ECE 385

Fall 2021

Experiment # 8

SOC with USB and VGA Interface in SystemVerilog

Xu Ke / Zhu Xiaohan

LA4/Thursday & 18:00-20:50 Huang Tianhao

I. Introduction

In this lab, we use C code VHDL with a monitor and a keyboard to create a very simple game in which we control a ball where it bounced off the walls and where it could not have travelled diagonally. It was controlled by *W, A, S* and *D* key to travel up, left, down and right, and after each key press, the corresponding ASCII value will also be displayed on the FPGA board. The monitor can display content via VGA connection, and the keyboard sends data to the FPGA board using USB interface. Video Graphics Array (VGA) is a kind of I/ O with 15 pins, which are used to send the analog component RGBHV video signals, where RGBHV stands for red, green, blue, horizontal SUNC and vertical SYNC. Universal Serial Bus (USB) is an industry standard that sets requirements for cables and connectors and protocols for connectivity, communication, and power supply between computers, peripherals, and other devices. The NIOS II chip can handle keycodes sent from the keyboard, and the DE2 board has a USB controller to handle data transfers.

II. Written description of the operation of your circuit

1. Written description of the entire Lab 8 system

In this lab, we have NIOS interaction similar to Lab7 since some modules are reused (i.e. CLK_0, NIOS2_GEN2_0, Onchip_Memory2_0, SDRAM, and SDRAM_PLL), which are the backbones of the system operation. Clk_0 generates a clock signal that is to synchronize all components on the same clock. Nios2_gen2_0 is our processor to perform operations and read instructions. We have onchip_memory2_0 for the memory storage, which is faster than SDRAM but much less in storage space. SDRAM_PLL generates the clock that enters SDRAM 3ns later than the system clock to give the output enough time to stabilize.

One of the new modules we introduced is the JTAG_UART module, which allows the communication with NIOS II using terminals on computer. The other modules we have are hPI_address (select the HPI register to write), HPI_CS (chip select), HPI_R (read), HPI_W (write), and oTG_hPI_data (our data), which is specific to the way the Cypress EZ-OTG (CY7C67200) chip that handles the protocol.

2. Written description of the USB protocol

a) IO_write and IO_read:

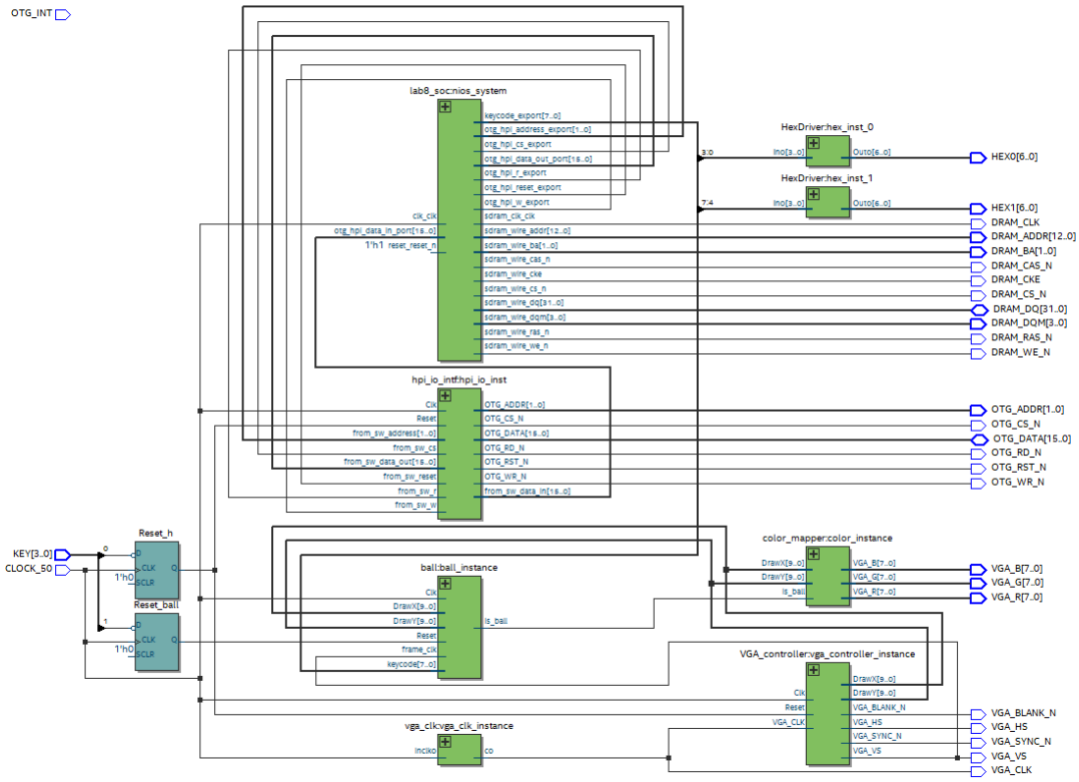
The Nios II handles the USB protocol with its software, and the keycode that is extracted is outputted to other hardware that checks for keypresses (ie. ball.sv). We had to write out functions such as IO_write and IO_read based on the datasheet showed the timing diagram and the order of the operations. For IO_write: 1) Write address in hpi_address location 2) Set cs to 0 (active low) 3) Set w to 0 (active low) 4) Write data in hpi_data location 5) Set w to1 6) Set cs to w; And for IO_read: 1) Write address in hpi_address location 2) Set cs to 0 (active low) 3) Set w to 0 (active low) 4) Store

hpi_data in temp variable 5) Set w to1 6) Set cs to1 7) Return temp. The order of the signals is very important in order to match up with the protocol, and incorrect order/ programming can lead to failure of reads or writes

b) USBRead and USBWrite:

There is a tri-state buffer between EZ-OTG and NIOS II. The inputs are from NIOS II and they are gated by 1 gate delay because of the tri-state buffer. OTG-Address will determine which register of EZ-OTG the NIOS II will write to. To be specific, 00 will correspond to HPI DATA, 01 will correspond to HPI MAILBOX,10 will correspond to HPI ADDRESS, 11 will correspond to HPI STATUS. NIOS II will perform USB-Write and USB-Read to interact with HPI registers.

3. Block diagram of Top Level



4. Module descriptions

a) VGA Controller

```
module  VGA_controller (input              Clk,          // 50 MHz clock
                                           Reset,        // Active-high reset signal
                        output logic       VGA_HS,       // Horizontal sync pulse.  Active low
                                           VGA_VS,       // Vertical sync pulse.  Active low
                        input              VGA_CLK,      // 25 MHz VGA clock input
                        output logic       VGA_BLANK_N,  // Blanking interval indicator.  Active low.
                                           VGA_SYNC_N,   // Composite Sync signal.  Active low.  We don't use it in this lab,
                                                         // but the video DAC on the DE2 board requires an input for it.
                        output logic [9:0] DrawX,        // horizontal coordinate
                                           DrawY         // vertical coordinate
                        );

    // 800 pixels per line (including front/back porch)
    // 525 lines per frame (including front/back porch)
    parameter [9:0] H_TOTAL = 10'd800;
    parameter [9:0] V_TOTAL = 10'd525;

    logic VGA_HS_in, VGA_VS_in, VGA_BLANK_N_in;
    logic [9:0] h_counter, v_counter;
    logic [9:0] h_counter_in, v_counter_in;

    assign VGA_SYNC_N = 1'b0;
    assign DrawX = h_counter;
    assign DrawY = v_counter;

    // VGA control signals.
    // VGA_CLK is generated by PLL, so you will have to manually generate it in simulation.
    always_ff @ (posedge VGA_CLK)
    begin
        if (Reset)
        begin
            VGA_HS <= 1'b0;
            VGA_VS <= 1'b0;
            VGA_BLANK_N <= 1'b0;
            h_counter <= 10'd0;
            v_counter <= 10'd0;
        end
        else
        begin
            VGA_HS <= VGA_HS_in;
            VGA_VS <= VGA_VS_in;
            VGA_BLANK_N <= VGA_BLANK_N_in;
            h_counter <= h_counter_in;
            v_counter <= v_counter_in;
```

Inputs: Clk, Reset, VGA_CLK,

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

Description: This module handles the synchronization of signals where VS implies vertical sync and HS implies horizontal sync of the VGA signal we are outputting in addition to "drawing" pixels

Purpose: This module is used to display the ball bouncing on the screen, as an output from the FPGA

b) Ball

```
module  ball ( input          clk,                  // 50 MHz clock
                              Reset,                // Active-high reset signal
                              frame_clk,            // The clock indicating a new frame (~60Hz)
               input [7:0]    keycode,
               input [9:0]    DrawX, DrawY,         // Current pixel coordinates
               output logic   is_ball               // Whether current pixel belongs to ball or background
              );

    parameter [9:0] Ball_X_Center = 10'd320;  // Center position on the X axis
    parameter [9:0] Ball_Y_Center = 10'd240;  // Center position on the Y axis
    parameter [9:0] Ball_X_Min = 10'd0;       // Leftmost point on the X axis
    parameter [9:0] Ball_X_Max = 10'd639;     // Rightmost point on the X axis
    parameter [9:0] Ball_Y_Min = 10'd0;       // Topmost point on the Y axis
    parameter [9:0] Ball_Y_Max = 10'd479;     // Bottommost point on the Y axis
    parameter [9:0] Ball_X_Step = 10'd1;      // Step size on the X axis
    parameter [9:0] Ball_Y_Step = 10'd1;      // Step size on the Y axis
    parameter [9:0] Ball_Size = 10'd4;        // Ball size

    logic [9:0] Ball_X_Pos, Ball_X_Motion, Ball_Y_Pos, Ball_Y_Motion;
    logic [9:0] Ball_X_Pos_in, Ball_X_Motion_in, Ball_Y_Pos_in, Ball_Y_Motion_in;
    logic [7:0] W = 8'h1A;
    logic [7:0] A = 8'h04;
    logic [7:0] S = 8'h16;
    logic [7:0] D = 8'h07;

    //////// Do not modify the always_ff blocks. ////////
    // Detect rising edge of frame_clk
    logic frame_clk_delayed, frame_clk_rising_edge;
    always_ff @ (posedge Clk) begin
        frame_clk_delayed <= frame_clk;
        frame_clk_rising_edge <= (frame_clk == 1'b1) && (frame_clk_delayed == 1'b0);
    end
    // Update registers
    always_ff @ (posedge Clk)
    begin
        if (Reset)
        begin
            Ball_X_Pos <= Ball_X_Center;
            Ball_Y_Pos <= Ball_Y_Center;
            Ball_X_Motion <= 10'd0;
            Ball_Y_Motion <= Ball_Y_Step;
        end
```

Inputs: Clk, Reset, Frame_CLK, [7:0] Keycode, [9:0] DrawX, [9:0] DrawY
Outputs: logic is_ball
Description: This module updates the position and motion of the ball only at the rising edge of frame clock and if no keys are pressed it keeps the motion unchanged.
Purpose: This module is used to calculate the positions and reacts to keypresses which are from the user via the keyboard.

c) Color Mapper

```
module  color_mapper ( input              is_ball,            // Whether current pixel belongs to ball
                                                                //   or background (computed in ball.sv)
                       input        [9:0] DrawX, DrawY,        // Current pixel coordinates
                       output logic [7:0] VGA_R, VGA_G, VGA_B  // VGA RGB output
                     );

    logic [7:0] Red, Green, Blue;

    // Output colors to VGA
    assign VGA_R = Red;
    assign VGA_G = Green;
    assign VGA_B = Blue;

    // Assign color based on is_ball signal
    always_comb
    begin
        if (is_ball == 1'b1)
        begin
            // White ball
            Red = 8'hff;
            Green = 8'hff;
            Blue = 8'hff;
        end
        else
        begin
            // Background with nice color gradient
            Red = 8'h3f;
            Green = 8'h00;
            Blue = 8'h7f - {1'b0, DrawX[9:3]};
        end
    end

endmodule
```

Inputs: is_ball, [9:0] DrawX, [9:0] DrawY
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: This module decides which color to be output to VGA for each pixel and whether the pixel belongs to ball or background and uses RGB color selection.
Purpose: This module is used to draw the ball, background, and implement RGB colors on screen.

d) hpi_io_intf

```
module hpi_io_intf( input        Clk, Reset,
                    input [1:0]  from_sw_address,
                    output[15:0] from_sw_data_in,
                    input [15:0] from_sw_data_out,
                    input        from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, // Active low
                    inout [15:0] OTG_DATA,
                    output[1:0]  OTG_ADDR,
                    output       OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N // Active low
                   );

// Buffer (register) for from_sw_data_out because inout bus should be driven
//   by a register, not combinational logic.
logic [15:0] from_sw_data_out_buffer;

// TODO: Fill in the blanks below.
always_ff @ (posedge Clk)
begin
    if(Reset)
    begin
        from_sw_data_out_buffer <= 16'h0000;
        OTG_ADDR                <= 2'b00;
        OTG_RD_N                <= 1'b1;
        OTG_WR_N                <= 1'b1;
        OTG_CS_N                <= 1'b0;
        OTG_RST_N               <= 1'b0;
        from_sw_data_in         <= 16'h0000;
    end
    else
    begin
        from_sw_data_out_buffer <= from_sw_data_out;
        OTG_ADDR                <= from_sw_address;
        OTG_RD_N                <= from_sw_r;
        OTG_WR_N                <= from_sw_w;
        OTG_CS_N                <= from_sw_cs;
        OTG_RST_N               <= 1'b1;
        from_sw_data_in         <= OTG_DATA;
    end
end

// OTG_DATA should be high Z (tristated) when NIOS is not writing to OTG_DATA inout bus.
// Look at tristate.sv in lab 6 for an example.
assign OTG_DATA = ~from_sw_w ? from_sw_data_out_buffer : {16'bz};

endmodule
```

Inputs: Clk, Reset, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, [15:0] from_sw_data_out, [1:0] from_sw_address

Outputs: OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N, [1:0] OTG_ADDR, [15:0] from_sw_data_in,

Description: This module is the interface between NIOS II and EZ-OTG chip, a hardware tri-state buffer using buffer (register) for from_sw_data_out.

Purpose: This module is used to send read, write, cs, reset, data and address signals to the EZ-OTG chip, and OTG_DATA should be high Z (tristated) when NIOS is not writing to OTG_DATA inout bus.

e) lab8

```
module lab8( input                CLOCK_50,
             input        [3:0]   KEY,          //bit 0 is set up as Reset
             output logic [6:0]   HEX0, HEX1,
             // VGA Interface
             output logic [7:0]   VGA_R,        //VGA Red
                                  VGA_G,        //VGA Green
                                  VGA_B,        //VGA Blue
             output logic         VGA_CLK,      //VGA Clock
                                  VGA_SYNC_N,   //VGA Sync signal
                                  VGA_BLANK_N,  //VGA Blank signal
                                  VGA_VS,       //VGA virtical sync signal
                                  VGA_HS,       //VGA horizontal sync signal
             // CY7C67200 Interface
             inout  wire  [15:0] OTG_DATA,      //CY7C67200 Data bus 16 Bits
             output logic [1:0]  OTG_ADDR,      //CY7C67200 Address 2 Bits
             output logic        OTG_CS_N,      //CY7C67200 Chip Select
                                 OTG_RD_N,      //CY7C67200 Write
                                 OTG_WR_N,      //CY7C67200 Read
                                 OTG_RST_N,     //CY7C67200 Reset
             input               OTG_INT,       //CY7C67200 Interrupt
             // SDRAM Interface for Nios II Software
             output logic [12:0] DRAM_ADDR,     //SDRAM Address 13 Bits
             inout  wire  [31:0] DRAM_DQ,       //SDRAM Data 32 Bits
             output logic [1:0]  DRAM_BA,       //SDRAM Bank Address 2 Bits
             output logic [3:0]  DRAM_DQM,      //SDRAM Data Mast 4 Bits
             output logic        DRAM_RAS_N,    //SDRAM Row Address Strobe
                                 DRAM_CAS_N,    //SDRAM Column Address Strobe
                                 DRAM_CKE,      //SDRAM Clock Enable
                                 DRAM_WE_N,     //SDRAM Write Enable
                                 DRAM_CS_N,     //SDRAM Chip Select
                                 DRAM_CLK       //SDRAM Clock
                    );

    logic Reset_h, Clk;
    logic [7:0] keycode;
    logic [9:0] DrawX,DrawY;
    logic is_ball;

    assign Clk = CLOCK_50;
    always_ff @ (posedge Clk) begin
        Reset_h <= ~(KEY[0]);        // The push buttons are active low
        Reset_ball <= ~(KEY[1]);     // Reset the ball's position
    end
```

Inputs: CLOCK_50, [3:0] KEY, OTG_INT

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

Inouts: [15:0] OTG_DATA, [31:0] DRAM_DQ

Description: This module is the top level, all inputs and outputs go through it, and this module helps them communicate with one another.

Purpose: This module is used to connect the NIOS to all the blocks and drivers.

f) Platform Designer Modules

clk_0	Clock Source
clk_in	Clock Input
clk_in_reset	Reset Input
clk	Clock Output
clk_reset	Reset Output

This is the clock module which simply the 50Mhz generated by the FPGA. The clk goes from here to all the other clocks inputs

⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM) I...
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input

This is our on-chip memory, which is often smaller than SRAM in size but faster and actually on the chip. The data width is 32 bits and the total memory size is 16 bytes

⊟ **sdram**	SDRAM Controller Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
wire	Conduit

This is our SDRAM that we use to store the software program due to the limited on-chip memory. We have to use an SDRAM controller to interface with the bus since we have row/column addressing and constantly needs to refresh in order to retain data.

⊟ **sdram_pll**	ALTPLL Intel FPGA IP
inclk_interface	Clock Input
inclk_interface_...	Reset Input
pll_slave	Avalon Memory Mapped Slave
c0	Clock Output
c1	Clock Output

This module generates the clock that goes into the SDRAM. The PLL allows us to account for delays, specifically 3ns in order to have the SDRAM wait for the outputs to stabilize.

⊟ **sysid_qsys_0**	System ID Peripheral Intel FP...
clk	Clock Input
reset	Reset Input
control_slave	Avalon Memory Mapped Slave

This is an ID checker which ensure the compatibility between hardware and software.

⊟ **nios2_gen2_0**	Nios II Processor
clk	Clock Input
reset	Reset Input
data_master	Avalon Memory Mapped Master
instruction_master	Avalon Memory Mapped Master
irq	Interrupt Receiver
debug_reset_request	Reset Output
debug_mem_slave	Avalon Memory Mapped Slave
custom_instructi...	Custom Instruction Master

This is an IP based 32-bit CPU which can programmed using a high-level language.

⊟ **keycode**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple 8 bit-wide PIO block, which outputs the keycode from the IO_READ (keyboard).

⊟ **otg_hpi_address**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which outputs the 2-bit value corresponding to the specific HPI register.

⊟ **otg_hpi_data**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple 32 bit-wide PIO block, which is inout because data is both read from and written to here.

⊟ **otg_hpi_r**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “read” enable signal

⊟ **otg_hpi_w**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “write” enable signal

⊟ **otg_hpi_cs**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “chip enable” signal

⊟ **otg_hpi_reset**	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit

This is a simple PIO block, which is a 1bit output corresponding to a “reset” signal

III. Answers to both hidden questions

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

One advantage is that if you want to use PS/2 mouse or keyboard, you need to shut down the system, plug it in then reboot it while for USB you can just plug it into the system and after a second or so you can use it.

One disadvantage is that PS/2 has faster response than USB because PS/2 directly connects to the MOBO while USB connects to the BUS which then makes contact with the MOBO.

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 new value of Ball_Y_Motion_in will be used when Ball_Y_Pos is updated.

Ball_Y_Pos_in and Ball_Y_Motion_in indicates the next position and next movements, while Ball_Y_Pos and Ball_Y_Motion indicates the current position and current movements

If we write "Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion", when the ball reaches the edge of the screen, it will bounce back in the next cycle which means it will run into the wall a little bit. And when we press the key (not in the same direction as the previous one), it will also be a little bit delay (one cycle) until it turns. But if we write "Ball_Y_Pos_in = Ball_Y_Pos + Ball_Y_Motion_in;" As soon as we press the key (not in the same direction as the previous one) or reaches the wall. The ball changes direction immediately in that cycle which is what we want.

V. Answers to post-lab questions

1. What is the difference between VGA_clk and Clk?

VGA_Clk runs at 25Mhz where as Clk runs at 50 Mhz while the VGA_Clk is used to update the monitor frames while the Clk is used for the FPGA.

2. 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?

Because an integer is 32 bits while a char is 8 characters, for otg_hpi_data we need the integer pointer due to 32-bit-wide data while for otg_hpi_r we only need a char pointer to point to the HPI registers because we only have 4 registers and the system cannot allocate less than a byte. Thus, defining otg_hpi_data as an integer pointer and otg_hpi_r as a char pointer can save space and can satisfy the demand.

3. Document the Design Resources and Statistics in following table.

LUT	**2688**
DSP	**10 (2+2+4+2)**
Memory (BRAM)	**55296**
Flip-Flop	**2236**
Frequency	**118.12mHZ**
Static Power	**105.28mV**
Dynamic Power	**27.66mW**
Total Power	**207.12mW**

VI. Conclusion

In general, there were few problems during the demo, except that the ball might move diagonally in some corners, which we solved by reversing one direction after the collision and clearing the other. On a hexadecimal display, the pressed key displays correctly on the FPGA board.

Although the lab seems to be difficult at first, as we understood the different aspects and worked on it, the pieces began to decline together. The most confusing part and biggest problem was that a keyboard didn't work with our board sometimes, which caused a lot of confusion and took some time to the debug but easy problem to solve with the keyboard from the laboratory. Generally speaking, the lab provides a good introduction to implementing I/O and gives us inspiration about our final project.