Dev Halai – FPGA Capstone Handout.

Designing a Sobel Image Processor Application FPGA Device.

For source files please check the following link:

https://www.dropbox.com/scl/fo/zopu2iilenfja66jdkjv5/AM_nD5e3WMt3CKN9hhcrR2s?rlkey=xquso3ji1e1eeghc1u9nrltua&dl=0

Contents

What's the purpose of an image filter application, especially on an FPGA?

Think of a car or any modern device where you might need visual data as an input to perform other actions. Like a dashboard on a smart car. Needs cameras with data to see where the road is, detect where objects and people are in real time.

High level overview of process:

Input is going to be a greyscale image

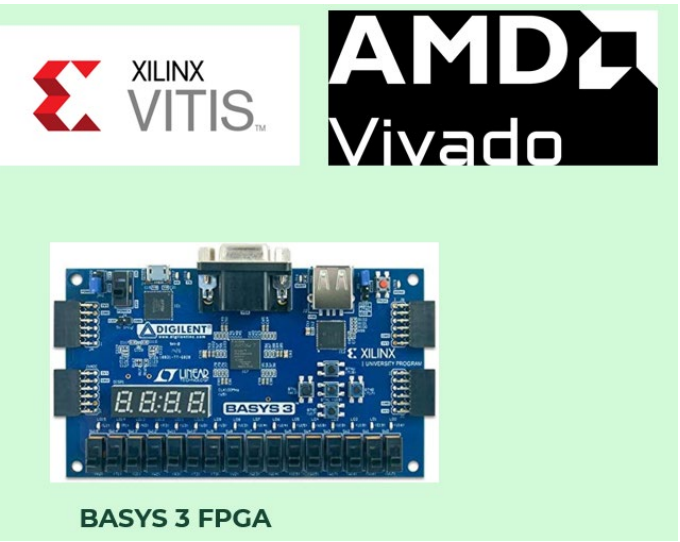
Apply a **threshold** over it.
Anything over our threshold is white, anything below is black.

Using this threshold apply our **sobel** filter (edge detection)

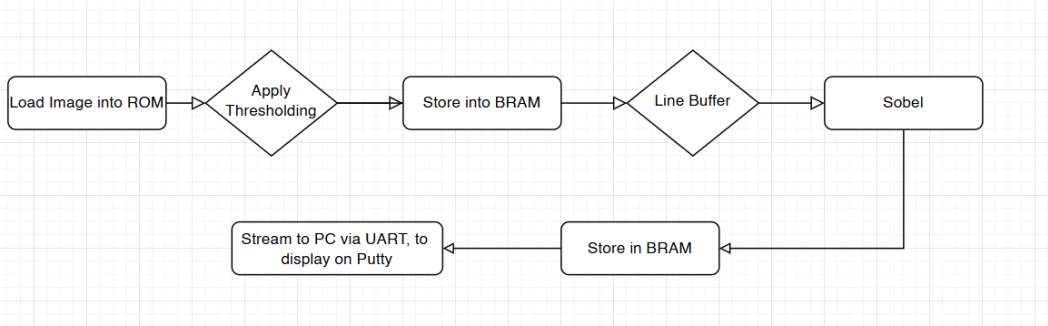
Output so we can see our result produced by FPGA.

Tech Stack used:

- **Vitis** through **Microblaze** for rapid prototyping, using **C** language.
- **AMD / Xilinx** suite for IDE and debugging tools such as simulation and ILA.
- **FPGA Basys 3** board, with **VHDL** as language, as well as **UART** to stream display to **Putty** terminal.

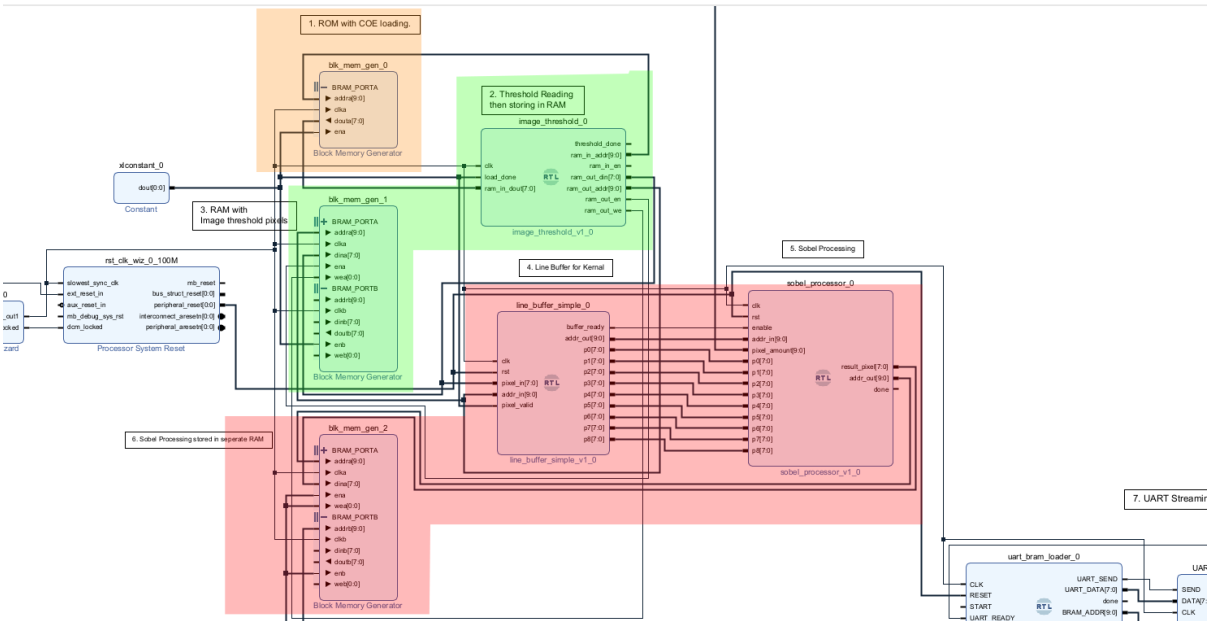


High Level Approach



Initial idea how I would approach the project. Loading the data in a ROM, applying a threshold filter, store those in another BRAM. Then for sobel I would need a line buffer to allow for the 9 pixels required each clock cycle for the algorithm. Store this in another BRAM and stream the output to putty via UART.

Block Diagram for main design.



On the left we have the clock wizard to generate a 100 Mhz clock.
In **Orange** is the image loaded as a COE to initialize our ROM with the grey scale image.

In **Green** is the threshold process which takes data from the ROM, applies a threshold filter and stores it in a dual port RAM (blk mem gen 1).

In **Pink** is our sobel filter. Sobel uses kernal convolution. So for this module we had to work on not just our immediate pixel, but surrounding ones too. A line / data buffer was needed to input the correct data into our sobel filter calculation. Which then the individual pixels could be stored in the BRAM at the correct address.

Overview : Top Level

All the modules used in my top level wrapper. Mix between vivado libraries such as clock wizard and black memory generators and custom modules such as my image threshold, sobel, line buffer and bram uart readers.

Overview : Resources Used

Intra-Clock Paths - clk_out1_top_clk_wiz_0_0

Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	8.177 ns	0.000 ns	0	5942
Hold	0.023 ns	0.000 ns	0	5942
Pulse Width	8.750 ns	0.000 ns	0	3530

LUT	FF	BRAM	URAM	DSP	Start	Elapsed
0.00 %	**0.00 %**	**0.00 %**	**0.00 %**	**0.00 %**	**5/15/25, 8:15 AM**	**00:00:34**
11.16	8.46 %	12.00 %	0.00 %	0.00 %	5/15/25, 8:16 AM	00:02:35
					5/13/25, 8:56 PM	35:19:04

In order to meet timing requirements and reduce violations I had to reduce clock to 50Hz since image buffer had too much slack. Previous design failed on timing as shown below.

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS
synth_1 (active)	**constrs_1**	**Synthesis Out-of-date**					
impl_1	constrs_1	Implementation Out-of-date	-0.607	-0.607	0.028	0.000	9.213

Significant Code : Image Threshold

```
entity image_threshold is
    Port (
        clk             : in  std_logic;
        load_done       : in  std_logic;
        threshold_done  : out std_logic;

        -- Input BRAM
        ram_in_dout     : in  std_logic_vector(7 downto 0);
        ram_in_addr     : out std_logic_vector(9 downto 0);
        ram_in_en       : out std_logic;

        -- Output BRAM
        ram_out_din     : out std_logic_vector(7 downto 0);
        ram_out_addr    : out std_logic_vector(9 downto 0);
        ram_out_en      : out std_logic;
        ram_out_we      : out std_logic
    );
end image_threshold;
```

Image on the left shows the ports my image threshold module. This module was designed to work with dual port RAMs in mind.

```
    process(clk)

        -- variables update in same clock cycle, good for simple calcs with no need to be held.

        variable pixel      : unsigned(7 downto 0);
        variable out_pixel  : std_logic_vector(7 downto 0);

    begin
        if rising_edge(clk) then
            case state is

--              waits for a load signal to start to make sure pixel is ready to be read otherwise will read empty values
                when IDLE =>
                    if load_done = '1' then
                        ptr        <= (others => '0');
                        done_flag  <= '0';
                        state      <= READ_WRITE;
                    end if;

--              Was made with dual port ram in mind. We can take input pixel, apply threshold, and write to different ram port.
                when READ_WRITE =>
                    -- Read one pixel

                    -- unsigned converts it from 8bit vector to a decimal, just makes it easier for comparison.
                    pixel := unsigned(ram_in_dout);

                    -- Threshold logic
                    if pixel > THRESHOLD then
                        out_pixel := x"FF"; -- white
                    else
                        out_pixel := x"00"; -- black
                    end if;

                    -- Output write
                    ram_in_en      <= '1';
                    ram_in_addr    <= std_logic_vector(ptr);
                    ram_out_en     <= '1';
                    ram_out_we     <= '1';  -- write enable for LSB byte
                    ram_out_addr   <= std_logic_vector(ptr);
                    ram_out_din    <= out_pixel;
```

A simple FSM (finite state machine) was used to set the module as active. With this design there was no need to put read and write in separate states as it was designed to work with dual port rams. An image threshold looks at the brightness of a pixel (usually greyscale), and it compares it to a threshold value. If it below the threshold value it usually all black, if it above, its usually white. With nothing in between.

For this calculation since it was a simple comparison, variables were instead of signals to use less resources and start outputting the threshold value the same cycle the pixel data was inputted.

```
                    -- Increment address
                    -- Ptr was added here because AXI uses 32 bit addrss and might have to loop over by 4.
                    if ptr < IMG_SIZE - 1 then
                        ptr <= ptr + 1;
                    else
                        state <= DONE;
                    end if;

                when DONE =>
                    ram_in_en      <= '0';
                    ram_out_en     <= '0';
                    ram_out_we     <= '0';
                    done_flag      <= '1';

                when others =>
                    state <= IDLE;

            end case;
        end if;
    end process;

    threshold_done <= done_flag;

end Behavioral;
```

After all the pixels were done going through thresholding and loaded into the BRAM, the module would give a 1 bit output signal so the sobel module could know when to start.

Testbench for Image Threshold Module

The threshold module was very simple in terms of logic, the longer part was simulating since I needed to fill it with fake image pixel data that was randomly generated.

Unfortunately I could not give in actual pixel data with time constraints so I generated random values instead.

```
93      -- Test sequence
94      test_process: process
95      begin
96          -- Wait some time then start
97          wait for 20 ns;
98
99          -- Populate input RAM with pseudo-random values
100         for i in 0 to 1023 loop
101             input_mem(i) <= std_logic_vector(to_unsigned((i * 37) mod 256, 8));
102         end loop;
103
104         -- Trigger threshold processing
105         load_done <= '1';
106         wait for 10 ns;
107         load_done <= '0';
108
109     end process;
110
111 end Behavioral;
112
```

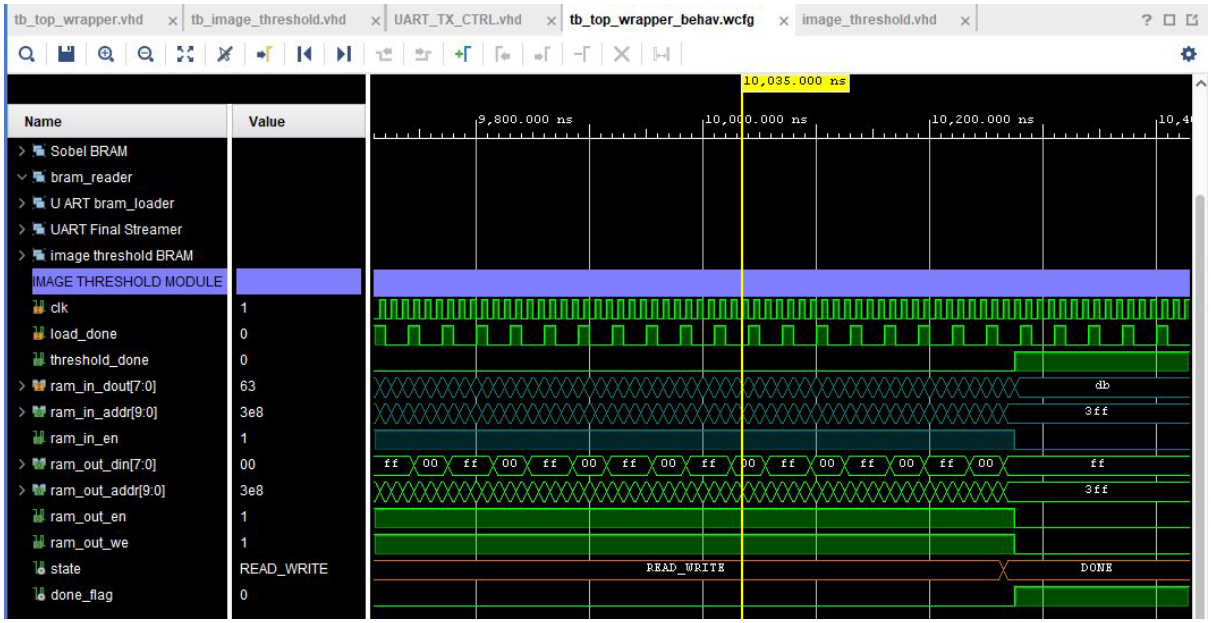


Image threshold testbench shows random pixel values coming in through ‘ram_in_dout’ and a change of values all being either 00 or ff showing that it’s working correctly. And at the end of the address, 3ff (1024 addresses) the state goes from read/write to done and sends a ‘done’ signal.

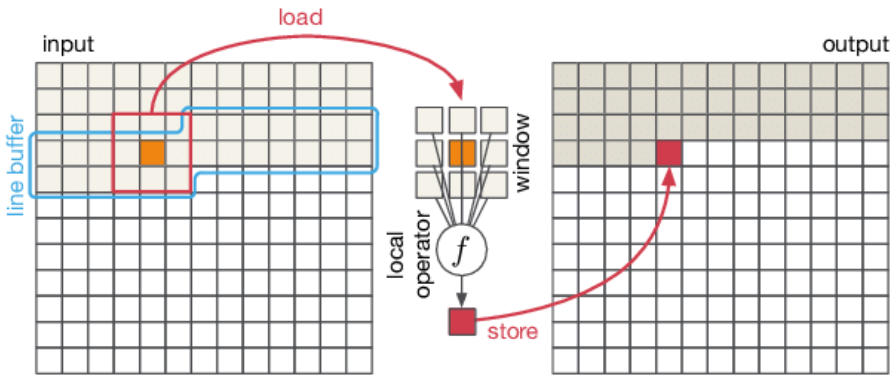
Significant Code: Image Buffer

This is the buffer needed to supply our sobel kernel with a continuous stream of 9 pixels so it can perform the calculation. In order to this we have to create a buffer where we preload a large number of pixels from the BRAM. Then we can start moving data from the BRAM whilst simultaneously running our sobel calculations within the same cycle.

```
25  entity line_buffer_simple is
26      Port (
27          clk         : in  std_logic;                        -- System clock
28          rst         : in  std_logic;                        -- Reset
29          pixel_in    : in  std_logic_vector(7 downto 0);     -- Incoming pixel
30          addr_in     : in  std_logic_vector(9 downto 0);     -- Address of current pixel
31          pixel_valid : in  std_logic;                        -- Indicates valid pixel input
32
33          buffer_ready : out std_logic;                       -- High when window is valid
34          addr_out     : out std_logic_vector(9 downto 0);    -- Delayed address output
35
36          -- Output: 3x3 pixel window (top to bottom, left to right)
37          p0, p1, p2 : out std_logic_vector(7 downto 0);      -- Top row
38          p3, p4, p5 : out std_logic_vector(7 downto 0);      -- Middle row
39          p6, p7, p8 : out std_logic_vector(7 downto 0)       -- Bottom row
40      );
41  end line_buffer_simple;
```

For this module to work. I needed to take in the pixels from the BRAM. But the output was very important since it required the following:

- A signal several clock cycles ahead to let sobel the pixel stream was ready.
- Address output identical to the input but delayed to store sobel pixels to correct location.
- 9 registers that would continuously update each clock cycle so sobel could work.



The image above is similar to what we are doing. Placement and buffer lengths vary but it's allowing enough pixels for Sobel to work continuously.

```
53      type line_buffer_type is array (0 to IMG_WIDTH - 1) of std_logic_vector(7 downto 0);
54      signal lb1, lb2, lb3, lb4 : line_buffer_type := (others => (others => '0'));
55
56      -- Current column index for writing into the line buffers
57      signal col_index   : integer range 0 to IMG_WIDTH - 1 := 0;
58
59      -- Counts how many rows have been processed (used to determine readiness)
60      -- rows 0 - 2 are used for kernal processing, we load in row 3 while we go left to right to keep it continous.
61      signal row_counter : integer range 0 to 3 := 0;
62
63      -- Temporary signals for the current pixel in each of the 3 rows used for window
64      signal shift_reg1, shift_reg2, shift_reg3 : std_logic_vector(7 downto 0) := (others => '0');
65
66      -- Shift registers holding last 3 pixels from each row
67      signal sr_p0, sr_p1, sr_p2 : std_logic_vector(7 downto 0);
68      signal sr_p3, sr_p4, sr_p5 : std_logic_vector(7 downto 0);
69      signal sr_p6, sr_p7, sr_p8 : std_logic_vector(7 downto 0);
70
71      -- Address pipelining to align address timing with pixel window
72      signal addr_stage1, addr_stage2, addr_stage3, addr_stage4 : std_logic_vector(9 downto 0) := (others => '0');
```

Image above is showing the data types we are using and what is needed for the calculation. We need 4 line buffers, 3 for sobel to run, and 1 to act as a buffer so we can continuously run. The pixels will be loaded sequentially from left to right, then move down to up, this is why we have a column index and row count.

```
84          elsif rising_edge(clk) then
85              if pixel_valid = '1' then
86
87                  -- each pixel gets stored in lb1(col_index) straight away.
88                  -- each pixel gets stored from right to left, and moves from down to up.
89                  -- This is feeding the line buffers, it also then gets pushed up,
90
91                  -- (1) Shift all line buffers down by one row at current column index
92                  lb4(col_index) <= lb3(col_index);
93                  lb3(col_index) <= lb2(col_index);
94                  lb2(col_index) <= lb1(col_index);
95                  lb1(col_index) <= pixel_in; --actually starts from this line, (becaues pixels are
96
97                  -- (2) Pipeline address to align with output window timing
98                  addr_stage1 <= addr_in;
99                  addr_stage2 <= addr_stage1;
100                 addr_stage3 <= addr_stage2;
101                 addr_stage4 <= addr_stage3;
102                 addr_out    <= addr_stage4;
103
```

The address had to be delayed between clock cycles. Since we are ‘buffering pixels’, their input address needs to match the time the sobel filter receives them so we can load them back to their corresponding address as the rest of the images.

Sr_p0 … sr_8 will be registers. These are going to hold data for up to 3 clock cycles before each one is replaced. These will be individually fed to the sobel algorithm. Shift registers are used for the technique since they're fast and this is a way to permit data for a small amount of time.

```
                -- (3) Grab the current pixel from each row and store in horizontal shift regs
                shift_reg1 <= lb4(col_index);  -- top row
                shift_reg2 <= lb3(col_index);  -- middle row
                shift_reg3 <= lb2(col_index);  -- bottom row

                --each clock cycle these are being updated with new values.
                -- each clock cycle we pass those current values to a shift register, to hold data
                -- shown below.

                -- (4) Shift horizontal pixels left (simulate 3-pixel window across)
                sr_p0 <= sr_p1;
                sr_p1 <= sr_p2;
                sr_p2 <= shift_reg1;

                sr_p3 <= sr_p4;
                sr_p4 <= sr_p5;
                sr_p5 <= shift_reg2;

                sr_p6 <= sr_p7;
                sr_p7 <= sr_p8;
                sr_p8 <= shift_reg3;

                -- (5) Increment column and manage row count
                if col_index = IMG_WIDTH - 1 then
                    col_index <= 0;

                    if row_counter < 3 then
                        row_counter <= row_counter + 1;
                    end if;

                    -- (6) Assert buffer_ready once 3 full rows are available
                    if row_counter = 2 then
                        buffer_ready <= '1';
                    end if;
                else
                    col_index <= col_index + 1;
                end if;
            end if;
        end if;
    end process;

    -- Final pixel window outputs
    p0 <= sr_p0; p1 <= sr_p1; p2 <= sr_p2;
    p3 <= sr_p3; p4 <= sr_p4; p5 <= sr_p5;
    p6 <= sr_p6; p7 <= sr_p7; p8 <= sr_p8;

end Behavioral;
```

Testbench shown together with sobel filter on next few pages.

Significant Code: Sobel Filter

```
entity sobel_processor is
    Port (
        clk           : in  std_logic;
        rst           : in  std_logic;
        enable        : in  std_logic;

        addr_in       : in  std_logic_vector(9 downto 0);
        pixel_amount  : in  std_logic_vector(9 downto 0);  -- number of pixels expected

        -- 3x3 Window Pixels
        p0, p1, p2    : in std_logic_vector(7 downto 0); -- Top row
        p3, p4, p5    : in std_logic_vector(7 downto 0); -- Middle row
        p6, p7, p8    : in std_logic_vector(7 downto 0); -- Bottom row

        result_pixel  : out std_logic_vector(7 downto 0);
        addr_out      : out std_logic_vector(9 downto 0);
        done          : out std_logic
    );
end sobel_processor;
```

Sobel Filter takes the inputs from the line buffer output, and is now able to perform a calculation for the centre pixel (p4) by using its neighbouring pixels.

```
architecture Behavioral of sobel_processor is
    constant THRESHOLD     : integer := 128;
    signal pixel_count     : unsigned(9 downto 0) := (others => '0');
    signal processing_done : std_logic := '0';
begin

    process(clk, rst)
        variable gx, gy    : integer range -2048 to 2047;
        variable magnitude : integer;
    begin
        if rst = '1' then
            result_pixel    <= (others => '0');
            addr_out        <= (others => '0');
            pixel_count     <= (others => '0');
            processing_done <= '0';

        elsif rising_edge(clk) then
            if enable = '1' and processing_done = '0' then

                -- convert hex value to unsigned (decimal).
                -- convert to integer, because unsigned can't be negative and we need negative values in sobel.

                -- Compute Sobel Gx
                gx := to_integer(unsigned(p2)) + 2 * to_integer(unsigned(p5)) + to_integer(unsigned(p8))
                    - to_integer(unsigned(p0)) - 2 * to_integer(unsigned(p3)) - to_integer(unsigned(p6));

                -- Compute Sobel Gy
                gy := to_integer(unsigned(p6)) + 2 * to_integer(unsigned(p7)) + to_integer(unsigned(p8))
                    - to_integer(unsigned(p0)) - 2 * to_integer(unsigned(p1)) - to_integer(unsigned(p2));

                -- Magnitude approximation
                magnitude := abs(gx) + abs(gy);

                -- Thresholding
                if magnitude > THRESHOLD then
                    result_pixel <= x"00"; -- edge
                else
                    result_pixel <= x"FF"; -- no edge
                end if;

                addr_out <= addr_in;

```

$$G_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

$$G_y = \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

The calculation is done using variables because we do not need to store this data. In order to find the sobel edge value we need the get the x and y gradients first. We do this by getting the edge pixels respectively and multiplying them by the amount respective of the figure on the right top. After we do this for both x and y, we usually use Pythagoras to get the value of g. (gx^2 + gy^2) square rooted. However this is 'expensive' in terms of fpga resources so doing abs(gx) and abs (gy) gives us a good approximation.

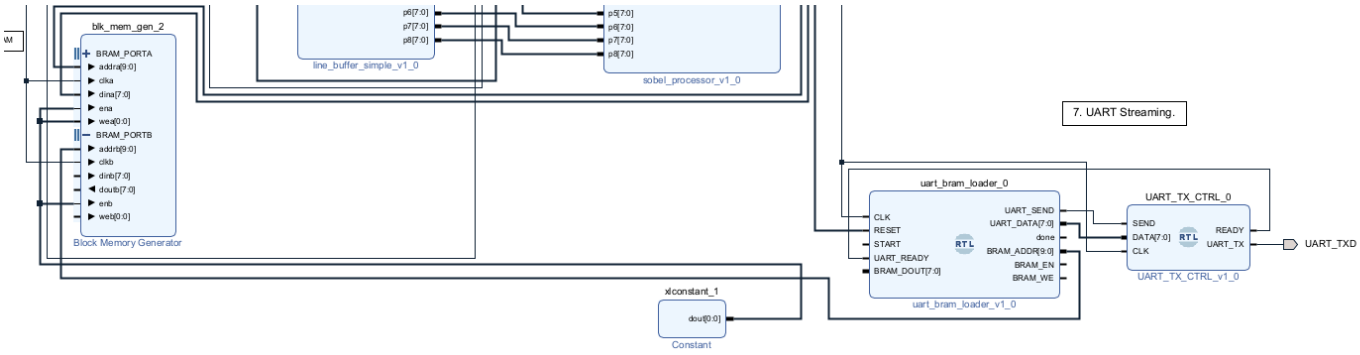
Like the threshold we see if our gradient has a strong value past our 'threshold' mark. This will output either black or white and let us know if the value is deemed an edge.

Testbenches for Sobel

Figure above shows the line buffer working as well as the sobel module. First the pixels are streamed into line buffer as shown in purple. After the first 3 rows are loaded and sobel has enough to start working, we get the buffer ready signalling sobel to start (in green). These values are read from BRAM 1 which stored the threshold pixel images.

Now f or the sobel module, we can see the result pixel start changing, different from the threshold pixel values, showing they are indeed going through a different algorithm. All the results are either 00 or ff which is expected. And right at the end when all of them are done, we get the done signal in aqua colour turn to ‘1’ allowing UART to receive a notification we’re ready to stream because our data has been fully processed and stored in our final BRAM.

Significant Code: UART streaming via VHDL.



To check the data was correct and to get a visual output, I used UART to stream the pixels to a Putty terminal with a few VHDL modules. Above shows the block diagram setup.

UART_TX_CTRL_0 was provided by the demo file on the board. This is only a transmitter UART module. Since it requires us to only send data 1 byte a time, I made a custom BRAM loader using a FSM (finite – state machine). This was done in the module UART_BRAM_LOADER.

```
42  entity UART_TX_CTRL is
43      Port ( SEND : in  STD_LOGIC;
44             DATA : in  STD_LOGIC_VECTOR (7 downto 0);
45             CLK : in  STD_LOGIC;
46             READY : out  STD_LOGIC;
47             UART_TX : out  STD_LOGIC);
48  end UART_TX_CTRL;
```

Above is the port for the UART TX CTRL module provided by the board. It needed adjustments as it was setup to stream at 9600 baud rate for a 100mhz processor. Since I used a 50 MHz processor I had to edit the transfer rate.

```
--constant BIT_TMR_MAX : std_logic_vector(13 downto 0) := "10100010110000"; --10416 = (round(100MHz / 9600)) - 1
constant BIT_TMR_MAX : std_logic_vector(13 downto 0) := "01010010111000"; -- 5208 for 50mHz
```

In the UART_BRAM_LOADER I also included a function that will convert all data values (which I knew would be pixel values) into ASCII characters.

This way I could map specific character values to certain characters and represent the different values to make an image.

Variables were used here, since there was no need to store this data across clock cycles.

Function show on the right ‘grayscale_to_ascii(pixel)’. Idea and code was taken from one of my peers – Charlie Watson.

```
22  architecture Behavioral of uart_bram_loader is
23
24      type state_type is (IDLE, READ, WAIT_CYCLE, SEND, WAIT_LOW, DONE);
25      signal state : state_type := IDLE;
26
27      signal address       : unsigned(9 downto 0) := (others => '0');
28      signal send_pulse    : std_logic := '0';
29      signal data_to_send  : std_logic_vector(7 downto 0) := (others => '0');
30      signal current_byte  : std_logic_vector(7 downto 0) := (others => '0');
31
32      signal use_ascii     : std_logic := '1';  -- Set to '0' for raw output
33
34      constant END_ADDR : unsigned(9 downto 0) := to_unsigned(1024, 10);  -- 1024 bytes
35
36      -- ASCII grayscale mapping
37      function grayscale_to_ascii(pixel : std_logic_vector(7 downto 0)) return std_logic_vector is
38          variable value : integer := to_integer(unsigned(pixel));
39      begin
40          if value < 32 then
41              return x"23"; -- '#'
42          elsif value < 64 then
43              return x"2B"; -- '+'
44          elsif value < 96 then
45              return x"2E"; -- '.'
46          elsif value < 128 then
47              return x"2D"; -- '-'
48          elsif value < 160 then
49              return x"3A"; -- ':'
50          elsif value < 192 then
51              return x"2A"; -- '*'
52          elsif value < 224 then
53              return x"25"; -- '%'
54          else
55              return x"20"; -- ' '
56          end if;
57      end function;
58
```

```
79                  when IDLE =>
80                      if START = '1' then
81                          address   <= (others => '0');
82                          BRAM_EN   <= '1';
83                          BRAM_WE   <= '0';
84                          BRAM_ADDR <= std_logic_vector(address);
85                          state     <= READ;
86                      end if;
87
88                  when READ =>
89                      state <= WAIT_CYCLE;
90
91                  when WAIT_CYCLE =>
92                      current_byte <= BRAM_DOUT;
93                      state <= SEND;
94
95                  when SEND =>
96                      if UART_READY = '1' then
97                          if use_ascii = '1' then
98                              ascii_char := grayscale_to_ascii(current_byte);
99                              data_to_send <= ascii_char;
100                         else
101                             data_to_send <= current_byte;
102                         end if;
103                         send_pulse <= '1';
104                         state <= WAIT_LOW;
105                     end if;
106
107                 when WAIT_LOW =>
108                     if UART_READY = '0' then
109                         if address < END_ADDR - 1 then
110                             address   <= address + 1;
111                             BRAM_ADDR <= std_logic_vector(address + 1);
112                             state     <= READ;
113                         else
114                             state <= DONE;
115                         end if;
116                     end if;
117
118                 when DONE =>
119                     BRAM_EN <= '0';
120                     BRAM_WE <= '0';
```

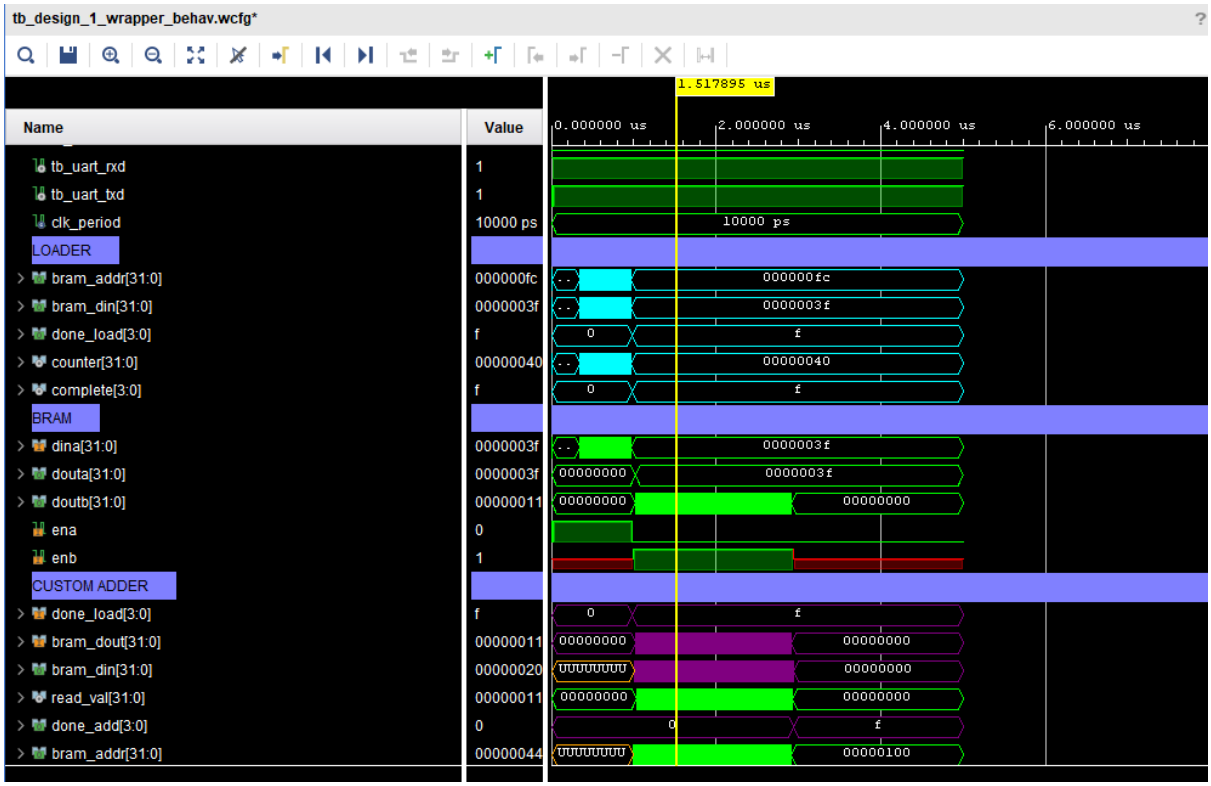
This is the finite state machine used in the ‘uart_bram_loader’ module.

Since we can only send data one byte at a time, and need to wait for a response when done before loading another pixel, a state machine was used. This made it easy to track when to get data from the BRAM (in a sequential process), when to send data, and when to retrieve data again (once the UART_TX module confirmed data was sent across successfully).

A testbench wasn’t really used for this module, I essentially had to trust it was working. Because UART requires a ‘accepted/done’ signal once data is sent from the external source. It felt redundant simulating that since it would be up to an external factor.

Test Study - BRAM Sample File.

Since I was going to be using BRAM a lot in my project to store data, read data, manipulate and write back. I decided to create a mini-isolated project focusing on loading and adding numbers and writing back to get comfortable with the process.



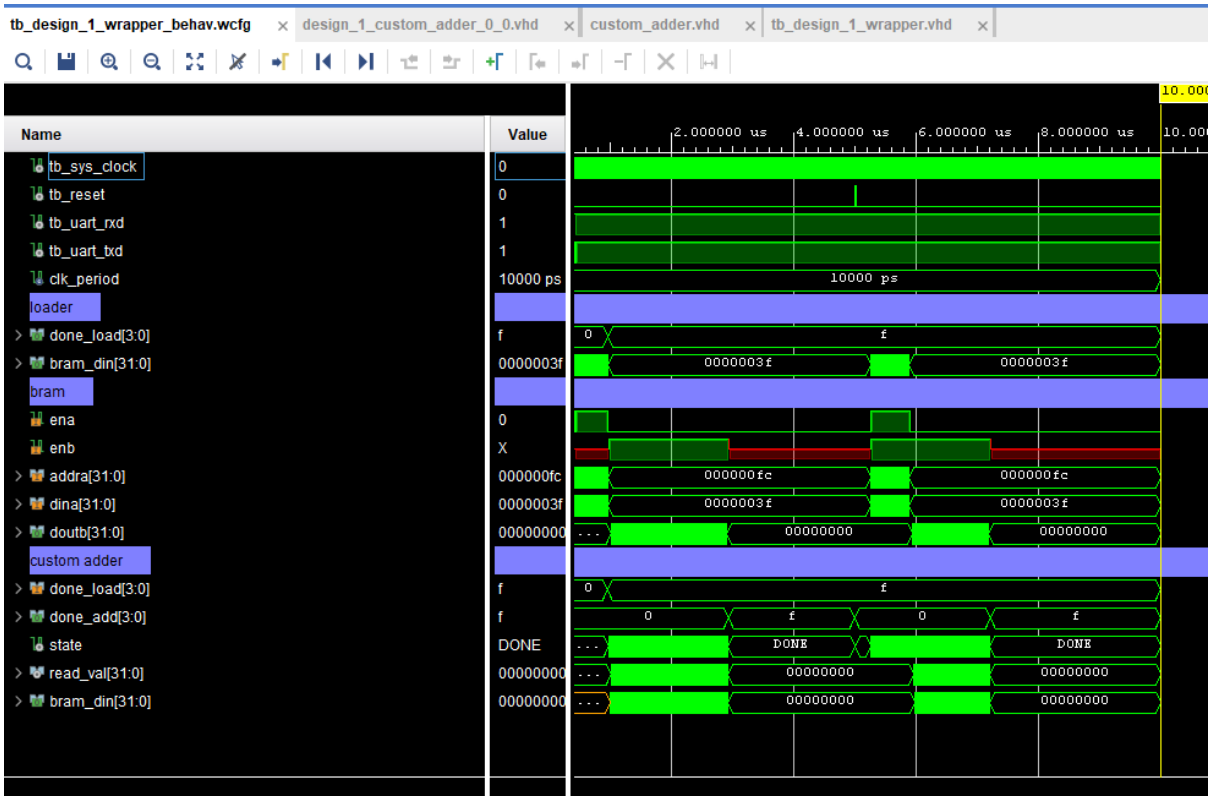
The testbench was incredibly useful here especially with the ability to group / divide and colour data to distinguish between so many signals and off different data widths.

Data is 32 bits since original plan was to either stream or write data back to vitis to use with PMOD. Later changed to 8 bits to save space on FPGA board.

This proved invaluable.

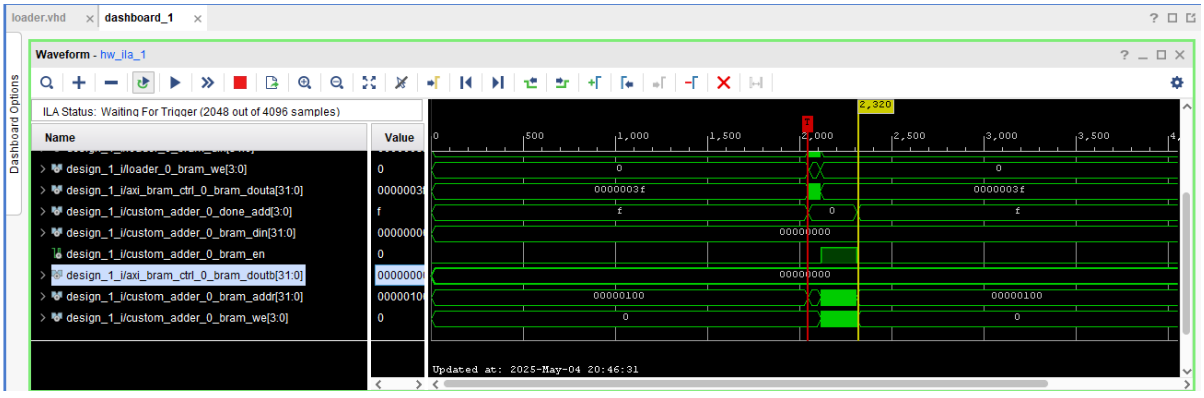
Additional tests and findings

1. Reset button working (makes catching ILA easier). Test bench working as expected.



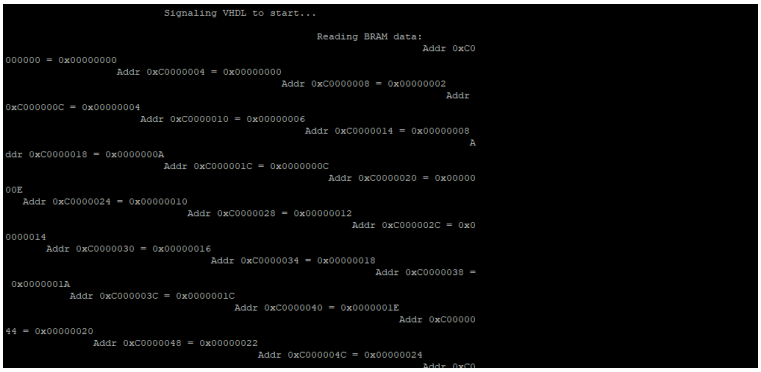
Simple connection but proved vital when needing to capture data on ILA. When programming FPGA with bitstream, the calculations happen too fast before ILA gets opportunity to be armed.

So a reset was used so I could set up an ILA, and reset the device and then see when signals are met.



Above on ILA. Port B isn't being read as it should for BRAM. We can't read the info for some reason.

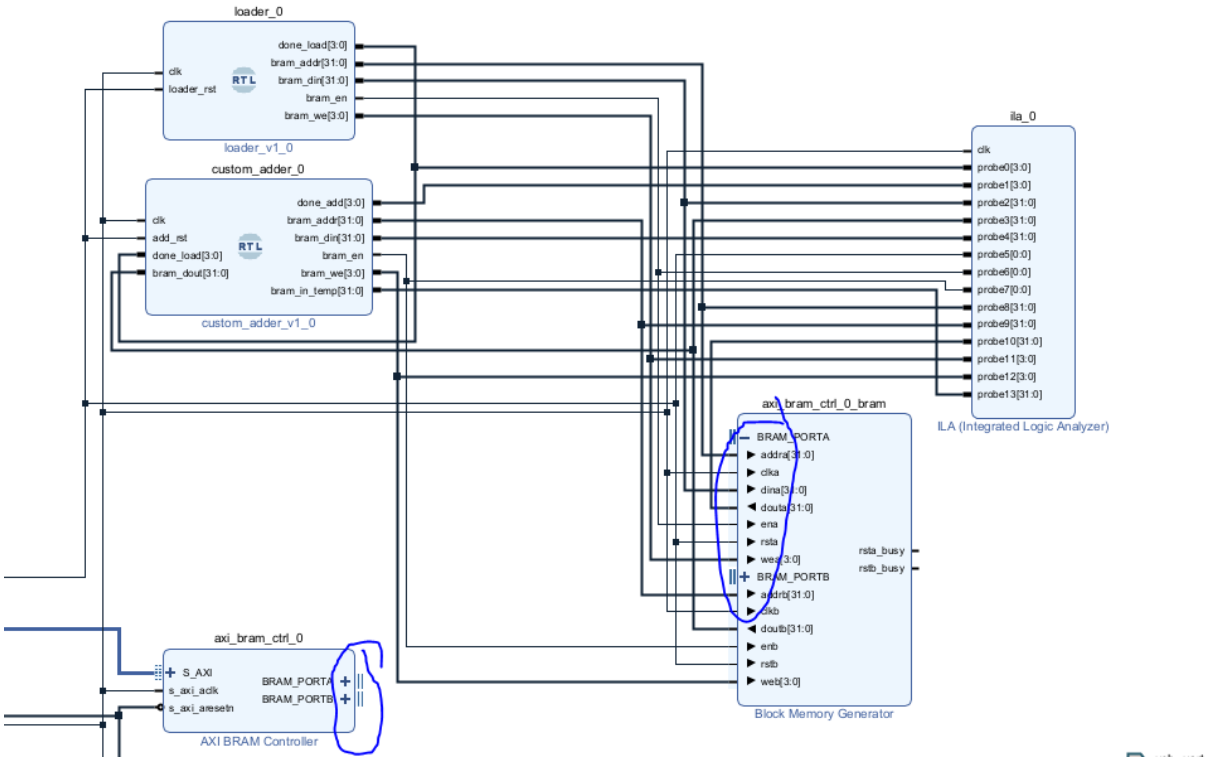
2. Able to read my AXI BRAM Data in Vitis (for Pmod if wanted to use). Basic UART showing incremented address and array of increasing numbers.



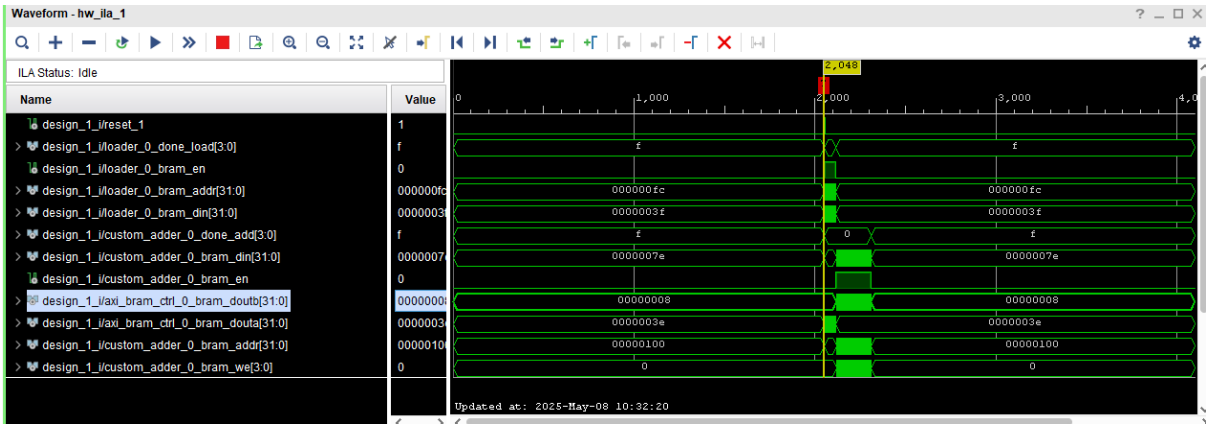
Output from Putty.

```c
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_GPIO_0_DEVICE_ID
#define BRAM_BASEADDR  XPAR_BRAM_0_BASEADDR
#define BRAM_WORDS     256    // Number of 32-bit words (adjust if needed)
#define WORD_OFFSET    4      // Each word is 4 bytes

XGpio Gpio;

int main() {
    xil_printf("=== FPGA BRAM Read Tool ===\n");

    // Initialize GPIO
    if (XGpio_Initialize(&Gpio, GPIO_DEVICE_ID) != XST_SUCCESS) {
        xil_printf("GPIO Init Failed\n");
        return XST_FAILURE;
    }
    XGpio_SetDataDirection(&Gpio, 1, 0x00);  // GPIO as output

    // Optional: Signal VHDL to begin
    xil_printf("Signaling VHDL to start...\n");
    XGpio_DiscreteWrite(&Gpio, 1, 0x01);  // Custom start signal (if used)
    sleep(1);

    xil_printf("\nReading BRAM data:\n");
    for (u32 i = 0; i < BRAM_WORDS; i++) {
        u32 addr = BRAM_BASEADDR + (i * WORD_OFFSET);
        u32 data = Xil_In32(addr);
        xil_printf("Addr 0x%08X = 0x%08X\n", addr, data);
    }

    // Optional: Reset GPIO signal
    XGpio_DiscreteWrite(&Gpio, 1, 0x00);

    xil_printf("\nRead complete.\n");
    return 0;
}
```

Code in C to read data from AXI once loaded.



AXI Bram was connected to block memory generator. I didn't know the signals beneath were being overridden by AXI. PortA worked fine, but portB was causing issues. Disconnecting made it work. I realised its best to have one port for AXI, rest for regular BRAM access without AXI.



Data is now coming out of portb as expected, and my custom adder din (final result) has the data I expected.