DSD COURSE PROJECT

Progettazione di un Filtro

FIR passa basso

Supervisor:
Luca Fanucci

Author:
Daniele Battista

Computer Engineering
A.Y. 2013/2014

Contents

1. Introduction

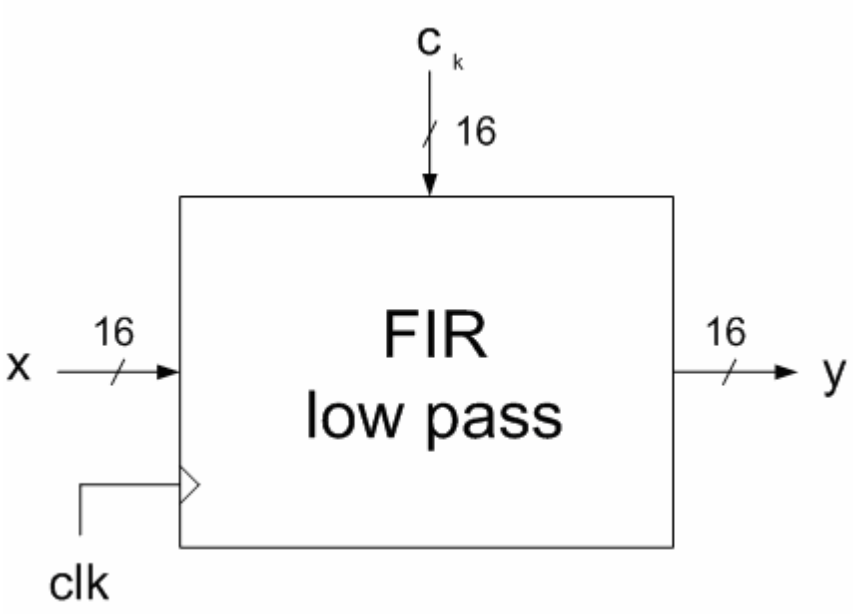
The project requires to design a digital circuit that implements a FIR (low pass) filter (finite impulse response) of order N (N=6) with a normalized cutoff frequency $f_T = 0.2\, f_C$ (f_C = sampling frequency). The mathematical law that rules the filter is shown below:

$$y[n] = \sum_{i=0}^{N} ci \cdot x[n-i]$$

where:

1) *x[n]* represents the input at instant n;
2) c_i representes the corresponding factor;
3) *y[n]* represents the output at instant n;

Inputs, output and factors should be represented by 16 bits. The factors values are: $c_0 = 0.0135$, $c_1 = 0.0785$, $c_2 = 0.2409$, $c_3 = 0.3344$, $c_4 = 0.2409$, $c_5 = 0.0785$, $c_6 = 0.0135$.



Generally, talking about this type of filters, it is possible to say that the impulse response should become null in a finite time, otherwise they would be called Infinite Impulse Response (IIR). The main characteristics of these filters are : the non-requirement of a feedback, meaning that errors in rounding values are not so important in summed iterations, simplifying the design of the filter; the intrinsic stability; the factors symmetry that let a designer show a possible easy way to implement the project and the whole

architecture. Filters like FIRs are usually used in radio-electronic and optical components.
As far as the design of this filter, it's necessary to give a particular attention to the mathematical law that rules in it. It is clear that, what we need, is an adder, to implement the sum shown in the formula, and a multiplier, to have the result of the product between factors and input. Obviously for adders it is good to use the ripple-carry adder, summing bit by bit and having as result not only the sum, but also the carryout, very important in the implementation of the filter.
Another component, which is required in the implementation, is a register, or better, a series of registers that are able to define a delay chain, helpful for our goal. In fact, if we give a better look at the formula, we'll discover that, if the input is given to the filter for a time that is "not short", the output will be affected not only by the presence of the input at time t, but by the presence of all those inputs at times *t-1,t-2* ecc.. up to the input given at time *t-6*. And each of the inputs will be multiplied by a different factor at any new computation. Let's try to explain it in a better way showing a simple example: let's suppose we have an input K at time t_0: for the fact that all other registers won't contain any value, the output will be:

$$y[0] = c_0 \cdot x[0] = c_0 \cdot K$$

Then, at time t_1, another input arrives with value U, so the output will be:

$$y[1] = c_0 \cdot x[1] + c_1 \cdot x[0] = c_0 \cdot U + c_1 \cdot K$$

and so on. As far as we would like to have a general description of the whole system, a complete formula that shows us this is:

$$y[n] = c_0 \cdot x[n] + c_1 \cdot x[n-1] + c_2 \cdot x[n-2] + c_3 \cdot x[n-3] + c_4 \cdot x[n-4] + c_5 \cdot x[n-5] + \\ + c_6 \cdot x[n-6]$$

From this result we can see that our last input will be affected by the prior factor, the previous one by the second factor and so on. What is important to observe is that, noticing that factors are symmetric, the first and the last input could be first summed and then multiplied for the corresponding factor. This consideration could be extended to all input except for the one that has to be multiplied by the middle factor. So, summing up results, a general expression of our output should be the following:

$$y[n] = c_0 \cdot (x[n] + x[n-6]) + c_1 \cdot (x[n-1] + x[n-5]) + c_2 \cdot (x[n-2] + x[n-4]) + c_3 \cdot x[n-3]$$

Given all these considerations, it is necessary to develop a solution that could implement the filter. One possible solution is to divide the filter in three layers, the first one embedding the late chain of registers, the second one containing 7 multipliers (one for each register output), the third one holding a chain of 6 adders able to present the output. This one is the simplest solution and the one which is practical to develop only taking a look at the mathematical formula the filter embeds. Therefore looking at the formula that has been provided above, we can see that another solution can be implemented: instead of having 3 layers in the architecture, we can add one more, representing the layers of prior adders summing inputs before multiplication. In this case we need 3 adders for the first layer, 4 multipliers for the second layer, 3 adders for our last layer. This is an optimized solution that minimizes the number of multipliers and this is the one we are going to develop in this project. It is obvious that not all adders are equal and inputs could have different size. At the end of both solution, it is required a little circuit that selects only 16 bits given from a wire that should contain an higher number of bits, due to all operations done in all previous layers.

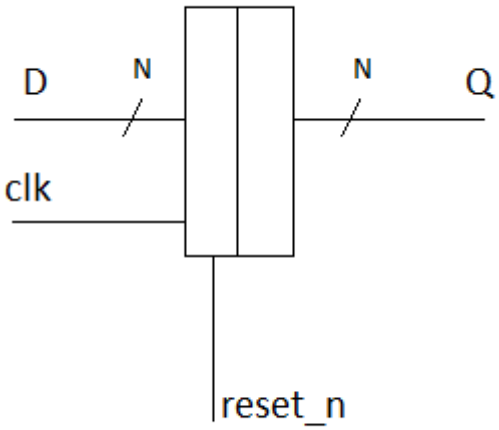
2. Architecture Description

Once these two solutions have been provided and taking in account that the second one is an optimized one, what we need to do is to develop this solution showing all features and all components required in it. Starting from the input we can see that, at the very beginning, there is a register, or better, a chain of registers. This solution has been considered for the fact that registers are memory components, built with flip-flops D-edge triggered, and so, at every clock cycle, they adjust their output to their actual input. This is very useful for our goal because the filter input at time t_0 will be present only for the first register. At time t_1, if the input changes, we have the incoming input shown to the first register, instead the last one shown to the second register, due to clock arriving. If we consider that input may change without a precise scheme, we'll notice that each register will contain a different value and this value will pass from a register to the following one in a clock cycle. After 7 clock cycles, the value passed the whole chain of registers and is going to "disappear" at next clock cycle, if the previous value is different. We are able to provide a VHDL solution for this, showing exactly what happens at each clock cycle:

```
REG:process(CLK)
        begin
                if(rising_edge(clk)) then
                        if(reset_n = '0') then
                                Q <= "0000000000000000";
                        else
                                Q <= D;
                        end if;
                end if;
        end process REG;
```

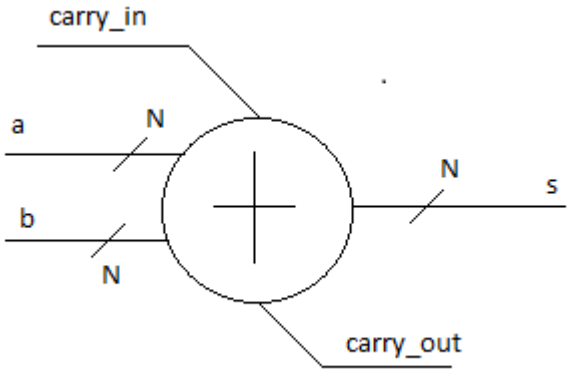
where:

1) D represents the input;
2) Q represents the output;
3) clk represents the clock;
4) reset_n represents the reset wire;

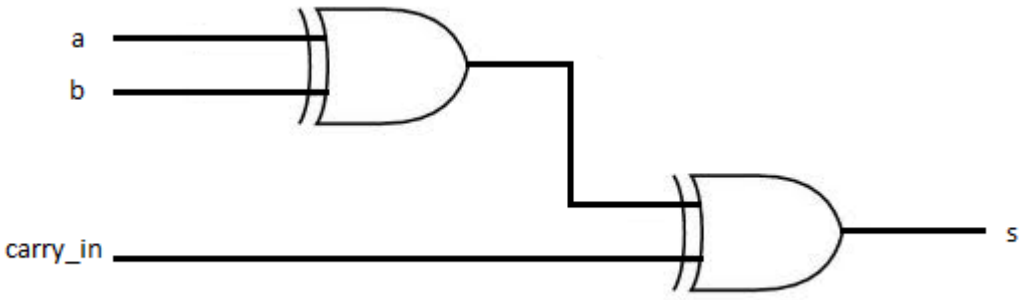


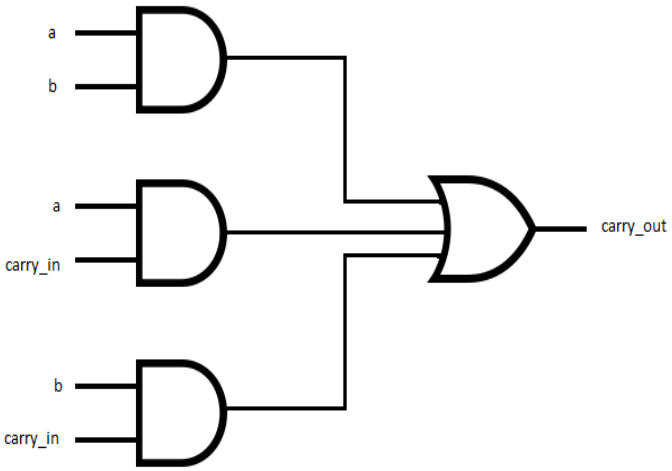
Taking in account that the filter has got 7 different coefficients, we need 7 registers to store all values that will be multiplied by the corresponding coefficient.

Another component that should be provided to develop this filter project is, without any doubt, an adder. It is necessary to use it just after the chain of registers, summing values that come out from them. In this project we decided to use ripple-carry adders, which are developed as the way shown in the figure. There we can see that we have 3 inputs and 2 outputs: *a* and *b* represents values to be summed and they are represented on N bits, *carry_in* is a bit that contain "1" if the previous sum overflew, *s* represents our result extended on N bits, *carry_out* is a value that shows us if the sum overflew. To produce these output values we need a little bit of logic inside, and in the case:



s[i] = a[i] XOR b[i] XOR carry_in
carry_out = (a[i] AND b[i]) OR (a[i] AND carry_in) OR (b[i] AND carry_in)

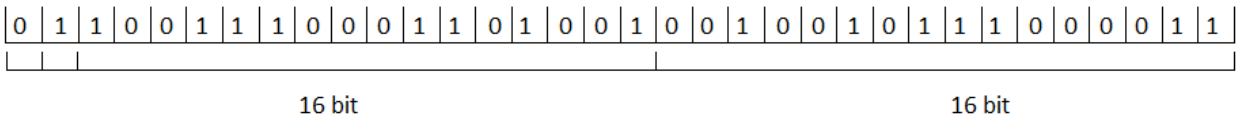




In this architecture we use 2 types of ripple-carry adder, mainly for the fact that in our second layer we have 16-bits inputs, instead, after multipliers, we need 34-bits adders. Considerations about all these features will be given below, after all components presentation.

As we know that in the third layer we need multipliers, we have to specify how to multiply 2 numbers within a component. First, we need to observe that, supposing equal size for both incoming values, the result will be extended on a number of bits equal to the two factors sum. There are no other inputs or outputs. We know that outputs coming from previous adders are extended on 16 bits, but if *carry_out* has been set to “1”, 16 bits are not enough. For this reason, we decide to concatenate each adder result with the corresponding *carry_out*, in order to have a complete and precise representation of what we have. This consideration, and implementation, is necessary if we consider the moment in which input is equal to a rectangle with amplitude 2^16 -1. This is the case in which the representation will not be safe and we’ll have problems with all

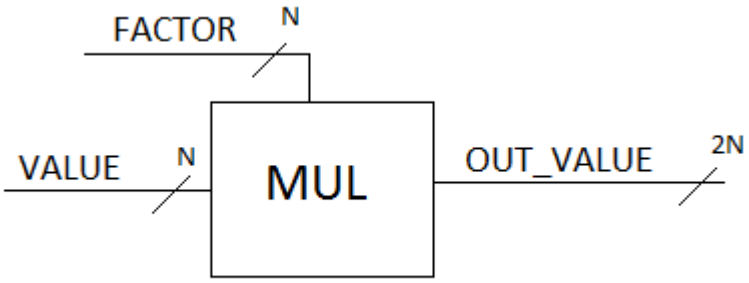
operations needed after. In particular, after many trials, it has been observed that instead of having a signal that reaches the incoming value, it goes over that value and then it stabilize itself to a lower value till when the input changes. After all these considerations we know that incoming values to our multipliers will be given on 17 bits and so our result will be on 34 bit. This is also important when we have to develop last adders. In fact they should be 34-bits adders, in which *carry_out* won't be of great interest. A justification of this could be found in this reasoning: our input value is represented on 16 bit having lsb_x = c, our coefficients are represented on 16 bit, then extended to 17 in order to have a symmetric multiplier, having lsb_c = 2^(-16). Considering the special case in which our input is a rectangle with amplitude 2^16 -1 and knowing that the sum of all coefficients is equal to 1.0002, our result, given on 34 bits will be equal to 65548.107. Our 34-bits will be composed this way:



first 16 least significant bits represent all digits after comma, due to the multiplication by factors; following 16 bits are those who represent our output value; following bit is important to understand if our result can be represented on 16 bits: if it is equal to "1", it means that the output is higher than 2^16-1 and so we have to saturate it to 2^16-1, in order to have a good representation. Last bit will be forever zero because the highest obtainable number can be represented on 33 bits. We have this bit for the fact that we decided to have multiplication inputs on the same bit size, 17. After all these considerations, we can pass to multiplier implementation, showing a development like this:

```
MULTIPLICATION: process(value, factor)

    variable reg : std_logic_vector(33 downto 0);
    variable add : std_logic_vector(17 downto 0);

    begin

        reg := "00000000000000000"&factor;
        for i in 0 to 16 loop
```

```
            if ( reg(0) = '1' ) then
                add := ('0'&value)+('0'&reg(33 downto 17));
                reg := add&reg(16 downto 1);

            else

                reg:= '0'&reg(33 downto 1);

            end if;

        end loop;

        out_value <= reg;

    END  process MULTIPLICATION;
```

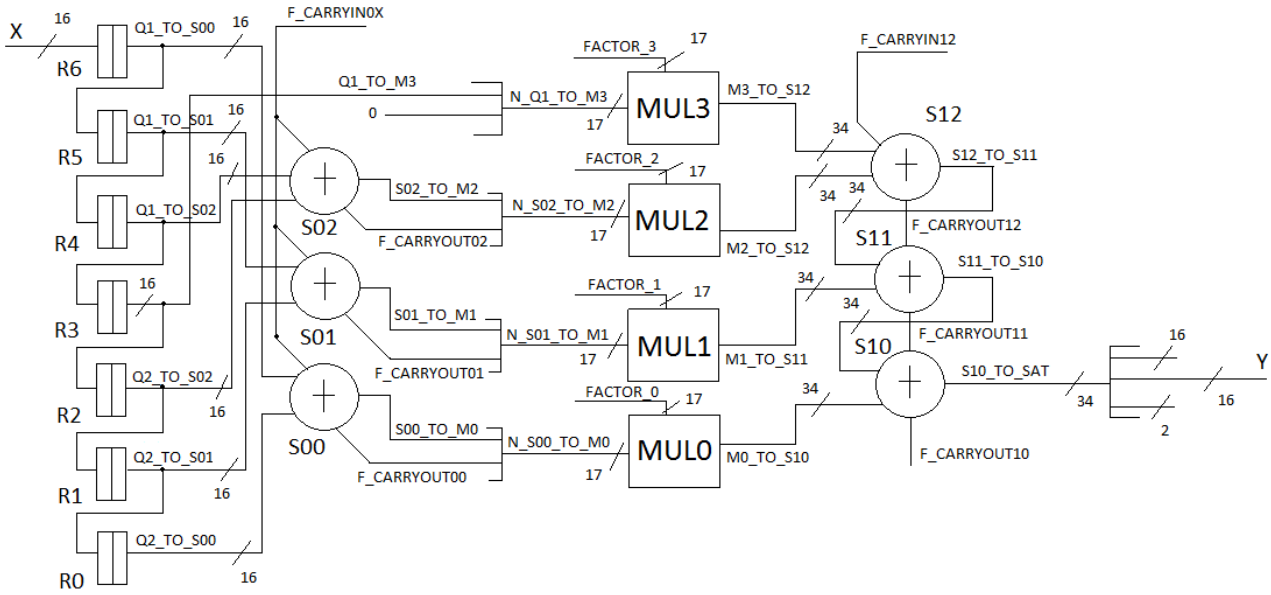


As we can see, we tried to implement the normal mathematical algorithm with some modifications. The algorithm works this way: it is needed to prepare 2 variables, one called "*reg*" which represents our final result and one called "*add*" representing mid-sums useful to get to the final result. At the beginning we prepare our "*reg*" concatenating (N - size_of_factor) '0' with our factor. Then we compare last bit of *reg* with 1: if they are not equal, the value of "*reg*" is shifted to right adding a '0' at the beginning, otherwise we prepare "*add*" summing our "*value*" concatenated with a '0' with the most significant part of "*reg*" concatenated with a '0'. This sum will be the most significant part of our new "*reg*" concatenated to all bits remained in the old "*reg*", except for the bit we used before to compare with '1'. When this loop is repeated for all bits contained in our "*factor*", it is possible to assign "*reg*" to the output valueso that it can be visualized. In order to show that this is a correct solution we can say that the sum, computed on "*add*", is made only if we have '1' on last "*reg*" bit. This sum is computed summing value with the most significant part our "*reg*" that, considering a lot of iterations, will embody all sums done up to now. Then it is assigned to the most significant part of "*reg*" and shifted in order to analyze all bits coming from "*factor*" and carrying the final result from the most significant part to the least significant part. Of

course we needed some concatenations and some extensions on our adders to have a better precision on our operations. In order to clarify our procedure and not to complicate structure and code, we added the library "*IEEE.STD_LOGIC_UNSIGNED.ALL*" that is able to compute the "*+*" sign. Another solution would lead to use an adder, for example the one we used in our structure, but this solution has been preferred.

After last operations, for all problems discussed above, we have to develop a circuit that selects only 16 bits taken from *S10_TO_SAT* 34 bits. This circuit is very simple and it is represented by a discard circuit in which we select bits from the 32nd to the 17th. We can discard for sure the 34th for the fact that the highest number obtainable as result can be represented on 33 bits. We have to pay attention to the 33th bit because, if it is going to be '1', the integer part of our general result won't be representable on 16 bits, so we adopted, for this case, the saturation solution. In this manner our output 16 bits will be all equal to '1'. Least 16 significant bits will be discarded, representing all numbers after comma.

It has been reached the moment in which it is possible to declare our arichitecture as structural. We implemented it trying to solve all little problems present in this architecture and then, using a mapping, we connected all our pieces, already tested, in order to show how the FIR filter is done inside. The designed solution is shown below.

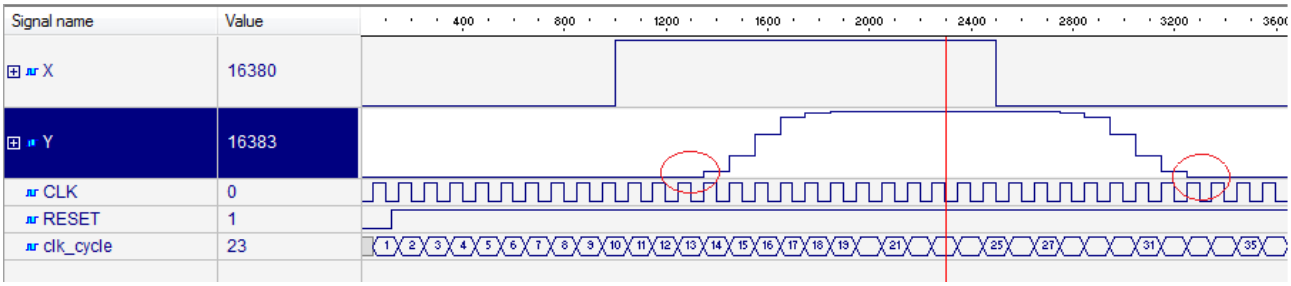


3. VHDL Code

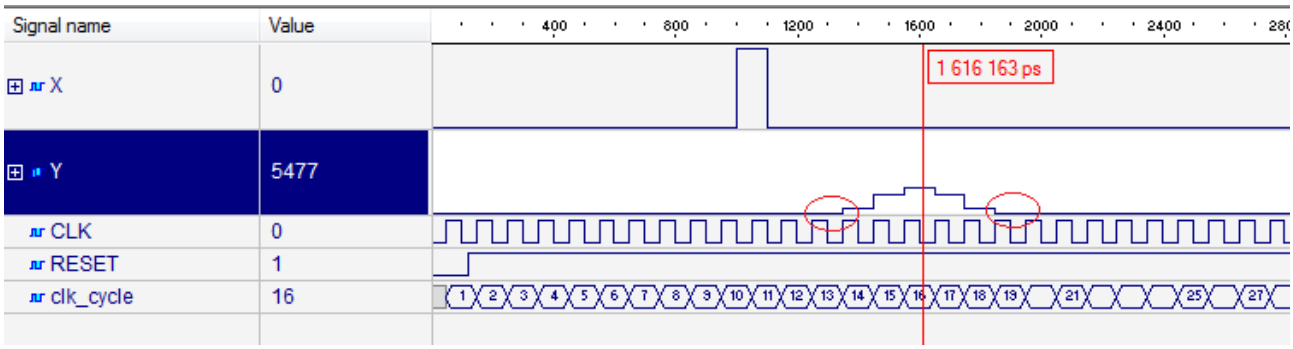
About VHDL code it is possible to say that each component has been developed on a single file and first tried with its specific testbench. Taking a look at our filter we can see that the page in which it is described is completely full of signals and components declaration. We decided to choose this approach in order to be very clear. Code won't be reported here because would be attached to the whole relation. We tried to be very clear adding a lot of comments not to create some problems to readers.

4. Test bench verification

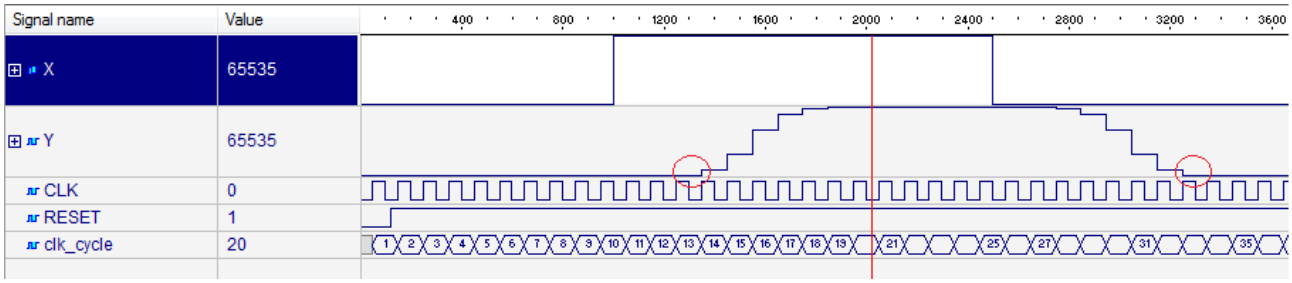
Testing a circuit is one of the most difficult and interesting things when you have to deal with arguments and subjects like these. In order to prove how a filter works and which is the response it has any time you provide it a new input, the tester should try with lots of signals, like rectangles, dirac-deltas, sinus, cosines ecc. For such a kind of project we have decided to try the filter with only 2 kinds of wave and, in particular, rectangle and dirac-deltas. In order to show many behaviors the filter could have, it has been decided to try many rectangles and dirac-deltas with different values so that it will be simple for a reader to understand the general situation. Let's start with a general rectangle with amplitude 16380 ("0011111111111100").



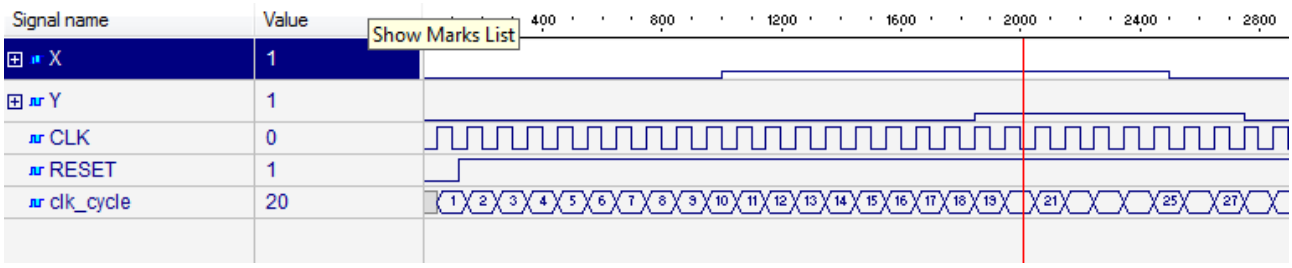
As we can see from this simple example, the filter provides the right output after a while, because of all registers, and the output reaches the input value in 7 steps. The first one is not clear (the one circled in red) because of a small value that cannot be represented in such a kind of graph. However there are seven steps for the fact that our input shifts from one register to another and when it reaches the next one, it adds to our input all its computation chain that provides a higher value to the output. It is possible also to observe that the final output value is not completely equal to our input but it is a little bit higher for the fact that the sum of all coefficients is not equal to 1, but it is equal to 1.0002. Now let's pass and analyze a new input: dirac-delta with the same value.



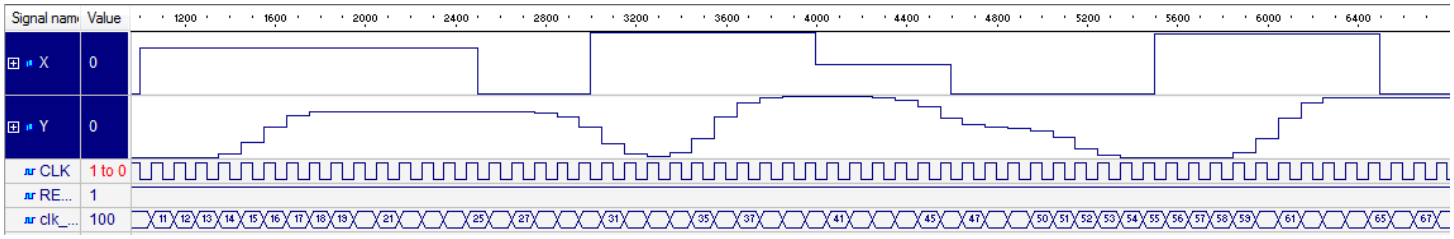
From this graph we can see that, apart the same consideration made above about red circled positions, now the behavior could seem different, but reasoning on it, we have to notice that once the input is present for one register, it goes through and will be present for the next one. Doing so, as far as a delta input is "valid" only for a time, the prior one won't have anymore the input and so it will affect only a register per time. This means that the maximum obtainable value each time will be our input multiplied by the relative coefficient. Our peak will be represented by our input multiplied by the highest coefficient. Let's now pass to show a situation in which we have a rectangle with the maximum possible amplitude.



We can now observe that situation is not changed if we consider our first result. The only important thing is that now the maximum output reached, when it multiplied by all coefficients, is not higher than the output for the fact that it has been decide to saturate our response if the whole result couldn't be represented on 16 bits. An operation like this is possible also for all considerations made above in which it has been stated that a new bit is added to multipliers inputs and so our general response will be represented on 34 bits. Giving some other considerations, it is possible to state that if we give as input a rectangle with amplitude 1, we'll have a result different from zero only when our input value reaches the register tied to the highest coefficient. Let's show this.



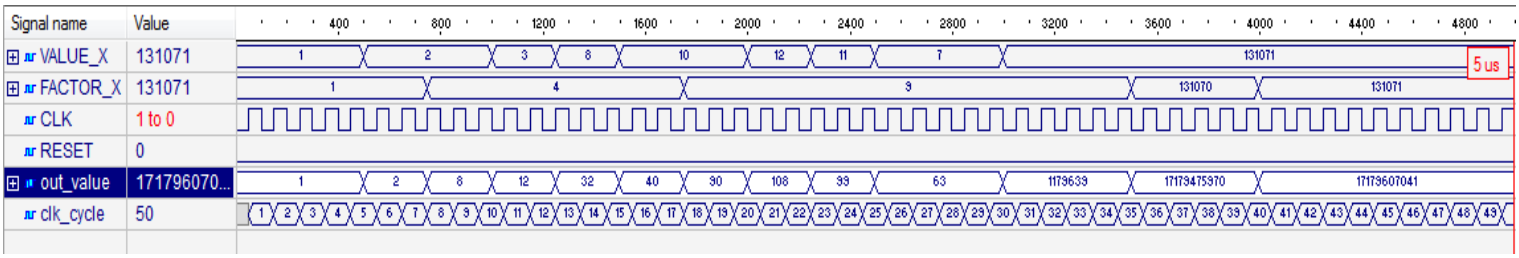
Here it is not possible to see all steps that lead to the highest value because they will be represented by numbers smaller than 1. So only the final value is shown. To give a general description of how the filter works, we are going to show a situation in which many inputs are given in order to see our the response will be affected. Inputs will be taken pseudo-randomly.



This is what the filter does providing it a testbench with entries like the ones shown below.

```
WHEN 10 => X <= "0000000000000000";

when 20 => X <= "1100001110001011";
when 30 => X <= "1100000000011011";
when 50 => X <= "0000000000000000";
when 60 => X <= "1111111111111111";
when 80 => X <= "0111111111111111";
when 92 => X <= "0000000000000000";
when 110 => X <= "1111111100110000";
when 130 => X <= "0000000000000000";
```

Another important thing to show could be a test done our multiplier, the only sub-component that has been developed from scratch. We'll provide some inputs to our network and we'll show which will be our results.



In the previous figure VALUE_X and FACTOR_X represent out inputs, out_value represents out output.

5. Xilinx Spartan 3 – ISE Tool Development

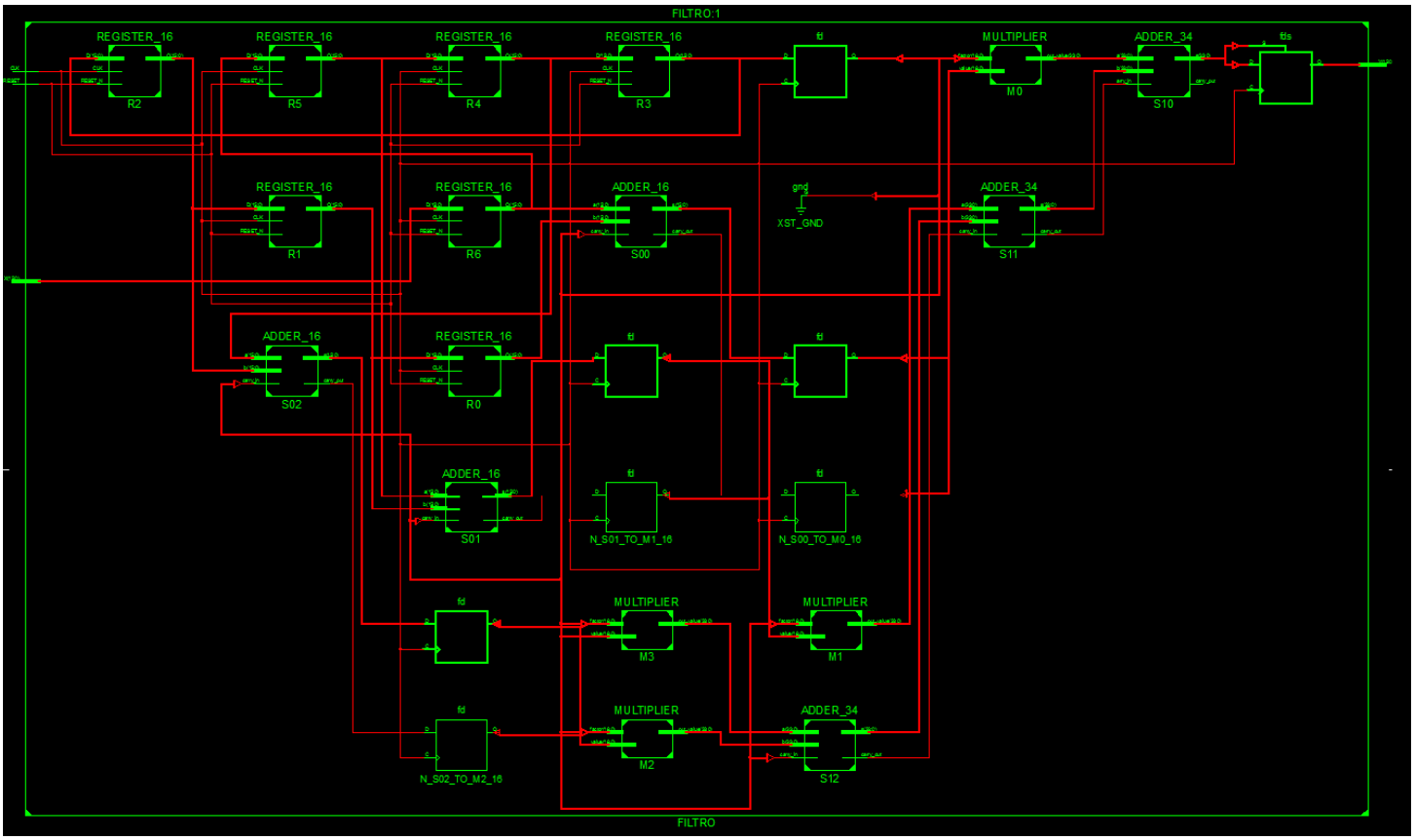
At the very end of the project, we tried to synthesize it on the ISE tool from Xilinx Spartan 3. We tried to follow the approach in which we gave basic configurations and we tried to understand which were the constraints needed for this case. The procedure followed consisted in the creation of a new project and in the execution of all steps required to arrive at the "Generate programming File" phase. During the synthesis we met some warnings that, after a check, were visualized as failed initialization of some variables which were not going to be very useful in our project.

During this phase we observed all results that were shown on the "Device Utilization Summary" and without giving a detailed explanation of what resulted, we are going to demonstrate it with a picture.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	195	26,624	1%	
Number of 4 input LUTs	809	26,624	3%	
Number of occupied Slices	510	13,312	3%	
Number of Slices containing only related logic	510	510	100%	
Number of Slices containing unrelated logic	0	510	0%	
Total Number of 4 input LUTs	848	26,624	3%	
Number used as logic	809			
Number used as a route-thru	39			
Number of bonded IOBs	34	487	6%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	2.24			

At this point we decided which constraints we were able to set in order to see if they were met or not. After some trials we observed that the maximum frequency, taking the variable shown in the figure below as sample, reached is 28460837.89 hz, that is about 28.4 Mhz. Taking in account this, we fixed as a time constraint the "Data Path Delay", as shown below, to 40.0 ns and of course this requirement is met.

```
Slack (setup path):     4.864ns (requirement - (data path - clock path skew + uncertainty))
  Source:               N_S02_TO_M2_4 (FF)
  Destination:          Y_4 (FF)
  Requirement:          40.000ns
  Data Path Delay:      35.136ns (Levels of Logic = 27)
  Clock Path Skew:      0.000ns
  Source Clock:         CLK_BUFGP rising at 0.000ns
  Destination Clock:    CLK_BUFGP rising at 40.000ns
  Clock Uncertainty:    0.000ns
```



In the figure above we want to show what is possible to see through the "View RTL Schematic" option available in the "Synthesize – XST" phase. In the figure all connections established in our architecture are presented to see how the filter is inside.

6. Conclusions

What has been realized and presented here is a simple and intuitive way to project and develop a Low Pass-Filter. We tried to exploit the simplicity of this filter passing through mathematics that, in many cases, could help you to realize things and to understand better. We are sure that there will be many other ways to do a thing like this, but this was, for us, the best one to implement.

In the final testbench provided, it has been introduced a particular input made of a lot of steps and delta-diarc in order to show which is the response to such a kind of signals and to test the filter doing a lot of trials, considering our possibilities.

This presentation has been provided in English in order to try to understand things and learn them in a better way. We'll regret for all errors a reader could find in it and we'll be very pleasant for each good advice.