University of Burgundy

Master of Science in Computer Vision – 2nd Year

Real Time Imaging and Control Module

2D Filter Implementation on Images using FPGA and VHDL

by

Francois Legrand

Vamshi Kodipaka

Olivier Agbohoui

Dated: 2nd January 2020

Supervisor:

Dr. LeMaitre Cedric

CONTENTS

1. Introduction

In Real time, we desire every process and technology to sophisticate us in real time. This is the basis for the "Real Time Imaging and Control" module. We acquire images in real time, thanks to many acquisition devices [1]. Now, these acquired images in almost processed real time for certain applications. For such, we turn to Digital Signal Processors (DSPs) or Field Programmable Gate Arrays (FPGAs). DSPs take a digital signal and process it to improve the signal into clearer sound, faster data or sharper images and making it ideal for applications that can't tolerate delays. As the result complexity increases and demands efficient hardware platforms grows. FPGA has gained a lot of traction in the real-time community, as a replacement for the traditional DSP solutions.

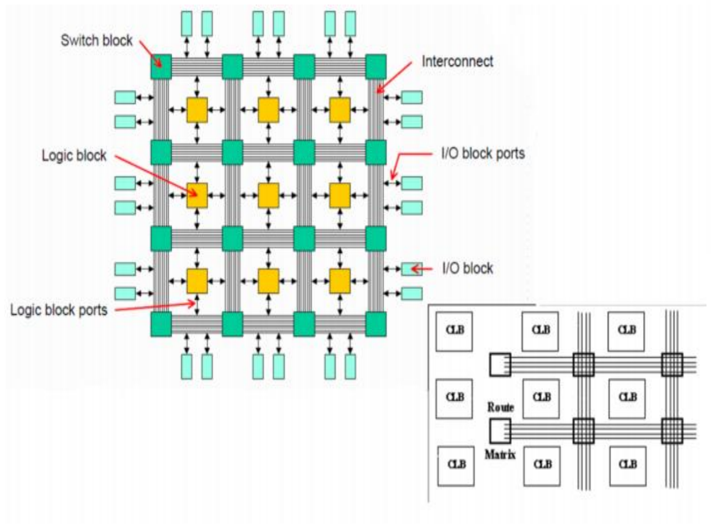


Fig.1(a): FPGA Structure

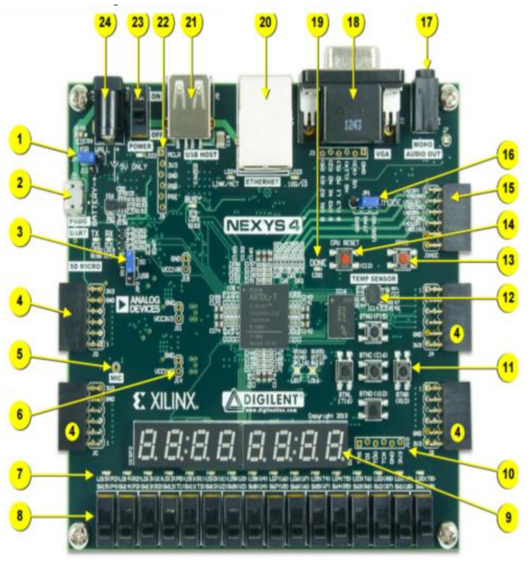


Fig.1(b): Nexsys4 board-FPGA

FPGAs are re-programmable silicon chips has logic blocks and programmable routing resources. One can configure these chips by custom hardware functionality and can develop digital computing using HDL programming and compile them down to a configuration file or bit-stream that shows how to wire components together. In addition, FPGAs are completely reconfigurable and instantly re-compliable circuit configurations. They provide hardware-timed speed and reliability. Unlike processors, they are truly parallel in nature with independent processing task on logic blocks. Consequently, the performance of one part of the application is not affected to another. FPGAs are indeed revolutionizing image and signal processing due to their advancements like re-configurability and parallel processing. So, we acquire images in real time and use the parallel processing capability of FPGA to solve our sophistications.

Specifications: 1. 240 DSP Slices, 2. 16 user Switches, 3. Analog to Digital Converter (XADC), 4. 4860k bits of fast block RAM, 5. 15850 logic slices, 6. Internal clock speed exceeds 450MHZ. Check datasheet.

From Fig. 1(b) we note; it has USB, ethernet,12-bit VGA and other ports, built in devices and sensors as: accelerometer, temperature sensor, MEMs digital microphone, speaker amplifier, and a lot of IO devices. Suppose, real time video with frame rates 25 per second requires millions of operations per second on every pixel even for a small gray-scale image. The performance depends on limitation of a conventional serial processor. To overcome this problem of FPGA we it as substitute. An FPGA is a matrix of logic blocks that are connected by a switching network. The logic blocks has parallelism exists in: spatial parallelism and temporal parallelism.

1.1 Project Task

Our task is to implement 2D filter to process the images using FPGA and VHDL (Very High Speed Integrated Circuit Hardware Description Language). FPGA is a compromise between the flexibility of general purpose processors and the hardware based speed of ASICs (application specific integrated circuit) [2]. Here, Nexys 4 Artix-7 FPGA Board used for image processing task with different kernels on image of size 128X128 resolution and the kernel size is 3*3 pixels. FPGA can be programmed to partition an image and distribute them to multiple pipelines all of which could process data concurrently. Output of this task can be observed via 12-bit VGA port which is built-in the board. Software for simulation and implementation is Xilinx ISE Design Suite and language is VHDL. Methodology and Results are demonstrated in future sections.

1.2 Xilinx ISE and VHDL

Xilinx ISE is a software produced by Xilinx for synthesis & analysis of HDL designs, perform timing analysis, examine RTL diagrams and configure the target device with the programs. Xilinx ISE is a design environment for FPGA and is tightly-coupled to the architecture of such chips specific to vendors. The development of VHDL was initiated in 1981 by US Department of Defense to address the hardware life cycle crisis [3].

VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems like FPGA. VHDL is used mostly for writing text models that describe a logic circuit. Such a model is processed by a synthesis, only if it is part of the logic design. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design. This collection of simulation models is commonly called a *testbench*.

2. Strategy

2D filter implementation task is split into two main parts, **Cache memory** and **Processing Unit**.

1. Cache Memory: As the camera gives us the pixels in sequential order we need a logic structure which aims to temporarily store the data before processing and for simultaneous accessing of pixels by the processing unit.

2. Processing unit: It access the pixels provided by the cache memory and has a pipeline structure perform convolution and to process pixels according to the kernel applied.

Firstly, kernel to be scanned/moved through each pixel of the image, do the mathematics and replace the center pixel value with the corresponding output. One way to do this operation in FPGA requires all the image pixels to be accessed before using Cache. Now, we require a portion of image pixels to be loaded in cache with the help of FIFO that depends on the image size and mask dimension. Only 259 pixels are required to be accessed/loaded in cache before the system starts doing the operations and sends the first output pixel to another memory block for a processed image. This number is given as follows:

pixels required = (image width _ (kernel height - 1)) + kernel width

1. To design a cache that stores neighborhood pixels for the mask multiplication. *As Fig. 2(b)*
2. Multiply the mask, add respective pixels and divide by appropriate number to get final pixel.
3. To display/synchronize the output pixel values to VGA port.

2.1 Cache Memory

The basic use of cache memory is to store data that are frequently re-referenced by software during operation. Fast access of cache increases the overall speed. So, the simultaneous pixel accesses enables a 3x3 pixel neighborhood to be accessible in one clock cycle. The structure is based on Delay flip-flop (D-FF) registers and First-In-First-Out (FIFO) memory. The structure we used is shown in Fig.2.

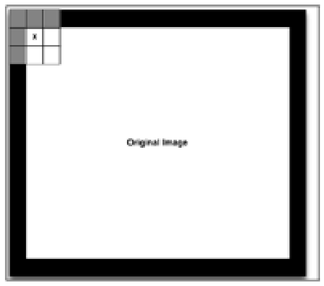
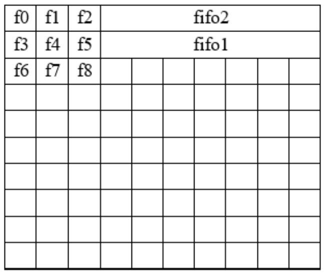


Fig. 2(a). Neighborhood access



f0	f1	f2	fifo2						
f3	f4	f5	fifo1						
f6	f7	f8							

Fig. 2(b) Storage in Cache memory- 1st clock cycle

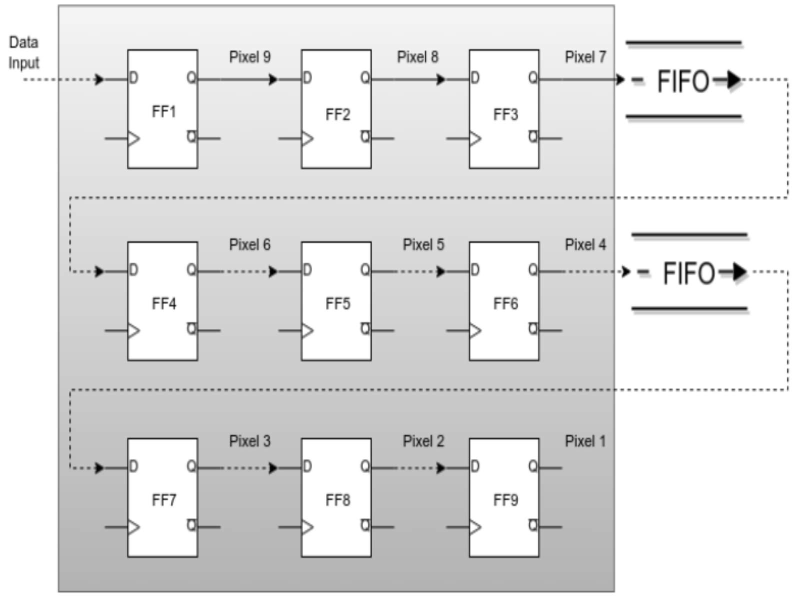
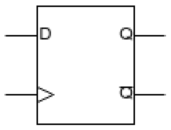


Fig. 2 Cache Memory

Remember, D-Flip Flops are used as shift registers. *Quickly recollecting the D-Flip Flop.*



Clock	D	Q_{next}
Rising edge	0	0
Rising edge	1	1
Non-Rising	X	Q

```
flip_flop : process (CLK, R)
begin

if (R = '1') then temp <= (others => '0');
elsif rising_edge(CLK) then
        if(EN = '1') then
            temp <= D;
    end if;
end if;
end process flip_flop;

Q <= temp;
```

Fig.3 As per program and logic, it uses a temporary signal as a buffer to carry the hardware input from D as long as Reset (R) is zero and Enable (EN) is high on each rising edge of the clock. As soon as the process is completed buffer shifts its contents to output Q.

In practice, a serial to parallel shift is required to push 8-bits of each pixel one by one, to the cache and after when all 259 pixels (from eqn. 1) are pushed to the memory pipeline, nine pixels that corresponds to the position of mask applied should be pulled-out. This idea can be well understood by the following illustration Fig.2. The input data is pushed from the First Flip-Flop FF1 assuming that all the Flip-Flops are synchronous sharing the same clock signal. As soon as the valid data is received by the last Flip-Flop FF9 (after 259 clocks), the output values i.e. Pixel1-Pixel9 will be ported to the filter component for the mathematical operations. Design in Fig. 2 should work, but it is still not convenient to store all the pixels in individual memory units (Flip-Flops) when the system requires the pixel values that corresponds to the mask only. An efficient way is to use a FIFO to store all the pixels for each row of image except for those which corresponds to the kernel width. Thus FF1, FF2 and FF3 should stay and rest of the FFs of the first row should be replaced by a FIFO of depth 125 bytes. This should follow for the second row also.

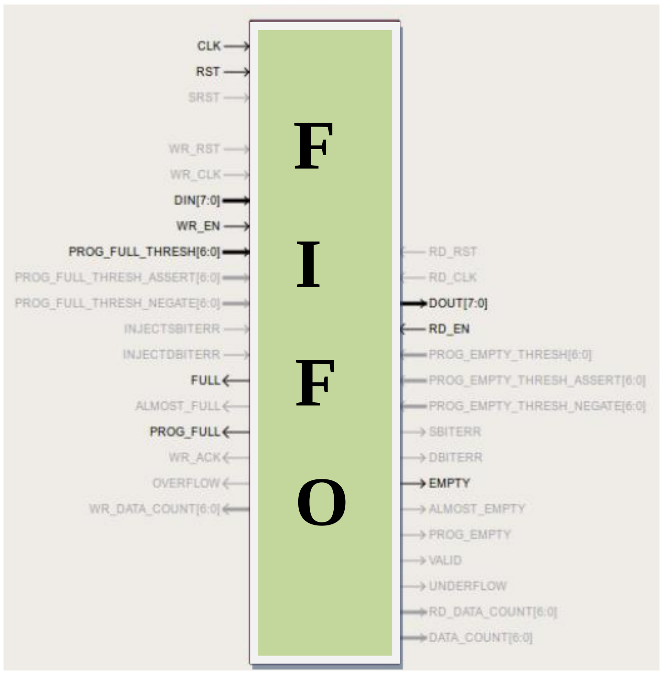
```
FF1: FFname generic map (8) port map (CLK => CLK, R => '0', EN => '1', D => I, Q => Q1);
FF2: FFname generic map (8) port map (CLK => CLK, R => '0', EN => '1', D => Q1, Q => Q2);
FF3: FFname generic map (8) port map (CLK => CLK, R => '0', EN => '1', D => Q2, Q => Q3);
FIFO1: bt_fifo port map (clk => CLK, rst => '0', din => Q3, wr_en => wr_en1, rd_en => prog_full1,
        prog_full_thresh => pft, dout => b1, full => full_1, empty => empty1, prog_full => prog_full1);
```

This code use a generic Flip-Flop as in Fig.3, as an 8-bit Flip-Flop with Reset always OFF and Enable always ON. The input byte from the memory is given as input to FF1 and the output from FF3 is given as input to the first FIFO1. FIFO is generated using IP (CORE Generator) provided by XILINX with write width equals 8 (corresponds to 8-bit pixel value) and write depth equals 128 (corresponds to the image width that is 128 pixels).

FIFO has a Single Programmable Full Threshold Input Port that sets a flag *prog_full* when the number of bytes exceeds the threshold value. The threshold can be set in *prog_full_thresh*, in our case it is 123 in decimal. When the memory reaches 123 bytes, it will raise the flag in the next clock cycle (making total number of bytes in FIFO as 124). The flag *prog_full* is connected to *rd_en* i.e. read enable signal of FIFO itself, which allows to read data out from the *dout* port. Thus in the next clock cycle 128th byte is shifted in FF1 (still in the first row) whereas the first one is fed to FF4. This is how the timing of byte shift is synchronized. These numbers relates with the size of image and kernel size as described by the following equation:

$$prog\_full\_thresh = image\_width - kernel\_width - 2$$

This applies to the second row of the cache also. As the first pixel of the image is pushed until it reaches FF9, the outputs of all flip-flops are then transferred to mask component that do all the mathematics to apply the kernel and sends the processed output to our system. Schematic of a FIFO is in Fig.4.



2.2 Processing Unit

By the virtue of Section 2.1, we now have the proper access to pixels for the processing. As described earlier, we need to apply 3*3 kernel on the image window, so we need multipliers to multiply the kernel value with pixel value and then add all the values and then divide by the coefficient as defined by the kernel. Here, we require 9 multipliers to multiply kernel value and pixel value. For the addition part, we implemented a pipeline like structure with levels as seen in Fig. 5 because to enhance the processing speed of the processing unit as in this type of design we can simultaneously get the added output after every clock cycle. For the multipliers used in level 1, we used one operand as unsigned 8 bit pixel values and other operand as signed 4-bit (kernel values) and the result of multiplier is 12 bit in length. *The multipliers latency is set to one clock cycle* in order to read nine pixels from the cache memory after every clock cycle.

The adders at level 2, are signed 12 bit addition whose result is 13 bit signed value. The adders at level 3, are signed 13 bit addition whose result is 14 bit signed value. The adders at level 4, are signed 14 bit addition whose result is 15 bit signed value. The adder at level 5, are signed 15 bit addition whose result is 16 bit signed value. The *adders-latency is one clock cycle* for the synchronous implementation of pipeline structure. In the design the ninth pixel is not added to any pixel until the level 4, and at level 5 it is added using the 'adder8' after extending its sign for bit compatibility. To make this pixel in synchronous to other pixel values, 3 FFs (f1, f2 and f3) are used to delay the pixel 9 input to the 'adder8'.

The divider at level 6, takes 16 bit signed divided and 6 bit signed divisor and gives out 16 bit quotient. The latency of the divider is 20 clock cycles. So, after 20 clock cycles of divider input, the output is available at divider output pins.

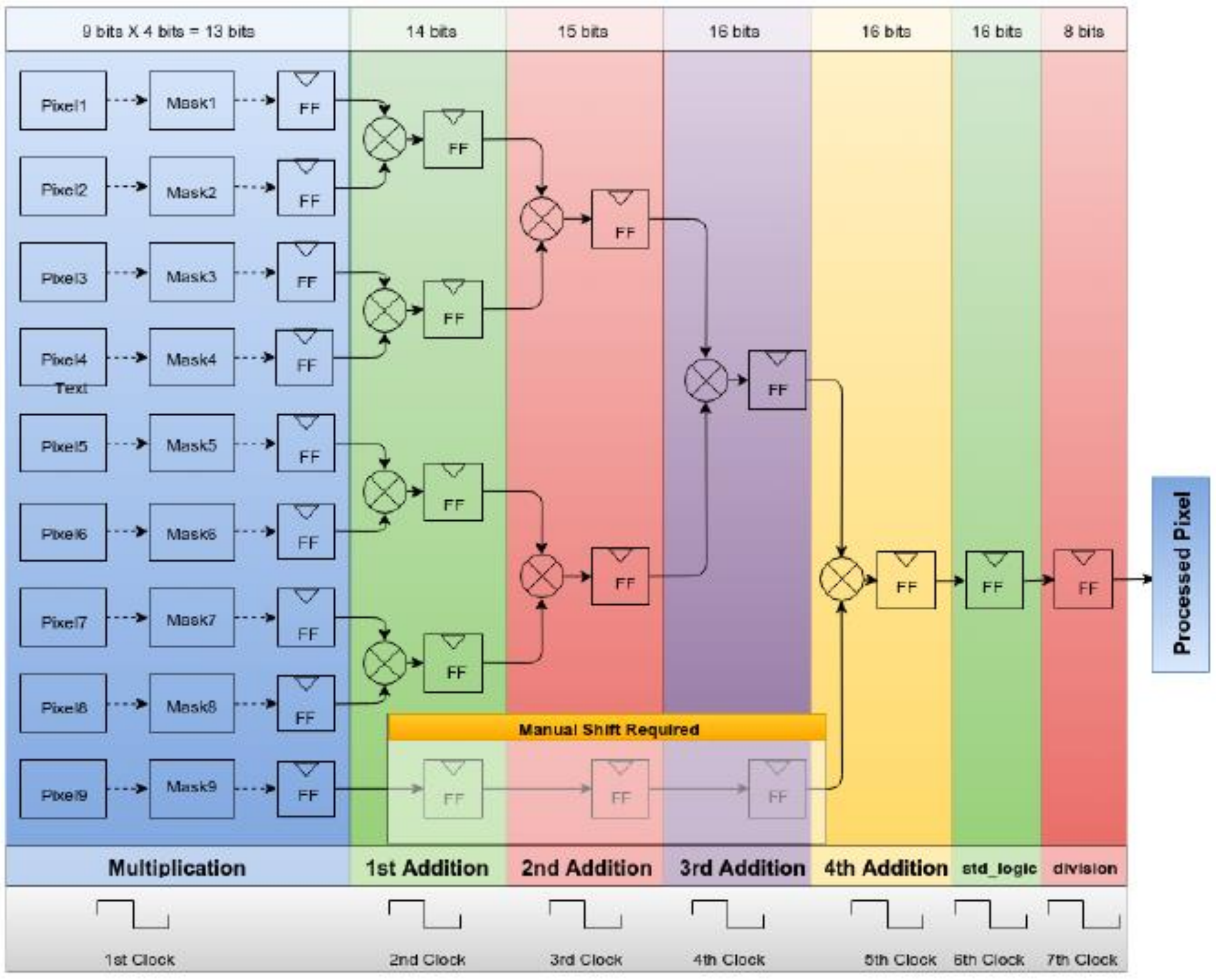
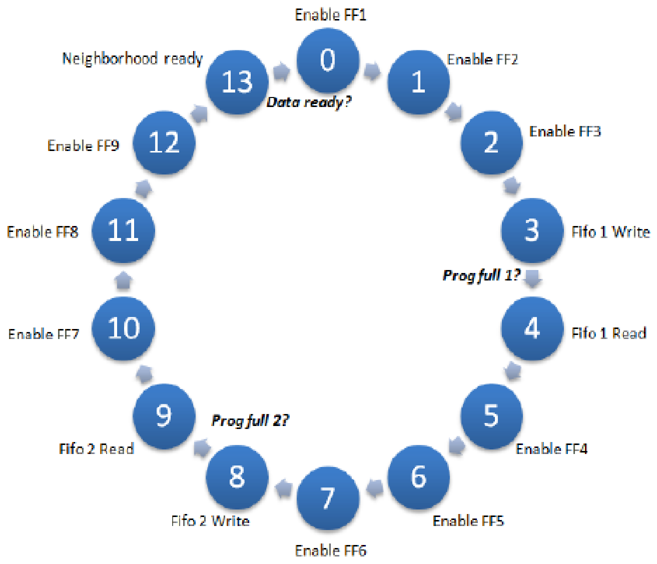


Fig. 5 Kernel Convolution Pipeline - Processing Unit

The multipliers and adders from level 1 to level 5 takes 5 clock cycles and divider takes 20 clock cycles, *so in total the processing unit takes 25 clock cycles to get the output after* the input is fed.

2.3 Control structure

We implemented read and write process module in our simulation for reading the data of image from a 'dat' file and writing the result after processing to another 'dat' file. The complete block diagram and process flow is as shown in Fig.6.



3. Results and Discussion

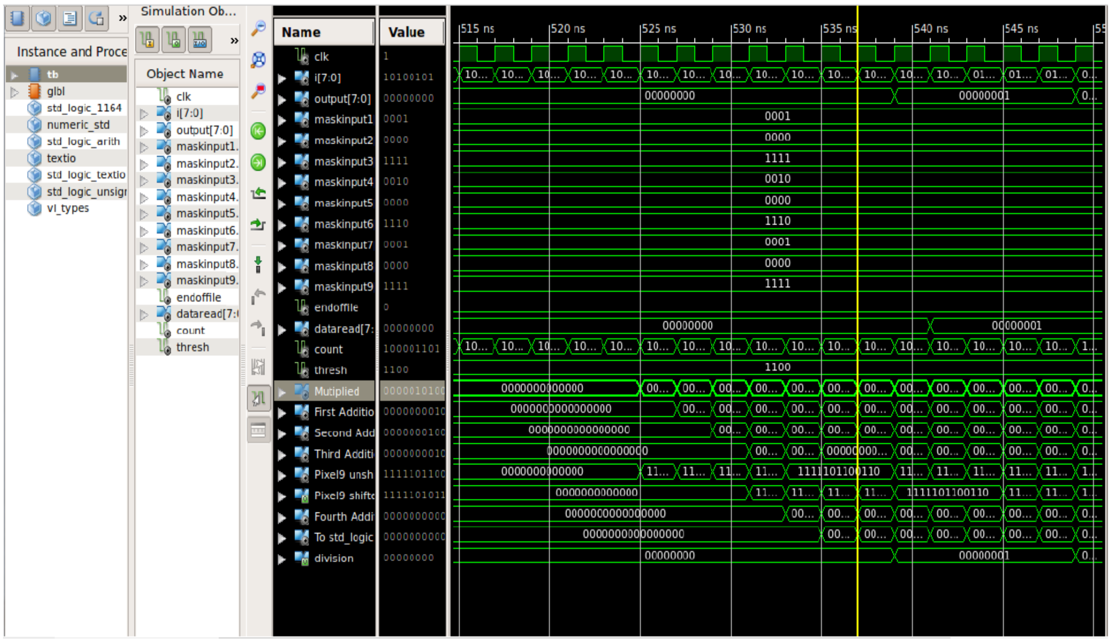


Fig.7 Test bench diagram Simulation Timing Diagram

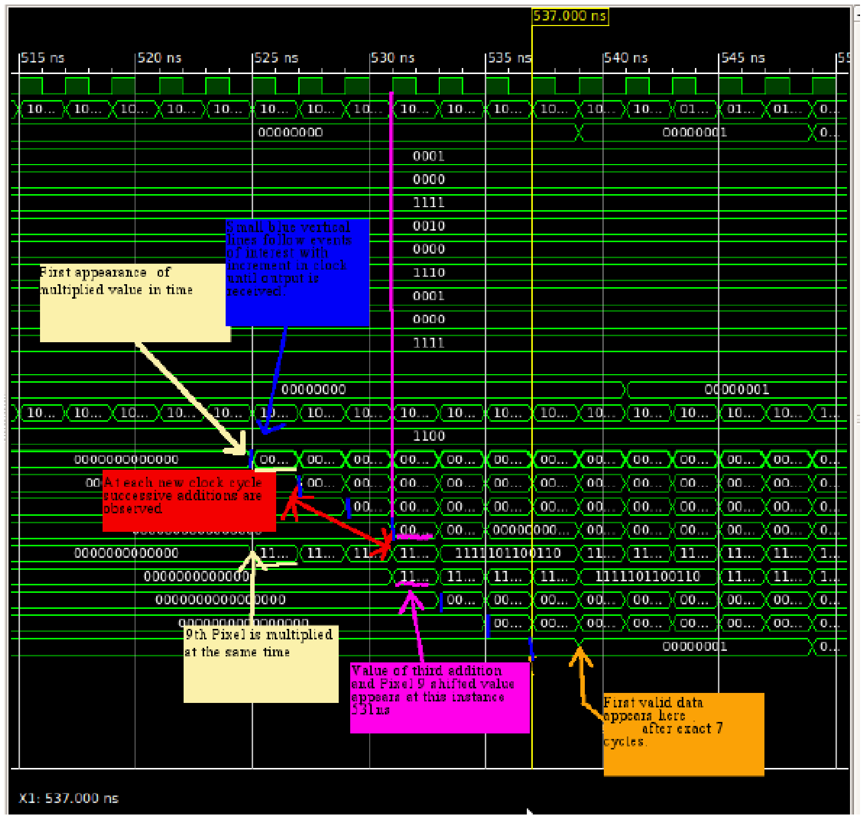
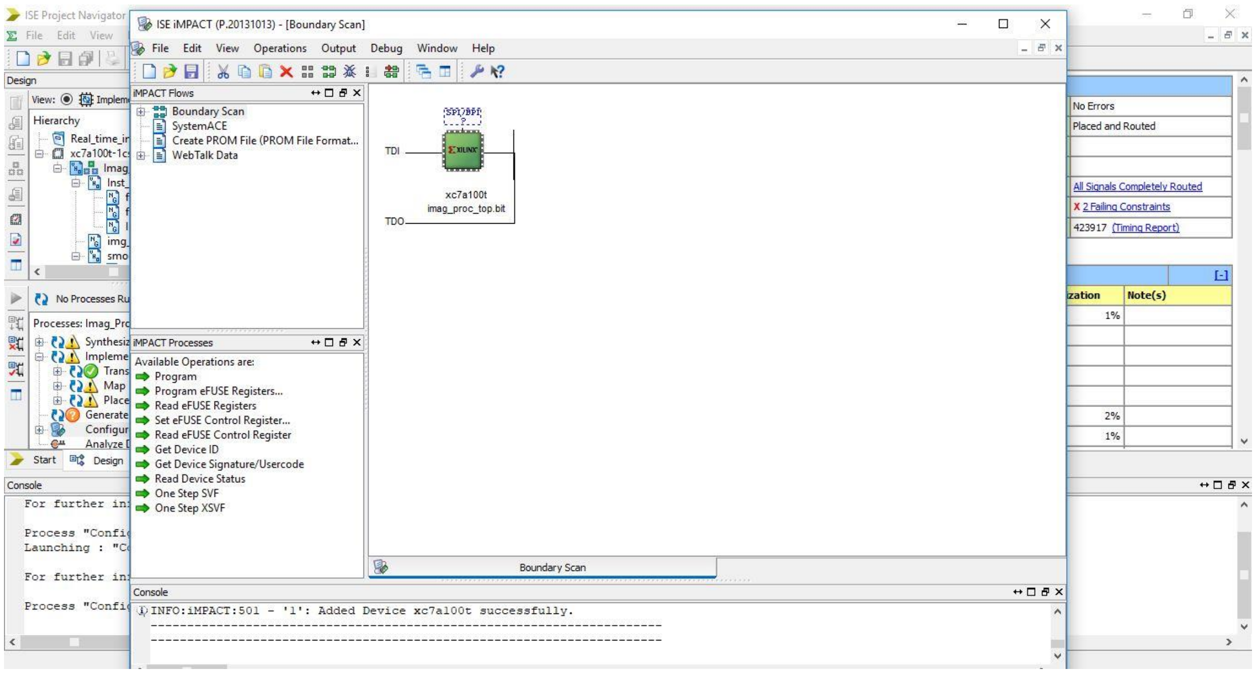

537.000 ns
515 ns
520 ns
525 ns
530 ns
535 ns
540 ns
545 ns
First appearance of multiplied value in time
Small blue vertical lines follow events of interest with increment in clock until output is received.
At each new clock cycle successive additions are observed
9th Pixel is multiplied at the same time
Value of third addition and Pixel 9 shifted value appears at this instance 531ns
First valid data appears here after exact 7 cycles.
X1: 537.000 ns

Fig.8 Test bench Output

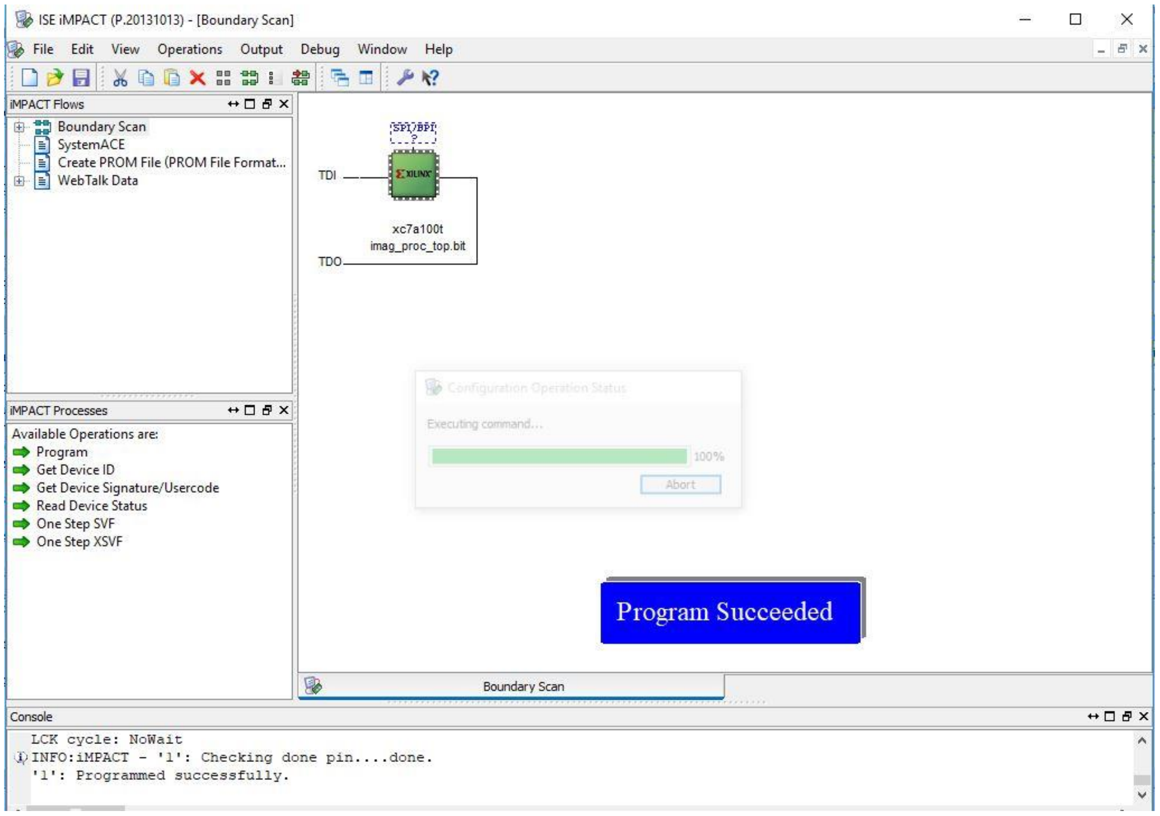

ISE iMPACT (P.20131013) - [Boundary Scan]
File Edit View Operations Output Debug Window Help
iMPACT Flows
Boundary Scan
SystemACE
Create PROM File (PROM File Format...
WebTalk Data
iMPACT Processes
Available Operations are:
Program
Program eFUSE Registers...
Read eFUSE Registers
Set eFUSE Control Register...
Read eFUSE Control Register
Get Device ID
Get Device Signature/Usercode
Read Device Status
One Step SVF
One Step XSVF
SPI/BPI
TDI
TDO
xc7a100t
imag_proc_top.bit
Boundary Scan
Console
No Errors
Placed and Routed
All Signals Completely Routed
2 Failing Constraints
423917 (Timing Report)

```
INFO:iMPACT:501 - '1': Added Device xc7a100t successfully.
----------------------------------------------------------------------
----------------------------------------------------------------------
```

Fig.9 Boundary Scan to find the FPGA board

To Test FPGA:

Tools> iMPACT > Double click Boundary Scan > Initialize Chain. You will see Fig.10.



```
LCK cycle: NoWait
INFO:iMPACT - '1': Checking done pin....done.
'1': Programmed successfully.
```

Fig.10 While and after the search for FPGA board connection (if board found)

3.2 MATLAB Simulation

3.2.1 Sobel Filter

The result after applying sobel filter as shown below, is as shown in fig. 23.

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

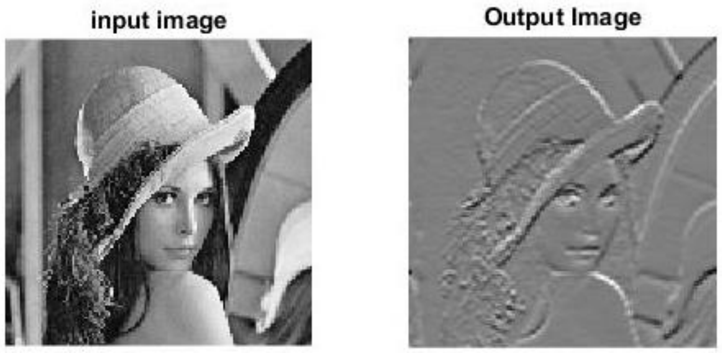


Fig.11 Result after applying sobel kernel shown above

The result after applying sobel filter as shown below, is as shown in fig. 24.

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

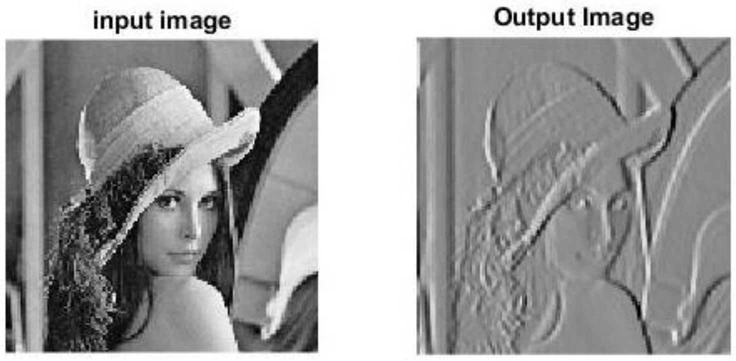


Fig.12 Result after applying sobel kernel shown above

The result after applying diagonal sobel filter as shown below, is as shown in fig. 25.

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

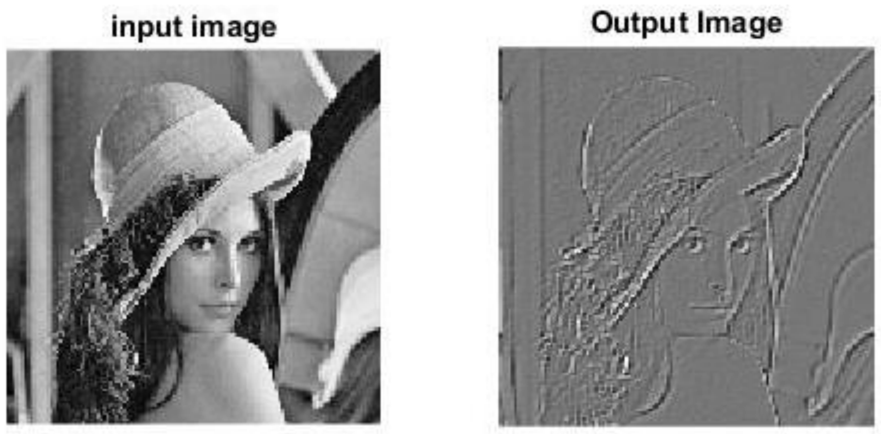


Fig. 13 Result after applying sobel kernel shown above

3.2.2 Gaussian Filter

The result after applying Gaussian kernel as shown below, is as shown in fig. 26.

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

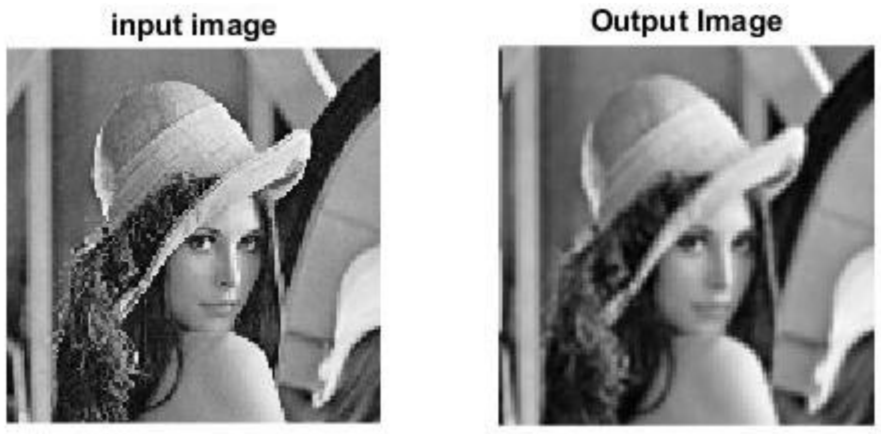


Fig. 14 Result after applying Gaussian kernel shown above

3.2.1 Average Filter

The results after applying average filter kernel as shown below, is as shown in fig. 21.

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

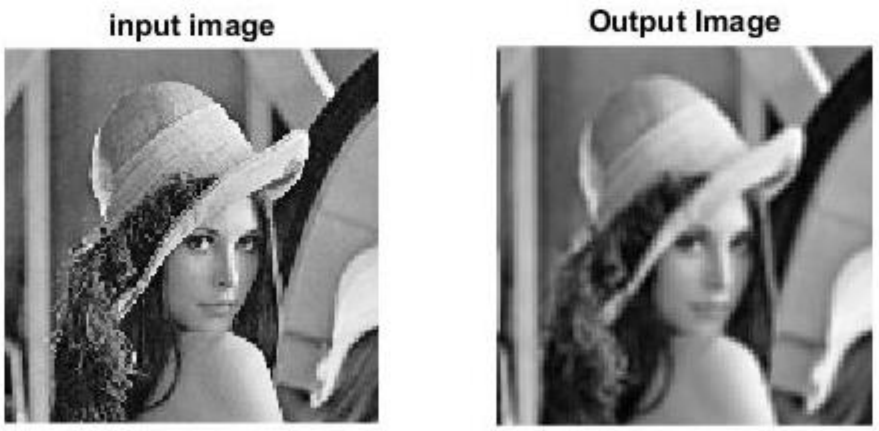


Fig.15 Result after applying average kernel shown above

3.3. Display through VGA

Though VGA output is not discussed in this report in detail, however a brief explanation on how it is done in this project is presented in this section. Since the scope of this project is limited to real-time image processing of a still picture, thus it implies that the image has to be stored in the memory of the chip. In this project, the top module of vhdl project is already provided that used a block memory generator to build a Single Port ROM with read width of 8bits and read depth of 128X128 pixels. A .coefficient of the image is provided and should be uploaded to the IP Core Generator when making a ROM. The pixels are picked up, one by one, from the ROM while monitoring its address bits. This bytes are then transferred to another vhd file that is responsible to store these bytes along with a processed-pixel byte into separate FIFOs. As soon as the FIFOs are full, the data is displayed on the screen and the program starts again from the ROM address 0.

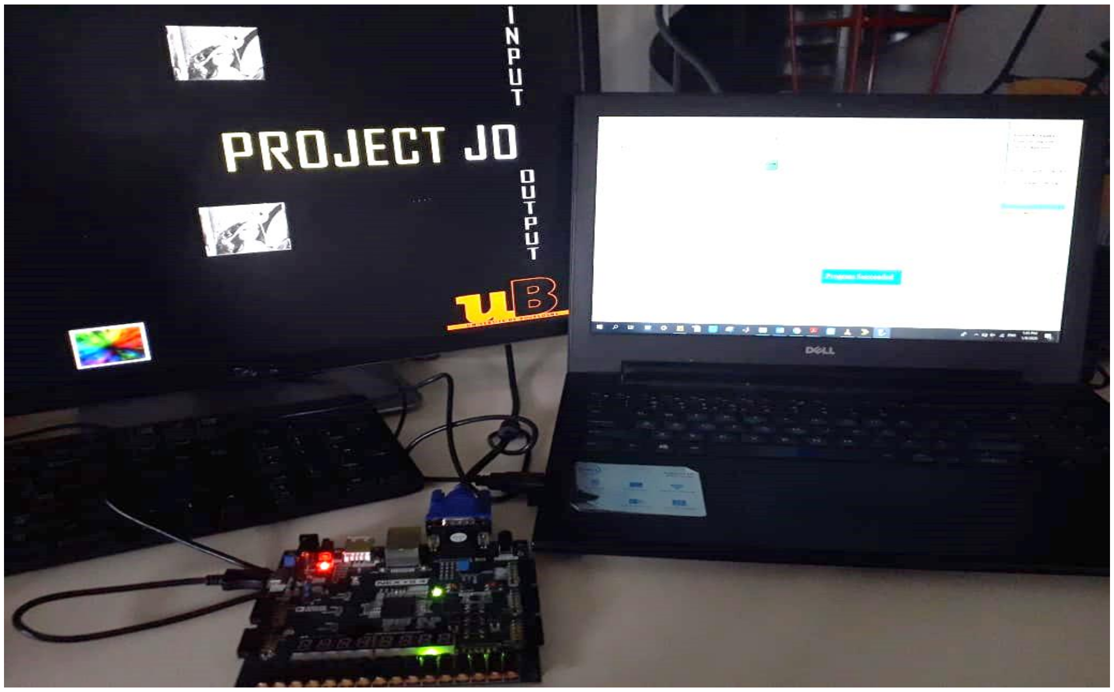


Fig.16 Setup for the Project

The state machine is already implemented for this project, and the code which is developed for this project is integrated with it to work. If the MATLAB results are observed closely, it is seen some pixels of the image are shifted to the other side. This happens because the system designed is always write enabled, however due to a delay of 7 cycles while the kernel do the mathematics creates a shift of 7 cycles. If the system is not efficient, the delay is higher so is the shifting of the pixels. In the strategy adopted in this project, the FIFO that stores the processed byte of the image is not write enabled until it compensates for the garbage and delayed values. This period of time depends on the size of image and the size of kernel. For a 128X128 sized picture with a 3X3 mask this number is 128+128+3+7 which equals 266 bytes. Thus a state is added in the program that waits until 266 bytes are read from the ROM and then after it starts writing the processed bytes in the processed data FIFO. Figure 9 shows the output of the project on the screen through VGA.

OPERATION	ClOCK CYCLES REQUIRED	TIME (ns)
Cache	128 + 128 + 3 = 259	518
Flag Check before Bytes are transferred to Kernel	1 (for checking non-zero value in FF9 output) + 1 (for raising one flag)	4
First Multiplication	1	2
Total	524 ns	

Fig.17 Time calculation for result of first multiplied value

3.4 2D Filter Outputs – Implementation Results

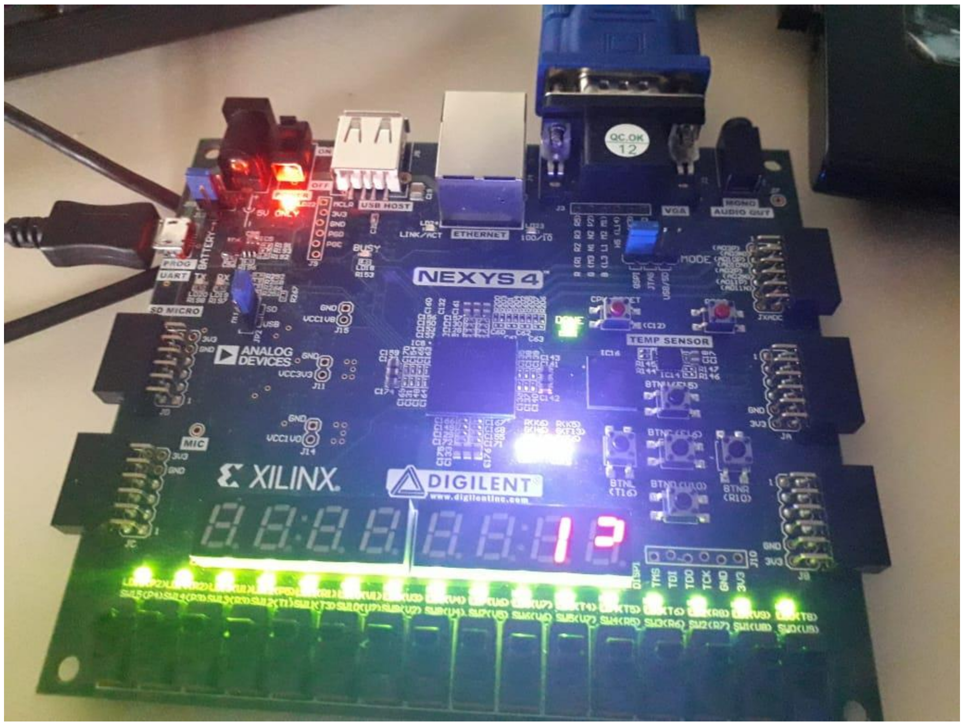


Fig.18 Testing FPGA

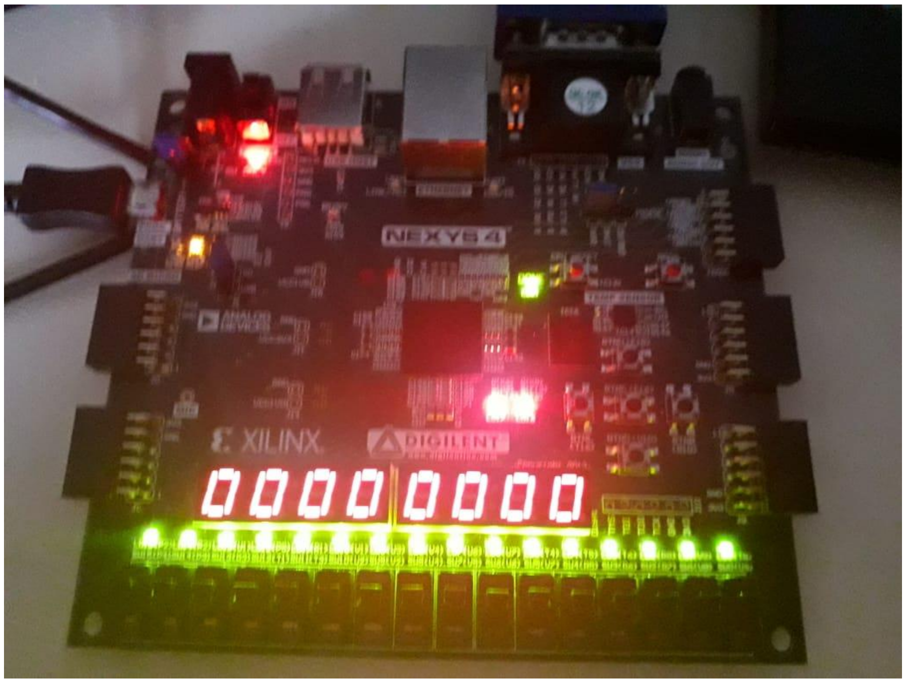


Fig.19 Initializing the Setup

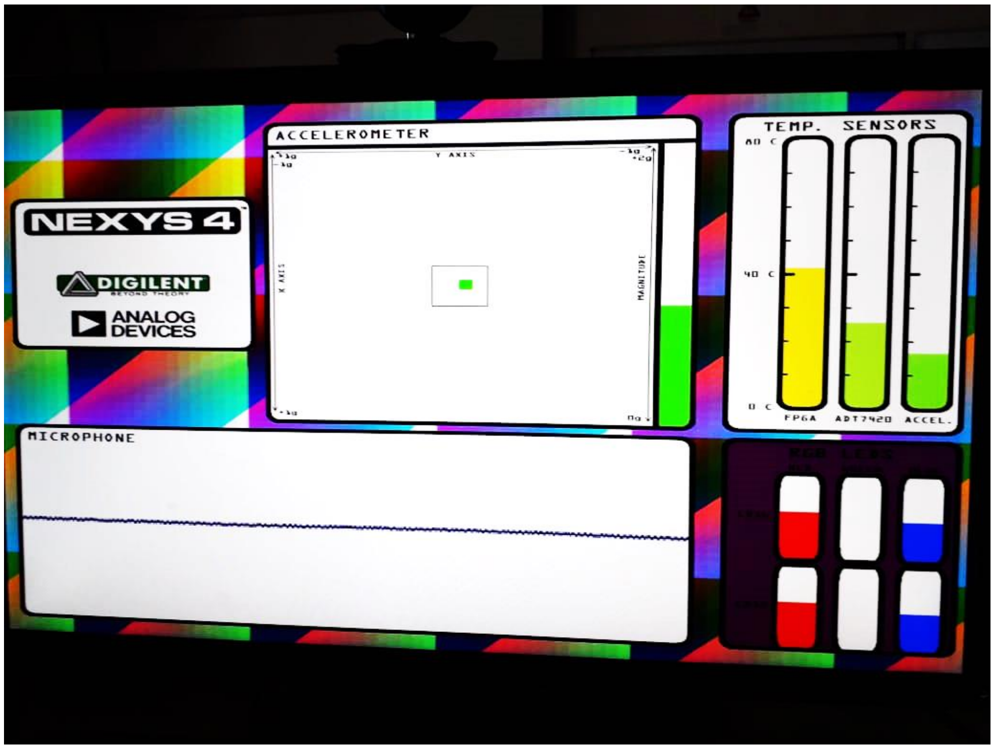


Fig.20 Default screen output once connected to FPGA before the program installed

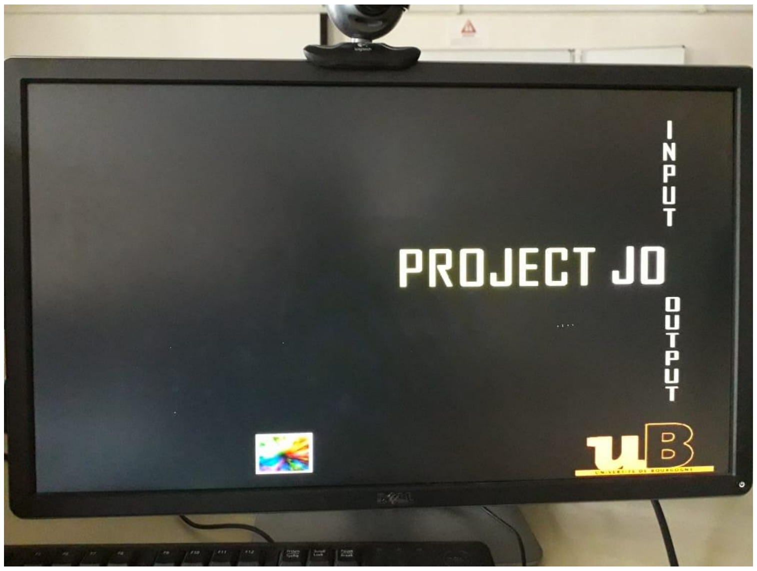
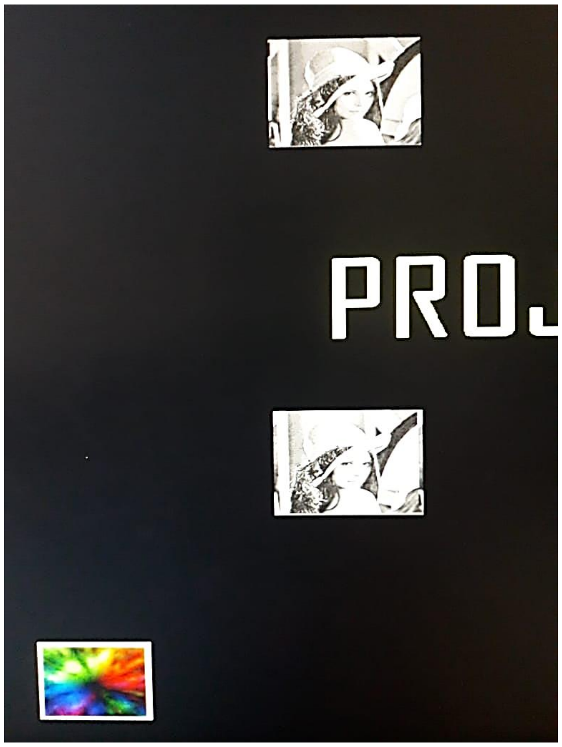


Fig.21 All connections are properly set and ready for the program installation

Programmed outputs displayed:



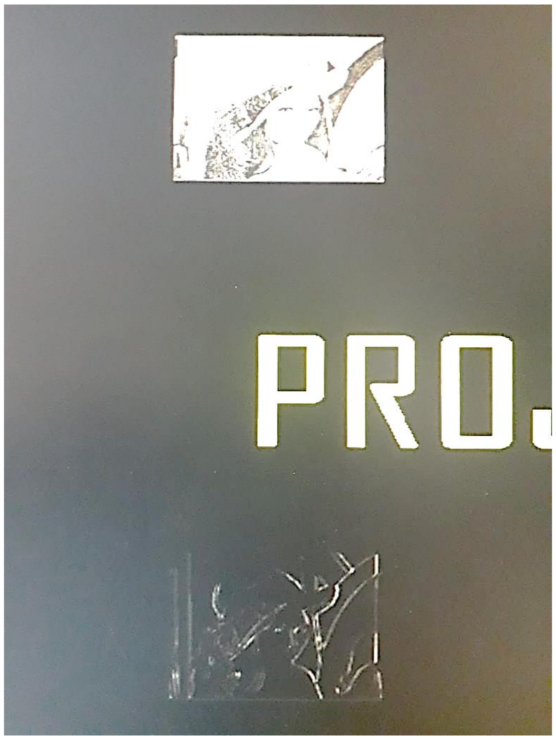


Fig.22 Initial image loaded when FPGA is ON

Fig.23 Sobel-y output on screen

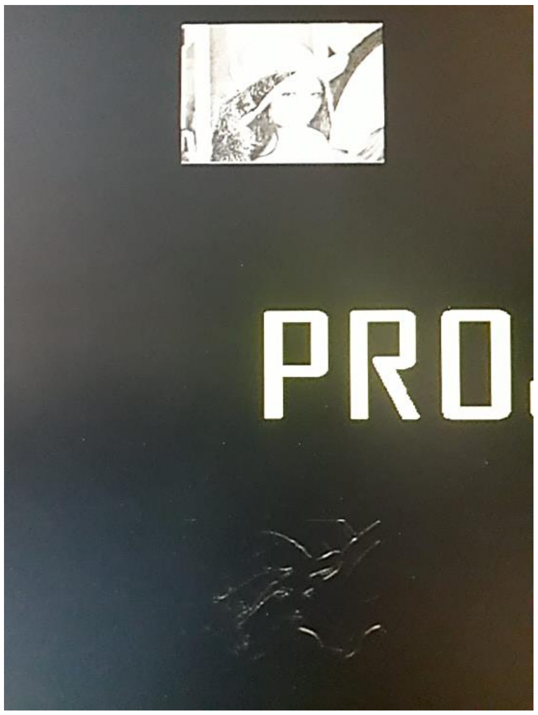


Fig.24 Sobel-x output on screen

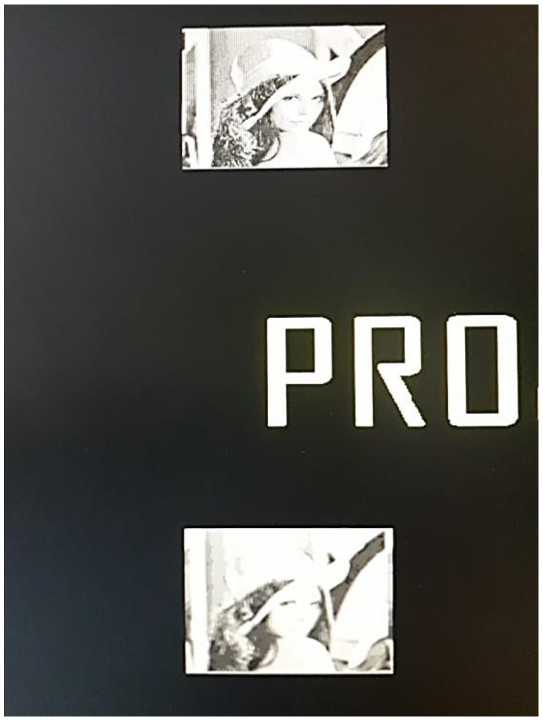


Fig.25 Gaussian output on screen

4. Conclusion

We have simulated the 2D filter on images of Lena to accelerate the processing of images using FPGA and VHDL. We have developed cache memory for simultaneous accessing of pixels of 3*3 neighborhood and we have designed a pipelined architecture of processing unit to speed up the complete processing of data. Our strategy took 16,410 clock cycles to read, process and write an image of 128*128 pixels in resolution. If we have an FPGA with 400 MHz clock frequency it usually takes 41 ms to process the complete image.

Project video: https://www.youtube.com/watch?v=j0Ibz0MF99c

5. REFERENCES

[1] C. T. Johnston, K. T. Gribbon, D. G. Bailey Implementing Image Processing Algorithms on FPGAs.

[2] Ye, L., Yao, K., Hang, J. et al., "A hardware solution for real-time image acquisition systems based on GigE camera", in *Journal of Real Time Image Processing*, Vol. 12, Issue 4, pp. 827-834, Dec. 2016.

[3] Xilinix ISE - Wikipedia

[4] https://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/

[5] https://github.com/OsamaMazhar/2DFiltersFPGA