UNIVERSITÉ DE BOURGOGNE

Masters IN COMPUTER VISION AND ROBOTICS

---

# Real time Image processing using FPGA

Report

---

Muhammad Zain BASHIR
Solène Guillaume

Centre Universitaire Condorcet - uB, Le Creusot
January 2018

# Contents

## 1. Introduction

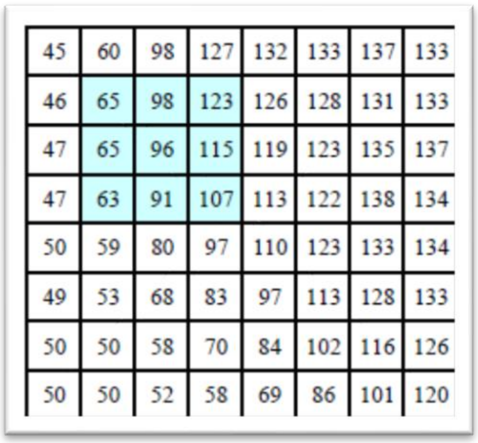
Real-time image processing is a computationally expensive task. A simple 30 frames per second video sequence might require millions of operations to perform some processing in real-time. A general-purpose CPU, is therefore, unsuitable for such a task. This unsuitability of a conventional CPU mainly comes from its serial nature; it performs operations in a sequence i.e. one after the other. To make computation tasks faster we make use of an FPGA. An FPGA or Field Programmable Gate Array is highly parallel in nature, so they can perform they can offer significant performance improvements over CPUs.

A FPGA is an integrated circuit designed to be configured by a designer after manufacturing. It is basically a collection of logic blocks and a switching network. The blocks can be connected in many combinations to perform complex combinational functions or simple logic tasks. These combinations are usually specified using a hardware description language (HDL).

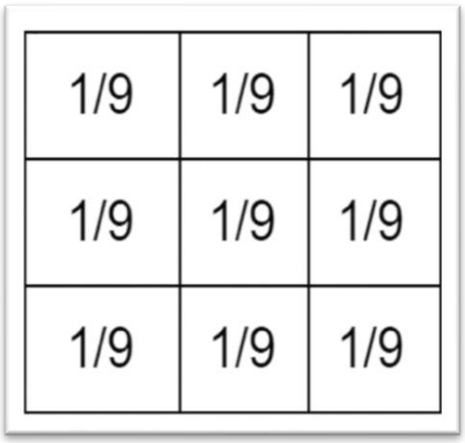
This project focuses at implementing some basic 2D filters on a 128x128 grayscale image using NEXYS 4 Artix-7 FPGA board. The description language used is VHDL using Xilinx ISE editor.

## 2. Implementation

A 2D filtering process requires accessing a certain number of pixels of the image called the neighborhood (figure 1 (a)), performing the convolution operation between the neighborhood pixels and the filter coefficients (figure 1 (b)) followed by the replacement of the middle pixel of the neighborhood by the output of the convolution operation.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 45 | 60 | 98 | 127 | 132 | 133 | 137 | 133 |
| 46 | 65 | 98 | 123 | 126 | 128 | 131 | 133 |
| 47 | 65 | 96 | 115 | 119 | 123 | 135 | 137 |
| 47 | 63 | 91 | 107 | 113 | 122 | 138 | 134 |
| 50 | 59 | 80 | 97 | 110 | 123 | 133 | 134 |
| 49 | 53 | 68 | 83 | 97 | 113 | 128 | 133 |
| 50 | 50 | 58 | 70 | 84 | 102 | 116 | 126 |
| 50 | 50 | 52 | 58 | 69 | 86 | 101 | 120 |



| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

(a) (b)

*Figure 1: (a) A 3x3 neighborhood shown colored. (b) The coefficients of a typical 3x3 averaging filter.*

To successfully apply a filter to an image using FPGA we need to follow the following pipeline (figure 2); (1) read an image file, (2) access the neighborhood pixels of the image, (3) apply the convolution function to the neighborhood pixels and (4) write the processed image file. These tasks broadly divided into three main parts.

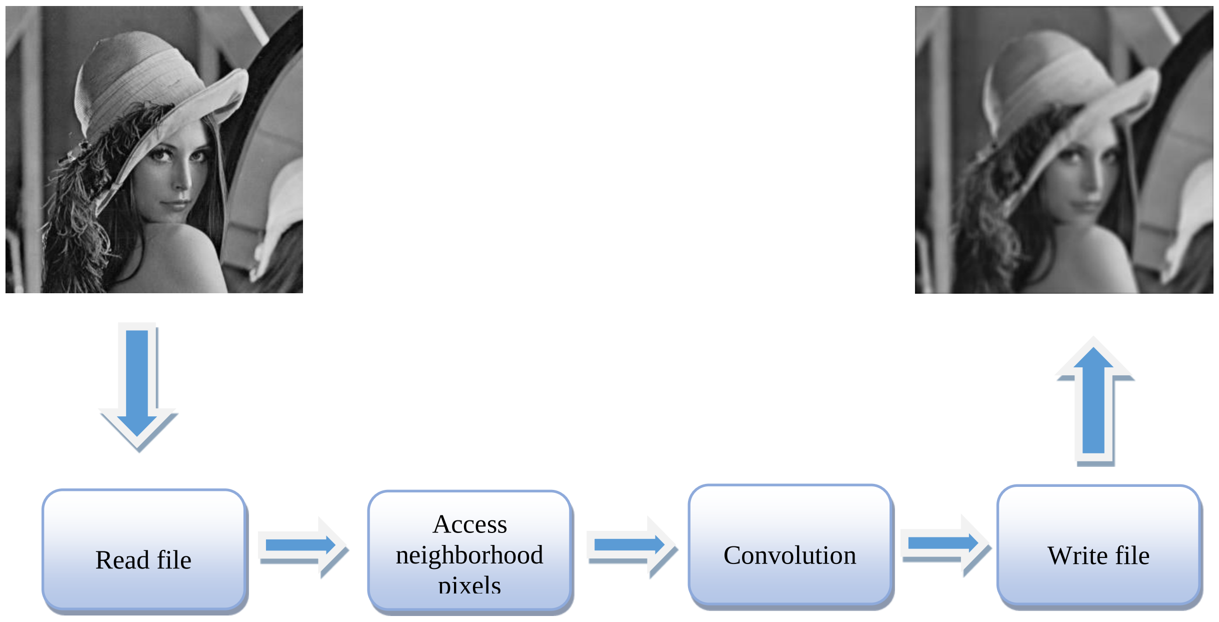


*Figure 2: Filtering pipeline using FPGA*

To implement the above given pipeline the task is broken down into three main components:

1) **Cache memory**
   *To access the neighborhood pixels*

2) **Processor**
   *Implements the actual filter (convolution)*

3) **Read/Write component**
   *Reads the image file and writes the pixels after they are processed*

## 2.1. Cache memory

As mentioned earlier, cache memory is the component that is used to access the neighborhood pixels. Cache memory is created by a combination of flip flops and FIFO memory. A schematic of the cache memory is shown in figure 3.

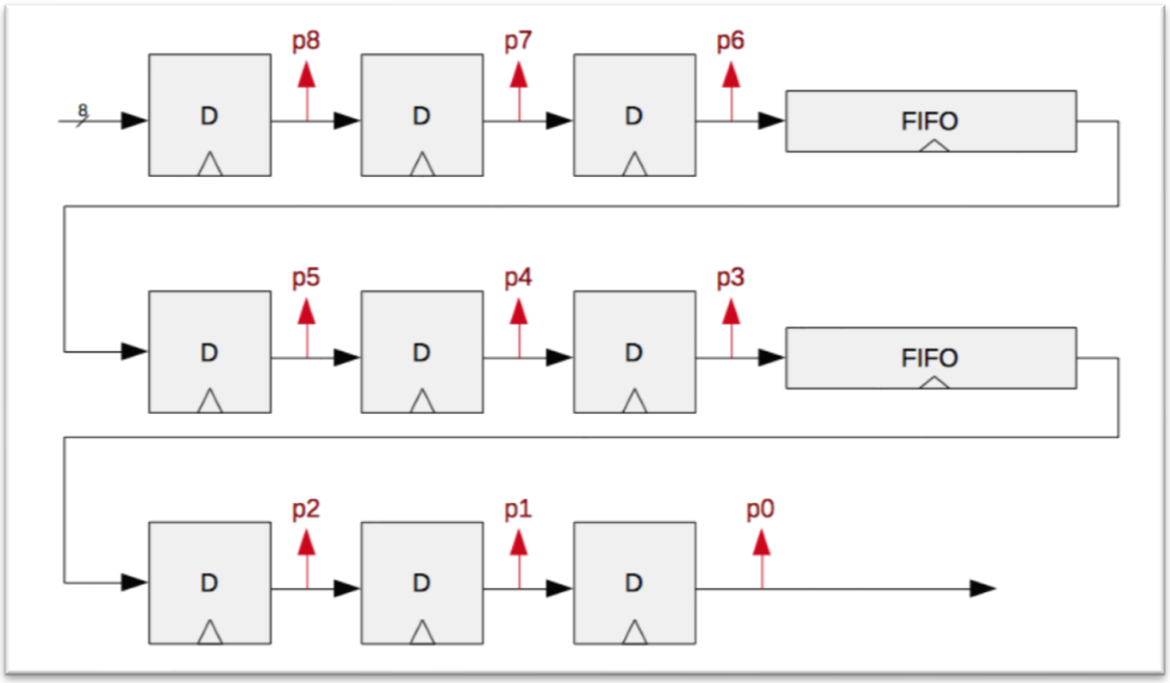
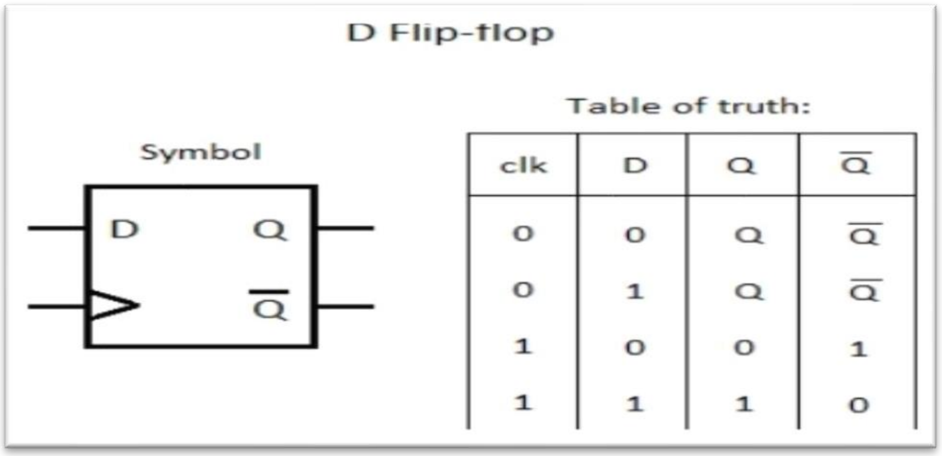


*Figure 3: Schematic of the cache memory. Flip flops and FIFO's are shown*

Before discussing the working of the cache memory let us briefly see how each unit of the cache memory works.

### 2.1.1. Flip Flop

For our purposes, a flip flop is simply a memory element. It is used to store state information. The ‘D’ or data flip flop that we are using simply copies the input state (D) to the output port (Q) at each rising edge of the clock. A flip flop with its truth table is shown in figure 4.



| clk | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Q | $\overline{Q}$ |
| 0 | 1 | Q | $\overline{Q}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

*Figure 4: D- flip flop with its truth table*

### 2.1.2. FIFO memory

A FIFO (first-in-first-out) is a memory unit that stores data in a first in out first out fashion. This means that the first piece of information written to the memory will be the first one to be read. A schematic of FIFO is given in the figure below.

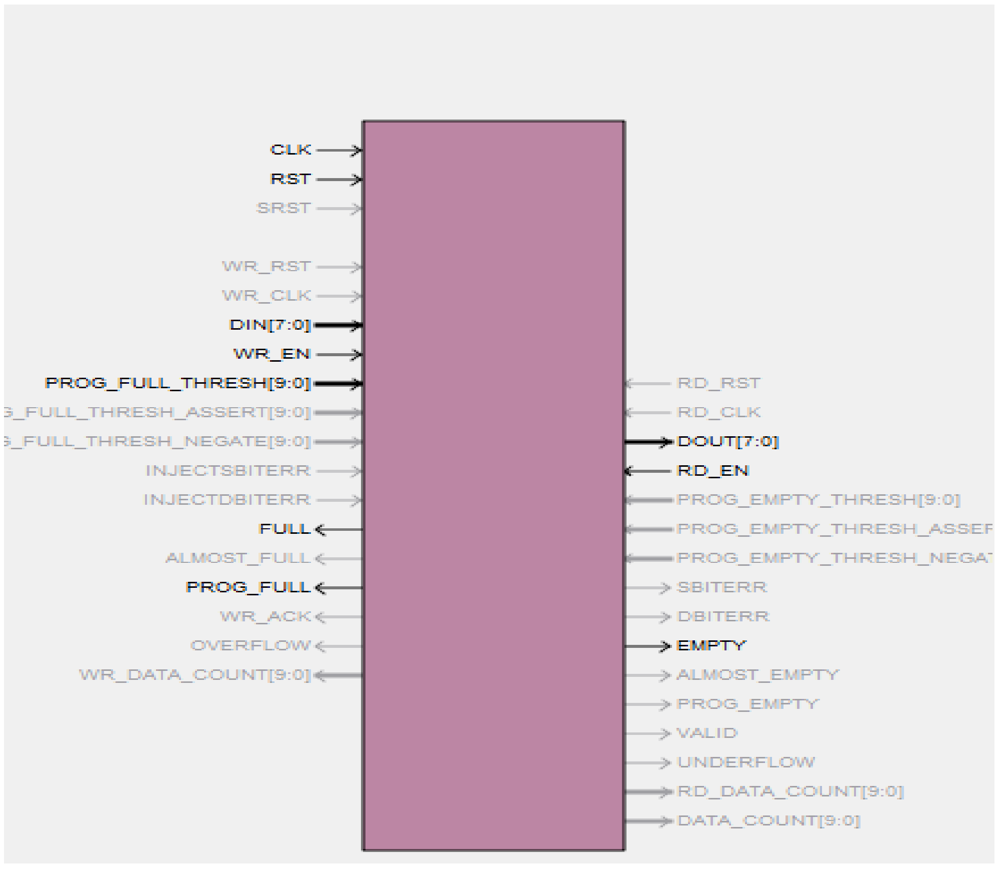


*Figure 5: FIFO memory*

The *DIN* and *DOUT* ports are where the data is written and read from the FIFO respectively. For our case we set it to a size of 8 bits each. This memory unit can be programmed to raise a flag as soon as it reaches a certain value. This certain value is given in using the *PROG_FULL_THRESH* port. The flag that is raised when it the FIFO is full is called *PROG_FULL*. The reading and writing modes are enabled only when the *RD_EN* and *WR_EN* pins are set to 1. The *EMPTY* pin is raise to 1 if the all the data has been read from the FIFO.

### 2.1.3. Working

All the 9 flip flops shown in figure 3 are connected to the same clock so each at each rising edge they simply shift the data (pixel value) to the right. After 3 clock cycles the write enable of the first FIFO is set to 1. The clock continues to tick for a further 125 cycles which writes the next 125 pixels to the FIFO. Since the image size is 128x128, we have been able to read the first row of our image. The

*PROG_FULL_THRESH* is set to 125 so it raises the *PROG_FULL* flag as soon as the first 128 pixels have been read. This flag sets *RD_EN* pin of the same FIFO to 1. Another 3 cycles later the *WR_EN* of the second FIFO is set to 1. This FIFO also stores 125 pixels before it is full and passes the pixels to the following flip flop. 3 more clock cycles later the next 3 flip flops are full and the cache memory has read 259 pixel values. Every clock cycle following this shifts the neighborhood to the right by one pixel. The neighborhood pixels are accessed directly from the outputs of the 9 flip flips using some signals. The figure below is a graphical representation of the above described process.

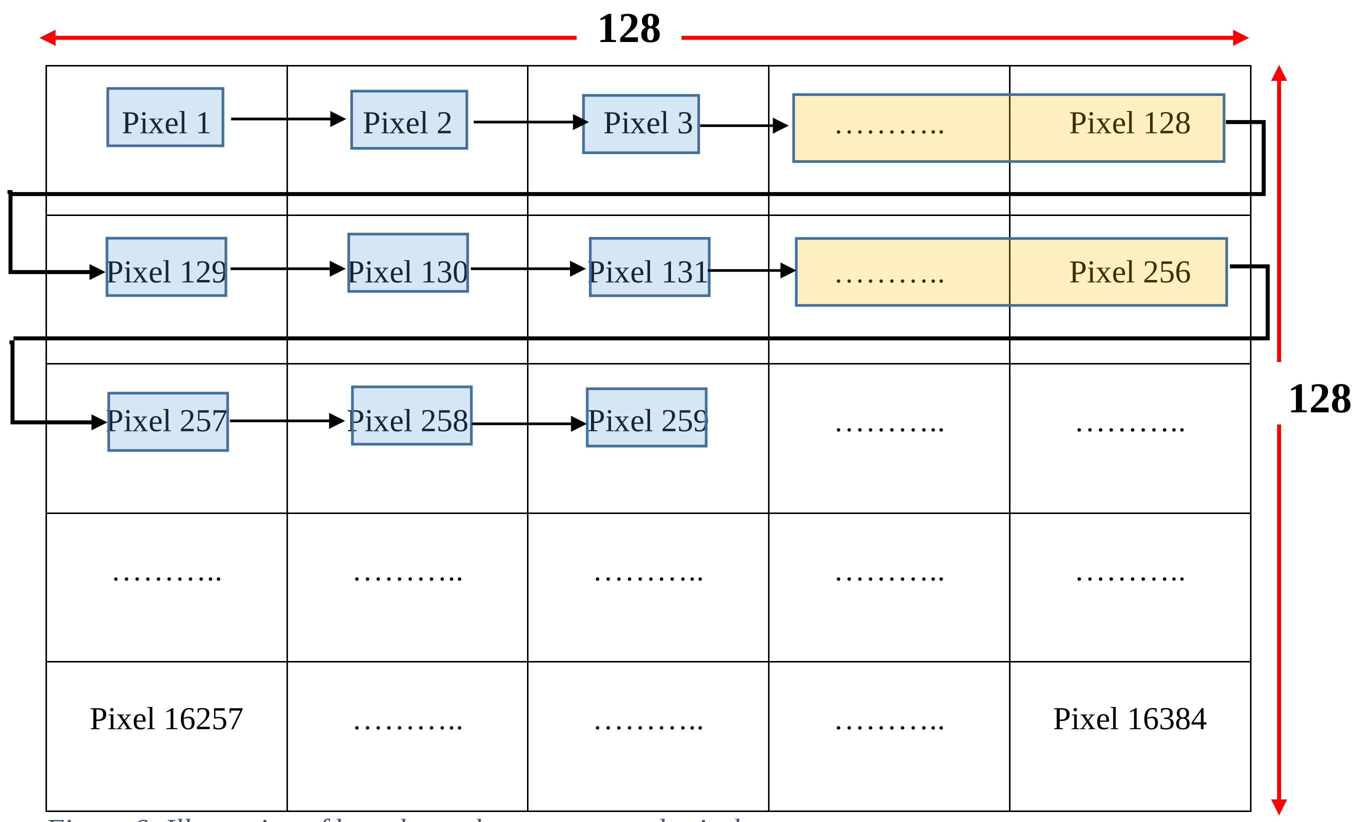


*Figure 6: Illustration of how the cache memory reads pixels*

## 2.2. Processor

Once we have started to read the pixels, we pass the neighborhood pixels to the processor unit. The processor performs the convolution operation i.e. it multiplies the neighborhood pixels with kernel coefficients, adds up the multiplied values, and divides by a constant to normalize the value to the range 0-255. Each component of the processor is explained in detail below.

### 2.2.1. Multipliers

To filter an image, we need to multiply the neighborhood pixels with the kernel coefficients. Figure 7 is an illustration of the multiplier positions in the filtering pipeline.

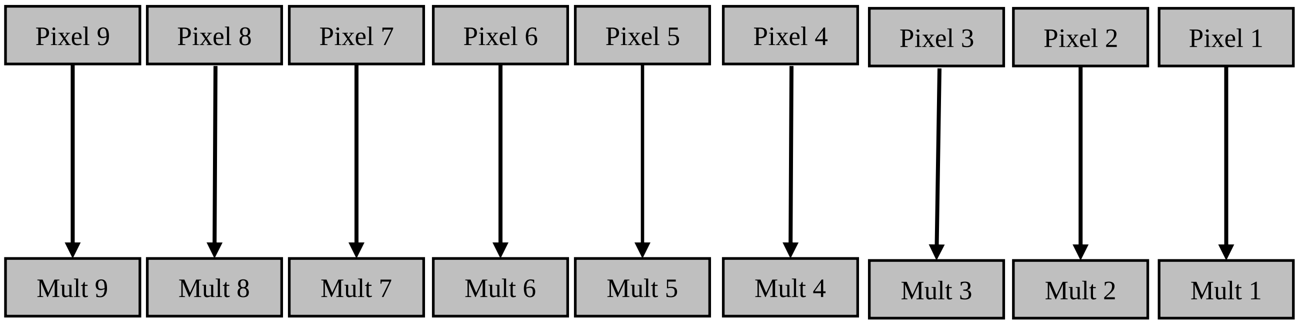
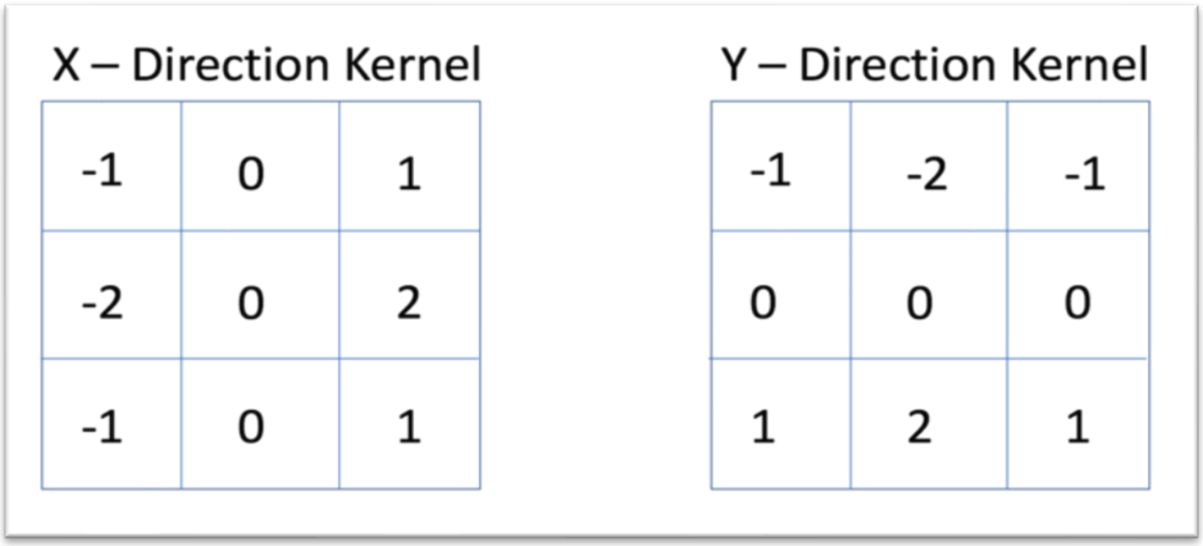


*Figure 7: Multiplier position in the filtering pipeline*

The multipliers used are 4 bits signed multipliers so that they can encode a maximum integer value of positive or negative 8. This is enough for the common filters we wish to implement. An example of the coefficients of a Sobel filter is given below.



X – Direction Kernel

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

Y – Direction Kernel

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

*Figure 8: Sobel filter coefficients*

### 2.2.2. Flip flops

Multiplication is a computationally heavy task. To allow for sufficient computation time, we pass the output of the multipliers to D-flip flops. The flip flop size is determined by adding the number of bits of a pixel and the number of bits of the multiplier. After incorporating the flips flops, our filtering pipeline given in figure 7 becomes the one given in the figure below.

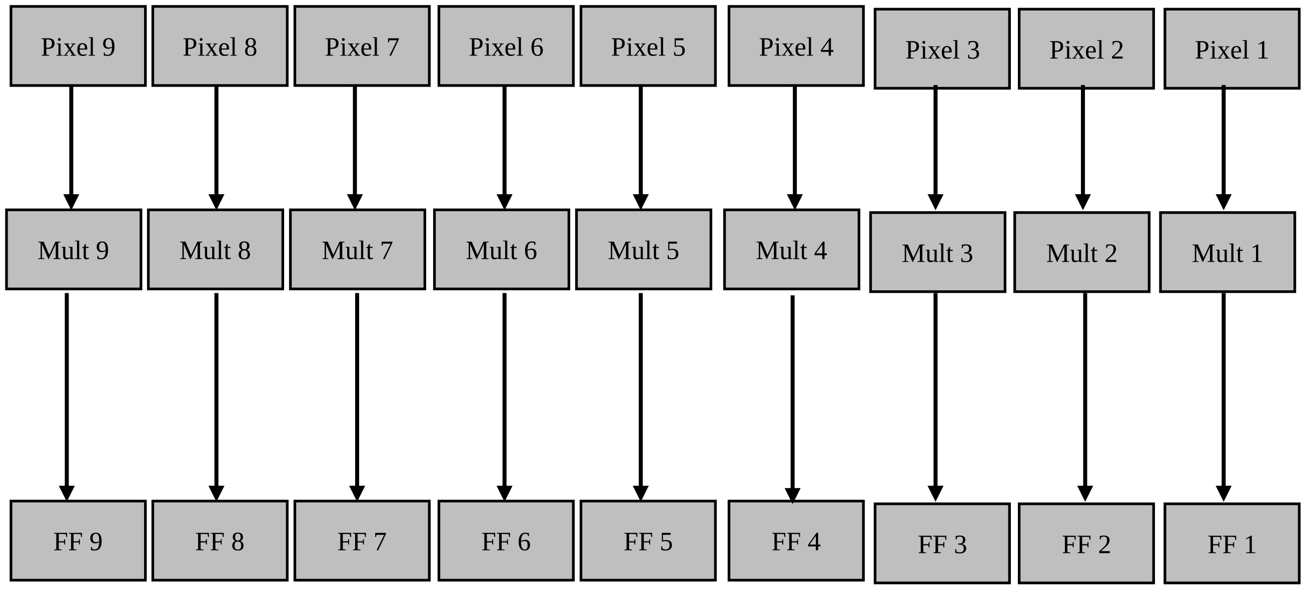


*Figure 9: Filtering pipeline after incorporation of flip flops*

### 2.2.3. Adders

The filtering process involves adding up the values after they have been multiplied. We must, however, be careful of the size of the size of adders. Adding two same bit numbers can result in produce an answer with one more bit. So, at each adder level we have to increase the size of the adders.

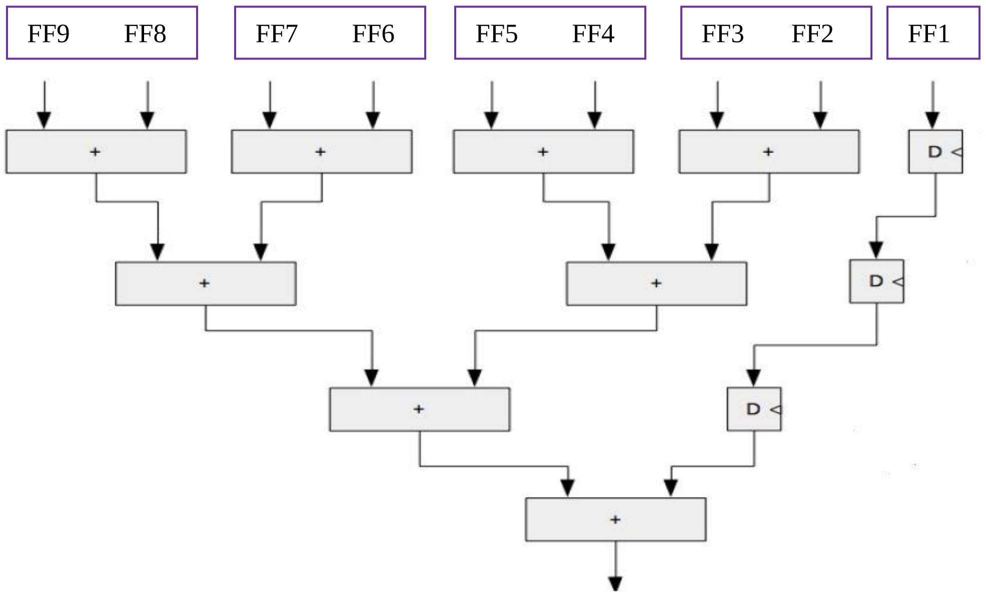


*Figure 10: Adder configuration*

### 2.2.4. Divider

The final output of the adders is passed through a divider which normalizes the values in the range 0-255. The complete filtering pipeline with the divider is given below.

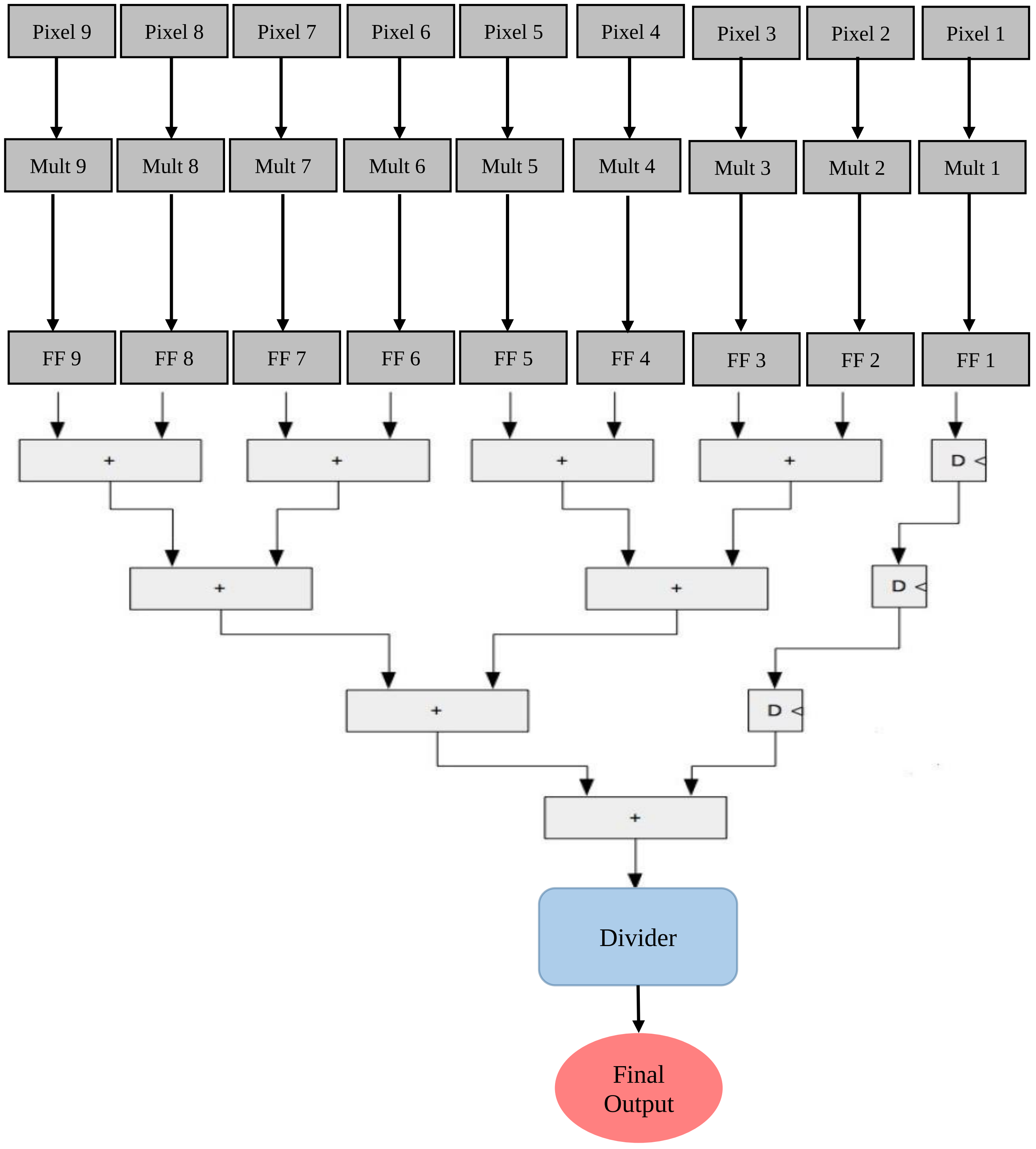


*Figure 11:Complete filtering pipeline*

### 2.2.5. State machine

A state machine is also used in the processor unit to ensure that subsequent components in the module are enabled at the right time. As soon as the file reading starts, pixel counter starts counting the number of pixels and each clock cycle the state of the signal *STATE* shifts to the next state. We need a total of 6 states; one to enable the all the flip flops immediately after the multipliers, 4 for the 4 layers of adders (one for each layer) and the last one for the divider.

## 2.3. Read and write

The Lena grayscale image we used had already been binarized and vectorized such that the .*dat* file we used consists of 1 column and 16384 (128x128) rows.

Our read and write module starts as soon as the image file is opened. This raises a flag called *FLAG_READ* which tells our cache memory to start working. A counter then starts counting the number of clock cycles. Since the size of cache memory is 259 pixels, we must read 259 pixels in order to get a continuous stream of pixels. The processing only starts when 259 pixels have been loaded into the cache memory counted by the counter which raises a flag called *NEIGHBORHOOD_READY*. This flag is an output port of our cache memory which is passed then to the read/write module. Since the processing is being done in real time, this flag is also used to start writing the pixels. Since there is a bit of delay introduced due processing, we introduce a small delay before our modules starts to write the processed pixels.

As soon as all the pixels have been read from the file, there are still 259 pixels left in the cache memory. Closing the file brings back the flag *FLAG_READ* back to 0. The counter restarts counting at this moment and after 259 counts it sets the flag *NEIGHBORHOOD_READY* to 0. Since this flag is connected to a start writing signal in the read/write module, the module stops writing pixels. The figure below shows the relationship between read/write module and other components.

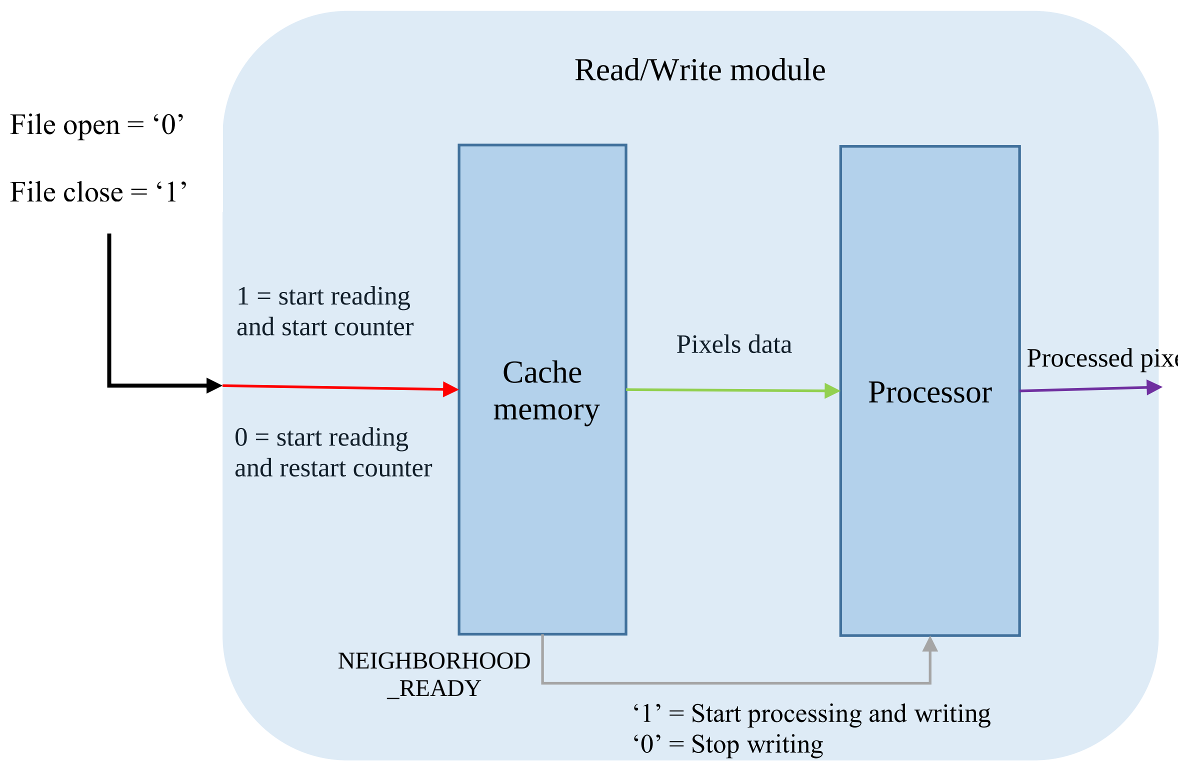


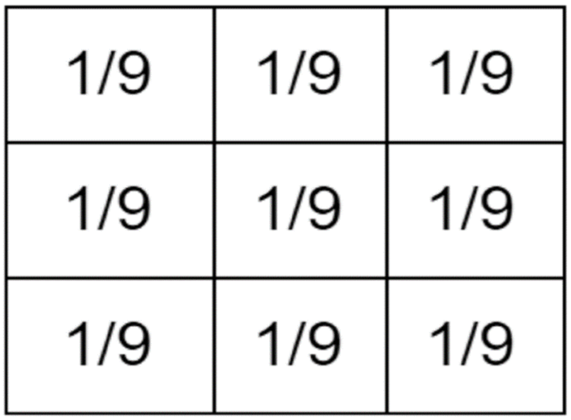
*Figure 13:The relationship between read/write module and other components*

## 3. Testing and results

To run the simulation and filter the Lena image three filters were applied; Averaging, Gaussian and Sobel.

### 3.1. Averaging filter

This filter acts as a low pass filter and has the effect of blurring an image. Given below is the Kernel used for averaging filter.



| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

*Figure 14: Averaging filter Kernel*

The figure below shows the result of applying the averaging filter to Lena image when compared with the original image.

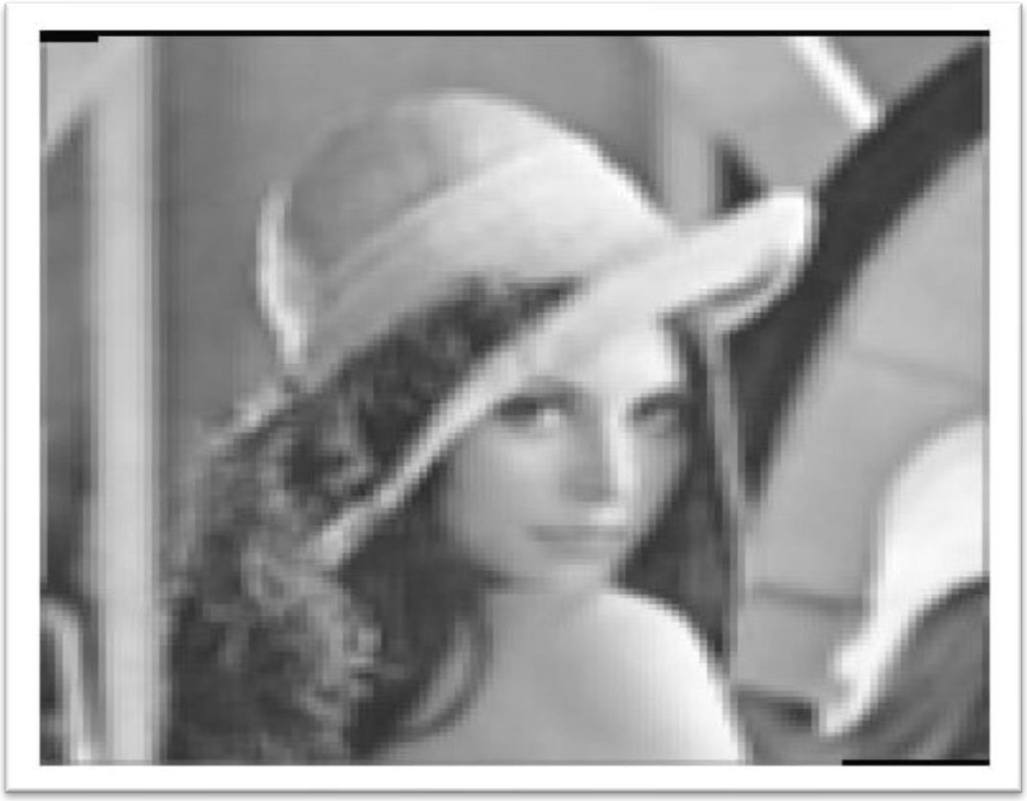
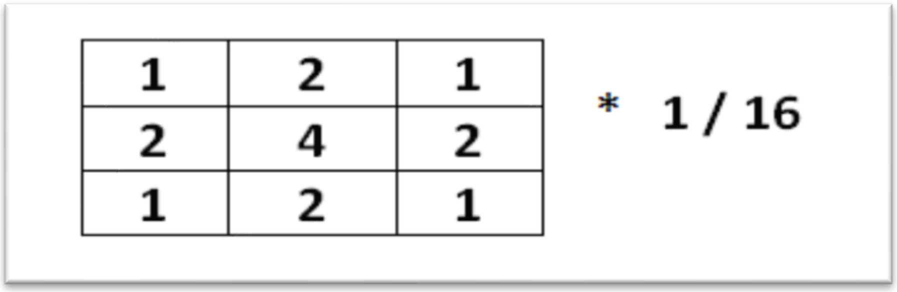


*Figure 15: Averaging filter*

## 3.2. Gaussian filter

Gaussian filter is also a low pass filter however it gives more weight to the middle pixel hence also called weighted average filter. It also has the effect of blurring the picture but less as compared to the averaging filter. This filter is mostly used for noise removal. Given below is the kernel used to apply the Gaussian filter.



| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

* 1 / 16

*Figure 16:Gaussian filter Kernel*

The result of applying a Gaussian Kernel to the Lena image is given below in figure 17.

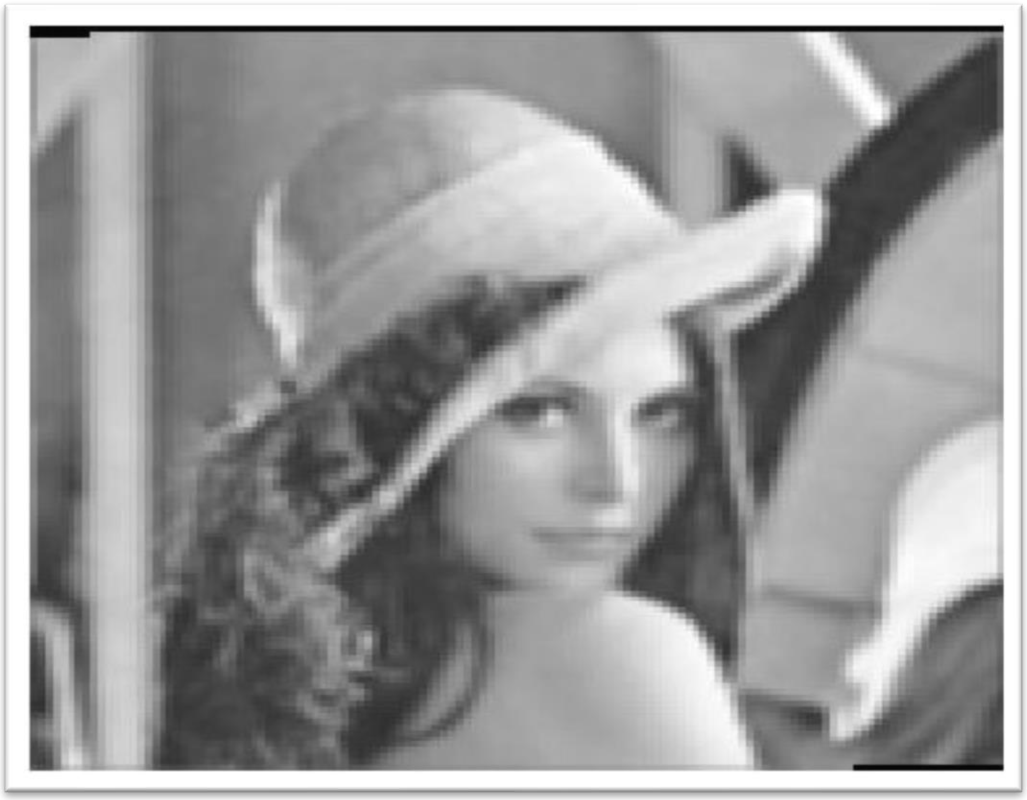
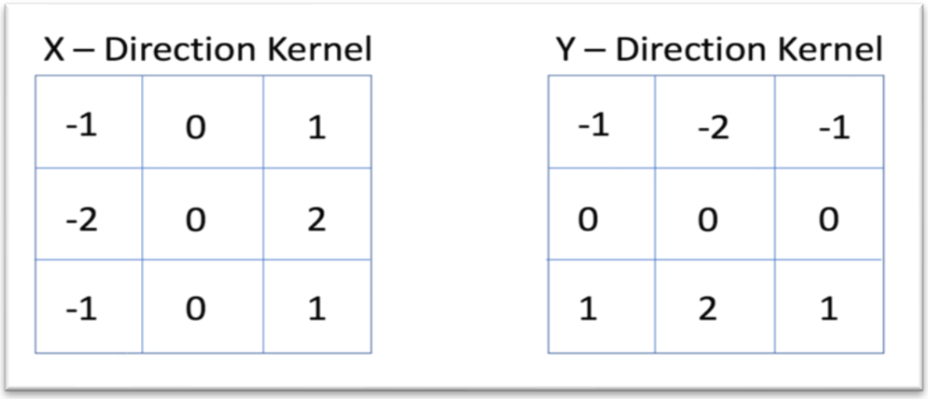


*Figure 17:Gaussian filter*

### 3.3. Sobel y-filter

The Sobel y-filter is used to detect edges along the y direction in the image. The kernels used for both Sobel x and y direction are shown in below.



X – Direction Kernel

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

Y – Direction Kernel

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

*Figure 18: Sobel Kernels*

The result of applying a Sobel filter to the Lena image is shown below.

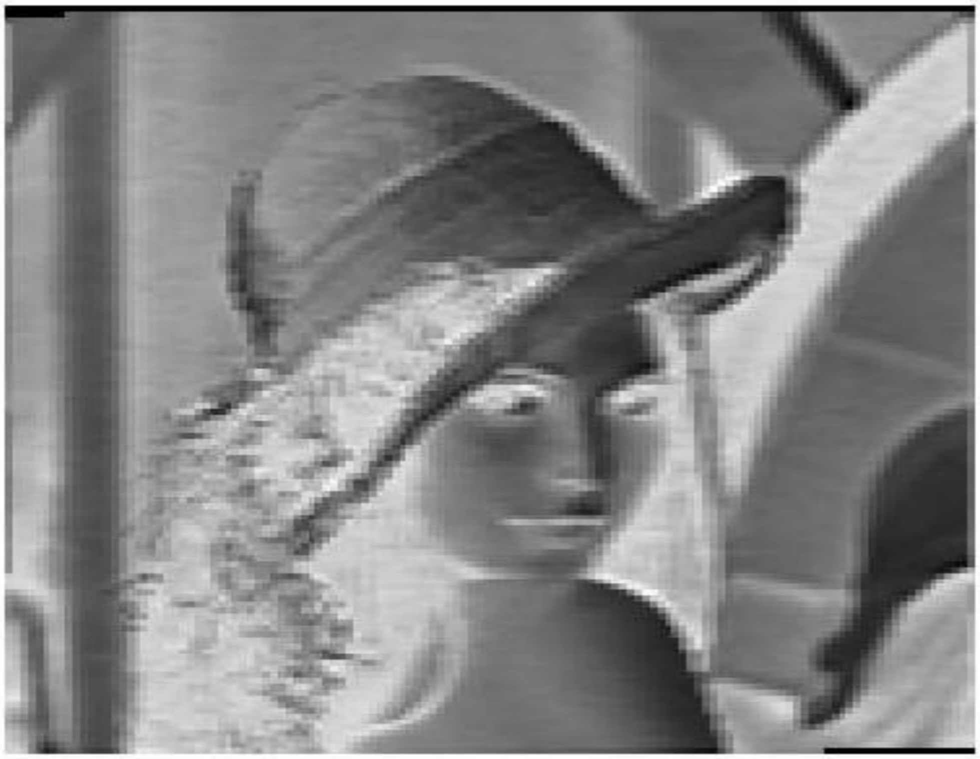


*Figure 19:Sobel filter*

## 3.4. Edge detection

Many filters other than Sobel can be used for edge detection. The one we used is shown below in figure 20.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

*Figure 20: Laplacian Kernel*

The result of applying the Laplacian kernel to the Lena image is shown below.

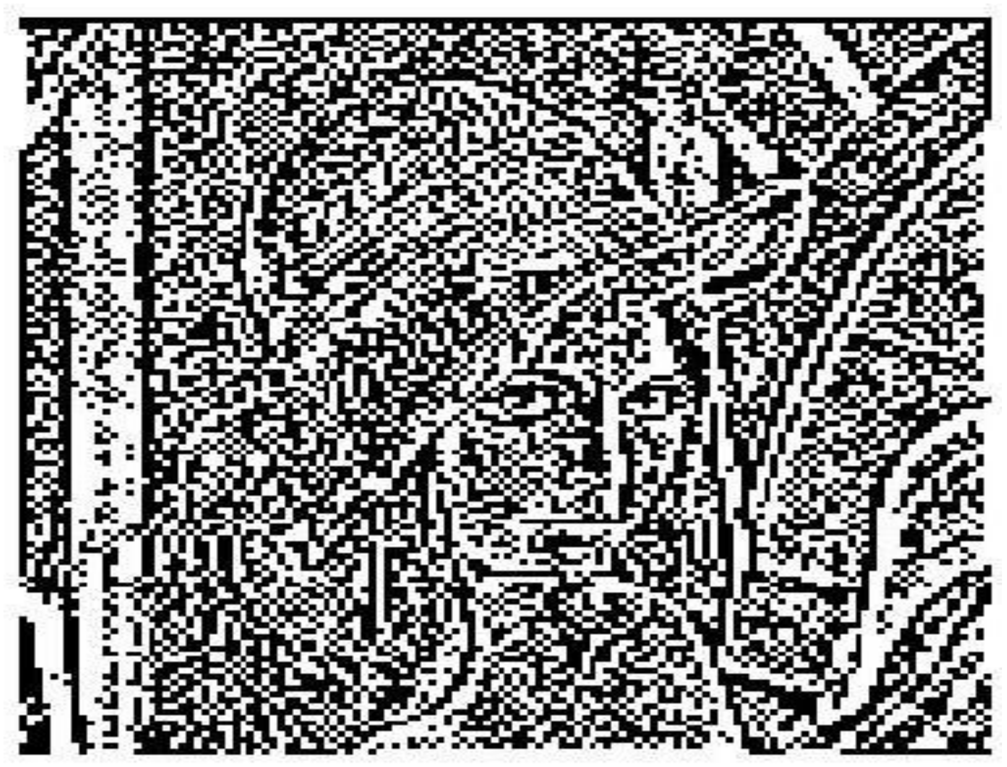


*Figure 21: Laplacian filter*

Note: To apply all these filters we have to change the *input a* of the multipliers according to the Kernels. If there is a division like in the Gaussian kernel, we have to set the divisor of the divider to the desired value otherwise it has to be set to 1.

## 4. Problems

There were a few problems we faced during the course of this project. First of all we were unable to read all the pixels from the cache memory and as a result we had to add a few zeros to compensate for the un-read pixels and to make the image 128x128 in size. This is clearly visible in our results that the first row and a few pixels of the last row are black.

Secondly, adding a divider increases the computation time and introduces a delay in writing the processed pixels. As a result, our resulting image was shifted by a few pixels. We manage it by introducing a hard-coded delay before our module starts writing. This we believe is not a good solution and must be corrected.

The third problem we are unable to solve were the incorrect results for Sobel filter. The Sobel filter which is supposed to detect edges was not able to give the desired output for our simulation.