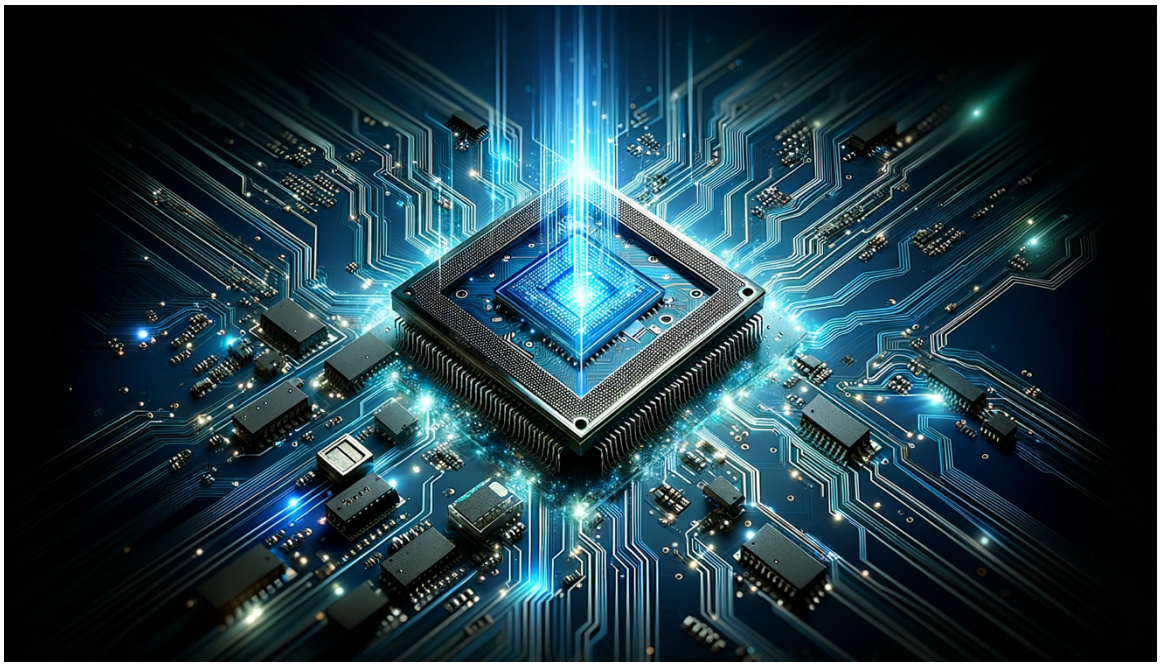


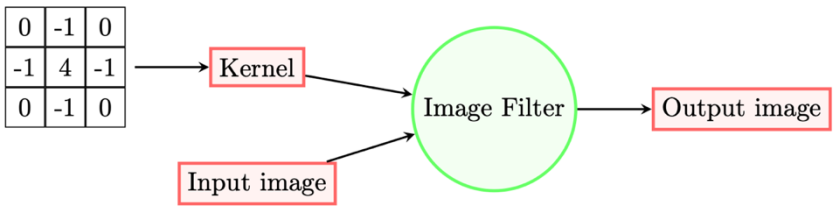
Hardware Assignment 3

IMPLEMENTATION OF A 3X3 IMAGE FILTERING OPERATION

Priyanshu Agrawal (2022CS11641) | Aditya Prakash (2022CS11595)
COL215 Fall'23

Introduction

In this assignment, we have advanced from the foundations laid in Hardware Assignment 2, focusing on the implementation of an efficient 3 x 3 filter applied to an input image to produce the final output. This enhancement involves processing for the eight surrounding pixels of the current pixel, an expansion from the previous limit of two. To achieve this, we have adopted a modular and comprehensible approach, integrating MAC operations and a Finite State Machine (FSM) framework. The task involves handling an input image of 64 x 64 pixels, processed through a filter kernel sized 3 x 3.



Note: In this assignment, we significantly restructured our code to integrate a Finite State Machine (FSM), diverging from our initial methodology in Part 1. Initially, we adapted our approach from Hardware Assignment 2, retaining the concept of pipelining but expanding from one to three pipelines. Despite achieving accurate testbench simulations with correct values in the RAM, we encountered persistent difficulties with the VGA display functionality. The code had also become considerably lengthy, complicating our debugging efforts. Despite exhaustive attempts to rectify these issues, the display remained non-functional. Consequently, we decided to redevelop the code from scratch, with a focus on FSM integration. This strategic decision markedly improved the modularity and clarity of our code, making it more manageable and understandable.

Components required:-

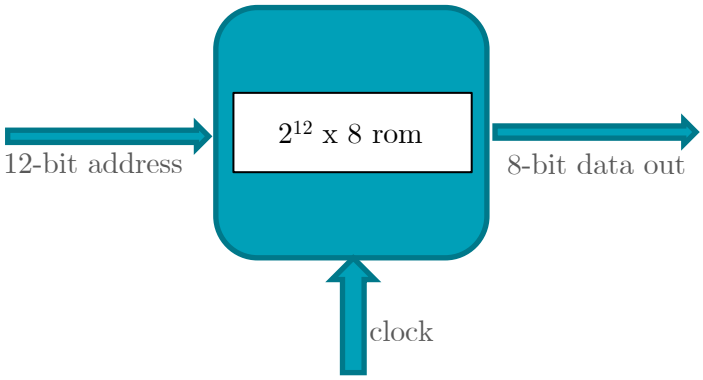
1. Memory Elements :
    a. ROM
    b. RAM
    c. Registers
2. Compute unit :
    a. Multiplier-Accumulator Block (MAC)
    b. Finite State Machine (FSM)
3. VGA Controller

USING BRAMS AND BROMS

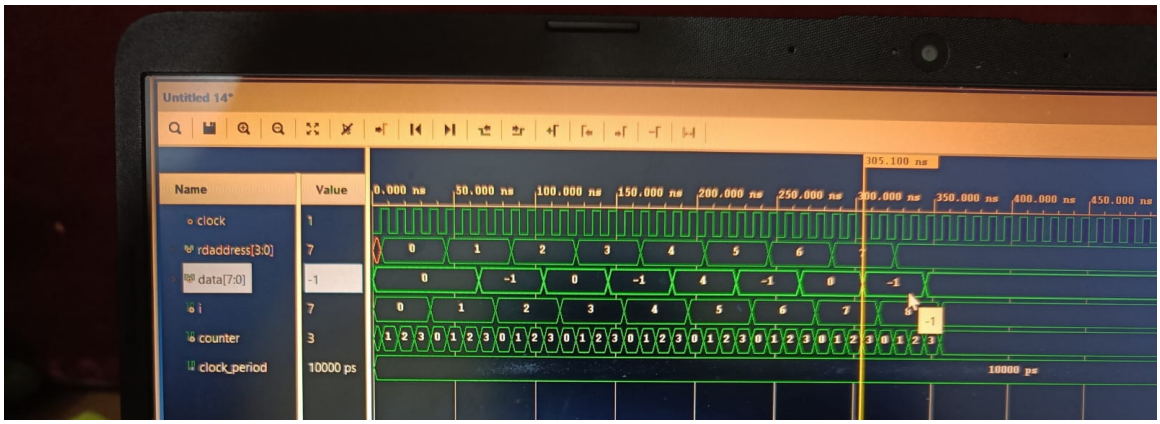
We will use block memory generators instead of distributed memory generators for this part of the assignment. BRAMs and BROMs are preferred when dealing with large blocks of memory. They typically offer faster access times and higher bandwidth compared to LUT-based memory. Since they are a dedicated memory on the FPGA, they do not consume any logic resources, making it more efficient.

BROM

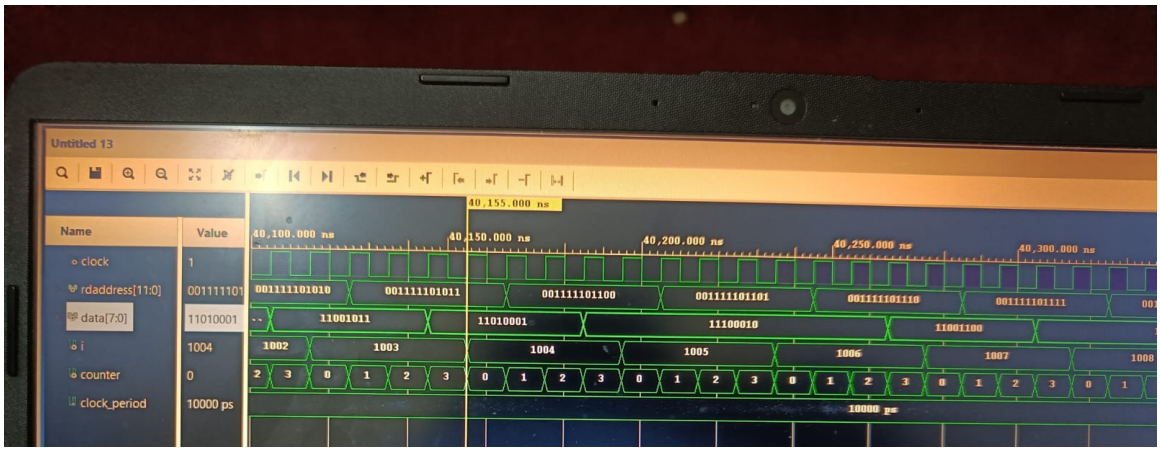
A simple ROM consists of three ports: an address in, a clock in, and a data out. We needed to create two ROMs, one for the filter and one for the image data. We initialized our image ROM with the given image COE file (2^{12} x 8 bits) and the filter ROM with the given filter COE file (9 x 8 bits). The requisite memory is generated automatically with Vivado. We can design a testbench to check this. The code for the same is already attached.



Testbenches for both ROMs have been included in the submission.



The simulated waveform for the Filter ROM



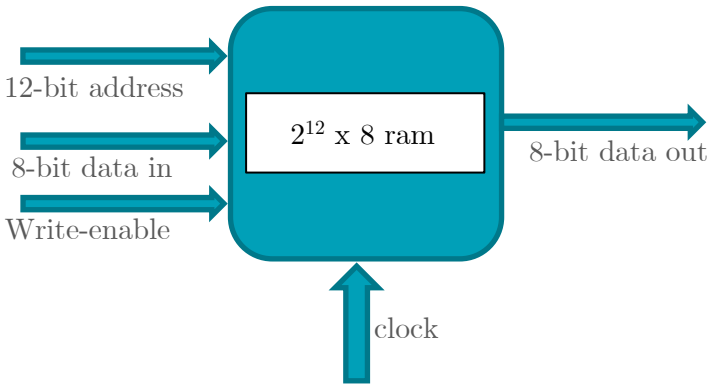
The simulated waveform for the face_bin_64 image ROM

BRAM

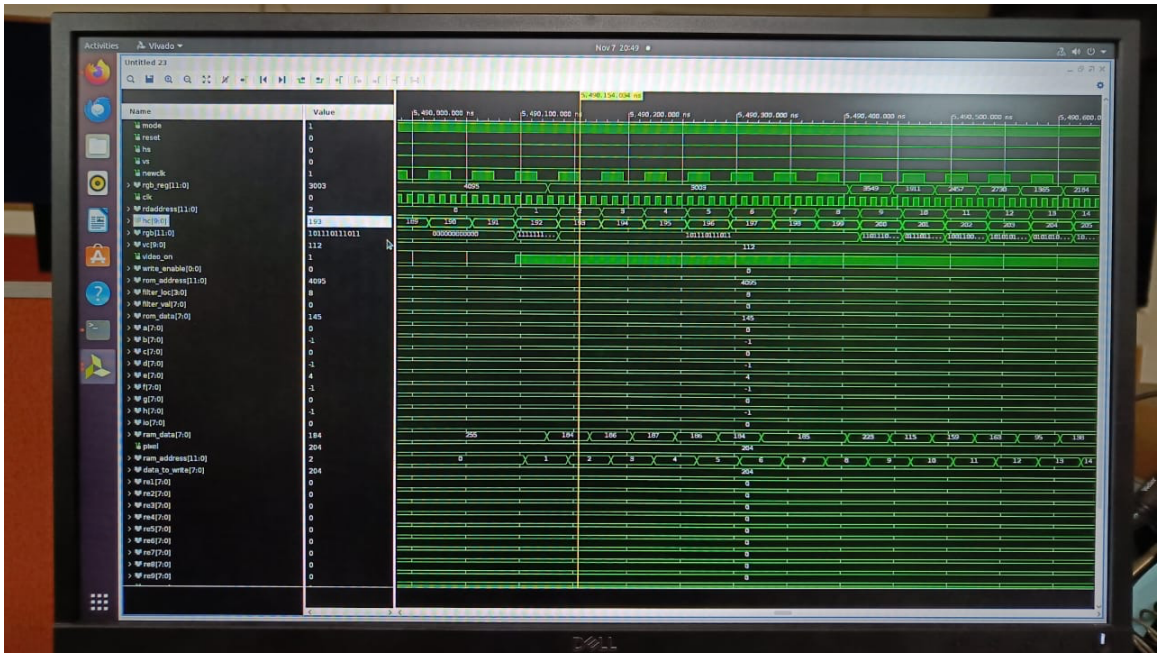
The process of making a RAM is very similar to that of ROM, except instead of initializing a RAM, we leave it empty. A typical single-port RAM has five ports:

1. An address in port
2. A data in port
3. A clock in port
4. A write-enable in port
5. A data out port

For a RAM, the write-enable port dictates whether we are writing data into the RAM or reading data from the RAM.



Testbench for the RAM has been included in the submission.



The simulated waveform for the RAM

REGISTER

This VHDL module implements a synchronous register that loads data on the rising edge of the clock and can be reset to zero when the reset signal is active. The current value of the register is always available at the output, providing a simple and common building block for digital circuits. It consists of the following ports :

1. A clock in port
2. A reset in port
3. A data_in port
4. A data_out port

The utilization of this synchronous register is favored over direct signal assignments to ensure synchronization with the clock. This approach is crucial to prevent any unexpected or unstable behaviors in the circuit's operation. Given that our application was already functioning synchronously, the integration of this register into our code was not required.

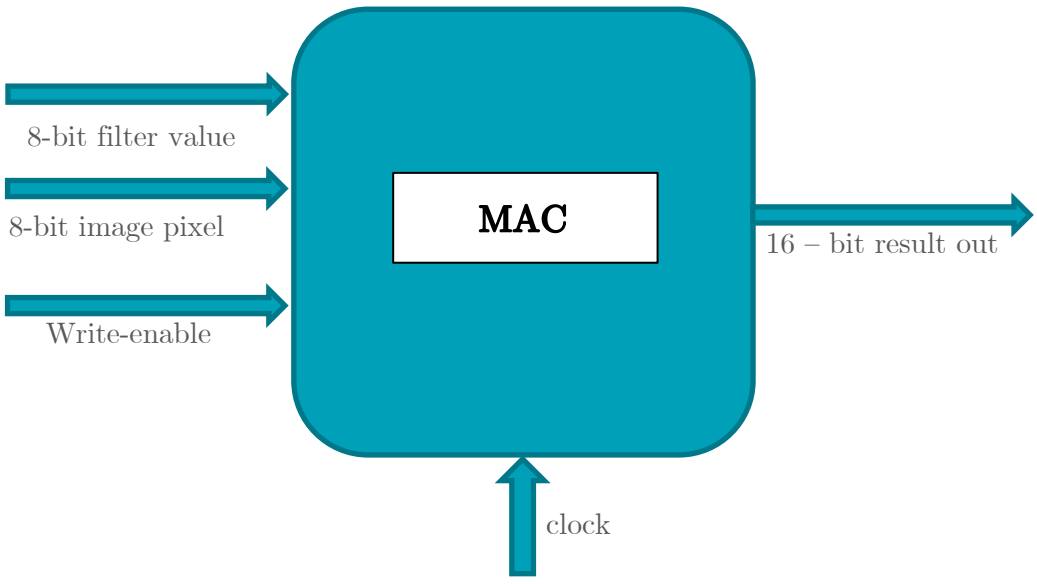
MULTIPLIER-ACCUMULATOR BLOCK

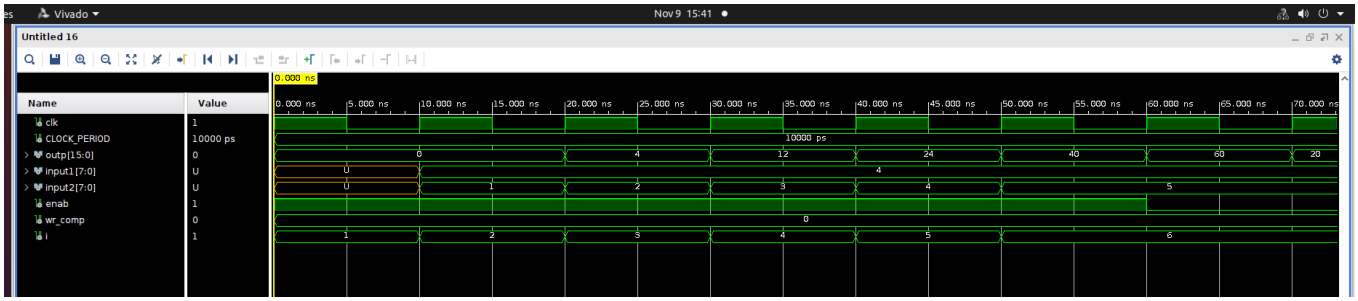
The MAC unit performs simultaneous multiplication and accumulation operations. In our implementation of the MAC, it consists of 5 ports:-

1. Two data in ports
2. A write-enable port
3. A clock in port
4. A data out port

The VHDL MAC module multiplies two 8-bit vectors, accumulates the result on clock edges, and outputs the 16-bit accumulated value. The write enables signal controls whether to accumulate (we = '1') or load the multiplication result into the accumulator (we = '0').

To ensure that the accumulated result is set to 0 after calculations done for each pixel, we give the two data in ports 0 values and turn the write enable to 0 this ensure that the result is reset to 0 to allow further calculations to be done.





The simulated waveform for the MAC testbench

In the testbench we have added 0,4,8,12,16,20 and return the final accumulated value of 60. Once write enable is switched to 0 the current product gets mapped to the accumulated result and result gets the value 20 which is the current product value.

Testbench for the MAC was created to verify its proper functionality. The same is included in the submission.

Applying the Filter

The image we are given is on an 8-bit grayscale. That is, each pixel represents a value between 0-255. Therefore, our job is to apply the filter and then ensure that the final output pixel value also lies in the same range for it to be displayed correctly. This is done in 2 steps.

Step 1: Calculate the pixel value:

In this assignment, we applied the following formula to get the output image.

$$\begin{aligned} O(i,j) = a*I(i-1,j-1) + b*I(i-1,j) + c*I(i-1,j+1) \\ +d*I(i,j-1) + e*I(i,j) + f*I(i,j+1) \\ +g*I(i+1,j-1) + h*I(i+1,j) + i*I(i+1,j+1) \end{aligned}$$

Here, O(i,j) represents the output pixel at the i,j location, and the alphabets represents the values stored in the filtered image. For the corner cases when the indices go out of bounds, the values have been taken as 0.

Step 2: Image normalization

Since the values calculated after applying the filter can go out of range, we need to ensure that the output pixel value fits in the 0-255 range so that it can be stored in an 8-bit unsigned format. This is done by linear normalization using the following formula:-

$$New\_I(i,j) = (I(i,j) - old\_min) * \frac{new\_max - new\_min}{old\_max - old\_min} + new\_min$$

For us the new_max = 255 and the new_min = 0 therefore giving us the following:

$$New\_I(i,j) = (I(i,j) - min) * \frac{255 - 0}{max - min} + 0$$

Since the only way to calculate the old_max and old_min values is by calculating each pixel value first, we had to, in essence, run the same code twice first to compute and store the maximum and minimum values, then again to store the data into the ram by first normalizing it with the max and min values.

FINITE STATE MACHINE(FSM)

Finite State Machines (FSMs) are powerful models in digital design. Comprising states, transitions, and inputs, FSMs excel at representing sequential logic. They find widespread use in control systems, protocol implementations, and hardware design, simplifying complex behaviors by breaking them down into manageable states and transitions.

The FSM manages data flow in our system, retrieving information from ROMs, delivering precise data to the MAC, writing results to RAM, and connecting with the RAM with the VGA Display module for image presentation. This streamlined process ensures efficient and cohesive operations throughout, facilitating effective data processing and display on the monitor.

The FSM consists of 3 states idle, processing and display.

1. **Idle -** This is the default state of the system. When we press a button then the system is triggered to start the filter and display process.
2. **Processing -** The FSM reads from the ROM , applies the filter , normalizes the data, and writes to RAM in this process. It further has 5 process_states that perform specific operations.
    a. **read_rom -** In this substate the addresses are provided to image rom , filter rom and ram. Then the calculate process_state is called.

b. **calculate** - In this process_state data is taken from the filter rom and image rom and sent to the MAC for accumulation. Then again read_rom is called with updated filter position. Once the filter position reaches the value of 8 (starting from 0 and incremented by 1) means that all the 9-pixel values have been added to the MAC then either calculate_min_max or write_ram process_states are called depending on whether min_max_done is 0 (Meaning the min pixel and max pixel values have not yet been found) or 1(Meaning the data is ready to be normalized and written to ram).

c. **calculate_min_max** - In this state data is extracted from the MAC and compared with the current min and max values to check if they need to be updated. Then go_next state is called.

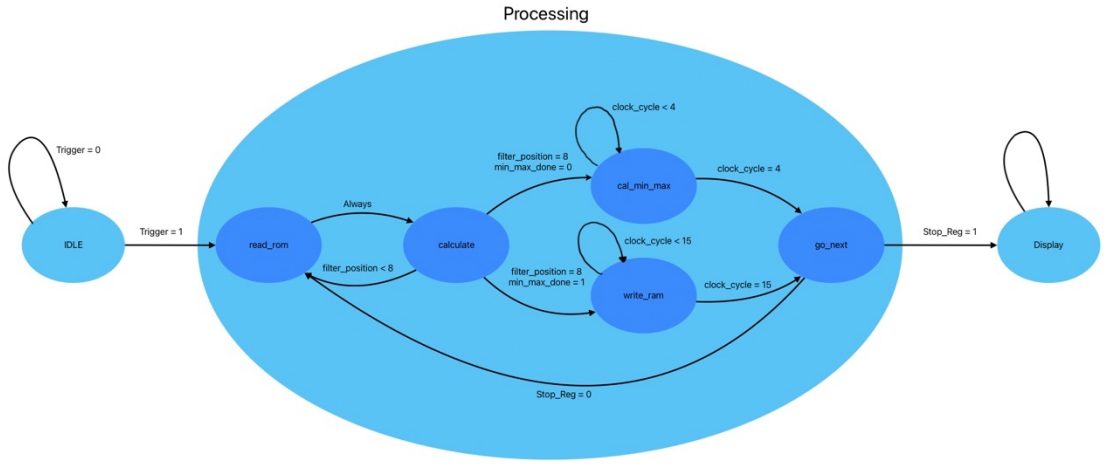
d. **write_ram** - In this state, after extracting data from the MAC, data is normalized and written into RAM. After this go_next state is called.

e. **go_next** - In this state the value of i and j are updated. If the value of i and j both become 63 then if :
   i. **min_max_done = 0** : min_max_done is set to 1 and read_rom is called. Meaning that minimum and maximum pixel values have been calculated.
   ii. **min_max_done = 1** : stop_reg is set to 1 indicating that the RAM has been populated and the system is ready to move into the Display state.

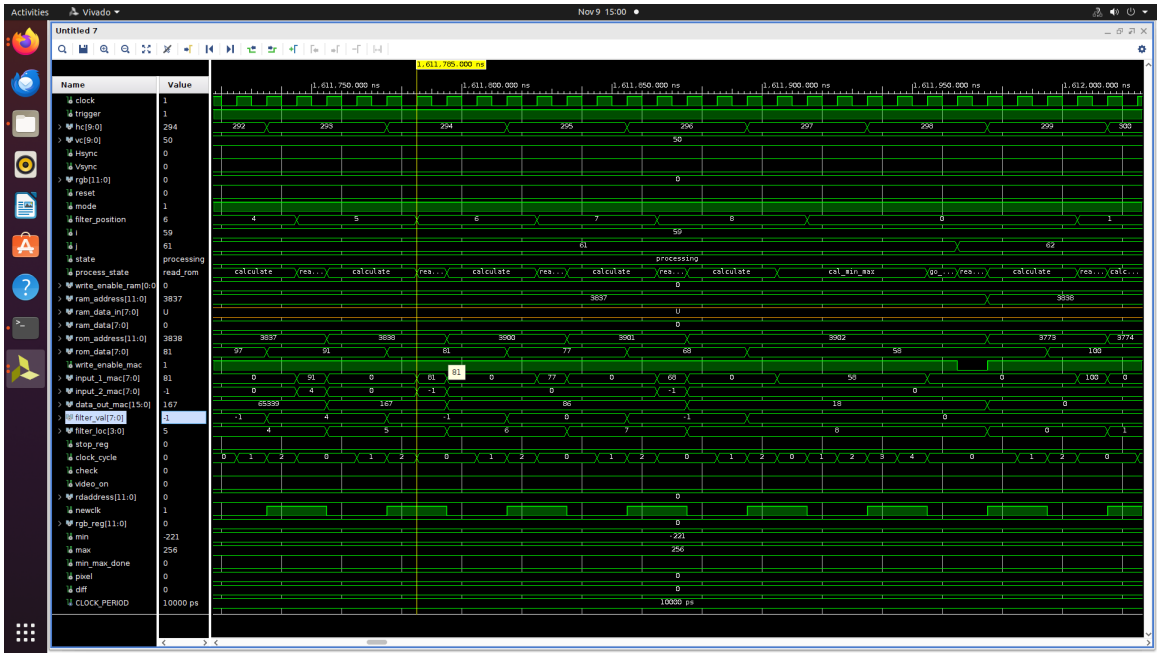
   Otherwise read_rom process_state is called.

3. **Display** – In this state FSM calls the VGA Display Module to feed the correct memory address to the ram module and ensures the correct image is displayed by sending the correct Hsync and Vsync values.

A state transition diagram has been attached below for better understanding of the various states and working of the FSM.



Finite State Machine Diagram

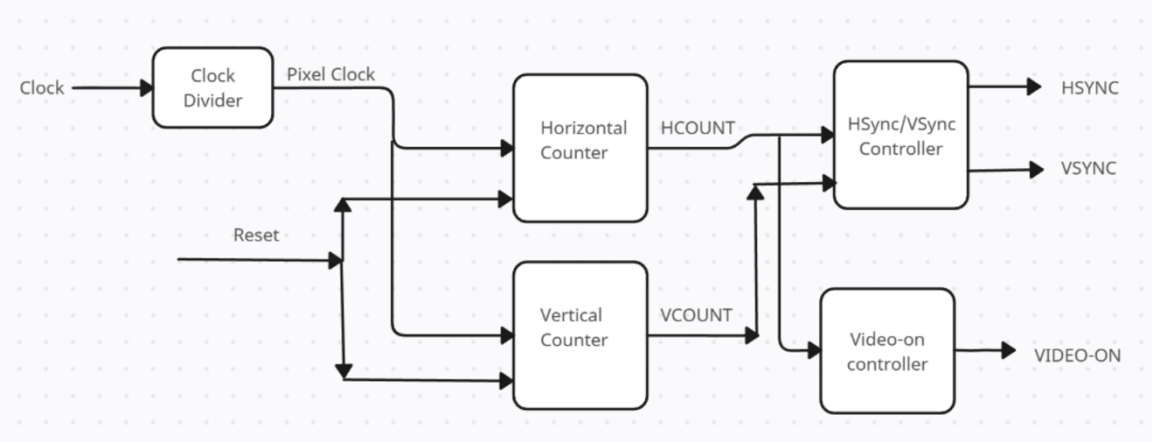


The simulated waveform when in cal_min_max process state

Displaying from the RAM using VGA

Making the VGA controller module:

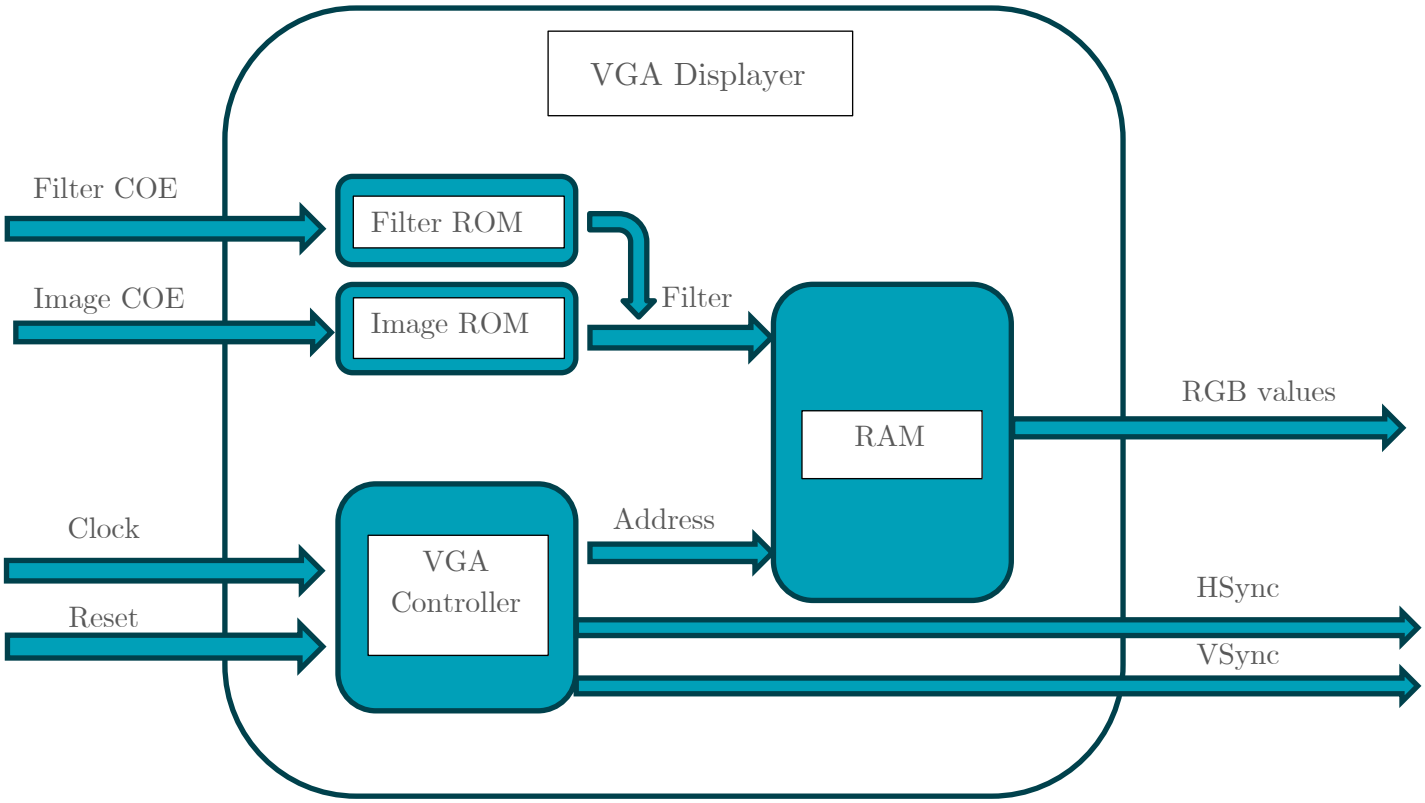
Firstly, we need a VGA controller module that controls the movement of x and y coordinates on the screen. Since the VGA display requires a slower clock (25 MHz) than the one provided (100 MHz), we also implemented a clock divider. We also need to maintain the *HSync* and *VSync* values for the display. A basic block diagram for the module is below:

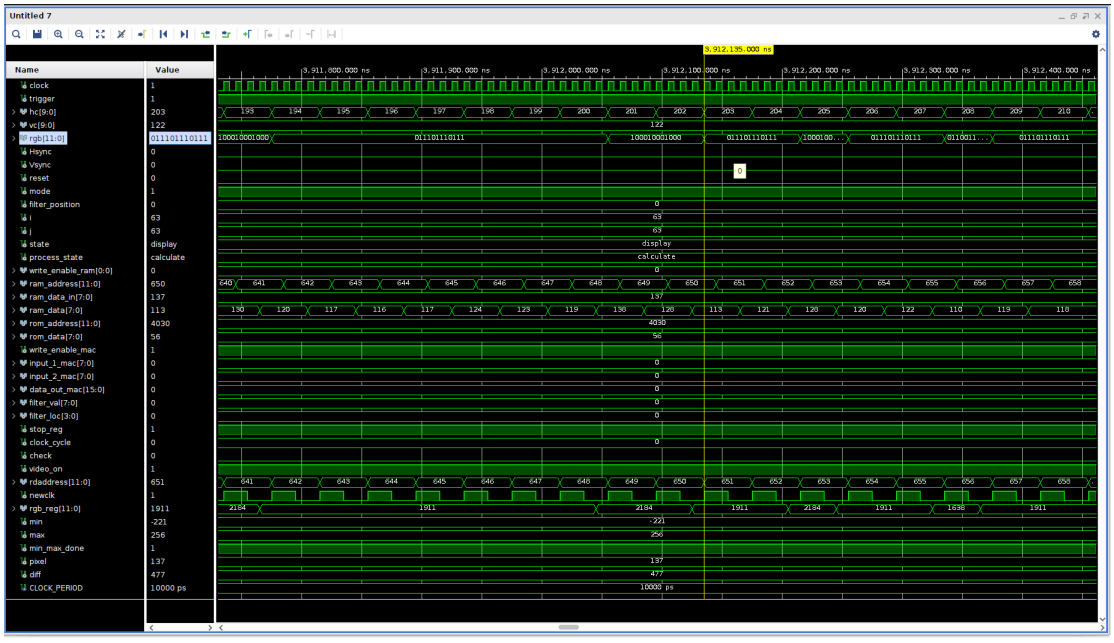


The *video-on* signal controls when we need to display our image and when we need to display the default colour. The resulting address can be found by the x and y coordinates, i.e., the *HCOUNT* and *VCOUNT* values.

Linking the controller to memory:

The VGA controller is now linked to the memory in the final module, which encapsulates all the above components. It is directly connected to the Basys3 board. It derives its inputs from the board and outputs to the board's ports. The address obtained from the x and y values returns the corresponding stored data to the output port containing the *RGB* values. Our design first allows all data to be read from the ROM, then the filter is applied, and the resulting image is stored in the RAM. Once this process is over, the display process is started, and the data is returned to be displayed. The final block diagram is drawn below.

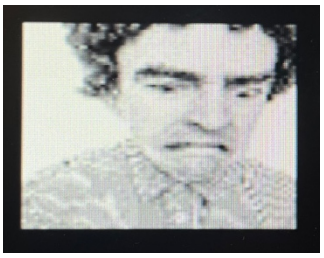




The simulated waveform for the FSM testbench during display state

OUTPUT IMAGES

Face:



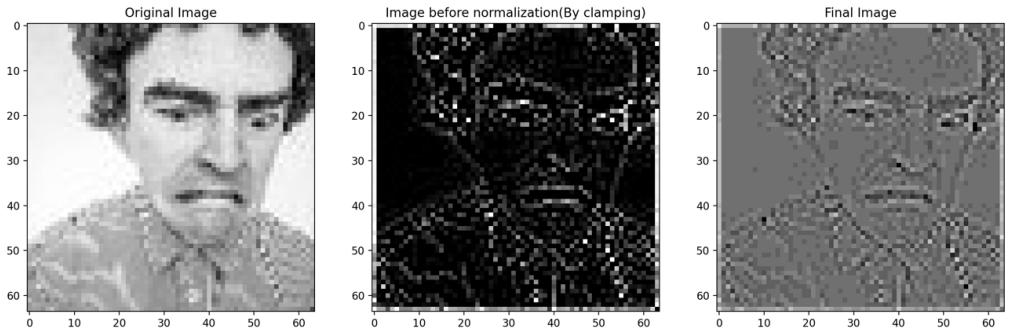

Coins :


Lighthouse :

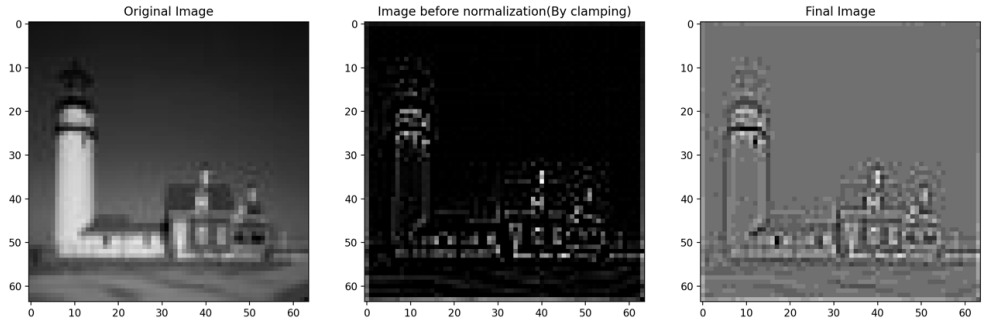

We created a python code using NumPy and matplotlib to get a sense of how the final images will look. Below are the images we obtained using Python. They match with the images we got on display.

THE EXPECTED OUTPUT IMAGES CREATED USING PYTHON

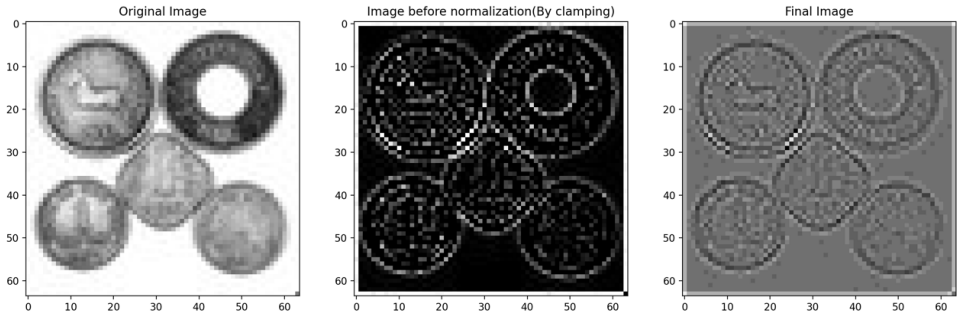
Face:



Light House:



Coins:



All source codes and utilization reports are attached with the report.

REPORT ENDS HERE