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 11, 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*
- *Josep Renard*
- *Markku Eerola*

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.

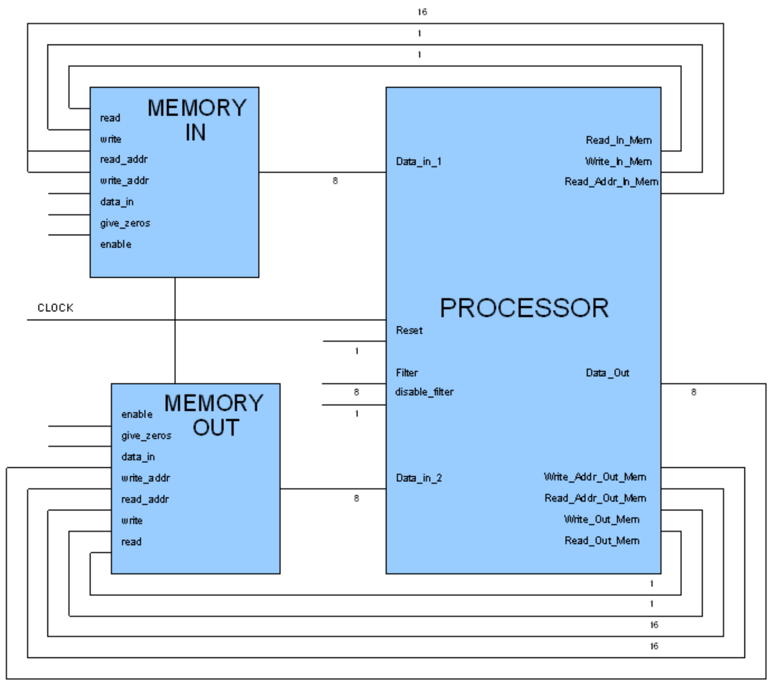


Figure 1: Filter unit design

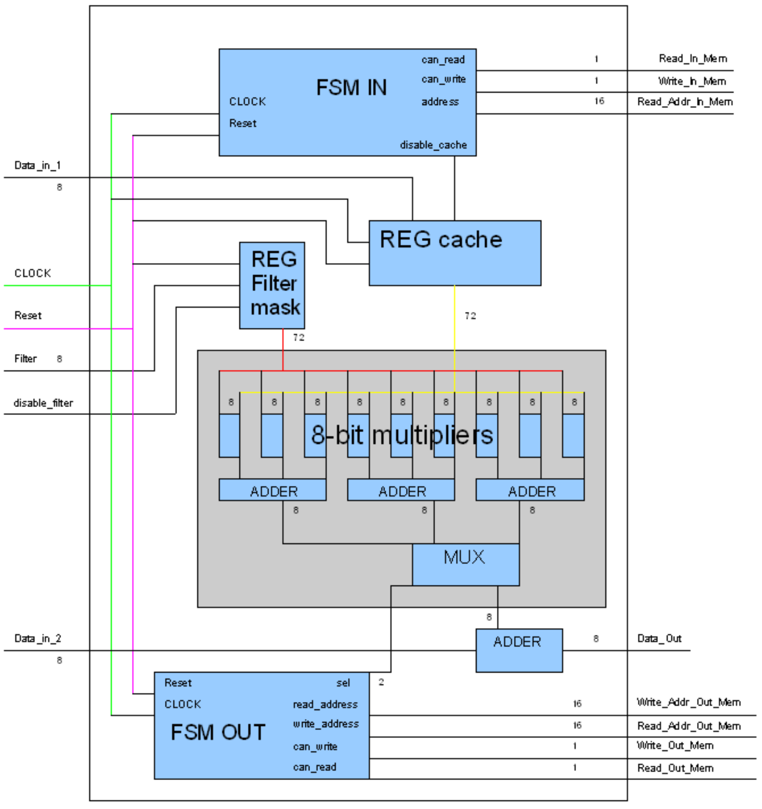


Figure 2: Processor architecture

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

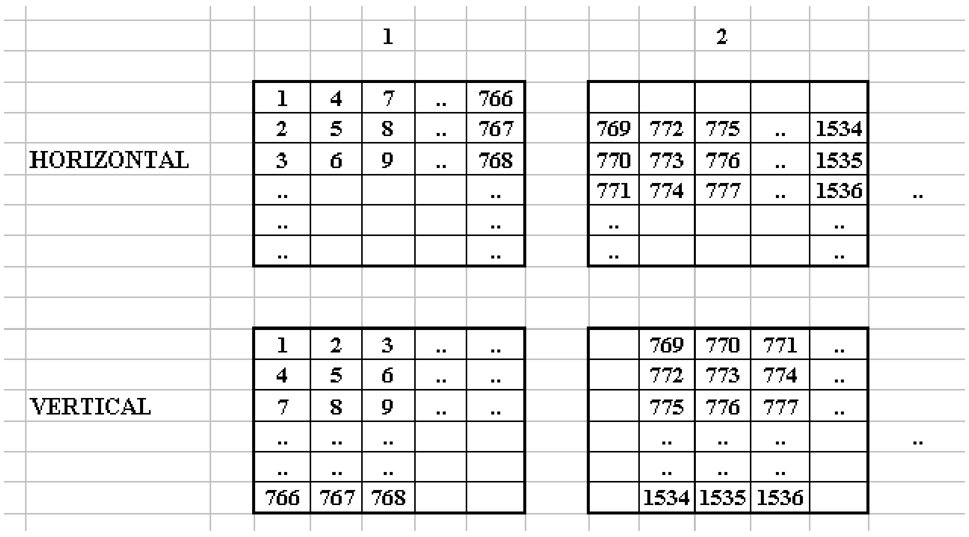
3.2 Memory Read and Write by Processor

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 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.

Hence we have chosen the approach in which at no point of time, would both the Controllers be active. This can be seen from the figure **??**.

3.3 Memory Access Sequence



			1					2			
	1	4	7	..	766						
	2	5	8	..	767	769	772	775	..	1534	
HORIZONTAL	3	6	9	..	768	770	773	776	..	1535	
	..				..	771	774	777	..	1536	..
	..				..	..				..	
	..				..	..				..	
	1	2	3	..	..		769	770	771	..	
	4	5	6	..	..		772	773	774	..	
VERTICAL	7	8	9	..	..		775	776	777	..	
	..	..	..				..	..	..		..
	..	..	..				..	..	..		
	766	767	768				1534	1535	1536		

Figure 3: Sequence of Memory Address Access

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 3). 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 cycles. The actual reports can be seen in the appendix, but a summary of the results can be seen in table 1.

Table 1: Summary of Design Vision reports

$\mathbf{T}_C[ns]$	$\mathbf{P}_{stat}[mW]$	$\mathbf{P}_{dyn}[mW]$	$\mathbf{P}_{tot}[mW]$	$\mathbf{A}_{comb}[\mu m^2]$	$\mathbf{A}_{tot}[\mu m^2]$	$\mathbf{T}_{cp}[ns]$
7	0.11	1.60	1.71	44067	53079	4.7
5	0.11	1.71	1.82	44067	53079	4.7
3	0.13	2.19	2.32	49595	58611	2.9
2	0.20	2.60	2.80	58668	67700	1.9

6 Results

Table 2: Summary

$\mathbf{T}_C[ns]$	**Critical Path** [ns]	**N. cycles** (256 x 256)	$\mathbf{E}_{pc}[mW/MHz]$	**AREA** $[\mu m^2]$
7	4.7	3121152	0.01197	53079
5	4.7	3121152	0.00910	53079
3	2.9	3121152	0.00696	58611
2	1.9	3121152	0.00560	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:

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.