Lab 3: Input from a PS/2 Keyboard

Cory Brynds

cory.brynds@ucf.edu

EEL5722C: FPGA Design

Section 0012

Due: 18 October 2023

Submitted: 18 October 2023

1.0 Objectives

The objective of this experiment is to use the USB port on the BASYS 3 FPGA development board to interface with a USB keyboard that supports the PS/2 communication protocol. When a key on the number row is pressed, the corresponding number is displayed on the rightmost digit of the seven-segment display. If any other key besides the number row keys is pressed, the display will turn off.

2.0 Equipment

- Xilinx Vivado Design Suite
- BASYS 3 FPGA Development Board
- USB Keyboard

3.0 Experimental Explanation

3.1 Description of Implementation

While modern-day keyboards have largely moved on from PS/2 communication to the USB standard, most still support backward compatibility with the protocol. The BASYS 3 supports emulation of a PS/2 data bus through a PIC24FJ128 microcontroller, which provides the FPGA with the ability to act as a USB HID host. With a single mouse or keyboard connected to the BASYS 3, the PIC24 sends signals to the FPGA, two of which, PS2_CLK and PS2_DAT, implement the PS/2 interface.

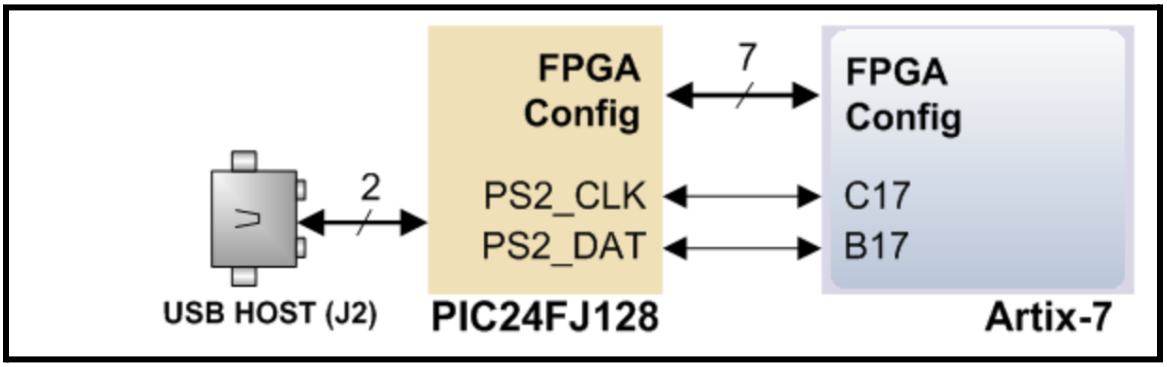
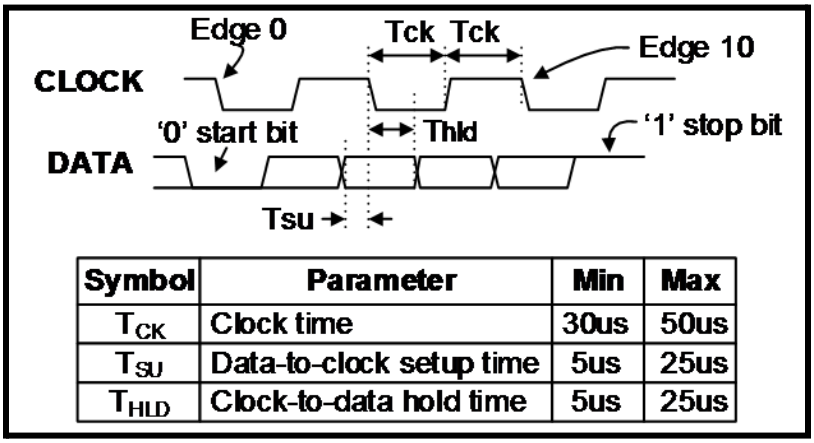


Figure 1: Diagram of USB connection to PS/2 interface (Source: Digilent Reference Manual)

The PS/2 protocol uses two bus wires, clock and data, to communicate between the peripheral and host device. Each message sent from the peripheral to the host is in the form of an 11-bit word of the following structure:

Start bit	Data byte (8 bits)	Odd parity bit	Stop bit



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Figure 2: PS/2 timing information (Source: Digilent reference manual)

When a keyboard is initially connected to the BASYS 3, a command of 0xAA is sent to the FPGA before commands can be received. After this "self-test passed", the PS/2 interface uses scancodes, where each key on the keyboard is assigned a unique code. Upon keypress, the code is sent to the receiver, and it is transmitted every ~100ms if the key is held down. To handle special functions such as shift and extended keys, additional commands are sent, but that is beyond the scope of this experiment. The following figure displays the scancode for each key.

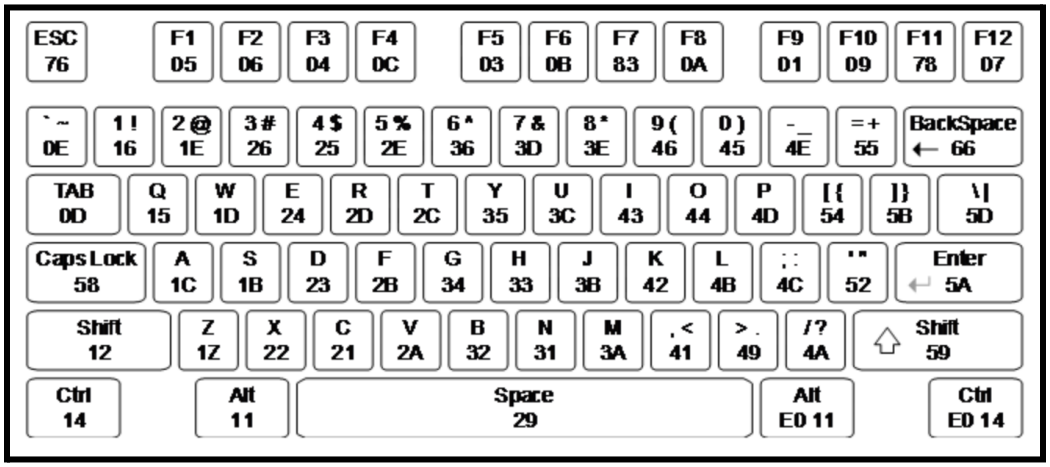


Figure 3: Key scancode mappings (Source: Digilent reference manual)

To implement the decimal counter, the rightmost digit of the seven-segment display was used. When a key from the number row of the USB keyboard was pressed, the corresponding number was displayed on the seven-segment display. In the event that a key other than the number row was pressed, or an unrecognized scancode was received, all of the seven segments on the display would turn off. Below is a block diagram of the different modules used to implement this project.

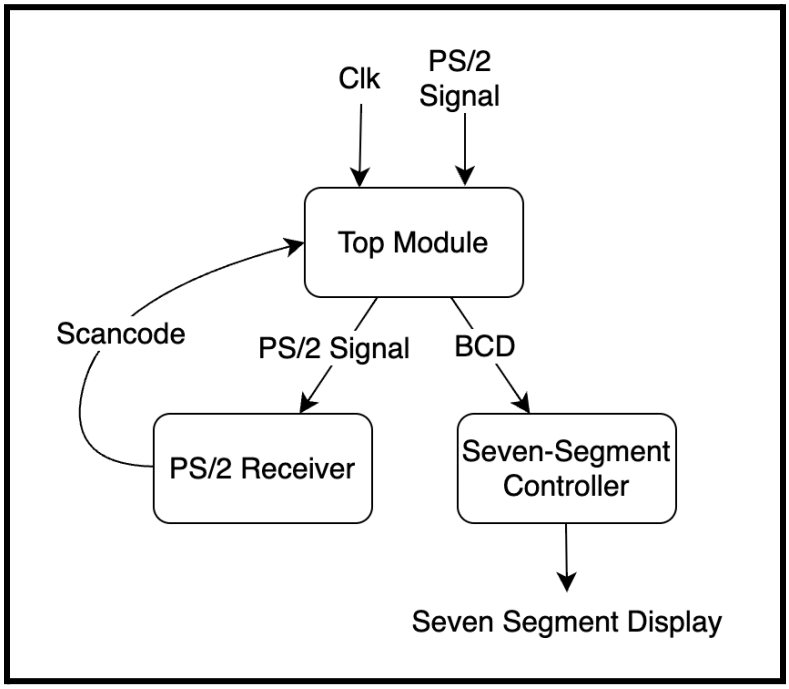


Figure 4: Block diagram of Verilog modules

3.2 Module Descriptions

Top module: Instantiates the PS/2 receiver and seven-segment controller modules. Implements a case statement based on the scancode data to map the scancodes for '0' - '9' to the corresponding BCD, which is then passed to the seven-segment controller module.

PS/2 Receiver: Takes the clock, PS/2 clock, and key data as input and outputs the scancode value based on the received bitstream from the keyboard.

Seven-segment controller: Instantiates the module to convert the BCD digit to the common anode configuration for the display and selects the rightmost digit of the display.

Seven-segment converter: Takes the BCD value for the digit as an input and outputs the corresponding 7-bit value to drive the common-anode seven-segment display.

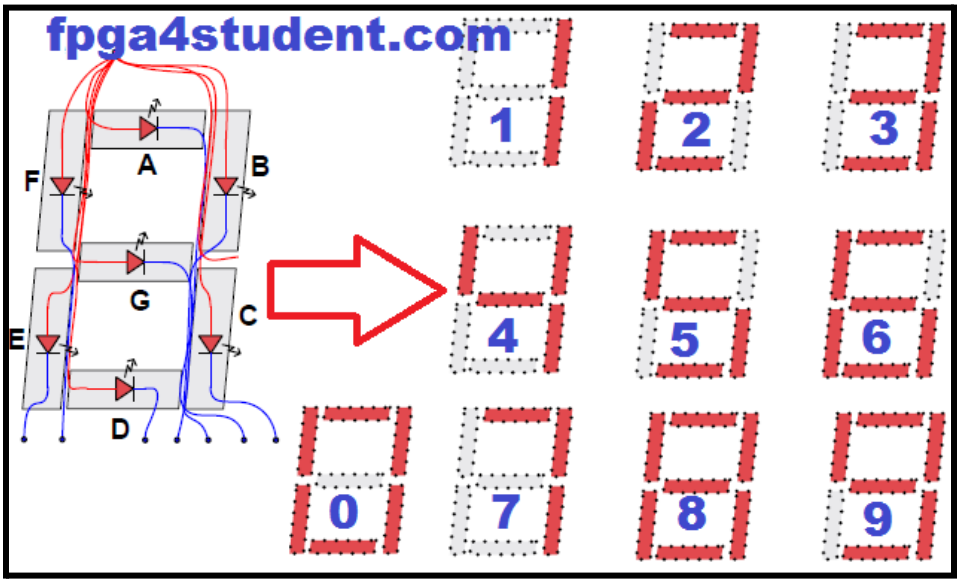


Figure 5: Seven-segment display diagram (source fpga4student.com)

4.0 Results

4.1 Verilog Modules

```
// Driver module to instantiate control logic and map scancodes to BCD values
module top(
	input clk, PS2clk, key_data,
	output [6:0] seg,
	output [3:0] an
);

reg[3:0] dec;
wire[7:0] c_data;

// Instantiate the module to decode the PS/2 signals
PS2_receiver receiver(
	.clk(clk),
	.PS2clk(PS2clk),
	.key_data(key_data),
	.c_data(c_data)
);

// Instantiate the module for the control logic of the seven segment display
seven_segment_controller controller(
	.clk(clk),
	.digit(dec),
	.seg(seg),
	.an(an)
);

// Assigns PS/2 scancodes to the corresponding BCD
always @ (c_data) begin
	case (c_data)
		8'h45:
			dec = 4'b0000;
		8'h16:
			dec = 4'b0001;
		8'h1E:
			dec = 4'b0010;
		8'h26:
			dec = 4'b0011;
		8'h25:
```

```
                        dec = 4'b0100;
                8'h2E:
                        dec = 4'b0101;
                8'h36:
                        dec = 4'b0110;
                8'h3D:
                        dec = 4'b0111;
                8'h3E:
                        dec = 4'b1000;
                8'h46:
                        dec = 4'b1001;
                default:
                        dec = 4'b1010;
        endcase
end

endmodule

// Reads the bitstream from the USB keyboard and decodes the corresponding scancode
module PS2_receiver(
        input clk, PS2clk, key_data,
        output reg [7:0] c_data
);

reg [10:0] store;
reg [3:0] count;
wire parity_check;

// Store sets of 11 bits into the 'store' register
always @ (negedge PS2clk) begin
        if (count < 10) begin
                store[count] <= key_data;
                count <= count + 1;
        end
        else begin
                count <= 0;
        end
end

// Check for parity bit to signal the end of a PS/2 signal
assign parity_check = ~((store[1] ^
        store[8])^(store[2]^store[3])^(store[4]^store[5])^(store[6]^store[7]));
```

```
// If a valid command has been issued, store scancode in c_data
always @ (posedge clk) begin
	// If the start of a command has been issued
	if (store[0] == 0 && parity_check == store[9]) begin
		c_data <= store[8:1];
	end
end

endmodule

// Sets the rightmost seven-segment display as active and instantiates the module to
// convert from BCD -> common anode configuration
module seven_segment_controller(
	input clk,
	input [3:0] digit,
	output [6:0] seg,
	output [3:0] an
);
seven_segment_converter converter(.dec(digit), .seg(seg));

assign an = 4'b1110;
endmodule

// Converts from BCD representation -> LED configuration to display digits in common
anode
module seven_segment_converter(
	input [3:0] dec,
	output reg [6:0] seg
);
always @ (*) begin
	case (dec)
		4'b0000: seg = 7'b1000000; // or 7'h3F
		4'b0001: seg = 7'b1111001; // or 7'h06
		4'b0010: seg = 7'b0100100; // or 7'h5B
		4'b0011: seg = 7'b0110000; // or 7'h4F
		4'b0100: seg = 7'b0011001; // or 7'h66
		4'b0101: seg = 7'b0010010; // or 7'h6D
		4'b0110: seg = 7'b0000010; // or 7'h7D
		4'b0111: seg = 7'b1111000; // or 7'h07
		4'b1000: seg = 7'b0000000; // or 7'h7F
		4'b1001: seg = 7'b0010000; // or 7'h4F
```

```
            default: seg = 7'b1111111;
        endcase
end
endmodule
```

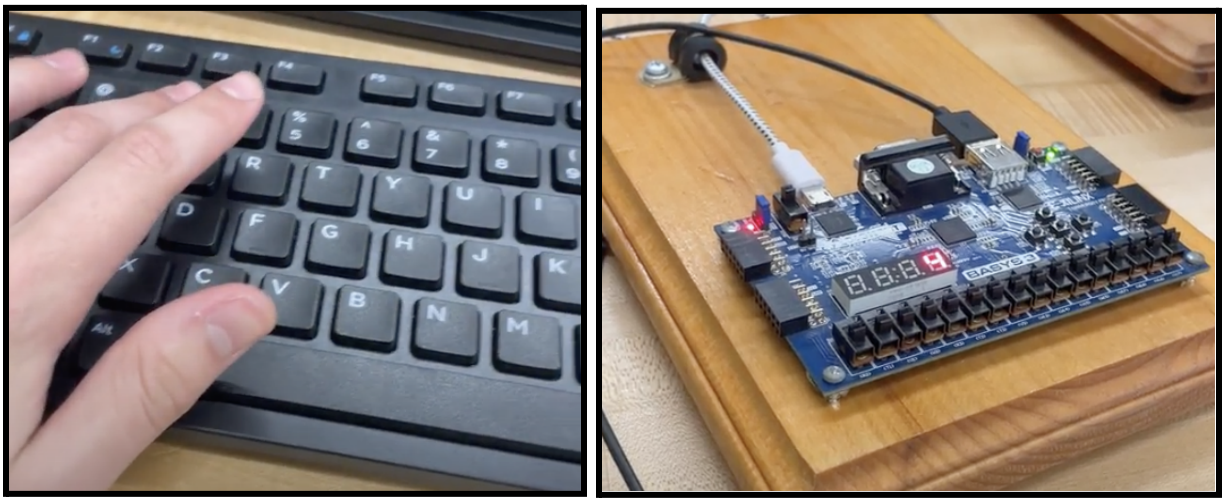
4.2 Constraints File

```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add [get_ports
clk]

#7 segment display
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

set_property PACKAGE_PIN V7 [get_ports dp]
set_property IOSTANDARD LVCMOS33 [get_ports dp]

set_property PACKAGE_PIN U2 [get_ports {an[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
set_property PACKAGE_PIN U4 [get_ports {an[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
set_property PACKAGE_PIN V4 [get_ports {an[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
set_property PACKAGE_PIN W4 [get_ports {an[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]

#USB HID (PS/2)
```

```
set_property IOSTANDARD LVCMOS33 [get_ports PS2clk]
set_property IOSTANDARD LVCMOS33 [get_ports key_data]
set_property PACKAGE_PIN C17 [get_ports PS2clk]
set_property PACKAGE_PIN B17 [get_ports key_data]
```

4.3 Video of Experimental Implementation

Video of Experimental Implementation



Figures 6-7: Screen captures from the video of the experimental implementation

4.4 Explanation of Results

The results of this experiment were as expected. When one of the number keys (0-9) was pressed, the corresponding digit would appear on the rightmost segment of the seven-segment display. When a non-number row key was pressed, all of the segments would turn off. There were no noticeable bugs or glitches with this implementation.

Overall, the implementation of this experiment was more straightforward than the VGA driver module from the previous experiment, for a few reasons. First of all, while the PS/2 communication protocol has been mostly superseded by USB, most keyboards support backward compatibility, and there was rich documentation available to implement the PS/2 protocol. Secondly, there were much fewer timings and signals to keep track of than with the VGA protocol, as PS/2 only has a clock and data line. As a result, very few issues were encountered when implementing this experiment on the BASYS 3 board.

The largest challenge with this experiment was simply understanding the PS/2 protocol and how data words were organized. The concept of odd versus even parity needed to be explored, and a corresponding function needed to be implemented to check the data bits for odd parity. After this issue was resolved, it was relatively straightforward to map the 11 data bits into the 8-bit scancode used to represent each keystroke. Finding which keypress corresponded to each scancode was straightforward, as the Digilent reference manual provided ample documentation.

5.0 Conclusion

Through this experiment, a deeper understanding of the interaction between hardware and software was developed. The task required the creation of Verilog modules to interpret PS/2 communication signals from a USB keyboard and drive a corresponding seven-segment display, and it demonstrated how external inputs can be translated into hardware actions through Verilog code.

One of the major components of this experiment was the designing of logic to interpret the keyboard inputs and drive the seven-segment display. The logic needed to be robust to ensure that only the digits 0-9 from the number row were displayed, while any other key press resulted in the display being turned off. Additionally, this experiment showed the potential for applications of FPGA-based systems, particularly in the realm of human-machine interfaces. The ability to interface external hardware like a keyboard with an FPGA and drive specific outputs opens the door for more complex systems capable of a wide range of responses based on user input.

In conclusion, this project bridged theoretical knowledge with practical application, providing a better understanding of FPGA design, programming in Verilog, and human-machine interactions.

6.0 References

Basys 3 Reference Manual. Digilent Reference. Digilent. https://digilent.com/reference/programmable-logic/basys-3/reference-manual.

Draw.io. https://app.diagrams.net/.