Istanbul Technical University
Electrical and Electronics Faculty
Electronics and Communication Engineering

Introduction to Embedded Systems (EHB 326E)

2021-2022 Fall

(CRN: 10791)

Lecturer: Prof. Dr. Müştak Erhan Yalçın
Asisstant: Araş. Gör. Serdar Duran

PROJECT NO. 8

FIR FILTER USING PICOBLAZE

GROUP NAME : GROUP AAA

--Serden Sait Eranil--
040170025

--Fatmanur Babacan--
040170251

--Riselda Kodra--
040180935

Contents

A) What is a FIR (Finite-duration Impulse Response)?

A filter is a mathematical means of modifying certain frequencies of a signal relative to the others. By determining the frequency content of the filter, a discrete or continous time signal that realizes a specific frequency content can be designed.

A discrete-time system can be expressed by a set of parameters in the transfer function or difference equation that results in a desired impulse response or frequency response within a specified bandwidth. These systems can be categorized into two basic systems which are infite-duration impulse response and finite-duration impulse response systems. If we were to express those two systems in a basic way, an IIR system is an approximation of a transfer function that is a rational function of z-transform variable z and a FIR system is a polynomial approximation [1].

As it can be guessed, IIR systems can be derived from the continous-time filters and they are used mostly when the discrete-time filters were first used. On the other hand, since FIR filters cannot be obtained by such techniques that are used in determining IIR systems, FIR filters came into use after practical implementations become quite important because infinite-duration systems cannot be implemented in practice.

In this project, a FIR filter is of consideration. Therefore, we skip the explanation of the IIR filters.

A FIR filter can be described by several structures. However, some of these forms have special properties; therefore, they are the most popular ones. These forms are direct, cascade and parallel form structures. Generally speaking, the direct form is one of the easiest and mostly used one and the one that we use in this project. The direct form realization can be described by the non-recursive difference equation as can be shown in the equation 1.

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k) \tag{1}$$

$$h[n] = \begin{cases} b_n & n = 0,1,\dots,M \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

As it can be seen from the equation 1, the output consists of a weighted linear combination of M-1 past values of the input and the weighted current value of the input. The structure is illustrated in the Figure 1 below.

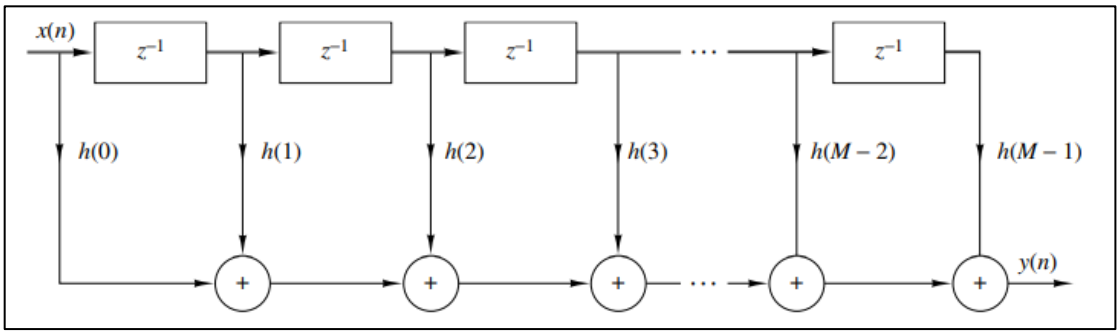


Figure 1: Direct form realization of FIR system[2]

Since this structure resembles a tapped delay line or a transversal system, the direct-form realization is often called a transversal or tapped-delay-line filter [2].

One can easily notice from equation 1 that this equation is a discrete convolution. This is reasonable because the output of a linear time-invariant (LTI) system is described by the convolution of the impulse response of the system and the given input. Here, the filter is our system whose coefficients are determined by the designer to implement desired specific functionality. In FIR filters, for example, linear phase response can be provided if those coefficients of the filter satisfy the symmetry condition given as in the equation 3a or 3b.

$$h[M-n] = h[n] \; for \;\; n = 0,1,\dots,M \quad (3a)$$
$$h[M-n] = -h[n] \; for \; n = 0,1,\dots,M \quad (3b)$$

Here, in this project we did not use a special type of FIR filter; however, if the convenient coefficients are chosen, a filter with any characteristics of passband, cut-off frequency etc. can be created. In order to test the hardware, we created we will present random numbers of positive and negative coefficients with different magnitudes to encapsulate the all cases.

B) Definition of the problem and how it can be done

In this project, it is asked to implement a FIR filter which realizes the equation (4) via connecting the I/O ports of the picoblaze to the FPGA's 18x18 hardware multiplier.

$$y[n] = a_1x[n] + a_2x[n-1] + a_2x[n-1] \cdots + a_8x[n-7] \quad (4)$$

One can easily notice that equation 4 is the open form of equation 1 and M=7. Therefore, it just required to implement a convolution operation in Vivado, Xilinx and control the operation of this convolution via the Picoblaze microprocessor.

What we need to do is simply shift the input signal one by one, and while shifting these inputs multiply them with their corresponding weights (coefficients) using the multiplier of Vivado's 18x18 multiplier and add them to produce the output array.

For example, let's produce the output for a given coefficients a[n] = [1 2 3 4 5 6 7 8] and x[n] = [1 2 3]. While we are shifting the input array to the right we fill the emptied places with zeros as shown.

x[n] = [1 2 3]	* a[1] = 1	→	x'[n] = [1 2 3]
x[n-1] = [0 1 2 3]	* a[2] = 2	→	x'[n-1] = [0 2 4 6]
x[n-2] = [0 0 1 2 3]	* a[3] = 3	→	x'[n-2] = [0 0 3 6 9]
x[n-3] = [0 0 0 1 2 3]	* a[4] = 4	→	x'[n-3] = [0 0 0 4 8 12]
x[n-4] = [0 0 0 0 1 2 3]	* a[5] = 5	→	x'[n-4] = [0 0 0 0 5 10 15]
x[n-5] = [0 0 0 0 0 1 2 3]	* a[6] = 6	→	x'[n-5] = [0 0 0 0 0 6 12 18]
x[n-6] = [0 0 0 0 0 0 1 2 3]	* a[7] = 7	→	x'[n-6] = [0 0 0 0 0 0 7 14 21]
x[n-7] = [0 0 0 0 0 0 0 1 2 3]	* a[8] = 8	→	x'[n-7] = [0 0 0 0 0 0 0 8 16 24]

If we add all those x'[n-i] values we obtain the **y[n] = [1 4 10 16 22 28 34 40 37 24].** We can verify the result via checking its size since the convolution of two signals with sizes n and k will result an array with n-k-1 elements. Here n=3, k=8 so we are expecting 10 elements in the y[n], the result is correct. However, we won't implement this example, since it is too far easy and there is no input signal with 3 samples. We will implement examples with negative numbers.

C) Algorithm in FIDEX IDE (Software Part)

Our software part consist of 4 parts which are loading the filter coefficients, writing coefficients to the spadRAM, outputting these coefficients from spadRAM, and reading the input values from BRAM and outputting to the filter part.

C.1) Loading the registers

```
22      ;======== a's ========;
23          load s0 5 ;a1          5
24          load s1 243 ;a2      -13
25          load s2 1 ;a3          1
26          load s3 249 ;a4      -7
27          load s4 245 ;a5      -11
28          load s5 7 ;a6          7
29          load s6 3 ;a7          3
30          load s7 250 ;a8      -6
```

Figure 2: Loading the filter coefficients

We start the software part first loading the registers with the desired filter coefficients. Here, we need to explain the number format. Since the Picoblaze works on the unsigned 8-bit numbers, negative numbers cannot be represented and operations cannot be made upon these numbers in software part [3].

Digital systems use 2's complement number format in order to represent signed number in a consistent way. This method is used in order for signed operations (addition, subtraction, multiplication) to be identical to those for unsigned binary numbers [4].

For example, the s1 register's value is 243 and its binary representation is "11110011", this binary number is treated like a negative number in Vivado environment because the MSB is 1. The algorithm for determining the negative numbers work is that the magnitude of the two should add to 256. Namely, if we want to represent -13 in Vivado environment, we should use the unsigned representation of 243 because 13+243=256. Therefore, to check our algorithm whether it works or not we include negative numbers like this. Here the filter coefficients a[n] = [5 -13 1 -7 -11 7 3 -6]. These numbers do not have a special meaning, we selected them randomly so that our algorithm not only works for a special set of numbers.

C.2) Writing to spadRAM

```
32      ;======== a's ram ========;
33          wrmem s0 0x00
34          wrmem s1 0x01
35          wrmem s2 0x02
36          wrmem s3 0x03
37          wrmem s4 0x04
38          wrmem s5 0x05
39          wrmem s6 0x06
40          wrmem s7 0x07
```

Figure 3: Writing to the spadRAM to set free registers

This part is unarguably the most easy and understandable part of the software part, here the content is registers are just written into the address of spadRAM starting from 0x00. The final address 0x07 is important because it will be used for reading in part 3.

C.3) Outputting the coefficients

```
42          ;=====a'lari ver=====;
43          load s0 0x07 ;a8
44          load s1 0xFF
45          load sE 8; counter
46 lp:      comp sE 0
47          jump z end1
48          rdmem s2 (s0)
49          wrprt s2,(s1) ;as
50          sub s0 1
51          sub s1 1
52          sub sE,1
53          jump lp
```

Figure 4: Reading from spadRAM and outputting

In this part of the software section, we first read the values from the spadRAM starting with 0x07. There is no difference between starting 0x00 and 0x07 because it is just for convention all of the coefficients will be ready before the input signal reaches the multipliers. s0 register holds the address of the final place of the coefficients in spadRAM.

s1 register holds the BUS address, 0xff, this address is important because in the hardware section we will use this address to discriminate the coefficients and inputs sent from PicoBlaze via out_port.

s2 register is used to hold the value of the currently read coefficient and its content is written to address 0xff hold by s1. Then decreasing the s0, s1 we are outputting the different values of coefficients starting from a8 to a1. sE register is used only for looping and it has no different usage throughout the program.

C.4) Outputting the input array

```
55 end1:    ;=====x's=====;
56          load sF 0x1f ;x0
57          load sD 0x00 ;x0
58 lp1:     comp s1 sD
59          jump z end
60          rdprt s1 (sF)
61          rdprt s1 (sF);
62          wrprt s1,0x00 ;xs
63          add sF 1
64          NOP
65          jump lp1
66 end:     jump end
```

Figure 5: Reading from BRAM and outputting

This part is the most important part of this section because the input values which a filter takes and applies a transformation. For example, if we thought filter coefficients to be equal and to be normalized to its length, then this FIR filter would be the moving average filter. It would take the input values and apply the averaging over by one by. Therefore, once the filter specification is determined the coefficients of that filter does not change; however, inputs values are subject to change almost all the time.

You may wonder where it is reading values from. But this will be explained in the hardware part since the content of the block ram is written there. However, this part also needs some explanation.

First of all, sF register is loaded with the 0x1f. This address is arbitrarily chosen just to state which address of BRAM to look at and has no implicit meaning. sD value is loaded 0x00 and used for comparison statement. Until the 0 input is read from the loop continues to read values from BRAM. This algorithm is not correct completely because if a zero input is read in between then it will stop looping even if it should not. This is quite easy to solve by just using the number of inputs as a looper but we prefer this way to save us from stating the number of inputs every time we changed the inputs in the design. Since it is just for testing, it will not cause any problem.

There is a duplicate rdprt instruction. The reason for that is when we request a data from the BRAM, it sends this value after one clock cycle; however, the rdprt instruction reading the value that it currently sees. Therefore, adding the same instruction again, we made sure that the desired value is read without error.

Finally, the x value is outputted with wrprt instruction via the BUS address 0x00. This BUS address is used to separate between the coefficients and the input values outputted by just one port serially in the hardware part.

Also, it will be explained in hardware part but as you may notice there is 16 clock cycles between consecutive examples of x values. That means that, FIR filter part of the hardware waits for the next input value for 16 clock cycles. Therefore, FIR filter part of the hardware will work 16 times slower than the actual clock. NOP operation is added with the aim of making it this number 2^4, and no implicit meaning.

D) Analyzing the Algorithm in Vivado (Hardware Part)

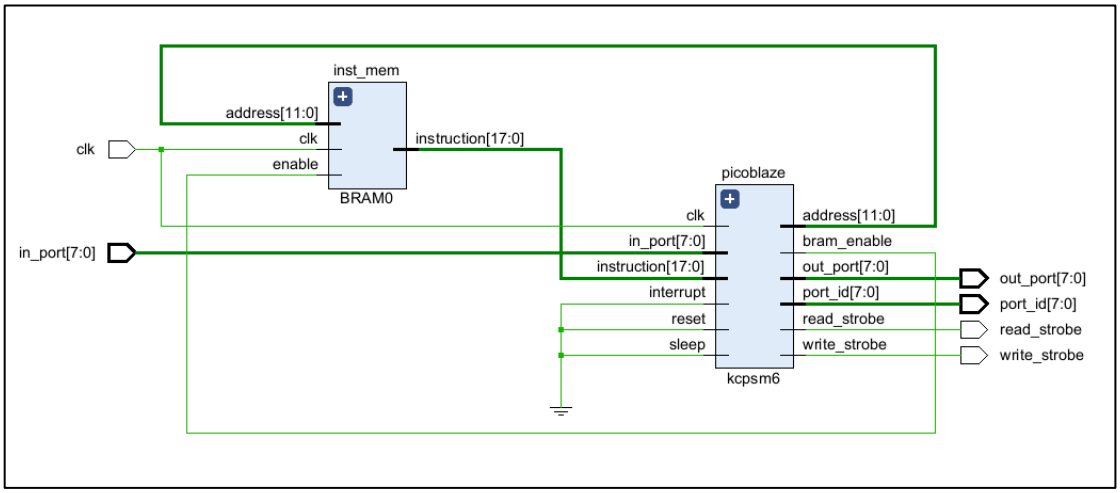


Figure 6: RTL Schematic of the top module

The RTL schematic of the top module which comprises PicoBlaze and instruction memory for the PicoBlaze and the I/0 ports can be seen from the Figure 6. read_strobe is used for reading input values from BRAM and write_strobe is used inside a module named selector to discriminate the values sent by PicoBlaze serially.

Since we are trying to implement the structure shown in Figure 1, we need 7 8-bit D Flip Flops, 8 multipliers, and 8 adders. These modules will be connected in the IP design of Vivado, Xilinx. However, since it won't fit to show all the design here, we will divide the IP design into parts and explain them as in the software section. The IP design basically consists of 4 parts which are, memory part, selector part, clocking part, FIR part.

D.1) Memory Part

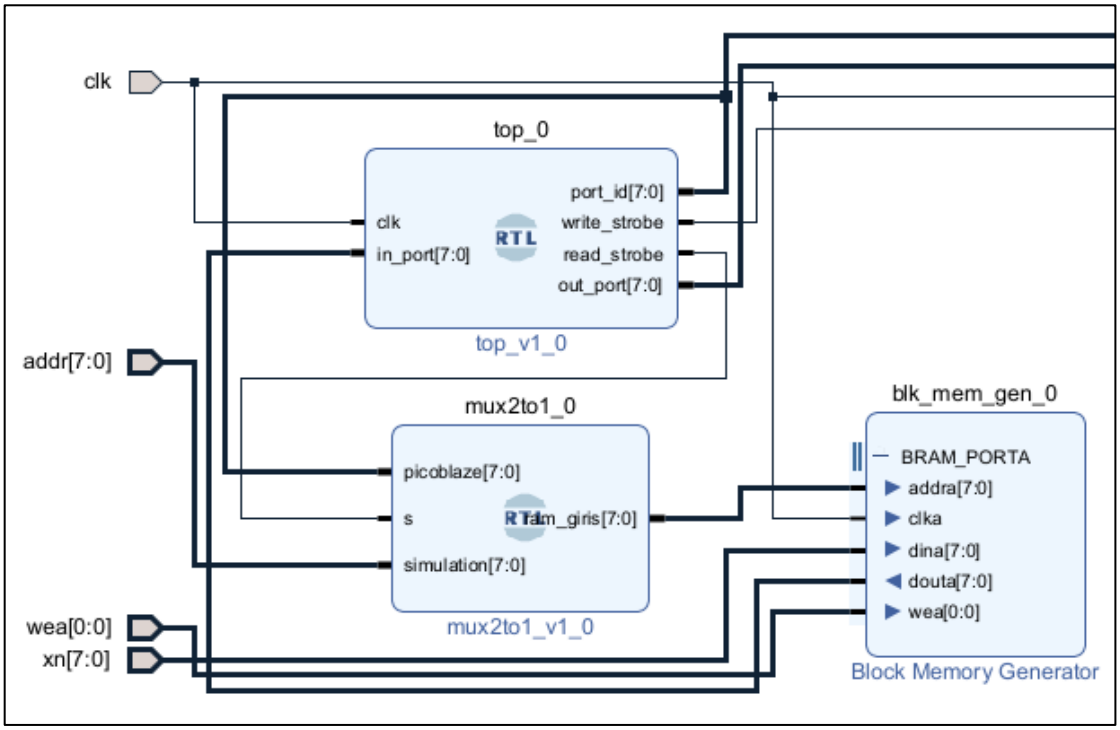


Figure 7: Memory Part consisting of the microprocessor, block ram and multiplexer

In the memory part of the hardware section, how the content is written to BRAM and how the content inside BRAM is read and what is the use of multiplexer will be explained in detail.

When the program is started, it loads the addresses specified in the testbench with the input values specified again in the testbench. Since the BRAM is a single port device, there is selection algorithm to decide who is going to use the RAM when. Therefore, the multiplexer will decide via its select input which is tied to read_strobe of the PicoBlaze. When read_strobe is low, the simulation input of multiplexer which is given by testbench will be transferred to addra of BRAM. On the other hand, when read_strobe is high, it means PicoBlaze wants to read values from the BRAM, therefore PicoBlaze input of the multiplexer which is tied to port_id will be transferred to ram_giris output which is tied to addra of the BRAM.

The wea (write enable) pin which allows designer to write values into ram by setting it high, and xn pin where the contents of the BRAM will be given serially are also given by the testbench file. As you can remember from the software part, PicoBlaze reads the input values starting from 0x1f address; therefore, we should write the xn values starting from that address to make sure that PicoBlaze reads the desired content. Also, the last input which is given by testbench should be zero so that PicoBlaze understand that the number of inputs that should be read is completed.

Let's analyze the testbench code.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity sim_top is
end sim_top;

architecture Behavioral of sim_top is
component fir_wrapper is
    Port ( addr : in STD_LOGIC_VECTOR ( 7 downto 0 );
           clk : in STD_LOGIC;
           wea : in STD_LOGIC_VECTOR ( 0 to 0 );
           xn : in STD_LOGIC_VECTOR ( 7 downto 0 ));
end component fir_wrapper;

signal clk :  STD_LOGIC;
signal addr:STD_LOGIC_VECTOR ( 7 downto 0 );
signal xn:STD_LOGIC_VECTOR ( 7 downto 0 );
signal wea : STD_LOGIC_VECTOR ( 0 to 0 );
begin
uut: fir_wrapper port map(
                addr => addr,
                clk => clk,
                wea => wea,
                xn=>xn);


process
begin
    clk <= '0';
    wait for 5ns;
    clk <= '1';
    wait for 5ns;
end process;
```

```
P_STIMULI : process begin

addr    <= "00011111"; --1f
xn      <= "11111111"; -- -1
wea(0)      <= '1';
wait for 10ns;
addr    <= "00100000"; -- 0x1f
xn      <= "00000010"; -- 2
wait for 10ns;
addr    <= "00100001"; -- 0x20
xn      <= "11111101"; -- -3
wait for 10ns;
addr    <= "00100010";-- 0x21
xn      <= "00000100"; -- 4
wait for 10ns;
addr    <= "00100011";-- 0x22
xn      <= "11111011"; -- -5
wait for 10ns;
addr    <= "00100100";-- 0x23
xn      <= "00000110";-- 6
wait for 10ns;
addr    <= "00100101"; -- 0x24
xn      <= "11111001"; --- -7
wait for 10ns;
addr    <= "00101101";-- 0x2c
xn      <= "00000000"; --
wait for 10ns;
wea(0) <='0';
wait for 10000000ns;

end process P_STIMULI;

end Behavioral;
```

Figure 8: Testbench file

From Figure 8, one can easily see the codes that are realizing the situation stated earlier. First it imports the component called fir_wrapper which is our IP design file. Then it maps the port inputs.

There is two different process that are running concurrently. The first one our first and actual clock which is 100 MHz and the second process is for writing the contents of BRAM to the specified addresses. As stated earlier, the starting address 0x1f is selected arbitrarily, and here also the inputs are selected arbitrary negative and positive inputs to see the performance of the FIR filter. Before writing the input values we set the wea input high and after the writing we set it low. Also notice that zero is given as the last input to make sure that the loop is terminated in software part.

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

entity mux2tol is
  port (simulation: in std_logic_vector(7 downto 0);
        picoblaze: in std_logic_vector(7 downto 0);
        s : in std_logic;
        ram_giris : out std_logic_vector(7 downto 0));
end mux2tol;

architecture behaviour of mux2tol is
begin
  process (simulation, picoblaze, s)
  begin
    if s = '0' then
      ram_giris <= simulation;
    else
      ram_giris <= picoblaze;
    end if;
  end process;
end behaviour;
```

Figure 9: Multiplexer to decide who is going to use BRAM

The VHDL code for the multiplexer used in IP design. As it can be seen it is a simple multiplexer. The simulation input is tied to testbench and PicoBlaze input is tied to port_id of PicoBlaze and multiplexer directs these input pins to ram_giris depending on the s bit.

D.2) Selector Part

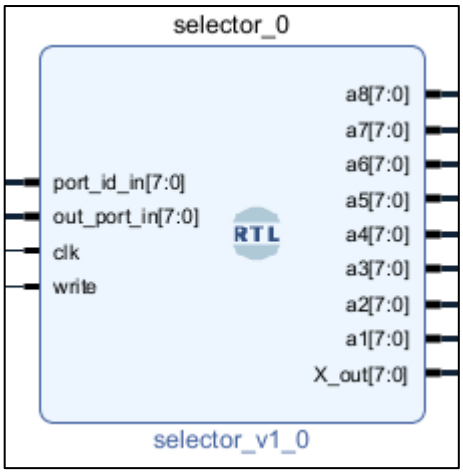


Figure10: Selector

```
entity selector is
Port (
        port_id_in : in std_logic_vector(7 downto 0);
        out_port_in : in std_logic_vector(7 downto 0);
        a8 : out std_logic_vector(7 downto 0):="00000000";
        a7 : out std_logic_vector(7 downto 0):="00000000";
        a6 : out std_logic_vector(7 downto 0):="00000000";
        a5 : out std_logic_vector(7 downto 0):="00000000";
        a4 : out std_logic_vector(7 downto 0):="00000000";
        a3 : out std_logic_vector(7 downto 0):="00000000";
        a2 : out std_logic_vector(7 downto 0):="00000000";
        a1 : out std_logic_vector(7 downto 0):="00000000";
        X_out : out std_logic_vector(7 downto 0):="00000000";
        clk : in std_logic;
        write : in std_logic);
end selector;
```

Figure 11: Entity definition of selector

In Figure 10 and 11, the block diagram and its port definitions are shown respectively. This selector module have a special role in our design. It takes the coefficients and inputs from the PicoBlaze and it transfers the content of the out_port based upon their address given by port_id_in.

```
signal x   :  std_logic_vector(7 downto 0):="00000000";
begin
process(clk)
begin
if (rising_edge(clk)) then
    if (write = '1') then
        if   (port_id_in = "11111111") then --0xff
        a8 <= out_port_in;
        elsif(port_id_in = "11111110") then --0xfe
        a7 <= out_port_in;
        elsif(port_id_in = "11111101") then --0xfd
        a6 <= out_port_in;
        elsif(port_id_in = "11111100") then --0xfc
        a5 <= out_port_in;
        elsif(port_id_in = "11111011") then --0xfb
        a4 <= out_port_in;
        elsif(port_id_in = "11111010") then --0xfa
        a3 <= out_port_in;
        elsif(port_id_in = "11111001") then --0xf9
        a2 <= out_port_in;
        elsif(port_id_in = "11111000") then --0xf8
        a1 <= out_port_in;
        elsif(port_id_in ="00000000") then
        X_out <=out_port_in;
        x <= out_port_in;
        else
        X_out <= x;
        end if;
    else
        X_out <= x;
    end if;
end if;
end process;
end Behavioral;
```

Figure 12: The process of selector

In the Figure 12, one can see how the discrimination between the values are performed. Since we determine which coefficient will be sent by which address, we can determine which one is sent just by controlling the port_id at that moment.

It can be seen from the Figure 4 that we are starting to output coefficients by the address 0xff, and then by decreasing the address value by one, PicoBlaze sent the next coefficient of the FIR filter. Since coefficients do not change frequently, giving them to FIR part parallelly is considered easier. Therefore, each coefficient has a unique port assigned for it. Also, as it can be seen again in Figure 4, when the address is 0x00, input values are given to selector.

The coefficients represented by a1 to a8 are given to their corresponding multipliers and X_out is given to the D Flip Flop input as it will be explained later on FIR part.

D.3) Clocking Part

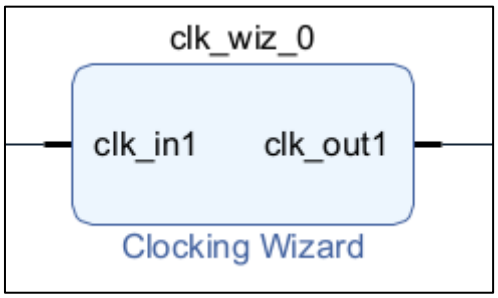


Figure 13: Clock generated by Clocking Wizard

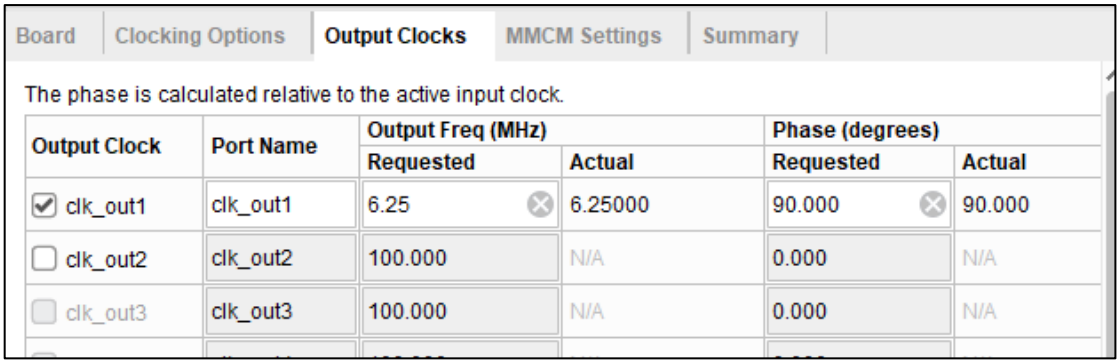


Figure 14: The important settings for Clocking Wizard

The IP block shown in Figure 13 is a block of IP Catalog inside Vivado, Xilinx. What it does basically is that it gets an input clock and produces an output clock with desired frequency and phase. Dividing the clock frequency is required in this design because PicoBlaze outputs the xn values serially in every 16 clock cycles as stated in the last part of the software section. That means that FIR part needs to be running slower than the PicoBlaze in order to satisfy the synchronization between multipliers and D Flip Flops.

In Figure 14, the settings where output frequency and phase are configured is shown. The output frequency is 6.25 MHz because 100 MHz/16 = 6.25 MHz and phase is selected 90 degrees to synchronize the rising edges of input and output clocks.

D.4) FIR Part

In the Figure 1, the structure of direct form of FIR filter is shown. What we are trying to do in this part of the hardware section is basically the same thing inside Figure 1. PicoBlaze will output the input values that it reads from BRAM to selector module and if the addresses are correct, selector will give these values to the X_out output.

X_out pin is connected the first multiplier and D Flip Flops. There are 8 multipliers and 7 D Flip Flops. The X_out pin fiven to first D Flip Flops will be delayed exactly one clock (generated one) everytime it passes one D Flip Flops. While passing through the D Flip Flops, the output of these D Flip Flops will be given to its corresponding multiplier with its corresponding coefficient. And the outputs resulted from the multiplication of these two inputs are added instantaneously by the Adder/Subtractor IP block of Vivado. Then the resulting array will be observed at the output called yn.

Please note that this algorithm is exactly the same one shown in Figure 1. The blocks represented by z^{-1} delays its input in time and directed arrows are the multiplication.

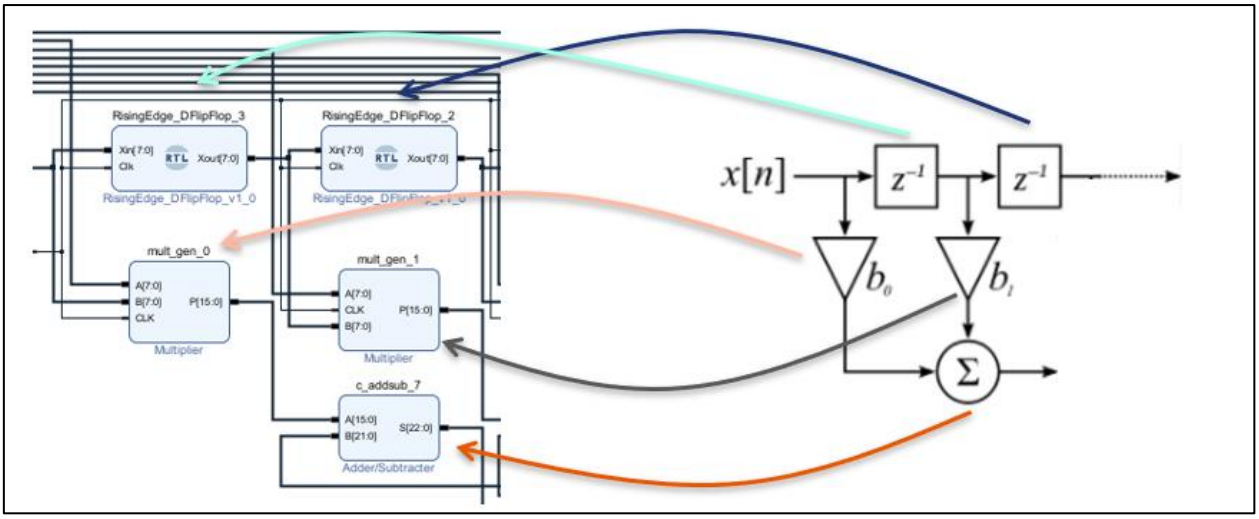
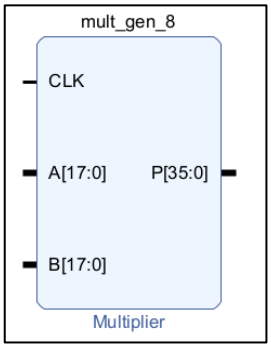


Figure 15: D Flip Flops and multiplier and adder blocks

In Figure 15, two multiplier block, two D Flip Flops and one adder is shown. This figure is only one part of the FIR part. The overall block diagram will be given after all of the design is explained.



In Figure 16, the 18x18 multiplier block is shown. It takes two signed inputs and produces again a signed output. Realizing the multiplication process inside the PicoBlaze is quite hard and inefficient. However, the multipliers presented to users by Vivado, Xilinx are pretty fast and efficient multipliers. It produces the output after one clock cycle. Its inputs are 18-bit by default; however, since PicoBlaze operates on 8-bit unsigned numbers, we configured it so that it takes 8-bit signed inputs and produces a 16-bit signed output.

Figure 16: 18x18 multiplier

```
1  Library IEEE;
2  USE IEEE.Std_logic_1164.all;
3
4  entity RisingEdge_DFlipFlop is
5     port(
6        Clk :in std_logic;
7        Xout : out std_logic_vector(7 downto 0);
8        Xin :in  std_logic_vector(7 downto 0)
9     );
10 end RisingEdge_DFlipFlop;
11 architecture Behavioral of RisingEdge_DFlipFlop is
12 begin
13  process(Clk)
14  begin
15     if(rising_edge(Clk)) then
16    Xout <= Xin;
17     end if;
18  end process;
19 end Behavioral;
```

Figure 17: VHDL code for 8-bit D Flip Flops

The block diagrams for D Flip Flops are created by us and the code for implementing it is given in Figure 17. As it can be seen from the Figure, it is just a basic D Flip Flops with 8-bit inputs and outputs. By putting 7 D Flip Flops, we made sure that the input signal will be delayed 7 times and by multiplying with corresponding coefficients at the end we will obtain equation 4.

The adder structure that can be seen in Figure 15 is added from the IP catalog of the Vivado, Xilinx. It is very efficient block and it produces the result immediately. However, it should be note that the last adder's output is the input of the previous adder. Also, when two 16-bit number are added, the result will be a 17-bit number otherwise overflow may occur. Similarly, one 15-bit number and 21-bit number are added the resulting number will be 22-bit long. In this project the output of last adder is 22-bit and this adder is shown in Figure 15. Now let's see the overall block diagram.

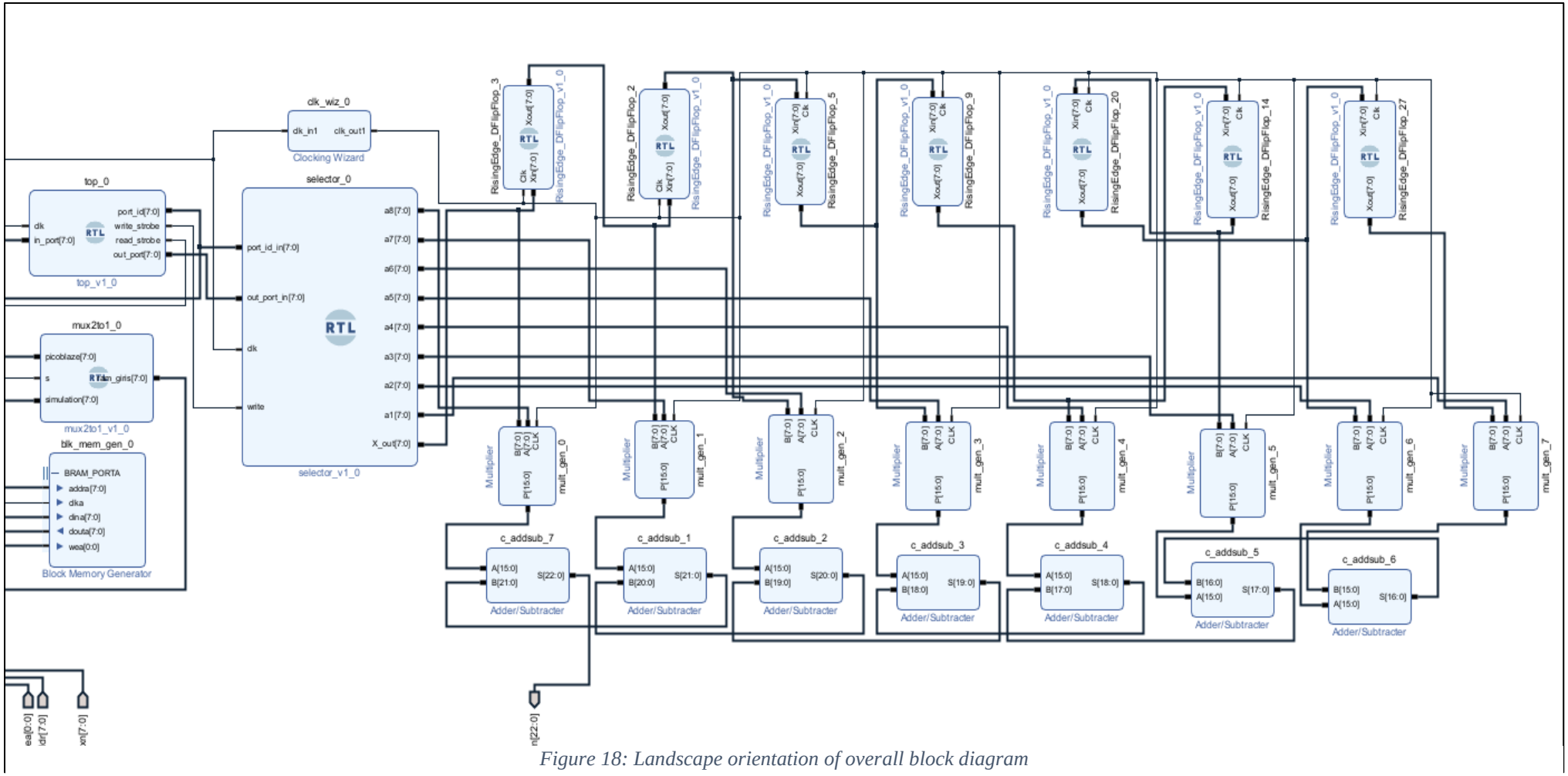


Figure 18: Landscape orientation of overall block diagram

The overall block diagram is presented in the Figure 18. All of the parts that are explained in this part can be seen on this figure. Although, we have done a little changes in the poisitions and orientations of the block so that they fit into one page. The leftmost of the image does not fit exactly; thus, some the connection cannot be seen on Figure 18. However, the connections of leftmost part is described in the first part of hardware section in Figure 7.

D.5) Summary of the section

Since FIR implementation is a datapath dominant design, this section is more important than the basic software section. Therefore, we give a brief summary of what is happening in the design shown in Figure 18.

First of all, the content of the BRAM is written via testbench file. The input values and the addresses of these input values are arbitrarily chosen in order to test the algorithm. After the contetnt of the BRAM is written, PicoBlaze wants to read the content from the BRAM; therefore, it sets the read_strobe high. Since read_strobe is tied to selector bit of the multiplexer in front of the BRAM, the address pin of BRAM is now tied to the port_id pin of PicoBlaze instead of simulation. Later on, the values are read from the BRAM, the filter coefficients and the input values are given to the selector block.

Secondly, the selector block gets these output values and their adresses, and by checking their adresses it directs the out_port to the corresponding output. For example, if the port_id value is between 0xff and 0xf8, that means PicoBlaze is sending the filter coefficients. On the other hand, if port_id value is 0x00, that mean PicoBlaze starts to send the input signal serially.

Thirdly, there should be a new and slower clock because PicoBlaze sends new input in every 16 clock cycle (actual clock). Therefore, we need to divide the frequency of actual clock by 16 and adjust the phases of these clock to ensure the synchronization. By feeding this clock to the FIR part, we made sure that all of the modules inside FIR part will work synchronously.

Finally, the last part is the most important part and the can be interpreted as the datapath of the processor. Here, the input array is shifted one clock cycle everytime it encounters a D Flip Flops. While being shifted through this chain of D Flip Flops, they are fed along with their corresponding coefficents to the multipliers. These multipliers multiply this shifted input and the coefficent and give the result after one clock cycle. Adding all of the outputs of the multipliers, we get the output array. Another representation of the algorithm can be seen in Figure 19.

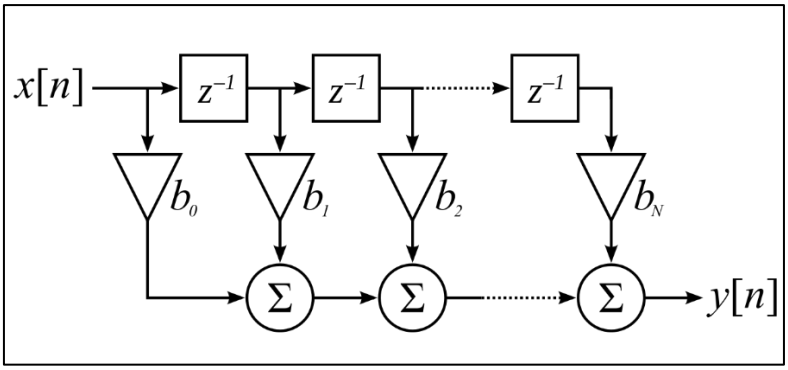


Figure 19: Another representation of the algorithm [5]

E) Time Diagrams and MATLAB verification

We explained all parts of the algorithm in previous two sections and their subsections. Now it is time to check the results of the algorithm.

```
>> x = [-1, 2, -3, 4, -5, 6, -7];
>> a = [-6, 3, 7, -11, -7, 1, -13, 5];
>> y = conv(x,a)

y =

     6   -15    17    -8     6    -5    17    14   -27   -37   140  -110   121   -35
```

Figure 20: MATLAB code for verification of the convolution operation

In Figure 20, the convolution function of MATLAB is used to check whether our algorithm works without errors. Here our x value x[n]= [-1 2 -3 4 -5 6 -7] as we showed in the testbench code in Figure 8 and our coefficients are a[n] =[-6 3 7 -11 -7 1 -13 5] as explained in the software section in Figure 2.

The resulting array is **y[n] = [6 -15 17 -8 6 -5 17 14 -27 -37 140 -110 121 -35].**

Now let's see if our algorithm does its job.

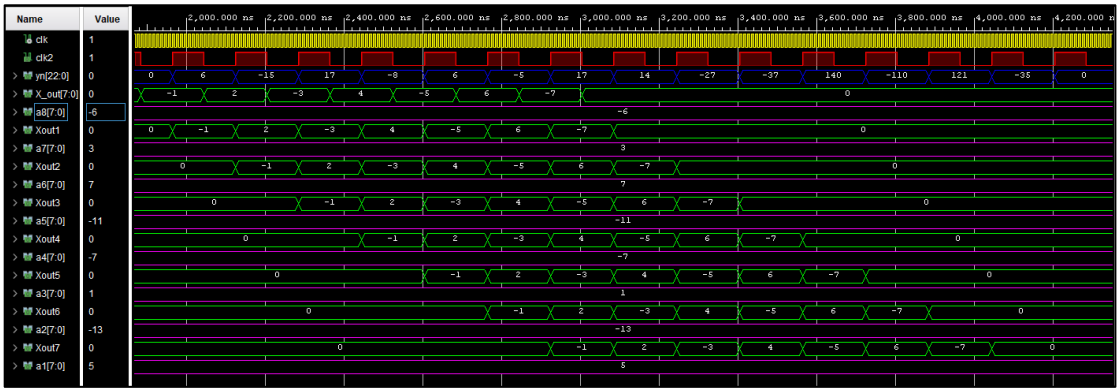


Figure 21: Time diagram of coefficients and inputs with their delayed objects

Here, in Figure 21, it can be seen that the convolution operation is realized correctly.

In this diagram the signal with blue color illustrates the output array y[n].

The yellow signal is the
actual 100 MHz clock and the red signal is the second clock with 6.25 MHz.

The signals with green color show the input signal and its delayed versions throughout the D Flip Flops chain.

Magenta color represents the filter coefficients and as it explained earlier, they are outputted parallelly, therefore they do not change over time.

```
>> a=[-6 3 7 -11 -7 1 -13 5];
>> xn = [-7 4 6 -2];
>> conv(a,xn)

ans =

    42   -45   -73   135    41  -115    75   -67   -60    56   -10
```

Figure 22: MATLAB code for verification of the convolution operation

We include another example to show the algorithm works for all inputs and all input sizes. Here, another input array example can be seen in Figure 22 and the result of the convolution is 11 elements long as expected.

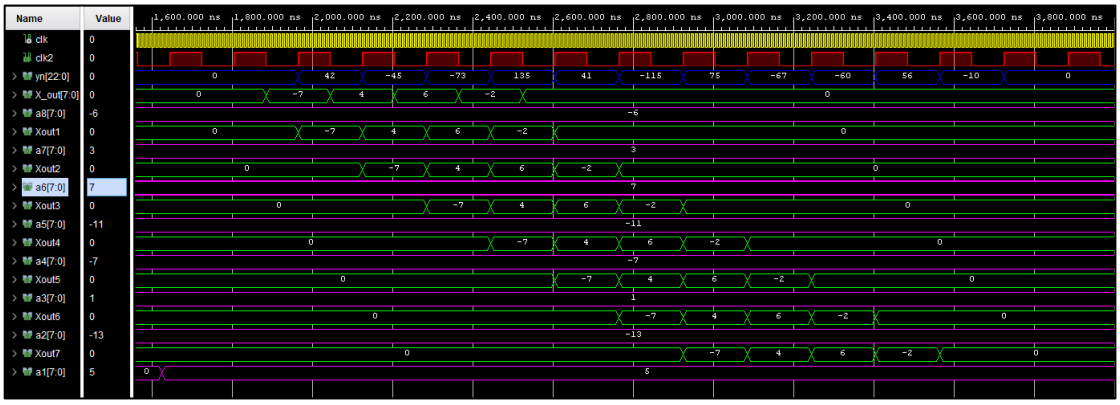


Figure 23: Time diagram of coefficients and inputs with their delayed objects

In this example, we did not change the coefficients. It is reasonable because the coefficients of a filter generally are not subject to change. Therefore, in this example of input array, we changed both the size of the input array and the elements of the array. As it can easily seen from the Figure 23, x[n] = [-7 4 6 -2], a[n] = [-6 3 7 -11 -7 1 -13 5]. The resulting array is **y[n] = [42 -45 -73 135 41 -115 75 -67 60 -37 56 -10].**

F) References

[1] A. V. Oppenheim and R. W. Schafer, Discrete-time signal processing, Third edition, Pearson new international edition. Harlow: Pearson, 2014.

[2] J. G. Proakis and D. G. Manolakis, Digital signal processing, Fourth edition, Pearson new international edition. Harlow: Pearson, 2014

[3] Xilinx, ”PicoBlaze 8-bit Embedded Microcontroller User Guide”, https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf

[4] “Two’s complement,” Wikipedia. Dec. 24, 2021. Accessed: Jan. 25, 2022. [Online]. Available:https://en.wikipedia.org/w/index.php?title=Two%27s_complement&oldid=1061841497

[5] “Finite impulse response,” Wikipedia. Jan. 11, 2022. Accessed: Jan. 25, 2022. [Online].Available:https://en.wikipedia.org/w/index.php?title=Finite_impulse_response&oldid=1064944393