Real-time Image Processing using FPGA Université de Bourgogne

Osama Mazhar
MSCV6

15th December, 2015

Contents

List of Figures

1 Introduction

1.1 General

A 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 of the system is limited because of the limitation of a conventional serial processor when it comes to real time image processing. To overcome this problem a field programmable gate array usually called FPGA can be used as an alternative to the serial processor. An FPGA is a matrix of logic blocks that are connected by a switching network. The logic blocks as well as the switching network, both are reprogrammable that allows an application specific hardware to be constructed while at the same time maintaining the ability of the system to the change the functionality with ease.

In Image processing tasks, parrallelism exists in two forms; spatial parrallelism and temporal parallelism. FPGA can be programmed to partition an image and distribute the resulting sections to multiple piplelines all of which could process data concurrently. FPGA is a compromise between the flexibility of general purpose processors and the hardware based speed of ASICs (application specific integrated circuit) [1].

In this project, Nexys 4 Artix-7 FPGA Board is used to implement an image processing task of applying different kernels to an image of size 128X128. The output of this task can be observed via 12-bit VGA port which is built-in the board. Methodology and Results of this project can be seen is the following sections.

2 Methodology

The conventional method to apply a kernel to the image needs the 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 the kernel can get the neighborhood values and gives the output. This method is an inefficient use of memory blocks and may limit the capability of the chip to process large-sized images in real-time.

In this report, an efficient way to apply mask over an image is proposed that only requires a portion of image pixels to be loaded in cache that depends on the image size and mask dimension. This number is given as follows:

$$pixels\_required = (image\_width \times (kernel\_height - 1)) + kernel\_width \tag{1}$$

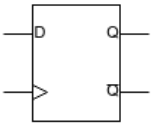
Thus for this project, only 259 pixels are required to be accessed/loaded in cache before the system starts doing the mathematics and sends the first output pixel to another memory block for a processed image.

The complete task of applying a mask over an image is divided into three parts each having several sub-parts as follows:

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

2.1 Cache Design

As discussed in the introduction that it is required to access 259 pixels of the subject image before the mask is applied. This task is subdivided into two operations, the first one is fetching the data from some memory block and storing it in some other memory units. Fetching the data will be discussed in the coming sections, while we talk about the storage of neighborhood pixels for mask multiplication here. It is a well-known concept in digital electronics that flip-flops are sequential building blocks of memory. More specifically a D-flip-flop, which is also known as a "data" flip-flop, captures the value of its D-input at a definite portion of a clock cycle (either as a rising or a falling edge) and shift that data to its output Q. At other times, Q retains its value until the same clock event happens and it shifts a new input to its output.



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

Figure 1: D-flip-flop symbol and truth-table

2.1.1 Coding the flip-flop:

A flip-flop thus as described by the truth-table shown in fig. 1 is designed using vhdl in ISE Project Navigator provided by Xilinx as follows:

```
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;
```

The above program 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 the buffer shifts its contents to output Q.

2.1.2 Structure of Cache:

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.

The above figure shows the concept of how the cache is supposed to work. 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.

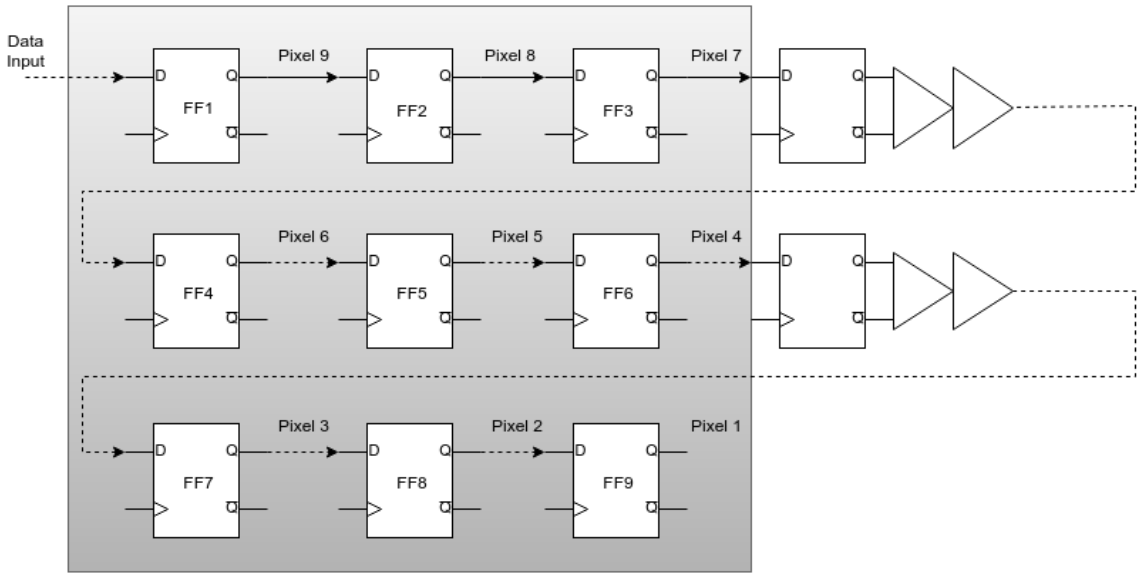


Figure 2: Illustration of the idea of application of the mask

Although the design shown 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. Thus an efficient way is to use a FIFO to store all the pixels for each row of the image except for those which corresponds to the kernel_width. Thus FF1, FF2 and FF3 should stay and rest of the flip-flops of the first row should be replaced by a FIFO of depth 125 bytes. This should follow for the second row also.

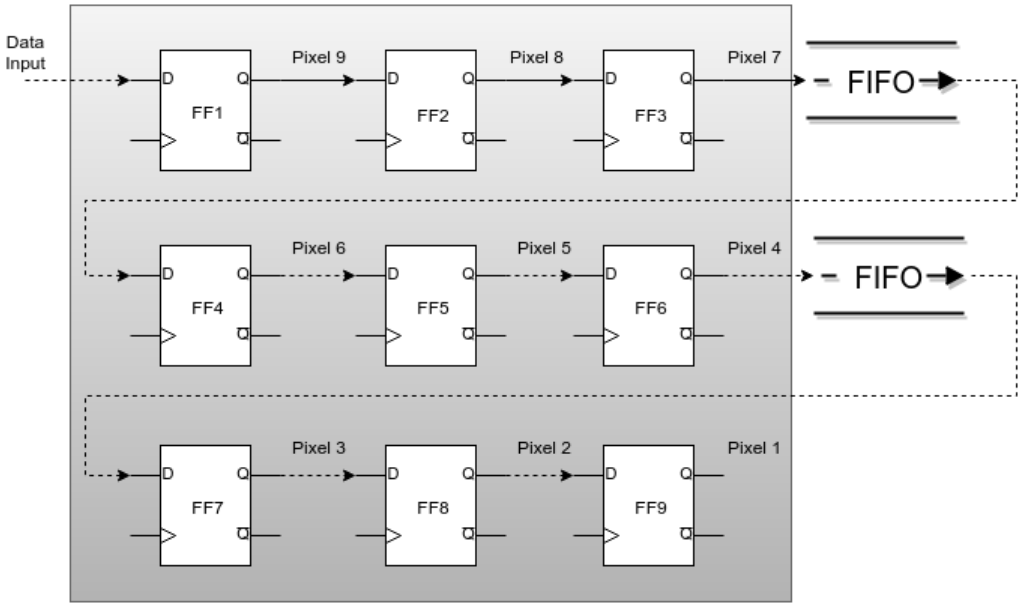


Figure 3: Illustration of Improved Cache

2.1.3 Coding the cache

The cache as described in the previous section is designed using flip-flops and FIFOs in vhdl as follows:

```
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);
```

The above program use a generic flip-flop which is already defined in section 2.1, 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).

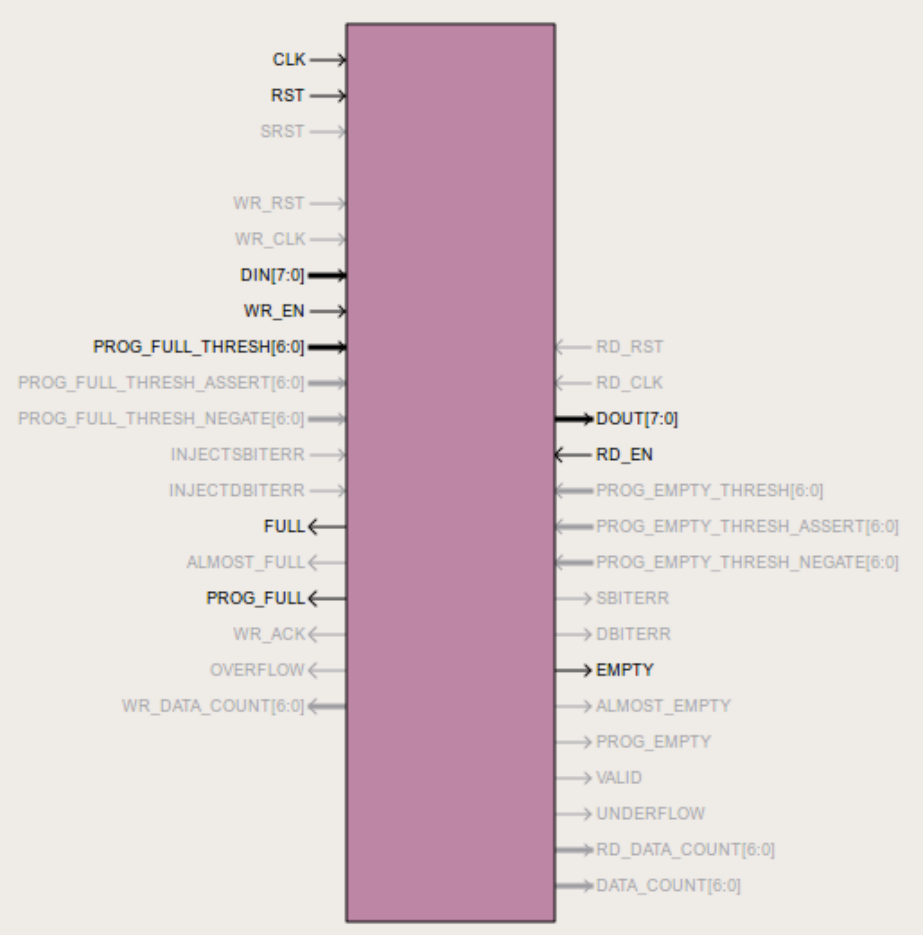


Figure 4: Schematic of a FIFO

The 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:

$$prog\_full\_thresh = image\_width - kernel\_width - 2 \tag{2}$$

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.

2.2 Kernel Convolution:

The kernel design involves multiplication of mask values to the corresponding pixel values that are transferred by the *main* entity, then addition following by division by 8 (corresponds to 8 neighborhood pixels).

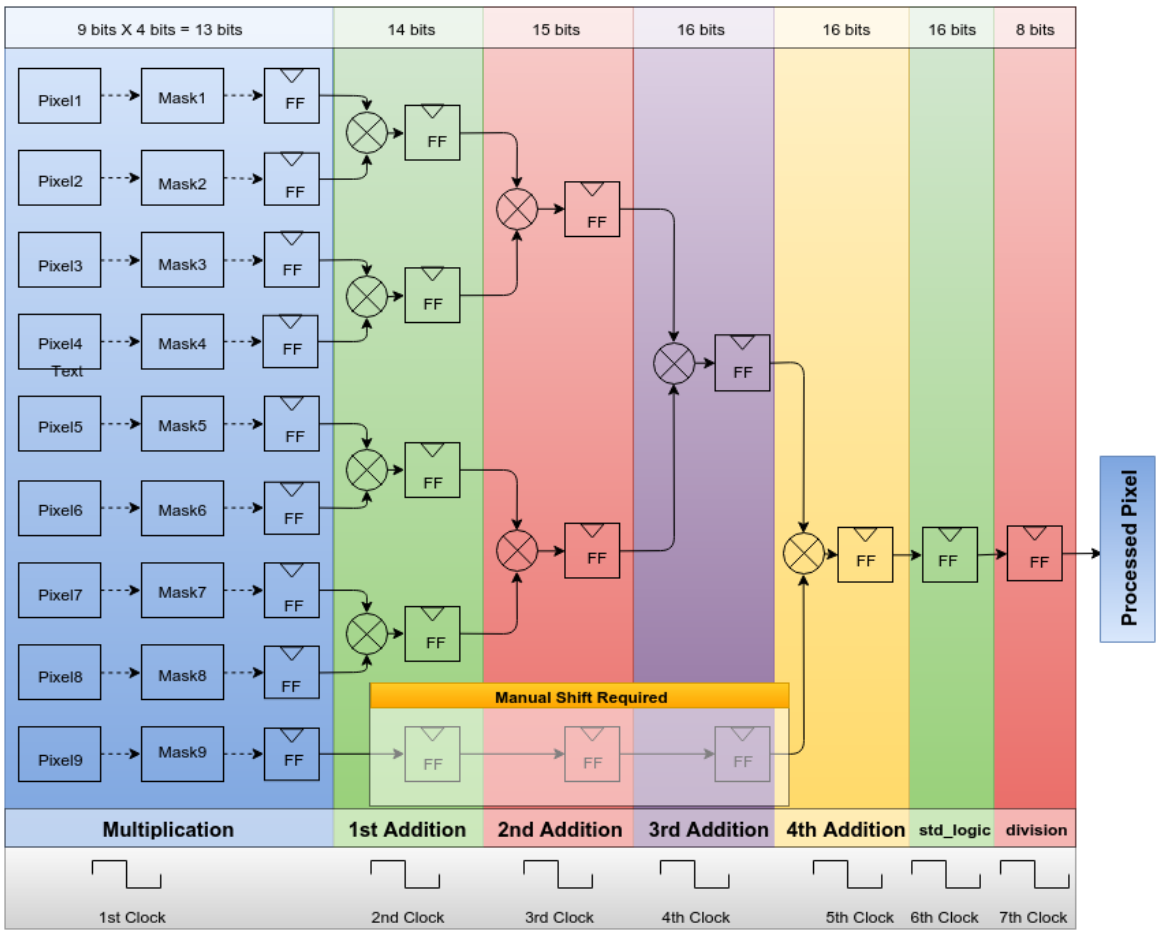


Figure 5: Kernel convolution pipeline overview

This is done using a series of commands all defined under a process that executes them at every rising edge of the clock. This makes a configuration as illustrated in Fig. 5. The timing diagram of the output confirms that the kernel required at least 7 clock cycles to push a valid data for the processed pixels. The simulation and implementation of this code on FPGA board is discussed in the following sections.

3 Results:

This section talks about simulation that involves time analysis of the data transfer and data manipulation, preliminary MATLAB results and VGA output of FPGA board.

3.1 Simulation and Time Analysis

Simulating a code is usually known as *test benching* in FPGA programming. Test bench emulate a FPGA, providing clock input to the *main* entity and other components that subscribe it in their code.

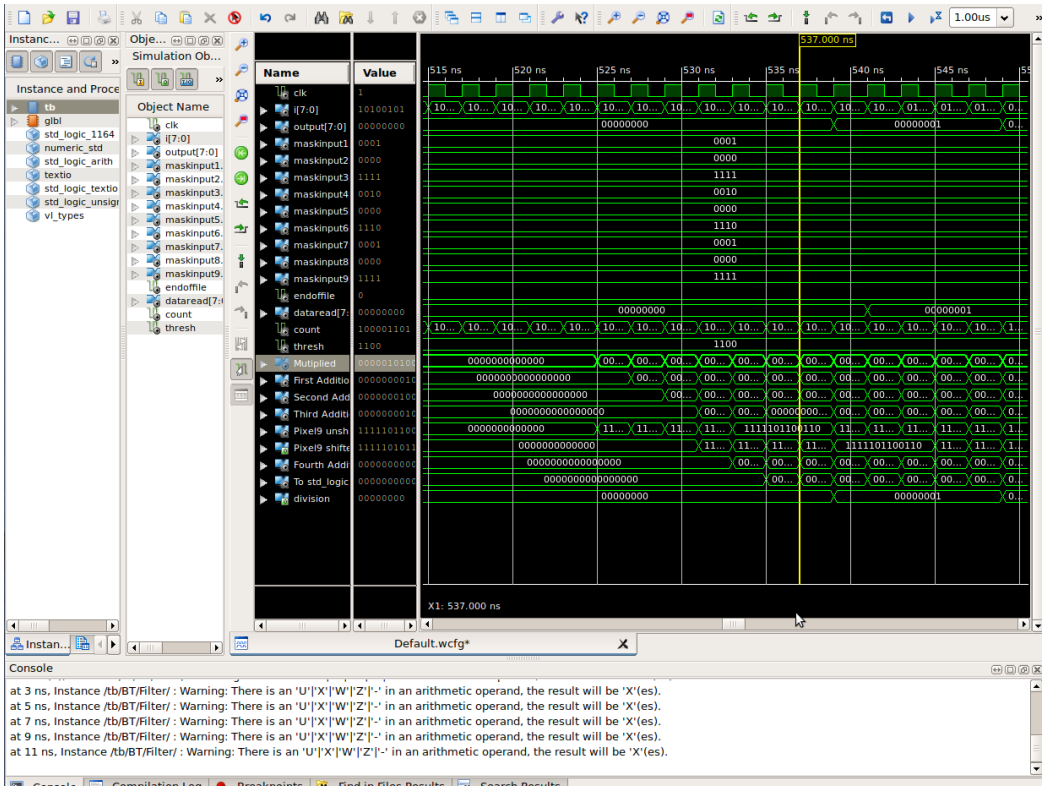


Figure 6: Example of test bench output

Inputs can be given as binary or HEX values to the input pins or ports whereas output can be seen on a screen in the timing diagram that is similar to a logic analyzer output. An example of the timing diagram can be seen in Fig. 6 which illustrates the actual output of the image processing code developed for this project. A detailed analysis of the simulation output is shown in Fig. 7. Different colored boxes represent different events. The first result of multiplication of pixels appears at 525ns. The period of 1 clock cycle is 2ns. The calculated number of cycles required until the first multiplication result should appear is given in Table 1.

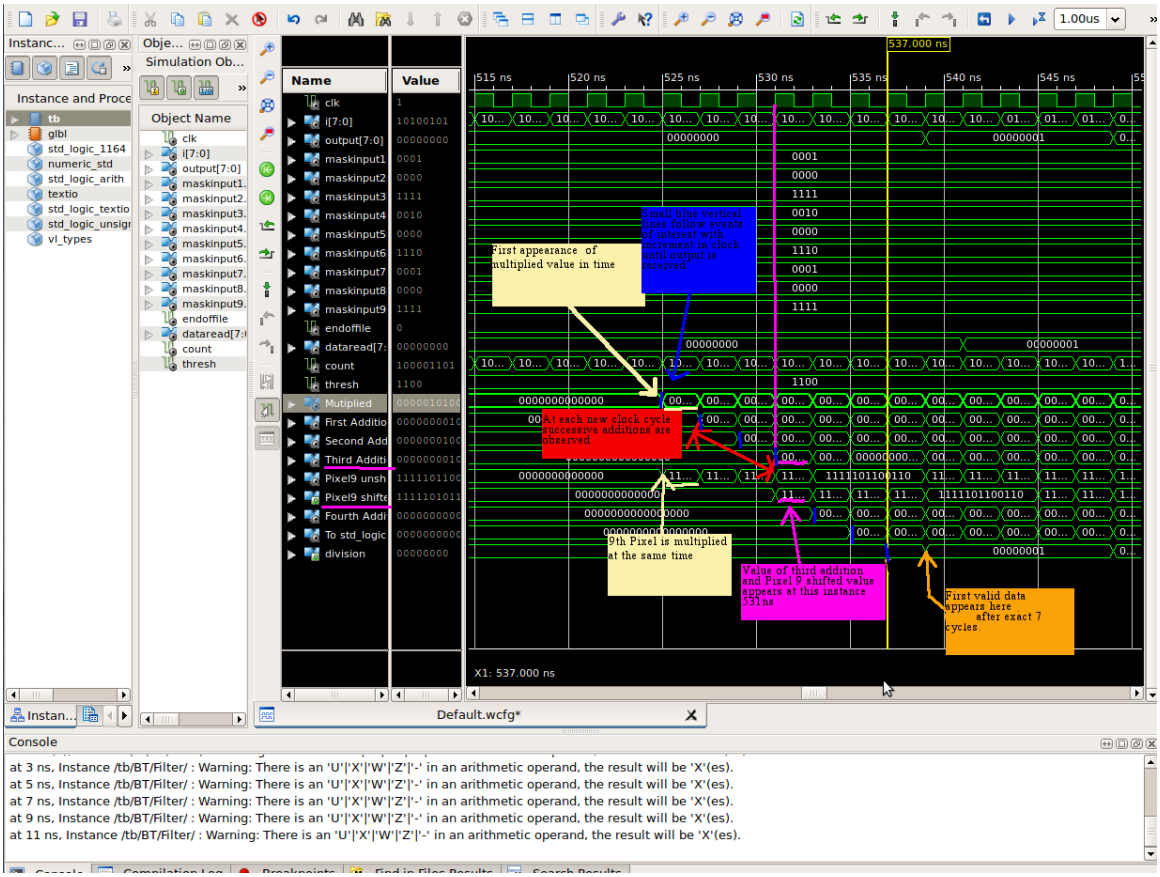


Figure 7: Analysis of test bench output

Fig. 7 shows that the first multiplied value appears at 525ns which is because the Input started at time 1ns. The comparison between the calculated and observed results in the test bench proves the harmony of system and thus the processed bytes are saved in another .txt file before the results are converted into image for observation. MATLAB code is used to convert the data of a *.txt* file into an image. Following section will show the results of some masks applied to a given image. Adding a time constraint *.ucf* file and checking for the best timing for the filter convolution performance, it is seen that the it takes only 4.38ns to process the mask and send the processed pixel out.

3.2 Results using MATLAB:

Four different masks are applied to the given image which is 128X128 pixels as already stated. The output from the MATLAB code is shown in Figure 8. MATLAB code reads the binary data

Table 1: Time calculation for result of first multiplied value

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	

from a text file, converts it into decimal equivalent and reshape the matrix into a 128X128 matrix.

Figure 8: MATLAB Results of the code.

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 vhd 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 .coe file 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.

Figure 9: VGA Output of 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.

References

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