University of Bourgogne

France

VHDL Project

Image Processing of Two Dimensional Filter

January 11, 2018

Submitted by :
Mr. Nayee Muddin Khan
DOUSAI
Mr. Yamid ESPINEL

Submitted to :
Dr. Julien DUBOIS

Contents

List of Figures

1 Objective

The objective of our report is to process the data flow from the required image using a two dimensional linear filtering. We will read the given lena image dat file which consists of the pixel values and we will process this with the help of cache memory and processing module. The resulted image is also used to apply different filters.

2 Introduction

Image processing is considered as one of the important and useful techniques in the field of computer vision. But we implement this technique in VHDL is not that simple as the other existing software's.

Every image we need for image processing consists of 2D matrix.Processing a 2D image in FPGA is not a good idea and its complicated. It will lead to excessive delays and resources. So we convert the 2D image into a linear 1D array which is stored as .dat file.

To test our results we will use the student version software ISE Design Suite from Xilinx. The student version of the software is downloaded from Xilinix website [1]. As we check our testbench results we can implement the same code on the hardware by using the Nexys 4 board which is a complete, ready-to-use digital circuit development board based on Artix-7 Field Programmable Gate Array (FPGA) from Xilinx which is represented in the figure1. We can have a look of the detailed description of the board from the website [2].

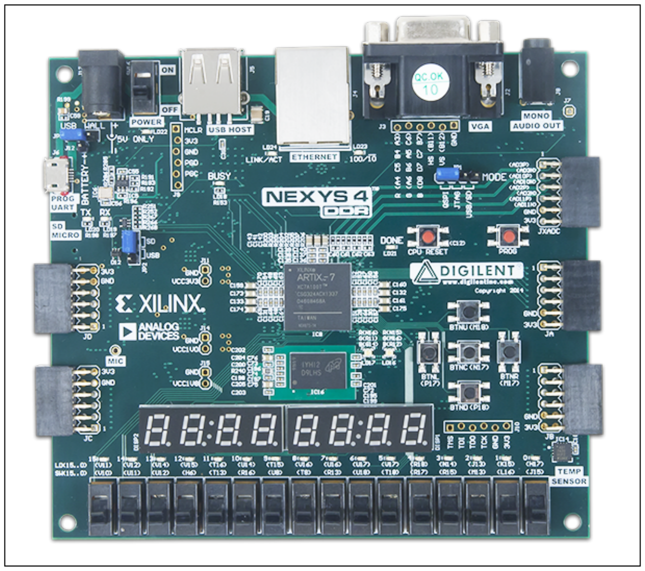


Figure 1: Nexys 4 FPGA Board

3 Implementation

Our project is divided in several modules, each of which handles a specific task to perform the processing of the image. We first list the modules we are currently using along with their purposes and then we explain in detail how they work. Like this, the modules that conform our project are:

1. **Cache memory:** This module is in charge of extracting the 3x3 neighborhoods from the image, taking as input a chain of bits coming from a .dat file and giving a new 3x3 neighborhood every clock period [3].
2. **Processing unit:** Here we take one 3x3 neighborhood coming from the Cache Memory module and perform 2D filtering by convolving with a predefined kernel. As a result, it will give a pixel which is the result of the convolution of the neighborhood with the kernel.
3. **Testbench module:** In this part we read the image file and we pass the chain of bits coming from it to the Cache Memory module, whose output is bridged directly to the Processing module. Once this last module is ready, the output file will be written with the processed pixels in a binary format.

As seen before, the project has been structured in a way that all the process is done efficiently and in the less amount of time possible. It's also a very flexible implementation which lets the user to add new processing features without affecting the previous made modules.

Now we continue by explaining the procedure followed to build the different modules and the strategies used to control the flow of data between them, which is a critical part when it comes to avoid loss of data or incorrect image reconstruction.

3.1 Cache Memory

The aim of this module is to extract a 3x3 neighborhood from an incoming flux of pixels for every new clock period as in figure 2. In order to gather the first top-left neighborhood, we have to receive all the data corresponding to the 2 first rows and the 3 first pixels of the third row, meaning that the pixels that are not part of the desired neighborhood have to be stored in a temporary memory. To do this, we followed the strategy given by the course director, which can be illustrated as in figure 3.

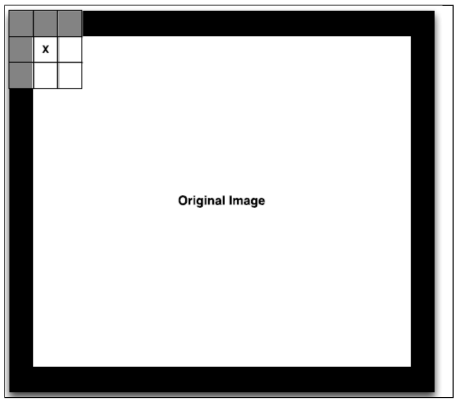


Figure 2: Neighbourhood boundaries of an image

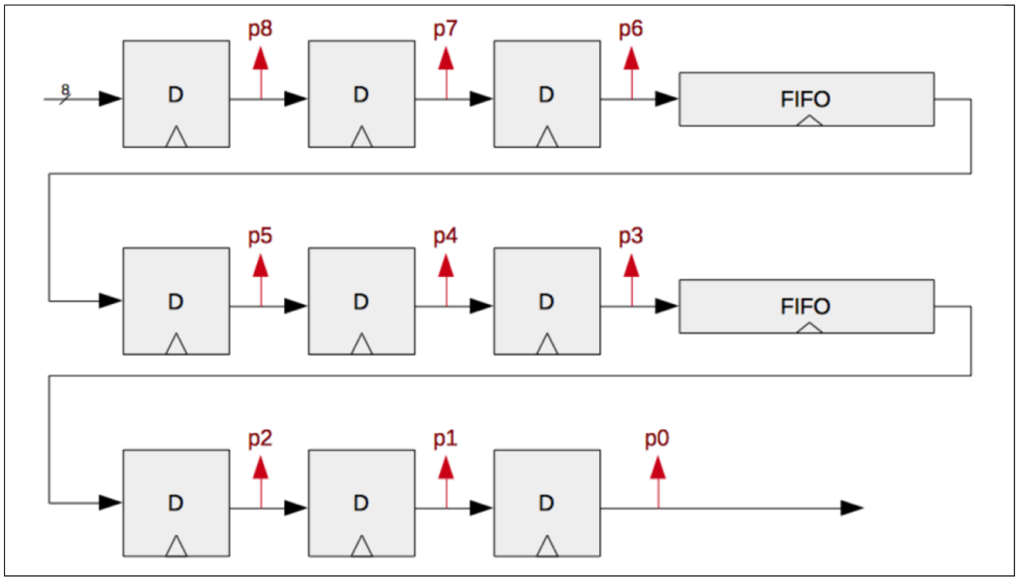


Figure 3: Cache memory structure

According to this approach, the pixels will come serially into the first top-left flip-flop and so, every clock period the pixels will be moving through the structure. Once the first three pixels are read, the next clock periods the FIFO module will start receiving pixels and so it will store them until it reaches an amount of 125; during this storage time the FIFO shouldn't give any output, but once it has reached the predefined threshold it will begin providing the stored pixels. As the flux of pixels never stops, the FIFO will be always full and keep giving data with a shift of 125 pixels between the third flipflop and the fourth flipflop.

This structure is implemented in VHDL language but it still requires of some control strategy that activates the reading and writing of the FIFOs, and also tells the Processing module when the first neighborhood is available. One way to do it is to implement a state-machine approach that sequentially activates each flipflop and FIFO according to the flux of pixels that are passing through. In our case, we have decided to implement the control structure shown in figure 4.

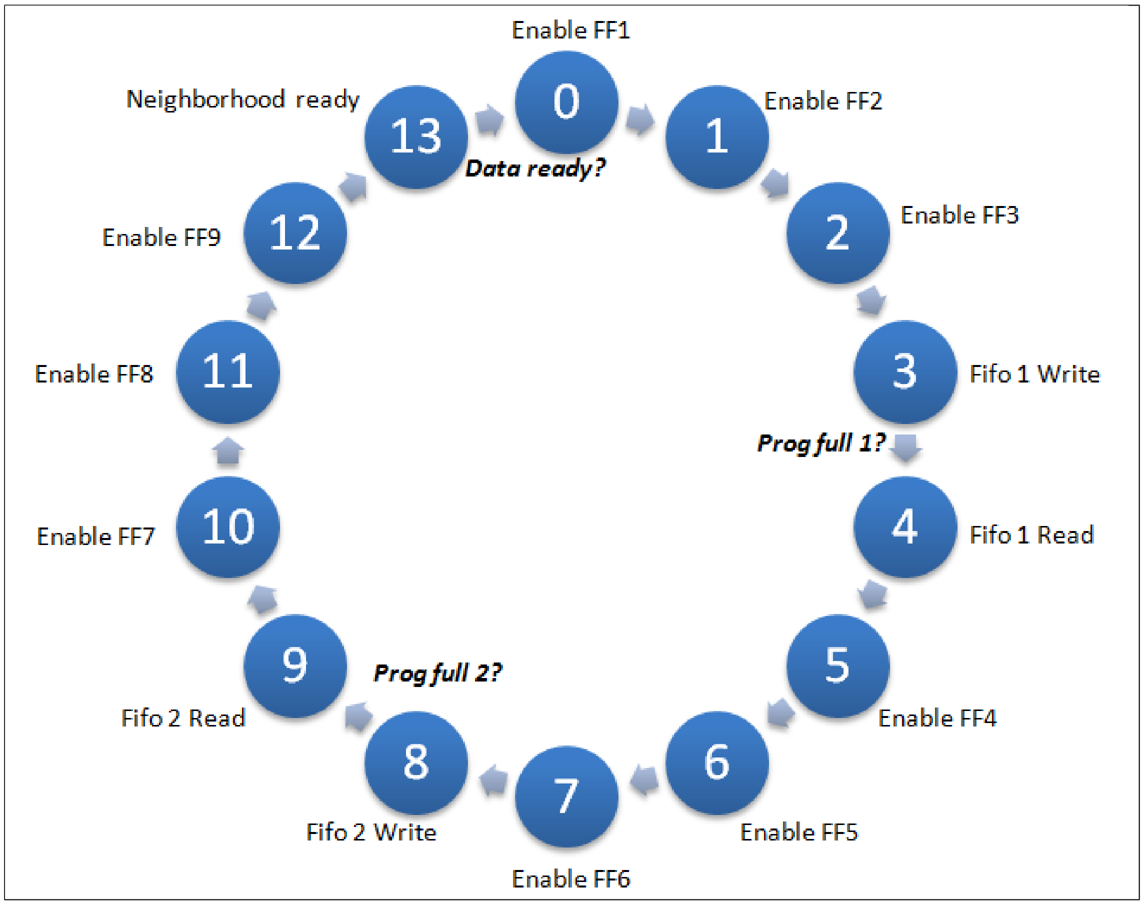


Figure 4: Control structure for Cache memory (FF = Flipflop)

As seen in figure 4, each state will activate a certain flipflop or Fifo, ensuring a correct flux of data and the extraction of the expected neighborhood pixels. For this we have created an array of enables (going from 0 to 13), activating each element in every state. In the case of the Fifo modules, a threshold of 125 has been defined, meaning that as soon as it has this amount of pixels it will begin providing the previously stored ones, ensuring that only the pixels we need will be available as outputs from the Flipflops. Once the process is completed, the first neighborhood will be extracted and so it will be ready to be taken. In this way, other processes like the processing unit will begin operating as soon as there is a neighborhood available, avoiding incorrect calculations because of dummy pixels and finally the saving of fake data in the resulting image.

3.2 Processing unit

In this part of the experiment, the nine pixels given by the Cache Memory module are going to be processed according to a chosen filter kernel. Taking into account that there will be a constant flux of pixels coming in, we have to implement a pipelined architecture that is able to take all the pixels and perform the processing operations in the most efficient way. One good approach is to take pairs of pixels and add them sequentially until all of them have been combined, so we implement this strategy (which was originally suggested by the course director) in the way shown in figure 5.

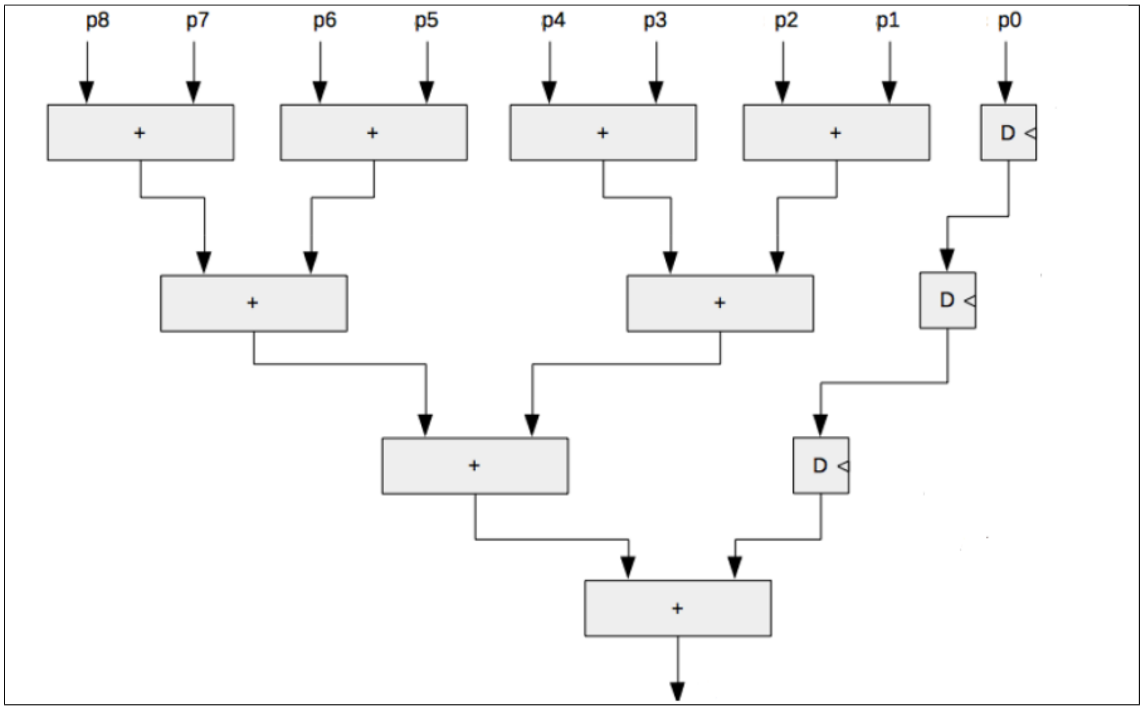


Figure 5: Base pipelined structure for neighborhood processing

To implement this pipelined structure in VHDL we first create an Adder module from the Xilinx's IP Core Generator assistant. Because with this approach we would be basically creating an average filter, our final result after adding all the 9 pixels will be a 12 bit number (255x9 = 2295 = 12bit). So, we set the parameters of the Adder accordingly and then instantiate this module 8 times in the processing unit; we also create a 12bit Flipflop for pixel 0 and instantiate it 3 times. This flipflop will help us to carry the pixel value through all the rows of the structure until its finally used.

If we check the general objective of the experiment, which is to do image processing with the help of a predefined kernel, we realize that this structure has to be extended with nine multipliers (one for every pixel) and one divider (to perform the final normalization). Then we make use of the multiplier and divider that come with the Xilinx software, remembering that the coefficients that we are going to use for the Sobel and gaussian filter may have high and/or negative values. To ensure a proper calculation we have selected a bus-width of 28 bit for all the elements (adders, flipflops and multipliers) and made them to work with signed type values.

Now we have to control the flux of data though the structure to ensure that all the new neighborhoods are processed as they come, and also to know when the first result is available.

We implement a similar strategy to the one made for the Cache Memory module, by building a state-machine algorithm that sequentially activates all the elements in the structure. As the multipliers don't have enable ports, there is the need of adding a row of Flipflops after the multipliers so we can be sure that we have their results. The structure after adding the required multipliers, Flipflops and dividers is shown in figure 6.

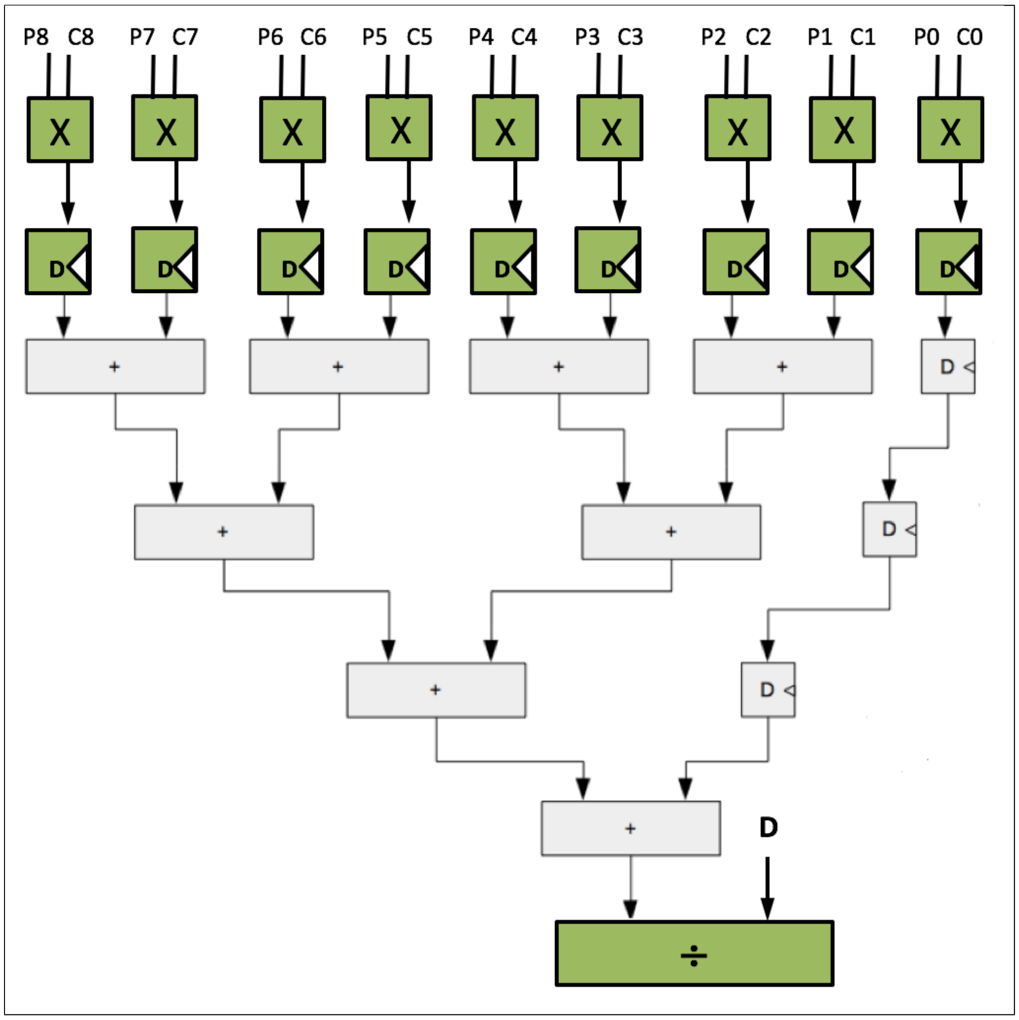


Figure 6: Complete processing unit: Green boxes represent the new components

As mentioned before, the state-machine to be built has to sequentially activate the components of every row in the structure until the final divider is reached. Like this, it will be able to trigger a flag indicating the availability of the first processed neighborhood. This control strategy is implemented as shown in figure 7.

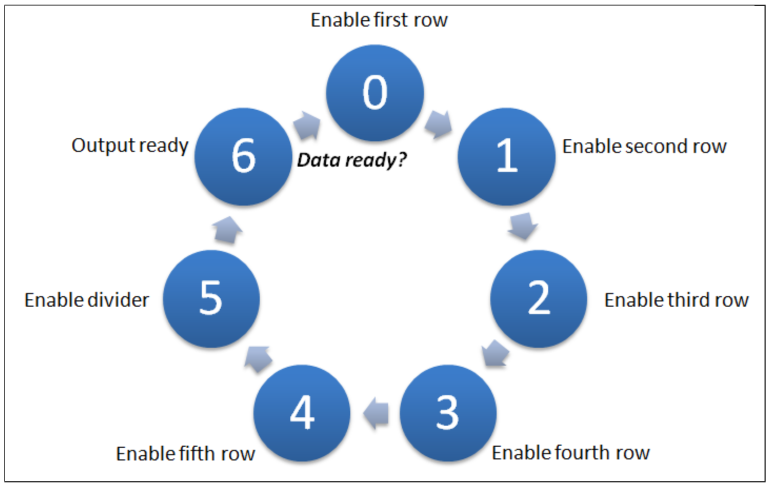


Figure 7: Control scheme for the processing unit

The state-machine shown in figure 7 controls the execution of every row of components in the Processing unit. The first row of components refer to the Flipflops located under the multipliers, the second row refer to the first line of 4 adders and the pixel 0 Flipflop, and so until the last enable which tells to an external process (the Testbench unit for our case) when the first result is available.

3.3 Testbench module

The aim of this unit is to connect the two previous modules so that, given some pixels read from an image file, it can write out the result of the processed neighborhoods coming from the Processing module. As this is a set of simulation instructions, it will perform clock generation, file reading and writing, and communication with other modules using the computer's processor instead of implementing directly on the FPGA board [4].

The testbench starts by generating a clock with a 20ns period; this clock will control all the processes in the program. Then, a binary lena image is read from the hard disk (one line per clock period) so each 8bit pixel is sent to the Cache Memory module every time a new line is read. After the Cache Memory has finished collecting the first neighborhood, it will warn the Testbench about it and give the corresponding 3x3 neighborhood.

After the first neighborhood is given, the following ones will come at every new clock period until all the pixels have been processed. These neighborhoods are then passed to the Processing unit, which will convolve the desired filter kernel with the given neighborhood. This unit is activated as soon as the first neighborhood is collected by the Cache Memory, and will give a completion signal once it has processed all the available neighbors, activating then a writing process that will keep saving the processed neighborhoods until the unit tells it to stop.

The output file is then a binary image as the input image, but will contain the processed pixels according to the selected filter. This final image is plotted in Matlab to validate the results and perform the respective corrections if there is need to.

4 Filters

After we have the final resulted image, we will implement different filters as per our project requirements. This filters are used to detect the edge detection inside the image. Before we talk about filter's we should know about the edge detection concept for the images. There can be two different ways to differentiate edges in the images. One is explained as the vertical divider for different pixels while the other can be varied as the ramp function as mentioned in the below figure 8. In our simple language edge detection means, converting 2D image into set of curves by extracting salient features.

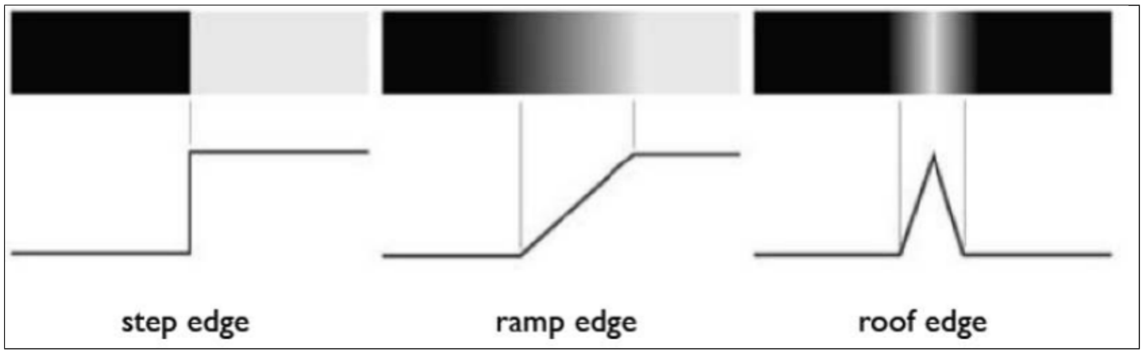


Figure 8: Edge Detection by using ideal edge, ramp edge and roof edge

4.1 Average Filter with coefficients

As we process and get the results from the acquired dat file we can implement some basic filters on this image to find the edges. For the basic Average Filter we have some pre-defined coefficients as mentioned below:

[1 1 1] [1 0 1] [1 1 1] and [1 1 1] [1 1 1] [1 1 1]

After applying this average coefficients we will get the blurred image of the lena which is shown in results section.

4.2 Sobel Filter

Sobel Filter is also considered to detect edges for the 2D image. It is often referred as Sobel Feldman Operator. This is an discrete differential operator to find edge pixels in the image [5]. Sobel operator is constructed by using two 3 X 3 kernel Masks , which gets the image gradients in x and y direction respectively as shown in figure 9. For our project we have applied filter both in x and y direction and the results are plotted. At each point we can get the magnitude of the gradient by summing up both x and y direction gradients.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 9: Sobel Filter using 3x3 Kernel masks for x and y direction

4.3 Gaussian Filter

Gaussian blurring for the image is generated by convolving an image with a kernel of Gaussian values. In one dimensional kernel we will use blur the image in both horizontal and vertical image gradiants. In layman language, gaussian is basically same as the convolving image with the gaussian function. Gaussian blur is considered as low pass filter as it has the effects to reduce the image high frequency components. It uses gaussian function for calculating the transformation to apply for each pixel in the image [6]. The kernel which is used for gaussian blur is shown in figure 10

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 10: Gaussian blur kernel used for lena image

5 Final Results

In this section, we are labelling all the results as per the project requirement. We will save all the extracted dat files from ISE Design Suite and this file is used to read the pixels by using MATLAB. In the below figures we will display all the different results we have extracted from different dat files.

Original Lena Image

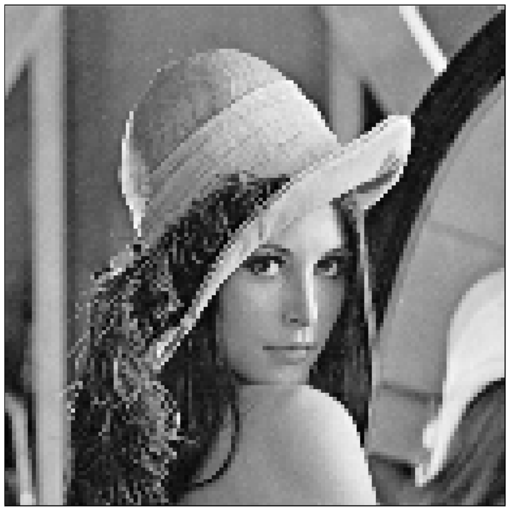


Figure 11: Original Lena Image

Average Filter Results

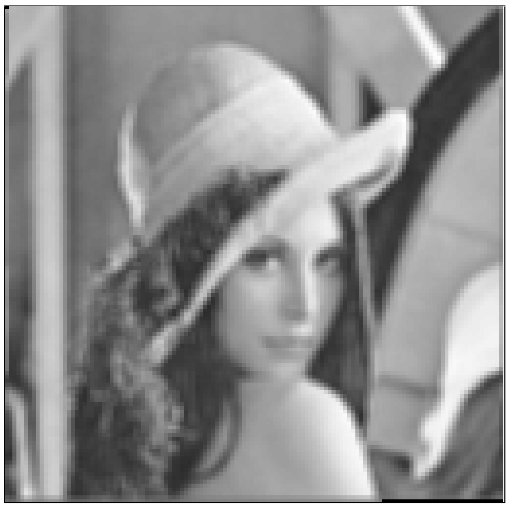


Figure 12: Average Lena Image with [1 1 1] [1 0 1] [1 1 1] coefficients

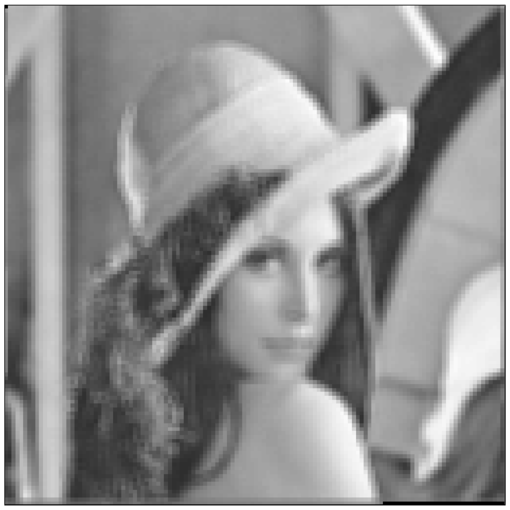


Figure 13: Average Lena Image with [1 1 1] [1 1 1] [1 1 1] coefficients

Sobel Filter Results

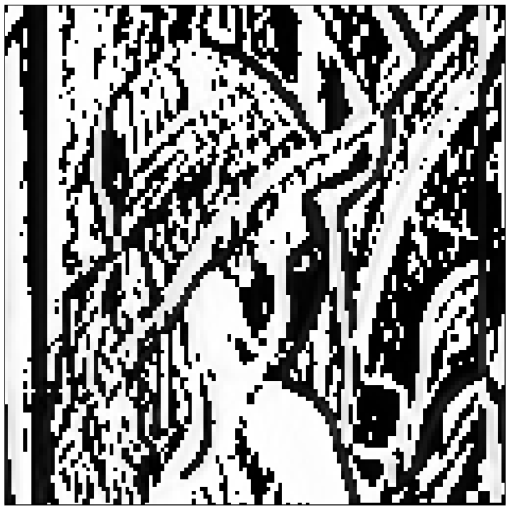
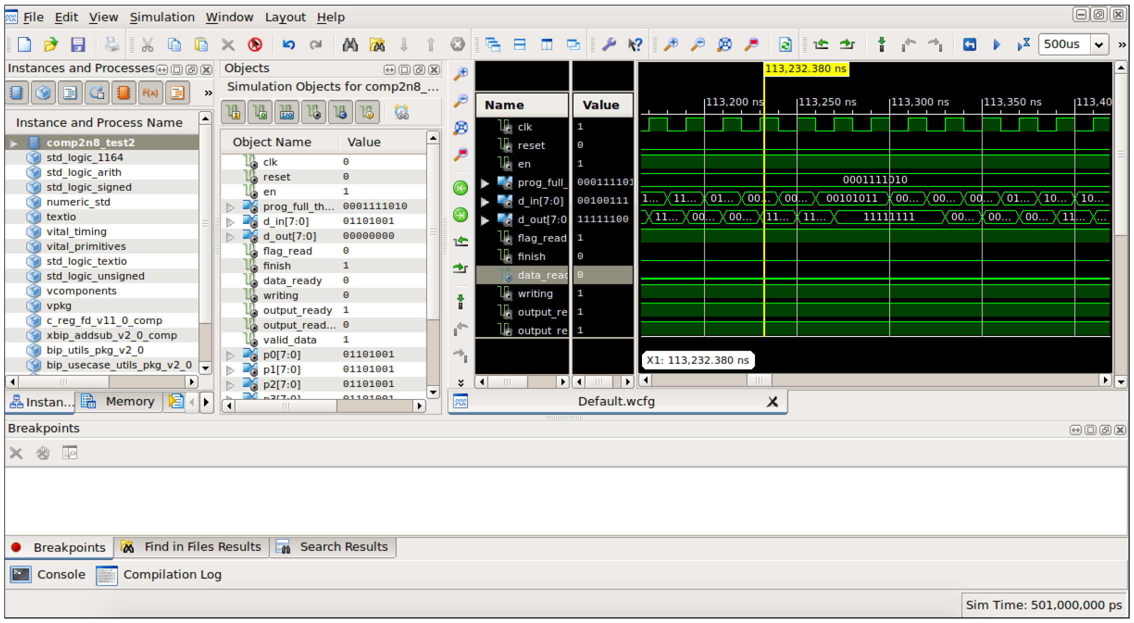


Figure 14: Sobel Horizontal Image gradient result



Object Name	Value
clk	0
reset	0
en	1
prog_full_th...	0001111010
d_in[7:0]	01101001
d_out[7:0]	00000000
flag_read	0
finish	1
data_ready	0
writing	0
output_ready	1
output_read...	0
valid_data	1
p0[7:0]	01101001
p1[7:0]	01101001
p2[7:0]	01101001

Name	Value
clk	1
reset	0
en	1
prog_full_	000111101
d_in[7:0]	00100111
d_out[7:0	11111100
flag_read	1
finish	0
data_read	0
writing	1
output_re	1
output_re	1

```
113,232.380 ns
113,200 ns   113,250 ns   113,300 ns   113,350 ns   113,40
0001111010
1... 11... 01... 00... 00... 00101011 00... 00... 00... 01... 10... 10...
11... 00... 00... 11... 11... 11111111 00... 00... 00... 11... ...
X1: 113,232.380 ns
Sim Time: 501,000,000 ps
```

Figure 15: Simulated output for Sobel Horizontal Image gradient

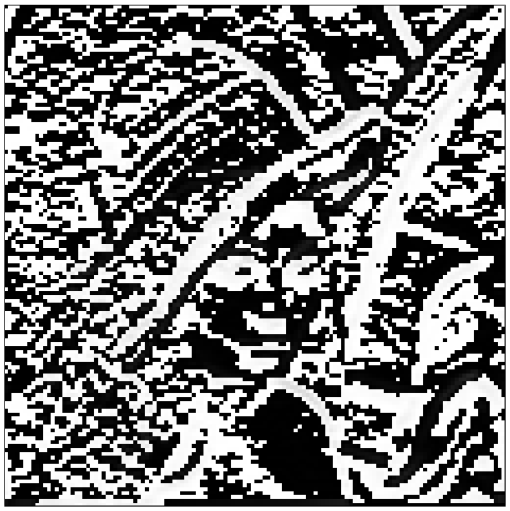


Figure 16: Sobel Vertical Image gradient result

Gaussian Filter Results

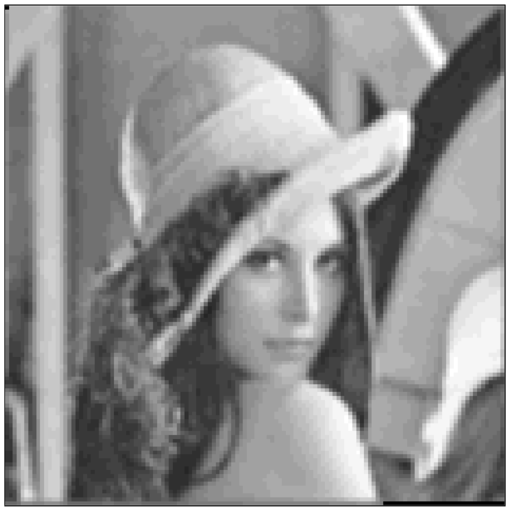


Figure 17: Gaussian Filter Lena Image

6 Conclusions

We have been able to perform image processing in VHDL and take advantage of several of its advantages such as parallelism, flexibility, expandability and efficiency, letting us to perform several mathematical operations simultaneously and with the possibility to adjust our program to use an optimal amount of resources.

As for the results, we see how for the Averaging and the Gaussian blur the results were satisfactory, giving a proper smoothing of the image according to the selected kernels. On the other way, the results for the Sobel filter were not the expected mainly because the 8 bits taken for each pixel did not represent all the information given by the Divider, which was 24 bit. I we had taken all the 24 bit and wrote to the file, we would have been able to obtain more coherent horizontal and vertical components.

A special attention must be put into the size of the elements to be used. The bus width of those must be selected according to the expected output given some input data. For our case, the output size from the last adder of the Processing unit is taken as reference for all the other components. To give an example, if the last adder has an output length of 16 bit, all the other components will have 16 bit buses. Like this, it's important to estimate the maximum possible output value given the image resolution and the kernel values.

If the result is contained in, for example, a 16 bit signed variable, a proper normalization would be needed to downsample it to a 8 bit unsigned component. Like this, most the information will be kept and a final coherent reconstruction will be generated.

7 References

1. Xilinix Software Download link

2. Nexys Board Description

3. S.S.Omran and I.A.Imory. ”Design of two dimensional reconfigurable cache memory using FPGA”. In ICEDSA, 6-8 December, 2016

4. About Testbenches

5. G. Chaple and R.D.Daruwala. ”Design of Sobel operator based image edge detection algorithm on FPGA”. In ICCSP, 3-5 April,2014

6. Gaussian Blur Filter