ECE 385

Fall 2019

Experiment #8

SOC with USB and VGA Interface in SystemVerilog

Xinglong Sun Churan He

ABD T 8AM

Mihir Iyer

Introduction

In this lab, we connect the monitor to the VGA port and the keyboard to the USB port of our FPGA board and make a small red ball moving based on our keystrokes. Moreover, when the small red ball hits the boundary of the screen, it will bounce back in the opposite direction. We handle all the “ball moving” logic in hardware and all the I/O in software. Since a low speed transmission would suffice for the USB keyboard, we handle the USB protocol in the software on NIOS II. Particularly, we extract the keycode from the USB keyboard through two “stub methods” and send it to the hardware for further processing. However, since we couldn’t drive our USB data inout bus directly, it’s necessary to create some sort of interface that parses the outputs (address, data, etc.) that come out from our software into the actual pins on the CY7C67200 USB chip. We will have a specific module that handles this task. For interfacing with the VGA monitor, we have a specific VGA controller module that keeps track of the sweeping “electron beam” position. Therefore, we know the exact coordinates of the beam at each pixel timetick. We can then use the monitor to customly display based on the coordinates.

Written Description of the System

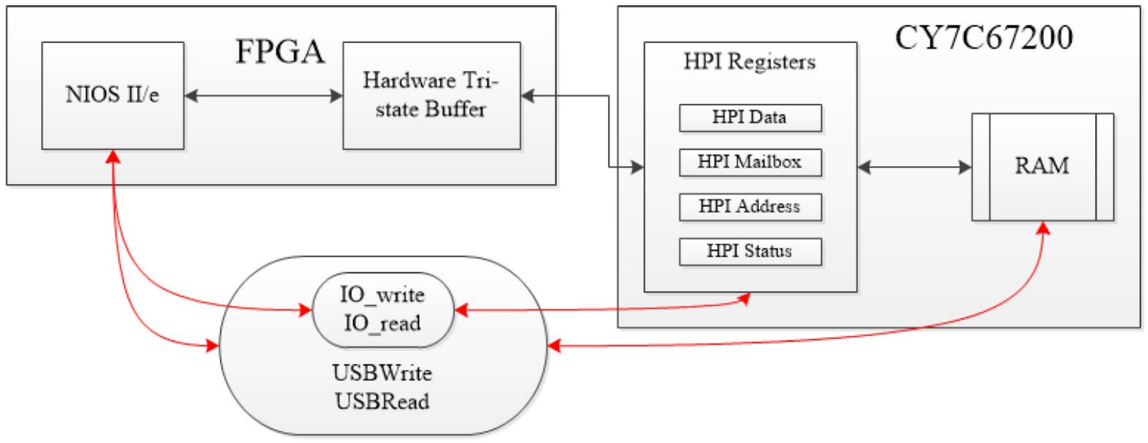
a. Written Description of the entire Lab 8 system

In Lab 8, we build a system based on NIOS II CPU which can take the input from the USB keyboard and control the motion of the ball displayed on the VGA Monitor. The system hardware contains an hpi_to_intf for the interface between NIOS II and the EZ_OTG chip, the lab8_soc for the CPU we need, vga_clk for the 25MHz VGA clock signal we need, VGA_controller for display configuration and setup, ball, color_mapper for the background, and HexDriver to show the keyboard input. The software contains the driver for the EZ_OTG chip (USB read and write functionality), and the motion control for the ball (direction control based on keyboard input, and bounce on the edge).

b. Describe in words how the NIOS interacts with both the CY7C67200 USB chip and the VGA components

One thing to notice is that we handle all the VGA related operations in SystemVerilog and only use NIOS II to interact with the USB chip. We first create several PIO blocks in QsYs including otg_hpi_data and otg_hpi_address related pins to output the corresponding values from software. These values will connect with the actual pins on the USB chip after going through the hpi_to_intf interface module. From the CY7C67200 datasheet, we see that to read a value from

USB, we first need to write the address to access to HPI_ADDR. Then, we read from HPI_DATA to get the data we want. For USB write, we first write the address to be loaded to HPI_ADDR. Then, we write the data to be stored to HPI_DATA. Since HPI_ADDR and HPI_DATA are all internal registers of the USB chips and we need to manipulate them to achieve our desired write and read, we need a way to access them. However, the EZ-OTG's memory space is not memory mapped to the NIOSs II address space. We write two helper functions UsbRead and UsbWrite to make our software talk to the EZ-OTG. Particularly, the two functions achieve the sequence of operations described above. To further simplify things, we create another two helper functions IORead and IOWrite to read from and write to a specific HPI register(HPI_ADDR, HPI_DATA, etc.). In this way, we manage to make NIOS II interact with the USB chip.



Flowchart showing the connections between our USB chip and NIOS II

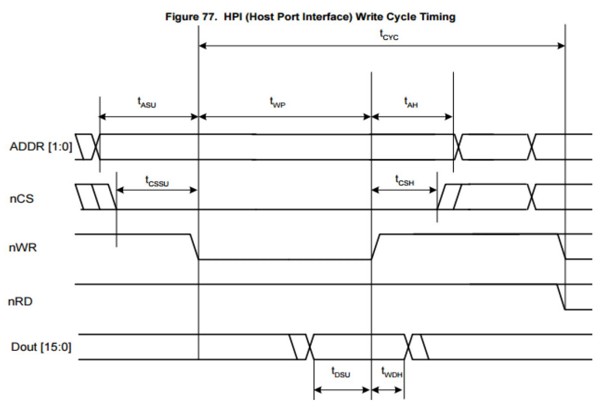
c. Written description of the CY7 to host protocol (HPI)

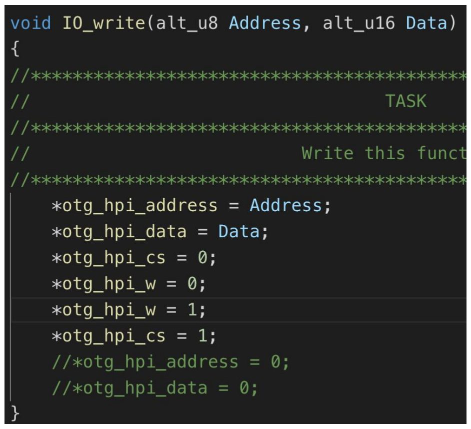
Since the design that using more pins for CY7's RAM access would increase the total cost of the chip, an HPI module are built in the CY7 chip. HPI has 4 registers, HPI Data, HPI Mailbox, HPI Address and HPI Status. For us to write and read data from USB device, we mainly need to care about HPI Data dnd HPI Address, along with other necessary signals Read, Write, Chip Select and Reset. When we want to write data to a specific place of the CY7 RAM, we need to put valid Data and Address to the corresponding HPI registers, then set Chip Select and Write to 0 (active low). To read data from the CY7 RAM, we need to firstly put the valid address to the HPI Address register, then set Chip Select and Read to 0, then we are able to read the data of the CY7's specified memory location from the HPI Data register.

d. Describe the purpose of the UsbRead, UsbWrite, IO_read and IO_write functions in the C code

IO_read/IO_write

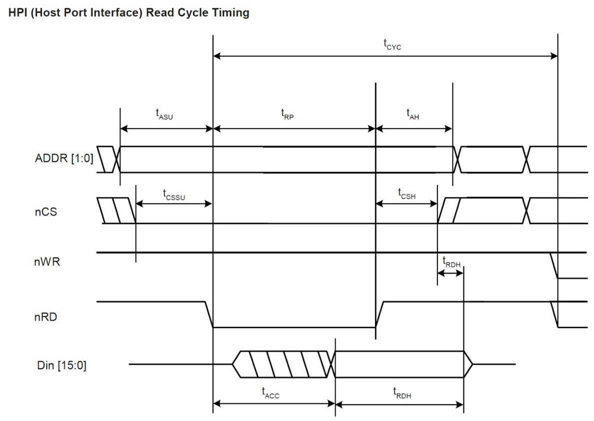
IO_read/IO_write read from and write to a specific HPI register including HPI_DATA, HPI_MAILBOX, HPI_ADDR, and HPI_STATUS. We follow the timing diagrams provided on the USB Chip datasheet as shown below:

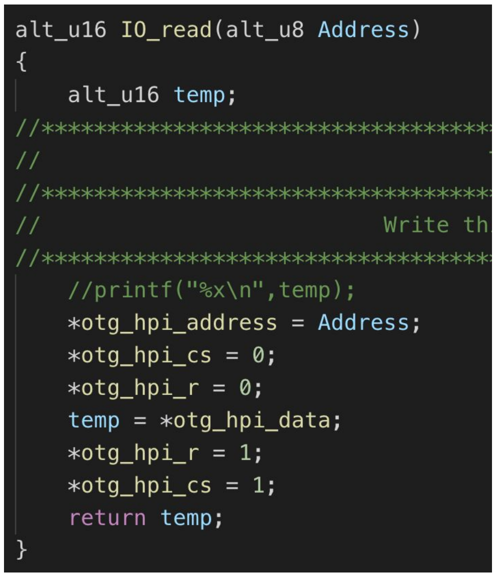




```
void IO_write(alt_u8 Address, alt_u16 Data)
{
//*****************************************
//                                  TASK
//*****************************************
//                          Write this funct
//*****************************************
    *otg_hpi_address = Address;
    *otg_hpi_data = Data;
    *otg_hpi_cs = 0;
    *otg_hpi_w = 0;
    *otg_hpi_w = 1;
    *otg_hpi_cs = 1;
    //*otg_hpi_address = 0;
    //*otg_hpi_data = 0;
}
```

IO_write timing diagram and specific implementations





```
alt_u16 IO_read(alt_u8 Address)
{
    alt_u16 temp;
//*******************************
//
//*******************************
//                        Write th
//*******************************
    //printf("%x\n",temp);
    *otg_hpi_address = Address;
    *otg_hpi_cs = 0;
    *otg_hpi_r = 0;
    temp = *otg_hpi_data;
    *otg_hpi_r = 1;
    *otg_hpi_cs = 1;
    return temp;
}
```

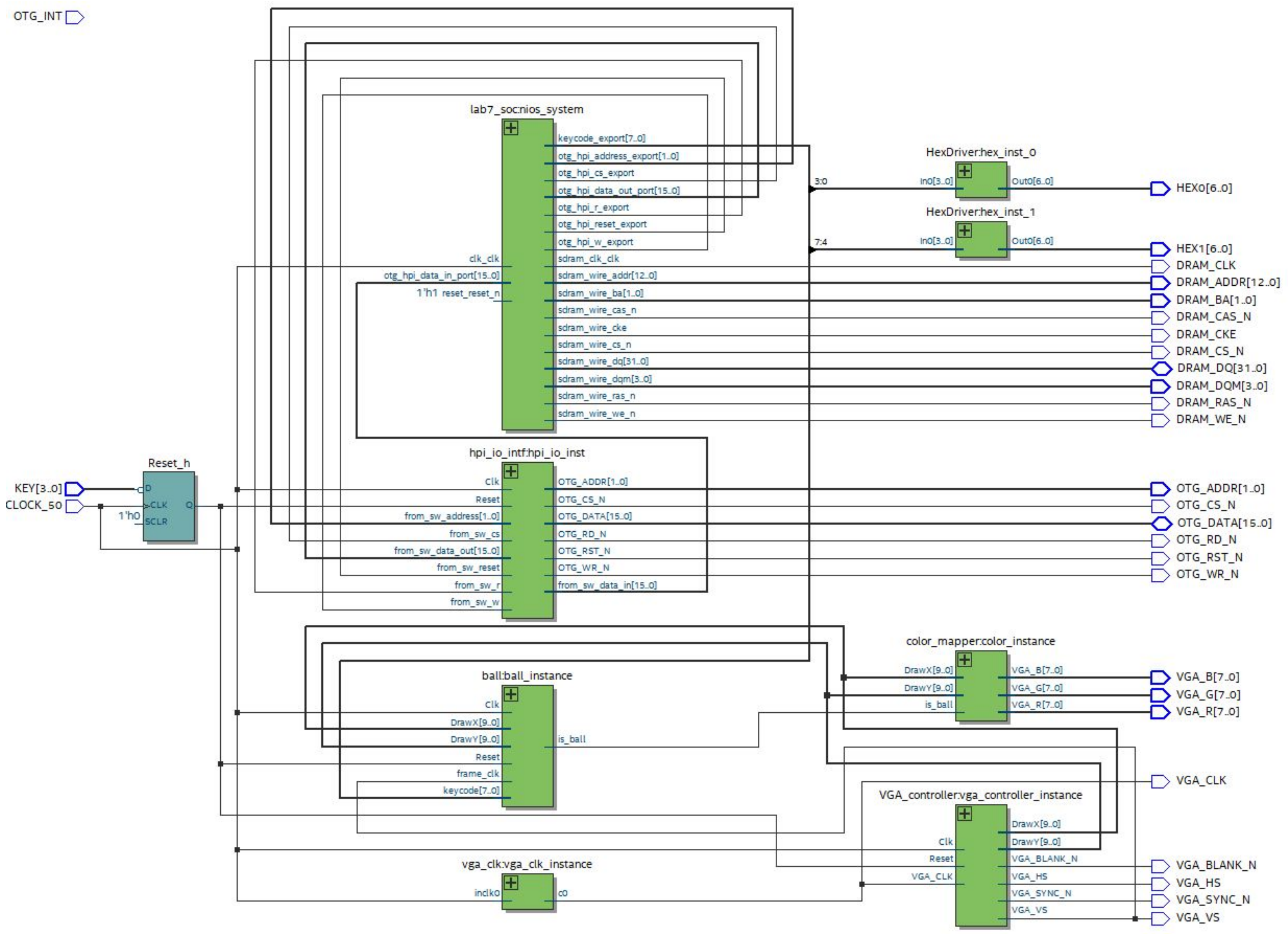
IO_read timing diagram and specific implementations

UsbRead/UsbWrite

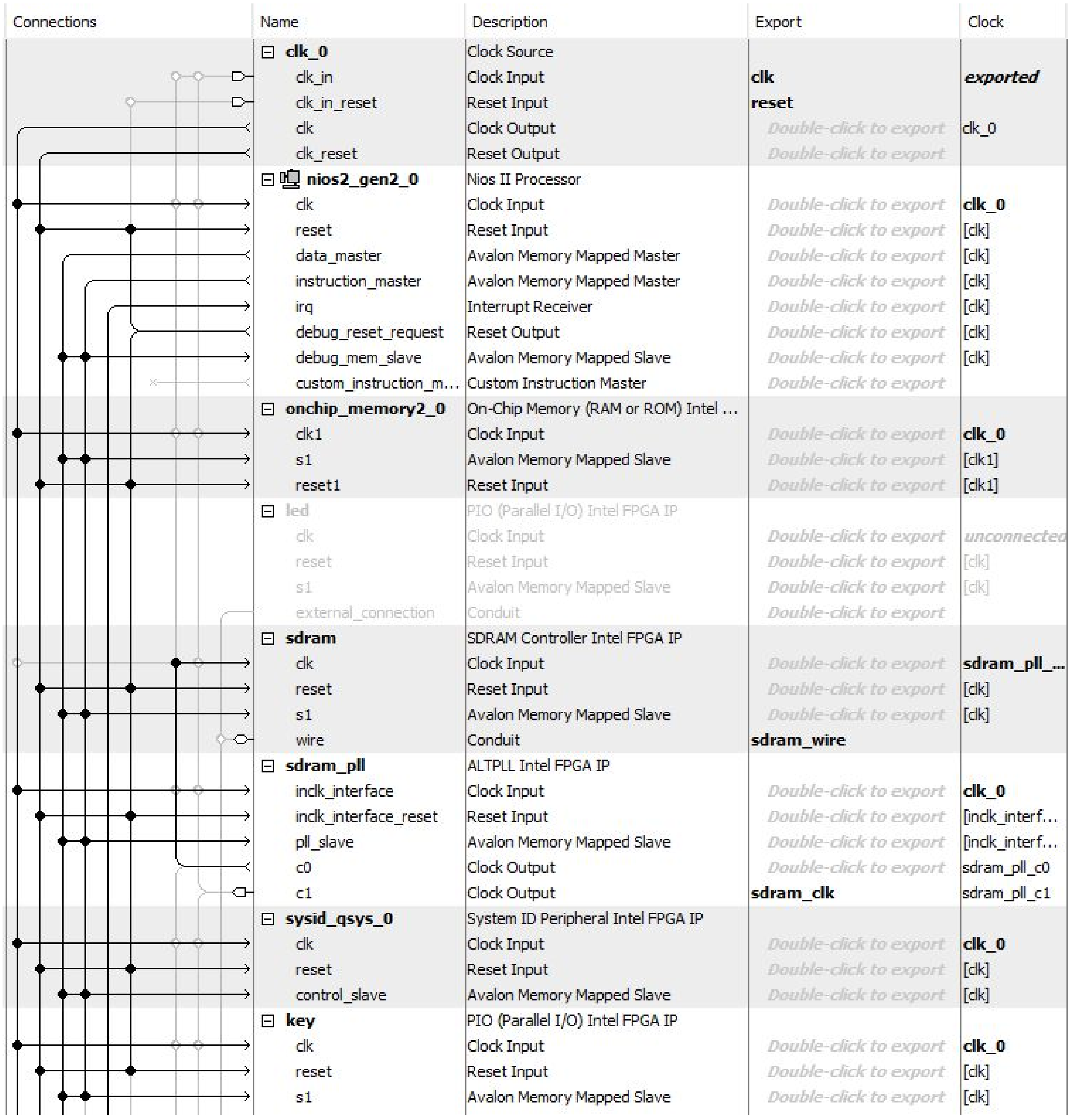
UsbRead and UsbWrite are based on the helper function IO_read and IO_write. UsbRead is using the IO_write to write the address to the HPI Address register, and using IO_read to read the data from the HPI Data register. Similarly, UsbWrite is using the IO_write to write the address to the HPI Address register, and then using IO_Write again to write the data to the HPI

Data register. UsbRead and UsbWrite are useful to read and write data from keyboard during the keyboard initialization and the actual ball motion control.

Block Diagrams

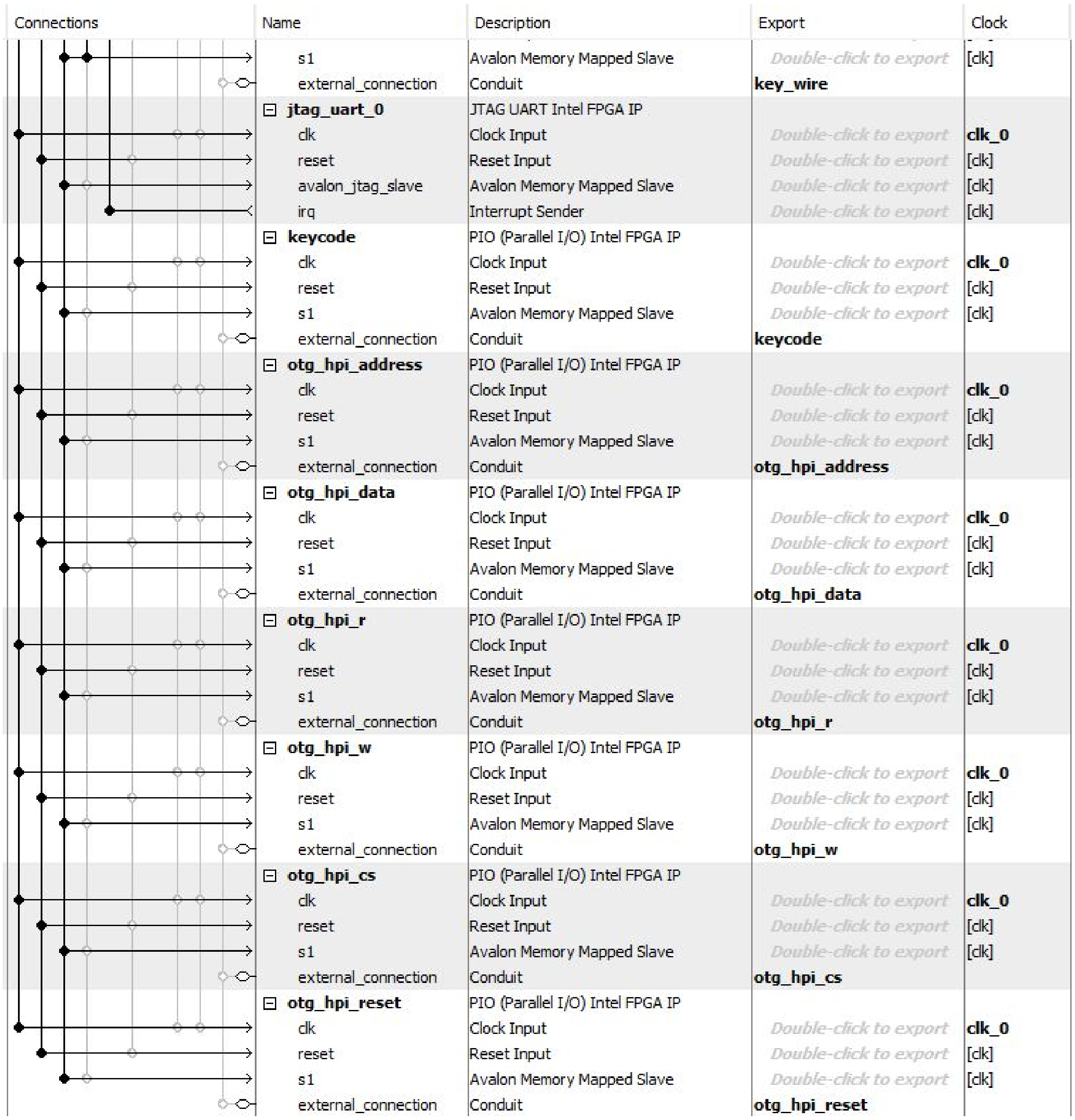


RTL View of HW System



Connections	Name	Description	Export	Clock
	⊟ **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]
	debug_reset_request	Reset Output	*Double-click to export*	[clk]
	debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	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]
	reset1	Reset Input	*Double-click to export*	[clk1]
	⊟ **led**	PIO (Parallel I/O) Intel FPGA IP		
	clk	Clock Input	*Double-click to export*	***unconnected***
	reset	Reset Input	*Double-click to export*	[clk]
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
	external_connection	Conduit	*Double-click to export*	
	⊟ **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]
	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...
	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]
	⊟ **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]

Components and Connections in SOC 1



Connections	Name	Description	Export	Clock
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]
	external_connection	Conduit	**key_wire**	
	⊟ **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]
	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]
	external_connection	Conduit	**keycode**	
	⊟ **otg_hpi_address**	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]
	external_connection	Conduit	**otg_hpi_address**	
	⊟ **otg_hpi_data**	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]
	external_connection	Conduit	**otg_hpi_data**	
	⊟ **otg_hpi_r**	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]
	external_connection	Conduit	**otg_hpi_r**	
	⊟ **otg_hpi_w**	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]
	external_connection	Conduit	**otg_hpi_w**	
	⊟ **otg_hpi_cs**	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]
	external_connection	Conduit	**otg_hpi_cs**	
	⊟ **otg_hpi_reset**	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]
	external_connection	Conduit	**otg_hpi_reset**	

Components and Connections in SOC 2

Module descriptions
SystemVerilog Modules:
lab8.sv
Inputs:
CLOCK_50,
[3:0] KEY,
OTG_INT
Inout:
[15:0] OTG_DATA,
[31:0] DRAM_DQ
Outputs:
[6:0] HEX0, HEX1,
[7:0] VGA_R, VGA_G, VGA_B,
VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS
[1:0] OTG_ADDR,
OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N,
[12:0] DRAM_ADDR,
[1:0] DRAM_BA,
[3:0] DRAM_DQM,
DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N,
DRAM_CLK
Purpose & Description:
It's the top level of our hardware system. The lab8.sv contains lab8soc, hpi_to_intf, vga_clk, vga_controller, color_mapping, ball and hexdriver. It connects these components together, and connect off-chip components including the SDRAM, EZ-OTG chip and key inputs.

hpi_io_intf.sv
Inputs:
Clk, Reset,
[1:0] from_sw_address,
[15:0] from_sw_data_out,
from_sw_r, from_sw_w, from_sw_cs, from_sw_reset
Inout:
[15:0] OTG_DATA
Outputs:
[15:0] from_sw_data_in,
[1:0] OTG_ADDR,
OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N

Purpose & Description:

This serves as an interface between NIOS II and EZ-OTG chip. The details of its descriptions and purposes are explored in the previous sections. Specifically, all the "from_sw_address" labeled inputs are the outputs from our NIOS II software. All the OTG labeled outputs go into the actual pins on the EZ-OTG chip. We have an inout pin OTG_DATA because we both need to read from and write to our USB device registers. Since inout bus should be driven by some register instead of combinational logic, a buffer register is necessary here. Concretely, when "from_sw_w" is HIGH, we assign OTG_DATA to be the same as "from_sw_data_out_buffer". Otherwise, we assign it to HighZ to properly read the value.

VGA_controller.sv

Inputs:

Clk,
Reset,
VGA_CLK,

Outputs:

VGA_HS,
VGA_VS,
VGA_BLANK_N,
VGA_SYNC_N,
[9:0] DrawX,
[9:0] DrawY

Purpose & Description:

VGA_controller is the hardware for the VGA signal protocol. It needs to provide signals for HS and VS, as well as BLANK (implemented with counters). These signals can let a VGA monitor know that our system is generating a valid VGA signal, thus it can output the display content correctly.

color_mapper.sv

Inputs:

Is_ball,
[9:0] DrawX, DrawY

Inout:

Outputs:

[7:0] VGA_R, VGA_G, VGA_B

Purpose & Description:

This module decides which color to be output to VGA for each pixel. It takes in a single bit representing whether the current pixel is ball or not and two 10 bit signals DrawX and

DrawY representing the current pixel coordinates. The logic to determine “is_ball” is accomplished in the “ball.sv” module. If is_ball is HIGH, we output xFF for all RGB values to make the ball white. Otherwise, we output a proper background color.

ball.sv

Inputs:

Clk,
Reset,
frame_clk,
[7:0] keycode,
[9:0] DrawX, DrawY,

Outputs:

is_ball

Purpose & Description:

ball.sv is the hardware drawing the ball graph for each picture of the display content. It takes the keycode to determine the ball’s motion, along with other characteristics like bounce on the edge and not moving diagonally. Then, the X, Y coordinates of a picture would be sent to the ball.sv from color_mapping.sv, if a pixel contains the ball component, is_ball would return 1 to the color_mapping, where control the actual VGA display content.

Qsys Modules:

Module: nios2_gen2_0

Exports:

Description:

This is an economy version processor we allocate for NIOS II. It handles all the logical operations and computations our program might use. One more thing to notice is that this processor also supports JTAG Debug, which means we can debug using “printf” in Eclipse.

Module: onchip_memory2_0

Exports:

Description:

This is the onchip_memory module for our system, which is always used as cache or data buffer. On-chip memory is relatively “expensive” since it required a lot more logic elements, but it can significantly improve the memory operation speed compared on external memory chips.

Module: sdram

Exports: sdram_wire

Description:

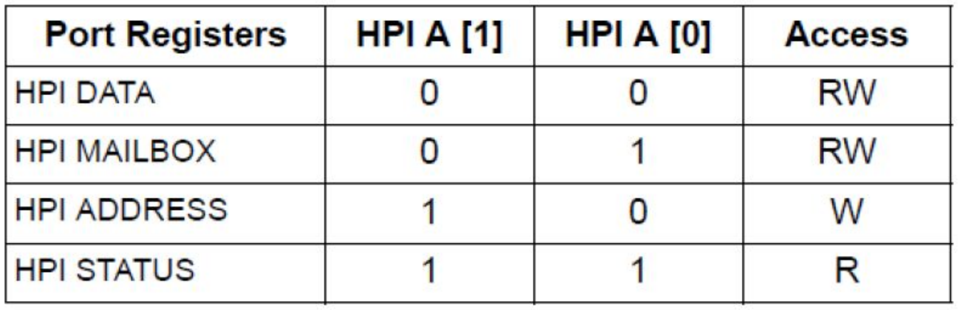
This is the SDRAM controller for NIOS II CPU. SDRAM need to be refreshed every time as we do memory operations, we don't need to write a driver by ourselves, because this existing SDRAM IP module can be used in this case.

Module: sdram_pll
Exports: sdram_clk
Description:
Pll stands for "Phase Lock Loop." As suggested by the name, it's used to generate a second clock signal shifted by an amount. As explained in detail in the following subsection(INQ Question), this block is necessary to prevent skew operation.
Module: sysid_qsys_0
Exports:
Description:
This system ID checker is to ensure the compatibility between hardware and software. This module will give us a serial number, which the software loader checks against when we start the program. This prevents us from loading software onto an FPGA which has an incompatible NIOS II configuration.

Module: key
Exports: key_wire
Description:
This is a PIO block used to handle our key inputs. By connecting the export "key_wire" to the physical keys on board, we can directly receive the user input key data in NIOS II software program.

Module: keycode
Exports: keycode
Description:
This is a PIO block to show the keycode pressed on the hex display. At most two keys pressed at the same time can be at the keycode module, then the hex display can display two key's keycode.

Module: otg_hpi_address
Exports: otg_hpi_address
Description: This is a 2-bit signals representing the address that should be accessed on the EZ-OTG chip. The signals come out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips. Details of the addresses are listed as follows:



Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Module: otg_hpi_data
Exports: otg_hpi_data
Description: This is a 16-bit signals representing the data to be written to or read from the EZ-OTG chip. Therefore, this is an inout port. The signals that come out of the software will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_r
Exports: otg_hpi_r
Description: This is a 1-bit signal representing the read signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_w
Exports: otg_hpi_w
Description: This is a 1-bit signal representing the write signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_cs
Exports: otg_hpi_cs
Description: This is a 1-bit signal representing the chip select signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Module: otg_hpi_reset
Exports: otg_hpi_reset
Description: This is a 1-bit signal representing the reset signal on the EZ-OTG chip memory. The signal comes out of the software and will be parsed to connect with the actual address pins on the EZ-OTG chips.

Answer to Post Lab Questions

a. What is the difference between VGA_CLK and Clk?

VGA_CLK is the specific clock signal for the VGA image output. VGA_CLK is 25 MHz, and it's determined by the screen refresh rate, which is 60 Hz. The screen refresh rate 60 Hz means our system must generate VGA signal 60 times a second, therefore the time for one frame is 16.67 ms. In each frame, we need a 525 times counter for vertical sync signal (more than 480, need time for front porch, VS and back porch), and a 800 times counter for horizontal sync signal (more than 640, need time for front porch, VS and back porch). Therefore, the overall frequency for refreshing one pixel is 60 * 525 * 800 = 25.2 MHz, which we approximate to 25 MHz (50 MHz / 2 using flip flop). Clk is the frequency our system running on (CPU, SDRAM), which is generated by the 50 MHz on-board crystal.

b. In the file io_handler.h, why is it that the otg_hpi_data is defined as an integer pointer while the otg_hpi_r is defined as a char pointer?

In Qsys, we define otg_hpi_data as a 16 bit data and otg_hpi_r as a 1 bit data. However, in C the smallest addressable chunk of data in C is a byte (char is a byte). Therefore, defining the otg_hpi_r as a char pointer will be enough to store a single bit signal and defining the otg_hpi_data as an integer(32 bit) pointer will be enough to store a 16-bit long signal.

Hidden Question #1/2:
What are the advantages and/or disadvantages of using a USB interface over PS/2 interface to connect to the keyboard?

1. USB interface supports hot plug, but the PS/2 interface doesn't (need to reboot the entire system). Thus we can connect a USB keyboard to the system at any time we want.
2. USB keyboard can only handle limited multi key press, but PS/2 keyboard are able to sense more key press at the same time.
3. USB operates in polling mode, but PS/2 works in interrupt mode. Thus PS/2 could provide a less response delay in the system.

Hidden Question #2/2:

Notice that Ball_Y_Pos is updated using Ball_Y_Motion.
Will the new value of Ball_Y_Motion be used when Ball_Y_Pos is updated, or the old?
What is the difference between writing
"Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion;" and
"Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion_in;"?

How will this impact behavior of the ball during a bounce, and how might that interact with a response to a keypress?

Old Ball_Y_Motion and Ball_X_Motion are used when calculating the new X-Y coordinates. Qualitatively, we can consider Pos and Motion to be the current coordinates and movements undergoing and Pos_in and Motion_in to be the next coordinates and movements. Concretely, it's essentially wrong to update Ball_Y_Pos_in by Ball_Y_Motion_in because the timestamps of Ball_Y_Pos and Ball_Y_Motion_in do not match. If we write it this way, the next Y_Pos will depend on the current Y_pos and the next movement instead of the current movement.

Consider the case when the ball hits the bottom "wall" and is about to bounce back, and we update our Pos_in in the above defined way. If we keep pressing the "s" key, the ball, instead of continuously hits the wall and bounces back, will go through the wall and visually disappear.

Design Resources and Statistics

LUT	2712
DSP	10
Memory (BRAM)	55296 bits
Flip-Flop	2248
Frequency	77.29 MHz
Static Power	151.64 mW
Dynamic Power	23.39 mW
Total Power	175.03 mW

Conclusion

Our design works well, the ball displayed on the VGA monitor can response with our key input correctly. Initially we encountered a problem with the keyboard input, the initialization step would stuck at 6 with some keyboards. However we switched to another keyboard and our system works appropriately afterwards. The potential reason of the problem may be the limited current that the FPGA board can supply to the keyboard. This problem took us a really long time, but the good things is we understood how the reliability of the system depends on the supplied power. In the future, the lab manual can remind students about this problem, and it can significantly save a great effort for inexperienced students to debug their system.