UNIVERSITÉ DE BOURGOGNE

MSc COMPUTER VISION

REAL TIME IMAGING AND CONTROL

2D Filtering with Xilinx Artix-7 FPGA

Authors:
Sandeep MANANDHAR
Mohamed EISSA

Supervisor:
Prof. Julien DUBOIS

December 10, 2015

Abstract

In this report we briefly state how we implemented a 3×3 convolution kernel for 8-bit grayscale image. Nexys4^{TM} with Xilinx's Artix-7 FPGA board was used which is a low power and low cost device considered to be optimal for machine vision applications.
We have used a 128 $\times$ 128 grayscale image of Lena, which is initially loaded in a ROM from where the intensity values are read to convolution kernel via a series of $Flipflop$'s and $FIFO$'s. The internal arthmetics of the kernel has been mapped into a state machine to create a schedular. The available filters(kernels) are delta, blur, sobelX, sobleY, and laplacian filters. The filters can be selected via switches in the development board and the images can be viewed in a Monitor connected via VGA cable.
We have used Xilinx ISE to develop and simulate the project. It is a powerful software tool to synthesize designs, perform timing analysis, as well as analyze RTL diagrams.

Contents

List of Figures

1 Introduction

In this project, we aim to design a 3×3 convolution kernel for 8-bit grayscale image. We read the image from a ROM and feed it to a series of $Flipflop$'s and $FIFO$'s and then into a kernel where arithmetical operations are performed to convolve a 3×3 patch of the image.
The series of flipflops and FIFO's facilitate to make a sliding window action over the image[]. With the help of LogiCore IP FIFO generator, we create two 8-bit FIFO's of length 1024. The fifo has an inbuilt feature for thresholding at some length of queue which asserts a flag. We use this flag to dequeue from the FIFO.
The kernel for convolution has 6-stages of multipliers and adders. These arithmetic operators work on rising edge of the clock by using flipflops between each stages. The kernel requires division of 8 of 4. We perform this by simple bit shifting. Other integer division has been left out because of the complexity.

2 Kernel

2.1 Algorithm

As we can see in figure 1, the pipeline begins with multiplication of pixel and mask. The 8-bit pixel is multiplied with 4-bit signed integer.

```
PIXEL : in std_logic_vector(7 downto 0);
--
signal product : signed(12 downto 0);
--
signal mask : signed(3 downto 0):=x"1";
--
product <= mask*signed( "0" & (PIXEL) );
```

The multiplication requires both of the operands to be signed. Thus we copy the sign bit of the *PIXEL* which is 0 by the virtue of it being intensity value and create a 9-bit operand. The product is a 13-bit signed integer which is fed to a 14-bit adder as shown in figure 1 which works on every rising edge of the clock.
This happens with 8 of the pixels in kernel meanwhile the $9th$ pixel is fed to a flip-flop to synchronize the pipeline. The sum of 14-bit adders is cascaded with 15-bit adders which is then cascaded with 16-bit adder. The 17-bit adder then takes the result of 16-bit adder(sum of product of 8 pixels with their respective masks) and adds it with the $9th$ pixel which is in the pipeline synchronized by a cascade of 4 flip-flops.
The following snippet shows addition of two integers A and B. We copy the sign bit (*msb*) using $'left$ attribute and append it using & operator.

```
addAB <= (A(A'left) & signed(A)) + (B(B'left) & signed(B));
```

The absolute sum is then divided by a normalizing factor. For simplicity we have implemented division by 8 or 4 which can be done my shifting the bits to right. In the following snippet, we divide *absoluteSum* by 8.

```
OUTPUT <= std_logic_vector(absoluteSum(10 downto 3));
```

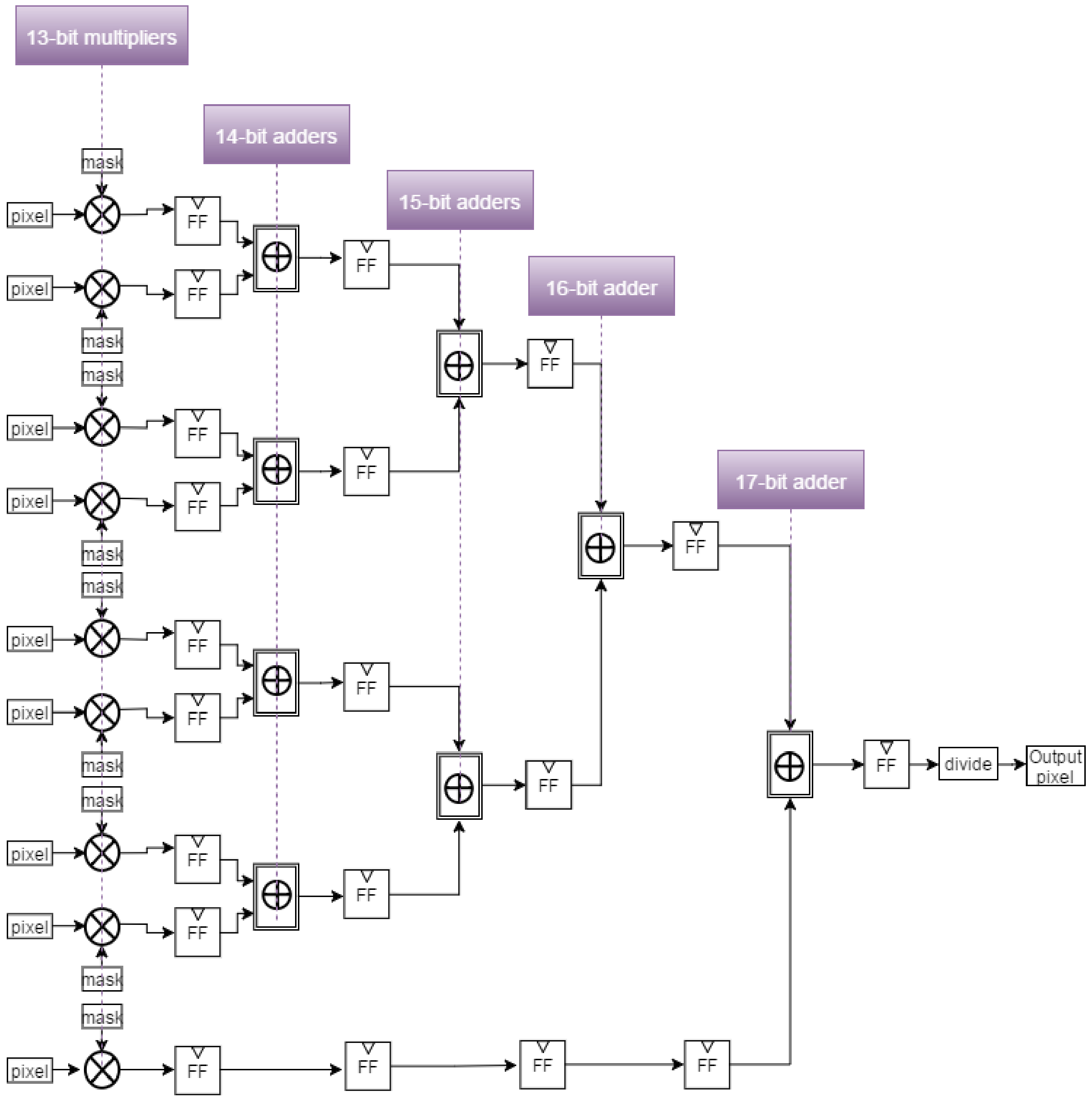


Figure 1: 6 stage pipepline for 3×3 convolution kernel

In this way, we have implemented a 6-stage convolution kernel. It takes 6-clock cycles to produce an output.

2.2 Kernel State Machine

The first version of our implementation, we had our kernel without WR_EN and RD_EN signals, which made it unaware of when it starts receiving data, and when it starts giving output. In our modified version, we proposed and implemented a state self-aware kernel using one of following state machines in 2 and 3

- Initial state is in $S1$. As the WR_EN signal is asserted, the state begin to transit to next state $S2$ with output $EN1$ asserted which enables the first delay Flipflop for 9th pixel multiplication result.

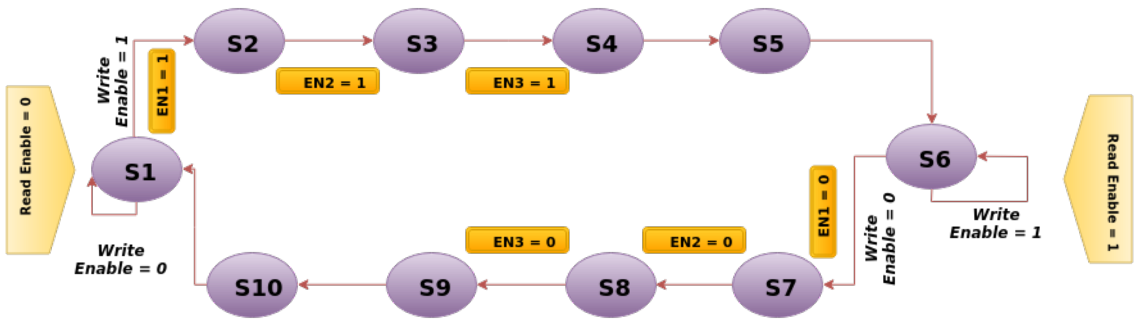


Figure 2: State Machine for 3×3 convolution mask: s1,s2,s3...s10 are states; *write_enable* is input; *read_enable* is output flag asserting that the kernel's output is ready

- $S2$ has only one transition to $S3$ with output $EN2$ asserted which enables the second delay Flipflop for 9th pixel multiplication result.
- $S3$ has only one transition to $S4$ with output $EN3$ asserted which enables the third delay Flipflop for 9th pixel multiplication result.
- $S4$ has only one transition to $S5$ to add one clock cycle delay for the last addition operation between the the first 8 pixels multiplication values and the 9th pixel multiplication value.
- $S5$ has only one transition to $S6$ to add one clock cycle delay for the division operation of the total sum.
- The state machine stays in $S6$ while the kernel WR_EN is active, with output kernel RD_EN equal to 1. Once kernel RD_EN is deasserted, the state machine transits to state $S7$ with output $EN1$ deasserted to disable the first delay FlipFlop.
- $S7$ has only one transition to $S8$ with output $EN2$ deasserted to disable the second delay Flipflop.
- $S8$ has only one transition to $S9$ with output $EN3$ deasserted to disable the third delay Flipflop.
- $S9$ has only one transition to $S10$ to add one clock cycle delay for the total sum operation between the the first 8 pixels multiplication values and the 9th pixel multiplication value for the last stream of pixels.
- $S5$ has only one transition to $S1$ to add one clock cycle delay for the division operation of the total sum for the last stream of pixels
- Once the state machine returns back to $S1$, $EN1$, $EN2$, $EN3$ and RD_EN1 are now equal to 0.

We also propose a minimized version of the previous state machine as in figure 3. The main difference in this state machine is that it has less number of state, since on every state is sensitive to the input of WR_EN signal; if true it will transit forward till $S6$ and if false, it will move backward to the initial state. However, the main issue with this machine is that it will not assert the RD_EN for a stream of pixels less than 5 in number.

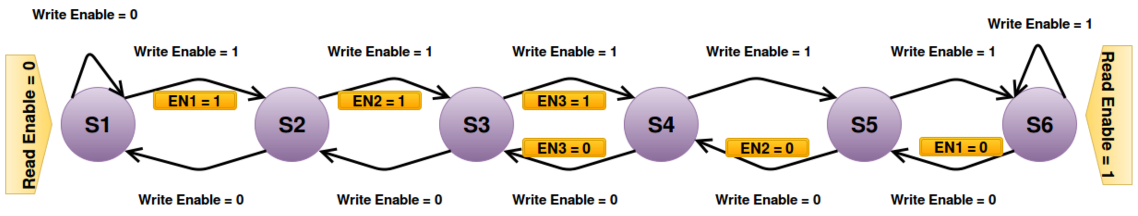
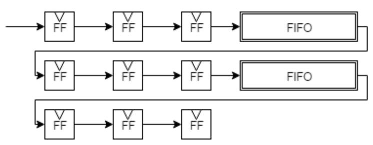


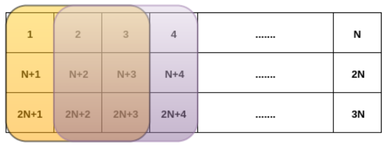
Figure 3: Kernel Minimized State Machine

3 Cache

We implement cache using a combination of nine 8-bits FlipFlops attached to two 8-bits FIFOs as in figure. 4.



(a) flipflops and FIFO



(b) Sliding Window

Figure 4: Flipflops and FIFO's with sliding window action

We use flipflops for simple delay shown in 5 as it latches the input value at some clock cycle t. The nine flipflops are used as the kernel input to represent a stream of 9 corresponding pixels that the 2D filter is going to operate on.

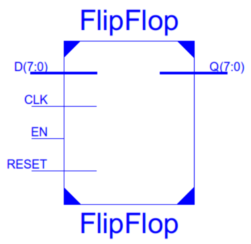


Figure 5: Delay Flipflop

The FIFO element is memory element that follow the paradigm of First in First out- the first word to be written in the FIFO, is the first word to be out. FIFO module is generated using Xilinx FIFO Generator v9.3 as figure 6
While the flipflops are used for a kernel input, FIFOa are used to store the remaining of each image rows. To do this we have to set a $prog_full_thresh$ for the FIFO to 125 which is the 3 Flipflops subtracted from the row size 128. But according the Xilinx FIFO Generator v9.3 documentation, the FIFO output will be delayed by two clock cycles once $prog_full$ is asserted, so we set $prog_full_thresh$ to 123 to make sure that the kernel is being applied on the correct corresponding pixels.

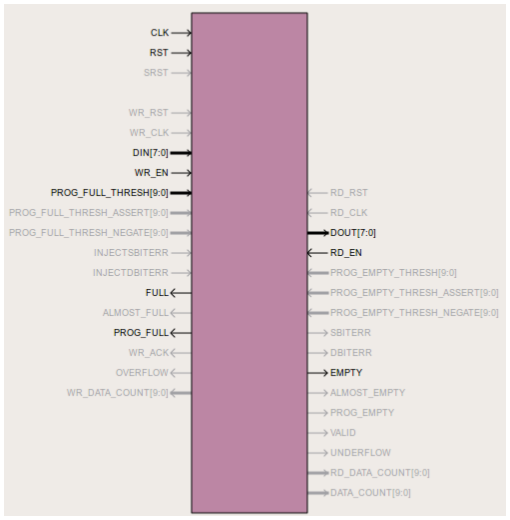


Figure 6: FIFO

3.1 Cache State Machine

The cache module has a WR_EN and RD_EN signals to be used by the top module to know when to start reading the output. For this, we implemented a simple state machine shown in 7 to synchronize the WR_EN and RD_EN

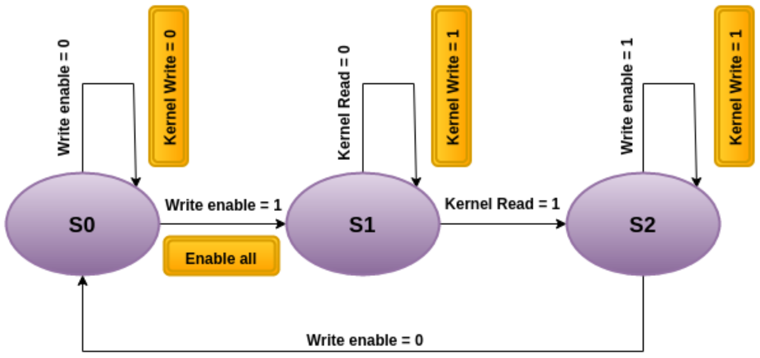


Figure 7: State Machine for Cache

- The state machine starts with initial state is in $S0$. It stays in the same state as long as WR_EN signal is 0 with output KWR_EN signal equal to 0 to keep the kernel disabled. As the WR_EN signal is asserted, the state machine transits to next state $S2$ with output EN asserted to enable the nine Flipflops and the FIFOs.
- The state machine stays in $S2$ while the kernel KRD_EN is 0 (i.e. still no output from the kernel), once kernel KRD_EN equal to 1, the state machine transits to state $S3$.
- The state machine stays in $S3$ while WR_EN is 1 (i.e. top module is still writing streams to the cache), once WR_EN equal to 0 (i.e., no more pixels to be processed and last pixel has been written to the cache), the state machine transits back to state $S0$ and deassert KWR_EN as an output.

4 Top Module

The top module is defined as in the following figure 8 to display the original image and the processed image to VGA display. To do this, we are using two FIFOs structures; one for the original image, second for the processed image. Image is loaded to both the first FIFOs and the 2D filter at the same time. The second FIFO is loaded with processed pixels from the 2D filter.

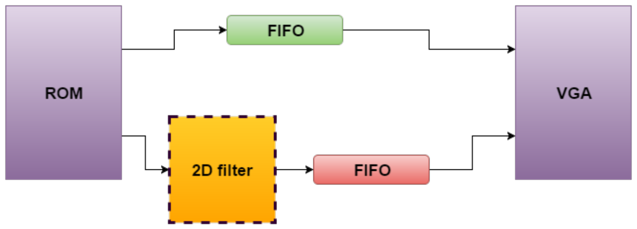


Figure 8: Top Module

We use the same provided state machine with only simple change to it as in figure 9. We make $filter\_wr\_en$ equal to $'1'$ instead of $wr\_en\_fifo\_proc$, and we assign the $wr\_en\_fifo\_proc$ signal to the RD_EN signal of the 2D filter.

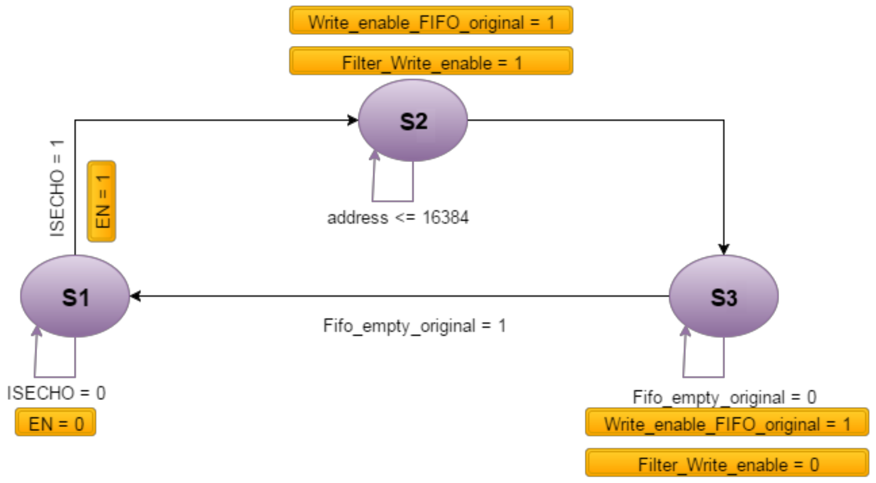


Figure 9: Top Module State Machine

- The state machine starts with initial state is in $S1$. It stays in the same state as long as the switch $ISEcho$ is off. Once the switch is on, the state machine transits to state $S2$ with output EN signal equal to 1 to enable reading from the ROM.
- In state $S2$, the image started to be loaded to the first FIFO and the 2D filter one pixel per clock cycle, with two main asserted outputs $WR\_EN\_FIFO\_ORIG$ to enable writing to first FIFO, and $FILTER\_WR\_EN$ to enable writing to the 2D filter. the state machine stays in this state till the last pixel of the image is loaded, then transits to the last state $S3$ with output EN signal equal to 0 to disable reading from the ROM

- In state $S3$, $WR\_EN\_FIFO\_ORIG$ and $FILTER\_WR\_EN$ signal are deasserted to disable writing to first FIFO and the 2D filter $DATA\_FIFO\_ORIG\_READY$ and $DATA\_FIFO\_PROC\_READY$ signals are asserted to enable displaying the original and processed image to VGA display. The second FIFO $WR\_EN\_FIFO\_PROC$ is deasserted later since it is connected to the 2D filter $FILTER\_RD\_EN$ signal.

5 Testing and Results

5.1 Convolution Mask

The proposed implementation was tested using a 128×128 grayscale image and the following set of known 2D filters

- Delta

Table 1: Delta Kernel

0	0	0
0	1	0
0	0	0

Delta filter does nothing but get the intensity value from anchor position. It simply outputs the same image as input. We have used this filter during debugging phase.

- Blur

Table 2: Blur Kernel

1	1	1
1	0	1
1	1	1

Blur filter is used to remove noise in the image. It takes average of the neighboring pixels and replace it at anchor position. We have used 8 as the normalizing factor for simplicity.

- Sobel Y

Table 3: Sobel-Y Kernel

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

Sobel-Y is used to get edge response in vertical direction. The kernel shown in 3 is normalized by division with 8.

- Sobel X

Table 4: Sobel-X Kernel

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

Sobel-X is used to get edge response in horizontal direction. The kernel shown in 4 is normalized by division with 8.

- Laplacian

Table 5: Laplacian Kernel

0	-1	0
-1	4	-1
0	-1	0

Laplacian is used to get edge response in both the horizontal and vertical directions. The kernel shown in 5 is normalized by division with 4.

5.2 Kernel Results

Since the kernel is an independent module, we tested it independently from others. We used few streams of nine 8-bit values and tested the performance of the kernel.
In figure 10, we can see 6 streams of data subjected to sobelX filter. We can see states of the kernel changing as well. The *read_en* signal is asserted after state *S*6 has been reached and stays there until all the data in stream has gone through the kernel and finally it gets deasserted at state *S*6. We achieve the results after $5\frac{1}{2}$ cycles. The $\frac{1}{2}$ cycle adds up during the immediate multiplication of coefficients with intensity values.

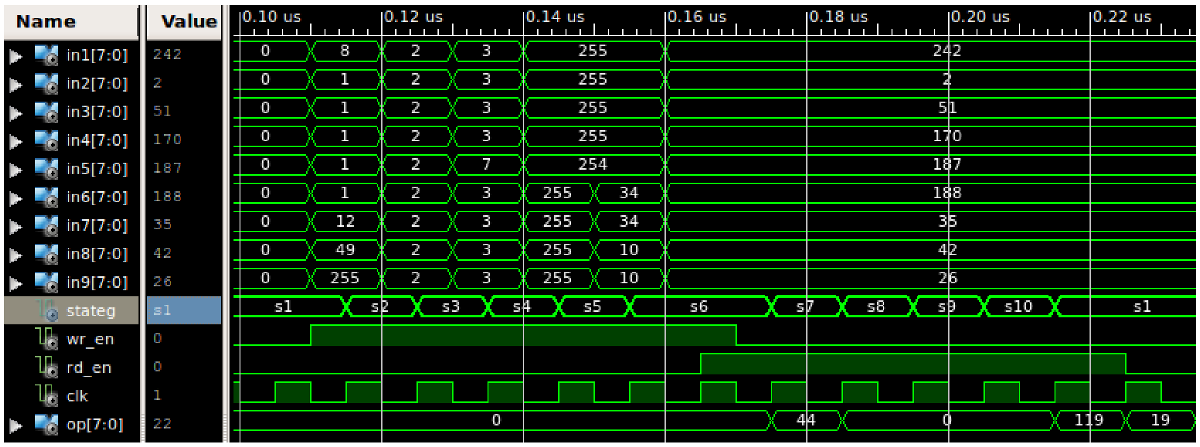


Figure 10: Timing diagram with state transition and write/read assertion

In figure 11, we show what is going inside the kernel in each clock cycle. For the simplicity, we did not show the multiplication stage. Results of each stage is shown in waveform with different colors. The 14-bit sum are shown in cyan , 15-bit sum are shown in magenta, the 16-bit sum are shown in yellow, the 17-bit sum is shown in white and the normalised 8-bit value is shown in red.

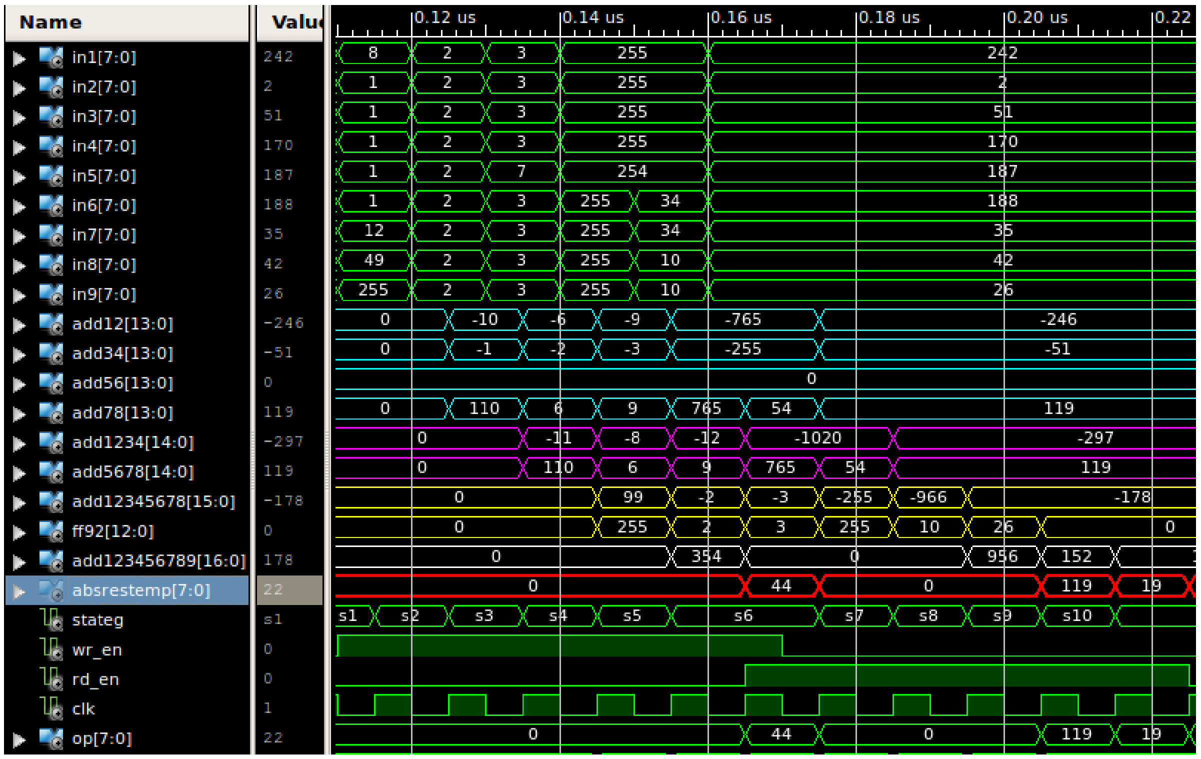


Figure 11: Timing diagram:each stages of arithmetic operations has been shown with waveform of different color; in cyan: sum of 2nd stage; in pink: sum of 3rd stage; in yellow: sum of 4th stage; in white: sum of 5th stage; in red: final output from 6th stage

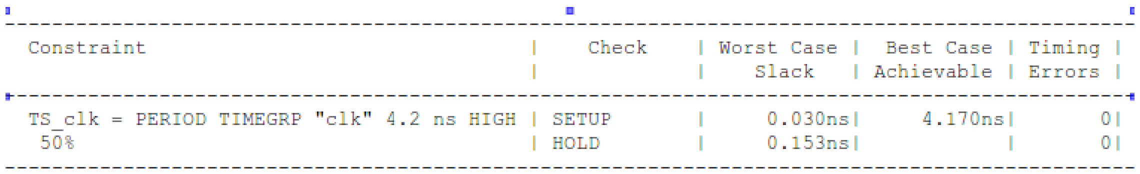

```
------------------------------------------------------------------------------------------------
  Constraint                                 |    Check     | Worst Case |  Best Case | Timing |
                                             |              |    Slack   | Achievable | Errors |
------------------------------------------------------------------------------------------------
  TS_clk = PERIOD TIMEGRP "clk" 4.2 ns HIGH | SETUP        |     0.030ns|     4.170ns|       0|
   50%                                       | HOLD         |     0.153ns|            |       0|
------------------------------------------------------------------------------------------------
```

Figure 12: For clk of 4.2 ns, we achieved the best case without timing errors

5.3 Simulation Results

We have compared our results with an implementation of the 2D filters using MATLAB. the results are shown below. For the five filters, we almost have identical results.

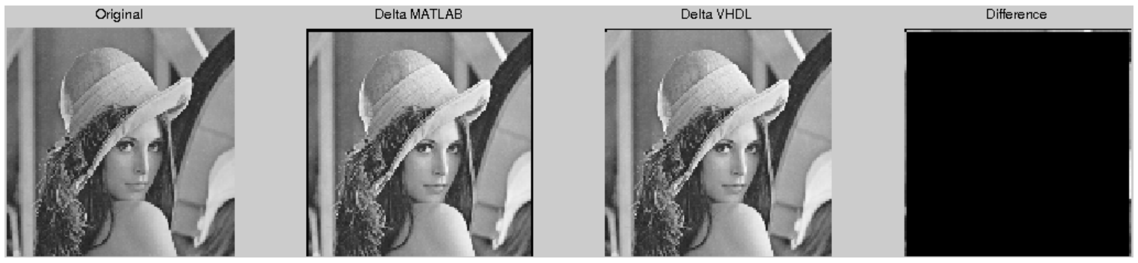


Figure 13: Delta Function: Comparison between MATLAB and VHDL implementation

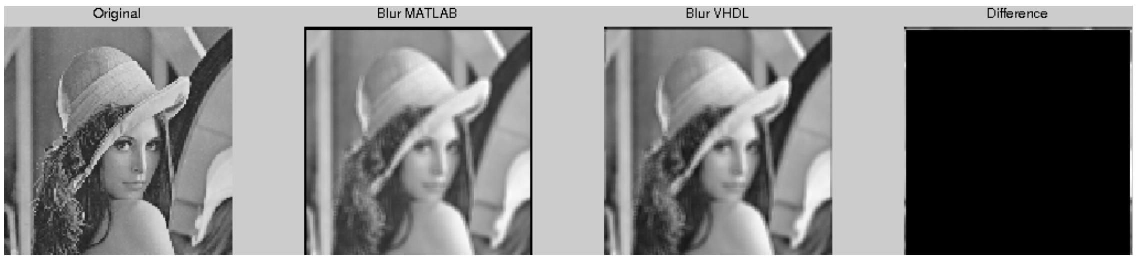


Figure 14: Blurring: Comparison between MATLAB and VHDL implementation

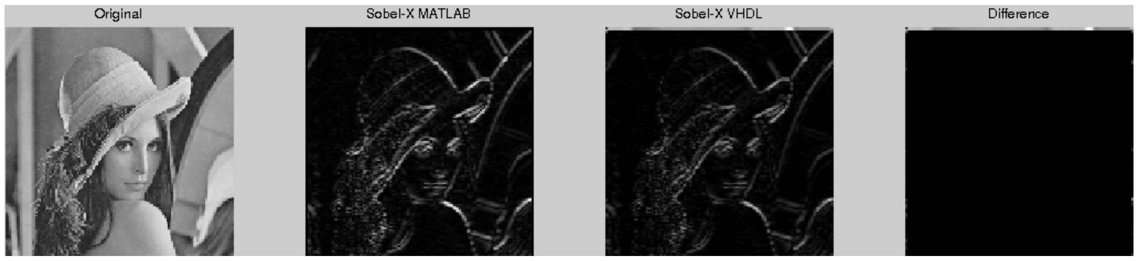


Figure 15: Horizontal Sobel: Comparison between MATLAB and VHDL implementation

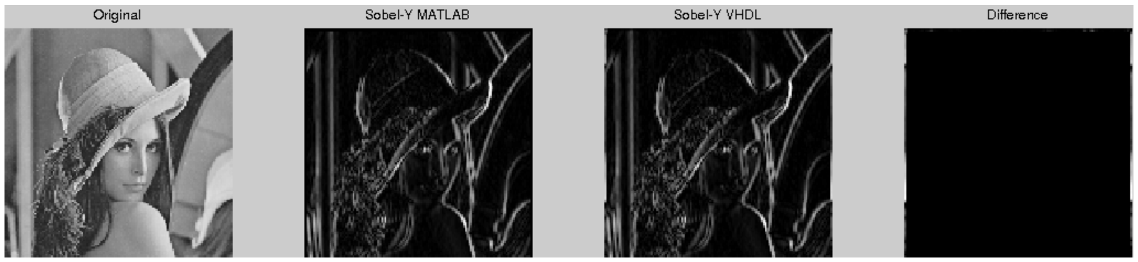


Figure 16: Vertical Sobel: Comparison between MATLAB and VHDL implementation

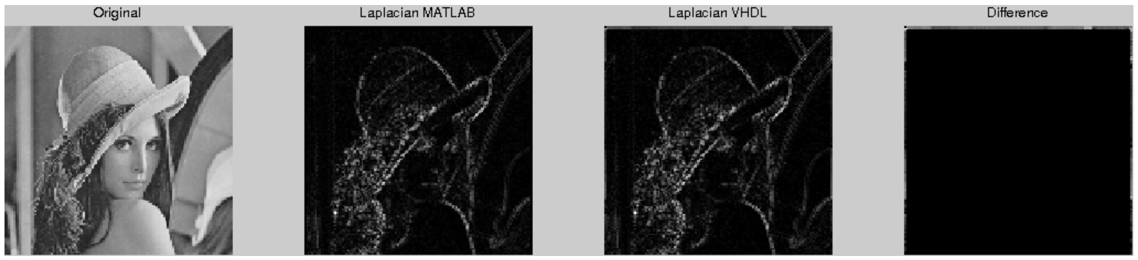
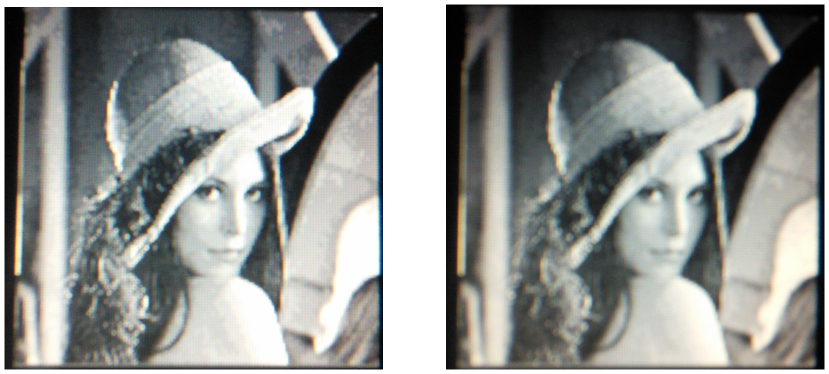


Figure 17: Laplacian: Comparison between MATLAB and VHDL implementation

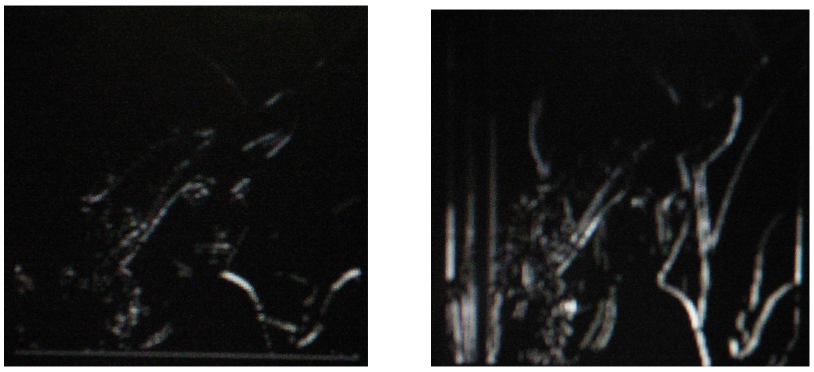
5.4 Final Results

The final results for the different implemented 2D filters on FPGA connected to a computer screen were as follows



(a) flipflops and FIFO (b) Sliding Window

Figure 18: Final FPGA result for Delta and Blur filters



(a) flipflops and FIFO (b) Sliding Window

Figure 19: Final FPGA result for Sobel-X and Sobel-Y filters

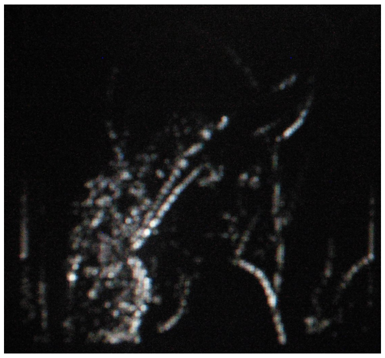


Figure 20: Final FPGA result for laplacian filter

6 Conclusion and Future Work

Our final implementation is motivated by the idea the kernel should be aware of the its own operation. To do so, we implemented a scheduler with state machine, with each state representing arithmetic operations going inside it. We also have implemented the kernel using counter which also produces satisfactory results. We chose to go with state machine because it makes the code more readable and independent of image size. However, on doing so, we put the constraint on the kernel for being of fixed size. In other words, the current kernel size is 3×3 and to increase it means to increase the number of states as well which is quite a work to program in itself. It would be interesting to make the kernel generic in size as well.
For future work for this project, we can try the system with higher images resolutions like 256×256 and 512×512, also try for RGB color images by applying the same filter for each color channel and combine the results, and investigate the possibility of implementing a 2D filter on a stream of frames either stored in the ROM or a camera live stream.

7 APPENDIX

7.1 User Manual

- To load the image, use the $switch$ labeled by $U9$
- For the filter selection, we have the last 3 $switches$ to labeled by $R3$, $P3$ and $P4$ to select the filter with $R3$ the least significant bit and $P4$ as the most significant bit.
- For Delta Filter, filter selection - 000.
- For Blur Filter, filter selection - 001.
- For SobelX Filter, filter selection - 010.
- For SobelY Filter, filter selection - 010.
- For Laplacian Filter, filter selection - 100.