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

Gopikrishna Erabati

Poorna Naga Avinash Narayana

Under the supervision of

Dr. Julien Dubois

CONTENTS

1. Introduction

As we are living in the 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, there is need to process these acquired images in almost real time for certain applications. For such real time applications we turn to Digital Signal Processors (DSPs) or Field Programmable Gate Arrays (FPGAs).

DSP is a special class of microprocessor designed to specifically address real time implementation issues. A DSP can process data in real time, making it ideal for applications that can't tolerate delays. Digital signal processors take a digital signal and process it to improve the signal into clearer sound, faster data or sharper images. As the complexity of real-time systems increases the need to introduce more efficient hardware platforms grows.

FPGA has gained a lot of traction in the real-time community, as a replacement for the traditional DSP solutions. FPGAs are re-programmable silicon chips. Using inbuilt logic blocks and programmable routing resources, one can configure these chips to implement custom hardware functionality. One can develop digital computing tasks in software using HDL programming and compile them down to a configuration file or bit-stream that contains information on how the components should be wired together. In addition, FPGAs are completely reconfigurable and instantly take on a brand new "personality" when we recompile a different configuration of circuitry. FPGAs provide hardware-timed speed and reliability. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing. FPGAs are indeed revolutionizing image and signal processing due to their advanced capabilities such as reconfigurability and parallel processing.

So, we acquire images in real time and use the parallel processing capability of FPGA to solve our real time issue of sophistication.

1.1 Project Task

From the above introduction, it is clear that we are trying to process something in all most real time using parallel processing.

Our task is to implement 2D filter to process the images using FPGA and VHDL (Very High Speed Integrated Circuit Hardware Description Language). The image if of 128*128 pixels in resolution and the kernel size is 3*3 pixels.

The software used for the simulation and implementation is Xilinx ISE (Integrated Synthesis Environment) Design Suite and the language is VHDL.

1.2 About Xilinx ISE and VHDL

Xilinx ISE is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Xilinx ISE is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors. The Xilinx ISE is primarily used for circuit synthesis and design, while ISIM or the ModelSim logic simulator is used for system-level testing[2].

The development of VHDL was initiated in 1981 by the United States Department of Defence to address the hardware life cycle crisis. The cost of reprocuring electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would *work the same* on any simulator and was independent of technology or design methodology [3].

VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as FPGAs and integrated circuits. VHDL can also be used as a general purpose parallel programming language. VHDL is commonly used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, 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

The 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 to process pixels according to the filter.

2.1 Cache Memory

The basic purpose of cache memory is to store data that are frequently re-referenced by software during operation. Fast access to these instructions increases the overall speed of the

software program. So, the basic purpose of implementing the cache memory in the overall pipeline is for simultaneous pixel accesses, which 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.1.

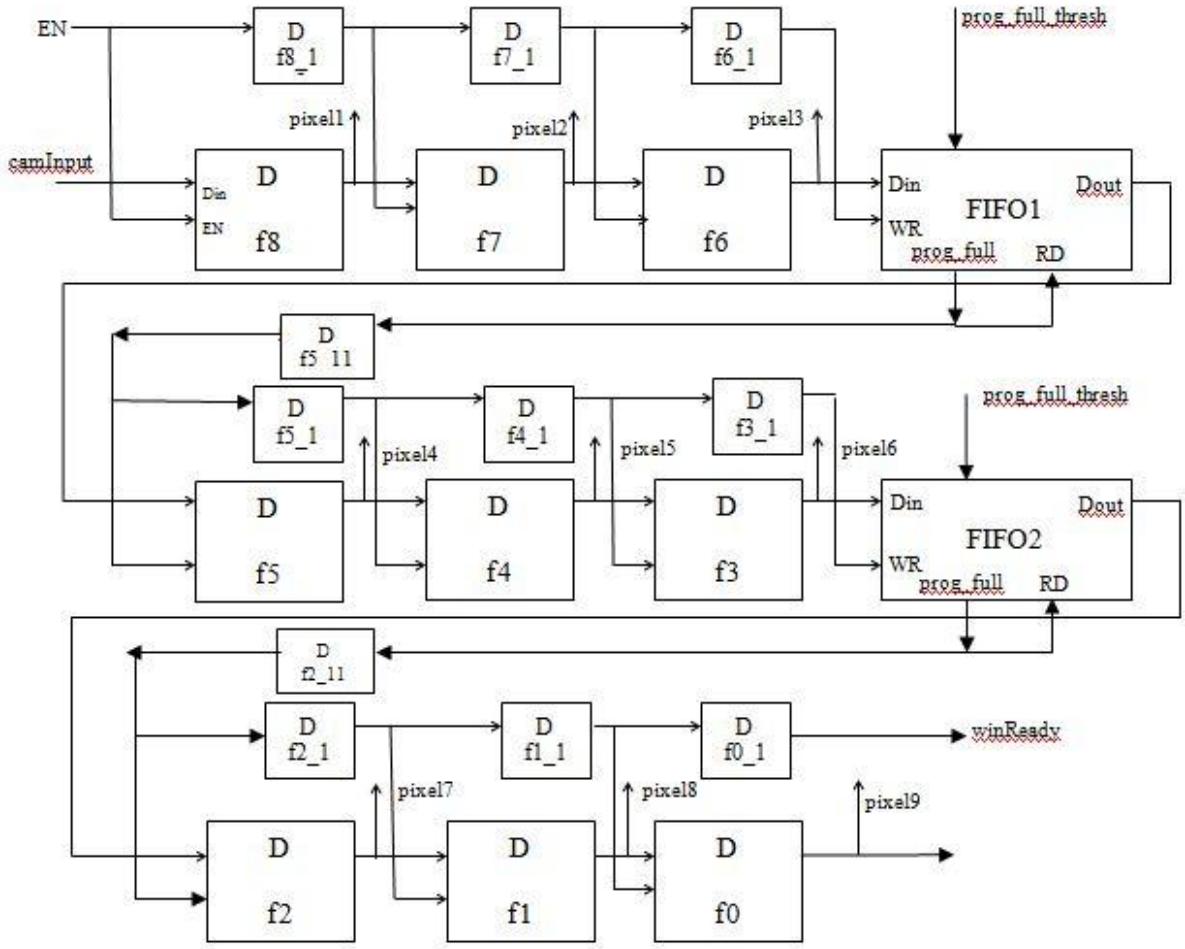

Fig. 1 Cache Memory

FF is a fundamental building block of digital electronics systems used in computers, communications, and many other types of systems. It is the basic storage element in sequential logic. The D flip-flop is widely used. It is also known as a "data" or "delay" flip-flop. The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change. The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line. These flip-flops are very useful, as they form the basis for shift registers, which are an essential part of many electronic devices.

FIFO is a method that relates to the organization and manipulation of data according to time and prioritization. In essence, the queue processing technique is done as per a first-come, first-served

behavior. The first data which is added to the queue will be the first data to be removed. FIFO is used for synchronization purposes in computer and CPU hardware. FIFO is generally implemented as a circular queue, and thus has a read pointer and a write pointer. Here, in our case FIFO is used to store the pixels which are not accessed in a clock cycle for processing and which are accessed in next clock cycles. There is data bus for FIFO which can be of 1024 bits and control signals to Write (WR) and Read (RD) the data. There are other flags such as Full, Empty flags to know the status of FIFO whether the FIFO is full or empty. One more important feature of FIFO is "programmable full threshold" input where we can program the threshold required by us, to know that the FIFO is full for the threshold we input, by the "prog full" flag.

As we want to access 3*3 pixel neighborhood, for the first clock cycle we should be able to access first three columns of first three rows of an image. For this, we used nine FFs (for simultaneous accessing of pixels) and two FIFOs to store the remaining pixels of rows for shifting in the next clock cycle as seen in fig.1.

The storage of pixels of image in the cache memory for first accessible clock cycle of processing unit is shown in fig.2.

f0	f1	f2	fifo2						
f3	f4	f5	fifo1						
f6	f7	f8							

Fig.2 Storage of pixels in image in Cache memory of fig.1 for first accessible clock cycle of processing unit

In order for the user to know when the pixels are ready to access by the processing unit, we implemented a logic using D-FFs (f8_1 to f0_1).

The above configuration as seen in fig. 2 is achieved by the structure in fig.1 as follows:

For the layer 1, for the first clock cycle the data is stored in f8 FF and in next clock cycle the data in f8 FF is shifted to f7 FF and this continues until f6 FF. Meanwhile, when we start the input we also set the Enable (EN) input of the f8_1 FF to '1' and synchronously the Enable data is shifted (by the FFs f8_1 - f6_1) as the normal input is being shifted (by the FFs f8-f6). The output of f6_1 FF is connected to the Write (WR) control pin of FIFO, thereby in next clock cycle the data in f6 FF is written onto the FIFO and thereby the Empty flag of FIFO goes down. Here, we connect the 'prog full' flag to the Read (RD) control signal of FIFO. We used **'123' as**

our program full threshold input for the FIFO because as the FIFO reached the threshold the 'prog full' flag is set to '1' and in the next clock cycle the first input data is ready to be written on Data Out (dout) pin of FIFO. In the next clock cycle the data is written on dout pin of FIFO and now the data is ready to be shifted using f5 FF. So, here two clock cycles are needed for the data to be ready for f5 FF after the 'prog full' is set to '1' by the FIFO. For this reason the threshold of 123 is chosen (explanation with timing diagrams provided in Section 3.3). As there is delay in FIFO by one clock cycle as data is written onto 'dout' pin by. delay of one clock cycle, we used a f5_11 FF to be in synchronous with the data being written onto FFs. The 'prog full' flag is connected as input to f5_11 FF and the output of this FF is connected to f5_1 FF as a second level structure starts. After 128 clock cycles (as input is 128*128) the layer 1 sets the enable signal to '1', as input of f5_1 FF for layer 2 to start.

For the layer 2, the same logic follows as explained above for level 1. After 256 clock cycles, the layer 2 sets the enable signal to '1', as input of f2_1 FF for layer 3 to start.

For the layer 3, the same logic follows as above. After 259 clock cycles the first pixel which we input at layer 1, is ready at output of f0 FF, and also the Enable signal which we input at layer 1 to f8_1 FF is also ready at output of f0_1 FF as a 'winReady' signal, as a flag for the processing unit to start its processing.

As the clock triggers its cycles, the data is shifted one by one using FFs and FIFOs until the last window of the image for accessing the pixels is encountered. At the end of last pixel of the data, we set the 'readfinishedflag' to '1' in the simulation for the program to know that reading of data is finished.

2.2 Processing Unit

By the virtue of Cache memory we designed in section 2.1, we now have the proper access to pixels for the processing. 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.

The processing unit strategy we implemented is as shown in fig. 3. 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. 3 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. the divider is used to divide the added output according to the kernel used.

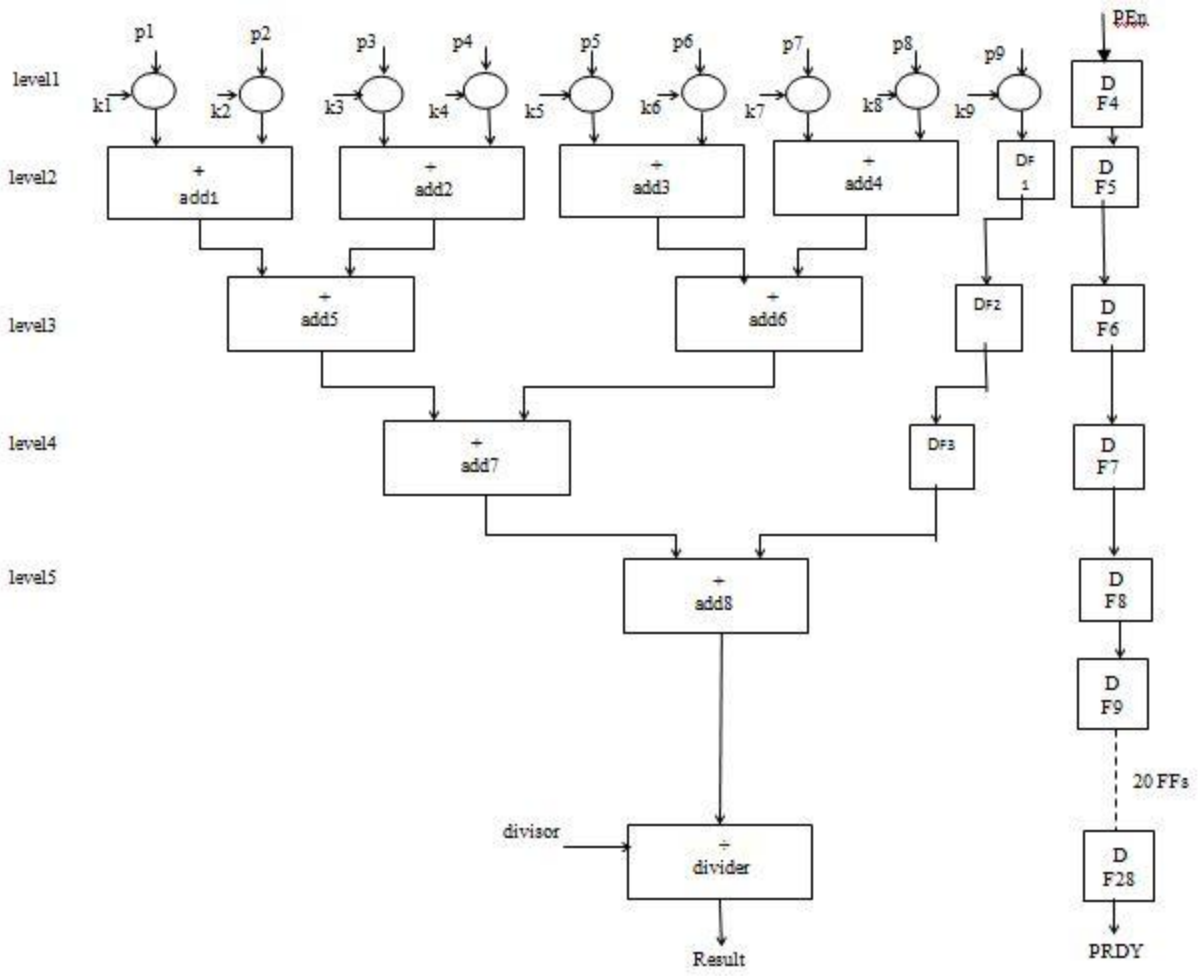

Fig. 3 Processing Unit

For the multipliers used in level 1, we used one operand as unsigned 8 bit (these are pixel values) and other operand as signed 4-bit (these are 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 adders 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 its 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.

In order for the user to know when the output of processor is ready, we implemented a **delay strategy of the signal** ('winReady' signal of Cache Memory) using D-FFs (f4 - f28), same as the time taken by the processing unit to get the result. 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. So, we implemented this delay using 25 D-FFs (f4 - f28), so that the 'winReady' signal at cache memory, which is set to '1' after first data is available for processing, is delayed by 25 clock cycles and its set to '1' as 'PRDY' flag to the user to know that output at processing unit is available for further use.

2.3 Read/Write Process Module and overall 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. 4.

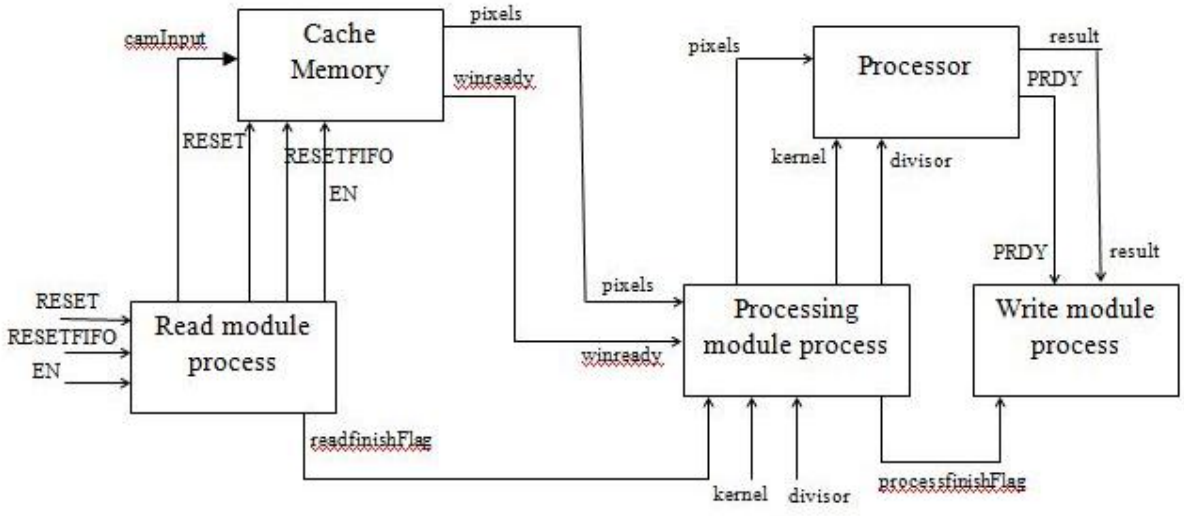

Fig. 4 Complete structure with simulation processes

In 'read' process module in simulation, firstly, we set the 'RESET' and 'RESETFIFO" flag to '1' and the 'EN' control signal to '0'. After 10 clock cycles we pull down the 'RESETFIFO' flag and after 15 clock cycles we pull down the 'RESET' flag and after 20 clock cycles we pull up the 'EN' flag and start the reading process. Here two resets are used instead of one because after we reset the FIFO, we observe that FIFO is taking 4-5 clock cycles for the 'Empty' flag to set and 'Full' flag to reset. If we use one reset, so after the reset is pulled down, the system starts its process but the FIFO is not yet ready for the operation.

Secondly, we open the 'dat' file to read the pixel data one per clock cycle, and the read data is connected to 'camInput' of cache memory. By the virtue of D-FFs (f8_1 to f0_1) as shown in fig. 1, the cache memory pulls up the signal as 'winReady' flag, which states that the first window of 9 pixels is ready for processing. At the end of data in file, the 'readfinishedflag' is set to '1', so that the user knows that the reading of data is finished.

As the 'winReady' flag is set to '1', as explained above, the processing units starts its processing by taking the 9 pixels as input and by multiplying, adding and dividing as explained in section 2.2. After the processing as the result is ready at the output of processing unit the 'PRDY' flag is pulled up. After the 'readfinishedflag' is set to '1', after 25 clock cycles 'processfinishedflag' is set to '1' to tell the user that processing of data is finished.

As the 'PRDY' flag is pulled up, the 'write' process module in simulation, counts the cycles for which it is writing the information. As the **cache memory doesn't know which pixels are at border** and which pixels processed value we should not write to the file, we used counter to count and give instruction not to write those values after counter reaches specified value and we reset the counter for the next row. For the values which needed to be written we set the 'writeflag' to '1' and others to '0'. By this strategy we were able to sole the border issues with the image and the rows and columns which we ignore are as shown (in read) in fig. 5, and we finally get 126*125 image as output.

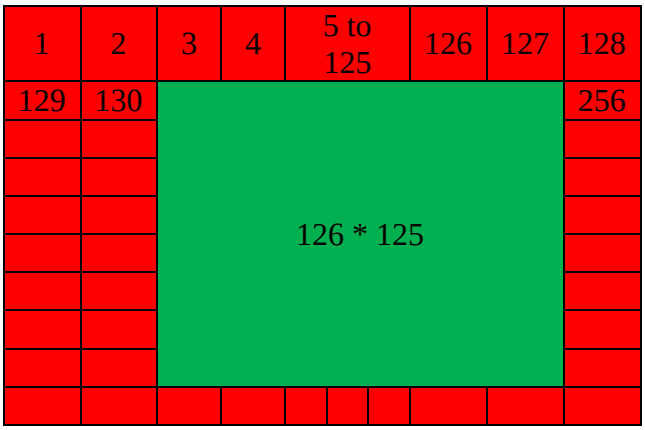

Fig. 5 The unconsidered rows and columns in our implementation(in red) which results in 126*125 image

As the 'processfinishedflag' is set to '1' by the process module in simulation, the writing of data to file by the writing module stops and thus it finishes the complete processing of pixel values and writing it to a file.

3. Simulation Results and Discussion

in our implementation we designed our tasks and managed to link them using a **modular approach**. So, in this approach we designed each and every module starting from FFs used in cache memory to entire structure ad tested each structure at its modular level and linked to other modules and tested at higher level of our modular approach.

3.1 D - Flip Flop

The result after testing the D-FF which delays the input fed to FF is as shown in fig. 6.

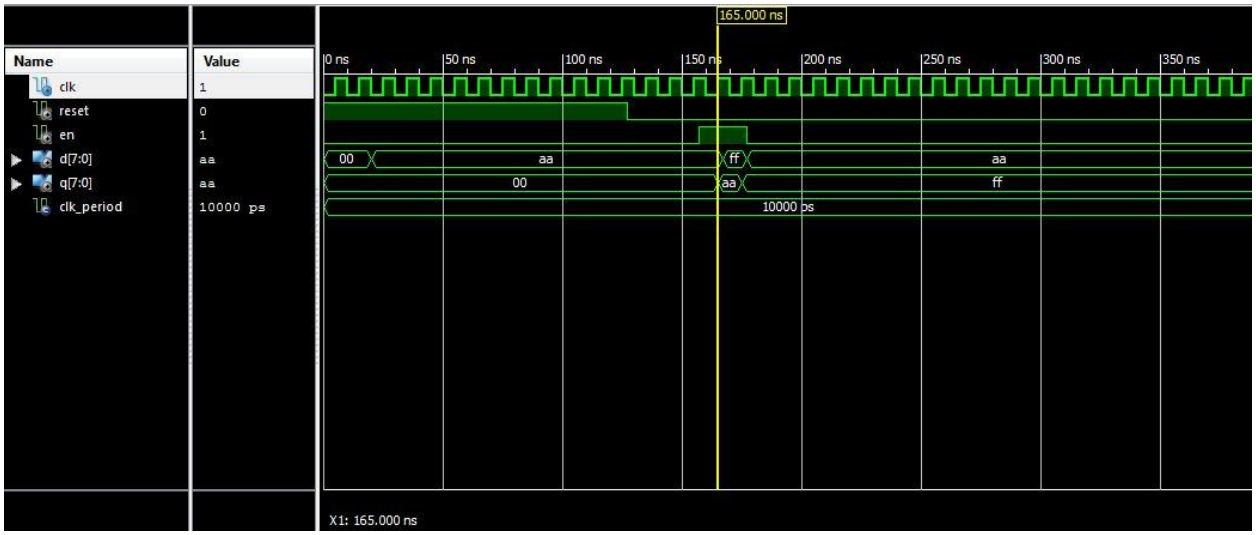

Fig. 6 FF test result

As we can see in the fig. 6, after the reset flag is set to '0' and 'en' control signal is set to '1', in the next rising edge of clock cycle the input which is on 'd' (0xAA) is written on output 'q' (0xAA). And on the next clock cycle as 'en' is high, the input (0xFF) is written on output (0xFF). After which the 'en' control is low, so the input (0xAA) is not written onto output asn output stays as previous output (0xFF).

3.2 FIFO

The result after testing the FIFO as explained in section 2.1, which outputs the first value it is inputed with, is as shown in fig. 7.

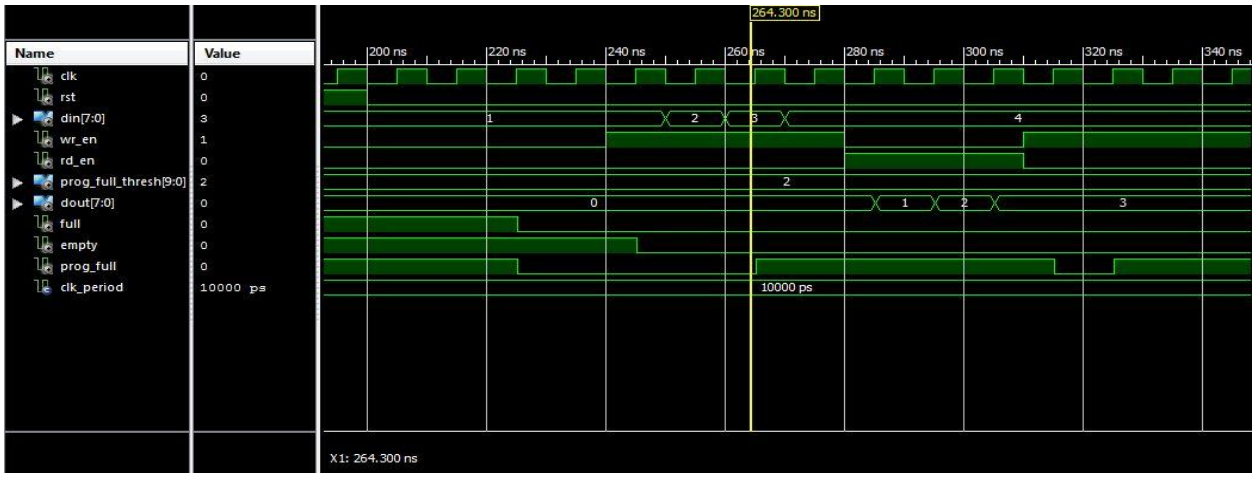

Fig.7 FIFO testing results

After the 'rst' control is set to '0' of FIFO, the 'full' and 'empty' flags of FIFO are both '1'. So, here the FIFO is not yet ready for the processing, so we wait for 4 clock cycles and then set the 'wr_en' control to '1', which instructs FIFO to write the data available at 'din'. We also set the 'prog_full_thesh' control signal to '2' which is programmable threshold to hold the values in FIFO (Here, we choose '2' , to **understand the working of FIFO**). After the 'wr_en' control signal is set to '1', the FIFO writes the data into it and we can see that the 'empty' flag is pulled down simultaneously from which we can say that the data is written onto FIFO. After 2 clock cycles, two data values are written onto FIFO as 'wr_en' is still on '1'. When FIFO tries to write at thirs clock cycle after 'wr_en' is set to '1', the 'prog_full' flag is pulled up, which shows that FIFO is full.

After few clock cycles, as 'rd_en' control signal is set to '1', in the next rising edge of clock cycle the data which is first written (here '1') is read out of 'dout' and sequentially other data until 'rd_en' is set to '1'.

3.3 Cache Memory

The result after testing the cache memory with the synthetic data (1 to 270 values with a counter, as input) is as shown in fig. 8 - 12. The 'prof_full_thresh' is set to 123.

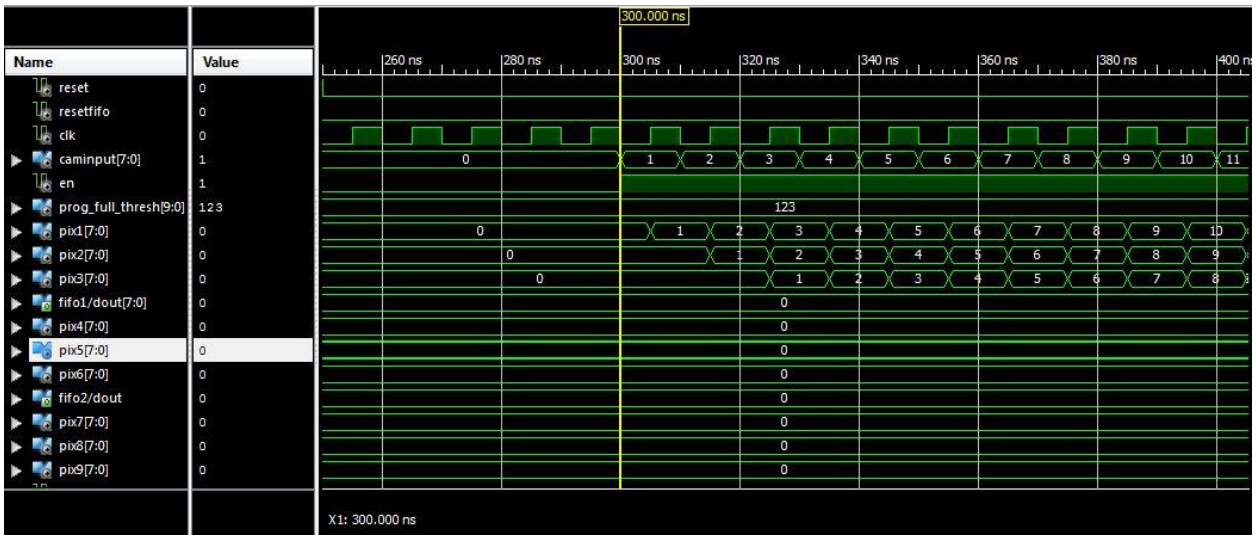

Fig. 8 Cache Memory Result

As the 'en' control signal is set to '1', which tells that input is being fed to 'camInput' signal, the data is shifted by D-FFs which acts as shift registers and the 'en' signal is also being shifted by the extra FFs.

Fig. 9 Cache Memory Result

As the 'prog_full_thresh' signal of fifo1 is set to value of '123', as 126 values are fed in to the layer 1 of cache memory the FIFO1 is full and 'prog_full' flag is pulled up as shown in fig. 9. the 'rd_en' control signal of fifo1 is also set to '1' which starts the reading of data from fifo1. But the first data we input at 'camInput' is available at 'dout' of fifo1 after one clock cycle of 'rd_en' is set to '1'. So, we used additional FFs to consider this delay in our system.

Fig. 10 Cache Memory Result

After 4 clock cycles (as 4 D-FFs are involved), of 'prog_full' of fifo1 is set to '1', the 'wr_en' of fifo2 is set to '1', which starts writing of fifo2 in layer 2 of cache memory. As we can see after one cycle of 'wr_en" of fifo2, the 'empty' flag is pulled down as some data is written onto fifo2.

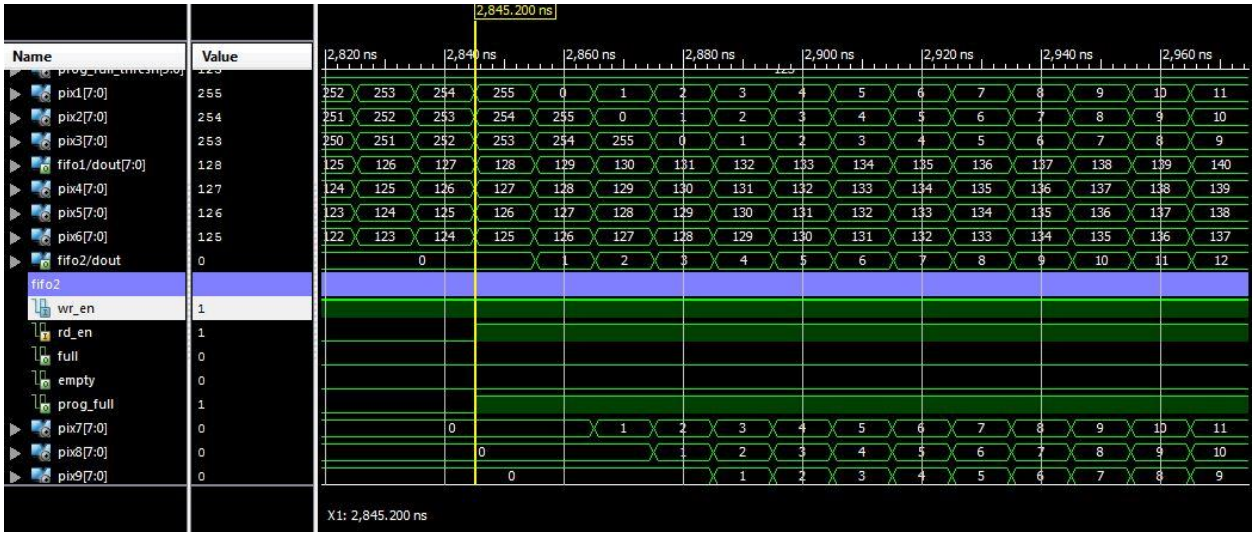

Fig. 11 Cache Memory Result

As the 'prog_full_thresh' signal of fifo2 is set to value of '123', as 126 values are fed in to the layer 2 of cache memory the fifo2 is full and 'prog_full' flag is pulled up as shown in fig. 11. The 'rd_en' control signal of fifo2 is also set to '1' which starts the reading of data from fifo2. Same as fifo1 the data available at 'dout' is delayed by one clock cycle this is compensated by an additional FF in the design.

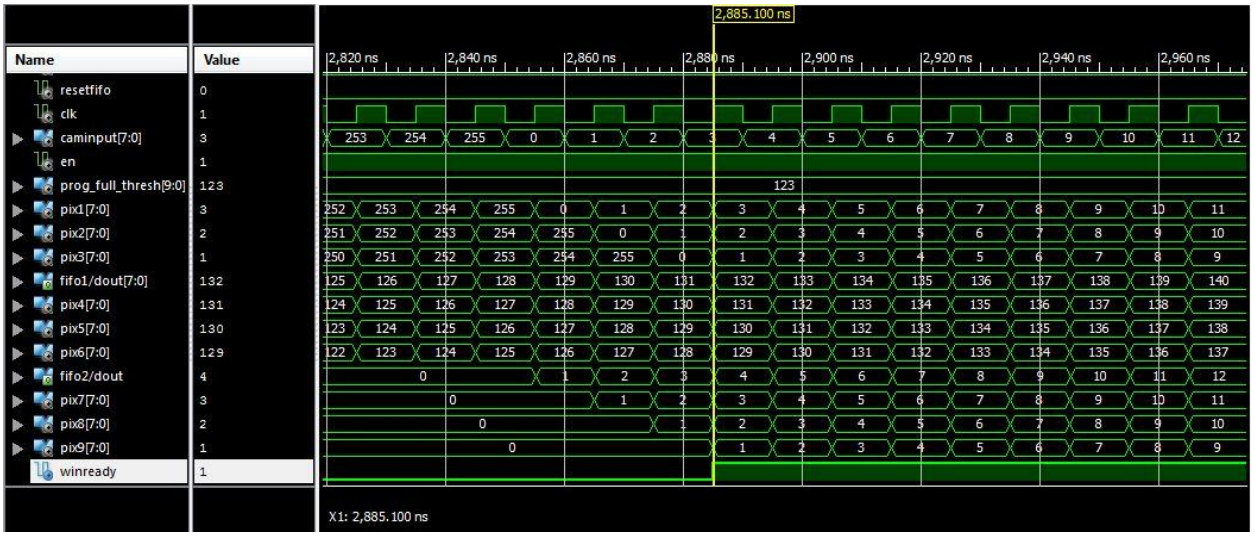

Fig. 12 Cache Memory Result

After 4 clock cycles the 'prog_full' flag of fifo2 is pulled up, the first nine pixel window is available at 9 FFs and the 'winReady' flag is pulled up to indicate this to processing unit.

The flow of pixels in cache memory is as shown in Tab.1 below.

TABLE 1 Our cache memory flow of synthetic data of pixels

Clk	P1	P2	P3	FIFO1		P4	P5	P6	FIFO2		P7	P8	P9
1	1				dout					dout			
2	2	1											
3	3	2	1										
4	4	3	2	1									
5	5	4	3	21									
6	6	5	4	321									
7	7	6	5	4321									
8	8	7	6	54321									
126	126	125	124	123 ... 1									
127	127	126	125	124 ... 1									
128	128	127	126	125 ... 2	1								
129	129	128	127	126 ... 3	2	1							
130	130	129	128	127 ... 4	3	2	1						
131	131	130	129	128 ... 5	4	3	2	1					
132	132	131	130	129 ... 6	5	4	3	2	1				
133	133	132	131	130 .. .7	6	5	4	3	21				
134	134	133	132	131 ... 8	7	6	5	4	321				
135	135	134	133	132 ... 9	8	7	6	5	4321				
136	136	135	134	133 ... 10	9	8	7	6	54321				
254	254	253	252	251 ... 128	127	126	125	124	123 ... 1				
255	255	254	253	252 ... 129	128	127	126	125	124 ... 1				
256	256	255	254	253 ... 130	129	128	127	126	125 ... 2	1			
257	257	256	255	254 ... 131	130	129	128	127	126 ... 3	2	1		
258	258	257	256	255 ... 132	131	130	129	128	127 ... 4	3	2	1	
259	**259**	**258**	**257**	**256 ... 133**	**132**	**131**	**130**	**129**	**128 ... 5**	**4**	**3**	**2**	**1**
260	260	259	258	257 ... 134	133	132	131	130	129 ... 6	5	4	3	2

3.4 Multiplier, Adder and Divider

The multiplier testing is as shown in fig. 13.

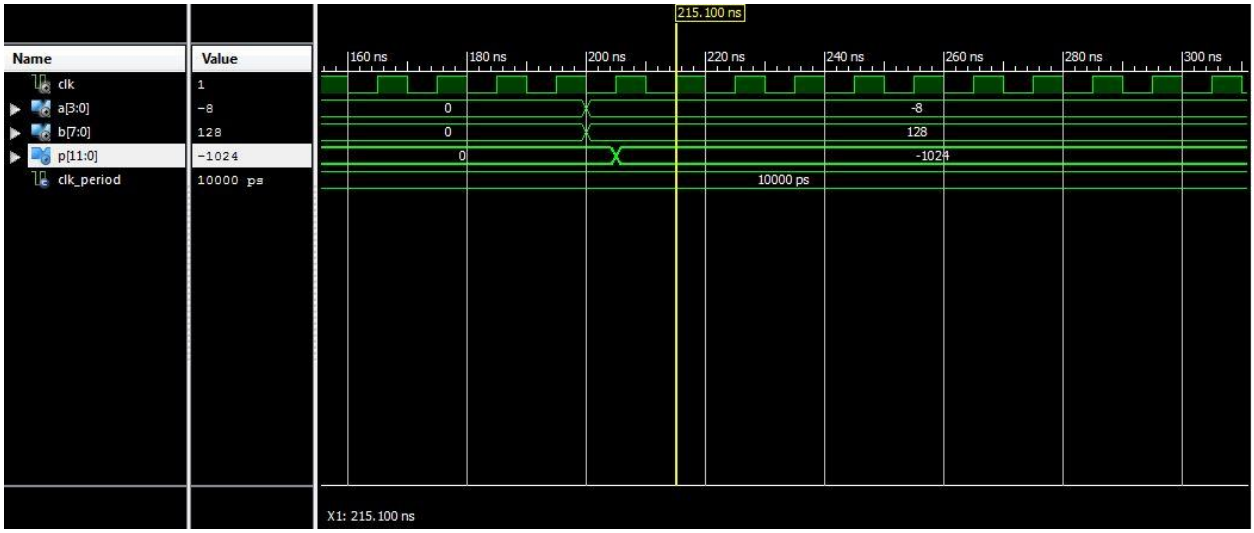

Fig. 13 Multiplier Testing

The result of signed multplication between a unsigned pixel value of '128' and signed kernal value of '-8' gives '-1024', a signed value of multplication. This is implemneted using IP Core generator of Xilinx.

The adder testing is as shown in fig. 14.

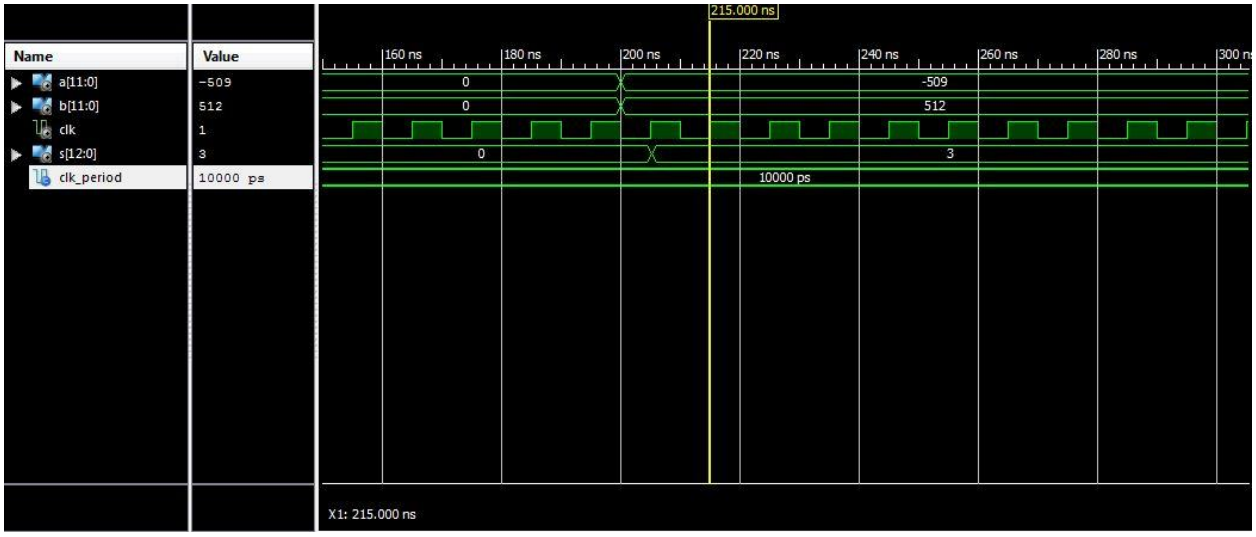

Fig.14 Adder testing

The result of addition of signed numbers '-509' and '512' which is '3' is shown as result.

The testing of divider is as shown in fig. 15.

Fig. 15 Divider testing

The **latency** of nearly 20 clock cycles is seen between input and output of the divider. This is compensated by D-FFs in our processing unit design.

3.5 Processing Unit

The testing of complete processing unit is as shown in fig. 16.

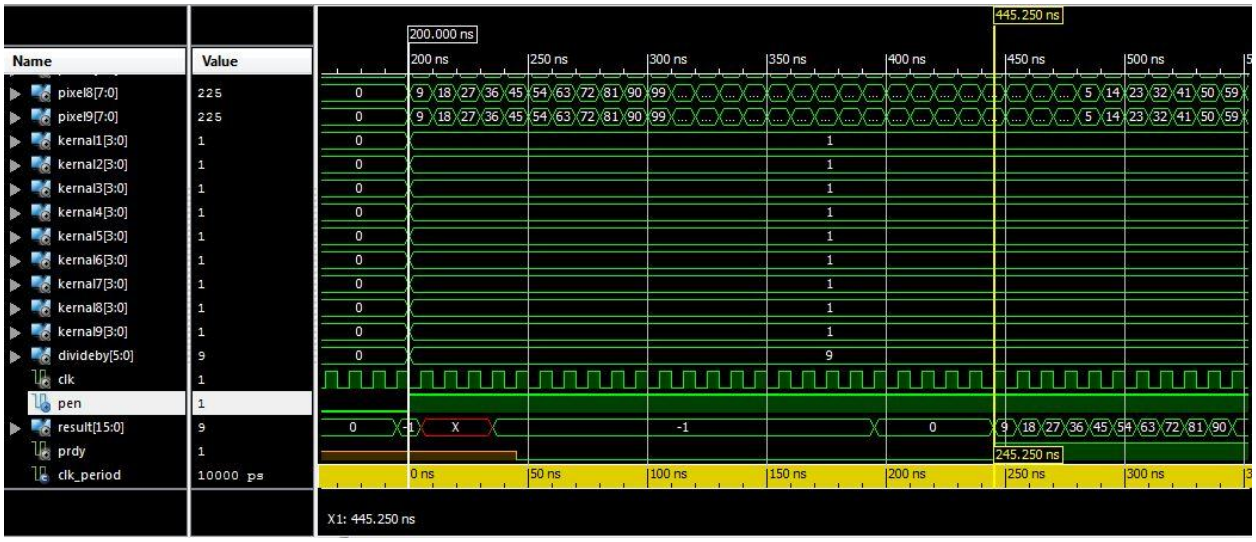

Fig. 16 Processing Unit testing

The **latency** of the complete processing unit with multipliers, adders and divider is nearly 25 clock cycles between input and output. Syntheitic data of all pixels with same values with incrementing by '9' after every clock cycle is fed to the input of processing unit, with kernel values all '1s' and divide by '9'. The result of first input which is '9' is seen after 25 clock cycles

which is the latency of the unit. As and when the result is ready the 'PRDY" flag is pulled up for the user to know that the result after processing is ready.

3.6 Complete Module (with read/write, cache memory and processor)

The testing of complete module is as shown in fig. 17 - 20. The full process for 128*128 pixel image takes nearly **16,410 clock cycles** to read the input, parallel process the input and to write the complete input to a file. **If clock period is 10ns , it would take 164.1 µs to get the result**.

Fig. 17 Complete module result

After the 'reset' and 'resetfifo' control signals are set to '0', the 'en' is set to '1' and the input from the file is fed to 'camInput'. And the cache memory continues its process to read the data and shift until the first window is available for processing.

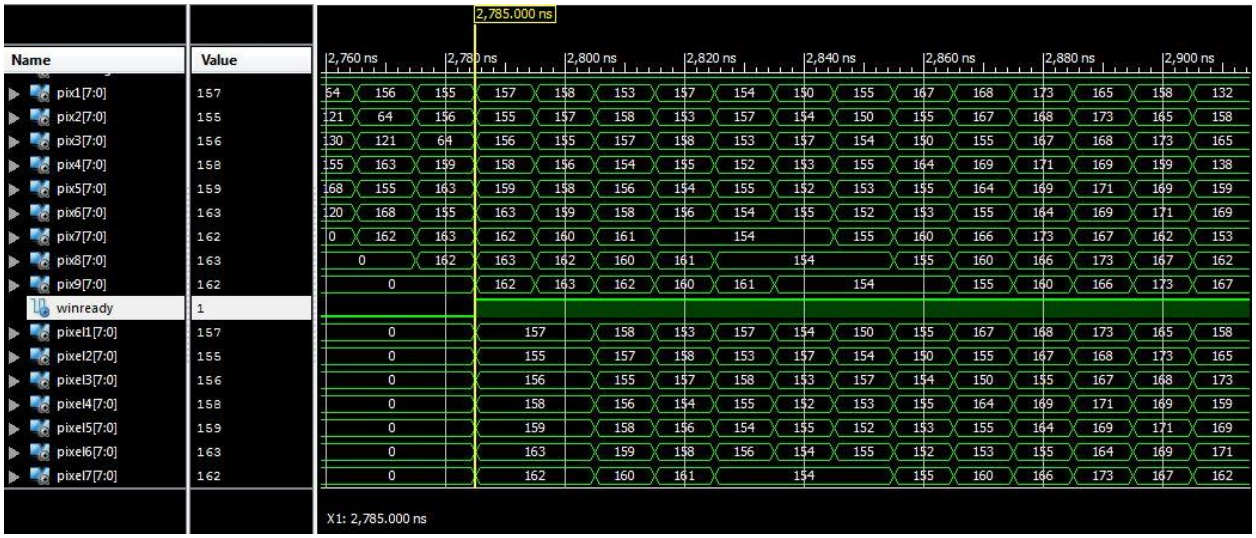

Fig. 18 Complete module result

As the first window of 9 pixels is available for processing the 'winready' flag is set to '1' as shown in fig. 18. So, now all the pixel value is fed to the processing unit. As 'winready' flas is set to '1', the 'pen' control signal of processing unit is also set to '1', which tells the procesing unit to start processing the input.

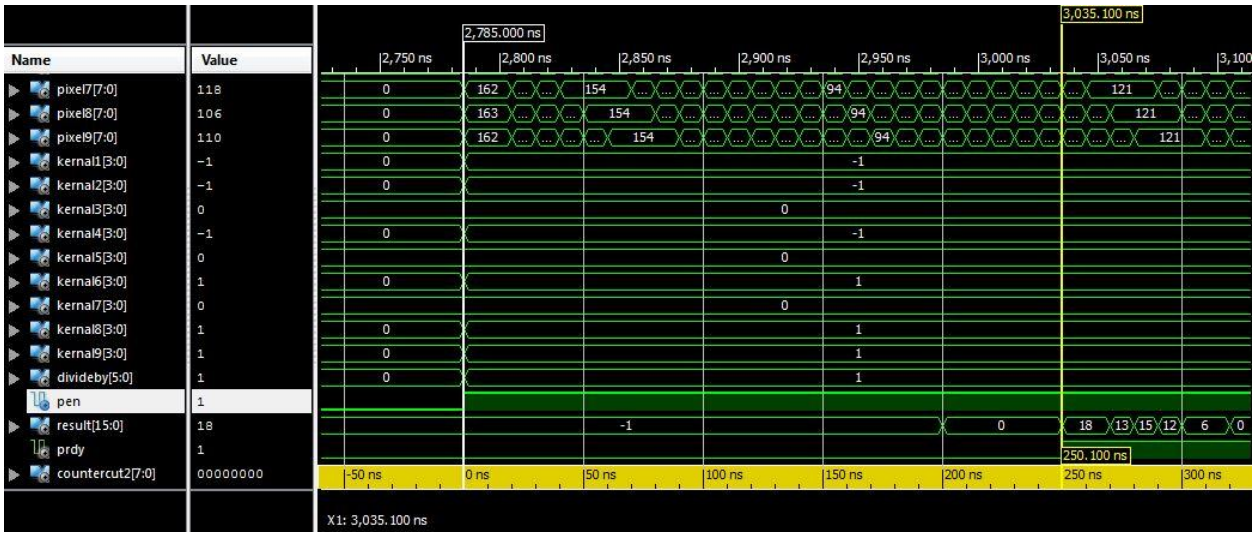

Fig. 19 Complete module result

The latency of processing unit is 25 clock cycles as described in section 3.5. We can see the latency between the 'pen' control signal and 'prdy' flag which is set to '1' after 25 clock cycles, which indicates that the result of first processing data is available at 'result' output. The 'write' module, starts the writing of result to a file, as 'prdy' flag is set to '1' as described in section 2.3.

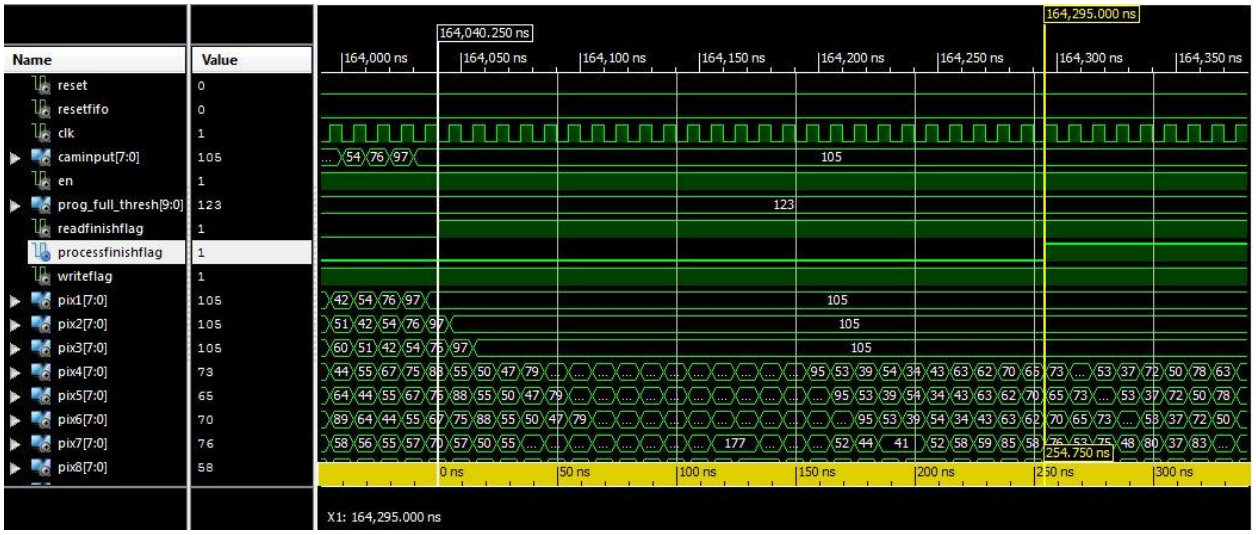

Fig. 20 Complete module result

After the complete data is read from the file, the 'readfinishflag' is set to '1' which indicates that complete data is read. As the latency of processing unit is 25 clock cycles, after 25 clock cycles

the 'processfinishflag' is set to '1' which indicates that processing of complete data is finished. And the complete data processed according to kernel chosen is written onto file.

3.7 Displaying Images in MATLAB

After the results are written onto the file, we have used MATLAB to check our results. We have written a code 'displayImage.m' which displays the image of the result. The code is found in the project code file and can be used in MATLAB.

3.8 2D Filter Results

3.8.1 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. 21 Result after applying average kernel shown above

The results after applying average filter kernel as shown below, is as shown in fig. 22.

$$\frac{1}{8}\begin{bmatrix}1 & 1 & 1\\ 1 & 0 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Fig. 22 Result after applying average kernel shown above

3.8.2 Sobel Filter

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

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

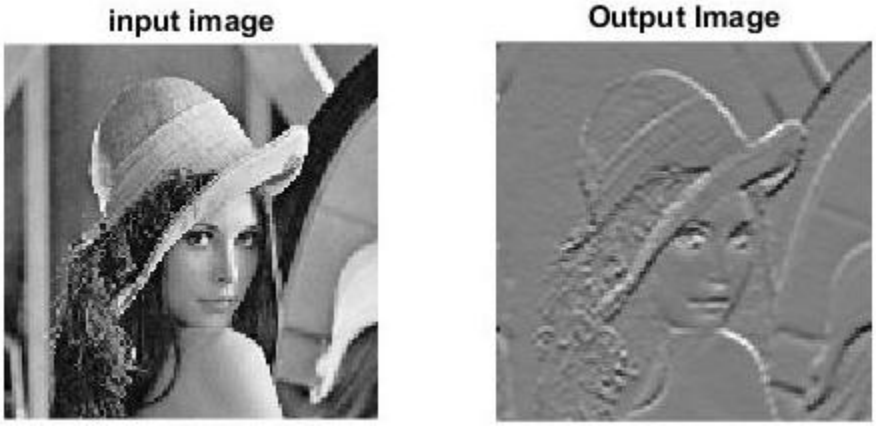

Fig. 23 Result after applying sobel kernel shown above

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

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

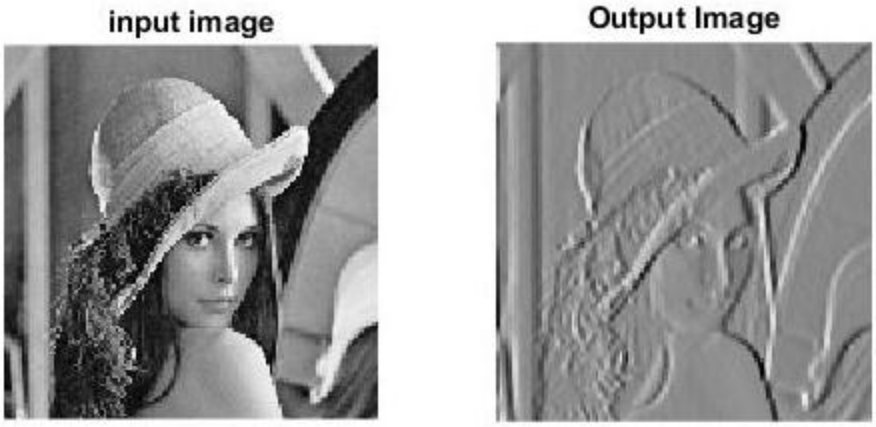

Fig. 24 Result after applying sobel kernel shown above

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

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

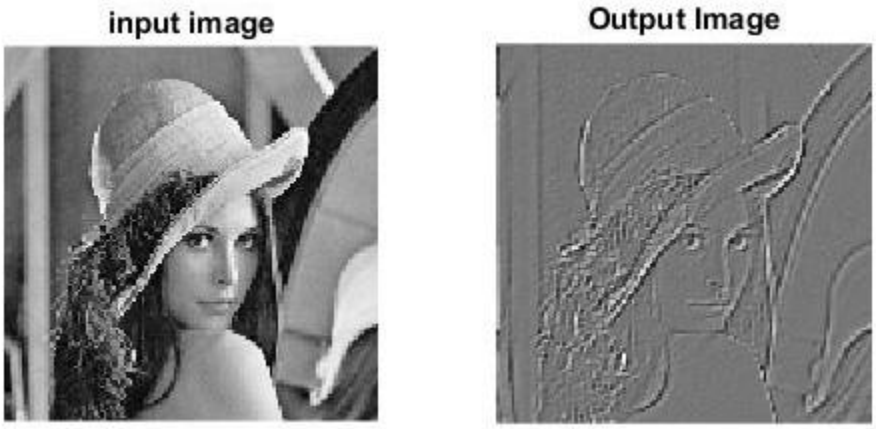

Fig. 25 Result after applying sobel kernel shown above

3.8.3 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. 26 Result after applying Gaussian kernel shown above

3.8.4 Motion Blurr

The result after applying motion blur kernel as shown below, is as shown in fig. 27.

$$\frac{1}{3}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Fig. 27 Result after applying motion blur kernel shown above

3.8.5 Emboss Filter (3D Shadow Effect)

The result after applying emboss kernel as shown below, is as shown in fig. 28.

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

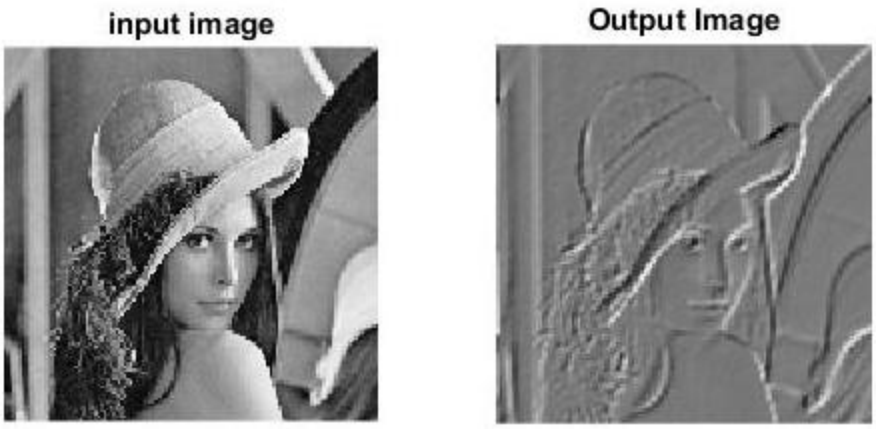

Fig. 28 Result after applying emboss kernel shown above

4. Characteristics of Complete Module

Characteristic	Min Value	Max Value
Input pixel bit length	8 bits	
Window size of kernel	3 x 3	
Prog_full_thresh of FIFO	1	1024
Kernel coefficients	-8	7
Divisor for divider	-32	31
Output pixel bit length	16	

5. 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.

REFERENCES

[1] 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.

[2] Xilinix ISE - Wikipedia

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

Appendix 1 - About Code

Implementation Files

File Name	Description	Components Used
FF_D	**Generic D-Flip Flop**	**-**
FIFO_new	FIFO from IPCORE	-
Cache_sample_new	**Complete Cache Memory**	**FF_D, FIFO_new**
My_mult	Multiplication between pixel and kernel	-
My_adder, my_adder2, my_adder3, my_adder4	Adders for pixels	-
My_divider	Divider for diving the sum	-
Processor sample	**Processing Unit**	**FF_D, my_mult, my_adder, my_adder2, my_adder3, my_adder4, my_divider**

Simulation Files

File Name	Description	uut
FF_D_gen_test	Test bench for Flipflop	FF_D
FIFO_tb	Test bench for FIFO	FIFO_new
Cache_sample_new_tb	**Test bench for Cache memory**	**Cache_sample_new**
My_mult_tb	Test bench for multiplier	My_mult
My_adder_tb	Test bench for adder	My_adder
My_divider	Test bench for divider	My_divider
Processor_sample_tb	**Test bench for procesor unit**	**Processor_sample**
readWriteCache_tb	**Test bench for complete unit for 2D Filter**	**Cahe_sample_new, processor_sample**