Fall 2019

California State University, Northridge

Department of Electrical & Computer Engineering



Lab Experiment 7

*Design Space Exploration of FIR Filter*

Tumen Khureltulga, Juan Silva, Freddy Nunez

(Group 16)

November 12, 2019

ECE 524L

**Table of Contents**

**I. Introduction and Problem Statements…….………………………...…Page 2**

**II. Procedure……..…………………………………………….…………...Page 2**

**III. Testing Strategy………………………………….....………………......Page 5**

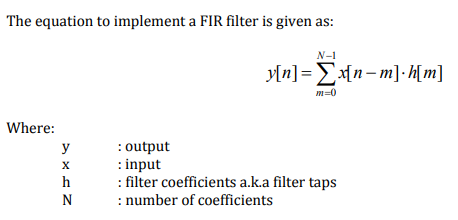
**IV. Results…………....…………………………………....…………….......Page 6**

**V. Analysis…………....……………………………....………………..……Page 7**

**VI. Appendix……………...………………………....………………….….Page 11**

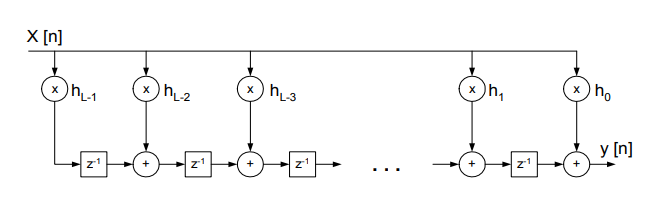
1. **Introduction and Problem Statements**

In this lab, we explore different ways of designing a digital low pass FIR filter. This filter is first modeled in Matlab in order for the coefficients to be generated. Then, the circuit is modeled on an FGPA using multiple designs. Design spaces are explored and design trade-offs are analyzed for area and performance. This FIR filter will be implemented using a transposed direct form, a serial form, and verified using Vivado’s IP core. Supplementary information on the FIR filter such as equations and specifications are provided in *Fig 7.1*. This filter must meet the following requirements: 20th order of 1.25 MHz pass band with a 5 Mhz clock rate. The taps are generated from Matlab and imported to Vivado via a COE file or a testbench.



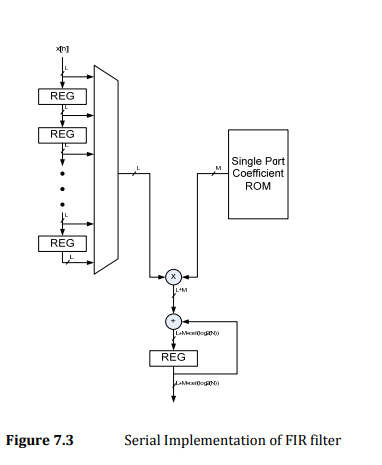
**Fig. 7.1** - Low-pass FIR Filter Implementation

1. **Procedure**

This experiment is broken down into multiple parts. The first part involves the use of Matlab to generate the taps for the filter coefficients. These coefficients are necessary because they are inputs for every design. The second part requires the implementation given the formula from *Fig. 7.1* using multiple multipliers, adders and registers. We first design the parallel implementation of the FIR filter shown in *Fig. 7.2*. This design is optimized for performance since the critical path delay is the output of 1 adder.

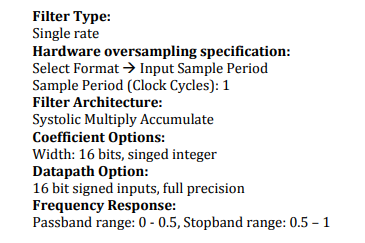
**Fig. 7.2 -** The transposed direct form of FIR filter

The transposed direct form of the FIR filter takes the input X and it is multiplied by all coefficient at once. At t=0 the first input of X[0] comes in and the product from X[0]\*h[19] is saved into the first array location in an array of 20x32. When t=1 new input X[1] comes in and the product from X[1]\*h[18] is added to the previously saved value, X[0]\*h[19]. The sum is saved to the next array location. The first output will become available after 20 clock cycles. This is how much time it takes to reach the last location of the array.   
 The h[m] is stored in 20 by 16 array. The y[n] gets stored in 20 by 33 array. It has “enable control signal called “en”. Every clock cycle, the design checks if enable is asserted. If en is not asserted, the design does not operate. If enable is asserted, every new value of x is multiplied by h[20] and it gets stored to the first row of the array, y[0]. It is only done for the first row. The other rows of y[n] are looped. The multiplication, addition, and shift operation are done inside the loop. The output is available when the value of the last row of y[n] array is shifted out.  
 The third part requires a serial implementation which uses a MUX, single-port ROM for the coefficients, and a multiplier that takes the product of outputs of the previous components. This data is accumulated and the output is available. The circuit shown in *Fig. 7.3*. This design is optimized for area since there are less components used to do the same operation.   
 The serial implementation of the FIR filter shown below takes the input X and it goes to the first location of the array and to the first input port of MUX. In the next clock cycle, an array is shifted and its output goes to the second input port of MUX. Therefore, MUX outputs the same value when t=0. The output of MUX will be the same for 20 clock cycles. This is because while the output of MUX stayed constant, the ROM is reading new values every clock cycles. MUX output and ROM output is multiplied, the product is saved in the register only at first. From thereon, the product is added to the previously saved value. This continues until all ROM values are read. Once this process is complete, the output will become ready.  
 The h[m] is stored in ROM of 20 by 16 array. The y[n] gets stored in 20 by 32 array. It has enable control signal called “en”. Every clock cycle, the design checks if enable is asserted. If en is not asserted, the design does not operate. If enable is asserted, the counter starts counting. It counts from 0 to 20. The counter output goes to the ROM address line and MUX select line. It reads h[m] from ROM and selects x[n] from MUX. The product of two values is sent to the accumulator, consisting of a 32-bit wide register and adder.



**Fig. 7.3 -** Serial Implementation of FIR filter

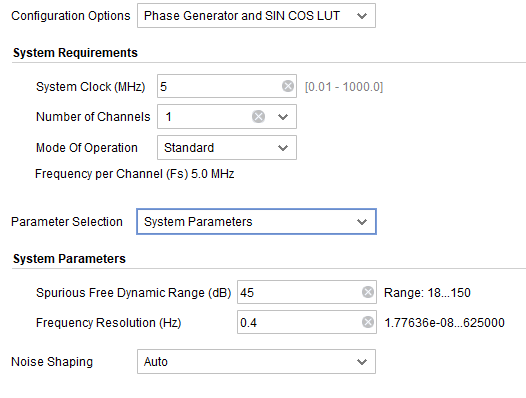
Finally, the fourth part takes advantage of Vivado’s tools to implement an FIR filter using the available IP Integrator. The FIR core parameters require specifications that are provided in *Fig 7.4.* The coefficient values are imported using a COE file and the sine wave inputs are generated using either a Direct Digital Synthesizer (DDS)or digitizing a sine wave input via a testbench. This implementation serves as verification to in order to compare between the two user-made implementations versus an established tool.



**Fig. 7.4 -** FIR Filter Specifications

1. **Testing Strategy**

The test bench is created such that it uses two Direct Digital Synthesizer (DDS), generates sine wave of 500kHz and 1.6 Mhz. For the IP core portion, a digitized sine wave input using the same parameters are implemented. Both DDS operate at system clock 5 MHz and have one channel. The parameter selection is chosen to be System Parameters so that output frequency can be set to the desired frequency. The most configurations are done in the configuration section of DDS. The configuration section of DDS options and summary are shown below in *Fig 7.5*.

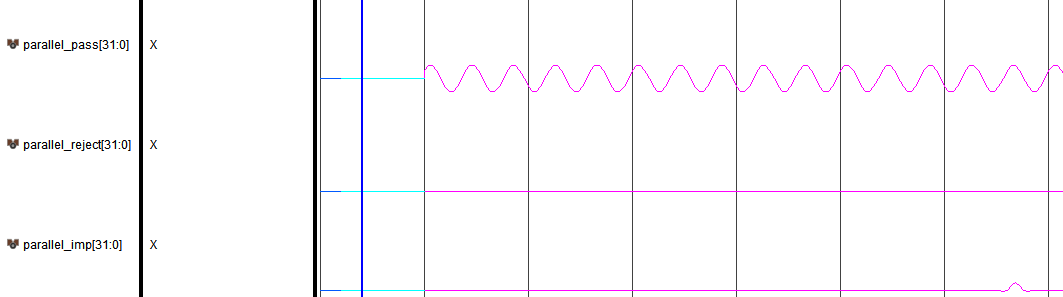


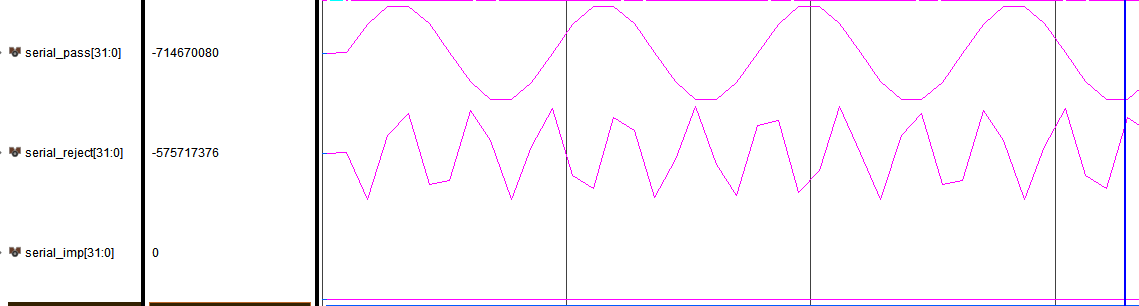
**Fig 7.5 -** Configuration and Summary section of DDS

The take out is if the number of channels increases, the frequency that each can handle goes down. For example, if the number of channels is 3, then each channel can do 5/3=1.66 Mhz at max. Since there are two DDS used in testbench, one not need to worry about this factor in this lab.  
 Once DDS is configured and initialized, the sine waves can be connected to the FIR filters. The output of the IP core generated FIR is compared with expected values generated in Matlab. A small deviation can be ignored. Then, IP core implementation can be used to verify the other two implementations because it is known that the IP core yield more solid result.

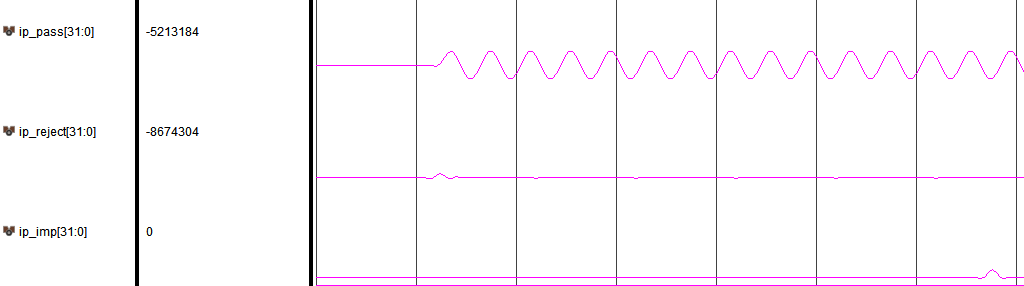
1. **Results & Data**

Simulated waveforms are shown in this section for all implementations.

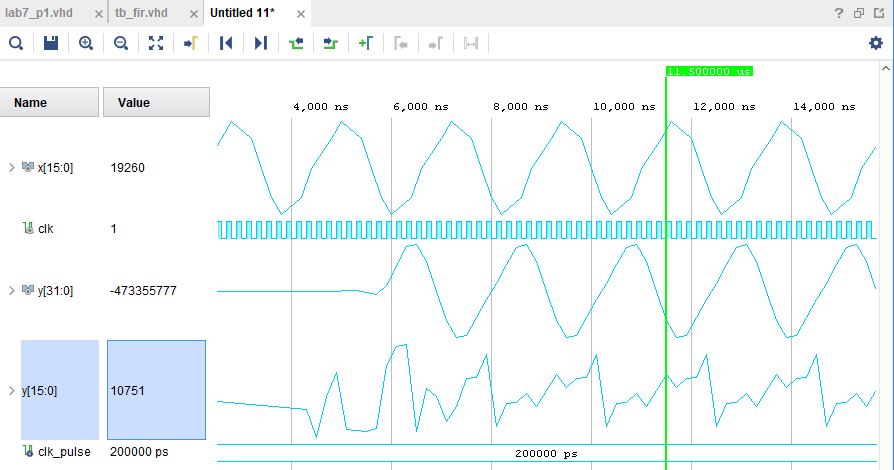
**  
Fig. 7.6** - The transposed direct form of FIR filter Waveforms



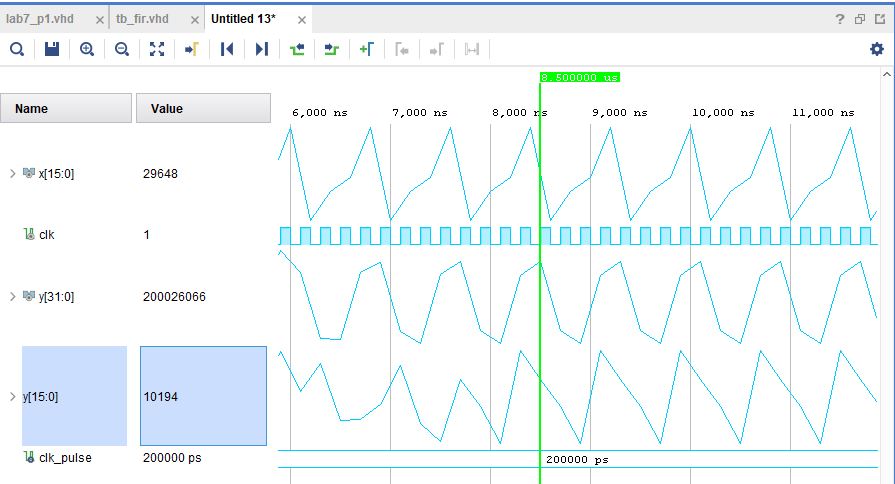
**Fig. 7.7 -** Serial Implementation of FIR filter Waveforms.



**Fig. 7.8 -** FIR Compiler Waveforms



**Fig. 7.9 -** FIR Catalog Results - 500 KHz Signal Passed

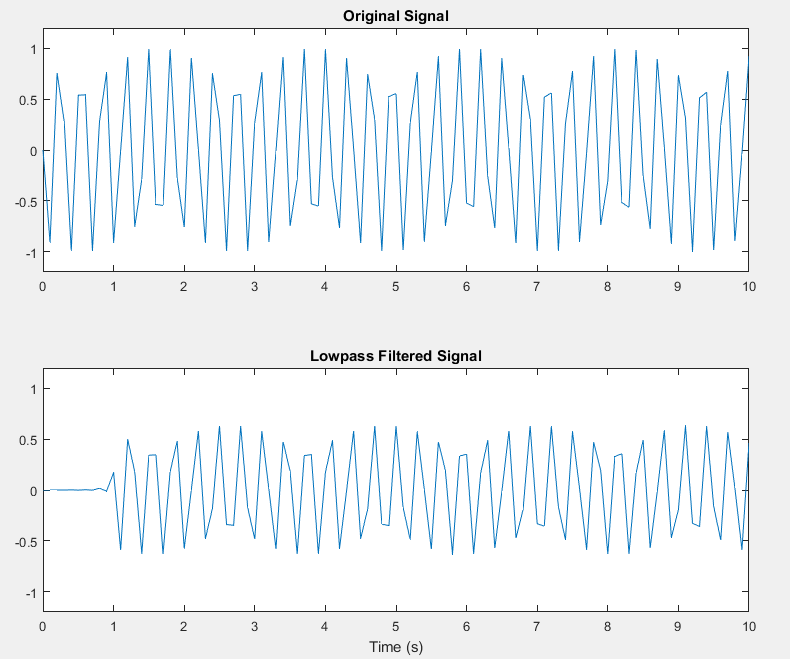


**Fig. 7.10 -** FIR Catalog Results - 1.6 MHz Signal Partially Blocked

1. **Analysis**

Notice that in Fig 4.2 sine wave of 1.6 MHz is not filtered out. In fact, the circuit is doing what it is meant to do. Fig 5.1 shows the output of the designed FIR filter in Matlab for Sine input of 1.6 MHz. Instead of filtering out, the filter attenuates the signal in this case. The Matlab filter as follows

| Fs = 10;  amp = 1;  f = 160000000;  phase = 0;  Ts = 1/Fs;  t = 0:Ts:10;  y = amp\*sin((2\*3.14\*f\*t) + phase);  mlo = [0 0.5 0.5 1];  blo = fir2(20,f,mlo);  outlo = filter(blo,1,y);  subplot(2,1,1) | plot(t,y)  title('Original Signal')  ylim([-1.2 1.2])  subplot(2,1,2)  plot(t,outlo)  title('Lowpass Filtered Signal')  xlabel('Time (s)')  ylim([-1.2 1.2]) |
| --- | --- |



**Figure 5.1** - Matlab Generated Waveform for 1.6 Mhz.

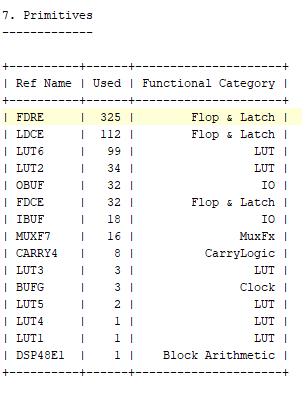
Comparing the input value and output for serial rejection, it is found that the original signal lost two-third of its power. In other words, the magnitude dropped by ⅔. In Matlab generated waveform, the signal attenuates one half of its signal strength.

In addition, the serial implementation does not sample the input every clock cycle like the parallel case. This serial implementation samples every 21 clock cycles. Therefore, it is justified to say the serial implementation does what intents to do. The output updated every 21 clock cycles. On the other hand, the parallel output is updated every clock cycle except the initial output.

Ip core generated waveform looks identical to the parallel case. All implementation successfully filters out the impulse frequency.

After synthesizing, the implemented circuit for parallel is made of 11 DSP blocks and 64 registers. There are 21 tabs. Each DSP block is used for 2 tabs.

On the other hand, the serial implementation used various resources, shown below in Fig 5.2. Notice that it used only one DSP block.

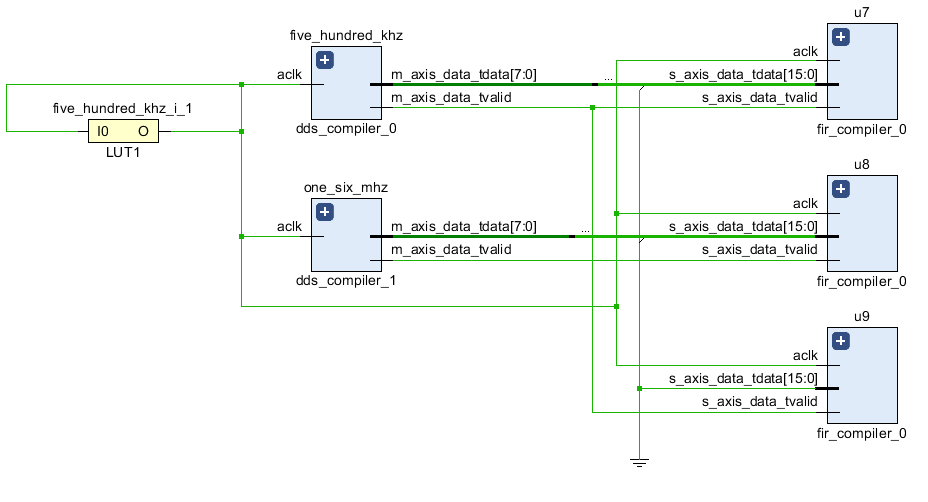


**Figure 5.2** - Serial Implementation Report Utilization.

Lastly, it takes 329 LUTs, 581 Registers, and 11 DSP blocks to implement FIR Compiler. The

Comparing all the models in terms of resource usage, serial implementation definitely uses fewer resources. The parallel is next and the FIR Compiler IP Core uses the most resources. It shows that even tough IP cores are easy to use, it can consume a huge amount of resources compared to VHDL coded modules.

In terms of speed, the FIR compiler and Parallel case are the same. In the serial case, speed is given up in order to save resources. Overall, the parallel is the winner because it is as fast as FIR Compiler and it uses fewer resources. In Webpack version of Xilinx, the FIR compiler takes 5 to 10 min to simulate.



**Figure 5.2** -IP Core generated FIR filter.

1. **Appendix**

| **Parallel** | **Serial** |
| --- | --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity parallel is  generic( ADDR:integer:=5;  WIDTH:integer:=16);  Port ( x:in signed (WIDTH-1 downto 0);  data\_out:out signed(2\*WIDTH-1 downto 0);  clk,en:in std\_logic);  end parallel;  architecture Behavioral of parallel is  type ROM is array (0 to 20) of std\_logic\_vector(WIDTH-1 downto 0);  type RAM2 is array (0 to 20) of std\_logic\_vector(2\*WIDTH-1 downto 0);  signal h: ROM;  signal y:RAM2;    begin  h <= (0 => x"0000", -- initializes first 5 locations in RAM  1 => x"FFAF",  2 => x"0000",  3 => x"FEE7",  4 => x"0000",  5 => x"031D",  6 => x"0000",  7 => x"F822",  8 => x"0000",  9 => x"1C9B",  10 => x"2E00",  11 => x"1C9B",  12 => x"0000",  13 => x"F822",  14 => x"0000",  15 => x"031D",  16 => x"0000",  17 => x"FEE7",  18 => x"0000",  19 => x"FFAF",  20 => x"0000");    process(clk)--shift register  variable temp\_y,temp2:signed(2\*WIDTH-1 downto 0);  variable temp\_prod:signed(2\*WIDTH-1 downto 0);  variable temp:signed(2\*WIDTH-1 downto 0);  variable temp\_h,temp\_x:signed(WIDTH-1 downto 0);  begin  if clk'event and clk='1' then  if (en='1') then  temp:=x\*signed(h(20));  y(0)<=std\_logic\_vector(temp);  for i in 1 to 20 loop  temp\_h:=signed(h(20-i));  temp\_y:=signed(y(i-1));  temp\_x:=x;  temp\_prod:=temp\_x\*temp\_h;  temp\_y:=temp\_prod+temp\_y;  y(i)<=std\_logic\_vector(temp\_y);  end loop;  data\_out<=signed(y(20));  end if;  end if;  end process;  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity serial is  generic( ADDR:integer:=5;  WIDTH:integer:=16);  Port ( x:in signed (WIDTH-1 downto 0);  data\_out:out signed(2\*WIDTH-1 downto 0);  clk,en:in std\_logic);  end serial;  architecture Behavioral of serial is  COMPONENT blk\_mem\_gen\_0  PORT (  clka : IN STD\_LOGIC;  addra : IN STD\_LOGIC\_VECTOR(4 DOWNTO 0);  douta : OUT STD\_LOGIC\_VECTOR(15 DOWNTO 0)  );  END COMPONENT;  signal addra:STD\_LOGIC\_VECTOR(ADDR-1 DOWNTO 0):=(others=>'0');  signal h,xn:STD\_LOGIC\_VECTOR(WIDTH-1 DOWNTO 0);  signal xh:signed(2\*WIDTH-1 DOWNTO 0);  signal sum\_reg:signed(2\*WIDTH-1 DOWNTO 0);  signal sum:signed(2\*WIDTH-1 DOWNTO 0):=(others=>'0');  signal shift\_cnt:unsigned(ADDR-1 DOWNTO 0):=(others=>'0');  type RAM is array (0 to 20) of std\_logic\_vector(WIDTH-1 downto 0);  signal MEM:RAM;  begin  addra<=std\_logic\_vector(shift\_cnt); --rom addr  output:process(clk)  begin  if en='1' then  if (shift\_cnt=1) then  data\_out<=(sum\_reg);  end if;  end if;  end process;  add:process(sum\_reg, xh) --addition  begin  if en='1' then  sum<=sum\_reg+xh;  end if;  end process;  multiply:process(h,xn) --multiplication  begin  if en='1' then  xh<=signed(h)\*signed(xn);  end if;  end process;  ROM : blk\_mem\_gen\_0  PORT MAP (  clka => clk,  addra => addra,  douta => h  );      accumulator:process(clk)  begin  if en='1' then  if shift\_cnt=2 then  sum\_reg<=(others=>'0'); --reset  else  if clk'event and clk='1' then  sum\_reg<=sum; --sample  end if;  end if;  end if;  end process;        shift\_reg:process(clk) --sample x  variable temp,temp1:STD\_LOGIC\_VECTOR(WIDTH-1 downto 0);  begin  if clk'event and clk='1' then  temp:=std\_logic\_vector(x);  MEM(0)<=temp;  for i in 1 to 20 loop  MEM(i)<=MEM(i-1);  end loop;  end if;  end process;    mux: process(shift\_cnt)  begin    case shift\_cnt is  when "00001" =>  xn <= MEM(0);  when "00010" =>  xn <= MEM(1);  when "00011" =>  xn <= MEM(2);  when "00100" =>  xn <= MEM(3);  when "00101" =>  xn <= MEM(4);  when "00110" =>  xn <= MEM(5);  when "00111" =>  xn <= MEM(6);  when "01000" =>  xn <= MEM(7);  when "01001" =>  xn <= MEM(8);  when "01010" =>  xn <= MEM(9);  when "01011" =>  xn <= MEM(10);  when "01100" =>  xn <= MEM(11);  when "01101" =>  xn <= MEM(12);  when "01110" =>  xn <= MEM(13);  when "01111" =>  xn <= MEM(14);  when "10000" =>  xn <= MEM(15);  when "10001" =>  xn <= MEM(16);  when "10010" =>  xn <= MEM(17);  when "10011" =>  xn <= MEM(18);  when "10100" =>  xn <= MEM(19);  when others =>  xn <= xn; --wanna do high z  end case;  end process;      addr\_mux\_select:process(clk) --addr generation and mux control  begin  if clk'event and clk='1' then  if (en='1') then  shift\_cnt <= "00001";  if(shift\_cnt /= 20) then  shift\_cnt <= shift\_cnt + 1;  else  shift\_cnt <= "00000";  end if;  end if;  end if;  end process;  end Behavioral; |
| **Test Bench** |  |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity general\_tb is  -- Port ( );  end general\_tb;  architecture Behavioral of general\_tb is  COMPONENT dds\_compiler\_0  PORT (  aclk : IN STD\_LOGIC;  m\_axis\_data\_tvalid : OUT STD\_LOGIC;  m\_axis\_data\_tdata : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0)  );  END COMPONENT;  COMPONENT dds\_compiler\_1  PORT (  aclk : IN STD\_LOGIC;  m\_axis\_data\_tvalid : OUT STD\_LOGIC;  m\_axis\_data\_tdata : OUT STD\_LOGIC\_VECTOR(7 DOWNTO 0)  );  END COMPONENT;  COMPONENT parallel  generic( ADDR:integer:=5;  WIDTH:integer:=16);  Port ( x:in signed (WIDTH-1 downto 0);  data\_out:out signed(2\*WIDTH-1 downto 0);  clk,en:in std\_logic);  end component parallel;  COMPONENT serial  generic( ADDR:integer:=5;  WIDTH:integer:=16);  Port ( x:in signed (WIDTH-1 downto 0);  data\_out:out signed(2\*WIDTH-1 downto 0);  clk,en:in std\_logic);  end component serial;  COMPONENT fir\_compiler\_0  PORT (  aclk : IN STD\_LOGIC;  s\_axis\_data\_tvalid : IN STD\_LOGIC;  s\_axis\_data\_tready : OUT STD\_LOGIC;  s\_axis\_data\_tdata : IN STD\_LOGIC\_VECTOR(15 DOWNTO 0);  m\_axis\_data\_tvalid : OUT STD\_LOGIC;  m\_axis\_data\_tdata : OUT STD\_LOGIC\_VECTOR(31 DOWNTO 0)  );  END COMPONENT;  constant PERIOD:time:=200 ns;  constant ADDR:integer:=5;  constant WIDTH:integer:=16;  signal clk,m\_axis\_data\_tvalid\_0,m\_axis\_data\_tvalid\_1:std\_logic;  signal m\_axis\_data\_tdata\_0,m\_axis\_data\_tdata\_1 :STD\_LOGIC\_VECTOR(WIDTH/2-1 DOWNTO 0);  signal parallel\_pass,parallel\_reject,parallel\_imp :signed(2\*WIDTH-1 DOWNTO 0);  signal serial\_pass,serial\_reject,serial\_imp :signed(2\*WIDTH-1 DOWNTO 0);  signal ip\_pass,ip\_reject,ip\_imp :STD\_LOGIC\_VECTOR(31 DOWNTO 0);  signal x\_0,x\_1,x\_2:signed(WIDTH-1 downto 0);  signal x\_ip\_pass,x\_ip\_reject:STD\_LOGIC\_VECTOR(15 downto 0);  begin  x\_0<=signed(m\_axis\_data\_tdata\_0 & "00000000");  x\_1<=signed(m\_axis\_data\_tdata\_1 & "00000000");  x\_ip\_pass<=STD\_LOGIC\_VECTOR(m\_axis\_data\_tdata\_0 & "00000000");  x\_ip\_reject<=STD\_LOGIC\_VECTOR(x\_1); | process --clock generation  begin  clk<='1';  wait for PERIOD/2;  clk<= not clk;  wait for PERIOD/2;  end process;  process --step function  begin  x\_2<= (others=>'0');  wait for 1000 ns;  x\_2<=x"7fff";  wait for PERIOD/10;  x\_2<= (others=>'0');  wait for 2000 ns;  end process;  five\_hundred\_khz : dds\_compiler\_0  PORT MAP (  aclk => clk,  m\_axis\_data\_tvalid => m\_axis\_data\_tvalid\_0,  m\_axis\_data\_tdata => m\_axis\_data\_tdata\_0  );  one\_six\_mhz : dds\_compiler\_1  PORT MAP (  aclk => clk,  m\_axis\_data\_tvalid => m\_axis\_data\_tvalid\_1,  m\_axis\_data\_tdata => m\_axis\_data\_tdata\_1);  u1:parallel  Port map ( x=>x\_0,data\_out=>parallel\_pass,clk=>clk,en=>m\_axis\_data\_tvalid\_0); -- 500khz  u2:parallel  Port map ( x=>x\_1,data\_out=>parallel\_reject,clk=>clk,en=>m\_axis\_data\_tvalid\_1); -- 1.6 mhz  u3:parallel  Port map ( x=>x\_2,data\_out=>parallel\_imp,clk=>clk,en=>m\_axis\_data\_tvalid\_1); -- impulse response  u4:serial  port map( x=>x\_0,data\_out=>serial\_pass,clk=>clk,en=>m\_axis\_data\_tvalid\_0); --500kh  u5:serial  port map( x=>x\_1,data\_out=>serial\_reject,clk=>clk,en=>m\_axis\_data\_tvalid\_1); --1.6 mhz  u6:serial  port map( x=>x\_2,data\_out=>serial\_imp,clk=>clk,en=>m\_axis\_data\_tvalid\_1); - impulse  u7 : fir\_compiler\_0  PORT MAP (  aclk => clk,  s\_axis\_data\_tvalid => m\_axis\_data\_tvalid\_0,  s\_axis\_data\_tdata => x\_ip\_pass,  m\_axis\_data\_tdata => (ip\_pass)  );  u8 : fir\_compiler\_0  PORT MAP (  aclk => clk,  s\_axis\_data\_tvalid => m\_axis\_data\_tvalid\_1,  s\_axis\_data\_tdata => x\_ip\_reject,  m\_axis\_data\_tdata => (ip\_reject)  );  u9 : fir\_compiler\_0  PORT MAP (  aclk => clk,  s\_axis\_data\_tvalid => m\_axis\_data\_tvalid\_0,  s\_axis\_data\_tdata => std\_logic\_vector(x\_2),  m\_axis\_data\_tdata => (ip\_imp)  );  end Behavioral; |

| **FIR Compiler Config** | **Testbench (DDS Config)** |
| --- | --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.NUMERIC\_STD.ALL;  entity FIR\_ip is  Port ( x : in STD\_LOGIC\_VECTOR (15 downto 0);  clk : in STD\_LOGIC;  y : out STD\_LOGIC\_VECTOR (31 downto 0));  end FIR\_ip;  architecture Behavioral of FIR\_ip is  --Instantiate FIR Filter  COMPONENT fir\_compiler\_0  PORT (  aclk : IN STD\_LOGIC;  s\_axis\_data\_tvalid : IN STD\_LOGIC; --harcode to '1'  s\_axis\_data\_tready : OUT STD\_LOGIC; --leave open  s\_axis\_data\_tdata : IN STD\_LOGIC\_VECTOR(15 DOWNTO 0); --input h  m\_axis\_data\_tvalid : OUT STD\_LOGIC; --leave open or hardcode to '1'  m\_axis\_data\_tdata : OUT STD\_LOGIC\_VECTOR(31 DOWNTO 0) --otuput y;  );  END COMPONENT;  signal aclk : STD\_LOGIC := '0';  signal s\_tready : STD\_LOGIC;  signal s\_tvalid, m\_tvalid : STD\_LOGIC;  signal s\_tdata : STD\_LOGIC\_VECTOR(15 downto 0);  signal m\_tdata : STD\_LOGIC\_VECTOR(31 downto 0);  begin  s\_tvalid <= '1';  UUT : fir\_compiler\_0  PORT MAP (  aclk => clk,  s\_axis\_data\_tvalid => s\_tvalid,  s\_axis\_data\_tready => s\_tready,  s\_axis\_data\_tdata => x,  m\_axis\_data\_tvalid => m\_tvalid,  m\_axis\_data\_tdata => y  );  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.NUMERIC\_STD.ALL;  entity tb\_FIRip is  -- Port ( );  end tb\_FIRip;  architecture Behavioral of tb\_FIRip is  component FIR\_ip is  Port ( x : in STD\_LOGIC\_VECTOR (15 downto 0);  clk : in STD\_LOGIC;  y : out STD\_LOGIC\_VECTOR (31 downto 0));  end component;  constant clk\_pulse : time := 200 ns;  signal x : std\_logic\_vector(15 downto 0);  signal clk : std\_logic := '0';  signal y : std\_logic\_vector(31 downto 0);  begin  UUT: FIR\_ip  port map (x => x, clk => clk, y => y);  --Clock process  clock: process(clk)  begin  clk <= not clk after clk\_pulse / 2;  end process clock;  -- --500 Kz passband frequency response  -- din\_500Khz: process  -- begin  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(0,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(19260,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(31163,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(31163,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(19260,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(0,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(-19260,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(-31163,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(-31163,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(-19260,16));  -- wait for clk\_pulse;  -- x <= std\_logic\_vector(to\_signed(0,16));  -- end process;  --1.6 Mhz passband frequency response  din\_1\_6Mhz: process  begin  wait for clk\_pulse;  x <= std\_logic\_vector(to\_signed(0,16));  wait for clk\_pulse;  x <= std\_logic\_vector(to\_signed(29648,16));  wait for clk\_pulse;  x <= std\_logic\_vector(to\_signed(-25247,16));  wait for clk\_pulse;  x <= std\_logic\_vector(to\_signed(-8149,16));  end process;  end Behavioral; |