ECE385

Spring 2020

Experiment #8

SOC with USB and VGA Interface in SystemVerilog

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction

In this lab, we implemented SOC with USB and VGA interface. We made use of the NIOS-II microprocessor we made in lab7 and added several PIO modules. We then implemented the interface between NIOS-II and EZ-OTG microprocessor by making a tristate buffer like in lab6. After that, we took a look at the software driver for USB and implemented the I/O interface between the two microprocessors and how to take control of the usb from NIOS-II microprocessor by implementing usb_read/usb_write functions. Finally, we connected the vga modules and modified ball.sv to make the ball bounce on the screen. As a result, we would be able to control the ball moving on the VGA screen in the four directions by pressing the keys on the USB keyboard that is connected via the FPGA board.

Written Description of Lab 8 System

Written Description of the entire Lab 8 System

Lab8 is mainly composed of 4 parts: the on chip NIOS-II and on chip EZ-OTG microprocessors, the USB keyboard that is connected to EZ-OTG(on chip CY7C67200), the VGA that is connected to the NIOS-II. What we mainly did is just create interfaces among those four components. The NIOS-II microprocessor is implemented through the platform designer that is inherited from lab7. The EZ-OTG interface is implemented in software and defined to be on the chip CY7C67200. The driver for the USB is in provided code and the VGA is controlled via modules on the top level of lab8. For interaction between NIOS-II and chip CY7C67200, we implemented the tristate buffer in the module hpi_io_intf.sv, which controls the data flow between the NIOS-II and chip CY7C67200. Data is written/read to/from the chip CY7C67200 through this module. The software I/O functions in io_handler.c controls the signals in hpi_io_intf.sv module by using the PIO modules' port address in NIO-II just like the way in lab7. To read data from the chip, we need to set the address signal(the register we want to read from) in the function and set several control bits like otg_cs. Then the data will be sent back at otg_hpi_data through the tri-state buffer

we implemented. To write the data to the chip, we need to set both the otg_address(the register on chip to write to) and otg_data and several control bits. The data also flowed through the tristate buffer. For VGA, the driver is directly implemented in the top level lab8.sv. There are four modules that are related to it: the vga_clk, VGA_controller, ball, color_mapper. We need the vga_clk that is different from the main clock because we wanted to generate a 25Mhz clock from PLL to make the VGA component work. The VGA_controller is used to locate the X/Y coordinate we would draw concurrently. The ball will actually control the ball moving on the screen. The keycode signal is added to this module to let the ball respond to the key pressed on the keyboard. Finally, the color_mapper maps the background screen to a color and the ball to another color that is controlled by signal VGA_R, VGA_G, VGA_B.

Written Description of the USB protocol

IO_read: this function serves as the connection between NIOS-II microprocessor and chip CY7C67200. It mainly plays with PIO ports on the NIOS-II that take control of the hpi_io_intf.sv module. By setting the address, the cs, the r signals that each correspond to a PIO module on NIOS-II, the driver on the chip will send back the data corresponding to the address back to NIOS-II. The data exchange is done through hpi_io_intf.sv module that serves as the tristate buffer.
Steps in c code:
Set the *otg_hpi_address we wanted to read from;
Set *otg_hpi_cs control signal to 0(active);
Set *otg_hpi_r control signal to 0(active);
Get the data from chip by reading from *otg_hpi_data;
Set *otg_hpi_r control signal to 1(inactive);
Set *otg_hpi_cs control signal to 1(inactive);

Note the order of setting those signals is following the hpi communication in AN6010 data sheet.

IO_write: this function serves as the connection between NIOS-II microprocessor and chip CY7C67200. It mainly plays with PIO ports on the NIOS-II that take

control of the hpi_io_intf.sv module. By setting the address we want to write to, the data we want to write, the cs control bit and the w control signal on the PIO ports on NIOS-II, we are able to write data to a certain register on the chip. The data flow is done through hpi_io_intf.sv module that serves as the tristate buffer.

Steps in c code:
Set the *otg_hpi_address we wanted to write to;
Set the *oth_hpi_data we wanted to write;
Set *otg_hpi_cs control signal to 0(active);
Set *otg_hpi_w control signal to 0(active);
Set *otg_hpi_w control signal to 1(inactive);
Set *otg_hpi_cs control signal to 1(inactive);

Note the order of setting those signals is following the hpi communication in AN6010 data sheet.

USB_write:
This function writes data to the internal registers of the CY7C67200 USB controller. It first takes the Address of the register as the data and HPI_ADDR as the address and calls IO_write, i.e. writing Address to HPI_ADDR register. It then takes Data we wanted to write to register as the data and HPI_DATA as the address. Note HPI_ADDR and HPI_DATA are registers on the chip. The reason we wanted to write both data and address into registers on the chip is because we do not have direct access to the address on the chip but could only modify registers. Having the above two registers set, EZ-OTG will then store the data in the DATA register into the address in the ADDR register.

USB_read:
This function reads data from the internal registers of the CY7C67200 USB controller. It first takes the Address of the register as the data and HPI_ADDR as the address and calls IO_write, i.e. writing Address to HPI_ADDR register. EZ-OTG will then fetch the data in the address specified by the HPI_ADDR

register and load it into the HPI_DATA register. The next step is to read the data from the HPI_DATA register by calling IO_read.

Block Diagram

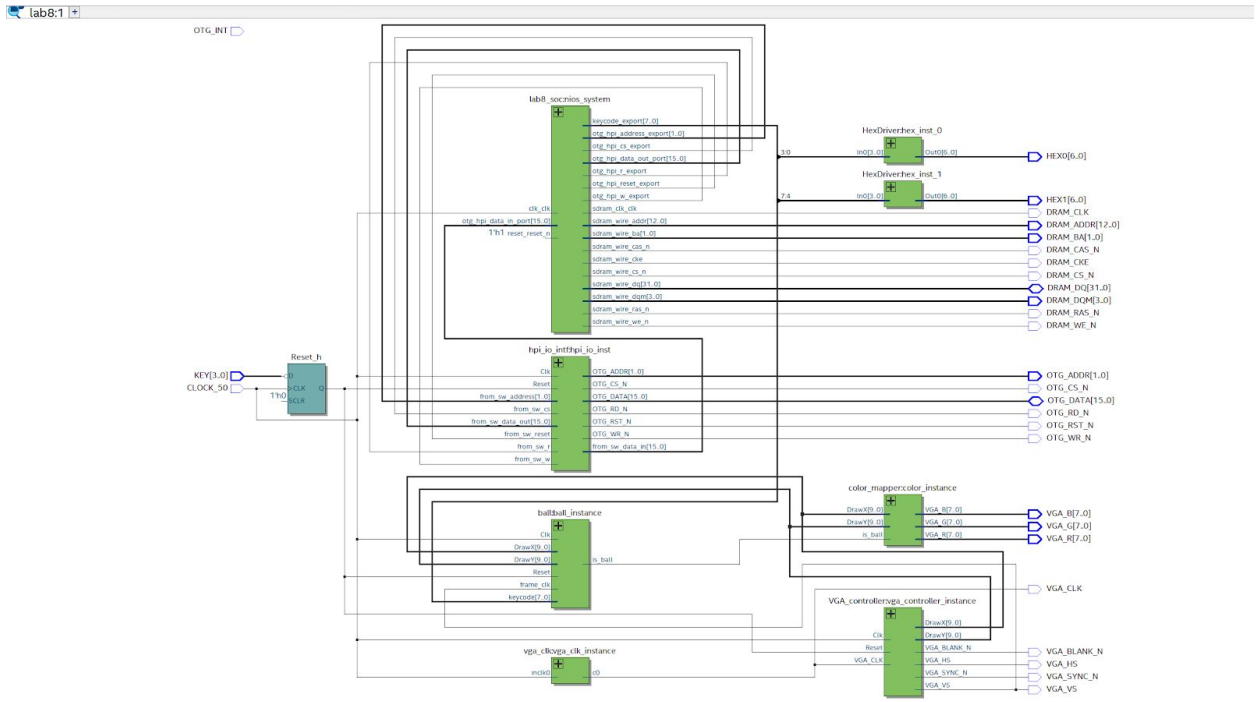

Figure 1: Lab8 top level view

Module Descriptions

Module: *lab8.sv*

Input: *CLOCK_50, [3:0] KEY, [15:0] OTG_DATA, OTG_INT, [31:0] DRAM_DQ.*

Output: *[6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [31:0] DRAM_DQ, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK.*

Description: This is the top level of this lab. It consists of all the hardware necessary for this lab including the module for NIOS-II with the DRAM, the modules for VGA, the module for the interface between NIOS-II and the EZ-OTG.

Purpose: This module is the top level for this lab. It connects all necessary modules together and serves as the top level interface for the user that makes all the inputs and outputs clear.

Module: *hpi_io_intf.sv*
Input: *Clk, Reset, [1:0] from_sw_address, [15:0] from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, rom_sw_reset, [15:0] OTG_DATA.*
Output: *[15:0] from_sw_data_in, [15:0] OTG_DATA, [15:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N.*
Description: This module takes the input from the NIOS-II that is generally the port for the PIOs and outputs signals that connect the EZ-OTG microprocessor.
Purpose: This module serves as the tristate buffer that is the I/O interface between NIOS-II microprocessor and EZ-OTG microprocessor. Hardware data exchange between those two processors is done in the way specified by this module. The software is able to control the PIO ports on the NIOS-II in order to make changes to those output signals that are connected to EZ-OTG.

Module: *ball.sv*
Input: *Clk, Reset, frame_clk, [7:0] keycode, [9:0] Draw_X, [9:0] Draw_Y.*
Output: *is_ball.*
Description: This module is the interface for the moving ball in the VGA. When VGA is rendering the screen pixel by pixel, it determines whether the current pixel defined by Draw_X and Draw_Y is a ball. It introduced the frame_clk that determines whether the frame is changed.
Purpose: This module is the control of the moving ball on the screen. It tells VGA at each frame whether to render a pixel to be the ball or not. When the ball hits the edge of the screen or when a specific keycode is pressed, the ball needs to respond in a way that makes it look like the ball is bouncing and is controlled by the key.

Module: *VGA_controller.sv*
Input: *Clk, Reset, VGA_CLK.*
Output: *VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] Draw_X, [9:0] Draw_Y.*

Description: This module takes necessary input VGA signals and outputs the x, y index of rendering concurrently. The VGA_HS is the horizontal pulse that becomes low when a row is rendered and VGA_VS is the vertical pulse that becomes low when a specific row corresponds to an index y is rendered. We can use this signal to determine whether we need to change a frame.
Purpose: This module is mainly the driver for the VGA that updates the necessary VGA signals at each VGA_CLK raising edge.

Module: *Color_Mapper.sv*
Input: is_ball, *[9:0] Draw_X, [9:0] Draw_Y.*
Output: *[7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B.*
Description: Given whether the current pixel is ball, give current pixel a RGB color corresponding to the VGA.
Purpose: The module give different RGB colors to the ball and the background.

Module: vga_clk.v:
Input: inclk0.
Output: c0.
Description: Generates a VGA clock via PLL that runs the VGA controller.
Purpose: We need to update the screen at around 60Hz, so we do not use the general clock but this VGA clock generated from PLL with 25MHz that has 800 * 525 clock edges per frame. Since 800 * 525 * 60Hz is around 25MHz.

Module: HexDriver.sv:
Input: [3:0] In0
Output: [6:0] Out0
Description: This module converts 4-bit input value into 7-bit value in LED format.
Purpose: In order to show the pressed keycode on the FPGA board, we use this module to convert the keycode value into LED form.

QSYS blocks:

Connections	Name	Description	Export	Clock	Base	End	Tags
☑	clk_0	Clock Source					
	clk_in	Clock Input	clk	*exported*			
	clk_in_reset	Reset Input	reset				
	clk	Clock Output	*Double-click to*	clk_0			
	clk_reset	Reset Output	*Double-click to*				
☑	nios2_gen2_0	Nios II Processor					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	data_master	Avalon Memory Mapped Master	*Double-click to*	[clk]			
	instruction_master	Avalon Memory Mapped Master	*Double-click to*	[clk]			
	irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31	
	debug_reset_request	Reset Output	*Double-click to*	[clk]			
	debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x1000	0x0000_17ff	
	custom_instruction_master	Custom Instruction Master	*Double-click to*				
☑	onchip_memory2_0	On-Chip Memory (RAM or ROM...					
	clk1	Clock Input	*Double-click to*	clk_0			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk1]	0x0	0x0000_000f	
	reset1	Reset Input	*Double-click to*	[clk1]			
☑	sdram	SDRAM Controller Intel FPG...					
	clk	Clock Input	*Double-click to*	sdram_pl...			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	1000_0000	0x17ff_ffff	
	wire	Conduit	sdram_wire				
☑	sdram_pll	ALTPLL Intel FPGA IP					
	inclk_interface	Clock Input	*Double-click to*	clk_0			
	inclk_interface_reset	Reset Input	*Double-click to*	[inclk_in...			
	pll_slave	Avalon Memory Mapped Slave	*Double-click to*	[inclk_in...	0x90	0x0000_009f	
	c0	Clock Output	*Double-click to*	sdram_pll_c0			
	c1	Clock Output	sdram_clk	sdram_pll_c1			
☑	sysid_qsys_0	System ID Peripheral Intel...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	control_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0xb0	0x0000_00b7	
☑	jtag_uart_0	JTAG UART Intel FPGA IP					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0xa8	0x0000_00af	
	irq	Interrupt Sender	*Double-click to*	[clk]			5
☑	keycode	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x30	0x0000_003f	
	external_connection	Conduit	keycode				
☑	otg_hpi_address	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x20	0x0000_002f	
	external_connection	Conduit	otg_hpi_address				
☑	otg_hpi_data	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x40	0x0000_004f	
	external_connection	Conduit	otg_hpi_data				
☑	otg_hpi_r	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x50	0x0000_005f	
	external_connection	Conduit	otg_hpi_r				
☑	otg_hpi_w	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x60	0x0000_006f	
	external_connection	Conduit	otg_hpi_w				
☑	otg_hpi_cs	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x70	0x0000_007f	
	external_connection	Conduit	otg_hpi_cs				
☑	otg_hpi_reset	PIO (Parallel I/O) Intel F...					
	clk	Clock Input	*Double-click to*	clk_0			
	reset	Reset Input	*Double-click to*	[clk]			
	s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	0x80	0x0000_008f	
	external_connection	Conduit	otg_hpi_reset				

Figure 2: Qsys blocks in lab8.

Functions of each block in the System-level block diagram.

clk_0:

This block generates the main clock signal for all other modules.

nois2_gen2_0:

This block represents the NIOS-II economic version. It is the microprocessor we used in this lab, which performs the computing and processing job in the hardware.

onchip_memory2_0:

It is the memory corresponding to the NIOS-II for loading and storing data that requires high-level performance.

sdram:

This block represents the sdram module. It makes a connection with the processor, so the system can use the memory specified by the sdram.

sdram_pll:

This block represents the PLL for the system. It controls a clock with a phase shift of 3ns prior to the main clock. In this way, the sdram will start processing before the clock edge, and the date would be faster processed after the main clock edge.

sysid_qsys_0:

This block represents the system id checker. The purpose of this block is to check the configuration of the hardware part and the software part to make sure the two matches.

key_code:

This block represents the parallel I/O port for switch input for FPGA. The keycode signal is connected at the top level. Software could use the address of the port of this block to read the keycode from the keyboard.

otg_hpi_address:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG address. It is connected to the hpi_io_intf module at the top level.

otg_hpi_data:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG data. It is connected to the hpi_io_intf module at the top level.

otg_hpi_r:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG read. It is connected to the hpi_io_intf module at the top level.

otg_hpi_w:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG write. It is connected to the hpi_io_intf module at the top level.

otg_hpi_reset:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG reset. It is connected to the hpi_io_intf module at the top level.

otg_hpi_cs:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG chip-select. It is connected to the hpi_io_intf module at the top level.

Answer to Hidden Questions

1. What are the advantages and/or disadvantages of using a USB interface over PS/2 interface to connect to the keyboard? List any two.

Advantages of USB:

1. USB is a bus while PS/2 is a port. So USB is able to connect many devices while PS/2 can only connect one.
2. USB with a driver like in this lab can support hot plug since it is polling while PS/2 can't since it uses interrupt.

Disadvantages of USB:

1. USB uses polling while PS/2 uses interrupt. Interrupt is more efficient than polling because it does not need to keep asking if a keystroke happened but interrupt a task and take a proficiency. So USB is slower.
2. When a USB bus is being used by other devices, the keystroke could be delayed. For PS/2 it is impossible since interrupts from a port always take the high priority in the operating system.

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?

The old *Ball_Y_Motion* was used to update the *Ball_Y_position*.

The old *Ball_Y_Motion* is used so the ball will respond to both the key-press and the boundary hitting with a one frame delay. As a result, when the ball hit the boundary, it would not stop and change immediately but get "into" the screen for one frame and start to bounce back.

If the new *Ball_Y_Motion_in* was used to make the update, the ball will respond to the key press and the boundary hitting immediately. It means if we hold the key

contrary to the boundary while the ball is at the boundary, the frame will stay almost still.

Answer to Post Lab Questions

1)

LUT	2660
DSP	0
Memory(BRAM)	55296 bits
Flip-Flop	2235
Frequency	77.8MHz
Static Power	105.28mW
Dynamic Power	27.66mW
I/O Thermal Power	75.37mW
Total Power	208.31mW

2)

- What is the difference between VGA_CLK and Clk?
  Clk is the clock for the system which has 50MHz while VGA_CLK is used to update the screen controlled by VGA that has 25MHz. VGA_CLK is used to run the VGA_controller that updates each frame at about 60Hz. The FPGA clock is the Clk.
- 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?
  An int pointer is 32 bits while a char pointer is 8 bits. Our data is actually 32-bit wide so we need to use a 32-bit pointer. The otg_hpi_r is either 0 or 1

so we can assign as little memory as possible. Since the smallest possible pointer in c is a char pointer, we use a char pointer instead.

Conclusion

In this lab, we implemented the interface among the NIOS-II, the EZ-OTG, the USB and the VGA. It seemed to be hard at first but after reading the instructions and the code, it became clear. We actually had experience with a hardware driver and implemented data exchange between different hardwares. We also learned about VGA and how it renders the screen. It is important that we learn the interface of the USB keyboard and how we can control the object on the VGA screen using the keyboard. We could further develop from this lab the way to implement simple games on the chips like those arcade games.