Lab 5: Image Filtering

Cory Brynds

cory.brynds@ucf.edu

EEL5722C: FPGA Design

Section 0012

Due: 16 November 2023

Submitted: 11 November 2023

1.0 Objectives

The objective of this experiment is to load a greyscale image into BRAM and implement low-pass filtering on the image each time the ‘0’ key on a USB keyboard is pressed.

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.

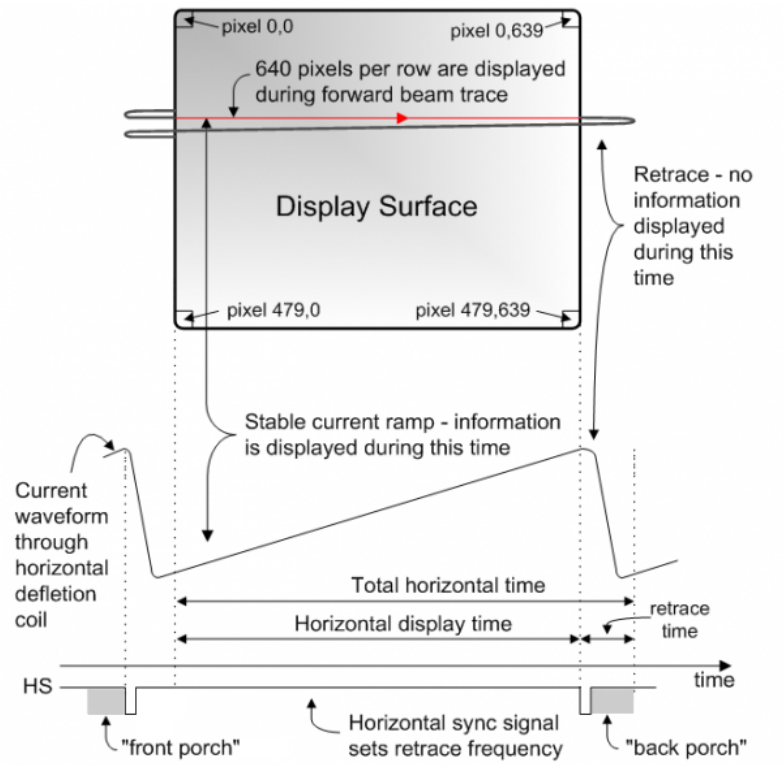


Figure 1: VGA display dimensions and timing (Source: Digilent)

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.

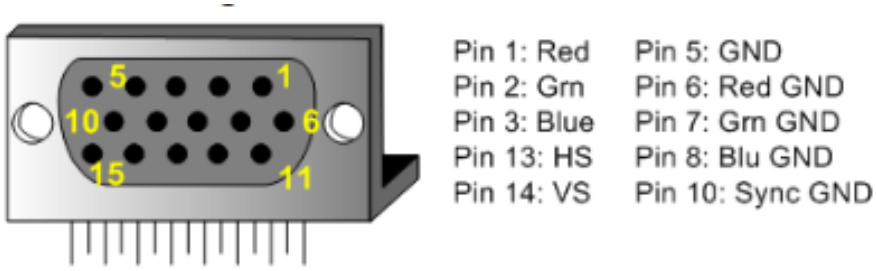


Figure 2: 15-pin VGA port (Source: Digilent)

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.

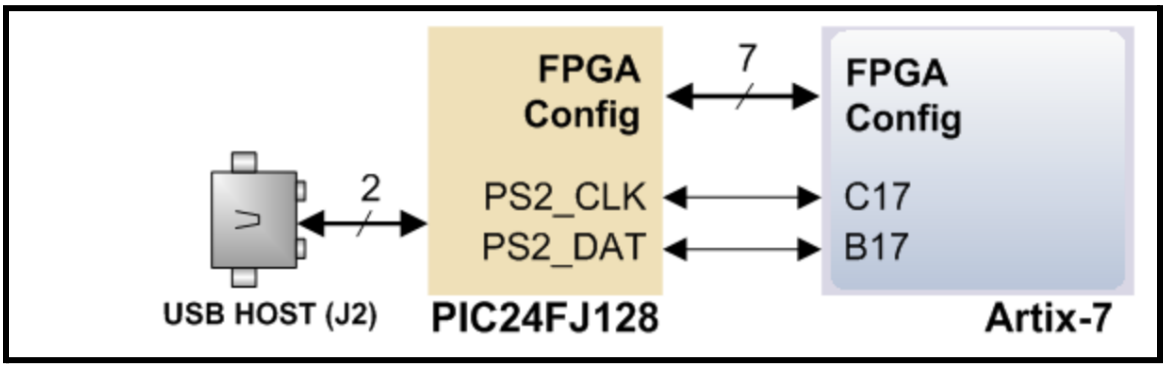
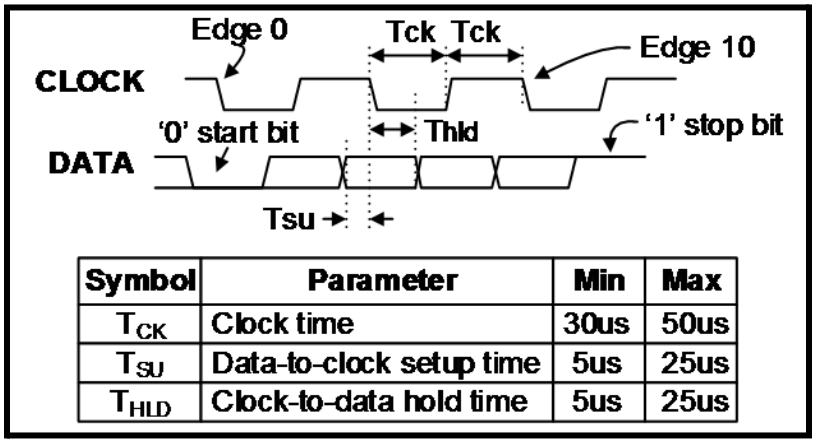


Figure 3: Diagram of USB connection to PS/2 interface (Source: Digilent Reference Manual)

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.

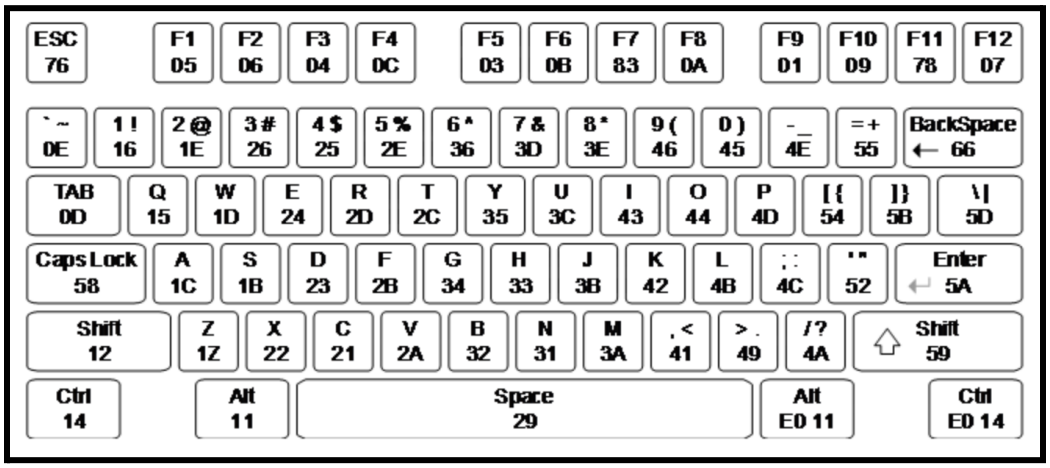
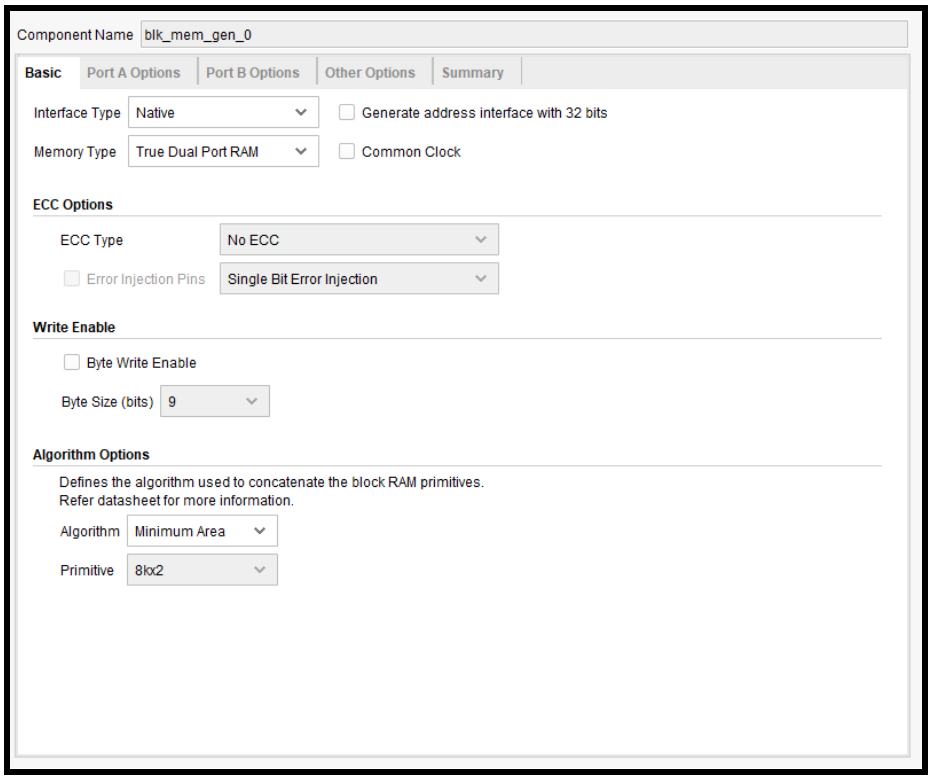
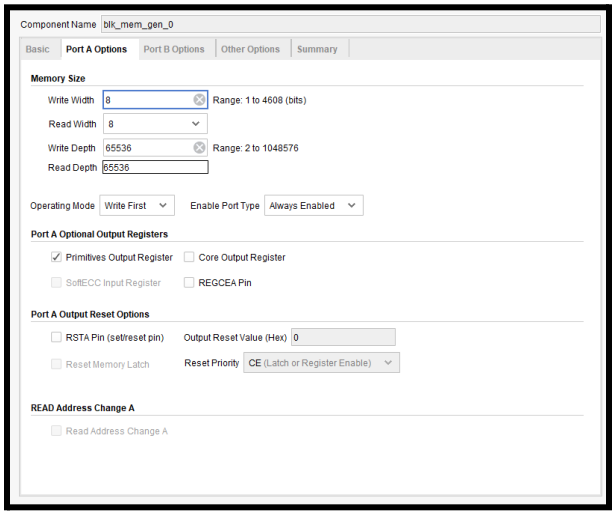


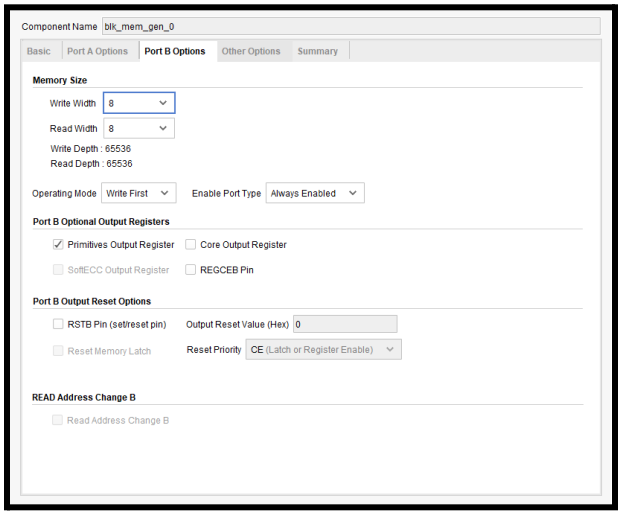
Figure 5: Key scancode mappings (Source: Digilent reference manual)

3.1C Block RAM

To store the pixel data for the image and store pixel values as the filter is applied, two instances of block memory must be instantiated. Both will need to be read from and written to, so True Dual-Port RAM must be generated using the Vivado block memory IP generator. Ports A and B will each have a read-and-write width of 8 to denote the 8-bit color values for each pixel, and a read-and-write depth of 65536 to store the 256 x 256 pixels in the image. The operating mode of both ports is write first, and the enable port is set to always enabled. Finally, the image to be filtered is loaded into the memory in the form of a .coe file.







Figures 6-8: Block memory generator settings

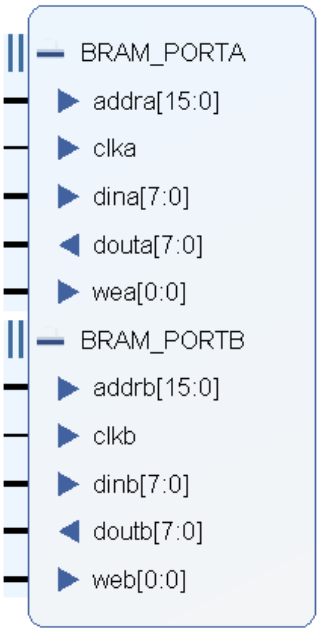


Figure 9: True Dual Port RAM module

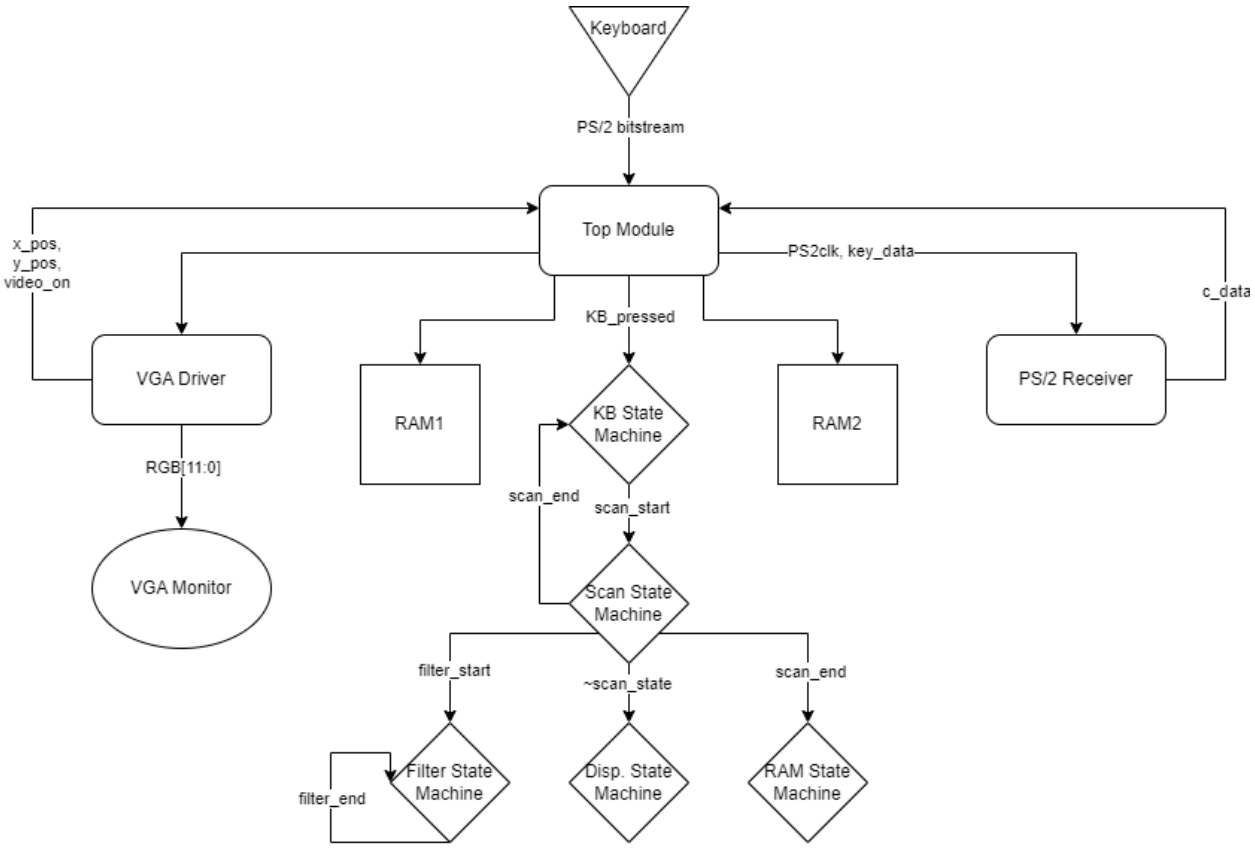


Figure 10: Block diagram of Verilog modules

3.2 Module Descriptions

Top module: Instantiates the VGA display controller, PS/2 receiver, and block memory modules. Implements five state machines to handle image filtering:

1. **Keyboard state machine**: begins change of state when the '0' key from the keyboard is pressed. Increments on every clock cycle until the value 1000000 is reached, a large enough value to ensure that all other state machines have finished (i.e. the image is completely filtered)
2. **Scan state machine**: begins incrementing when scan_start is asserted. Changes state until all pixels of the image have been scanned and filtered.
3. **Filter state machine**: begins incrementing when filter_start signal is asserted. On each state, stores the pixel information at a given address from RAM1 into the pixel register. In state 11, applies filter to the pixel. Then, writes the pixel to RAM2 in state 13. Ends after 15th state and resumes when the next pixel is ready to be filtered.
4. **Display state machine**: When scan_state is asserted low, display filtered image to the screen. In display_state 0, set the address of the pixel to be displayed from RAM1. In display_state 1, set the VGA color values to the 4 MSB of the pixel located in RAM1. If the pixel position is currently outside of the 256x256 pixel frame, display black to the monitor.
5. **RAM state machine**: Used to copy the filtered image stored in RAM2 to RAM1. Begin incrementing when scan_end is asserted. In RAM_state 0, set the address of both RAMs to 0, and write pixel data from RAM2 to RAM1 until RAM_state 65538 is reached, at which point RAM_state is set back to 0 to await the next scan_end assertion.

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.

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. New logic is added to ensure that when

a key is pressed, it is only asserted once, instead of being continually asserted until another key is pressed as in previous experiments.

Block memory: Two instances of True Dual-Port RAM are used as frame buffers. RAM1 stores the initial image, and the filtered image is written to RAM2 during the filter state machine. In the RAM state machine, the contents of RAM2 are written back to RAM1 to be displayed on the screen.

4.0 Results

4.1 Verilog Modules

Top Module

```
// Instantiates all of the necessary modules, displays the image on the
monitor, and handles the filtering action
module top(
    input clk, rst, PS2clk, key_data,
    output hsync, vsync,
    output reg [3:0] VGAR, VGAG, VGAB
);

wire [23:0] douta;
wire[10:0] x_pos, y_pos;
reg[18:0] kb_state;
reg[17:0] scan_state;
reg[3:0] filter_state, dec;
reg[16:0] RAM_state;
reg display_state, write_char, web2, web1;
wire video_on, kb_pressed, scan_start, scan_end, filter_end;
reg[15:0] addra1, addrb1, addra2, addrb2;
reg[7:0] dina1, dina2, dinb1, dinb2;
wire[7:0] c_data, douta1, douta2, doutb1, doutb2;
reg[7:0] pixels[8:0]; //2-dimensional reg of width 8 and height 9
reg[9:0] filtered_pixel;

// Instance of block ram to store the character data
blk_mem_gen_0 RAM1 (
    .douta(douta1),
    .doutb(doutb1),
```

```
    .addra(addra1),
    .addrb(addrb1),
    .dina(dina1),
    .dinb(dinb1),
    .web(web1),
    .clka(clk),
    .clkb(clk)
    );

// Instance of block ram to store the character data
blk_mem_gen_0 RAM2 (
    .douta(douta2),
    .doutb(doutb2),
    .addra(addra2),
    .addrb(addrb2),
    .dina(dina2),
    .dinb(dinb2),
    .web(web2),
    .clka(clk),
    .clkb(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)
    );
```

```
// Keyboard state machine
always @ (posedge clk) begin
    case(kb_state)
        0: begin
            if (kb_pressed)
                kb_state <= kb_state + 1;
        end
        1: kb_state <= kb_state + 1;
        2: begin
            if (scan_end)
                kb_state <= kb_state + 1;
        end
        1000000: kb_state <= 0; //Needs to be a large number to allow the
            RAM state machine to finish its operation
        default: kb_state <= kb_state + 1;
    endcase
end

// Scan State machine
always @ (posedge clk) begin
// Take the last two bits (4 states/pixel)
    case (scan_state[1:0])
        0: begin
            if (scan_start || scan_state > 3)
                scan_state <= scan_state + 1;
        end
        1: scan_state <= scan_state + 1;
        2: begin
            if (filter_end)
                scan_state <= scan_state + 1;
        end
        3: scan_state <= scan_state + 1;
        default: scan_state <= scan_state + 1;
    endcase
end

// Filter state machine and display state machine
always @ (posedge clk) begin
    case (filter_state)
```

```
        0: begin
            if(filter_start)
                filter_state <= filter_state + 1;
        end
        1: begin
            addra1 <= scan_state[17:2] - 257;
            filter_state <= filter_state + 1;
        end
        2: begin
            addra1 <= addra1 + 1;
            pixels[0] <= douta1;
            filter_state <= filter_state + 1;
        end
        3: begin
            addra1 <= addra1 + 1;
            pixels[1] <= douta1;
            filter_state <= filter_state + 1;
        end
        4: begin
            addra1 <= addra1 + 254; // 3rd pixel in the filter row, move
           to the next row
            pixels[2] <= douta1;
            filter_state <= filter_state + 1;
        end
        5: begin
            addra1 <= addra1 + 1;
            pixels[3] <= douta1;
            filter_state <= filter_state + 1;
        end
        6: begin
            addra1 <= addra1 + 1;
            pixels[4] <= douta1;
            filter_state <= filter_state + 1;
        end
        7: begin
            addra1 <= addra1 + 254; // 3rd pixel in the filter row, move
           to the next row
            pixels[5] <= douta1;
            filter_state <= filter_state + 1;
        end
```

```
        8: begin
            addra1 <= addra1 + 1;
            pixels[6] <= douta1;
            filter_state <= filter_state + 1;
        end
        9: begin
            addra1 <= addra1 + 1;
            pixels[7] <= douta1;
            filter_state <= filter_state + 1;
        end
        10: begin
            addra1 <= addra1 - 257; // Last filtered pixel, reset to top
           left
            pixels[8] <= douta1;
            filter_state <= filter_state + 1;
        end
        11: begin
            // Shift operation instead of division operation
            // 7/2^9 ~= 0.109 -> 1/9 = 0.111...
            filtered_pixel <= ((pixels[0] + pixels[1] + pixels[2] +
             pixels[3] + pixels[4] + pixels[5] + pixels[6] + pixels[7] +
             pixels[8]) * 7) >> 6;
            filter_state <= filter_state + 1;
        end
        12: begin
            addrb2 <= addra1; // addra1 indicates original pixel (top left
           of filter)
            filter_state <= filter_state + 1;
        end
        13: begin
            dinb2 <= filtered_pixel[7:0];
            web2 <= 1;
            filter_state <= filter_state + 1;
        end
        14: begin
            web2 <= 0;
            filter_state <= filter_state + 1;
        end
        15: begin
            filter_state <= 0;
```

```
        end
        default: begin
            filter_state <= filter_state + 1;
        end
    endcase

    // Are either starting scanning or have finished scanning
    if (scan_state == 0) begin
        case(display_state)
            0: begin
                addra1 <= x_pos * 256 + y_pos;
                display_state <= display_state + 1;
            end
            1: begin
                // If within bounds of the image (256 x 256), display the
                 image pixels
                if (x_pos > 0 && x_pos < 256 && y_pos > 0 && y_pos < 256)
                 begin
                    VGAR <= douta1[7:4];
                    VGAG <= douta1[7:4];
                    VGAB <= douta1[7:4];
                end
                // If outside of region, display black
                else begin
                    VGAR <= 0;
                    VGAG <= 0;
                    VGAB <= 0;
                end
                display_state <= display_state + 1;
            end
        endcase
    end
end

// RAM state machine
always @ (posedge clk) begin
    case (RAM_state)
        0: begin
            // On end of scan, begin incrementing RAM_state and initialize
           both addresses to 0
```

```
            if (scan_end) begin
                RAM_state <= RAM_state + 1;
                addrb1 <= 0;
                addra2 <= 0;
            end
        end
        // Write filtered image from RAM2 to RAM1
        1: begin
            web1 <= 1;
            dinb1 <= douta2;
            RAM_state <= RAM_state + 1;
        end
        // Repeat operation for 65536 cycles
        65538: begin
            web1 <= 0;
            RAM_state <= 0;
        end
        // Default state is increment addresses and copy from RAM2 to RAM1
        default: begin
            addrb1 <= addrb1 + 1;
            addra2 <= addra2 + 1;
            dinb1 <= douta2;
            RAM_state <= RAM_state + 1;
        end
    endcase
end

// Assign statements
assign kb_pressed = (c_data == 8'h45) ? 1:0;
assign scan_start = (kb_state == 1) ? 1:0;
assign filter_start = (scan_state[1:0] == 1) ? 1:0;
assign filter_end = (filter_state == 15) ? 1:0;
assign scan_end = (scan_state == 262143) ? 1:0;

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 [7:0] current_c_data;
reg [3:0] count;
reg [1:0] signal_high_counter;
wire parity_check;
reg [7:0] c_data_buffer;

// 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_buffer <= store[8:1];
        if (c_data_buffer == store[8:1])
            c_data <= 0;
        else
            c_data <= store[8:1];
    end
end

endmodule
```

Block Memory Generator

```
module blk_mem_gen_0(clka, wea, addra, dina, douta, clkb, web, addrb,
dinb, doutb)
  input clka;
  input [0:0]wea;
  input [15:0]addra;
  input [7:0]dina;
  output [7:0]douta;
  input clkb;
  input [0:0]web;
  input [15:0]addrb;
```

```
  input [7:0]dinb;
  output [7:0]doutb;
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]
```

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

4.3 Video of Experimental Implementation

Link to video of experimental implementation

Figure 11: Screen capture from the video of the experimental implementation

4.4 Explanation of Results

The results of this experiment were as expected. Once programmed, the BASYS 3 would display the correct 256x256 pixel image on the VGA monitor in the upper lefthand corner. When the '0' number key was pressed, the proper filtering action would occur, with each successive

key press filtering the image more and more until eventually it was reduced to a black screen. When a key other than ‘0’ was pressed, no filtering action occurred.

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 the ‘0’ scancode and determine which character to display on the screen and the VGA display driver module to handle displaying the image to the monitor. These modules could be seamlessly combined, along with the block memory, to implement this experiment with very few issues.

One challenge associated with this project involved the PS/2 receiver module. In past experiments, when a key on the keyboard was pressed and the scancode was decoded, that same scancode would be asserted until another key was pressed. For displaying characters to the seven-segment display or the monitor, this was sufficient, but this was not the desired outcome for the image filter. What was needed was a way to press a key and have it be asserted only once, to start the filtering action. So additional logic was implemented in the form of a scancode buffer. When a new scancode was detected, it would be stored in c_data and the buffer, and if the buffer was equal to the scancode detected on the next cycle, c_data would be set to 0, ensuring that each scancode was only asserted once.

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 an image from block RAM to the screen of 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, displaying visual information to a monitor, and implementing various state machines to handle sequential logic. All of these aspects 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 instantiate RAM with both read and write functionality, 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 final experiment before the final course 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/.