Edge Detection in Verilog

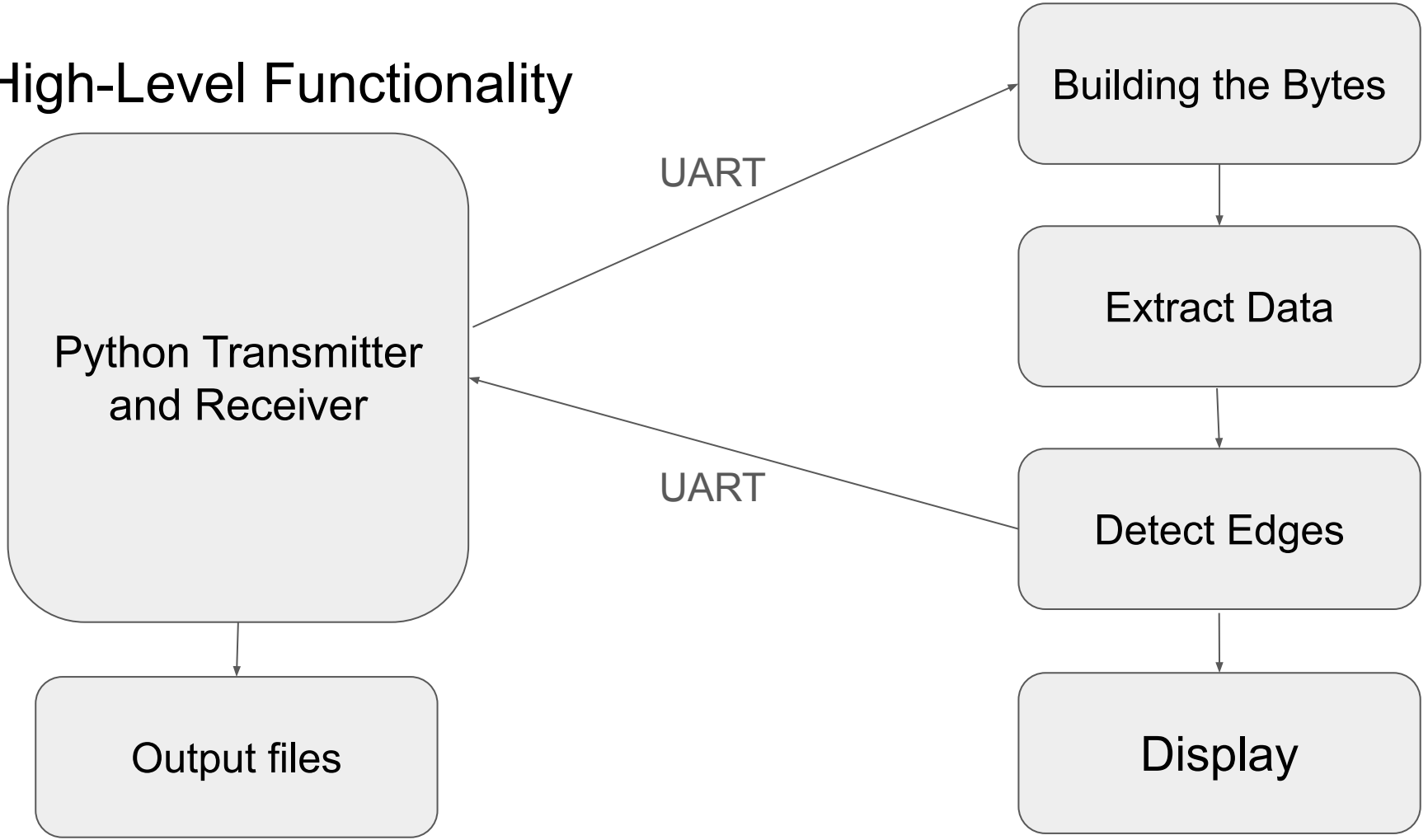
Thomas Kelly, Hamed Rastaghi, Xiteng Yao, Shining Yang

Table of Contents

Goals

- Perform edge detection in Verilog
- Send an image through the UART port byte-by-byte
- The image is processed in a streamlined fashioned
- The image is not rebuilt until the final output
- Save on memory and speed compared to alternative implementations

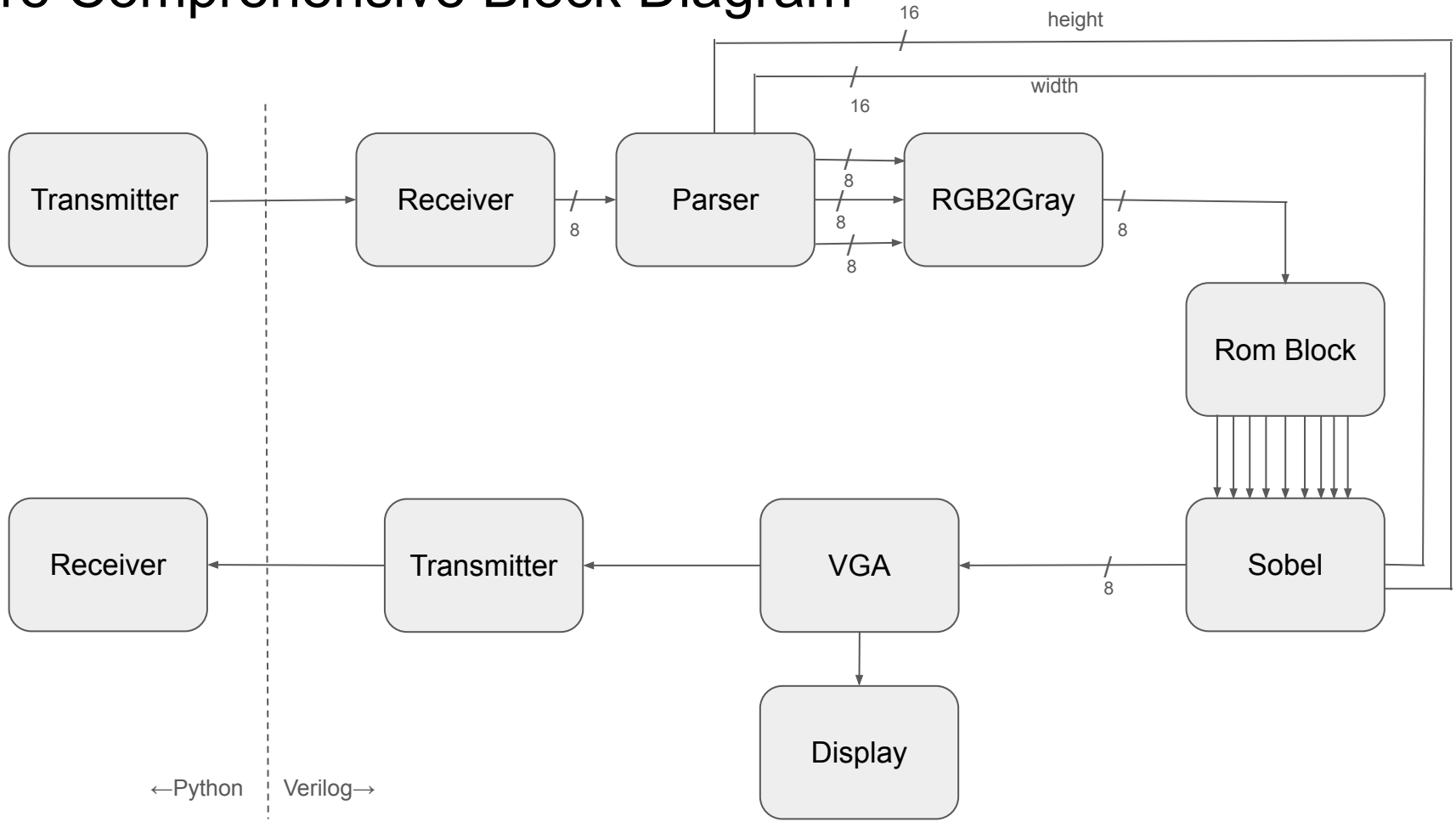
High-Level Functionality



Specification of the Edge Detector

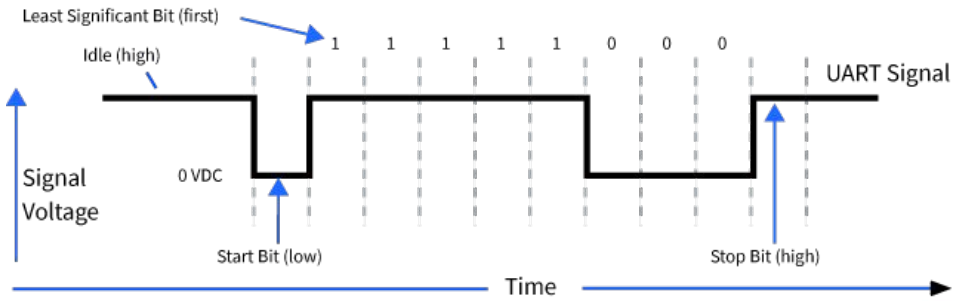
- Performance Requirement
  - Sobel edge detector module shall process 1 byte per cycle
  - UART modules shall send/receive 1 bit per 10 cycle, can be shorter if HW allows
- Image Size
  - Variable sized image accepted
  - Image size up to 1000 X 1000, can be increased if HW allow
- Image Type
  - Need to be able to accept RGB images
  - Possibly support continuous processing of images
- Tradeoff
  - The higher the resolution, the more memory will be used, and more processing time will be needed

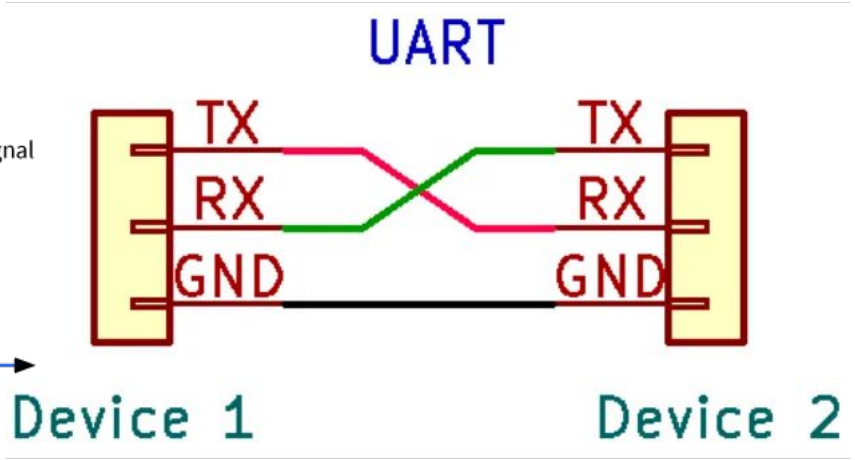
More Comprehensive Block Diagram



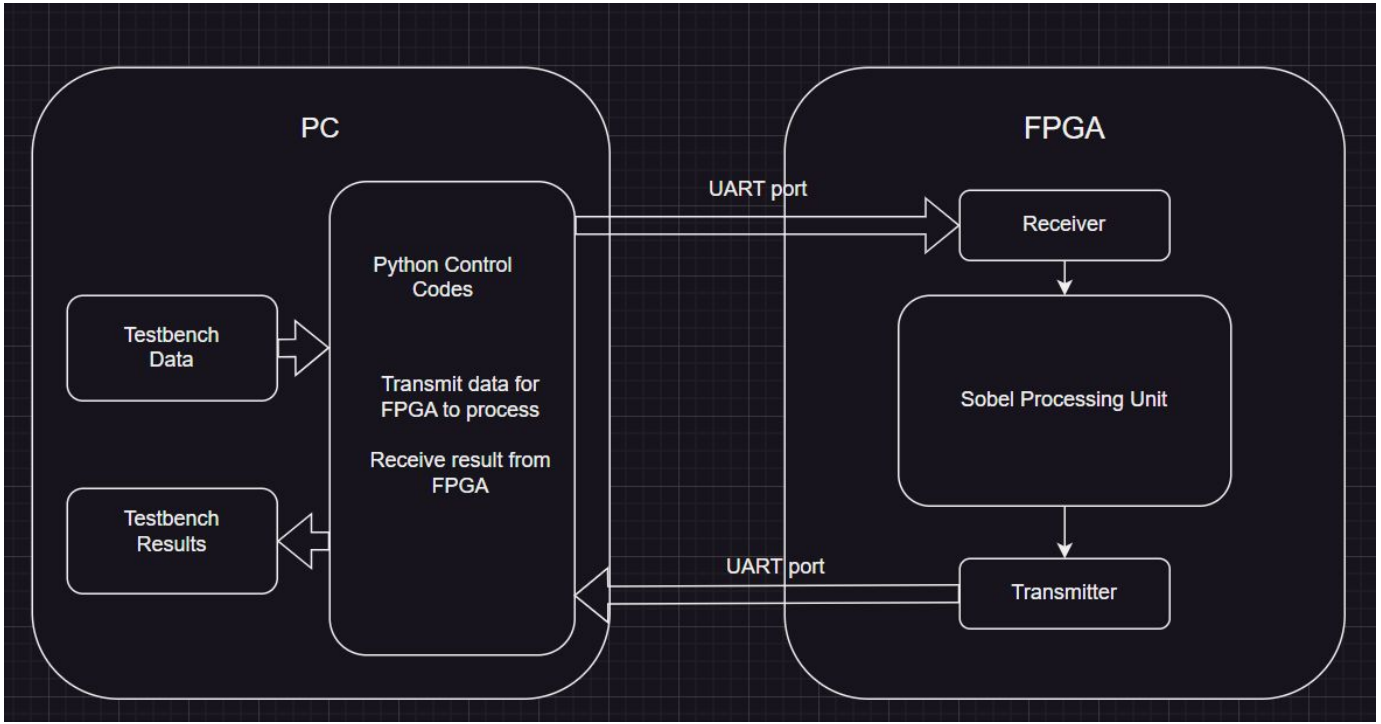
UART Basics - Xiteng

- Universal Asynchronous Receiver / Transmitter
- A serial connection between two devices to transfer data
- TX for transmitting, RX for receiving
- Work together with preset baud rates and frequencies
- UART signals have a special pattern





UART In Our Project - Xiteng



UART on the PC side - Xiteng

- We wrote several Python scripts to read the images from a folder, and send it to the UART port one by one
- Each file is processed independently
- The images can be processed as grayscale or RGB
- Also includes necessary metadata for the Sobel algorithm to use
- Enables frame by frame processing for videos

UART on the FPGA side

Receiver (Tom)

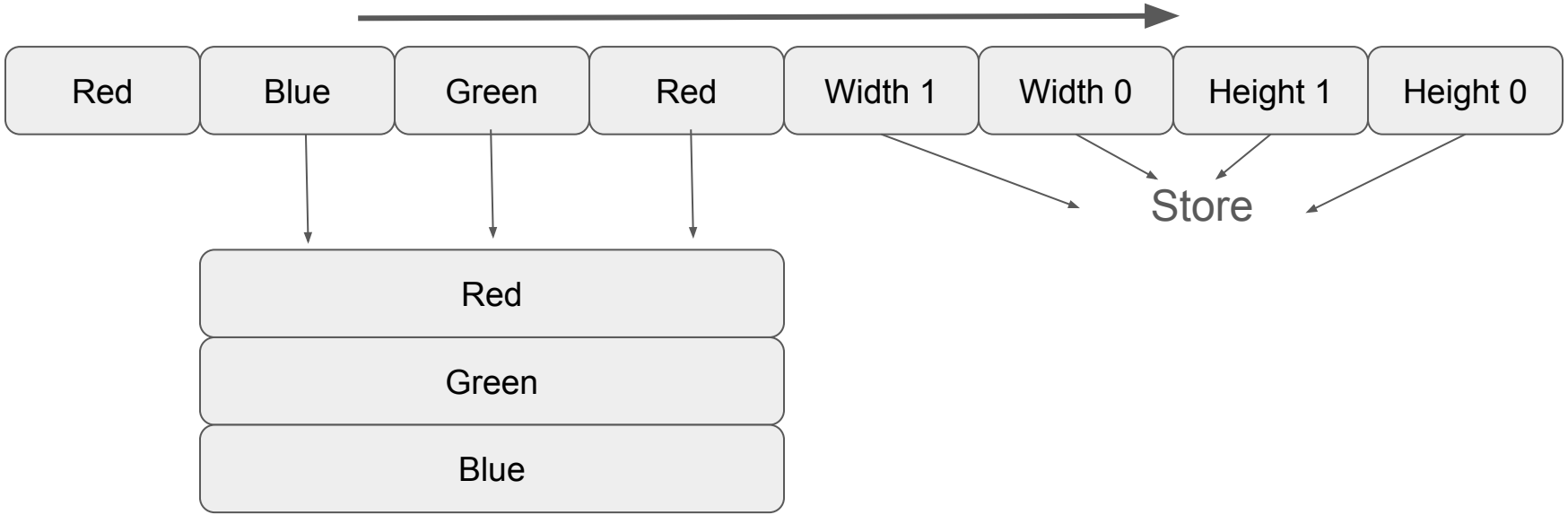
- Idle state: waiting for the logic-low start bit
- Sampling state: sample each of the bits in the middle
- Cleanup state: once the logic-high end bit is received, prepare for transmission of next byte
- Clock frequency is much higher than the baud rate, and their relationship is key

Transmitter (Xiteng)

- 5-State state machine
- Detects when one byte of data is processed
- Send it to the PC when data available
- Speed is limited because only one bit can be transmitted at a time
- Already tested on simulation and lab FPGAs

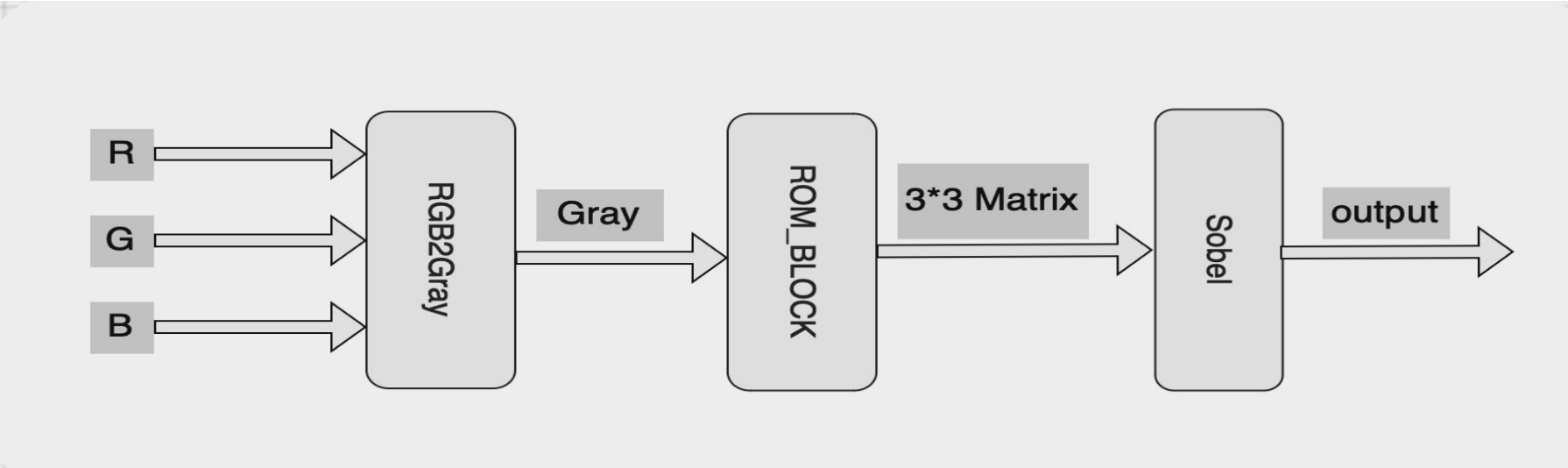
Parser (Tom)

- For a variable height and width, Sobel must learn these values before performing any processing
- The RGB2Gray module must receive all 3 values simultaneously, even though they are transmitted sequentially through UART

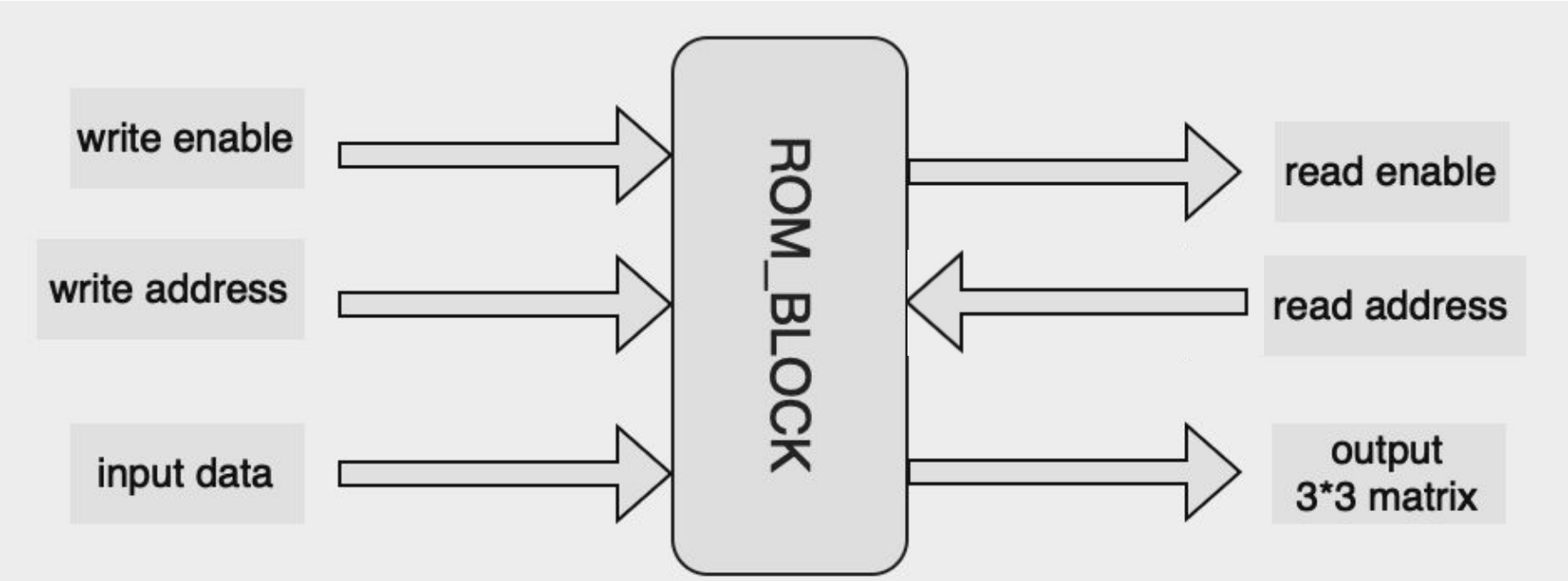


Algorithm(Hamed)

- RGB2gray module is streamlined
- There is a Rom Block between RGB2gray and Sobel



Rom Block(Hamed)



Sobel(Hamed)

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Here **A** is the source image.(a text file with all pixel values in it)

Gx: Horizontal derivative approximations

Gy: Vertical derivative approximations

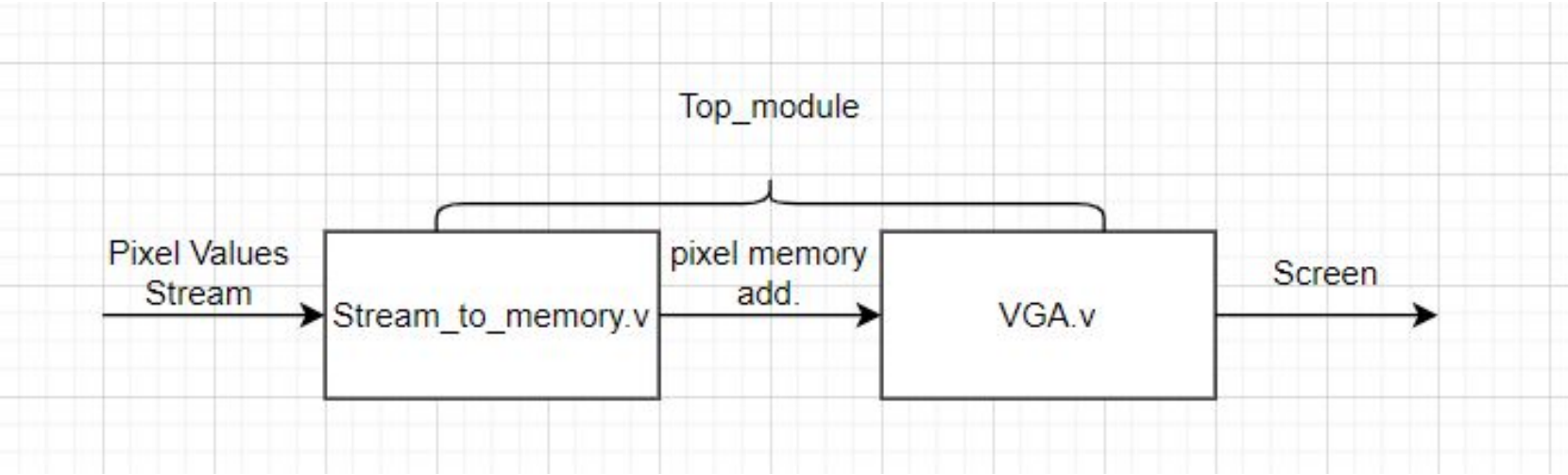
G: Processed image

Sobel(Hamed)

```
always @(posedge clk) begin
    if(!rstn) begin
        W_counter <=0;
        H_counter <= 0;
        ready  <= 0;
        final <= 0;
    end


    else begin

        if(start & !ready) begin
            data_out <= Gx + Gy;

            if(W_counter != W-1-2) begin
                W_counter <= W_counter + 1;
            end

            else begin
                W_counter <= 0;
                H_counter <= H_counter + 1;
            end


            if(W_counter == W-1-2    &&   H_counter == H-1-2) begin
                ready <= 1;
            end

        end


    end
end



always @(posedge clk) begin
    Gx <=    -data0 + data6          +
             -2*data1 + 2*data7  +
             -data2 + data8     ;

    Gy <=   -data0 - 2*data3 - data6      +
             data2 + 2*data5 + data8  ;
end


endmodule
```

Memory-VGA-Screen

- We firstly store the pixel stream in a memory block, then output the pixel values to VGA.v
  - Buffering and Synchronization
  - Data Manipulation
  - Error Handling and Test



Memory-VGA-Screen

- Stream-to-memory module
  - Input wire [x:0] address, x depends on size of the image
  - (We assign the address values in the top module)
  - Input [3:0] data_in : This is the stream of pixels
  - Output [3:0] data_out [y:z] , y,z depend on the size of the image
- VGA module
  - Input [3:0] data_in [y:z]
  - output reg [3:0] red
  - output reg [3:0] green
  - output reg [3:0] blue
  - output reg hsync, vsync

VGA Module(For 640*480 Resolution)

- HSYNC VSYNC: signals used for synchronizing the display of image data on a monitor.
- HSYNC Functioning: When a line of pixels (a row) is finished being displayed, the HSYNC signal tells the display to move the beam back to the left side of the screen to start displaying the next line.
- To let the image shown on screen is grayscale, we need to assign the grayscale pixel value (0-255) to all Red, Green and Blue ports.

VGA Module

```
assign HSYNC = (h_counter < (active_h_pixels + front_porch_h_pixels)) || (h_counter >= (active_h_pixels + front_porch_h_pixels + sync_pulse_h_pixels));
```

```
assign VSYNC = (v_counter < (active_v_lines + front_porch_v_lines)) || (v_counter >= (active_v_lines + front_porch_v_lines + sync_pulse_v_lines));
```

For different resolution, we need to make a adjustment

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Vertical timing (frame)

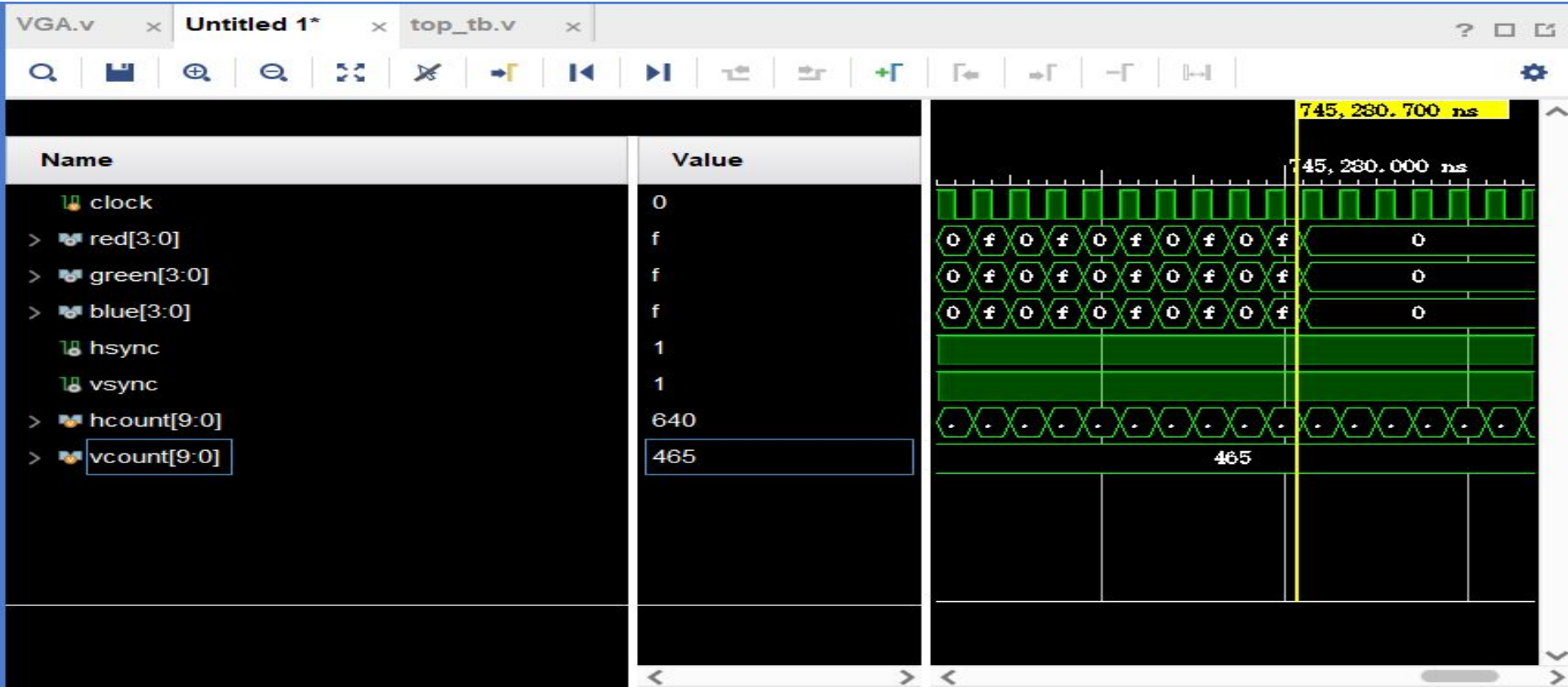
Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

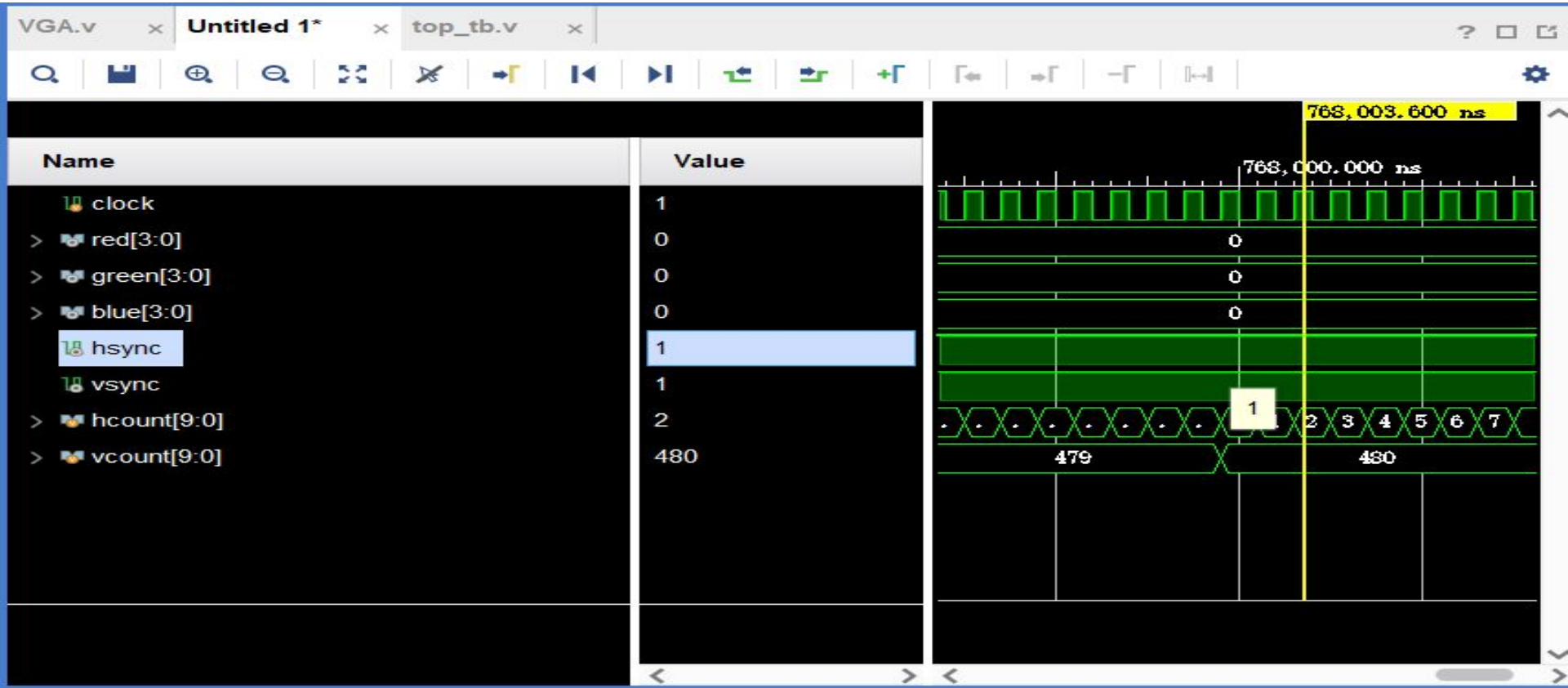
Constraints

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clock }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];

set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports { red[0] }]; #IO_L8N_T1_AD14N_35 Sch=vga_r[0]
set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports { red[1] }]; #IO_L7N_T1_AD6N_35 Sch=vga_r[1]
set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports { red[2] }]; #IO_L1N_T0_AD4N_35 Sch=vga_r[2]
set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports { red[3] }]; #IO_L8P_T1_AD14P_35 Sch=vga_r[3]

set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports { green[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports { green[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports { green[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports { green[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]

set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports { blue[0] }]; #IO_L2P_T0_AD12P_35 Sch=vga_b[0]
set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 } [get_ports { blue[1] }]; #IO_L4N_T0_35 Sch=vga_b[1]
set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 } [get_ports { blue[2] }]; #IO_L6N_T0_VREF_35 Sch=vga_b[2]
set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 } [get_ports { blue[3] }]; #IO_L4P_T0_35 Sch=vga_b[3]

set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports { hsync }]; #IO_L4P_T0_15 Sch=vga_hs
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports { vsync }]; #IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs
```

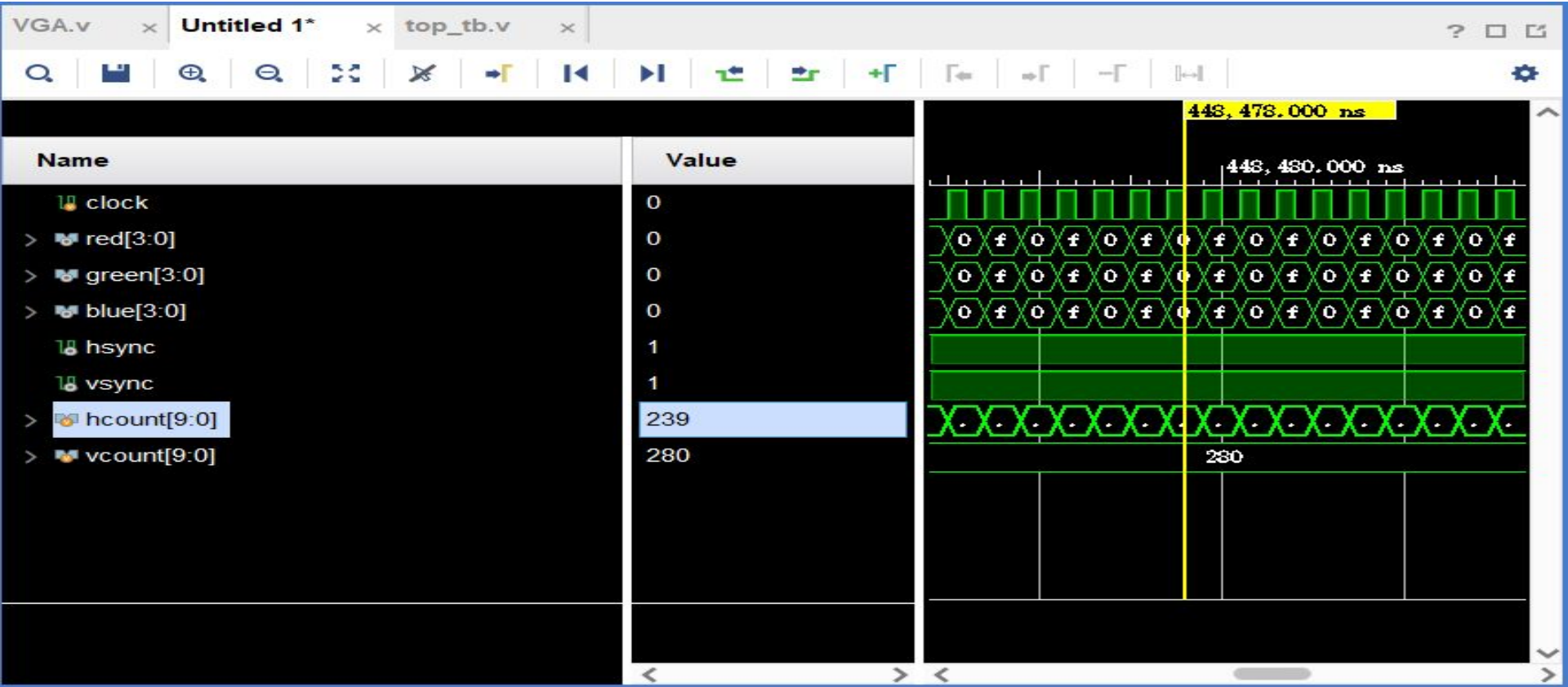
VGA Simulation Result



VGA Simulation Result



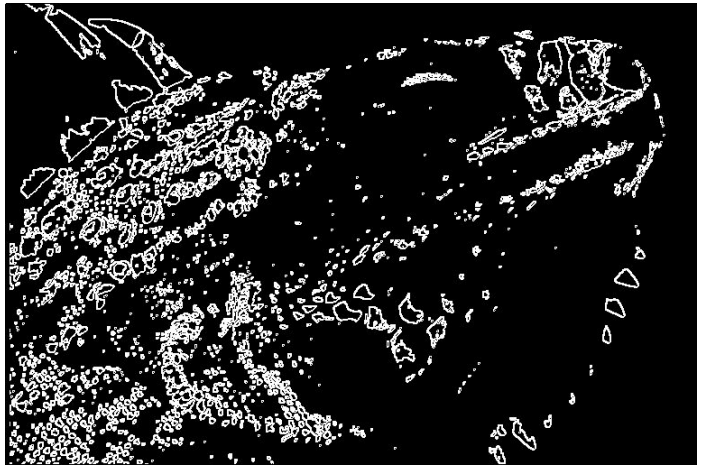
VGA Simulation Result



DetectoVision

Edge Detectors

DetectoVision



Edge Detectors

Successes

Running the entire pipeline without using the FPGA

- Constructed two simulations of the entire pipeline using Vivado
- The simulations can emulate the UART communication between PC and FPGA
- The simulation takes an image as input and outputs the processed image back to the PC using UART
- The images above were obtained this way

Shortcomings/What's Left

- The current version of Sobel stores the image, we are working to not have to rebuild the image until the final output
- The threshold of the sobel filter still needs adjustment
- We will add a switch to the algorithm so that the user can adjust the sensitivity of the algorithm(threshold) by changing the switch on the FPGA
- There is still a possibility for video
- We have not yet actually implemented this on an FPGA (sprinklers)

References

https://www.youtube.com/watch?v=sTHckUyxwp8

https://www.secureideas.com/blog/hardware-hacking-finding-uart-pinouts-on-pcbs

https://www.digi.com/resources/documentation/Digidocs/90001541/reference/r_serial_data.htm?TocPath=Serial%20communication|_____2

https://www.chipverify.com/verilog/verilog-file-io-operations

https://www.edaboard.com/threads/writing-values-of-a-variable-to-a-file-using-verilog-hdl.209993/