USB and VGA peripherals. During week 1, we implemented a simple accumulator and LED blinking program through the Eclipse software that communicated with the inputs and outputs of the FPGA. During week 2, we extended the peripherals used by implementing an SPI protocol between the NIOS-II and the external USB transceiver to control the movement of a ball on a VGA monitor through the W, A, S, and D keys on an external keyboard. To interface the external USB chip and VGA peripheral, a communication IP (SPI) is used. For this lab, we use an external MAX3421E USB transceiver, which interacts between a 'host' and a 'device.' For our implementation in this lab, the FPGA acts as the host and our external USB keyboard is the device connected to the 'host'/FPGA. The keycode PIO in the platform designer of the FPGA helps output the keycode from the USB keyboard input, which helps interface with the VGA portion of the design. The keycode goes through the ball routine to determine the x and y positions of the ball, which is then integrated with the VGA controller's DrawX and DrawY outputs to determine the R, G, B of a pixel at a given point in time, thus drawing the ball on our screen.

Pre-Lab Questions

1. Fill in the table from the INQ tutorial.

SDRAM Parameter	Short Name	Parameter Value
Data Width	[width]	16 bits
# of Rows	[nrows]	13 rows
# of Columns	[ncols]	10 columns
# of Chip Selects	[ncs]	1 chip select
# of Banks	[nbanks]	4 banks

2. Italicized questions from the INQ tutorial.

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

Nios II/e stands for the "economy" version of the Nios II CPU and uses fewer logic components and resources and is consequently slower. On the other hand, Nios II/f stands for the "fast" version of the Nios II CPU and uses more resources, has faster performance, and thus, is more efficient.

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

There are numerous advantages of executing programs using on-chip memory. First of all, accessing data or instructions from on-chip memory is often faster than using external memory. Also, on-chip memory can reduce costs and consumes less power than external memory.

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

Nios II uses a modified Harvard machine because both the instructions and data share the same memory space to store information; however, the instructions and data are on different buses. This allows Nios II to access the instructions and data simultaneously.

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?

On-chip memory needs access to both the data and program bus because it needs to read and write data and run instructions. However, LEDs are simply outputs and only need to read the data and display it. Therefore, it doesn't need a program bus and simply needs a data bus.

Why does SDRAM require constant refreshing?

SDRAM requires constant refreshing because it uses a capacitive storing mechanism, which it stores data through charges on capacitors. However, due to the leak of capacitors over a period of time, memory will also leak. Thus, to prevent this from happening, we have to refresh the SDRAM to keep the data accurate.

Note that there is one 32M*16 chips, so the total amount of memory should be 512MBits (64 Mbytes), make sure this is consistent with your above numbers; you will need to justify how you came up with 512 Mbit to your TA.

Since the SDRAM has a data width of 16, 13 rows, 10 columns, 4 banks, and 1 chip select. We can simply calculate the total memory by

total memory = $2^{(\#\ of\ rows)}2^{(\#\ of\ cols)}(\#\ of\ banks)(data\ width)$ = $2^{13}2^{10}4(16)\ \approx\ 512M$bits

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

As SDRAM has a data width of 16 bits, which is equivalent to 2 bytes, and an accessing time of 5.4 ns.

$$Max\ Transfer\ rate\ =\ data\ width\ (bytes)\ *\ \frac{1}{Accessing\ time}\ =\ 2\ *\frac{1}{5.4\ ns}\ \approx\ 370MB/s$$

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

Knowing that SDRAM is a synchronous type of memory, it has to work synchronously with the clock. Therefore, the clock will determine the period of time and rate an SDRAM should read and write memories. If the SDRAM runs too slowly, then it might not be able to transfer the correct data on time. In addition, the SDRAM may not be refreshed fast enough and may lose data if the clock is too slow, assuming the refresh is synchronized to the clock.

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?

The -1ns shift is used to compensate for the clock skew or in other words, the propagation delay between the controller and SDRAM. This will make sure the SDRAM clock signal arrives SDRAM chip after the controller sets up all the required signals.

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

Nios II starts execution from the base address of the SDRAM. In this case, it starts at the address x08000000. We do this step after assigning the addresses to make sure the processor knows which address to go to whenever there is an exception or reset.

You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 8), and how the set and clear functions work by working out an example on paper (lines 13 and 16).

This short program first declares an unsigned volatile int pointer for LED_PIO and assigns the address for the LED. The volatile keyword tells the compiler that the LED might change at any time even if the program doesn't modify it. Then, the program clears the content in LED and runs an infinite for loop. In the for loop, it sets the least significant bit by ORing the memory in LED and x01 and clears the least significant bit by ANDing the memory in LED and ~x01.

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}
will place 1, 2, 3, 4 into the .rodata segment.

.bss (Block Started by Symbol) - This section is used to store uninitialized global or static variables.

E.g.

```
int temp;
static int ans;
```

.heap - This section is used for dynamic memory allocation, using functions such as malloc.

E.g.

```
int *ptr = (int*) malloc(sizeof(int));
*ptr = 2;
free(ptr);
```

.rodata (Read-Only Data) - This section is used to store constant data that cannot be modified during runtime.

E.g.

```
const int temp = 0;
```

.rwdata (Read-Write Data) - This section is used to store initialized global or static variables that can be modified during runtime.

E.g.

```
int temp = 20;
```

.stack - This section is used for storing local variables, function arguments, and return addresses.

E.g.

```
int add( int x, int y){
        int sum = x+y;
        return sum;
}
```

.text - This section is used for storing executable code. This section contains the compiled instructions that are executed by the processor.

E.g.

```
int add( int x, int y){
        int sum = x+y;
        return sum;
}

int main(){
        int x = 1;
        int y = 2;
        int ans = add(x,y);
}
```

Written Description and Diagrams of NIOS-II System

I. Hardware Components (System Level Block Diagram)

The hardware aspects of our design for this lab include the NIOS-II processor, SDRAM controller, SDRAM PLL, system ID checker, on-chip memory block, as well as a number of PIO (parallel input/output) peripherals, the JTAG UART peripheral, and an SPI protocol. The NIOS-II processor is the core component that communicates with the peripherals through the Avalon Bus. The SDRAM serves as the storage component for code and data of our program, while the on-chip memory block is unused for the purpose of this lab. The SDRAM PLL helps compensate for the clock skew and ensure that the SDRAM runs properly with the SDRAM clock connected to the output of the SDRAM PLL. The system ID checker helps ensure compatibility between the software and hardware aspects of our design. For the blinking LED and accumulator programs of week 1, we utilized the key, led, and switch PIO peripherals. For the week 2 program, we add additional PIO peripherals to interface with the keyboard inputs and display the keycodes onto the FPGA through hex digits. The JTAG UART peripheral, on the other hand, enables us to use the terminal of the our computer running the program to communicate with the NIOS-II and crucially print statements that help with debugging. The SPI protocol then helps interface with the MAX3421 USB transceiver with the NIOS-II processor and FPGA board in order for data to be transferred from the external keyboard.

II. Lab 6.1

As mentioned above, in order to implement the blinking LED/accumulator program in Lab 6.1, we utilized the key, led, and switch PIO peripherals. The key peripheral consists of two buttons, with one as the reset signal and one as the accumulate signal. On the other hand, the leds help display the necessary content (whether just simply blinking or the accumulator output), and the switches help us input values to the accumulator program to add onto the value already 'accumulated' and displayed on the leds. These values and signals of these inputs/outputs are directly accessed in the Eclipse software building tool through referencing the correct base address of the peripherals.

III. MAX3421E USB Chip & VGA with NIOS-II

The MAX3421E USB transceiver chip enables communication between a microcontroller such as the NIOS-II and USB devices such as our externally connected keyboard that is used to control the movement of the ball in lab 6.2. The SPI port peripheral we implemented in the platform designer helps us integrate the MAX3421E with the NIOS-II, which enables transfer of data, and to read and write registers to the MAX3421E chip. Between the MAX3421E and the NIOS-II, the MAX chip is the slave device while the NIOS-II is the master device. The 'processed' data that the NIOS-II receives from the keyboard through the MAX3421E chip, namely the keycode that is then 'processed' through the ball.sv file to determine the x and y motion of the ball is then combined with the VGA components, namely the DrawX and DrawY signals, as inputs to the color_mapper module in order to assign the correct R, G, B values at each given pixel (given by DrawX and DrawY, or the VGA components). Combined with the horizontal and vertical sync components of the VGA, we are then able to write to a VGA monitor correctly.

IV. SPI Protocol

The Serial Port Interface (SPI) is essentially a communication protocol that interfaces microcontrollers or processors like the NIOS-II and external devices such as our keyboard for the purpose of this lab. The SPI is designed with a master-slave interface and uses four signals for communication, MISO (Master-in-Slave-out), MOSI (Master-out-Slave-in), SS (Slave Select), and CLK (Clock). The slave select signal allows us to select the slave device to communicate with, which is simply set to 0 for our purposes since we only have 1 slave device, the MAX3421E chip. In order for data to be transmitted from the NIOS-II to the MAX3421E, the data is transmitted through the MOSI signal. Conversely, for data to be transmitted to the MAX3421E from the NIOS-II, the data is transmitted through the MISO signal. The purpose of the CLK signal is to synchronize the data signals/transfers.

V. C Code Functions (Software Component)

Week 1.

Accumulator:
The code for the 6.1 lab is to simply read the values from the switches and accumulate button and write the LEDs. To do so, we started by getting the address of each component in Platform Designer. Afterward, we created pointers that point to the address of each component, which then allow us to modify the value in them by dereferencing. Then, we created a flag for determining if the accumulate button is released after pressing the button to make sure the value from the switch will only be added once even if the button is not released. Inside an infinite loop, we then detect if the accumulate button is pressed and if the flag is 0 or 1. Since the accumulate button is active low, if both the accumulate button and the flag are 0, then we add the value from the switch to the current counter and set the flag to 1 to prevent the problem of adding more than once. If the accumulate button is 1 or released, then we set the flag back to 0. Last but not least, if the counter is greater than 255, which is impossible for the LEDs to show, then we modulo the counter by 256. After all these steps, we set the LED value to the value of the counter.

Week 2

MAX3421E:
For 6.2 lab, we write 4 functions to allow Nios II to either read or write data into MAX3421E. In all these 4 functions, the alt_avalon_spi_command function is used by giving the base address of SPI and the number of writing and reading bytes. Not to mention, the slave select value will always be 0 in this lab as we only have 1 slave device. As SPI communicates with 4 ports, which are Slave Select, MOSI (Master Out Slave In), MISO (Master In Slave Out), and Clock, these 4 signals will determine which slave device to communicate with, and whether to read or write. The command byte will contain 5 bits of register address and 3 bits value to tell whether to read or write, where 000 (decimal 0) means to read and 010 (decimal 2) means to write.

MAXreg_wr (BYTE reg, BYTE val):
This function is responsible for writing data into a register. To do so, an array is created with 2 elements, each with a size of a byte. Then, the value of reg+2 is stored as the command byte in the first index to indicate that we are writing data, and the val is stored in the second index. Afterward, we use the alt_avalon_spi_command function with the SPI base address, slave select of 0, writing length of 2, and our array to write the value into the register. Last but not least, we check if the return value from the function is less than 0, which means there's an error.

MAXbytes_wr (BYTE reg, BYTE nbytes, BYTE* data):
This function is responsible for writing several bytes of data into the register. Similar to the previous function, we create an array with nbytes+1 elements as we need to write nbytes of data and have one command byte. Likewise, we put the command byte, which is reg+2, into the first index, and we run a for loop to put the rest of nbytes data into the array. Last but not least, we use the alt_avalon_spi_command function but this time with a writing length of nbytes+1, and we return data+nbytes.

MAXreg_rd (BYTE reg):
This function is responsible for reading data from a given register. To do so, we create a variable val for storing the data we read. Then, we call the alt_avalon_spi_command function with reading length of 1 and writing length of 1, where we write &reg and read the data into &val Last but not least, we check if the return value from the function is less than 0, which means there's an error.

MAXbytes_rd (BYTE reg, BYTE nbytes, BYTE* data):
This function is responsible for reading several bytes of data from the register. Similar to the previous function, we call the alt_avalon_spi_command function with a reading length of nbytes as we are reading nbytes of data and writing length of 1, where we write to &reg and read the data into &val. Last but not least, we check if the return value from the function is less than 0, which means there's an error.

VI. VGA Operation

Module: ball.sv
Inputs: [7:0] keycode, Reset, frame_clk
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: This is a positive-edge triggered module that will set the ball's position to the center and motion to 0 whenever the Reset signal is high. Otherwise, if the current ball position + the ball size exceeds the boundary in the x direction, then its motion in the x-axis will be opposite of the current motion, and motion in the y-axis will be 0. Vice versa for exceeding the boundary in the y direction. More importantly, the keycode will be used to determine the ball's motion. For example, if the code is 8'h04, which is the key A, then the motion in the x-direction will be -1 while the motion in the y-direction will be 0. Last, the ball's position will be updated using its current position and motion.

Purpose: This module is used to control the position and the motion of the ball depending on the reset and keycode signals.

Module: Color_Mapper.sv
Inputs: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY. [9:0] Ball_size
Outputs: [7:0] Red, [7:0] Green, [7:0] Blue
Description: This is a module that calculates if the current drawing position (DrawX, DrawY) is within the radius of Ball_size of the current ball position (BallX, BallY). If it is, then the Red, Green, and Blue will be set to 8'hff, 8'h55, and 8'h00 respectively. Otherwise, they will be set to 8'h00, 8'h00, and 8'h7f - DrawX[9:3].
Purpose: This module is used to determine the RGB for the VGA depending on the current position of the ball and drawing and the Ball_size.

Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This module contains a positive-edge triggered flip flop that will set hc and vc, which are the current drawing pixels to 0 whenever Reset is high. Otherwise, it will then increase hc by 1. If hc reaches the max for horizontal pixel, then it will be set to 0, and vc is incremented by 1. It also sets pixel_clk to the current clkdiv. Not to mention, the module will modify the hs and vs, which are the sync pulses for the display, depending on the current drawing pixel. Last, only when the hc is between 0-639 and vc is between 0-479 the display signal will be high.
Purpose: This module is used to determine the sync pulses and pixel_clk for VGA display. It will also run through each pixel and calculate the DrawX and DrawY and blank signals for the display.

VGA monitor contains 640 horizontal pixels x 480 vertical pixels. To display images on the monitor, there's an electric beam that will paint each pixel from left to right and top to bottom. As the VGA signal contains R, G, B, horizontal sync, and vertical sync signals, the RGB signals will determine the color that is painted while the horizontal and vertical sync signals are active low signals that tell when horizontal and vertical lines start and end.

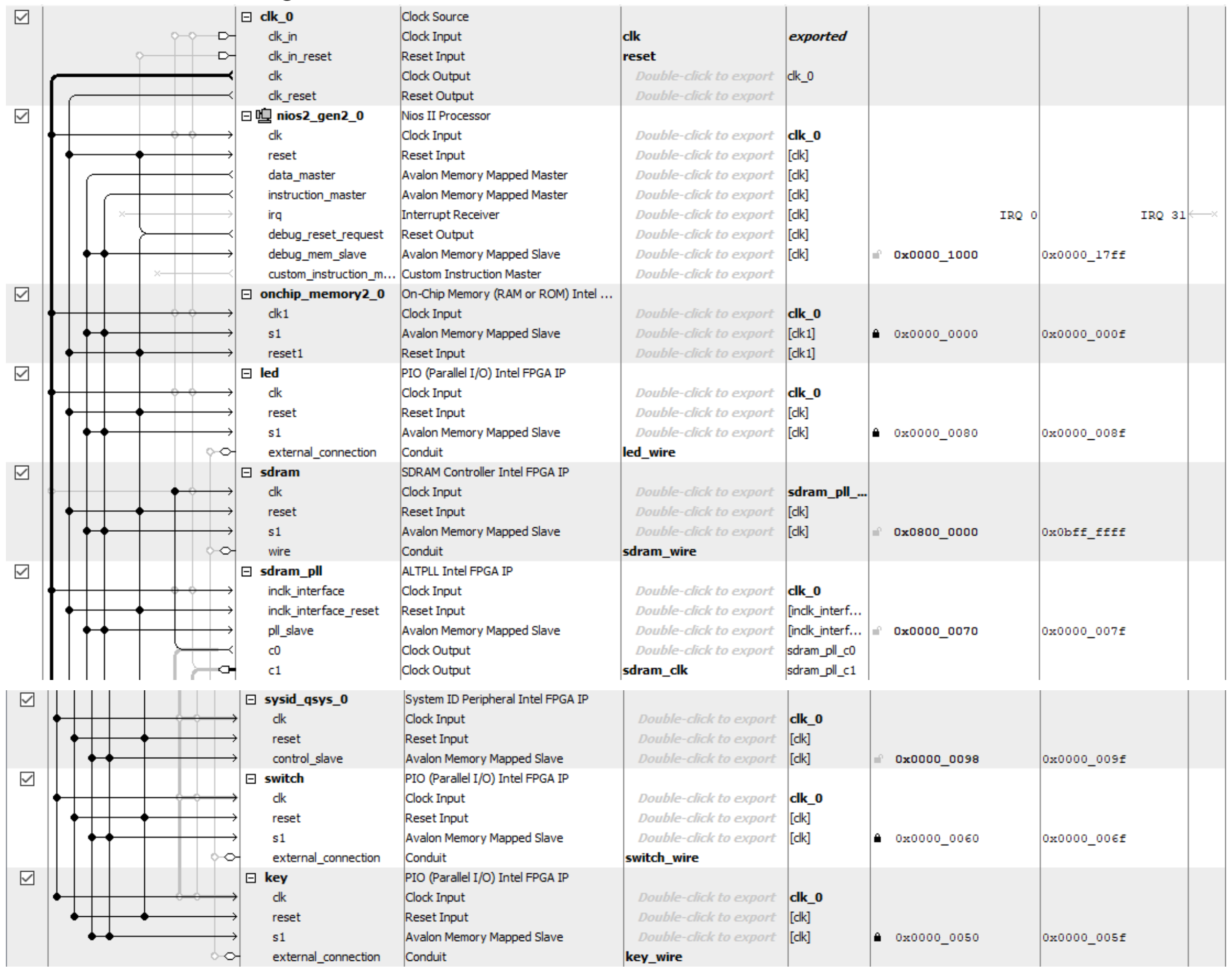
In this lab, the VGA controller module is used to iterate through each pixel to generate DrawX and DrawY. After rows and columns are finished reading, the horizontal and vertical sync signals hs and vs are then refreshed. The blank signal is also set to 1 whenever the coordinate is not within the display range. Last but not least, the pixel clock is generated with half of the frequency of the system clock. Then, the pixel clock is used as the input for the Ball module while DrawX and DrawY are the inputs for the Color mapper module.

Furthermore, the Ball module is used to calculate the x and y positions and motions of the ball. The ball is first started in the center of the screen with 0 motion. Then, the Ball module will determine the motions and positions of the ball depending on the input keycode, which is obtained from the keyboard. For example, if the key "D" is pressed, then the ball's motion in the x direction will be 1 while the motion in the y direction will be 0. If the ball touches the boundary, then the motion of the ball in that direction will be opposite while the other motion will be set to 0. More importantly, the module is then connected to the pixel clock from the VGA controller to have the same refreshing rate. Afterward, the BallX, BallY, and BallS are connected to the Color Mapper module.

The Color Mapper module takes the x and y positions and size of the ball from the Ball module along with DrawX and DrawY from the VGA controller module. It will then calculate whether the current drawing coordinate is within the radius of the ball size from the current ball position. If it is within the range, then it means the current drawing pixel is part of the ball, and hence, we have to make RGB to represent the color orange for the ball. Otherwise, we will look at the current DrawX and make RGB the corresponding background color, which is blue. Afterward, the RGB from the Color Mapper and the hs and vs signals from the VGA controller will then be used to display the image on the monitor.

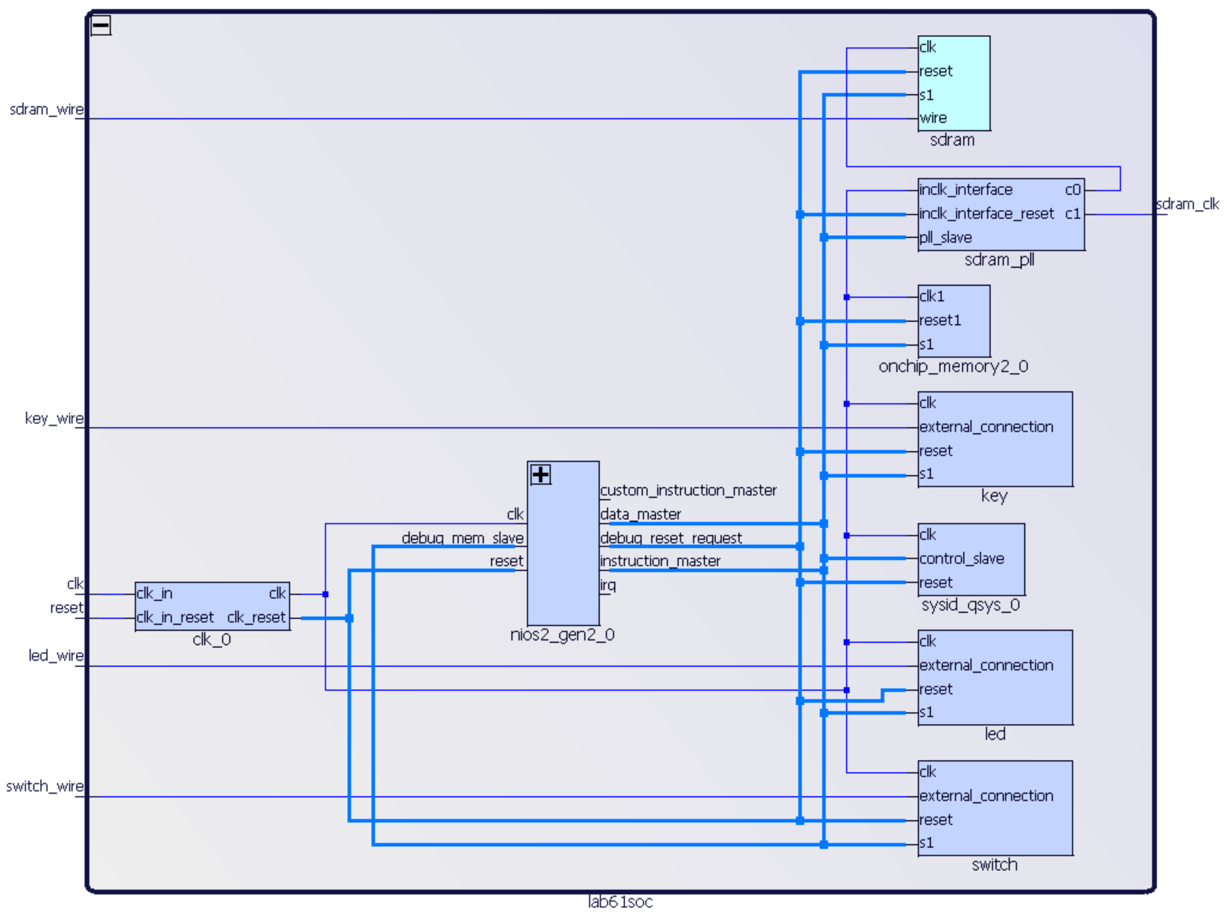
VII. System Level Block Diagram

Lab 6.1 Platform Designer

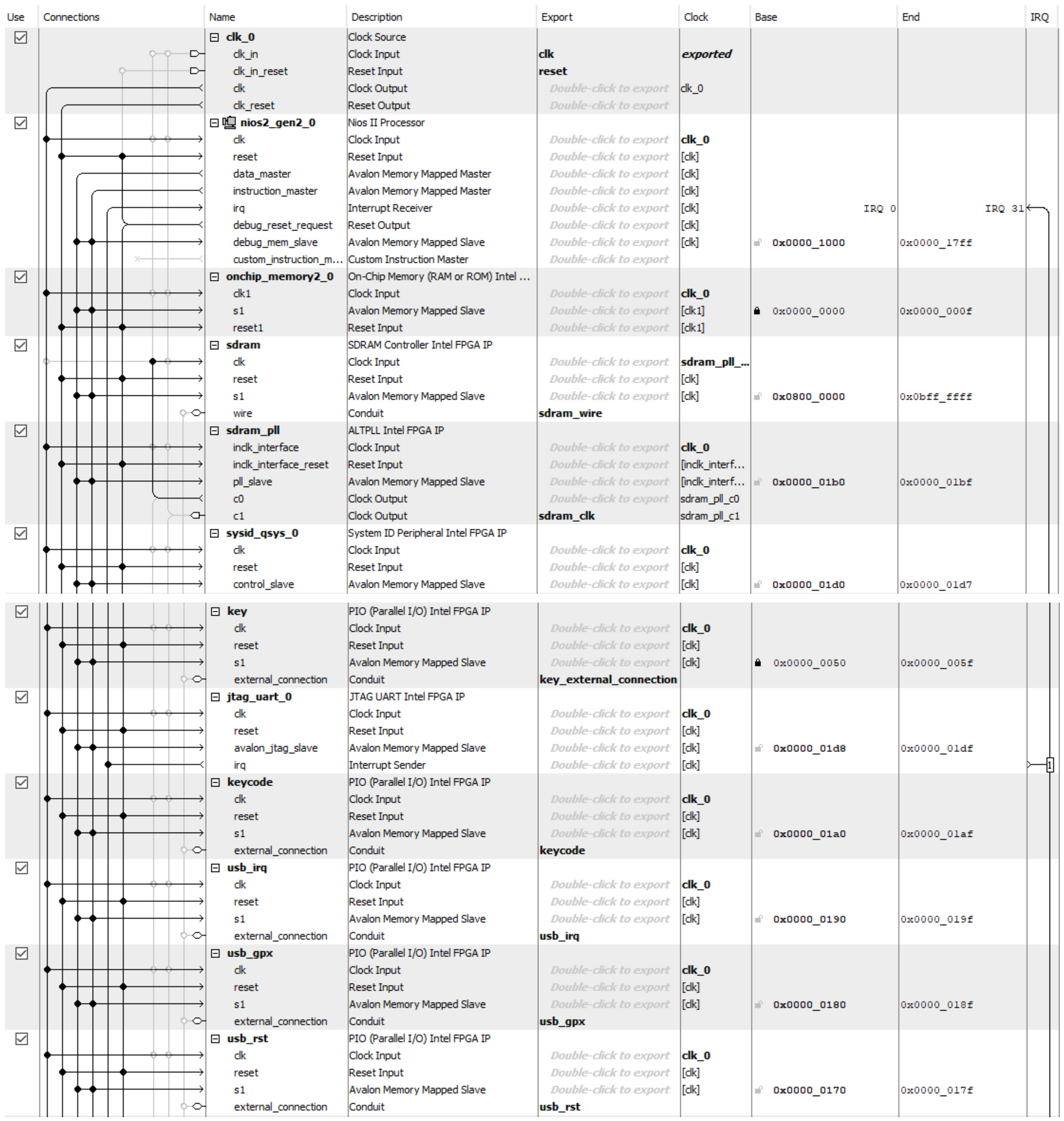


clk_0	Clock Source				
clk_in	Clock Input	**clk**	***exported***		
clk_in_reset	Reset Input	**reset**			
clk	Clock Output	*Double-click to export*	clk_0		
clk_reset	Reset Output	*Double-click to export*			
nios2_gen2_0	Nios II Processor				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset 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]		
led	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_0080	0x0000_008f
external_connection	Conduit	**led_wire**			
sdram	SDRAM Controller Intel FPGA IP				
clk	Clock Input	*Double-click to export*	**sdram_pll_...**		
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_interf...		
pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...	**0x0000_0070**	0x0000_007f
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_0098**	0x0000_009f
switch	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_0060	0x0000_006f
external_connection	Conduit	**switch_wire**			
key	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_0050	0x0000_005f
external_connection	Conduit	**key_wire**			

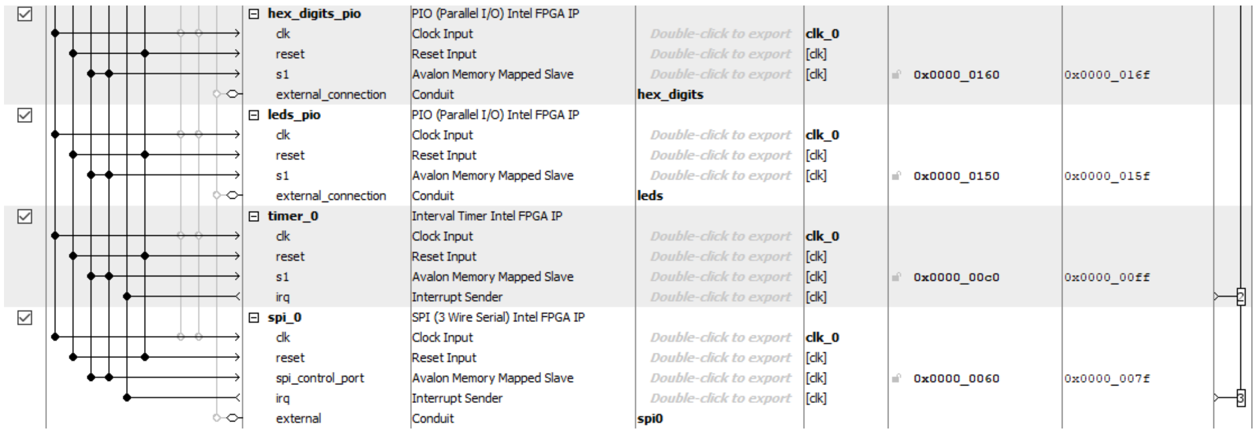
Lab 6.1 System Level Block Diagram



Lab 6.2 Platform Designer

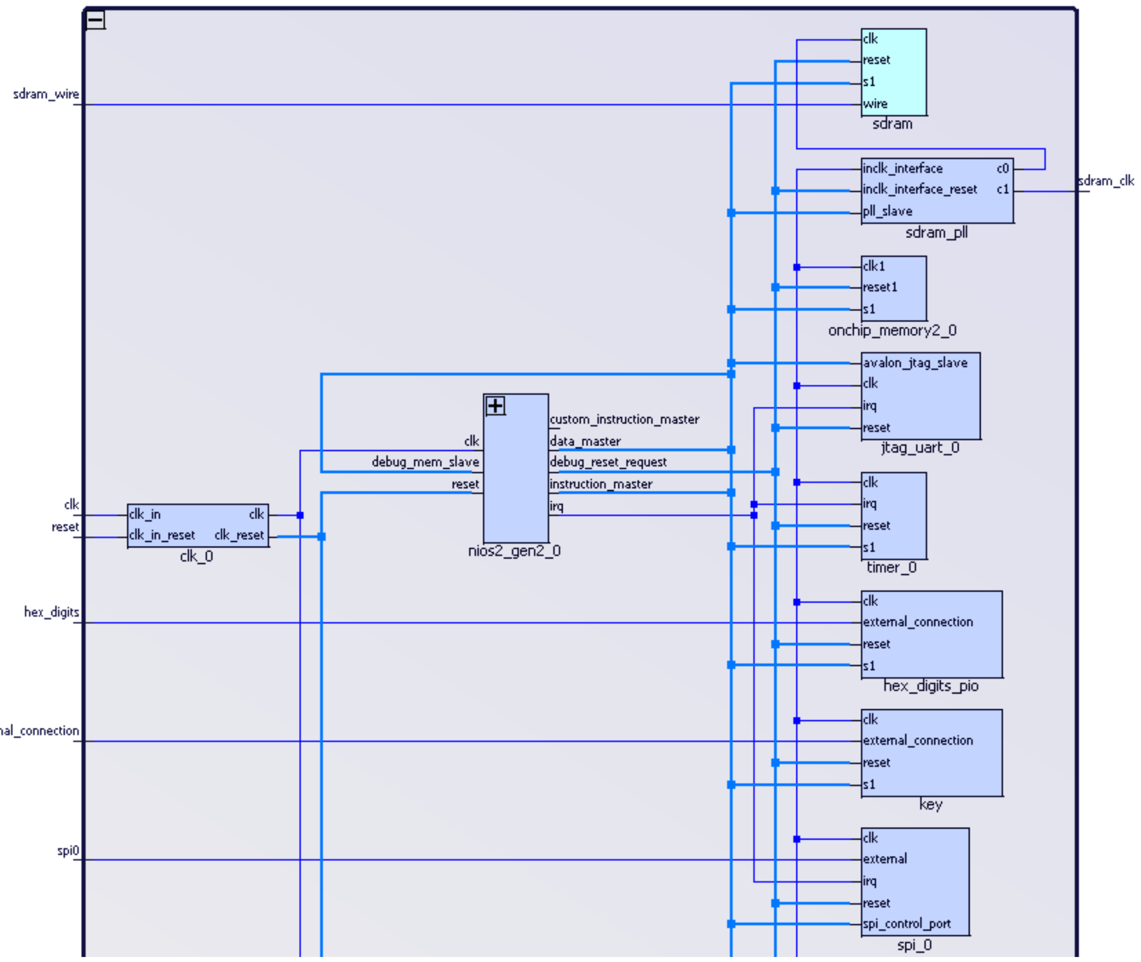


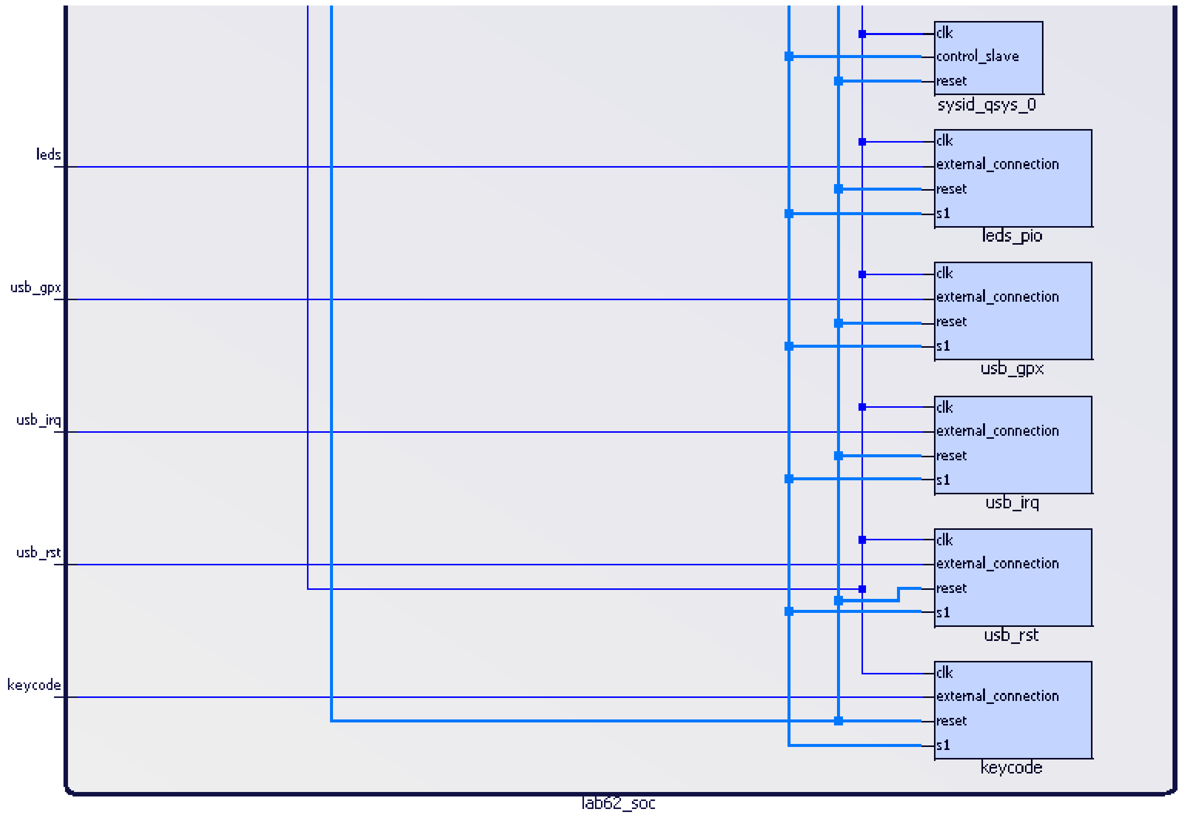
Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		⊟ **clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	*Double-click to export*	clk_0			
		clk_reset	Reset Output	*Double-click to export*				
☑		⊟ **nios2_gen2_0**	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_...**			
		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_interf...			
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...	0x0000_01b0	0x0000_01bf	
		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_01d0	0x0000_01d7	
☑		⊟ **key**	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_0050	0x0000_005f	
		external_connection	Conduit	**key_external_connection**				
☑		⊟ **jtag_uart_0**	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_01d8	0x0000_01df	
		irq	Interrupt Sender	*Double-click to export*	[clk]			1
☑		⊟ **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_01a0	0x0000_01af	
		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_0190	0x0000_019f	
		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_0180	0x0000_018f	
		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_0170	0x0000_017f	
		external_connection	Conduit	**usb_rst**				



Name	Description	Export	Clock	Base	End
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_0160	0x0000_016f
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_0150	0x0000_015f
external_connection	Conduit	leds			
timer_0	Interval Timer 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_00c0	0x0000_00ff
irq	Interrupt Sender	Double-click to export	[clk]		
spi_0	SPI (3 Wire Serial) Intel FPGA IP				
clk	Clock Input	Double-click to export	clk_0		
reset	Reset Input	Double-click to export	[clk]		
spi_control_port	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0060	0x0000_007f
irq	Interrupt Sender	Double-click to export	[clk]		
external	Conduit	spi0			

Lab 6.2 System Level Block Diagram





clk_0 - System clock with 50 MHz from the FPGA that is used to clock and reset other modules.

nios2_gen2_0 - 32-bit CPU that allows the developers to code and debug in C for the processor and send out designated control signals to modules.

sdram - the sdram is the off-chip memory block that is used to store code and data. It is interfaced with the NIOS-II processor through the memory controller that helps manage data transfers. It effectively serves as the bridge between data transferred between the processor and the SDRAM.

sdram_pll - Provides clock signal to compensate for the propagation delay that may arise between the input and output clocks, making them synchronized.

sysid_qsys_0 - This is used to ensure the hardware and software are compatible and updated to avoid any error in the connections between them.

<u>Lab 6.1:</u>

Onchip_memory2_0 - On-chip memory uses to store temporary data during program execution to reduce the time for accessing data.

leds_pio - This is used as the PIO output for LEDs to show the value in the accumulator key - This is used as a PIO input to determine the signal for accumulate button and reset button.

Lab 6.2:

jtag_uart__0 - The JTAG UART provides a communication interface between the FPGA and other devices. In this lab, it allows the user to use "printf" on the console for debugging.

keycode - Allows the processor to retrieve the specific code corresponding to the key that is pressed on the keyboard and is used in the Ball module to determine the motions of the ball.

usb_irq - This is used as the interrupt signal by the USB device to notify the computer about the key that is being pressed.

usb_gpx - The general purpose input/output peripheral is used to control the device functionality, including data transfer, etc.

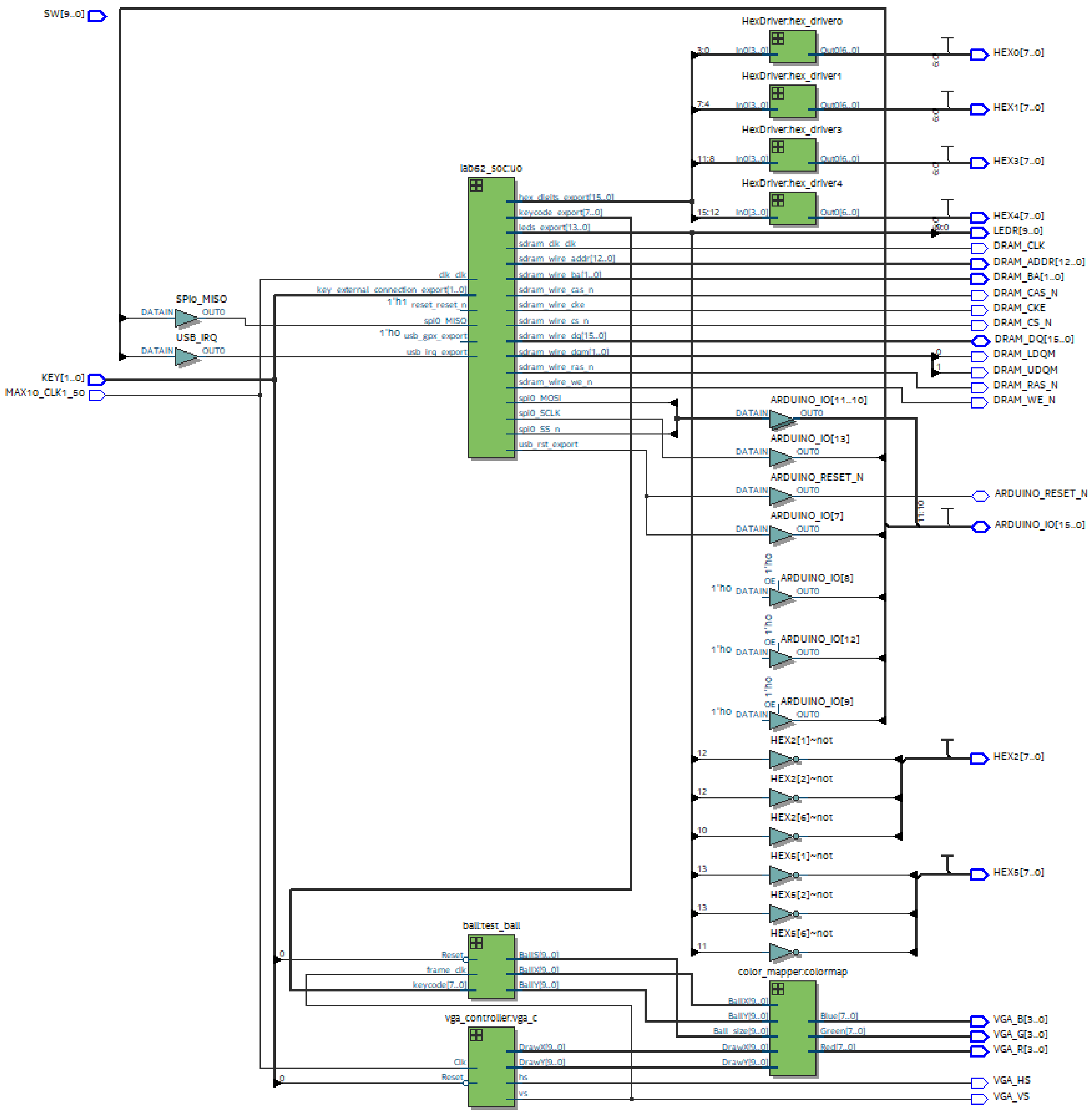
usb_rst - The hardware reset pin resets the USB controller. Active low, it will reset the chips internal registers.

hex_digits_pio - This is used to display the keycode of the currently pressed key on the two hex displays.

timer_0 - This is used to keep track of the time passed for Nios II and USB.

spi_0 - SPI_0 provides communication between a master device and one or more slave devices using four communication signals, SS (Slave Select), MOSI (Master Output Slave Input), Clock, and MISO (Master Input Slave Output). In this lab, Nios II is the master device while the MAX 3421E is the slave device. The SPI allows the master to request to read or write data from the USB device.

Top Level Block Diagram



SV Modules Summary

Module: ball.sv
Inputs: [7:0] keycode, Reset, frame_clk
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS
Description: This is a positive-edge triggered module that will set the ball's position to the center and motion to 0 whenever the Reset signal is high. Otherwise, if the current ball position + the ball size exceeds the boundary in the x direction, then its motion in the x-axis will be opposite of the current motion, and motion in the y-axis will be 0. Vice versa for exceeding the boundary in the y direction. More importantly, the keycode will be used to determine the ball's motion. For example, if the code is 8'h04, which is the key A, then the motion in the x-direction will be -1 while the motion in the y-direction will be 0. Last, the ball's position will be updated using its current position and motion.
Purpose: This module is used to control the position and the motion of the ball depending on the reset and keycode signals.

Module: Color_Mapper.sv
Inputs: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY. [9:0] Ball_size
Outputs: [7:0] Red, [7:0] Green, [7:0] Blue
Description: This is a module that calculates if the current drawing position (DrawX, DrawY) is within the radius of Ball_size of the current ball position (BallX, BallY). If it is, then the Red, Green, and Blue will be set to 8'hff, 8'h55, and 8'h00 respectively. Otherwise, they will be set to 8'h00, 8'h00, and 8'h7f - DrawX[9:3].
Purpose: This module is used to determine the RGB for the VGA depending on the current position of the ball and drawing and the Ball_size.

Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This module contains a positive-edge triggered flip flop that will set hc and vc, which are the current drawing pixels to 0 whenever Reset is high. Otherwise, it will then increase hc by 1. If hc reaches the max for horizontal pixel, then it will be set to 0, and vc is incremented by 1. It also sets pixel_clk to the current clkdiv. Not to mention, the module will modify the hs and vs, which are the sync pulses for the display, depending on the current drawing pixel. Last, only when the hc is between 0-639 and vc is between 0-479 the display signal will be high.
Purpose: This module is used to determine the sync pulses and pixel_clk for VGA display. It will also run through each pixel and calculate the DrawX and DrawY and blank signals for the display.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the Hex driver that takes in 4-bit value In0 from the register to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.

Purpose: This module is used to show the value in the adder in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: lab61.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [7:0] SW
Outputs: [7:0] LEDR, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Inouts: [15:0] DRAM_DQ
Description: This module instantiates module lab61soc, which was generated by Platform Designer, and all the connections on FPGA, such as the clock, I/O, and SDRAM signals.
Purpose: This module is used as a top-level to connect the Nios II system to the rest of the hardware.

Module: lab62.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW
Outputs: [9:0] LEDR, [7:0] Bval, Xval, [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, 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
Inouts: [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: This module is used to instantiate all the connections on the FPGA and other modules, such as vga_controller, color_mapper, and ball.
Purpose: This module is used as a top-level to instantiate the modules for displaying the ball moving on the VGA monitor and to connect NIOS II to the hardware components.

Post-Lab Questions

1. Fill in the table shown in IQT.30-32 with your design's resource and statistics.

LUTs	3413
DSP	10
Memory (BRAM)	55,296 / 1,677,312
Flip-Flop	2528
Max Frequency	90.65 MHz
Static Power	96.51 mW
Dynamic Power	60.05 mW
Total Power	177.63 mW

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?

No, the new value of Ball_Y_Motion will not be used. This is because both operations are clocked (within the always_ff block) and use the non-blocking assignment to update values, meaning that the Ball_Y_Motion is being updated at the same time we update the Ball_Y_pos with the original/old value of the Ball_Y_Motion, instead of the Ball_Y_Motion being updated first, then updating the Ball_Y_pos with the new Ball_Y_Motion value. A potential impact of this characteristic on the ball during a bounce would be a delay in the bouncing of the ball. In other words, the ball may continue to go in the same direction towards the edge of the screen/off the screen before bouncing off the 'edge' (may be slightly past if this phenomenon occurs). Similarly, this characteristic may impact the ball's interaction with a keypress by a delay in the response of the ball. One possible solution is to utilize blocking assignments instead of non-blocking assignments, which would mean that the updated Ball_Y_Motion would be used to determine the Ball_Y_pos. Another potential solution would be to have temporary variables storing the calculated Ball_Y_pos based on the updated Ball_Y_Motion value, then setting the main variables equal to the temporary variable. However, this should also depend on the usage of blocking assignments.

Conclusions

Through this lab, we learned about how to integrate Human Interface Device (HID) with our FPGA board and how the image is displayed on the VGA monitor. We learned about the interface between the NIOS-II device and the FPGA as well as the necessary protocols (SPI) to interact with external devices such as our keyboard. This can be further extended in our final project to include a variety of external input devices.

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

During the lab, our ball has the issue of traversing diagonally when it bounces off the right edge or moves along the edge. To fix the problem, we added an additional line of code when the ball reaches the boundaries to make the motion that is parallel to the boundary to 0. For instance, if the ball is currently moving toward +x direction and hit the boundary, then instead of only setting the x-motion to the opposite direction, we also set the motion in the y direction to 0 to avoid the situation where the ball moves in diagonal directions.

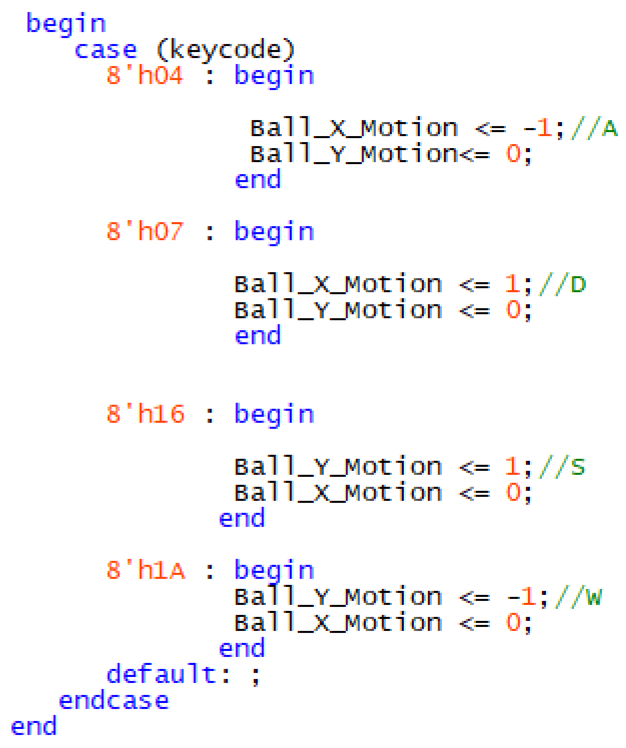
b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

No, we did not find anything particularly ambiguous or incorrect in the lab manual or given material.

Extra Credit

I. For extra credit, the provided ball.sv has a bug where it is possible to make the ball glitch off the top of the screen. Identify, document, and fix the bug in the code (and document your fix) in time for your lab report.

The bug described above is caused by the lack of begin and end statements surrounding the keycode case statements. In order to remove the bug, we simply added begin and end statements surrounding the case statements as demonstrated below. Making these changes groups the case statements together apart from the preceding code that checks if the ball is at the boundary/edge and requires bouncing. Thus, it prevents the ball motion from being overwritten if we are indeed at the edge of the screen.



```
 begin
    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
end
```