Lab 4: Display Characters to a Screen

Cory Brynds

cory.brynds@ucf.edu

EEL5722C: FPGA Design

Section 0012

Due: 2 November 2023

Submitted: 1 November 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 a VGA monitor in the top lefthand corner. If any other key is pressed, the letter ‘E’ will be displayed instead. If the return key is pressed, then the character currently displayed on the screen will be erased. Only one character may be displayed at a time. The pixel information for each character is in the .coe file format and is stored in single-port ROM. As a bonus, whichever character is displayed on the monitor will also be displayed on the seven-segment display on the BASYS 3.

2.0 Equipment

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

3.0 Experimental Explanation

3.1 Description of Implementation

3.1A VGA Monitor

To display graphics to a 640x480 resolution monitor, the BASYS 3 FPGA transmits the necessary signals and information using the VGA video standard. According to Digilent’s BASYS 3 reference manual, for the VGA standard,

> “Video data typically comes from a video refresh memory, with one or more bytes assigned to each pixel location (the Basys 3 uses 12 bits per pixel). The controller must index into video memory as the beams move across the display, and retrieve and apply video data to the display at precisely the time the electron beam is moving across a given pixel.”

The following diagram, also from the reference manual, shows the timing and signal information necessary for driving the display.

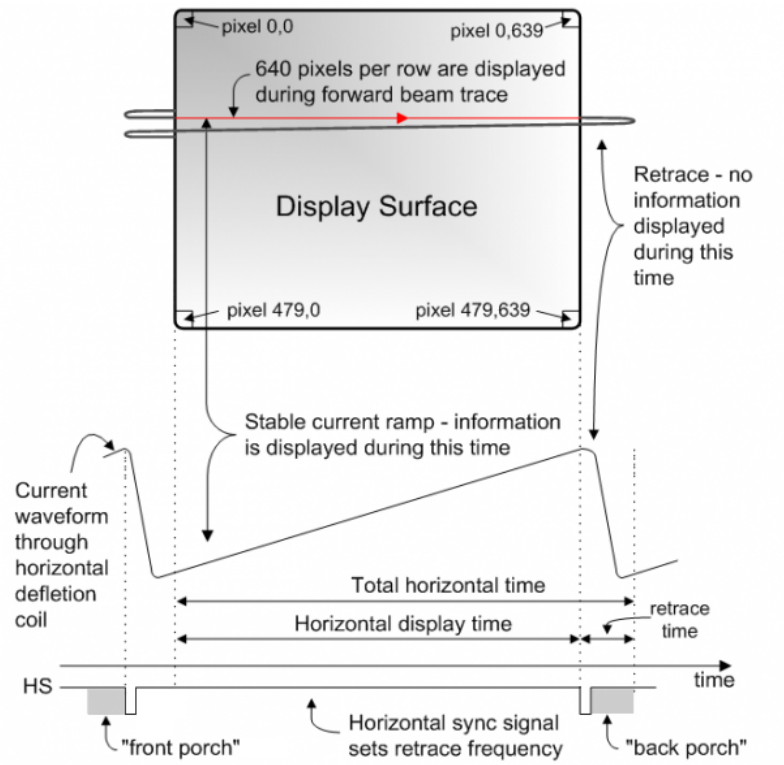


Figure 1: VGA display dimensions and timing (Source: Digilent)

To generate the necessary hsync, vsync, and color signals, wires must be declared in Verilog that map to each of the pins in the following 15-pin VGA connector.

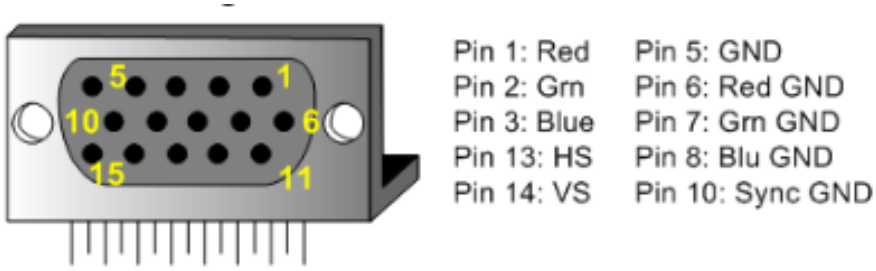


Figure 2: 15-pin VGA port (Source: Digilent)

Each color (red, green, and blue) is represented by 4 bits, meaning that there are 2^12 or 4096 possible colors able to be displayed on the screen.

3.1B PS/2 Keyboard

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.

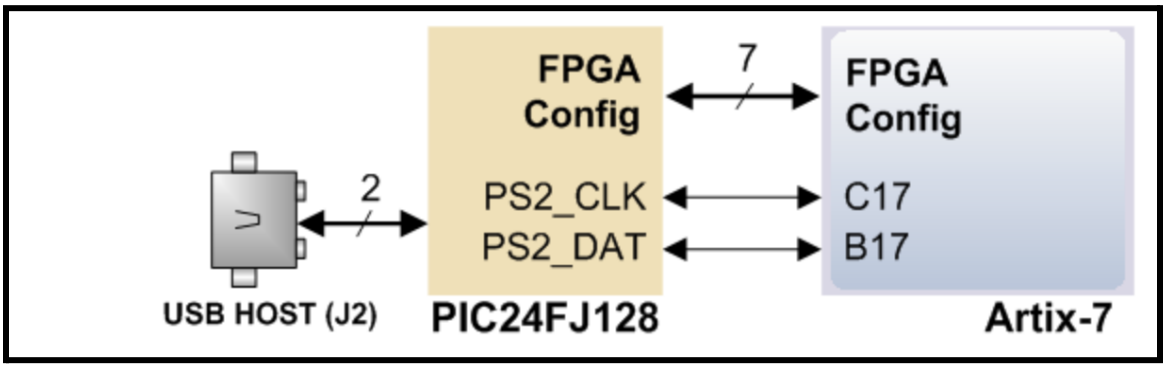
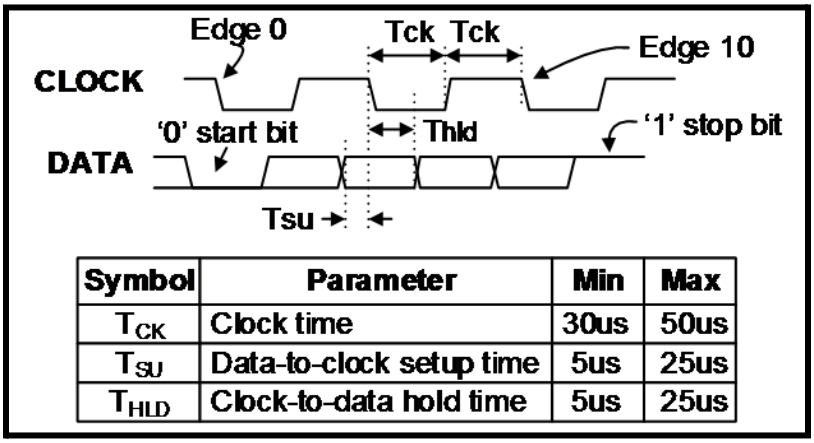


Figure 3: Diagram of USB connection to PS/2 interface (Source: Digilent Reference Manual)

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 4: 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.

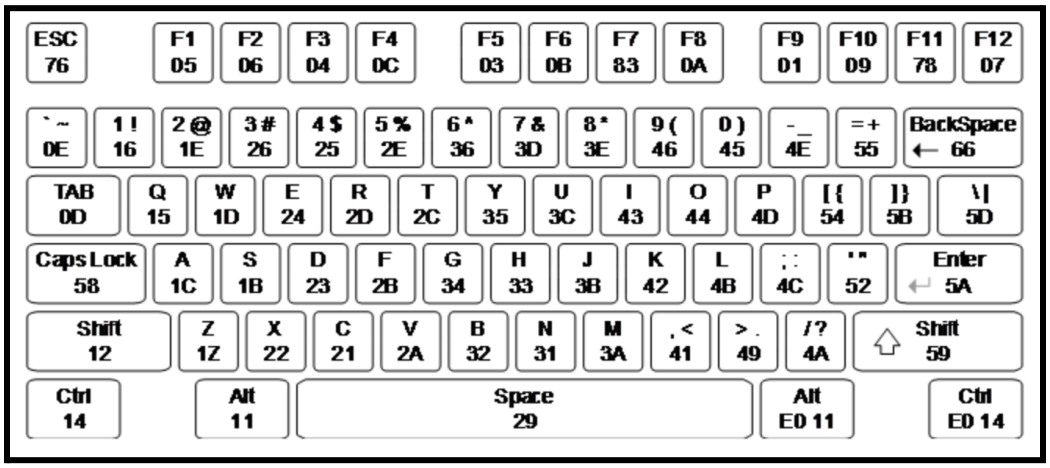


Figure 5: Key scancode mappings (Source: Digilent reference manual)

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.

3.1C Block RAM

To store the pixel information for each of the 12 characters, ‘0’ - ‘9’, ‘E’, and the blank space, some form of memory must be implemented. As the pixel data only needs to be read and not written anywhere, single-port read-only memory (ROM) can be used. Vivado has a built-in IP generator that can be used to autogenerate this ROM module using the Block Memory Generator.

The following information is known about the format of the numbers.coe file that contains the pixel information for each color value:

- There are 12 characters
- Each character is 8 pixels wide by 16 pixels tall
- Pixels are represented by 3 8-bit color values
- Every line in the file contains the color information for 1 pixel

With this information, it can be calculated that each character is represented by 8 * 16 = 128 pixels, and each pixel is 24 bits in size. Thus, the width of the ROM needs to be 24 bits, and the depth must be 128 pixels * 12 characters = 1536 bits. This means that the ROM will store a total of 24 * 1536 = 36864 bits.

Additionally, the enable port type will be set to “always enabled” to ensure that data can always be read, and the .coe file must be uploaded to the ROM from “other options.”

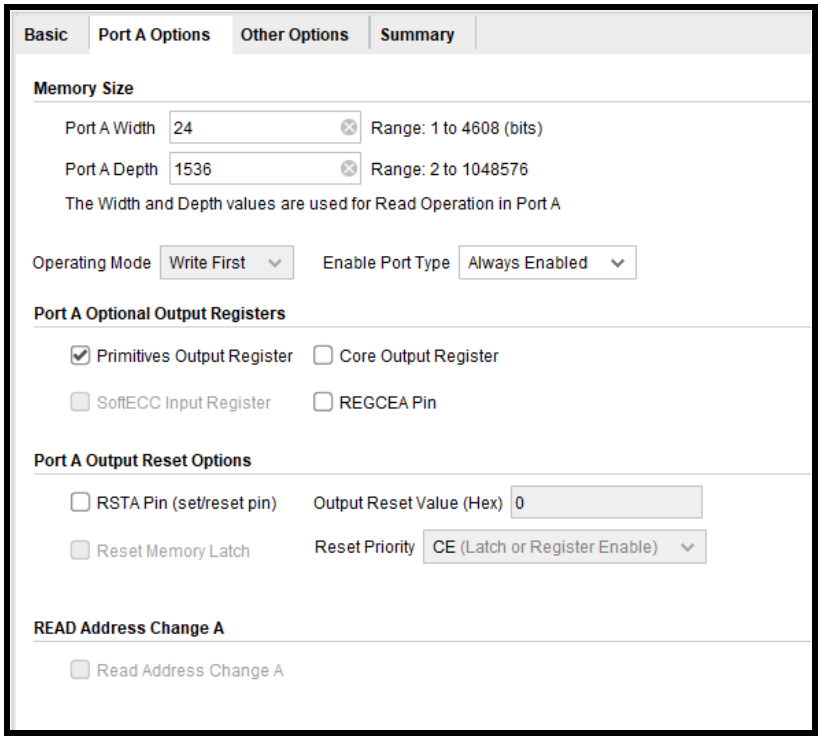


Figure 6: Block memory generator settings

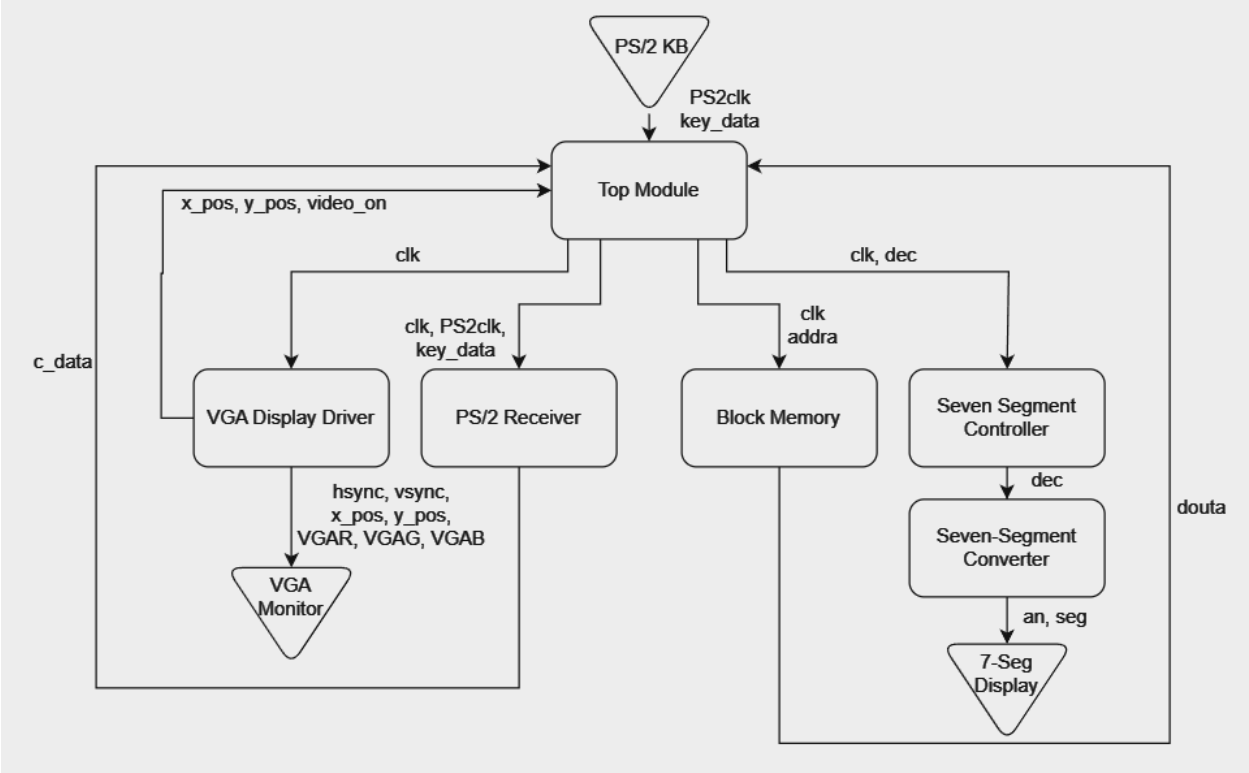


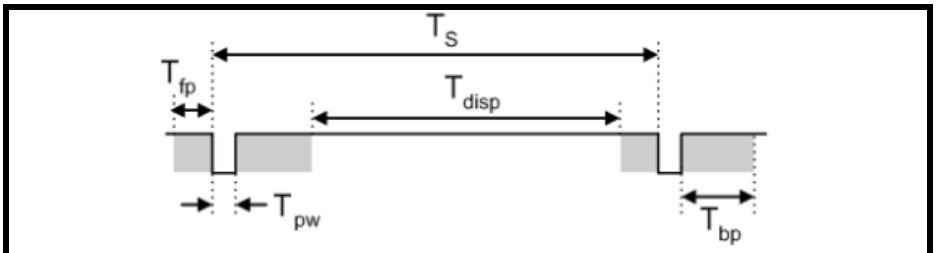
Figure 7: Block diagram of Verilog modules

3.2 Module Descriptions

Top module: Instantiates the VGA display controller, PS/2 receiver, block memory, 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, scancode 'E' to the BCD value 10, and return scancode to the BCD value 11.

VGA Display Driver: Handles the logic necessary to drive the VGA display.

- Divides 100MHz clock down to a 25MHz pixel clock, the frequency at which pixels are updated on the VGA monitor.
- Generates the hsync and vsync timing signals.
    - Vsync defines the rate at which the display is refreshed, or redrawn, 60 Hz for the monitor used in this experiment.
    - Hsync signals the number of lines to be redrawn at a given refresh frequency.
    - Timing information is taken from the following chart.



Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

Figure 8: Timing information for a VGA display

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.

Block memory: Given a memory address, retrieves the corresponding color information of an individual pixel in the character to be displayed. As VGA colors are only 4-bit, only the uppermost four bits of the 8-bit color from ROM are used.

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.

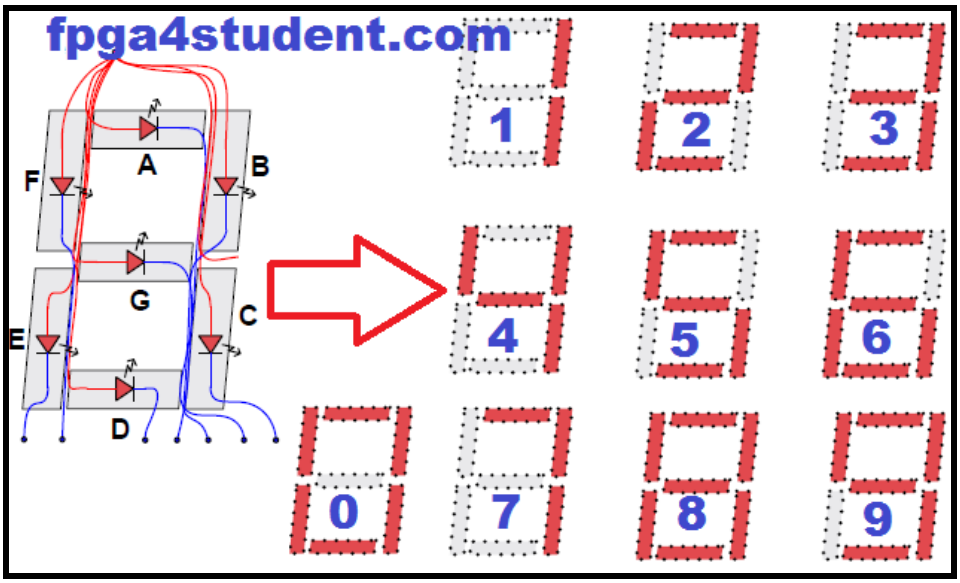


Figure 9: Seven-segment display diagram (source fpga4student.com)

4.0 Results

4.1 Verilog Modules

Top Module

```
// Instantiates all modules and handles displaying characters to the screen
module top(
	input clk, rst, PS2clk, key_data,
	output hsync, vsync,
	output reg [3:0] VGAR, VGAG, VGAB,
	output [3:0] an,
	output [6:0] seg
);

wire [23:0] douta;
reg[10:0] addra;
wire[10:0] x_pos, y_pos;
reg[3:0] dec;
wire[7:0] c_data;
wire video_on;
reg write_char;

// Instance of block ram to store the character data
character_block_mem inst (
	.douta(douta),
	.addra(addra),
	.clka(clk)
);
// Decodes the PS/2 signals
PS2_receiver receiver(
	.clk(clk),
	.PS2clk(PS2clk),
	.key_data(key_data),
	.c_data(c_data)
);

// Generates the signals to drive the VGA monitor
VGA_display_driver display_driver(
	.clk(clk),
	.hsync(hsync),
	.vsync(vsync),
```

```
        .p_tick(),
        .x_pos(x_pos),
        .y_pos(y_pos),
        .data_ena(video_on)
);

// Generates the signals to drive the seven-segment display
seven_segment_controller seg_ctrl(
        .clk(clk),
        .digit(dec),
        .an(an),
        .seg(seg)
);

// Only display character and read address in the top 8 x 16 pix corner of the screen
always @ (posedge clk) begin
        if (x_pos > 0 && x_pos < 8 && y_pos > 0 && y_pos < 16) begin
                addra <= (dec * 128) + (x_pos) + (y_pos) * 8;
                write_char <= 1'b1;
        end
        else
                write_char <= 1'b0;
        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;
                8'h5A:
                        dec = 4'b1011;
                default:
                        dec = 4'b1010;
        endcase
end

always @ (posedge clk) begin
        // On reset, if the video is off, or if a character is not being written,
        screen is black
        if (rst || ~video_on || ~write_char) begin
                VGAR = 0;
                VGAG = 0;
                VGAB = 0;
        end
        // Else display color data from block RAM
        else begin
                // Taking the 4 MSB of douta; can also do 4 LSB
                VGAR = douta[23:20];
                VGAG = douta[15:12];
                VGAB = douta[7:4];
                end
        end
endmodule
```

VGA Display Driver Module

```
module VGA_display_driver(
        input clk, rst,
        output hsync, vsync, data_ena, p_tick,
        output [9:0] x_pos, y_pos
);

// Horizontal Dimensions
localparam HORIZONTAL_DISPLAY_PIXELS = 640; // Horizontal display area
localparam HSYNC_PULSE_WIDTH = 96; // Horizontal pulse width for retrace
localparam HSYNC_FRONT_PORCH = 16; // Horizontal left border
localparam HSYNC_BACK_PORCH = 48; // Horizontal right border
localparam HORIZONTAL_PIXELS = HORIZONTAL_DISPLAY_PIXELS + HSYNC_PULSE_WIDTH +
HSYNC_BACK_PORCH + HSYNC_FRONT_PORCH - 1; // = 800 pixels

// Horizontal Timings
```

```
localparam H_RETRACE_START = HORIZONTAL_DISPLAY_PIXELS + HSYNC_FRONT_PORCH - 1;
localparam H_RETRACE_END = H_RETRACE_START + HSYNC_PULSE_WIDTH;

// Vertical Dimensions
localparam VERTICAL_DISPLAY_PIXELS = 480; // Vertical display area
localparam VSYNC_PULSE_WIDTH = 2; // Vertical pulse width for retrace
localparam VSYNC_FRONT_PORCH = 10; // Vertical top border
localparam VSYNC_BACK_PORCH = 29; // Vertical bottom border
localparam VERTICAL_PIXELS = VERTICAL_DISPLAY_PIXELS + VSYNC_PULSE_WIDTH +
VSYNC_FRONT_PORCH + VSYNC_BACK_PORCH - 1; // = 521 pixels

// Vertical Timings
localparam V_RETRACE_START = VERTICAL_DISPLAY_PIXELS + VSYNC_FRONT_PORCH - 1;
localparam V_RETRACE_END = V_RETRACE_START + VSYNC_PULSE_WIDTH;

reg [1:0] pixel_reg;
wire [1:0] pixel_next;
wire pixel_tick;

always @ (posedge clk) begin
	if (rst)
		pixel_reg <= 0;
	else
		pixel_reg <= pixel_next;
end

assign pixel_next = pixel_reg + 1;
assign pixel_tick = (pixel_reg == 0); // Divide 100MHz clock 4x down to 25MHz

reg vsync_reg, hsync_reg;
wire vsync_next, hsync_next;
reg[9:0] row_pix, next_row_pix, col_pix, next_col_pix;

// Sequential Logic
always @ (posedge clk) begin
	if (rst) begin
		row_pix <= 0;
		col_pix <= 0;
		vsync_reg <= 0;
		hsync_reg <= 0;
	end
	else begin
```

```
                row_pix <= next_row_pix;
                col_pix <= next_col_pix;
                vsync_reg <= vsync_next;
                hsync_reg <= hsync_next;
        end
end

// Combinational Logic
always @ (*) begin
        if (pixel_tick) begin
                if (row_pix == HORIZONTAL_PIXELS)
                        next_row_pix = 0;
                else
                        next_row_pix = row_pix + 1;
        end
        else
                next_row_pix = row_pix;
        if (pixel_tick && row_pix == HORIZONTAL_PIXELS) begin
                if (col_pix == VERTICAL_PIXELS)
                        next_col_pix = 0;
                else
                        next_col_pix = col_pix + 1;
        end
        else
                next_col_pix = col_pix;
end

assign hsync_next = (row_pix >= H_RETRACE_START && row_pix <= H_RETRACE_END);
assign vsync_next = (col_pix >= V_RETRACE_START && col_pix <= V_RETRACE_END);
assign data_ena = (row_pix < HORIZONTAL_DISPLAY_PIXELS && col_pix <
VERTICAL_DISPLAY_PIXELS);

assign hsync = hsync_reg;
assign vsync = vsync_reg;
assign x_pos = row_pix;
assign y_pos = col_pix;
assign p_tick = pixel_tick;

Endmodule
```

PS/2 Receiver Module

```
// 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 10 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
```

Block Memory Generator

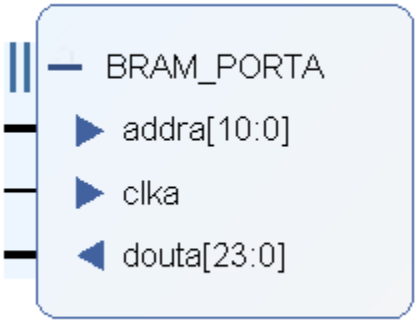


Figure 10: Symbolic representation of block memory

Seven-Segment Controller Module

```
// 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
```

Seven-Segment Converter Module

```
// Converts from BCD representation -> LED config. 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 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]}]

# VGA Connector
set_property PACKAGE_PIN G19 [get_ports {VGAR[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAR[0]}]
set_property PACKAGE_PIN H19 [get_ports {VGAR[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAR[1]}]
set_property PACKAGE_PIN J19 [get_ports {VGAR[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAR[2]}]
set_property PACKAGE_PIN N19 [get_ports {VGAR[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAR[3]}]
set_property PACKAGE_PIN N18 [get_ports {VGAB[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAB[0]}]
set_property PACKAGE_PIN L18 [get_ports {VGAB[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAB[1]}]
set_property PACKAGE_PIN K18 [get_ports {VGAB[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAB[2]}]
set_property PACKAGE_PIN J18 [get_ports {VGAB[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAB[3]}]
set_property PACKAGE_PIN J17 [get_ports {VGAG[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAG[0]}]
set_property PACKAGE_PIN H17 [get_ports {VGAG[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAG[1]}]
set_property PACKAGE_PIN G17 [get_ports {VGAG[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAG[2]}]
set_property PACKAGE_PIN D17 [get_ports {VGAG[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {VGAG[3]}]
set_property PACKAGE_PIN P19 [get_ports hsync]
set_property IOSTANDARD LVCMOS33 [get_ports hsync]
set_property PACKAGE_PIN R19 [get_ports vsync]
set_property IOSTANDARD LVCMOS33 [get_ports vsync]

#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

Link to video of experimental implementation

Figures 11-12: 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 top left of the VGA monitor. When a non-number row key was pressed, the letter ‘E’ was displayed to the monitor, and the character was erased if the enter key was pressed. There were no noticeable bugs or glitches with this implementation.

This experiment integrated many of the modules that have been developed over the past few experiments. It incorporated the PS/2 receiver module to decode scancodes and determine which character to display to the screen, the VGA display driver module to handle displaying characters to the monitor, and as a bonus, the seven-segment display controller module to display characters to the FPGA’s seven segment display. These modules were able to be seamlessly combined, along with the block memory, to implement this experiment with very few issues.

The main challenge with this experiment was becoming familiar with the Vivado IP generator and determining the necessary settings to configure the single-port ROM. Additionally, determining the conditions under which to read a certain address from memory based on x_pos

and y_pos required some critical thinking about how pixels were displayed on the monitor. However, most of the heavy lifting for the VGA display driver and PS/2 receiver was finished in previous experiments, so very little debugging was required.

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 display the character corresponding to each scancode to a monitor. It demonstrated many core aspects of working with digital hardware, including interfacing with software through a peripheral input such as a keyboard, configuring memory to store different information, and displaying human-readable information to a monitor. All of these aspects, input, memory, and output, are fundamental to understanding and working with digital systems.

This experiment opens the door for more advanced FPGA projects in the future. With the ability to now read in pixel information from external files and store that information in memory, much more complex and meaningful information can be outputted to a monitor. With knowledge of the PS/2 communication protocol and how to decode scancodes, almost any input from the user can now be captured, stored in memory, and displayed. Future projects, such as the final course project, will be able to leverage this knowledge and understanding to implement more complex and meaningful features. In conclusion, this project served to bridge together all past experiments and provide a toolkit for building more advanced FGPA projects in the future.

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/.