VLSI Design Lab - EE705

Project Report

Edge Detection using LOG on FPGA

Instructor : Prof. Sachin B. Patkar

Submitted By –

Mohd. Faizaan Qureshi	(203070062)
Kanak Vijay	(203070050)
Yogesh S Lahane	(203074003)
Umesh Nagar	(203070104)

1. Introduction

Edge Detection is an image processing technique for finding the boundaries of objects within images. It is very important and used for image segmentation and data extraction in areas such as image processing, computer vision and machine vision. It is widely used by self driving car companies to detect road and vehicles. And it can be done either in software or in hardware, but if we do in hardware specifically for this, then we can get good amount of accuracy and hardware can process more frames compared to softwares. In our project also, we do entire processing on FPGA. For this we tested the algorithm on python first and then we did hardware modeling of same algorithm with HDL.

2. Algorithm

We want to perform LOG which is Laplacian of Gaussian on image. So we can think of it as composite function , what we can do is, we can do by 2 methods :

First we can apply Gaussian function on image pixels and then we can apply Laplacian function of that.

Or, we can first compute Laplacian of Gaussian and then apply whole composite function to image pixels.

In this project, we use 2^{nd} option, first we calculated LOG filter and then we applied that filter on image. Application of the filter on image is called convolution, and this convolution is 2D , since image is also 2D array of pixels.

In python what we did =>

We use openCv library which is easy to use computer vision library, but that just to convert into grayscale.

All python Modules we need :

```
import numpy as np
import cv2
from PIL import Image
import matplotlib.pyplot as plt
from scipy import signal
```

And let's define some functions,

Gaussian function:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

In python, what we do is:

```
def gaussian(m, sigma):
    gaussian = np.zeros((m,m))
    m = m//2
    for x in range(-m, m+1):
            for y in range(-m, m+1):
                    #x1 = np.sqrt(2*np.pi) * sigma
                    x2 = np.exp(-(x**2 + y**2)/(2*(sigma**2)))
                    #gaussian[x+m, y+m] = (1/x1)*x2
                    gaussian[x+m, y+m] = x2
                    #gaussian = gaussian/np.sum(gaussian)
    return gaussian
```

Here m is order of Gaussian matrix, and sigma is function parameter.

To make a normalized Gaussian matrix, what we do is , we call function and divide it by sum of all elements :

```
g = gaussian(5,30)
g = g/np.sum(g)
```

Now, Laplacian function :

Laplacian operator is very well known in this field , i.e. people use universal operator.

$$\triangle = \nabla \cdot \nabla = \nabla^2 = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

There are 2 Laplacian operator , positive laplacian operator and negative laplacian operator.

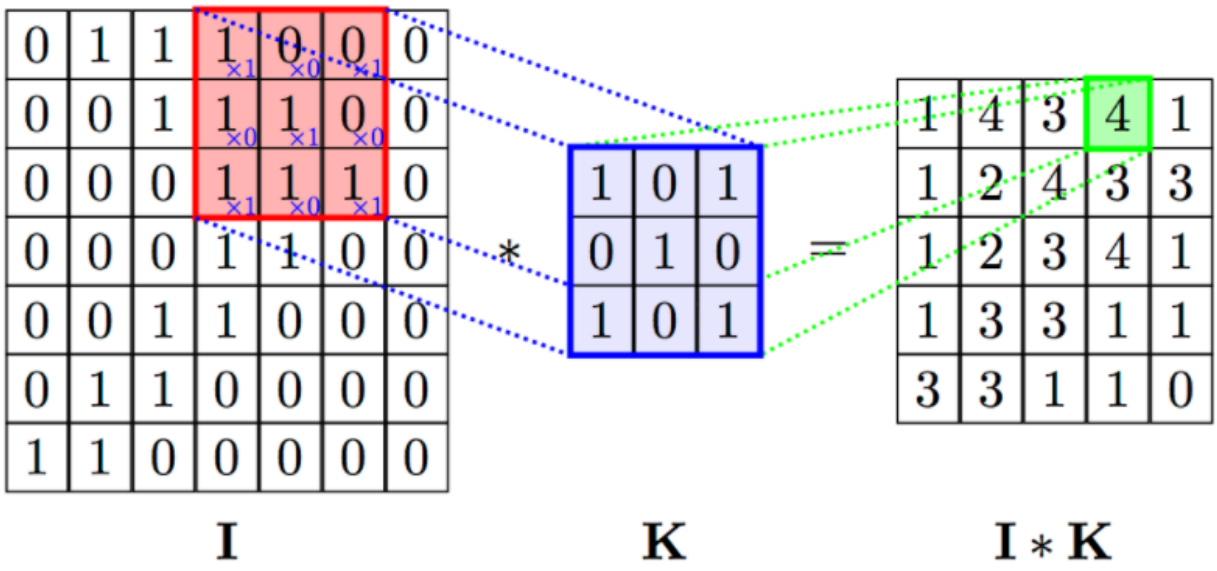
Positive Laplacian operator basically helps in finding outer edges and negative Laplacian operator helps in finding inner edges.

So, we define them in python as :

```
pos_laplacian_operator = np.array([[0, 1, 0],
                                   [1, -4, 1],
                                   [0, 1, 0]])

neg_laplacian_operator = np.array([[0, -1, 0],
                                   [-1, 4, -1],
                                   [0, -1, 0]])
```

Then we can find composite function of Laplacian and Gaussian matrix, by 2D convolution :

How we do convolution is , we take 2nd matrix, and slide it over 1st matrix :



Then we will get following LOG filter or kernel :

```
[[ 0.00000000e+00  3.99111753e-02  3.99777494e-02  3.99999654e-02
   3.99777494e-02  3.99111753e-02  0.00000000e+00]
 [ 3.99111753e-02 -7.96892025e-02 -3.99554223e-02 -3.99776753e-02
  -3.99554223e-02 -7.96892025e-02  3.99111753e-02]
 [ 3.99777494e-02 -3.99554223e-02 -8.88640979e-05 -8.89629181e-05
  -8.88640979e-05 -3.99554223e-02  3.99777494e-02]
 [ 3.99999654e-02 -3.99776753e-02 -8.89629181e-05 -8.90618207e-05
  -8.89629181e-05 -3.99776753e-02  3.99999654e-02]
 [ 3.99777494e-02 -3.99554223e-02 -8.88640979e-05 -8.89629181e-05
  -8.88640979e-05 -3.99554223e-02  3.99777494e-02]
 [ 3.99111753e-02 -7.96892025e-02 -3.99554223e-02 -3.99776753e-02
  -3.99554223e-02 -7.96892025e-02  3.99111753e-02]
 [ 0.00000000e+00  3.99111753e-02  3.99777494e-02  3.99999654e-02
   3.99777494e-02  3.99111753e-02  0.00000000e+00]]
```

This is 7 x 7 matrix, and also values in this matrix are float64 , i.e. 64 bit floating point number. We can further convert this into float 16 which basically we used in HDL:

```
array([[ 0.000e+00,  3.992e-02,  3.998e-02,  4.001e-02,  3.998e-02,
         3.992e-02,  0.000e+00],
       [ 3.992e-02, -7.971e-02, -3.995e-02, -3.998e-02, -3.995e-02,
        -7.971e-02,  3.992e-02],
       [ 3.998e-02, -3.995e-02, -8.887e-05, -8.899e-05, -8.887e-05,
        -3.995e-02,  3.998e-02],
       [ 4.001e-02, -3.998e-02, -8.899e-05, -8.905e-05, -8.899e-05,
        -3.998e-02,  4.001e-02],
       [ 3.998e-02, -3.995e-02, -8.887e-05, -8.899e-05, -8.887e-05,
        -3.995e-02,  3.998e-02],
       [ 3.992e-02, -7.971e-02, -3.995e-02, -3.998e-02, -3.995e-02,
        -7.971e-02,  3.992e-02],
       [ 0.000e+00,  3.992e-02,  3.998e-02,  4.001e-02,  3.998e-02,
         3.992e-02,  0.000e+00]], dtype=float16)
```

Now we have our LOG kernel or filter, now, let's apply LOG to image.

Here we apply LOG filter to grayscale image only, not RGB image, because RGB image is 3D image, i.e. 3 channels , so more computation, edge detection is all about intensity variation, and that we can capture in grayscale also without any loss.

Results we got :

This above is webcam data and live processing.

We can tune sigma values and size of filter i.e. m, to get different filters and may be we get more noises.

Now let's move to HDL part :

We have several submodules, what we do is we make constant ROM array to hold values of this LOG array. But we need floating point values, so we convert this float64 no. into binary 16 bit half precision floating point no:

```
type img_1d_vec_float is array(Natural range <>) of std_logic_vector(15 downto 0);

constant log_rom : img_1d_vec_float(0 to (m*n)-1) := (
   "0000000000000000", "0010100100011100", "0010100100011110", "0010100100011111", "00101001000
);
```

then we made submodules float_mul.vhd that performs multiplication of 2 floating point numbers. Entity of which looks like :

```
entity float_mul is
   generic (
   len : integer := 16;
   bits : integer := 4
   );
   port(
      clk : in std_logic;
      enable : in std_logic;
      A, B : in std_logic_vector(len-1 downto 0);
      result : out std_logic_vector(len-1 downto 0);
      ready : out std_logic
   );
end entity;
```

Further code is given in the file, this will do multiplication in 1 cycle, we could do it in multiple cycles , but to keep things simple we made it multiplicating in 1 cycle , and the result is verified upto 50MHz clock frequency tested on Labsland.

Another module is float_adder.vhd, which do floating point addition in 7 states :

```
type t_state is (RDY, START, NEGPOS, OP, SHIFT, WRT, RSLT);
signal state : t_state := RDY;
```

Entity of this module look like :

```
entity float_adder is
   generic (
      size : integer := 16;
      logsize : integer := 4
   );
   port (
      clk : in std_logic;
      enable : in std_logic;
      num1, num2 : in std_logic_vector(size-1 downto 0);
      result : out std_logic_vector(size-1 downto 0) := (others => '0');
      ready : out std_logic
   );
end entity;
```

Now apart from these modules, we also need module which convert image pixels to floating point numbers, since image pixels are 8-bit unsigned number, but in order to multiply image pixels with LOG ROM values which are basically 16 bit floating point number, so to multiply them, we have to convert image pixels into floating point numbers, so for that we made "convert_int_to_float1.vhd" module :

```
entity convert_int_to_float1 is
   port(
      clk : in std_logic;
      enable : in std_logic;
      A : in std_logic_vector(7 downto 0);
      B: out std_logic_vector(15 downto 0)
   );
end entity;
```

Another module which we need is "convert_float_to_int.vhd" to convert back into unsigned 8 bit integers, then that data we can directly give to VGA (also expect 8 bit no), and can also give to python for display purpose. :

```
entity convert_float_to_int is
   port(
      clk : in std_logic;
      enable : in std_logic;
      A : in std_logic_vector(15 downto 0);
      B: out std_logic_vector(7 downto 0)
   );
end entity;
```

Here we can ignore mantissa part , because we don't need to worry about less value or more value, since after edge detection processing, we get only combination of white and black pixels only, so less value would only mean to have less pixel value or dim color only.

Then we have another module which basically takes all modules and do :

1) First we have to convert integer values of a particular 7x7 submatrix of image matrix into float values, so what we did is, we used 49 such modules to convert 49 integer 8 bit values of image submatrix to 49 floating point values in single cycle.

2) First it do element wise matrix multiplication (not regular matrix multiplication). In our case LOG is 7x7 matrix, so we need to multiply 7x7 matrix from big image matrix with LOG matrix, then we will get 49 values. And in our case, we took 49 components of "float_mul" module to compute 49 multiplication in 1 single cycle.

3) Then we need to add those 49 values, but we use single adder to do this (because if use 48 adder modules then our circuit may become very huge, also resource can get exhausted). So use logics to add 49 values with single adder.

4) After addition we will get 1 single result in floating point form, so we need only 1 module to convert this result back into integer 8 bit unsigned number.

Entity of this module "mult_matrix_sum_float4.vhd" looks like :

```
entity mult_matrix_sum_float4 is
    generic (
--      index_i : integer := 0;
--      index_j : integer := 0;
        h : integer := 7;
        w : integer := 7
    );
    port (
        clk : in std_logic;
        mmsf_enable : in std_logic;
        index_i : in integer;
        index_j : in integer;
        m1 : in img_1d_vec(0 to ((row + (2*m_cap))*(col + (2*n_cap))) - 1);
        m2 : in img_1d_vec_float(0 to (h*w) - 1);
        --result : out std_logic_vector(15 downto 0) := (others => '0');
        result : out std_logic_vector(7 downto 0) := (others => '0');
        main_ready : out std_logic
    );
end entity;
```

This module basically generates 1 pixel of output, so we have to give it index to tell it which index we are talking about and how much kernel you have to slide.

So, we made one 7x7 matrix from whole big image matrix, based on index. Let's say I have image of size 640x480, so if I say index 0 of output, that means I will

take image[0 to 7, 0 to 7],here I explained the intuition, but we are actually dealing with single array, not multidimensional array.

So, based on 2 index and width and height, we can find index for 1D array =>

```
process (m1, index_i, index_j)
   variable new_i1 : integer := 0;
begin
   new_i1 := (cw*index_i) + index_j;
   for i in 0 to m-1 loop
      for j in 0 to n-1 loop
         m11((i*n) + j) <= m1(new_i1 + cw*i + j);
      end loop;
   end loop;
end process;
-------------------------------------------------
```

Then based on argument we made above, we can make 49-49 modules in this way :

```
g1 : for i in 0 to m-1 generate
   g2 : for j in 0 to n-1 generate
      conv_i_f : convert_int_to_float1 port map (clk, cif_en, m11((i*n) + j), cif_output((i*n) + j));
      mult : float_mul port map (clk, prod_en, cif_output((i*n) + j), m2((i*n) + j), tmp((i*n) + j));
   end generate;
end generate;
```

State machine that take care of converting integer to float, multiplication, addition and convert back from float to integer :

```
process (clk)
begin
   if(rising_edge(clk)) then
      case (state) is
         when idle =>
            if(mmsf_enable = '1') then
               main_ready_signal <= '0';
               state <= start;
            end if;

         when start =>
            cif_en <= '1';
            prod_en <= '0';
            add_en <= '0';
            --cfi_en <= '0';
            state <= cif;

         when cif =>
            cif_en <= '0';
            prod_en <= '1';
            add_en <= '0';
            --cfi_en <= '0';
            state <= multiplication;

         when multiplication =>
            cif_en <= '0';
            prod_en <= '0';
            add_en <= '1';
            --cfi_en <= '0';
            state <= addition;


   when addition =>
      if((dut_ready = '1') and (k=h*w)) then
         --main_ready_signal <= '1';
         add_en <= '0';
         --cfi_en <= '1';
         --cfi_A <= output;
         --cfi_A <= inter_result;
         state <= cfi;
      end if;

   when cfi =>
      --cfi_en <= '0';
      result <= cfi_B;
      main_ready_signal <= '1';
      state <= idle;
```

Then we have “con2d1.vhd” module that increase the index_i and index_j signal on getting ready signal from “mult_matrix_sum_float4.vhd” module, so in this way, we will get all processed pixels of same size of image i.e. w*h.

Entity for “conv2d1.vhd” looks like :

```
entity conv2d1 is
   generic (
      row : integer := 10;
      col : integer := 10;
      m : integer := 7;
      n : integer := 7
   );
   port (
      clk : in std_logic;
      img : in img_1d_vec (0 to (row*col) - 1);
      mask : in img_1d_vec_float(0 to (m*n) - 1);
      filtered_img : out img_1d_vec (0 to (row*col) - 1) := (others => (others => '0'));
      conv2d_ready : out std_logic
   );
end entity;
```

Here we have to make slight bigger matrix , which we call “canvas” . It is bigger because to get (0,0) index of output, then we need image sub-matrix somewhat above, so that center of that sub-matrix coincides with position (0,0) of output matrix, so we need to to take extra boundary across image and initialize them with zeros, so canvas initialization looks like :

```
canvas_initialization : process (img)
begin
   for i in m_cap to (row + m_cap - 1) loop
      for j in n_cap to (col + n_cap - 1) loop
         canvas((i*cw) + j) <= img(((i-m_cap)*(col)) + (j-n_cap));
      end loop;
   end loop;
end process;
```

Here also we will need FSM to control the flow. This FSM has 3 state only.

```
type t_state is (idle, start, fin);
signal state : t_state := idle;
```

FSM will look like :

```
process (clk)
begin
   if (rising_edge(clk)) then
      case state is
         when idle =>
            --this is important
            mmsf_enable <= '1';
            index_i <= 0;
            index_j <= 0;
            conv2d_ready <= '0';
            state <= start;
```

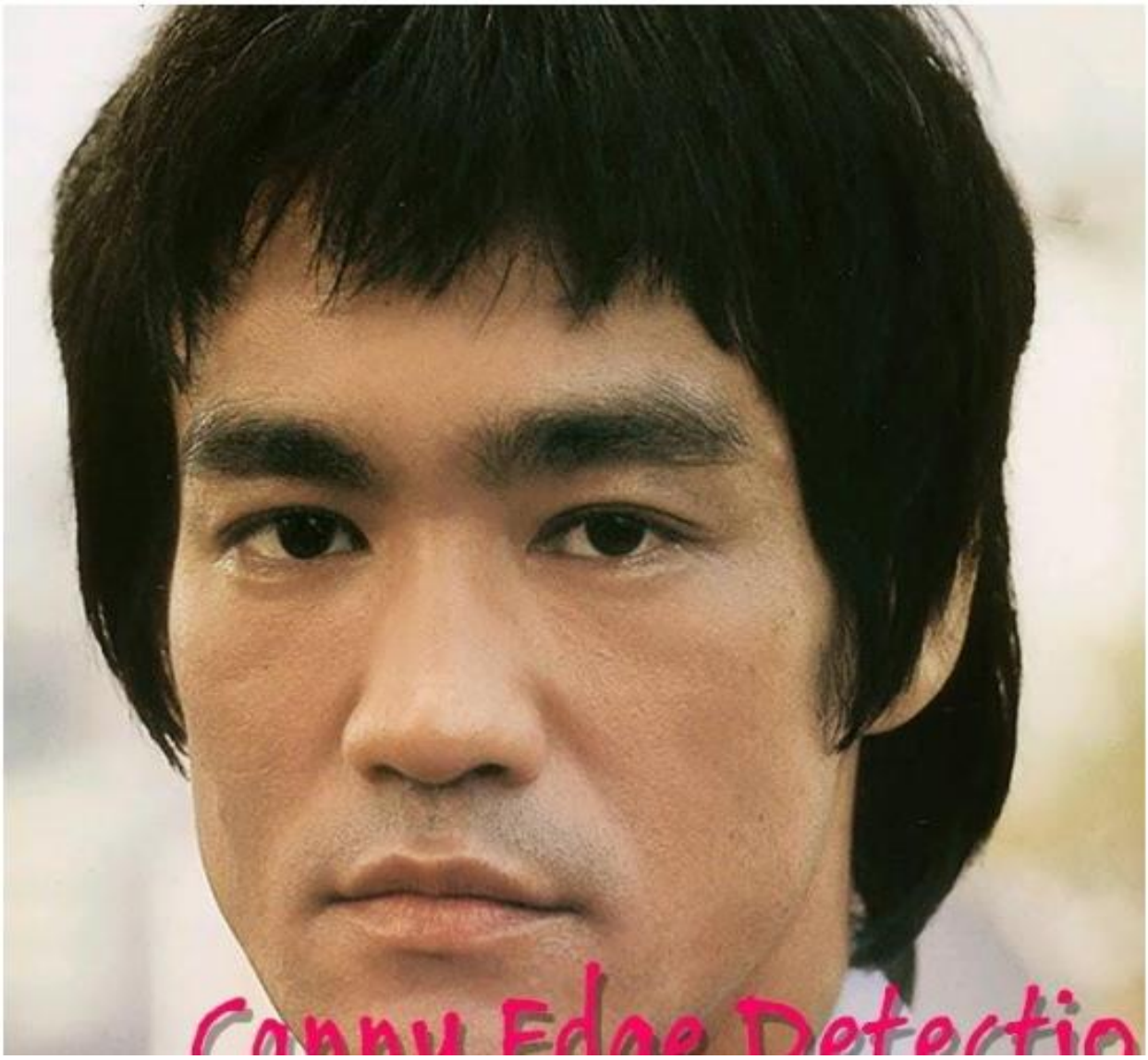
```
            when start =>
               if (ready='1') then
                  if (index_i < row) then
                     if (index_j < col) then
                        index_j <= index_j+1;
                     end if;
                     if (index_j = col-1) then
                        index_j <= 0;
                        index_i <= index_i + 1;
                     end if;
                  end if;
                  if ((index_i=row-1) and (index_j=col-1)) then
                     mmsf_enable <= '0';
                     conv2d_ready <= '1';
                     state <= fin;
                  end if;
               end if;

            when fin =>
               state <= idle;

         end case;
      end if;
   end process;
```

3. Simulation

Now let's simulate this.

1) Let's do the simulation for 100 x 100 image.

I have chosen this image from google :



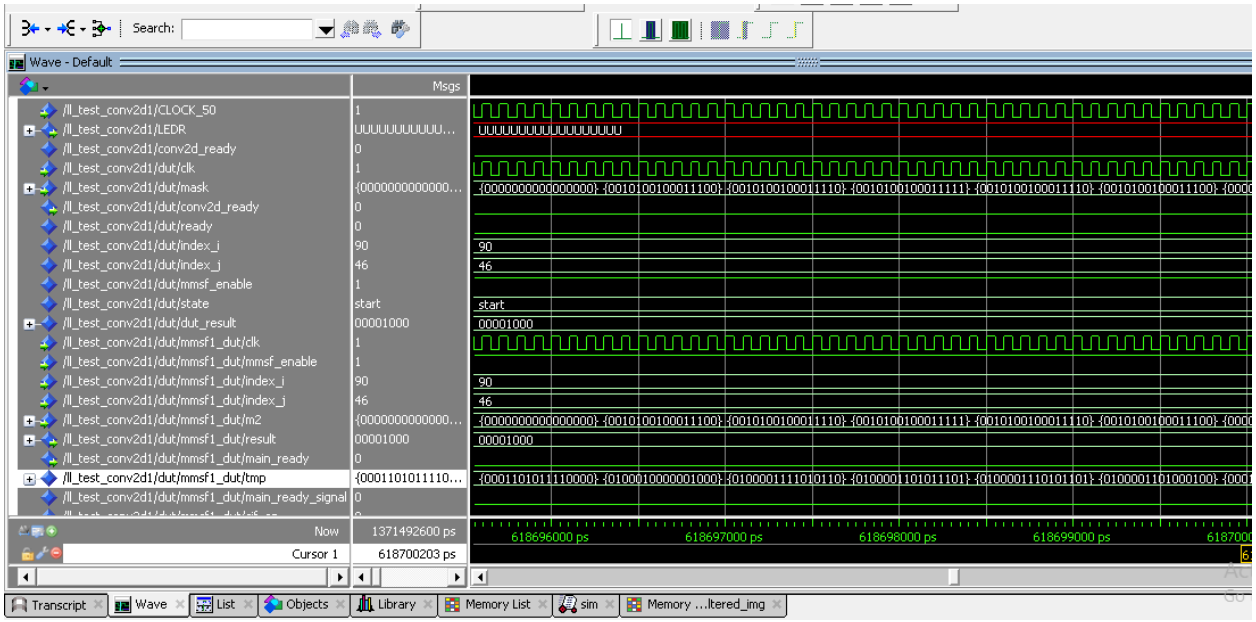
Now, let's first of all, convert this image into .mif files which VHDL can read, so we have created a python file to do this. I gave this file a name "img_8b_100s.mif" , which looks like this =>

Now we opened this file in VHDL and initialized the image signal, =>

```
constant ImaGE_FILE_NAME : string := "F:\python_projects\python essentials\edge_detection_project_log\img_8b_100s.mif";

subtype mem_type is img_1d_vec (0 to (row*col) - 1);

impure function init_mem(mif_file_name : in string) return mem_type is
    file mif_file : text open read_mode is mif_file_name;
    variable mif_line : line;
    variable temp_bv : bit_vector(7 downto 0);
    variable temp_mem : mem_type;
begin
    for i in mem_type'range loop
        readline(mif_file, mif_line);
        read(mif_line, temp_bv);
        temp_mem(i) := to_stdlogicvector(temp_bv);
    end loop;
    return temp_mem;
end function;
```

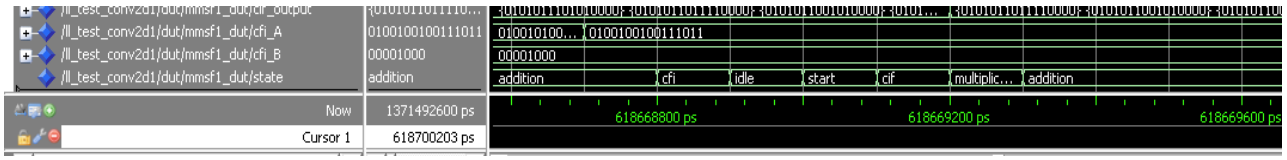
We have used impure function here, which is doing file handling in proper way.

We are getting this in modelsim as waveforms :



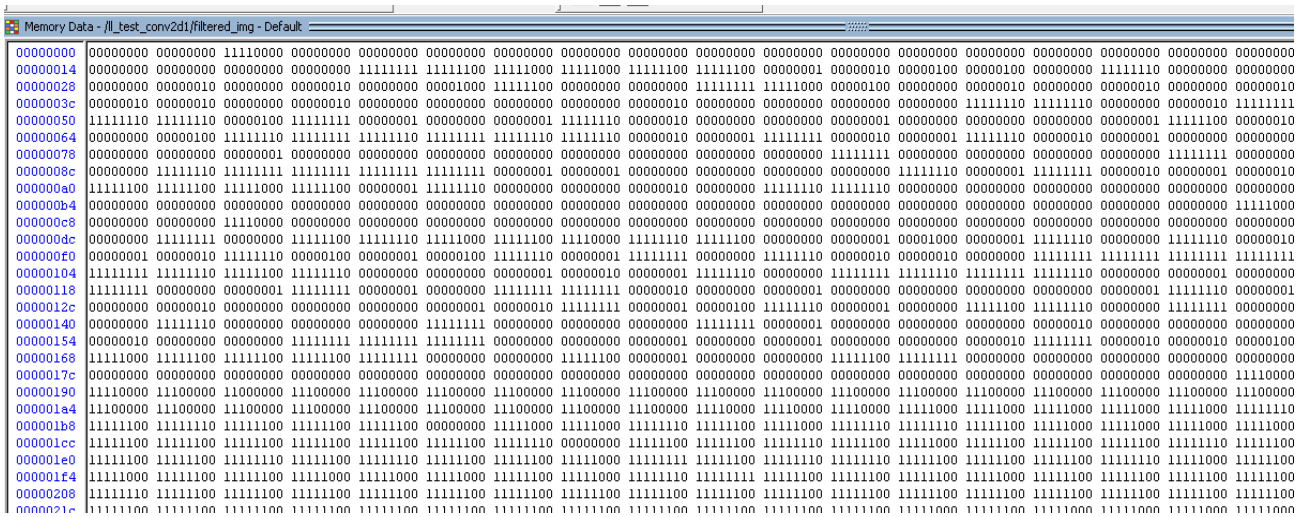
Let's see some useful things in this waveform =>

States of "mult_matrix_sum_float4.vhd" are clearly shown here, and changing as desired =>



We are enabling the 4 modules namely = "convert_int_to_float1", "float_mul", "float_adder", "convert_float_to_int" in a sequential manner, in this order only, according the flow of processing of pixels.

After it run completely, we are getting this result :=



But to make sense of this data, we will give this values to python for display purpose only, by using one more python file "dec_pixel_img_to_show.py"

But quality of result is not so good, but we can't expect more than that because this is just 100 x 100 resolution.

Now, let's simulate 200 x 200 version of same image.

We have followed same procedure , and we got :=>

4. Future Scope of this project =>

We can implement UART for sending data from camera module to FPGA.

We can implement SDRAM and SDRAM controller to behave as a buffer for incoming data and store there and then from SDRAM, we can transfer data to VGA.

We can also perform Computer Vision tasks , since we have pixels in SDRAM for some time, like facial expression recognition, age prediction, more noise free edge detection.

5. Video URLs Folder Link :

https://drive.google.com/drive/folders/1fGGQZOQsRUYaXAGDSbTWK0guLCZJ8SPE?usp=sharing

6. References :

1) Implementation of Gaussian Based Marr Hildreth Edge Detection Algorithm on Reconfigurable Hardware, By Soudabeh Mirzataraj.

2) FPGA Based Design and Implementation of Improved Edge Detection Algorithm using LOG Operator, by Enzeng Dong, Kaifeng Li, Jigang Tong.

3) FPGA implementation of filtered image using 2D Gaussian filter, by Leila kabbai , Anissa Sghaier , Ali Douik and Mohsen Machhout.