Lab 8 Report

ECE 385

Friday (ABB)

11:00AM-1:50PM

SOC with USB and VGA Interface in SystemVerilog

Ritvik Avancha and Steven Phan

(rra2 and sphan5)

Table of Contents

Lab 8 Report

Introduction

This week's lab is built off of the fundamental NIOS and QSYS topics displayed in last week's lab. The task this week was to expand on such topics by learning how to interface with external devices through the USB and VGA ports. Last week, we learned how C code works with the NIOS 2 processor. This week's software side of the lab focuses on data movement throughout HPI registers and data paths which will be discussed later on in the lab report. Overall, the general idea was to write code that would allow a connected keyboard to move a ball on the VGA screen by pressing W, A, S, or D.

Written Description of Lab 8 System

Summary

Given introductory files, we were asked to complete the design in order to get the monitor to display a bouncing ball (edge bounded) that could be controlled by W, A, S, and D keys. In order to do that, we had to add additional PIO blocks into the base QSYS blocks in order to handle USB connections. When that was completed, the next portion to complete was the io_handler functions and the usb functions. These functions perform single read/write operations without continuously reading/writing. Consequently, we could grab data from memory mapped registers and display it through the hex drivers on the board. These would eventually display the "keycode" of the current key being pressed. For all of this to work, we had to finish the hpi_io_intf which essentially managed the data movement and overall structure of the interfacing between the HPI and the EZ-OTG. In addition to that, the final connections made between the given vga modules and the completion of logic/boundary checking in ball.sv were made in order to get a fully functioning ball on the VGA monitor that would change its direction in response to W, A, S, and D keyboard presses.

The way the NIOS 2 system interacts with both USB and the VGA components is through the EZ-OTG as shown in Figure 2.1. The EZ-OTG handles the USB protocol and essentially acts as a USB host in establishing the keyboard connection with the NIOS 2 system.

The way it works is first the EZ-OTG chip is set up. Once a connection is detected, an address is assigned to this connected device and the type of device is identified from its descriptors. The appropriate configuration is made and data from the device is polled appropriately. Since the NIOS 2 system peripherals are mostly based on memory mapping, data from external devices can thus be directly accessed from the NIOS 2 processor via memory access functions. Similar concepts apply with the VGA components.

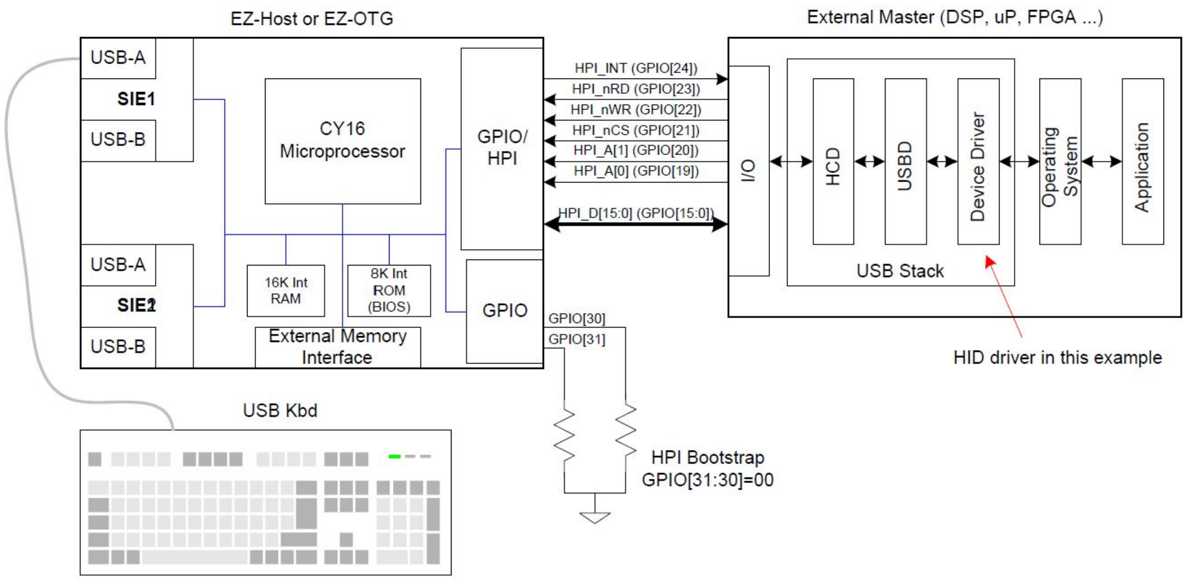

Figure 2.1: External Interfacing with EZ-OTG

The EZ-OTG (CY7C67200) has two functional modes: stand-alone mode and coprocessor mode. Additionally, the EZ-OTG comes with a built in RISC microprocessor, RAM, and ROM. The RAM offers direct memory access from address 0x0000 = 0x3FFF. The HPI allows for the connections between the EZ-OTG and the FPGA prototyping board as seen in Figure 2.1. The HPI functions as a standard 16-bit parallel slave bus interface.

In addition to setting up the EZ-OTG chip and completing the hpi_io_intf, we had to complete 4 functions in two different files. In io_handler.c, IO_write and IO_read had to be implemented. Both of these functions are a sequence of events that make up a “write” or a “read” (i.e. it handles the signals like read, write, chip select, etc.) IO_write takes an address and places it in otg_hpi_address, then sets cs and w to be 0 (the signals are active low). Then, data is placed into otg_hpi_data. This performs the write, then w and cs are both set to 1 again to signal the end of the write sequence. Conversely, IO_read follows generally the same sequence of events except r replaces w. We use a temporary variable to store the read data because we need to set both r and chip select back to 1 before we return the read value. In usb.c, this file specifically deals with the usb connection between the EZ-OTG chip and the DE2-115 FPGA board through the Host Port Interface (HPI). The HPI has 4 main registers that we are concerned with: (1) HPI_DATA, (2) HPI_MAILBOX, (3) HPI_ADDRESS, and (4) HPI_STATUS. To perform a "usb write", the address in which we want to write must be written into HPI_ADDR. This is done by using IO_write with HPI_ADDR as the destination, and the address as the "data" to be written. Then, we perform a write into HPI_DATA register with the actual data we want to write. Similarly, "usb read" the address from which we want to read must be placed into HPI_ADDR, then we return IO_read(HPI_DATA) since the data in memory would be in the HPI data register. In essence, IO_read and write directly deal with memory and data getting put into or read from a specific memory address whereas USB_read and write deal with interfacing with the USB via HPI registers (which are memory mapped).

Block Diagram

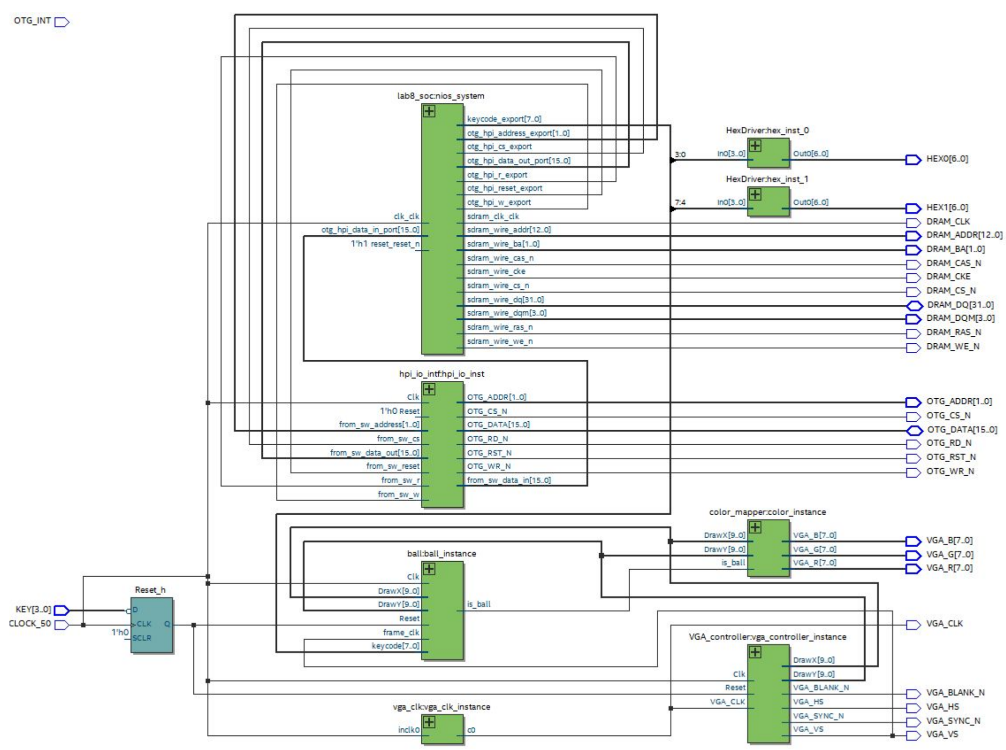

Figure 3.1: Top Level Block Diagram

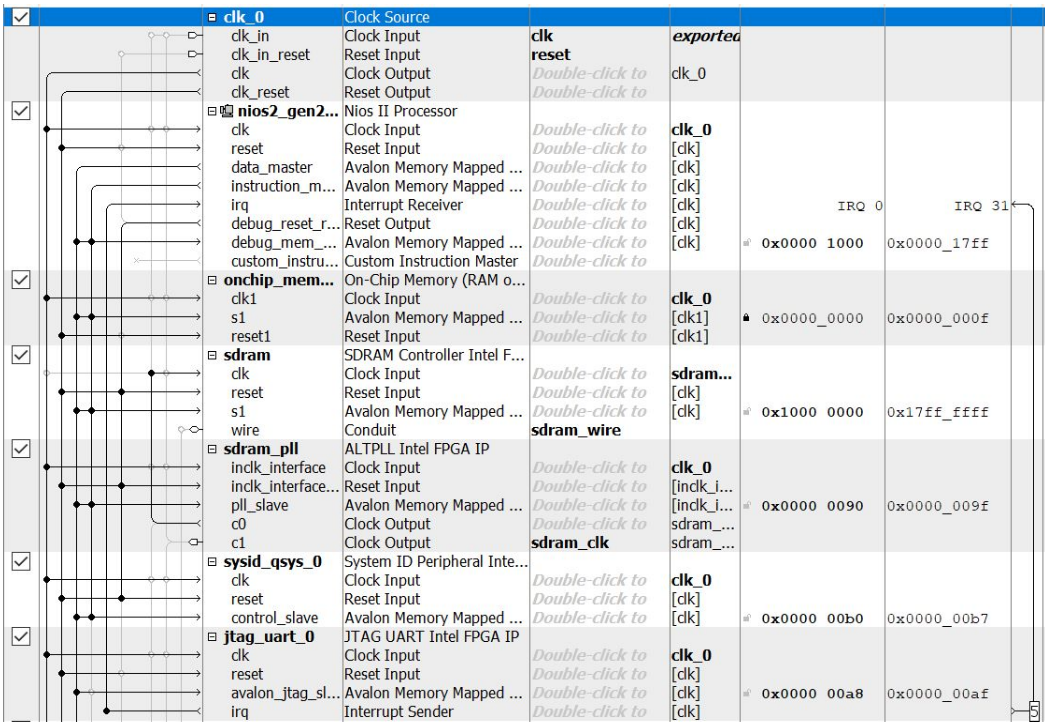

Figure 3.2: System Level Block Diagram (Part 1)

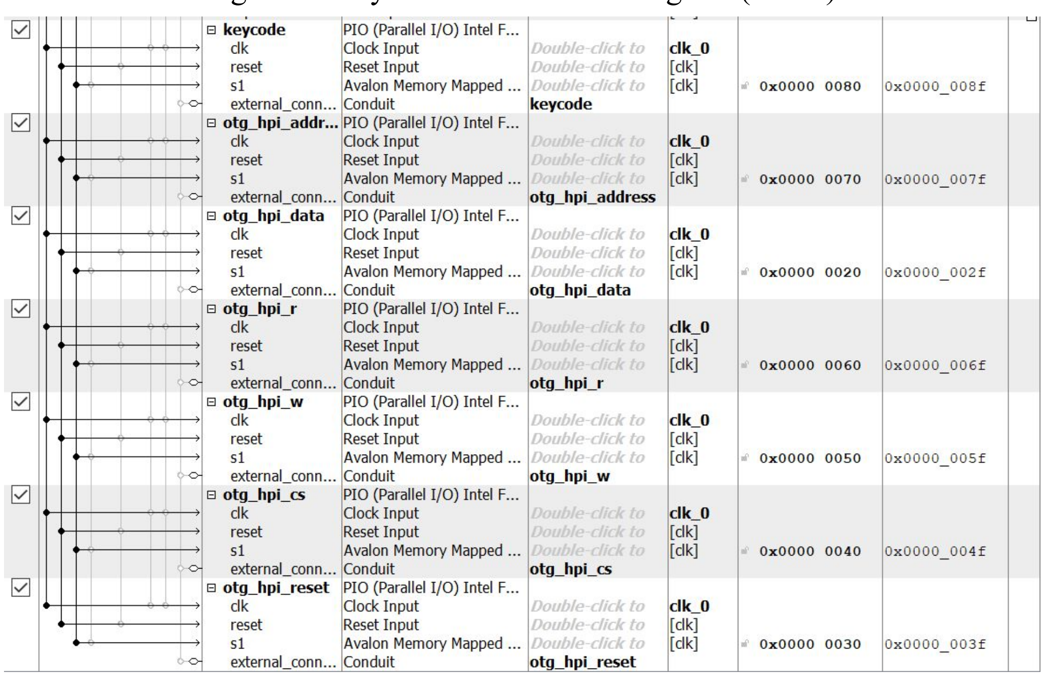

Figure 3.3: System Level Block Diagram (Part 2)

Descriptions

Module Descriptions

Module: **lab8.sv**
Inputs: CLOCK_50, [3:0] KEY, [7:0] SW, OTG_INT
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ
Outputs:[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, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, output logic [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N
Description: Top level module for lab 8.
Purpose: Manges IO between the board and all the other modules. Passes through outputs of the VGA controller to the display.

Module: **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
Description: This module handles the connections between the platform designed file and the IO.
Purpose: Serves as an interface between NIOS II and EZ-OTG chip.

Module: **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
Description: Outputs a vertical and horizontal video signal according to the VGA model for a video output to the display.
Purpose: Handles the appropriate VGA display output which is used to display the ball and the background.

Module: **HexDriver.sv**
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: Converts a 4-bit input into needed signals for 7-segment hex display on the FPGA board.
Purpose:Used to display the current values in PC/Switches/Memory contents in various states.

Module: **Color_Mapper.sv**
Inputs: is_ball, [9:0] DrawX, [9:0] DrawY
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: Outputs the RGB colors for the output video file.
Purpose: Creates a colorful background behind the moving ball on the display.

Module: **ball.sv**
Inputs: Clk, Reset, frame_clk, [7:0] keycode, [9:0] DrawX, [9:0] DrawY
Outputs: is_ball
Description: Uses the X and the Y coordinates of the given size ball to calculate the behavior of the ball for the next frame.
Purpose: Calculates the position of the ball object for each frame based on the keyboard inputs.

QSYS Block Descriptions

clk_0: A 50 MHz clock used to clock all the synchronous components of the design
nios2_gen2_0: NIOS 2 processor runs the compiled C code stored in SDRAM
onchip_memory2_0: NIOS 2's 16 byte memory
led: A PIO block that maps NIOS 2 processor to the green led on the board (allows code to drive LED pins)
sdram: Memory space used to store instructions and general storage
sdram_pll: Generates a 2nd clock that is out of phase with the main clock in order to ensure that when SDRAM refreshes, the input ports have time to stabilize
sysid_qsys_0: verification module for interfacing code with the board. It ensures that the code that is trying to get on the is meant for that board
otg_hpg_address: A PIO block indicates where a read/write is going to occur on the OTG address pins
otg_hpg_data: A PIO block that holds the data to be written into OTG pins
otg_hpg_r: A PIO block that enables data to be read from OTG pins
otg_hpg_w: A PIO block that enables data to be written to OTG pins
otg_hpg_cs: A PIO block that enables an operation to occur on the OTG
otg_hpg_reset: A PIO block that maps NIOS 2 processor to a push button on the board (allows button to directly affect code, i.e. clear the current sum on the LEDs)
keycode: A PIO block that maps NIOS 2 processor to a push button on the board (allows button to directly affect code)

Postlab

LUT	2,663
DSP	2 embedded multiplier blocks
Memory	55,296
Flip-Flop	2,115
Frequency (MHz)	85.22
Static Power (mW)	105.36
Dynamic Power (mW)	45.33
Total Power (mW)	228.85

Figure 5.1: Resources and Statistics

VGA_CLK and Clk are hooked up as two different independent signals. VGA_CLK and Clk are different signals as the VGA_CLK is used to refresh the video display by drawing pixel by pixel while Clk is used to drive the synthesized hardware. Because the two clocks serve different purposes, the VGA_CLK had to be a separate signal with a much lower frequency compared to the main Clk.

In the file "io_handler.h", the variable "otg_hpi_address" is of type integer pointer (int *) where as "otg_hpi_r" is of type character pointer (char *). "otg_hpi_address" is defined as an integer pointer because addresses are stored as 8-bit unsigned integers so a pointer to a memory location containing addresses as data should have the pointer type integer. Similarly, the data stored in read is of type char so the pointer to the memory location is of type char pointer.

Hidden Question 1

PS/2 keyboard sends a signal when a key is pressed. A USB keyboard requires the host to poll the keyboard for the pressed keys. One of the disadvantages for USB keyboards includes a dependence on the host to poll for keypresses so if the host device is in sleep USB keyboards cannot natively wake the host device. Additionally, an advantage of PS/2 keyboard has no limit registering the amount of keys pressed as opposed to USB keyboards which have a limit on registering multiple key presses.

Hidden Question 2

The old value of the ball will be used as the value of the ball is updated at the same time. When updating the Ball_Y_Pos, "Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion;" and "Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion_in;" seem similar but result in slightly different behavior. One of the statements has the next frame ball Y-position dependent on the current frame ball Y-position. On the other hand, the other statement has the current ball Y-position dependent on the next frame position of the ball leading to a frame delay. The implication of such a difference is the potential of the ball leaving the bounds of the display due to the delay in frames.

Conclusion

Our design allowed for a connected keyboard to move a ball on the VGA screen by pressing W, A, S, or D on the keyboard. Additionally when multiple keys were pressed on the keyboard, the ball did not move diagonally. The material given for the completion of this lab seemed sufficient. Specifically, the “Introduction to USB and EZ-OTG on NIOS II” guided us through the initial setup process.