Informatics and Mathematical Modelling
Technical University of Denmark

02207 : Advanced Digital Design Techniques

Low-pass Filter (2 x 1-D)

Examination Project

Group *dt07*

Markku Eerola (s053739)

Rajesh Bachani (s061332)

Josep Renard (s071158)

December 12, 2007

Contents

1 About the Report

In this project, we have designed and implemented the architecture for an image filtering processor which uses a 3x3 filter to perform a convolution on the image of size 256x256 pixels.

Following is the work done by the authors.

Authors by Section

- *Rajesh Bachani*
  - VHDL: FSM_In.vhd, FSM_Out.vhd, Processor_3.vhd, TB_Filter.vhd
  - Report: Sections 3 and 4
- *Josep Renard*
  - VHDL: Multiplier.vhd, parcial.vhd, CRA_15.vhd, CSA_15bit.vhd
  - Report: Sections 6 and 7
- *Markku Eerola*
  - VHDL: Adder_2.vhd, Adder_3.vhd, Mux_4.vhd, SHIFTREG.vhd, Memory.vhd, REG.vhd
  - Report: Sections 2 and 5

The rest of the report is organized as follows. In section 2, we explain the internal architecture of the processor. Then in section 3, we give the sequencing of the operations involved in the computation. This section explains the operations performed for memory initialization and the order in which the input memory is read and the output memory is read and written. Also, we here explain the order in which the memory is accessed, which are different for the horizontal and vertical movements of the filter mask. Section 4 explains the design of the controllers at the input and the output. In section 5, we provide the results from the synthesis of the design in Design Vision. The section 6 contains the images obtained after convolution, and also a summary of the results from the synthesis. Finally, we end the report with section 7 with a short explanation on what we think are the limitations of the work done here, and how it could be extended.

2 Design Architecture

The overall design of the filter unit can be seen in figure 1. More detailed architecture can be seen in figure 2.

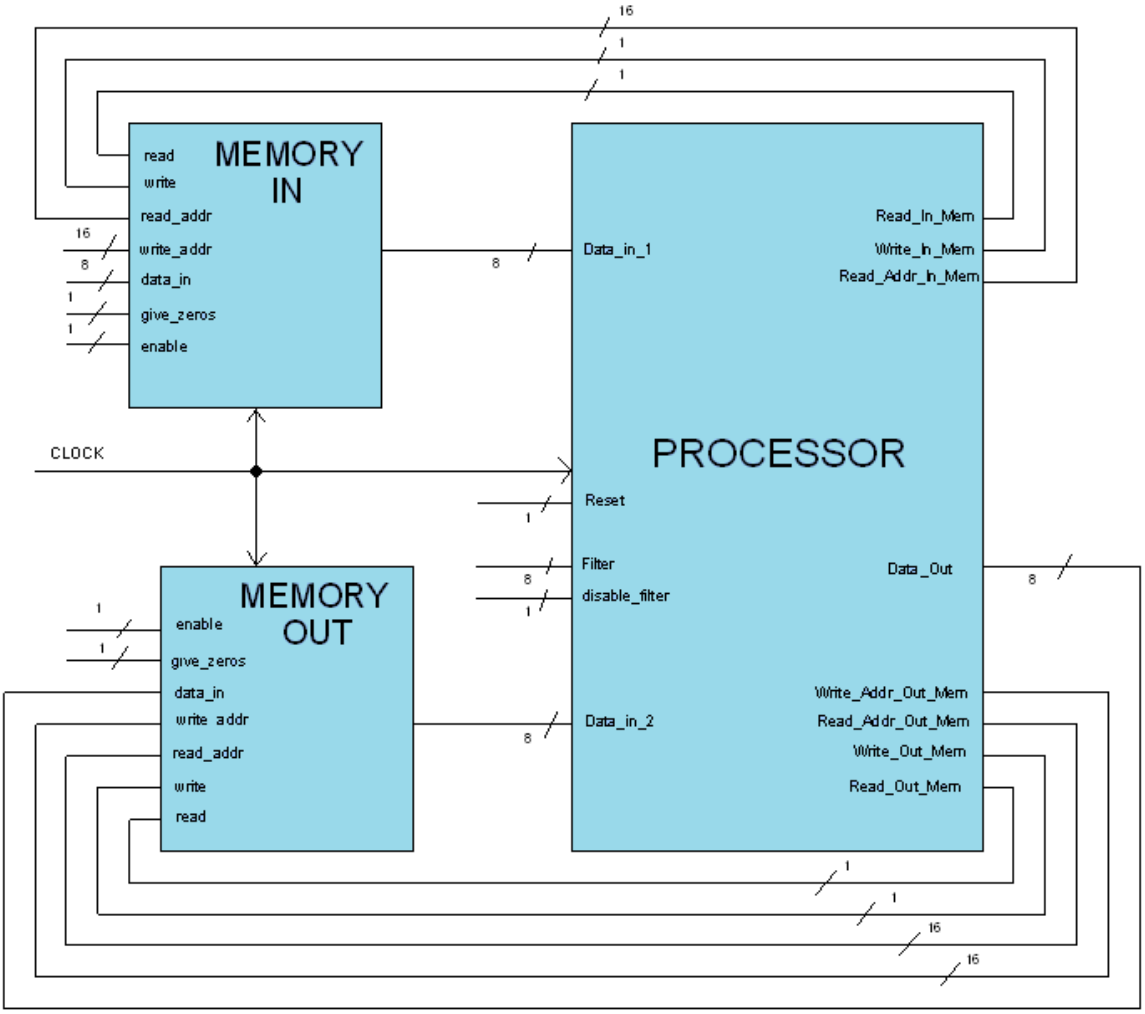


Figure 1: Filter Unit Design

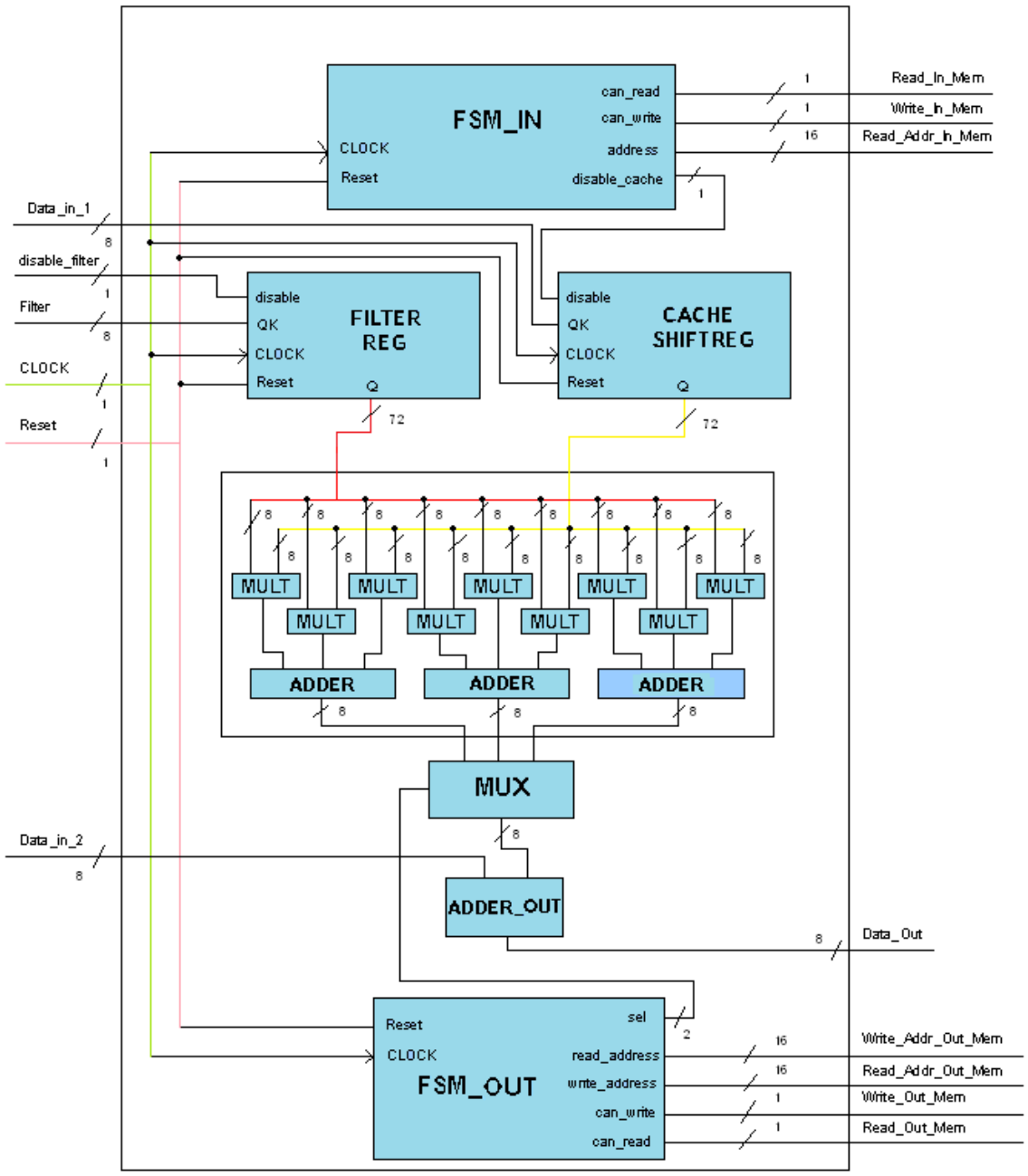


Figure 2: Processor Architecture

The input and the output signals of the processor are explained below, with their bit-width's.

- *Data_in_1*, 8 bits, transfers pixels of the image from the input memory to the processor's cache
- *Clock*, 1 bit, the common clock signal
- *Reset*, 1 bit, the common active low reset signal
- *Filter*, 8 bits, used to get the n pixels of the filter, 8 bits over n clock cycles ($n = 3$ here)
- *Disable_filter*, 1 bit, used to disable the filter, so the filter register is not written
- *Data_in_2*, 8 bits, transfers pixels of the image from the output memory to the processor
- *Read_In_Mem*, 1 bit, indicates the memory when to read pixels of the input image
- *Write_In_Mem*, 1 bit, indicates the memory when to write pixels to the input image (this is used only when the input memory is initialized)
- *Read_Addr_In_Mem*, 16 bits, indicates the address from where the next pixel should be read from the input memory
- *Data_Out*, 8 bits, transfers processed pixels from the processor to the output memory
- *Write_Addr_Out_Mem*, 16 bits, indicates the address to which the next pixel should be written in the output memory
- *Read_Addr_Out_Mem*, 16 bits, indicates the address from where the next pixel must be read from the output memory
- *Write_Out_Mem*, 1 bit, indicates the output memory when to allow writing of data
- *Read_Out_Mem*, 1 bit, indicates the output memory when to allow reading of data

3 Sequencing of Operations

This section contains a description of the sequence in which the processor performs the operations needed for convolution of the image.

3.1 Memory Initialization

Before the processor starts its operations, we have the memory initialization step, in which both the input and the output memory's are set to initial values. The input memory is initialized with the image pixels, which is done through the testbench. The output memory has to be initialized to zero.

3.2 Memory Read and Write by Processor

There are two controllers as part of the processor, the Input Controller and the Output Controller (indicated by FSM_In and FSM_Out in figure 2). The Input Controller is responsible for reading pixels from the input memory, which holds the original image. On the other hand, the output controller is responsible for reading as well as writing the computed pixels from and to the output memory. The output memory at the end of the computation, holds the pixels of the convoluted image.

It is very important that these two controllers be well synchronized with each other, so the operations are performed smoothly, and there is no data loss. In particular, when the Input Controller is active, the Output Controller should not perform any operation. This is because until the Input Controller has read the next 3 pixels (for the 3x3 filter; it would be n for nxn filter) from the input memory, the convolution is not stable, and so the Output Controller cannot write anything to the output memory. The vice-versa case is also true. So, when the Output Controller is active, the Input Controller should be inactive. This is because while the output memory is being written by the Output Controller, if the Input Controller reads new pixels, then the already computed values would be overwritten, and the synchronization is disturbed completely.

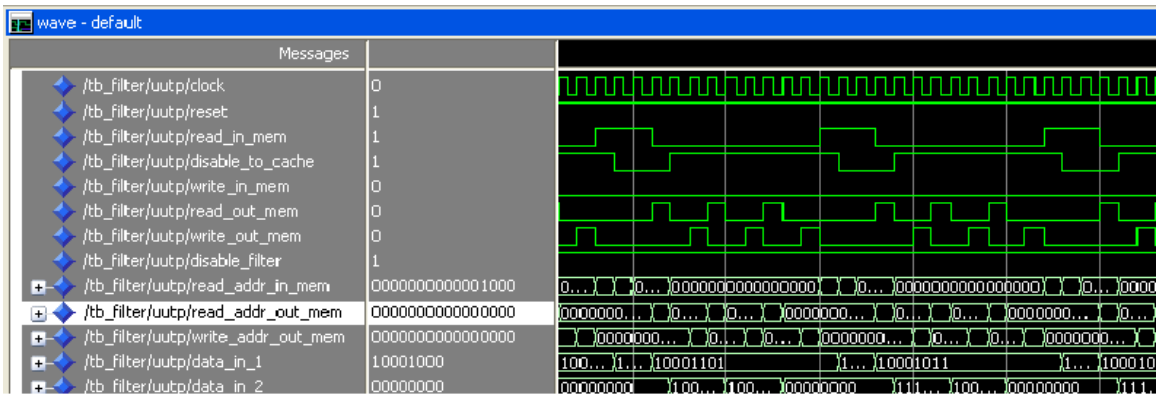


Figure 3: Read Write Synchronization between Input and Output Controllers(1)

Hence we have chosen the approach in which at no point of time, would both the Controllers be active, as shown in figure 3. As we can see, the signal *read_in_mem* is *1* for three clock cycles. The signal *disable_to_cache* becomes *1* after one clock cycle delay as compared to *read_in_mem*. Also, once the *read_in_mem* becomes *0*, the *read_out_mem* becomes *1*. Also at this time, the *disable_to_cache* becomes *0*. Then there is no activity for one clock cycle, during which the convolution is done, after which *write_out_mem* becomes *1* to write the convoluted pixel to the output memory. The *read_in_mem* then becomes *1* again after nine clock cycles, i.e. when the Output Controller has finished reading and writing three convoluted pixels to the output memory.

During the horizontal movement, when the horizontal end of the image is reached, the Output Controller becomes idle for a longer duration. This is because the Input Controller now has to load nine fresh pixels from the input memory, which happens in 27 clock cycles. So, once the end of the image is reached (this holds even for vertical movements), the Output Controller becomes idle for 27 cycles, and then starts reading and writing as usual, which goes on uptil the end of the image is again reached (which is 256 rounds of read and write).

Figure 4 depicts this situation in the simulation.

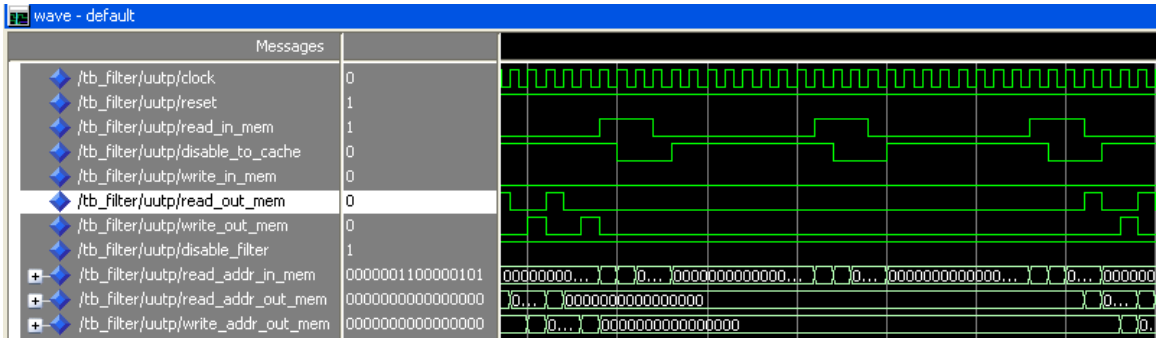


Figure 4: Read Write Synchronization between Input and Output Controllers(2)

3.3 Memory Access Sequence

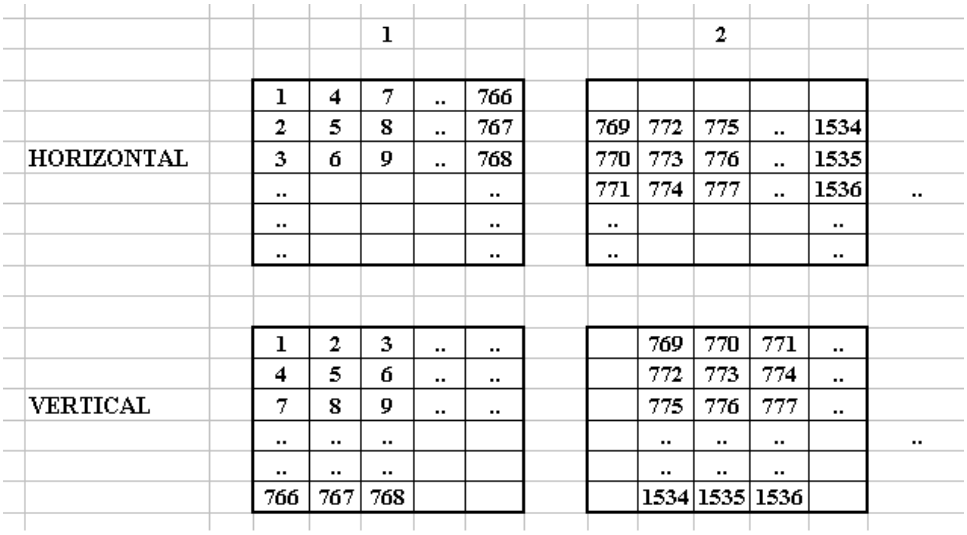


Figure 5: Sequence of Input Memory Address Access

The figure 5 shows the sequence in which the input memory pixels are accessed by the Input Controller. During the horizontal movement, the 3 pixels are accessed in every column, with the direction of movement of the access being the same as the direction of movement of the filter, i.e. from left to right. Once a horizontal end of the image is reached, the access is started from the next row, with again three pixels in every column and the direction of movement being horizontal.

When the horizontal movement is completed, the vertical movement takes place. During this, three pixels are accessed in every row of the image, and the movement is vertical, from top to bottom. On reaching a vertical end of the image, the reading starts at the top of the next column, again with 3 pixels in each row and movement from top to bottom.

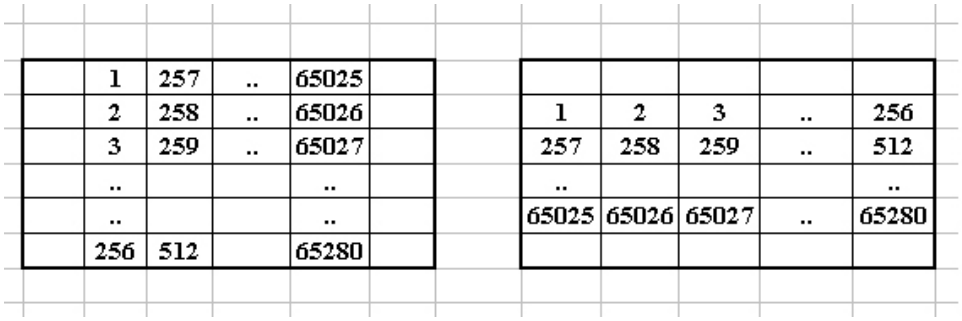


Figure 6: Sequence of Output Memory Address Access

The memory access for the Output Controller is simple as compared to the memory access by the Input Controller. The right part of the figure 6 shows the access for the horizontal movement. The top most and the bottom most rows are not written. Once a row is completed the next row pixels are read and written.

For the vertical movement, the first and the last columns are not written. The read and write sequence starts from the second column, and ends when the second last column is finished.

4 Finite State Machines

The following subsections explain the finite state machines describing the Input and the Output Controllers.

4.1 Input Controller

The Input Controller has the following 16 states.

1. State *init*. This is the initialization state of the controller. The next state is *init_in_memory_1*.

2. State *init_in_memory_1*. This state performs the initialization of the input memory. The Controller puts the *can_write* signal to *1*, so that the byte read from the *.hex* file is written to the *Data_In* of the input memory. The address to which the byte should be written is also given by the Controller, through the signal *address*. The next state is *init_in_memory_2*.

3. State *init_in_memory_2*. This state does the same thing as the previous state, and it increments a counter. Once the value of the counter is greater than 65536, which means that the entire memory has been written, the state changes to *h_read_1*.

4. State *h_read_1*. This state performs the read operation from the input memory for one pixel. The signal *can_write* is set to *1*. The next state is *h_read_2*.

5. State *h_read_2*. This state performs the read operation from the input memory for one pixel. The next state is *h_read_3*. The signal *disable_cache* is set to *0* here, since from now we want the cache to start loading the values from the input memory.

6. State *h_read_3*. This state performs the read operation from the input memory for one pixel. The next state is *h_cache*.

7. State *h_cache*. This state is used as a delay. The signal *disable_cache* should be delayed by one clock cycle as compared to the signal *can_read* since the byte from the memory comes one clock cycle after the *can_read* is active. The signal *can_read* is set to *0* here. The next state is *h_wait*.

8. State *h_wait*. This state is used as a wait state, during which the Output FSM is active. Also, the *disable_cache* signal is set to *1* here, so that no more values from the memory are read into the cache, until the Input Controller gets active again. The next state is *h_temp*.

9. State *h_temp*. This state is also the wait state, and the finite state machine keeps shuttling between this state and the previous state, for 27 clock cycles. Once the Output Controller writes the new pixels to the output memory, the wait time is over, and the next state is set to *v_read_1* and the vertical reading for the memory is started.

   The states for vertical movement are kept separate from the states for the horizontal movement. This is because the order in which the addresses are generated for the *address* signal, are different in both the movements (as also shown in figure 5). The purpose of the following states is the same though, with only the address values being different, so we skip the explanation.

10. State *v_read_1*.

11. State *v_read_2*.

12. State *v_read_3*.

13. State *v_cache*.

14. State *v_wait*.

15. State *v_temp*.

16. State *exit_in*. This is the exit state of the finite state machine.

4.2 Output Controller

The Output Controller has the following 18 states.

1. State *init*. This is the initialization state of the controller. The next state is *init_out_memory_1*.

2. State *init_out_memory_1*. This state initializes the output memory to *0*. The controller puts the *can_write* signal to *1* and the *write_address* is incremented every time in the state. The next state is *init_out_memory_2*.

   Important to note here, that the output memory is initialized by zeros, since the *Data_In_2* signal coming from the output memory is set to zero, using the *give_zeros* signal of the output memory. Also the multiplexer in this state gives zero since the select is forced to '11' by the controller. Hence the *Data_Out* of the processor is always zero in this state.

3. State *init_out_memory_2*. This state performs the same function as the previous state. A counter is maintained which if greater than 65536 indicates that the memory is initialized. Then the next state is *h_init_1*

4. State *h_init_1*. This state is created in order to wait for the cache shift register to get the pixels from the input memory. Actually, when the Input Controller finishes reading a line in the memory, during any of the horizontal or vertical movements, the Output Controller must wait for the time till the cache is filled with the new 9 pixels. The next state is *h_init_2*.

5. State *h_init_2*. This state performs the same function, and waits till the 9 pixels are filled in the cache shift register. This takes 27 clock cycles, since the Input Controller also remains idle in between. Once this is done, the next state is set to *h_read_1*.

6. State *h_read_1*. This state puts the *can_read* signal to *1*. Also, the corresponding *read_address* is set. The next state is *h_read_write*.

7. State *h_read_write*. In this state, the Controller remains idle, so that data is recieved from the output memory in the next clock cycle. The next state is *h_write_1*.

8. State *h_write_1*. This state forces the adder to be selected, by changing the *sel* signal. The old pixel from the output memory and the new pixel from the adder are added. The signal *can_write* is set to *1* and the *write_address* is set to the same value as the *read_address* in the previous state. If the end of the row or column is reached in the memory, the next state is *h_init_1*. Else the next state is *h_wait_1*. Also, if the end of the image is identified, then the vertical movement begins, and in that case the next state is set to *v_init_1*

9. State *h_wait_1*. In this state, the Controller waits for the Input Controller to read 3 new pixels from the input memory. The next state is *h_wait_2*.

10. State *h_wait_2*. This state performs the same function as the previous state. If 3 clock cycles are over, i.e. if the Input Controller has read 3 new pixels, then the Controller gets active again, and the next state is set to *h_read_1*.

    Again, we avoid giving explanation for the states during the vertical movement, since they all perform the same function as the states occuring during the horizontal movement.

11. State *v_init_1*.

12. State *v_init_2*.

13. State *v_read_1*.

14. State *v_read_write*.

15. State *v_write_1*.

16. State *v_wait_1*.

17. State *v_wait_2*.

18. State *exit_in*. This is the exit state of the finite state machine.

5 Synthesis

We synthesized the design using four different clock periods, namely 7ns, 5ns, 3ns and 2ns, and let Design Vision try to optimize the design for speed to get the fastest possible design. Turns out 2ns is the minimum clock period for our design, Design Vision was not able to synthesize a faster design even when we tried. To get meaningful power reports we simulated switching activity with the VSS Simulator and the activity was passed on to Design Vision. On top of power reports we also obtained area and timing reports from the design on all four clock periods. The actual reports can be seen in the appendix, but a summary of the results can be seen in table 1. The power breakdown for the designs on all four clock periods can be seen in table 2. In the breakdown $MULT_{conv}$ and ADD_{conv} refer to the power dissipation in all multipliers and adders involved in the convolution combined. For more detailed breakdown please refer to the appendix.

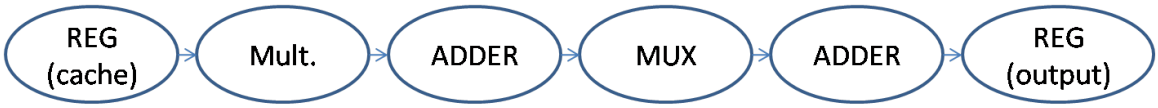
Table 1: Summary of Design Vision reports

$\mathbf{T}_C$ [ns]	$\mathbf{P}_{stat}$ [mW]	$\mathbf{P}_{dyn}$ [mW]	$\mathbf{P}_{tot}$ [mW]	$\mathbf{A}_{comb}$ [μm^2]	$\mathbf{A}_{tot}$ [μm^2]	$\mathbf{T}_{cp}$ [ns]
7	0.16	1.77	1.93	44067	53079	4.7
5	0.16	1.88	2.04	44067	53079	4.7
3	0.11	2.17	2.28	49595	58611	2.9
2	0.19	2.59	2.78	58668	67700	1.9

Table 2: Power breakdown, total power in mW

$\mathbf{T}_C$	$\mathbf{FSM}_{in}$	$\mathbf{FSM}_{out}$	$\mathbf{REG}_{filter}$	$\mathbf{REG}_{cache}$	$\mathbf{MULT}_{conv}$	$\mathbf{ADD}_{conv}$	**MUX**	$\mathbf{ADD}_{out}$	$\mathbf{REG}_{out}$
7	0.207	0.242	0.396	0.547	0.363	0.036	0.035	0.050	0.053
5	0.256	0.300	0.396	0.549	0.365	0.036	0.036	0.056	0.052
3	0.370	0.436	0.401	0.548	0.224	0.110	0.040	0.126	0.052
2	0.512	0.606	0.395	0.557	0.371	0.103	0.044	0.143	0.052

The critical path of the design (see figure 2) is the same for all four clock periods and goes through the cache register, the convolution multipliers, the convolution adders, the multiplexer, the adder and the output register (added for the synthesizer's sake, to constrain the path - doesn't contribute to the delay in any of the timing reports). The path is illustrated in figure 7 along with the delay information.



T_C	REG	Mult.	ADDER	MUX	ADDER	REG	T_{cp}
7	0.11	2.71	1.00	0.20	0.71	-	4.73ns
5	0.11	2.71	1.00	0.20	0.71	-	4.73ns
3	0.16	1.55	0.83	0.12	0.26	-	2.92ns
2	0.12	1.00	0.46	0.11	0.23	-	1.92ns

Figure 7: Critical path of the design

6 Results

The original image which is convoluted in the simulation is shown in figure 8. The convolution of this image takes a total time of 3252224 ns, which is broken down as (with a clock cycle of 2 ns):

1. Memory Initialization: 131072 ns (256 * 256 * 2 ns).
2. Horizontal Movement: 1560576 ns (256 * 254 * 12 * 2 ns), where 12 is the number of cycles taken for one horizontal movement of the filter.
3. Vertical Movement: 1560576 ns (256 * 254 * 12 * 2 ns), where 12 is the number of cycles taken for one vertical movement of the filter.

The left image in the figure 9 is computed using the filter mask of '000010000' and the right image in figure 9 is computed using the filter mask of '010101010'.

Figure 8: Original Image

Figure 9: Convolution Results

Provided below is a summary of the results obtained from the Synthesis of the design.

Table 3: Summary

T_C [ns]	**Critical Path** [ns]	**N. cycles** (256 x 256)	**E_{pc}** [mW/MHz]	**AREA** [μm^2]
7	4.7	3121152	0.01351	53079
5	4.7	3121152	0.01020	53079
3	2.9	3121152	0.00684	58611
2	1.9	3121152	0.00556	67700

7 Limitations and Extensions

We have designed and implemented the architecture for an image filtering processor which uses a 3x3 filter to perform a convolution on the image of size 256x256 pixels. Though the results of the convoluted image look promising, as shown in section 6, we are aware of some of the limitations of the work. Given more time, we would have liked to add the following missing aspects into the project.

1. We have just been able to implement the 3x3 filter for the convolution. Though, it was proposed that we would implement the higher dimension filters as well, including 5x5, 7x7 and 9x9, we were not able to do so, due to the initial problems we faced in the implementation of the 3x3 filter itself. We believe the results obtained in the section 6, in the form of the convoluted image could be better if the dimension of the filter is higher. In those cases, the blur effect on the image would be clearly evident, as compared to the case of the 3x3 filter. We would like to briefly mention how the design of the processor would be modified if we wish to convolute the image using higher dimension filters. If we consider the dimension of the filter as nxn, then we have the following:

    - Number of Adders = n^2

- Number of Multipliers = n
- Size of the Cache Shift Register = n^2
- Multiplexer would have n inputs and 1 output.
- Select signal from the Output Controller would be 3 bits for 5x5 and 7x7, and 4 bits for 9x9 filter.

In addition the synchronization of the Input and the Output Controllers would change due to the number of clock cycles required to get n new pixels from the memory and compute the output for n pixels at a time. This means the cases described in section 3.2 would now be the following for the $n * n$ filter:

- Number of cycles taken for the Input Controller to read pixels due to the filter movement (horizontal or vertical): n.
- Number of cycles taken for the Output Controller to read and write n new pixels to the output memory: $3 * n$, since there are three states for every pixel, namely, read, idle and write.
- Number of cycles for which the Output Controller waits in the case when the horizontal movement shifts to the next row or the vertical movement shifts to the next column: $[(n + 3 * n) + (n + 3 * n) + .. \ (n-1)times \ .. + (n + 3 * n)] + n$, which is $4 * n^2 - 3 * n$.

2. It is assumed that the filter is symmetric along the two dimensional x and y axes. We need this since the indices of the filter which need to be multiplied with the image pixels would change in horizontal and vertical movements. For simplicity therefore, we have made this assumption. The solution to this problem is quite simple though. We just need to have separate caches in the processor which hold the filter values in a different order. For horizontal movement we would use one cache, while the other one would be used for the vertical movement.

3. The mechanism which we have designed for the accessing the memory is ofcourse not the best way. Since we began the implementation with the sequence explained in the section 3.3, we did not change it later. Though, we realized that this is not an efficient way, since it consumes a high number of clock cycles in order to run through the entire image of 256x256 pixels.

8 VHDL Implementation Files

Following are the VHDL files which are core to the project. In addition we have a lot of test benches, which we have created to test the individual components. These extra files (along with the core files) are provided in the ZIP archive.

Listing 1: Memory.vhd

```
-- a simple 256*256 pixel (256*256*8 bits) Memory module
-- arranged in a linear fashion.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity MEMORY is
port(
   Clock:              in std_logic;
        Enable:        in std_logic;
        Give_Zeros: in std_logic;
        Read:          in std_logic;
        Write:         in std_logic;
        Read_Addr:     in std_logic_vector(15 downto 0);
        Write_Addr:    in std_logic_vector(15 downto 0);
        Data_in:       in std_logic_vector(7 downto 0);
        Data_out:      out std_logic_vector(7 downto 0)
);
end MEMORY;

--------------------------------------------------------------------------------

architecture BEHLMEMORY of MEMORY is

type memory_type is array (0 to 65536) of std_logic_vector(7 downto 0);
signal tmp_memory: memory_type;

begin
    process(Clock)
    begin
        if (Clock'EVENT and Clock='1' and enable='1') then
            if (Give_Zeros='1') then
                Data_out <= (Data_out'range => '0');
            elsif (Read='1') then
                Data_out <= tmp_memory(conv_integer(Read_Addr));
            end if;
            if (Write='1') then
                tmp_memory(conv_integer(Write_Addr)) <= Data_in;
            end if;
        end if;
    end process;

end BEHLMEMORY;
```

Listing 2: FSM_in.vhd

```
library ieee ;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity FSM_in_3 is

port (
   clock:               in std_logic;
        reset:          in std_logic;
        address:                out std_logic_vector(15 downto 0);
        can_read:   out std_logic;
        can_write:  out std_logic;
        disable_cache: out std_logic
);
end FSM_in_3;

architecture BEH_FSM_in_3 of FSM_in_3 is

    type state_type is (init, init_in_memory_1, init_in_memory_2, h_read_1, h_read_2,
                        h_read_3, h_cache, v_cache, h_wait, h_temp,
                         v_read_1, v_read_2, v_read_3, v_wait, v_temp, exit_in);
    signal next_state, current_state: state_type;

begin
state_reg: process(clock, reset)
begin
                  if (reset='0') then
                current_state <= init;
                  elsif (clock'event and clock='1') then
                     current_state <= next_state;
                  end if;
end process;

comb_logic: process(current_state)

    variable addr_h: INTEGER;
    variable addr_v: INTEGER;
    variable x: INTEGER;
    variable y: INTEGER;
    variable temp_address: INTEGER;
    variable counter: INTEGER;

begin

                case current_state is

       when init =>

           x := 1;
           y := 1;
           counter := 1;
           disable_cache <= '1';
           next_state <= init_in_memory_1;
           can_read <= '0';
           can_write <= '0';
           address <= (others => '0');

       when init_in_memory_1 =>
```

```
      next_state <= init_in_memory_2;
      can_write <= '1';
      address <= conv_std_logic_vector(counter,16);
      counter := counter + 1;

  when init_in_memory_2 =>

     can_write <= '1';
     address <= conv_std_logic_vector(counter,16);

      if(counter = 65536) then
         next_state <= h_read_1;
         counter := 1;
         can_write <= '0';
         address <= (others => '0');
      else
         next_state <= init_in_memory_1;
         counter := counter + 1;
      end if;



       when h_read_1 =>

       next_state <= h_read_2;
       can_read <= '1';
            addr_h := x;
       address <= conv_std_logic_vector(addr_h,16);
       disable_cache <= '1';

       when h_read_2 =>

            next_state <= h_read_3;
       can_read <= '1';
            addr_h := addr_h + 256;
       address <= conv_std_logic_vector(addr_h,16);
       disable_cache <= '0';

       when h_read_3 =>

            next_state <= h_cache;
       can_read <= '1';
            addr_h := addr_h + 256;
            x := x + 1;
       address <= conv_std_logic_vector(addr_h,16);
       disable_cache <= '0';

when h_cache =>
        disable_cache <= '0';
        can_read <= '0';

        next_state <= h_wait;

       when h_wait =>

       counter := counter + 1;
```

```
        can_read <= '0';
        address <= (others => '0');
        next_state <= h_temp;
        disable_cache <= '1';

    when h_temp =>

        counter := counter + 1;
        can_read <= '0';
        address <= (others => '0');
        disable_cache <= '1';

        if(counter > 8) then

           counter := 1;
           if(x > 65024) then
               x := 1;
               next_state <= v_read_1;
           else
               next_state <= h_read_1;
           end if;

        else
           next_state <= h_wait;
        end if;


        when v_read_1 =>

        disable_cache <= '1';
        next_state <= v_read_2;
        can_read <= '1';
             addr_v := x;
        address <= conv_std_logic_vector(addr_v,16);

        when v_read_2 =>

             next_state <= v_read_3;
        can_read <= '1';
             addr_v := addr_v + 1;
        address <= conv_std_logic_vector(addr_v,16);
        disable_cache <= '0';

        when v_read_3 =>

             next_state <= v_cache;
        can_read <= '1';
             addr_v := addr_v + 1;
             x := x + 256;
        address <= conv_std_logic_vector(addr_v,16);
        disable_cache <= '0';

    when v_cache =>
         disable_cache <= '0';
         can_read <= '0';
```

```
            next_state <= v_wait;

        when v_wait =>

            counter := counter + 1;
            next_state <= v_temp;

            can_read <= '0';
            address <= (others => '0');

            disable_cache <= '1';

           when v_temp =>

            counter := counter + 1;
            if(counter > 8) then

               counter := 1;
               if(x > 65536) then
                     y := y + 1;
                     x := y;
               end if;

               if(y = 255) then
                     next_state <= exit_in;
               else
                     next_state <= v_read_1;
               end if;
            else
               next_state <= v_wait;
            end if;

            can_read <= '0';
            address <= (others => '0');

        when exit_in =>

            can_read <= '0';
            address <= (others => '0');
            next_state <= exit_in;
            disable_cache <= '1';

           when others =>
                disable_cache <= '1';
                              next_state <= init;
            can_read <= '0';
            address <= (others => '0');

         end case;

end process;

end BEH_FSM_in_3;

configuration CFG_FSM_in_3_BEHAVIORAL of FSM_in_3 is
          for BEH_FSM_in_3
          end for;
```

```
end CFG_FSM_in_3_BEHAVIORAL;
```

Listing 3: FSM_out.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity FSM_out_3 is

port (
   clock:               in std_logic;
        reset:          in std_logic;
        read_address:           out std_logic_vector(15 downto 0);
        write_address:          out std_logic_vector(15 downto 0);
        can_read:   out std_logic;
        can_write:  out std_logic;
        sel: out std_logic_vector(1 downto 0)
);
end FSM_out_3;

architecture BEH_FSM_out_3 of FSM_out_3 is

    type state_type is (init, init_out_memory_1, init_out_memory_2, h_init_1, h_init_2,
    h_read_1, h_read_write, h_read_write_2, h_write_1, h_wait_1, h_wait_2,
     v_init_1, v_init_2, v_read_1, v_read_write, v_write_1, v_wait_1, v_wait_2, exit_in);
    signal next_state, current_state: state_type;

begin
state_reg: process(clock, reset)
begin
                  if (reset='0') then
                current_state <= init;
                  elsif (clock'event and clock='1') then
                     current_state <= next_state;
                  end if;
end process;

comb_logic: process(current_state)

    variable addr_h: INTEGER;
    variable addr_v: INTEGER;
    variable x: INTEGER;
    variable y: INTEGER;
    variable temp_address: INTEGER;
    variable counter: INTEGER;
    variable rwcount: INTEGER;
    variable sel_num: INTEGER;

begin

                case current_state is

       when init =>
           x := 257;
           counter := 1;
           rwcount := 1;
           sel_num := 0;
```

```
        next_state <= init_out_memory_1;

        can_read <= '0';
        can_write <= '0';
        read_address <= (others => '0');
        write_address <= (others => '0');

    when init_out_memory_1 =>

        can_write <= '1';
        sel_num := 3;
        sel <= conv_std_logic_vector(sel_num,2);
        write_address <= conv_std_logic_vector(counter,16);
        next_state <= init_out_memory_2;

        counter := counter + 1;

    when init_out_memory_2 =>

        can_write <= '1';
        write_address <= conv_std_logic_vector(counter,16);

        if(counter = 65536) then
           next_state <= h_init_1;
           counter := 1;
           can_write <= '0';
           write_address <= (others => '0');
           sel_num := 0;
           sel <= conv_std_logic_vector(sel_num,2);
        else
           next_state <= init_out_memory_1;
           counter := counter + 1;
           sel_num := 3;
           sel <= conv_std_logic_vector(sel_num,2);
        end if;


    when h_init_1 =>

        if(counter = 27) then

            counter := 1;
            next_state <= h_read_1;
            addr_h := x;

        else
            next_state <= h_init_2;
        end if;

        can_read <= '0';
        can_write <= '0';
        read_address <= (others => '0');
        write_address <= (others => '0');
        counter := counter + 1;

    when h_init_2 =>
```

```
        counter := counter + 1;
        next_state <= h_init_1;
        can_read <= '0';
        can_write <= '0';
        read_address <= (others => '0');
        write_address <= (others => '0');

        when h_read_1 =>

        next_state <= h_read_write;

        can_read <= '1';
        can_write <= '0';
        read_address <= conv_std_logic_vector(addr_h,16);
        write_address <= (others => '0');
        sel <= conv_std_logic_vector(sel_num,2);
        sel_num := sel_num + 1;

    when h_read_write =>

        next_state <= h_write_1;

        can_read <= '0';
        can_write <= '0';
        read_address <= (others => '0');
        write_address <= (others => '0');

        when h_write_1 =>

        rwcount := rwcount + 1;
        if(rwcount > 3) then
            if(x > 65277) then
               x := 2;
               next_state <= v_init_1;
               counter := 1;
               rwcount := 1;
               sel_num := 0;

            elsif((addr_h mod 256) = 0) then
                x := addr_h + 1;
                next_state <= h_init_1;
            else
                 next_state <= h_wait_1;
            end if;

           rwcount := 1;
           sel_num := 0;
        else
           next_state <= h_read_1;
        end if;

        can_read <= '0';
        can_write <= '1';
        read_address <= (others => '0');
        write_address <= conv_std_logic_vector(addr_h,16);
```

```
            addr_h := addr_h + 1 ;

        -- wait for 3 clock cycles - till FSM1 reads the next 3 pixels
            when h_wait_1 =>

            counter := counter + 1;
            if(counter > 3) then

               counter := 1;


                        x := x + 1;
                        addr_h := x;
                        next_state <= h_read_1;



            else
               next_state <= h_wait_2;
            end if;

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

        when h_wait_2 =>

           counter := counter + 1;
           next_state <= h_wait_1;

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

-------------------------------------------
--- States defining the Vertical Movement ---
-------------------------------------------

        when v_init_1 =>

           counter := counter + 1;
             if(counter > 27) then

               counter := 1;
               next_state <= v_read_1;
               addr_v := x;

           else
               next_state <= v_init_2;
           end if;

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');
```

```
when v_init_2 =>

    can_read <= '0';
    can_write <= '0';
    read_address <= (others => '0');
    write_address <= (others => '0');

    counter := counter + 1;
    next_state <= v_init_1;

     when v_read_1 =>

     next_state <= v_read_write;

     can_read <= '1';
     can_write <= '0';
     read_address <= conv_std_logic_vector(addr_v,16);
     write_address <= (others => '0');
     sel <= conv_std_logic_vector(sel_num,2);
     sel_num := sel_num + 1;

when v_read_write =>

    next_state <= v_write_1;

     can_read <= '0';
     can_write <= '0';
     read_address <= (others => '0');
     write_address <= (others => '0');


     when v_write_1 =>

     rwcount := rwcount + 1;
     if(rwcount > 3) then

        rwcount := 1;
        sel_num := 0;

        if(addr_v > 65280) then
            x := x + 1;
            if(x = 256) then
               next_state <= exit_in;
            else
               addr_v := x;
               next_state <= v_init_1;
            end if;
        else
            next_state <= v_wait_1;
        end if;

     else
        next_state <= v_read_1;
     end if;

     can_read <= '0';
```

```
            can_write <= '1';
            read_address <= (others => '0');
            write_address <= conv_std_logic_vector(addr_v,16);

            addr_v := addr_v + 256;

        -- wait for 3 clock cycles - till FSM1 reads the next 3 pixels
            when v_wait_1 =>

            counter := counter + 1;
            if(counter > 3) then

               counter := 1;
               next_state <= v_read_1;
            else

               next_state <= v_wait_2;

            end if;

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

        when v_wait_2 =>

           counter := counter + 1;
           next_state <= v_wait_1;
           addr_v := addr_v - 512;
           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

        when exit_in =>

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

           next_state <= exit_in;

            when others =>

           can_read <= '0';
           can_write <= '0';
           read_address <= (others => '0');
           write_address <= (others => '0');

                        next_state <= init;

         end case;

end process;
```

```
end architecture BEH_FSM_out_3;
```

Listing 4: REG.vhd

```
library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_signed.all;
   use IEEE.std_logic_arith.all;

   entity REG is
        port(
                D : in std_logic_vector(7 downto 0);
                Clock, Reset : in std_logic;
                Q : out std_logic_vector(7 downto 0));
end entity REG;

architecture BEH_REG of REG is
   begin
   p0: process (Clock, Reset) is
      begin
      if (Reset = '0') then
         Q <= (others => '0');
      elsif ((CLOCK = '1') AND (CLOCK'EVENT)) then
         Q <= D;
      end if;
   end process p0;
end BEH_REG;
```

Listing 5: SHIFTREG.vhd

```
library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_signed.all;
   use IEEE.std_logic_arith.all;

entity SHIFTREG is
      Port (   CLOCK : In    std_logic;
               RESET : In    std_logic;
             disable : In     std_logic;
                 QK : In    std_logic_vector (7 downto 0);
                  Q : InOut   std_logic_vector (71 downto 0) );
end SHIFTREG;

architecture BEH_SHIFTREG of SHIFTREG is

   begin
    process(RESET,CLOCK)
     variable i,j,k,l : integer;
     begin

       if ( RESET = '0' ) then
          for i in 0 to 71 loop
              q(i) <= '0';
          end loop;
       elsif ((CLOCK = '1') AND (CLOCK'EVENT)) then
```

```
          if(disable='0') then
                        for i in 71 downto 8 loop
                              q(i-8) <= q(i);
                        end loop;
                q(71 downto 64) <= qk;

                else
                     for i in 71 downto 0 loop
                              q(i) <= q(i);
                     end loop;

                end if;
             end if;

    end process;

end BEH_SHIFTREG;
```

Listing 6: CRA_15.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity CRA_15 is
port(           A : In std_logic_vector (15 downto 0);
                B : In std_logic_vector (15 downto 0);
                Cin : In std_logic;
                Cout : Out std_logic;
                Y : Out std_logic_vector (15 downto 0) );
end CRA_15;

architecture RTL of CRA_15 is

begin

       process(A,B,Cin)

            variable tempC   : std_logic_vector (16 downto 0);
            variable P   : std_logic_vector (15 downto 0);
            variable G   : std_logic_vector (15 downto 0);
            variable Yaux   : std_logic_vector (15 downto 0);


            begin

                tempC(0):= Cin;

                   for i in 0 to 15 loop

                      P(i):= A(i) XOR B(i);
                      G(i):= A(i) AND B(i);

                      Yaux(i):= P(i) xor tempC(i);
                      tempC(i+1):=G(i) OR (tempC(i) AND P(i));
                end loop;
```

```
                if (tempC(16)='1') then
                    Yaux(15 downto 0):= "1111111111111111";
                end if;
                for i in 0 to 7 loop
                  if (Yaux(i+8)='1') then
                      Yaux(15 downto 0):="1111111111111111";
                  end if;
                end loop;
                Y(15 downto 0)<= Yaux(15 downto 0);
                Cout<=tempC(16);

            end process;
    end RTL;


    configuration CFG_CRA_15_BEHAVIORAL of CRA_15 is
         for RTL
         end for;
end CFG_CRA_15_BEHAVIORAL;
```

Listing 7: CSA_15bit.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity csa15bit is
        Port (  A : In std_logic_vector (15 downto 0);
                B : In std_logic_vector (15 downto 0);
                C : In std_logic_vector (15 downto 0);
                Cin : In std_logic;
                Cout : Out std_logic;
                Z : Out std_logic_vector (15 downto 0);
                Y : Out std_logic_vector (15 downto 0) );
end csa15bit;

architecture BEHAVIORAL of csa15bit is
   signal aux : std_logic;


 begin
process(A, B, C, Cin)
        variable p : std_logic_vector (15 downto 0) ;
        variable g : std_logic_vector (15 downto 0) ;
        variable Z_aux : std_logic_vector (15 downto 0) ;
        variable Y_aux : std_logic_vector (15 downto 0) ;
        variable i : integer;
begin

for i in 0 to 15 loop
        p(i) := A(i) XOR B(i) ;
        g(i) := A(i) AND B(i) ;
end loop;

-- CARRY ------------------------------------
Y_aux(0) := Cin;
```

```
for i in 0 to 15-1 loop
        Y_aux(i+1) := g(i) OR (c(i) AND p(i));
end loop;
Y(15 downto 0)<=Y_aux(15 downto 0);
aux <= g(15) OR (c(15) AND p(15));
Cout<= aux;

-- SUM ------------------------------------
for i in 0 to 15 loop
        Z_aux(i) := p(i) XOR c(i);
end loop;


Z(15 downto 0)<= Z_aux(15 downto 0);

end process;
end BEHAVIORAL;

configuration CFG_csa15bit_BEHAVIORAL of csa15bit is
         for BEHAVIORAL
         end for;
end CFG_csa15bit_BEHAVIORAL;
```

Listing 8: parcial.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity parcial is
   port(   M1: in std_logic_vector(7 downto 0);
           M2: in std_logic_vector(7 downto 0);
           O1: out std_logic_vector(15 downto 0);
           O2: out std_logic_vector(15 downto 0);
           O3: out std_logic_vector(15 downto 0);
           O4: out std_logic_vector(15 downto 0);
           O5: out std_logic_vector(15 downto 0);
           O6: out std_logic_vector(15 downto 0);
           O7: out std_logic_vector(15 downto 0);
           O8: out std_logic_vector(15 downto 0) );
end parcial;

architecture BEHAVIORAL of parcial is
    signal aux: std_logic;

    begin

    process(M1,M2)
        variable i: integer;
        variable Pa1: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa2: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa3: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa4: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa5: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa6: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa7: std_logic_vector(15 downto 0):="0000000000000000";
        variable Pa8: std_logic_vector(15 downto 0):="0000000000000000";
```

```
    begin


        for i in 0 to 7 loop

            Pa1(i):= M1(0) AND M2(i);
            Pa2(i+1):=M1(1) AND M2(i);
            Pa3(i+2):=M1(2) AND M2(i);
            Pa4(i+3):=M1(3) AND M2(i);
            Pa5(i+4):=M1(4) AND M2(i);
            Pa6(i+5):=M1(5) AND M2(i);
            Pa7(i+6):=M1(6) AND M2(i);
            Pa8(i+7):=M1(7) AND M2(i);

        end loop;

        O1(15 downto 0)<= Pa1(15 downto 0);
        O2(15 downto 0)<= Pa2(15 downto 0);
        O3(15 downto 0)<= Pa3(15 downto 0);
        O4(15 downto 0)<= Pa4(15 downto 0);
        O5(15 downto 0)<= Pa5(15 downto 0);
        O6(15 downto 0)<= Pa6(15 downto 0);
        O7(15 downto 0)<= Pa7(15 downto 0);
        O8(15 downto 0)<= Pa8(15 downto 0);



end process;

end BEHAVIORAL;

configuration CFG_parcial_BEHAVIORAL of parcial is
         for BEHAVIORAL
         end for;
end CFG_parcial_BEHAVIORAL;
```

Listing 9: Multiplier.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- two 4-bit inputs and one 8-bit outputs
entity multiplier is
port(   num1, num2:     in std_logic_vector(7 downto 0);
        product:        out std_logic_vector(7 downto 0)
);
end multiplier;

architecture SCHEMATIC of multiplier is

   signal P0: std_logic_vector(15 downto 0);
   signal P1: std_logic_vector(15 downto 0);
   signal P2: std_logic_vector(15 downto 0);
   signal P3: std_logic_vector(15 downto 0);
   signal P4: std_logic_vector(15 downto 0);
```

```
signal P5: std_logic_vector(15 downto 0);
signal P6: std_logic_vector(15 downto 0);
signal P7: std_logic_vector(15 downto 0);
signal A1: std_logic_vector(15 downto 0);
signal A2: std_logic_vector(15 downto 0);
signal A3: std_logic_vector(15 downto 0);
signal A4: std_logic_vector(15 downto 0);
signal A5: std_logic_vector(15 downto 0);
signal A6: std_logic_vector(15 downto 0);
signal A7: std_logic_vector(15 downto 0);
signal A8: std_logic_vector(15 downto 0);
signal A9: std_logic_vector(15 downto 0);
signal A0: std_logic_vector(15 downto 0);
signal B1: std_logic_vector(15 downto 0);
signal B2: std_logic_vector(15 downto 0);
signal B3: std_logic_vector(15 downto 0);
signal B4: std_logic_vector(15 downto 0);
signal B5: std_logic_vector(15 downto 0);
signal carry_ex1,carry_ex2,carry_ex3,carry_ex4,
carry_ex5,carry_ex6,carry_ex7: std_logic;
signal ca: std_logic;


component csa15bit
     port ( A : In std_logic_vector (15 downto 0);
            B : In std_logic_vector (15 downto 0);
            C : In std_logic_vector (15 downto 0);
            Cin : In std_logic;
            Cout : Out std_logic;
            Z : Out std_logic_vector (15 downto 0);
            Y : Out std_logic_vector (15 downto 0) );
end component;

component ADDER_M
    port ( A : In std_logic_vector (15 downto 0);
           B : In std_logic_vector (15 downto 0);
           C: In std_logic_vector(15 downto 0);
            Z: In std_logic_vector(15 downto 0));
        end component;
component parcial
     port ( M1: in std_logic_vector(7 downto 0);
            M2: in std_logic_vector(7 downto 0);
            O1: out std_logic_vector(15 downto 0);
            O2: out std_logic_vector(15 downto 0);
            O3: out std_logic_vector(15 downto 0);
            O4: out std_logic_vector(15 downto 0);
            O5: out std_logic_vector(15 downto 0);
            O6: out std_logic_vector(15 downto 0);
            O7: out std_logic_vector(15 downto 0);
            O8: out std_logic_vector(15 downto 0) );
end component;
component CRA_15 is
     port(          A : In std_logic_vector (15 downto 0);
                    B : In std_logic_vector (15 downto 0);
                    Cin : In std_logic;
                    Cout : Out std_logic;
                    Y : Out std_logic_vector (15 downto 0) );
```

```
 end component;



begin

  ca<='0';

I_PAR : parcial
   Port Map(
             M1=>num1,
             M2=>num2 ,
             O1(15 downto 0)=>P0(15 downto 0),
             O2(15 downto 0)=>P1(15 downto 0),
             O3(15 downto 0)=>P2(15 downto 0),
             O4(15 downto 0)=>P3(15 downto 0),
             O5(15 downto 0)=>P4(15 downto 0),
             O6(15 downto 0)=>P5(15 downto 0),
             O7(15 downto 0)=>P6(15 downto 0),
             O8(15 downto 0)=>P7(15 downto 0)
             );


I_CSA1 : csa15bit
   Port Map (
            A(15 downto 0)=>P0(15 downto 0),
            B(15 downto 0)=>P1(15 downto 0),
            C(15 downto 0)=>P2(15 downto 0),
            Cin=>ca,
            Cout=>carry_ex1,
            Z(15 downto 0)=>A0(15 downto 0),
            Y(15 downto 0)=>A1(15 downto 0)
            );

I_CSA2 : csa15bit
   Port Map (
            A(15 downto 0)=>P3(15 downto 0),
            B(15 downto 0)=>P4(15 downto 0),
            C(15 downto 0)=>P5(15 downto 0),
            Cin=>ca,
            Cout=>carry_ex2,
            Z(15 downto 0)=>A2(15 downto 0),
            Y(15 downto 0)=>A3(15 downto 0)
            );
I_CSA3 : csa15bit
   Port Map (
            A(15 downto 0)=>A0(15 downto 0),
            B(15 downto 0)=>A1(15 downto 0),
            C(15 downto 0)=>A2(15 downto 0),
            Cin=>carry_ex1,
            Cout=>carry_ex3,
            Z(15 downto 0)=>A4(15 downto 0),
            Y(15 downto 0)=>A5(15 downto 0)
            );
I_CSA4 : csa15bit
```

```
    Port Map (
              A(15 downto 0)=>A3(15 downto 0),
              B(15 downto 0)=>P6(15 downto 0),
              C(15 downto 0)=>P7(15 downto 0),
              Cin=>carry_ex2,
              Cout=>carry_ex4,
              Z(15 downto 0)=>A6(15 downto 0),
              Y(15 downto 0)=>A7(15 downto 0)
              );
  I_CSA5 : csa15bit
    Port Map (
              A(15 downto 0)=>A4(15 downto 0),
              B(15 downto 0)=>A5(15 downto 0),
              C(15 downto 0)=>A6(15 downto 0),
              Cin=>carry_ex3,
              Cout=>carry_ex5,
              Z(15 downto 0)=>A8(15 downto 0),
              Y(15 downto 0)=>A9(15 downto 0)
              );
   I_CSA6 : csa15bit
    Port Map (
              A(15 downto 0)=>A7(15 downto 0),
              B(15 downto 0)=>A8(15 downto 0),
              C(15 downto 0)=>A9(15 downto 0),
              Cin=>carry_ex4,
              Cout=>carry_ex6,
              Z(15 downto 0)=>B1(15 downto 0),
              Y(15 downto 0)=>B2(15 downto 0)
              );
   I_CRA : CRA_15
    Port Map (
              A(15 downto 0)=>B1(15 downto 0),
              B(15 downto 0)=>B2(15 downto 0),
              Cin=>carry_ex5,
              Cout=>carry_ex7,
              Y(15 downto 0)=>B3(15 downto 0)
              );


    product(7 downto 0)<= B3(7 downto 0);




end SCHEMATIC;
```

Listing 10: Adder_2.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADDER_2 is
port(   A : In std_logic_vector (7 downto 0);
              B : In std_logic_vector (7 downto 0);
              Z : Out std_logic_vector (7 downto 0));
end ADDER_2;
```

```
architecture BEH_ADDER_2 of ADDER_2 is
   signal sum_out : std_logic_vector(8 downto 0);
   begin


   sum_out <= ('0' & A) + ('0' & B);

   Z(0) <= sum_out(8) OR sum_out(0);
   Z(1) <= sum_out(8) OR sum_out(1);
   Z(2) <= sum_out(8) OR sum_out(2);
   Z(3) <= sum_out(8) OR sum_out(3);
   Z(4) <= sum_out(8) OR sum_out(4);
   Z(5) <= sum_out(8) OR sum_out(5);
   Z(6) <= sum_out(8) OR sum_out(6);
   Z(7) <= sum_out(8) OR sum_out(7);

end BEH_ADDER_2;



--architecture BEH_ADDER_2 of ADDER_2 is
--    signal sum_out : unsigned(9 downto 0);
--    begin
--
--    process(A, B)
--
--       constant zeros: unsigned(1 downto 0) := (others => '0');
--       variable sum_int: INTEGER;
--
--    begin
--    sum_out <= (zeros & unsigned(A)) + (zeros & unsigned(B));
--    sum_int := conv_integer(sum_out);
--
--    if(sum_int < 255) then
--       Z <= std_logic_vector(sum_out(7 downto 0));
--    else
--       Z <= "11111111";
--    end if;
--
--    end process;
--end BEH_ADDER_2;
--
```

Listing 11: Adder_3.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADDER_3 is
port(   A : In std_logic_vector (7 downto 0);
                B : In std_logic_vector (7 downto 0);
                C : In std_logic_vector (7 downto 0);
                Z : Out std_logic_vector (7 downto 0));
end ADDER_3;
```

```
architecture BEH_ADDER_3 of ADDER_3 is
signal sum_out : std_logic_vector(9 downto 0);
   begin

      sum_out <= ("00" & A) + ("00" & B) + ("00" & C);

      Z(0) <= sum_out(9) OR sum_out(8) OR sum_out(0);
      Z(1) <= sum_out(9) OR sum_out(8) OR sum_out(1);
      Z(2) <= sum_out(9) OR sum_out(8) OR sum_out(2);
      Z(3) <= sum_out(9) OR sum_out(8) OR sum_out(3);
      Z(4) <= sum_out(9) OR sum_out(8) OR sum_out(4);
      Z(5) <= sum_out(9) OR sum_out(8) OR sum_out(5);
      Z(6) <= sum_out(9) OR sum_out(8) OR sum_out(6);
      Z(7) <= sum_out(9) OR sum_out(8) OR sum_out(7);


end BEH_ADDER_3;



--architecture BEH_ADDER_3 of ADDER_3 is
--    signal sum_out : unsigned(9 downto 0);
--    signal temp : std_logic_vector(7 downto 0);
--    signal c2, c1: std_logic;
--    begin
--
--
--    process(A, B, C)
--
--       constant zeros: unsigned(1 downto 0) := (others => '0');
--       variable sum_int: INTEGER;
--
--    begin
--    sum_out <= (zeros & unsigned(A)) + (zeros & unsigned(B)) + (zeros & unsigned(C));
--    sum_int := conv_integer(sum_out);
--    if(sum_int < 255) then
--       Z <= std_logic_vector(sum_out(7 downto 0));
--    else
--       Z <= "11111111";
--    end if;
--    end process;
--end BEH_ADDER_3;
```

Listing 12: MUX_4.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity MUX_4 is

port (
      SEL: in STD_LOGIC_VECTOR (1 downto 0);
      A,B,C: in STD_LOGIC_VECTOR(7 downto 0);
      SIG: out STD_LOGIC_VECTOR(7 downto 0));
end MUX_4;

architecture BEH_MUX of MUX_4 is
```

```
begin
    SEL_PROCESS: process (SEL,A,B,C)
    begin
      case SEL is
        when "00"  => SIG <= A;
        when "01"  => SIG <= B;
        when "10"  => SIG <= C;
        when others => SIG <= (others => '0');
      end case;
    end process SEL_PROCESS;
end BEH_MUX;
```

Listing 13: Processor_3.vhd

```
library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;

entity Processor_3 is
      Port (   CLOCK : In     std_logic;
               RESET : In     std_logic;

                    Read_In_Mem:                Out std_logic;
               Write_In_Mem : Out std_logic;

                    Read_Out_Mem:               Out std_logic;
               Write_Out_Mem:   Out std_logic;

                    Read_Addr_In_Mem:   Out std_logic_vector(15 downto 0);

               Read_Addr_Out_Mem:       Out std_logic_vector(15 downto 0);
                    Write_Addr_Out_Mem: Out std_logic_vector(15 downto 0);

                    Data_in_1:  In std_logic_vector(7 downto 0);
                    Data_in_2:  In std_logic_vector(7 downto 0);
                    Data_out: Out std_logic_vector(7 downto 0);
                    Filter: In std_logic_vector(7 downto 0);
                    disable_filter: In std_logic

                 );
end Processor_3;

architecture SCHEMATIC_PROC_3 of Processor_3 is

   component  ADDER_3
   port(         A : In std_logic_vector (7 downto 0);
                 B : In std_logic_vector (7 downto 0);
                 C : In std_logic_vector (7 downto 0);
                 Z : Out std_logic_vector (7 downto 0));
   end component ADDER_3;


   component  ADDER_2
   port(         A : In std_logic_vector (7 downto 0);
                 B : In std_logic_vector (7 downto 0);
                 Z : Out std_logic_vector (7 downto 0));
   end component ADDER_2;
```

```
component SHIFTREG is
   Port (  CLOCK : In     std_logic;
           RESET : In     std_logic;
           disable : In     std_logic;
           QK : In     std_logic_vector (7 downto 0);
           Q : InOut    std_logic_vector (71 downto 0) );
end component SHIFTREG;

component Multiplier
   Port (
         num1 : In     std_logic_vector (7 downto 0);
                num2 : In     std_logic_vector (7 downto 0);
         product : Out    std_logic_vector (7 downto 0) );
end component Multiplier;

component MUX_4 is
port (
      SEL: in STD_LOGIC_VECTOR (1 downto 0);
      A,B,C: in STD_LOGIC_VECTOR(7 downto 0);
      SIG: out STD_LOGIC_VECTOR(7 downto 0));
end component MUX_4;

component MUX_2 is
port (
      SEL: in STD_LOGIC;
      A,B: in STD_LOGIC_VECTOR(15 downto 0);
      SIG: out STD_LOGIC_VECTOR(15 downto 0));
end component MUX_2;

component MUX_2_1 is
port (
      SEL: in STD_LOGIC;
      A,B: in STD_LOGIC;
      SIG: out STD_LOGIC);
end component MUX_2_1;

component FSM_in_3
     port (
           clock:           in std_logic;
                reset:              in std_logic;
                address:            out std_logic_vector(15 downto 0);
                can_read:   out std_logic;
                can_write:  out std_logic;
                disable_cache: out std_logic
          );
 end component FSM_in_3;

 component FSM_out_3
     port (
           clock:           in std_logic;
                reset:              in std_logic;
                read_address:                 out std_logic_vector(15 downto 0);
                write_address:                out std_logic_vector(15 downto 0);
                can_read:   out std_logic;
                can_write:  out std_logic;
```

```
                sel: out std_logic_vector(1 downto 0)
            );
 end component FSM_out_3;

component REG is
     port(
             D : in std_logic_vector(7 downto 0);
             Clock, Reset : in std_logic;
             Q : out std_logic_vector(7 downto 0));
end component REG;

 signal disable_to_cache: std_logic;
 signal cache_bits: std_logic_vector(71 downto 0);
 signal filter_bits: std_logic_vector(71 downto 0);
 signal mult1_out: std_logic_vector(7 downto 0);
 signal mult2_out: std_logic_vector(7 downto 0);
 signal mult3_out: std_logic_vector(7 downto 0);
 signal mult4_out: std_logic_vector(7 downto 0);
 signal mult5_out: std_logic_vector(7 downto 0);
 signal mult6_out: std_logic_vector(7 downto 0);
 signal mult7_out: std_logic_vector(7 downto 0);
 signal mult8_out: std_logic_vector(7 downto 0);
 signal mult9_out: std_logic_vector(7 downto 0);
 signal add1_out: std_logic_vector(7 downto 0);
 signal add2_out: std_logic_vector(7 downto 0);
 signal add3_out: std_logic_vector(7 downto 0);
 constant zeros: unsigned(7 downto 0) := (others => '0');
 signal select_adder: std_logic_vector(1 downto 0);
 signal mux_out: std_logic_vector(7 downto 0);

 begin

    fsm_input:
    FSM_in_3 port map(CLOCK, RESET, Read_Addr_In_Mem, Read_In_Mem, Write_In_Mem, disable_t

    fsm_output:
    FSM_out_3 port map(CLOCK, RESET, Read_Addr_Out_Mem, Write_Addr_Out_Mem,
                       Read_Out_Mem, Write_Out_Mem, select_adder);

    cache:
    SHIFTREG port map(CLOCK, RESET, disable_to_cache, Data_in_1, cache_bits);

    filtermask:
    SHIFTREG port map(CLOCK, RESET, disable_filter, Filter, filter_bits);

    Mult1:
    Multiplier port map(cache_bits(7 downto 0), filter_bits(7 downto 0),mult1_out);

    Mult2:
    Multiplier port map(cache_bits(15 downto 8), filter_bits(15 downto 8),mult2_out);

    Mult3:
    Multiplier port map(cache_bits(23 downto 16), filter_bits(23 downto 16),mult3_out);

    Mult4:
    Multiplier port map(cache_bits(31 downto 24), filter_bits(31 downto 24),mult4_out);
```

```
        Mult5:
        Multiplier port map(cache_bits(39 downto 32), filter_bits(39 downto 32),mult5_out);

        Mult6:
        Multiplier port map(cache_bits(47 downto 40), filter_bits(47 downto 40),mult6_out);

        Mult7:
        Multiplier port map(cache_bits(55 downto 48), filter_bits(55 downto 48),mult7_out);

        Mult8:
        Multiplier port map(cache_bits(63 downto 56), filter_bits(63 downto 56),mult8_out);

        Mult9:
        Multiplier port map(cache_bits(71 downto 64), filter_bits(71 downto 64),mult9_out);

        Add1:
        Adder_3 port map(mult1_out, mult2_out, mult3_out, add1_out);

        Add2:
        Adder_3 port map(mult4_out, mult5_out, mult6_out, add2_out);

        Add3:
        Adder_3 port map(mult7_out, mult8_out, mult9_out, add3_out);

        Multiplexer:
        Mux_4 port map(select_adder, add1_out, add2_out, add3_out, mux_out);

        Add_new_value:
       Adder_2 port map(Data_in_2, mux_out, Data_out);

--       regster : Reg port map("11111111", Clock, Reset, data_out);

end SCHEMATIC_PROC_3;
```

Listing 14: TB_filter.vhd

```
library IEEE;
        use IEEE.std_logic_1164.all;
        use STD.textio.all;
        use IEEE.std_logic_arith.all;
        use IEEE.std_logic_textio.all;
        use IEEE.std_logic_signed.all;

entity TB_Filter is
end TB_Filter;

architecture A of TB_Filter is

   signal Clock: std_logic;

        signal mem_Enable:       std_logic;
        signal mem_give_zeros: std_logic;

        signal mem1_Data_in: std_logic_vector(7 downto 0);

   signal proc_RESET: std_logic;
   signal proc_Read_In_Mem:     std_logic;
   signal proc_Write_In_Mem:    std_logic;
```

```
    signal proc_Read_Out_Mem:     std_logic;
         signal proc_Write_Out_Mem:       std_logic;
    signal proc_Read_Addr_In_Mem:         std_logic_vector(15 downto 0);
    signal proc_Read_Addr_Out_Mem:        std_logic_vector(15 downto 0);
         signal proc_Write_Addr_Out_Mem: std_logic_vector(15 downto 0);

         signal proc_Data_in_1: std_logic_vector(7 downto 0);
         signal proc_Data_in_2: std_logic_vector(7 downto 0);
         signal proc_Data_out: std_logic_vector(7 downto 0);

         signal proc_filter_disable: std_logic;
         signal proc_filter: std_logic_vector(7 downto 0);

component MEMORY is
port(
    Clock:                in std_logic;
         Enable:          in std_logic;
         Give_Zeros: in std_logic;
         Read:            in std_logic;
         Write:           in std_logic;
         Read_Addr:       in std_logic_vector(15 downto 0);
         Write_Addr:      in std_logic_vector(15 downto 0);
         Data_in:         in std_logic_vector(7 downto 0);
         Data_out:        out std_logic_vector(7 downto 0)
);
end component MEMORY;

component Processor_3 is
        Port (  CLOCK : In     std_logic;
                RESET : In     std_logic;

                      Read_In_Mem:                  Out std_logic;
                Write_In_Mem: Out std_logic;
                      Read_Out_Mem:                 Out std_logic;
                Write_Out_Mem:    Out std_logic;

                      Read_Addr_In_Mem:    Out std_logic_vector(15 downto 0);

                Read_Addr_Out_Mem:        Out std_logic_vector(15 downto 0);
                      Write_Addr_Out_Mem: Out std_logic_vector(15 downto 0);

                      Data_in_1:  In std_logic_vector(7 downto 0);
                      Data_in_2:  In std_logic_vector(7 downto 0);

                      Data_out: Out std_logic_vector(7 downto 0);
                      Filter: In std_logic_vector(7 downto 0);
                      disable_filter: In std_logic
                    );
end component Processor_3;


begin

 --  UUTP : PROCESSOR_3
--       Port Map (CLOCK, proc_RESET, proc_Read_In_Mem, proc_Read_Out_Mem, proc_Write_Out_Mem,
--                 proc_Read_Addr_In_Mem, proc_Read_Addr_Out_Mem, proc_Write_Addr_Out_Mem,
--                 proc_Data_In_1, proc_Data_In_2, proc_Data_Out, proc_filter, proc_filter_dis
```

```
--
--    UUTM_in : MEMORY
--       Port Map (CLOCK, mem_Enable, '0', mem1_Read, mem1_Write,
--                 mem1_Read_Addr, mem1_Write_Addr, mem1_Data_In, mem1_Data_Out);
--
--    UUTM_out : MEMORY
--       Port Map (CLOCK, mem_Enable, mem_give_zeros, mem2_Read, mem2_Write,
--                 mem2_Read_Addr, mem2_Write_Addr, mem2_Data_In, mem2_Data_Out);

    UUTP : PROCESSOR_3
       Port Map (CLOCK, proc_RESET, proc_Read_In_Mem, proc_Write_In_Mem, proc_Read_Out_Mem, pr
                 proc_Read_Addr_In_Mem, proc_Read_Addr_Out_Mem, proc_Write_Addr_Out_Mem,
                 proc_Data_In_1, proc_Data_In_2, proc_Data_Out, proc_filter, proc_filter_disab

    UUTM_in : MEMORY
       Port Map (CLOCK, mem_Enable, '0', proc_Read_In_Mem, proc_Write_In_Mem,
                 proc_Read_Addr_In_Mem, proc_Read_Addr_In_Mem, mem1_data_in, proc_Data_In_1);

    UUTM_out : MEMORY
       Port Map (CLOCK, mem_Enable, mem_give_zeros, proc_Read_Out_Mem, proc_Write_Out_Mem,
                 proc_Read_Addr_Out_Mem, proc_Write_Addr_Out_Mem, proc_Data_Out, proc_Data_In_

     clock_signal:
     process begin
            Clock <= '1';
            wait for 1 ns;
            Clock <= '0';
            wait for 1 ns;
     end process;

     process

            CONSTANT NLOOPS : integer := 15;
            file cmdfile, outfile: TEXT;                    -- Define the file 'handle'
            variable line_in,line_out: Line; -- Line buffers
            variable good: boolean;          -- Status of the read operations
            variable A,B: std_logic_vector(7 downto 0);
            variable S: std_logic_vector(55 downto 0);
            variable Z: std_logic_vector(31 downto 0);
            variable ERR: std_logic_vector(55 downto 0);
            variable c : integer;
            -- constant TEST_PASSED: string := "Test passed:";
            -- constant TEST_FAILED: string := "Test FAILED:";

        begin

            proc_RESET <= '0';

            wait for 2 ns;

            mem_Enable <= '1';
            mem_give_zeros <= '1';

       -- initialize the filter with pixels
       proc_filter_disable <= '0';
       c := 1;
```

```
proc_RESET <= '1';
proc_filter_disable <= '0';
FILE_OPEN(cmdfile,"lena_256x256.mem",READ_MODE);
-- start filling memory 1 with the image pixels from hex file.

loop

   if endfile(cmdfile) then  -- Check EOF
      assert false
      report "End of file encountered; exiting."
           severity NOTE;
           exit;
   end if;

   readline(cmdfile,line_in);      -- Read a line from the file

   next when line_in'length = 0;  -- Skip empty lines

   hread(line_in,A,good);          -- Read the D argument as hex value
   assert good report "Text I/O read error" severity ERROR;

   mem1_Data_In <= A(7 downto 0);
   write(line_out,c);

  if(c = 10) then
        proc_filter_disable <= '1';
   elsif ((c mod 2) = 0) then
      proc_filter <= "00000001";
      else
      proc_filter <= "00000000";
  end if;

   wait for 2 ns;
   c := c + 1;



end loop;

     c := 1;
     mem_give_zeros <= '0';

loop

    wait for 2 ns;

    c := c + 1;

    if(c = 1560577) then  -- ( 3 + 9 ) * 254 * 256 (780288)
       exit;
  end if;

end loop;

write(line_out, string'("---------- END----------"));
writeline(OUTPUT, line_out);
```

```
 end process;

end A;
```