Final Report for Real Time Imaging

Stefanis Ioannis (jstefanis@teicrete.gr)
Jamilu Sulaiman (jamilu.sulaiman2602@gmail.com)

MsCV2 2018-2019
Universite de Bourgogne

Table of Contents

1. **Preface**

This report describes and analyses our methodology for implementing a fast processing pipeline for image editing by using VHDL. Very High Definition Language is an industry standard for fast producing configurable and validate code for hardware design.

2. **Description of the problem**

The problem we have to solve considers as an input a grayscale image of dimension 128x128 of Lena.

The Lena image
128x128 pixels

This image represents the grayscale input from a camera connected at the input side of the processing pipeline.

Our first task is to read the camera input (the image file) and store it in a FIFO (first in-first out) component for transfer. After, a memory cache will transform serial data to parallel and send them to the filtering component for spatial filtering of the image.
What interests us is how we will be able to exploit the high speed that is achievable in the case of hardware image processing for spatial image filtering. For achieving this we have to think smart ways for data pipelining and processing.

At the other end of the output, a file will store the values of the processing results for comparison with a similar software algorithm in Matlab platform.

3. **Structure of the project**

The implementation files
Due to VHDL versatility and the features of XILINX ISE 14.7 simulation and VHDL development Platform that we used, we were able to collaborate and work more efficiently.
Our project has its components split in different .vhd source files. These .vhd source files build a hierarchy for synthesizing more complex components from simpler ones.
The naming convention is according to their operation.
For example the prototype of a D flip-flop component will be stored at the file **d_ff.vhd** .

Creating test benches
Apart from the components that create the implementation of our solution, we have to develop the test bench were we will be able to test the performance of individual components but also their combination to more complex structures.
These files will have a suffix _tb which implies that they are a test bench in their file name. For example the name of the test bench of the D Flip Flop will be **d_ff_tb.vhd**

Using pre-built hardware components
Finally, we are able to use built in ip cores with ip meaning intellectual property which are optimized hardware components for every development suite which help us to use the functionality of more complex hardware structures directly in our projects. The only thing we have to do is to include them in our project structure and use them directly as components.

This solution was used to implement the various FIFO components in our project despite than creating all the submodules needed to creating an operating FIFO, with all the problems that follow it as mistakes from unexperienced users and bad performance in comparison with the optimized one.

4. **The Scenario**

An overview for a solution for this problem
For justifying the effort of developing a solution to this kind of problem we have to be able to think in hardware terms. We have to think about concurrency of the processes and also the exploitation of low level operations which are really fast in hardware level.

Our problem can be split in two parts. The first part is considering the transform of the serial transfer of the image data from a camera (a fifo in reality) in a process that uses a parallel data bus for achieving faster results for the pixel transfers. An image can be depicted as an 2D matrix, so it's a kind of problem that for our filtering process we have to take under consideration the N ± 1, M ± 1, pixels in rows and columns and we can see great throughput when implemented in parallel.

The Memory Cache
For this reasons the first part of our solution takes care about the memory caching of the image data. The data enter in a serial form into our circuit and they have to be converted to parallel ones. This component will be able to convert the serial data to parallel ones to be able to send them directly to the spatial image filter.

This component is implemented by a number of D Flip-Flops for control and pixel transfer and also some FIFO components imported as ip cores directly to the project.

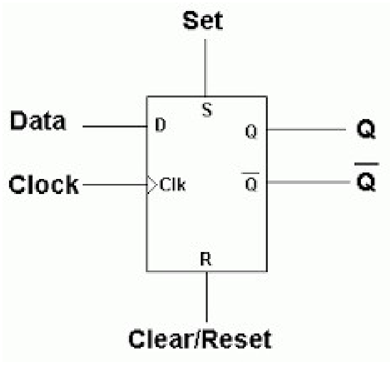
The Spatial Image Filter
The Spatial Image Filter will be able to perform pixel manipulation in a spatial manner to achieve a desired output. For example we can apply the average filter by applying an convolution process with the pixel value and a neighbor (usually a 3x3 one). The outcome will be a smoothed version of the image with each pixel influenced equally from the pixels of its neighborhood.

The Output
The output will be again serialized by a FIFO component and saved in a file for further comparison and analysis with a software process.
The components for File Input and Output are given ready for the implementation of the project.

5. **Detailed description of the components for memory caching and filtering**

The Memory Cache
The Memory cache consists from various subcomponents. The components which are useful for our serial to parallel converter are D Flip-Flops and FIFO Cores. Altogether, they form a Shift Register cirquit for shifting the serial data and converting them to parallel. The FIFO components are responsible for buffering the data for the presentation of each row of the image.

The Flip-Flop
In electronics, a **flip-flop** or **latch** is a circuit that has two stable states and can be used to store state information. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems.
D Flip-flops are used as a part of memory storage elements and data processors as well. D flip-flop can be built using NAND gate or with NOR gate. Due to its versatility they are available as IC packages. The major applications of D flip-flop are to introduce delay in timing circuit, as a buffer, sampling data at specific intervals. **Whenever the clock signal is LOW, the input is never going to affect the output state**. The clock has to be high for the inputs to get active. Thus, D flip-flop is a controlled Bi-stable latch where the clock signal is the control signal. Again, this gets divided into **positive edge triggered D flip flop and negative edge triggered D flip-flop**. Thus, the output has two stable states based on the inputs. (source Wikipedia)



A D Flip Flop

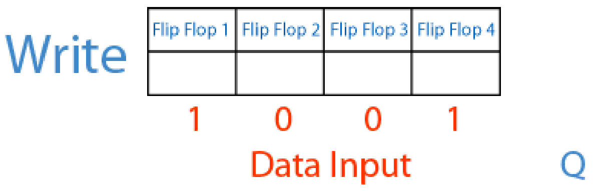
Inputs				Outputs	
S	R	D	>	Q	Q'
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	1	1

The Truth Table of a D Flip Flop

These flip-flops are very useful, as they form the basis for shift registers, which are an essential part of many electronic devices.

A Shift Register

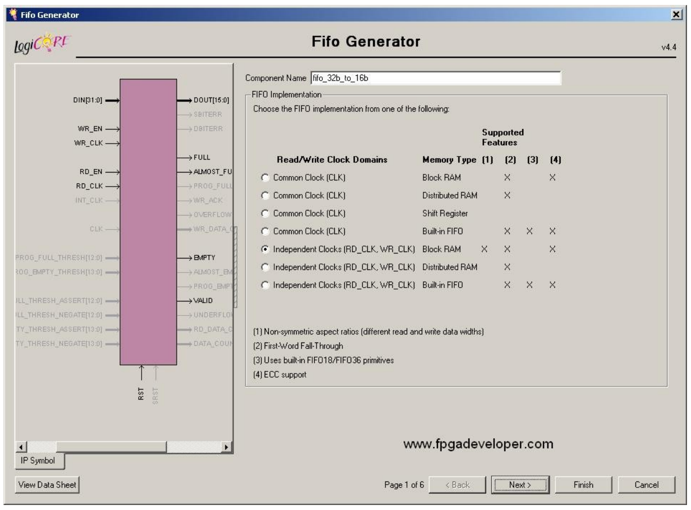
In digital circuits, a **shift register** is a cascade of flip flops, sharing the same clock, in which the output of each flip-flop is connected to the 'data' input of the next flip-flop in the chain, resulting in a circuit that shifts by one position the 'bit array' stored in it, 'shifting in' the data present at its input and 'shifting out' the last bit in the array, at each transition of the clock input.



A Shift Register (source Wikipedia)

The FIFO

FIFO is an acronym for **first in, first out**, a method for organizing and manipulating a data buffer, where the oldest (first) entry, or 'head' of the queue, is processed first. Depending on the application, a FIFO could be implemented as a hardware shift register, or using different memory structures, typically a circular buffer or a kind of list. The ends of a FIFO queue are often referred to as *head* and *tail*. A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing. Asynchronous FIFOs introduce metastability issues. We will use a sychronous type of FIFO in our implementation (wikipedia).



Fifo Generator v4.4

Component Name fifo_32b_to_16b

FIFO Implementation

Choose the FIFO implementation from one of the following:

		Supported Features			
Read/Write Clock Domains	Memory Type	(1)	(2)	(3)	(4)
Common Clock (CLK)	Block RAM		X		X
Common Clock (CLK)	Distributed RAM		X		
Common Clock (CLK)	Shift Register				
Common Clock (CLK)	Built-in FIFO		X	X	X
[✓ Independent Clocks (RD_CLK, WR_CLK)]	Block RAM	X	X		X
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM		X		
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO		X	X	X

(1) Non-symmetric aspect ratios (different read and write data widths)
(2) First-Word Fall-Through
(3) Uses built-in FIFO18/FIFO36 primitives
(4) ECC support

www.fpgadeveloper.com

A Xilinx FIFO IP Core configuration

Our implementation of the memory cache.

In our image averaging problem we consider a neighborhood of 3x3 for the pixel under process. This means that for a performance efficient solution for filtering we need to have for each pixel the previous and the next row pre-cached for processing.

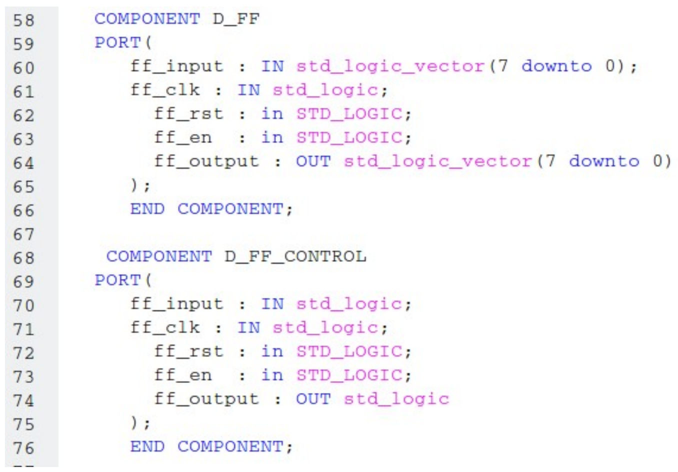
In our case with the Lena photo, we have an 128x128 image. Apart from the outer row and columns problem that we will consider later, the pixel at the point p(i,j) will need the pixels (i-1,j-1),(i-1,j),(i-1,j+1), (i,j-1),(i,j+1),(i+1,j-1),(i+1,j),(i+1,j+1). Each pixel row has a length of 128 pixels.

For example considering pixel p(2,2) we will need the following pixels:

(1,1),(1,2),(2,3),   (1),   (2),   (3)   (1),   (2),   (3)
(1,1), (p) , (1,3), or   (129),   (p), (131) or   128 + (1),   (p),   (3)
(2,1),(1,2),(2,3)   (257), (258), (259)   2*128 + (1),   (2),   (3) ...

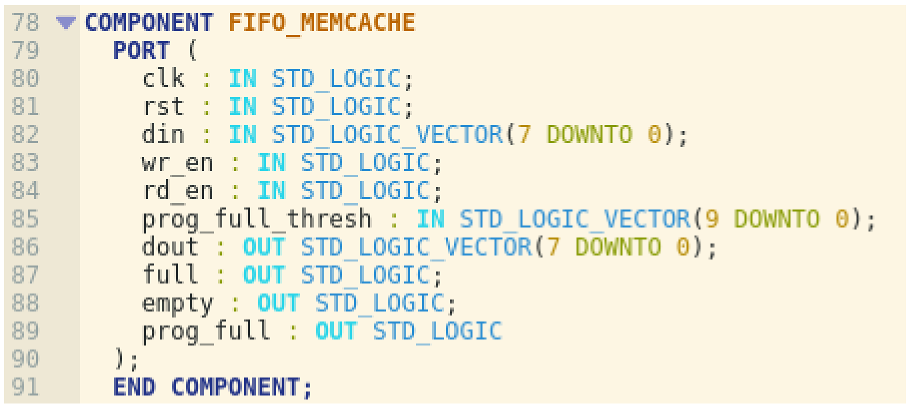
This problem can be resolved by creating a memory structure which can hold the pixels of each row and feed them to the filter in parallel. This serial to parallel memory structure can enable a concurrent processing for each pixel for achieving the spatial averaging outcome.
This memory structure will implemented by a series of D Flip-Flops operating as sift registers, controlled by another set of Flip-Flops which will be responsible for their enabling and connected with FIFO components which will operate as buffers for the 128 pixel offset for each image row.

The D Flip Flop implementation
If we open the d_ff.vhd file we can see the declaration of ports of the D Flip Flop entity



```
COMPONENT D_FF
PORT(
   ff_input : IN std_logic_vector(7 downto 0);
   ff_clk : IN std_logic;
     ff_rst : in STD_LOGIC;
     ff_en  : in STD_LOGIC;
     ff_output : OUT std_logic_vector(7 downto 0)
   );
   END COMPONENT;

 COMPONENT D_FF_CONTROL
PORT(
   ff_input : IN std_logic;
   ff_clk : IN std_logic;
     ff_rst : in STD_LOGIC;
     ff_en  : in STD_LOGIC;
     ff_output : OUT std_logic
   );
   END COMPONENT;
```

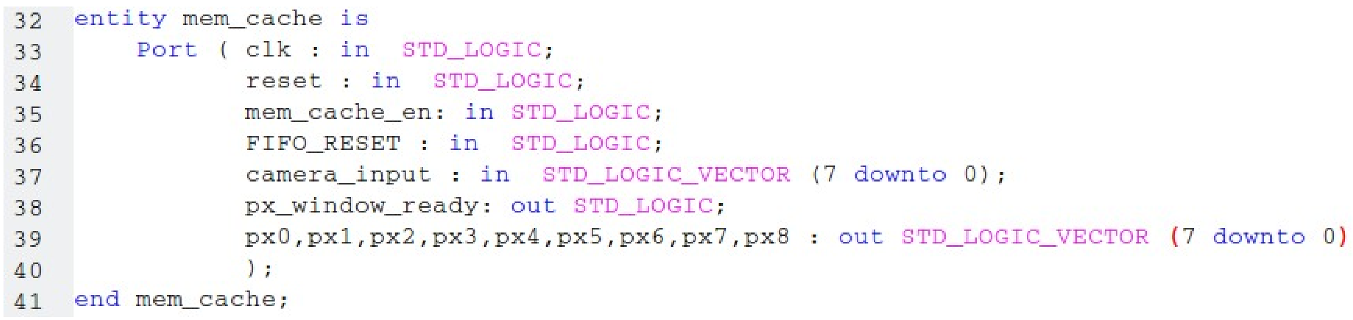
We have an enable **ff_en** logical input port.
An reset **ff_rst** logical input port.
A clock **ff_clk**.
The logical input **ff_input** and output **ff_output** .
This entity will be instantiate many times for the creation of the memory cache component.
The same structure also has the control D Flip-Flop and the only difference is behavioral.
The FIFO ip core implementation
We use the provide FIFO ip core of Xilinx with our customization for the input and output ports and also the behaviour of the component in our circuit. We have chosen an synchronous FIFO component for our problem.
If we open the **mem_cache.vhd** file we can see the port declaration for the FIFO ip core.



```
COMPONENT FIFO_MEMCACHE
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    prog_full_thresh : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    prog_full : OUT STD_LOGIC
  );
  END COMPONENT;
```

The FIFO takes as inputs the clock signal **clk**, the fifo reset **rst**, the write enable **wr_en** input that enables the writing of the data into the fifo, the **rd_en** input which enables the reading of the contents of the FIFO.
The **dout** which is the output of the FIFO, an 8 bit std logic vector which represents an 8bit pixel, the **empty** output signal which is high when the FIFO is empty.
The **full** output signal which is high when the FIFO is full.
The **din** which is an 8 bit std logic vector input which represents an 8bit pixel entering the FIFO for store.
The **prog_full**: This signal is asserted when the number of words in the FIFO is greater than or equal to the assert threshold.
The **prog_full_ thresh** This signal is used to input the threshold value for the assert
ion and de-assertion of the programmable full (**PROG_FULL**) flag. The threshold can be dynamically set in-circuit during reset.
After experimenting with the various values of **prog_full_thresh,** we discovered that the value of **123** is the adequate value for stop writing into the FIFO for compensating the lost clock cycles between the initialization of the FIFO and the time when the first pixel is available in the output **dout**. We have to understand the problem of shifting in terms of clock delays into the memory cache entity.

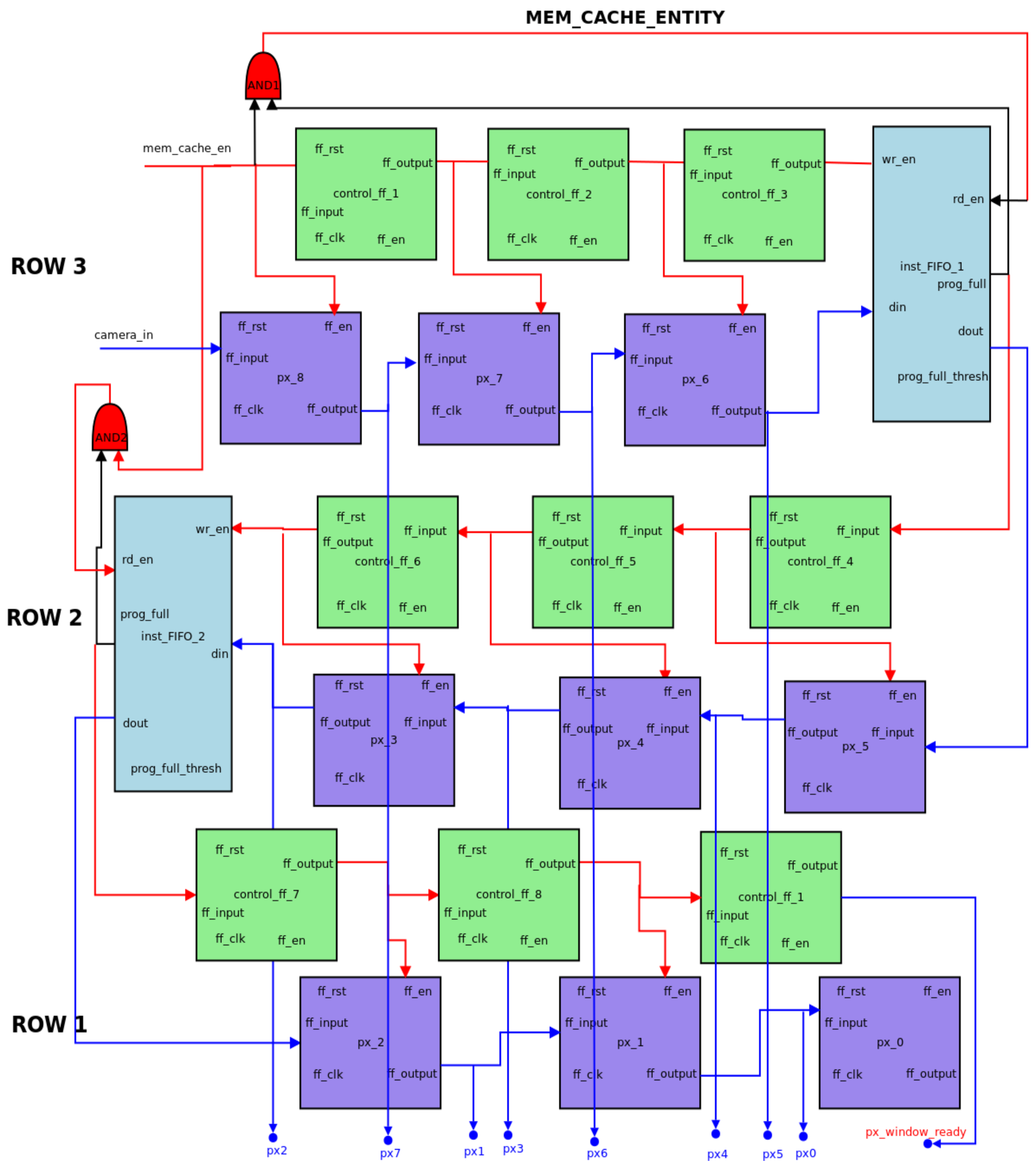
The cycle of data handling in memory cache.
The memory cache entity



```
entity mem_cache is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           mem_cache_en: in STD_LOGIC;
           FIFO_RESET : in  STD_LOGIC;
           camera_input : in  STD_LOGIC_VECTOR (7 downto 0);
           px_window_ready: out STD_LOGIC;
           px0,px1,px2,px3,px4,px5,px6,px7,px8 : out STD_LOGIC_VECTOR (7 downto 0)
           );
end mem_cache;
```

The memory cache entity takes as logical input the **clk** signal, the **reset** signal, the **mem_cache_en** for enabling the entity and the **FIFO_RESET** for resetting the FIFOs in the initial condition.
It also takes as an input in the **camera_input** port an std logic vector of 8bits which receives a pixel per clock.
The outputs of the memory cache entity is the **px_window_ready** output which asserts a flag when the entity is ready to output the pixels to the filter entity and also the nine 8bit std logic vector pixel outputs from **px0** to **px8** which send the prepared pixel data to the filter entity.
The D flip-flops belong in two categories. The control D FF and the shift D FF. The shift D FF as the name implies shift the pixels along the full memory cache pipeline passing by the two FIFO instances of the memory cache. The control FF on the other hand control this data flow inside the circuit.
The nine control flip-flops are named control_**ff_1** to control_**ff_9**. The nine pixel shifting flip-flops are named from **px_8** to **px_0**. There is also two and gates (**AND1**, **AND2**) used for control signal manipulation andl also two FIFO's, **inst_FIFO_1** and **inst_FIFO_2**.
When the input signal **mem_cache_en** is asserted, the control flip-flop propagate this signal once per clock cycle. In the first row of pixel FF we have also 3 control FF. On each clock cycle the pixel flip-flops are enabled sequentially and also shift the pixel they get from camera_in input.
The output of the third control FF connects directly with the input port **wr_en** and enables the write operation to the first FIFO, inst_FIFO_1. When **wr_en** is high, the FIFO starts to write data received to the port din. This FIFO fills with pixel data until **prog_full_threshold** is asserted which counts the number of words written into the fifo. Actually the first FIFO buffers the pixel data for

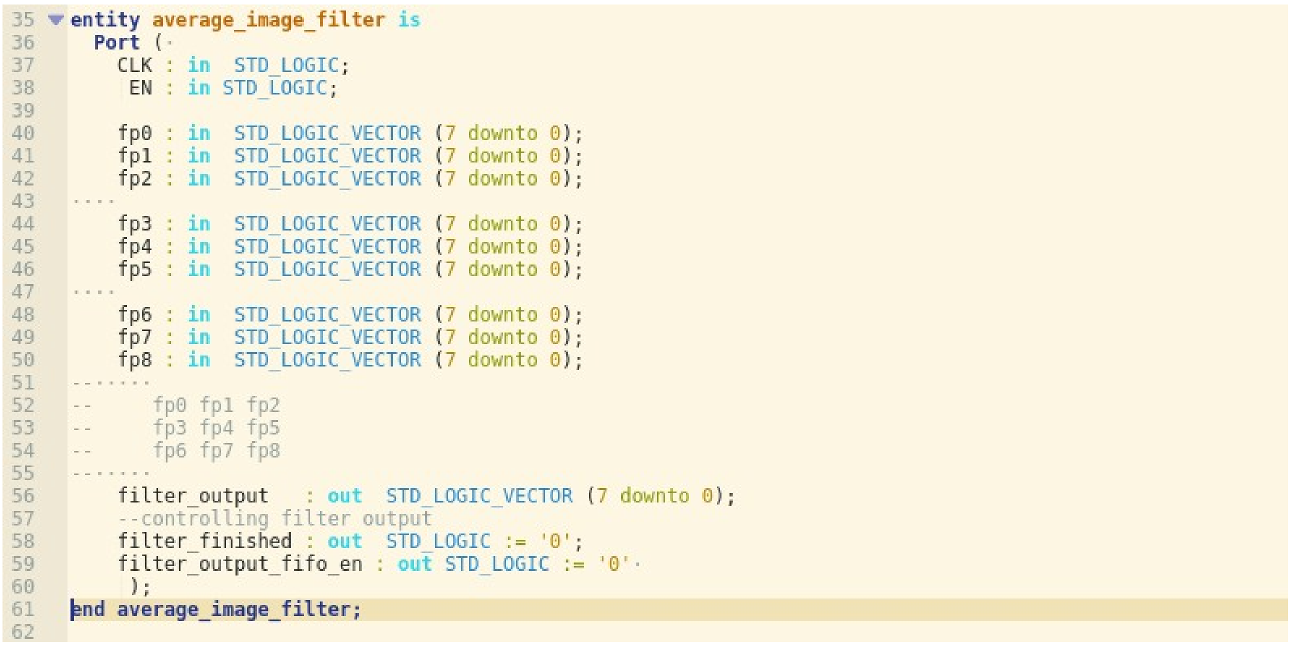
the first line of the image. When **prog_full_threshold** is asserted, the **prog_full** port is also asserted high.
This port is connected with the gate AND1 as input along with the signal **mem_cache_en**. The output of the AND1 gate is connected with the input port **rd_en** of the i**nst_FIFO_1**. So when the threshold has been reached and the **mem_cache** still receives an enable signal, the **inst_FIFO_1** performs an read operation, sending the written data from the **dout** output port to the next pixel flip-flop **px_5**.
The AND1 gate is necessary because we want to be able to halt the FIFO read operation when the mem_cache_en signal is not received any more. The **prog_full threshold** is set as a constant empirically to the value **123** in relation with the **128** pixels per row, because we have also to compensate for the data delay inside the FIFO.



On the second row of the **mem_cache** entity we have a set of **three control FF** and **three pixel FF**. The **control_ff_4** is connected to the **prog_full** output port of the first fifo, so when its asserted, the control flip flop propagates the enable signal to the next control ff and also enables the fourth pixel ff, **px_4**. After three cycles, the data reach the second FIFO, **inst_FIFO_2**. This FIFO is responsible for caching the pixels for the second row of the input image.
The connections are similar with the first FIFO, with the **din** connected to the **px_3** flip-flop, the **wr_en** connected to the sixth control ff, **control_ff_6** and also a second and gate **AND2** connected with **prog_full** and **mem_cache_en** as inputs and **rd_en** to the output. So when the programmed full threshold has been reached, the write operation to the FIFO halts and starts to output pixels from the output port **dout**. **dout** is connected with the last row of pixel flip-flops and actually with FF **px_2**. The **prog_full** output port is asserted high and enables the first control FF of the third row control_**ff_7** to propagate the enable signal to the next control FF but also to enable pixel FF **px_2** to read the pixel data from dout. At the end of the control ff there is an entity output port called **px_window_ready** which asserts higha as an enable signal to the next entity, the filter entity and that the pixels are cached and ready for processing.
Finally the pixel ff from **px0** to **px8** have the necessary pixels for creating an 3x3 mask and performing the spatial filtering. In each clock cycle, the two fifos buffer the rest pixels of the row and the filtering process flows without obstruction.

The filter entity

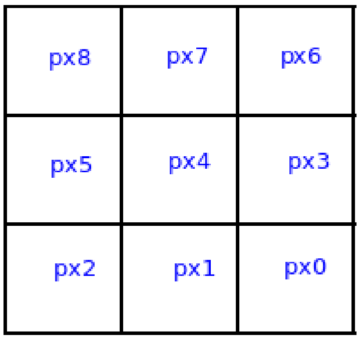
The second entity we created exists into the **average_image_filter_vhd**.



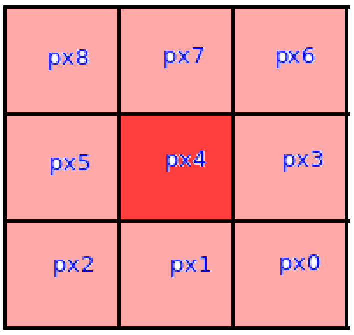
```vhdl
entity average_image_filter is
  Port (
    CLK : in  STD_LOGIC;
     EN : in STD_LOGIC;

    fp0 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp1 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp2 : in  STD_LOGIC_VECTOR (7 downto 0);

    fp3 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp4 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp5 : in  STD_LOGIC_VECTOR (7 downto 0);

    fp6 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp7 : in  STD_LOGIC_VECTOR (7 downto 0);
    fp8 : in  STD_LOGIC_VECTOR (7 downto 0);
--------
--     fp0 fp1 fp2
--     fp3 fp4 fp5
--     fp6 fp7 fp8
--------
    filter_output   : out  STD_LOGIC_VECTOR (7 downto 0);
    --controlling filter output
    filter_finished : out  STD_LOGIC := '0';
    filter_output_fifo_en : out STD_LOGIC := '0'
     );
end average_image_filter;
```

The average_image_filter entity has as inputs the clock **clk** and the enable ports and **fp0** to **fp8** pixel inputs. The outputs are:
filter_output, the 8bit pixel output of the current filtering process cycle.
filter_finished, a flag that informs that filter entity has finished image processing
filter_output_fifo_en. This is a flag that sends a signal to an external FIFO that will be responsible for getting the processed pixels sequentially and storing them to an output file. When this flag is raised the **wr_en** flag of the output FIFO will enable it to store pixels.

The filter operation
The average image filter uses a kernel of 3x3 neighborhood. The central pixel of this neighborhood is the pixel under processing. In the most generic form of this filter we just have to add all the intensity values of the pixel neighborhood and divide it by the size which is nine. Then the current value of the central pixel of the kernel is replaced with this average. In our case though we have to think in a different manner. Integer division is a really expensive operation in hardware so we will prefer to find a faster but less accurate method.



For spatial filtering we will use a neighborhood 3x3. The pixel under process is px4



the central pixel value is the average of its neighborhood

For this reason we will exploit the binary arithmetic. By shifting a binary number to the right, we divide it by two each time. This means that with a shift of 3 bits we have a division by eight.

So our solution instead of using nine values and taking the average, it will use 8 pixel values, omitting the value of the central pixel under process.
This is a nice approximation which boosts the performance of the filter. The other performance trick is to parallelize the addition of the neighborhood pixels. For this, we will create a tree of partial additions of depth three. In the first level we need 4 additions, in the second two so the result is seven additions for the final result. With simple bit manipulation we can have our result with really great performance. The full averaging operation is performed inside the process averaging.

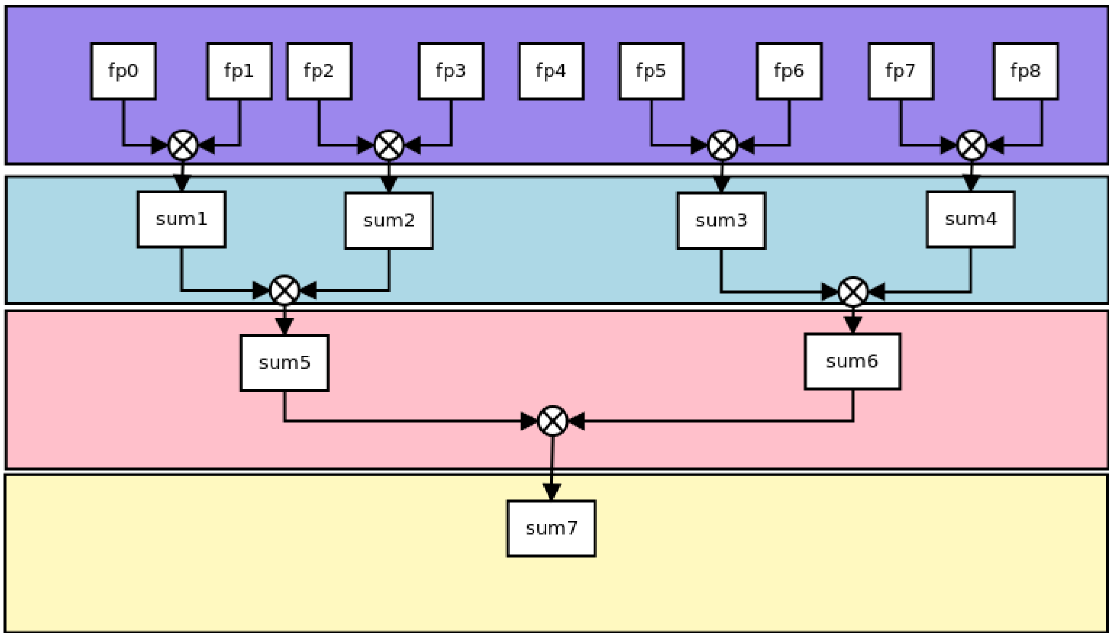
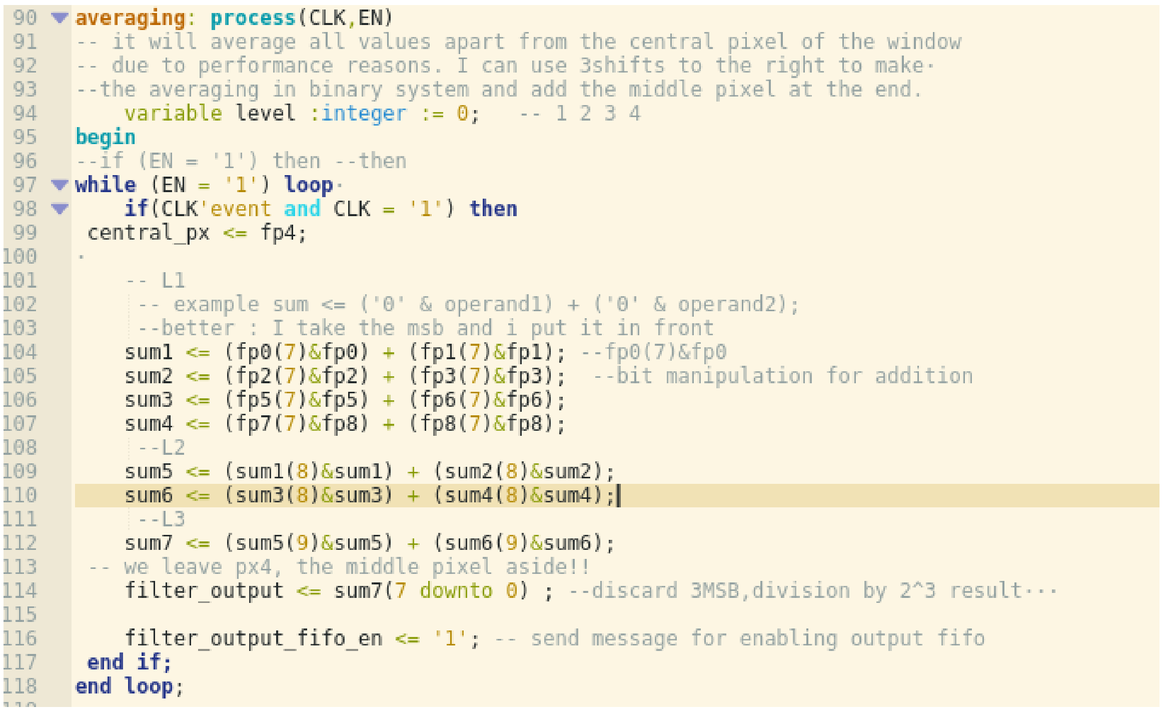


Image filtering additions tree

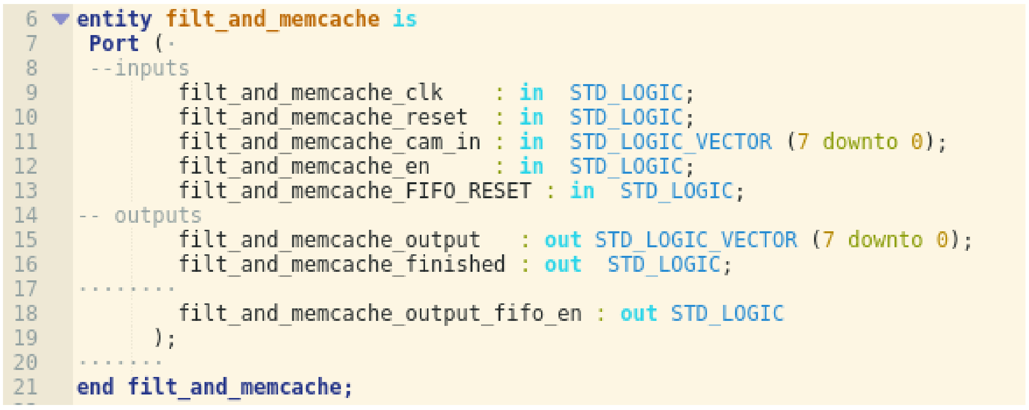


```vhdl
 90 averaging: process(CLK,EN)
 91 -- it will average all values apart from the central pixel of the window
 92 -- due to performance reasons. I can use 3shifts to the right to make
 93 --the averaging in binary system and add the middle pixel at the end.
 94     variable level :integer := 0;    -- 1 2 3 4
 95 begin
 96 --if (EN = '1') then --then
 97 while (EN = '1') loop
 98     if(CLK'event and CLK = '1') then
 99  central_px <= fp4;
100
101     -- L1
102      -- example sum <= ('0' & operand1) + ('0' & operand2);
103      --better : I take the msb and i put it in front
104     sum1 <= (fp0(7)&fp0) + (fp1(7)&fp1); --fp0(7)&fp0
105     sum2 <= (fp2(7)&fp2) + (fp3(7)&fp3);  --bit manipulation for addition
106     sum3 <= (fp5(7)&fp5) + (fp6(7)&fp6);
107     sum4 <= (fp7(7)&fp8) + (fp8(7)&fp8);
108      --L2
109     sum5 <= (sum1(8)&sum1) + (sum2(8)&sum2);
110     sum6 <= (sum3(8)&sum3) + (sum4(8)&sum4);
111      --L3
112     sum7 <= (sum5(9)&sum5) + (sum6(9)&sum6);
113  -- we leave px4, the middle pixel aside!!
114     filter_output <= sum7(7 downto 0) ; --discard 3MSB,division by 2^3 result...
115
116     filter_output_fifo_en <= '1'; -- send message for enabling output fifo
117  end if;
118 end loop;
```

filter averaging process

We have created the signals **sum1,sum2,sum3,sum4,sum5,sum6,sum7** for keeping the intermediate results in each level of additions.
sum1 to **sum4** are also **9bit std_logic_vector signals**. In each addition we attach in front of the operands their Most Sighnificant Bit. The extra bit keeps the carry of the addition.
Similarly to the next level **sum6,sum5** are **10bit std_logic_vector signals**. **Sum7** is **11bit**.
Finally at the line 114 where **filter_output <= sum7(7 down to 0)** we discard the 3 **MSB** of the sum7 result and automatically result to a division by 8.The value of **filter_outpu**t is the result output of the filter process.

Connecting memory cache and average filter
The next step is to connect the memory cache and the average filter to an uniform entity.
For this reason we created the file **filt_and_memcache.vhd**.



```vhdl
entity filt_and_memcache is
 Port (
 --inputs
        filt_and_memcache_clk    : in  STD_LOGIC;
        filt_and_memcache_reset  : in  STD_LOGIC;
        filt_and_memcache_cam_in : in  STD_LOGIC_VECTOR (7 downto 0);
        filt_and_memcache_en     : in  STD_LOGIC;
        filt_and_memcache_FIFO_RESET : in  STD_LOGIC;
-- outputs
        filt_and_memcache_output   : out STD_LOGIC_VECTOR (7 downto 0);
        filt_and_memcache_finished : out  STD_LOGIC;

        filt_and_memcache_output_fifo_en : out STD_LOGIC
      );

end filt_and_memcache;
```

This entity has as inputs the **filt_and_mem_cache_clk**, **filt_and_mem_cache_reset**, **filt_and_mem_cache_en**, **filt_and_mem_cache_FIFO_RESET** signals.
It also has the **cam_in** port as the pixel serial input to the memory cache. The outputs are an 8bit pixel output of the filter **filt_and_memcache_output** which sends the result to the OUTPUT FIFO, the flag **filt_and_memcache_finished** which asserts high when the process has finished and finally the flag

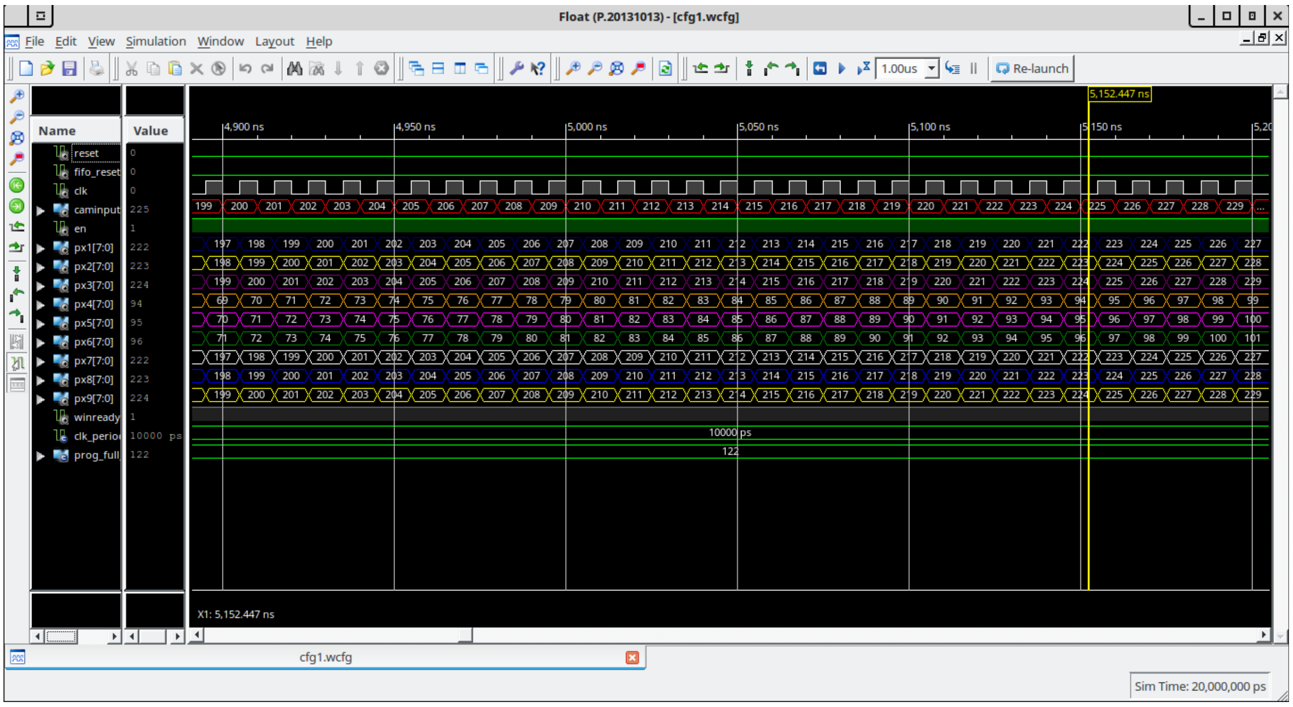
filt_and_memcache_output_fifo_en which sends an write enable signal to the external FIFO connected to **filt_and_memcache_output** port. This entity will be used in the final design of our project.

File input and output

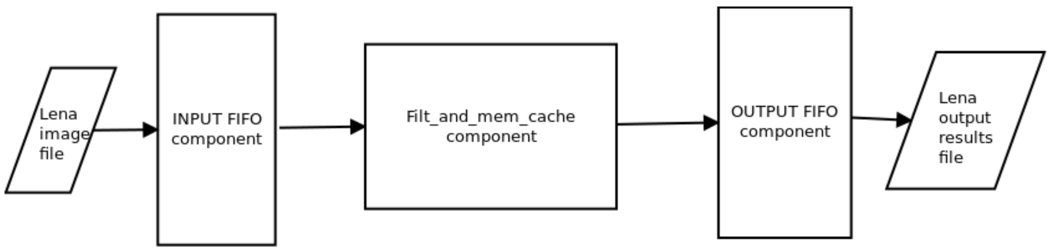
For the needs of reading the image data and writing the results, a file with the given entity example was given to us. Based on this file we created a file called which includes the required components for reading the data, saving the output and doing the processing.
In this file apart from the entity component **filt_and_memcache**, we created two FIFO instances. **INPUT_FIFO** and **OUTPUT_FIFO**. These **FIFOs** are responsible for the image input pixel by pixel and also the storage of the filtering result to a file named "**Lena128x128g_8bits_r.dat**" with the results. For the operations of the various components we have created the functions **reset()** for reseting the digital circuit. **fifo_reset()** for resetting the FIFOS of the circuit, **read_fifo_data()** for sending the data stored to a fifo, **write_fifo_data()** for reading the image.
read_fifo_to_f_memcache() process which enables the **filt_and_memcache** instance for reading the image data from the **INPUT_FIFO** and filtering.
The function **write_f_memcache_to_output_fifo()** sends the results of the filtering process to the **OUTPUT_FIFO** for storing them after to a file.

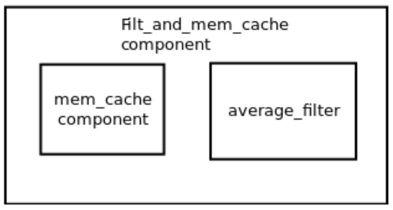
The testbench

For each entity we created a testbench for testing and debugging the performance of our solution. After, by using the ISIM app we can run a simulation and test the results.



Simulating memcache entity





The various instances of the image filtering component

6. **Conclusions**

VHDL is a really powerfull language for hw prototyping. On the other hand the compiling and the creation of the fpga is really time consuming. Considering the problem of pixels on the edges and the first line, It is a fact that after getting to the second line of image data, were able to have all the available pixels for image filtering.
One strategy should be to just copy the first line and shift all the processing for one pixel to the right and stop one pixel before the end. This should create a result of 126x127 pixels. Otherwise, and this is what we chose, you can just ignore it.

7. **References**

[1] DESIGN FOR EMBEDDED IMAGE PROCESSING ON FPGAS Donald G. Bailey, 2011,IEEE
[2] The course material.
[3] FPGA prototyping by VHDL examples. Pong P. Chu,2008,Wiley