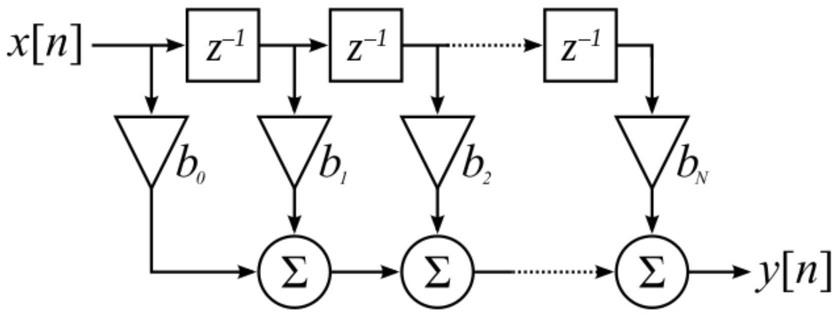
Introduction to FIR Filter:-

The term FIR abbreviation is "Finite Impulse Response" and it is one of two main types of digital filters used in DSP applications., it allows AC components and blocks DC components. The best example of the filter is a phone line, which acts as a filter. Because, it limits frequencies to a range significantly smaller than the range of human beings can hear frequencies.

A digital filter takes a digital input, gives a digital output, and consists of digital components. In a typical digital filtering application, software running on a digital signal processor (DSP) reads input samples from an A/D converter, performs the mathematical manipulations dictated by theory for the required filter type, and outputs the result via a D/A converter.

There are many filter types, but the most common are lowpass, highpass, bandpass, and bandstop.
A lowpass filter allows only low frequency signals below some specified cutoff through to its output, so it can be used to eliminate high frequencies. A low pass filter is handy, in that regard, for limiting the uppermost range of frequencies in an audio signal. It's the type of filter that a phone line uses.Digital FIR filters have many favourable properties, which is why they are extremely popular in digital signal processing.

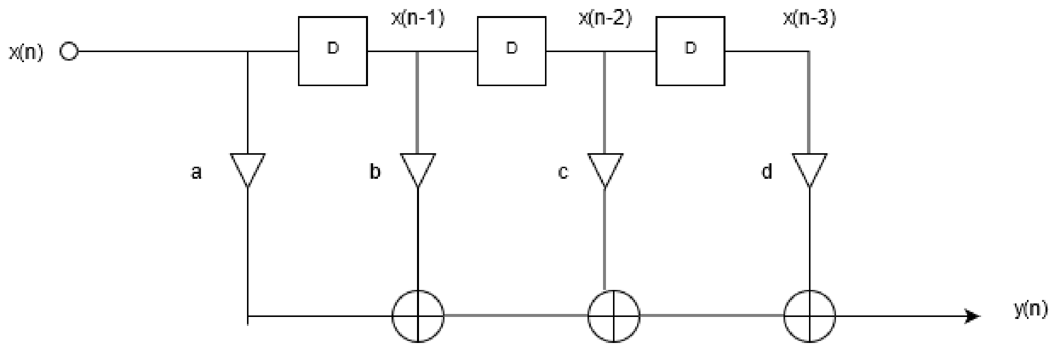


$$\boxed{y(n) = \sum_{k=0}^{M-1} b_k x(n-k) = \sum_{k=0}^{M-1} h(k)\, x(n-k)}$$

Architecture for implementation:-

Let’s consider a 4 Tap FIR Filter as shown below. The critical path or minimum time required for processing a new sample is limited by 1 multiple and 3 addition time. If the time taken for multiplication is T_M and T_A is the time taken for addition then the sample period is give by,

$$T_{Sample} >= T_M + 3\,T_A$$



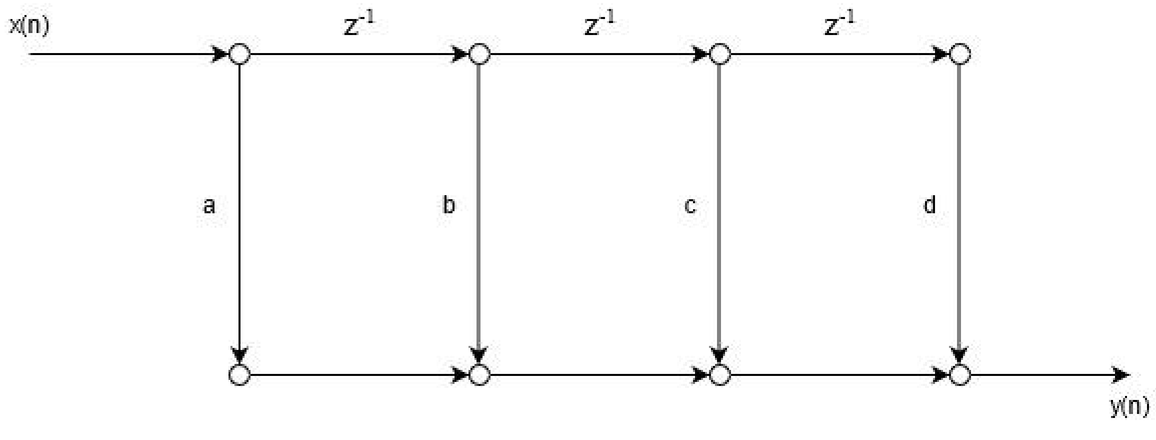
4 Tap FIR Filter

Therefore the sampling frequency is given by

$$f_{Sample} =< 1/(T_M + 3\,T_A)$$

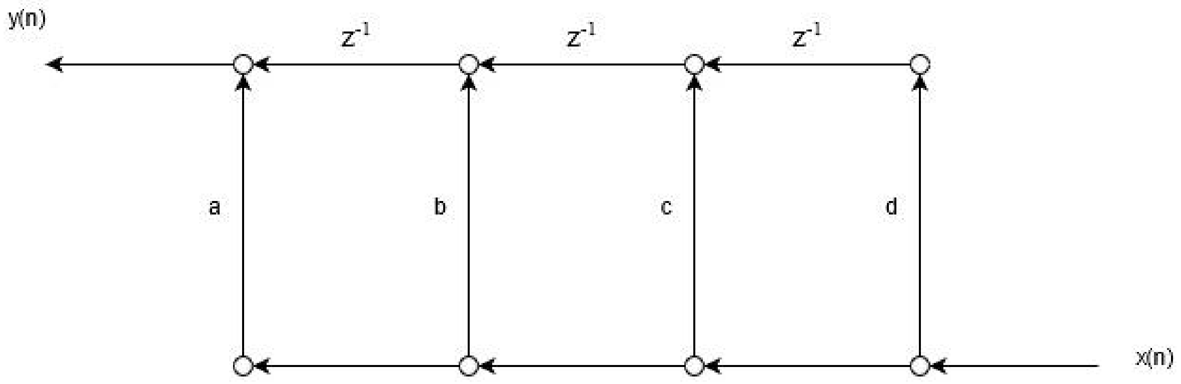
The above Direct form can only be used when the above equation is satisfied. When some real time applications need a faster input rate (sampling rate) then this structure can not be used.

The Signal Flow Graph of a 4tap FIR filter can be drawn as below,



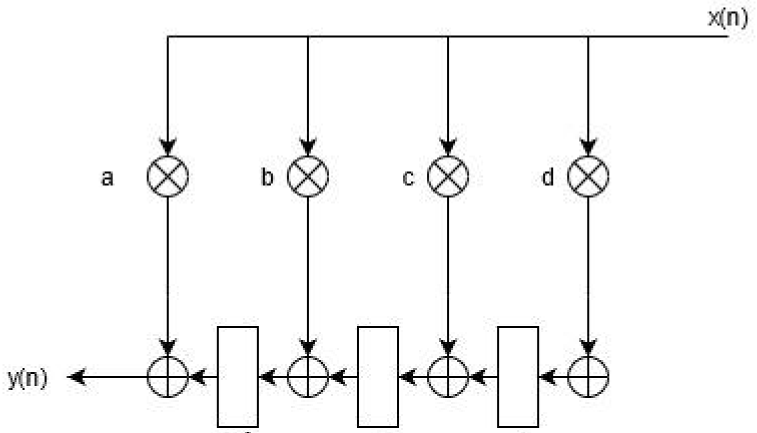
Transposition Theorem state that “Reversing the direction of all the edges of a signal flow graph (SFG) and interchanging the input and output ports preserves the functionality of the system ”

So we can change and write a new signal flow graph for the FIR Filter Direct form as below,

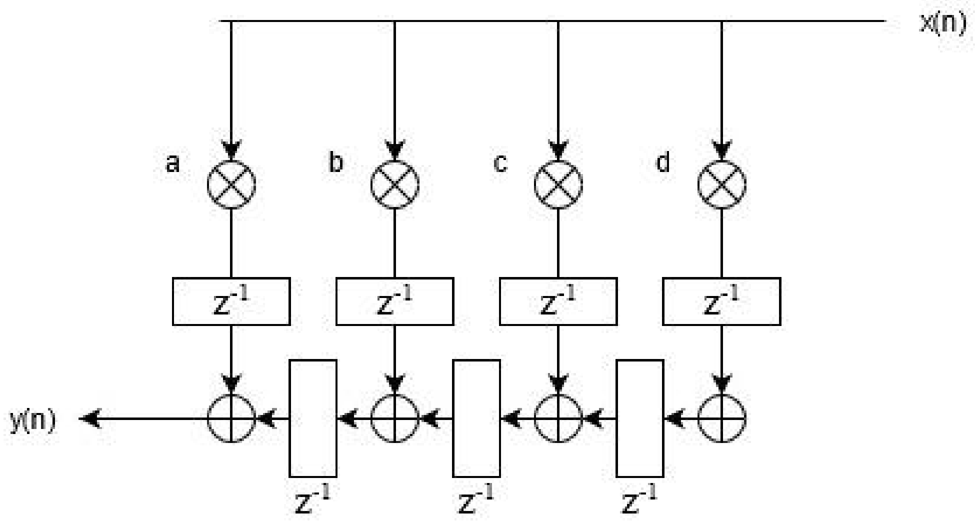


This is also known as Transposed SFG representation of FIR Filter. This leads to Data Broadcast structure where data are not stored but are broadcast to all the multipliers at the same time.

The 4 Tap Broadcast structure is shown below,



In this DBS we have a critical path of $T_M + T_A$. The sampling rate is increased. Now to further improve it we can use pipelined DBS in which the critical path is reduced to Max(T_M , T_A). But in pipelined DBS we will have latency of 1 Clock Cycle. The pipelined DBS is shown below for a 4 Tap Low pass FIR Filter.



Above FIR Filter structure is final Architecture for implementation in 4 Tap FIR Filter.

Filter Problem Specification:-

Design a Low Pass Pipelined FIR filter using Window Method having pass-band edge frequency 2 KHz and stop-band edge frequency 2.2 KHz. Filter also have Stop band attenuation of 50dB and sampling frequency is equal to 10 KHz.

Pass Band = 2.0KHz
Stop Band = 2.2KHz
Sampling frequency = 10KHz

And with Stop band attenuation of 50dB we will get N=16.

So we need to design a 16 Tap Low Pass FIR Filter and for maximum frequency of operation we will use Pipelined Data Broadcast Structure.

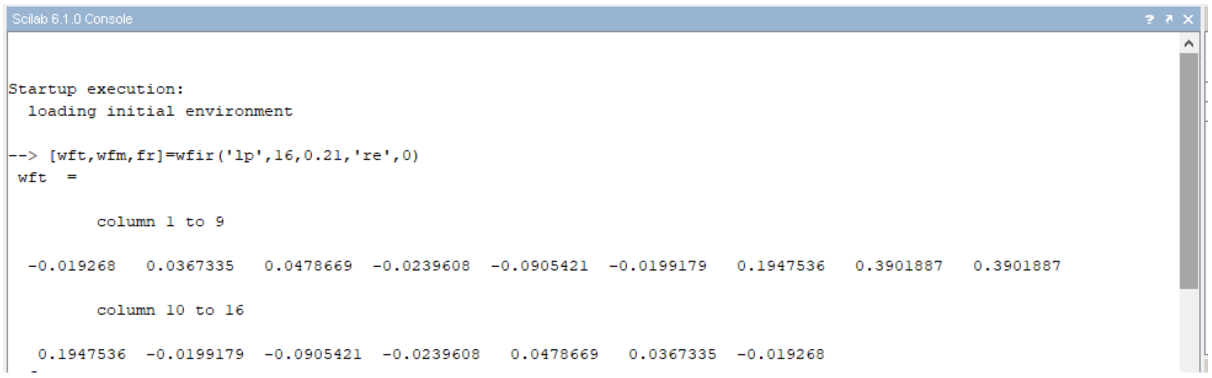
Filter coefficients: -

To find the filter coefficients we can use the Open **Source tool SciLab.**
In the SciLab command line we need a specific command to find the filter coefficients. This is shown below,

[wft,wfm,fr]=wfir('lp',16,0.21,'re',0)

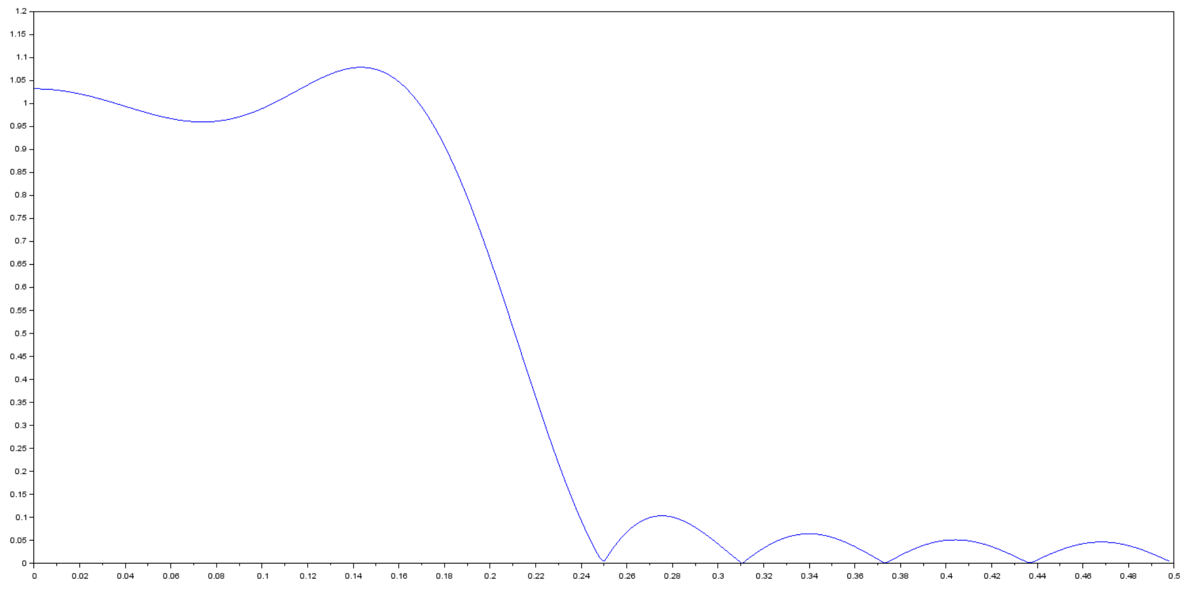
wft: A vector containing the windowed filter coefficients for a filter of length n.
wfm: A vector of length 256 containing the frequency response of the windowed filter.
fr: A vector of length 256 containing the frequency axis values associated to the values contained in wfm.

‘lp’	Low Pass
16	Tap Filter
0.21	normalised Frequency
‘re’	Rectangular Window



```
Startup execution:
  loading initial environment

--> [wft,wfm,fr]=wfir('lp',16,0.21,'re',0)
 wft  =

         column 1 to 9

  -0.019268   0.0367335   0.0478669  -0.0239608  -0.0905421  -0.0199179   0.1947536   0.3901887   0.3901887

         column 10 to 16

   0.1947536  -0.0199179  -0.0905421  -0.0239608   0.0478669   0.0367335  -0.019268
```

Magnitude Response: -

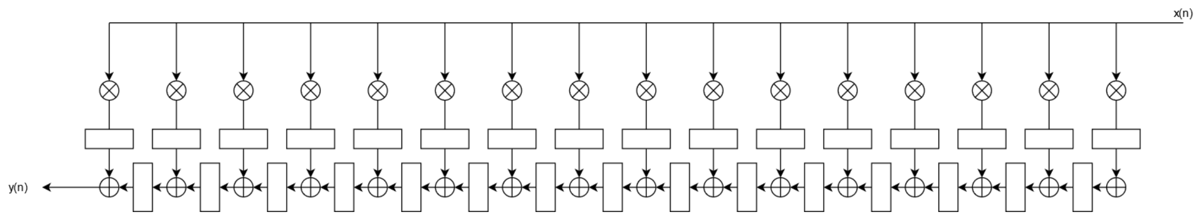


All the filter coefficients are less than 0 and are using 16 bits to represent them so we can assign 15 bits for fractional part and 1 bit for sign part and use Q1.15 Fixed Point format to represent them.
All the filter coefficients are shown below.

h(0) , h(15)	-0.019268	1.111110110001001
h(1) , h(14)	0.0367335	0.000010010110011
h(2) , h(13)	0.0478669	0.000011000100000
h(3) , h(12)	-0.0239608	1.111110011101111
h(4) , h(11)	-0.0905421	1.111010001101010
h(5) , h(10)	-0.0199179	1.111110101110100
h(6) , h(9)	0.1947536	0.001100011101101
h(7) , h(8)	0.3901887	0.011000111110001

Filter Architecture: -

From above analysis we found we need a 16 Tap Fir Filter to implement our Filter specified above. To maximize the frequency of operation we need to use pipelined DBS as shown below.



Relation between input and output for the above structure can be given by

$$y(n) = \sum_{K=0}^{15} h(k) * x(n-k)$$

Verilog model for filter architecture:-

```
module filter_16tap(clk,din ,dout ,reset);
   input clk,reset;
   input signed [15:0]din;
   output signed [31:0]dout;

   parameter depth=16;



   wire signed [31:0]am[15:0];              //multiplier output wire
   wire signed [31:0]add_out[15:0];    //adder output wire
   wire signed [31:0]q[15:0];                    // latch or ff output wire
   wire signed [31:0]bm[15:0];                 // pipelined register

   assign q[0]=31'd0;

   // filter coefficients

   wire signed [15:0]c[15:0];

   assign  c[0]=16'b1_111110110001001;                    //-0.019268
   assign  c[1]=16'b0_000010010110011;                    //0.0367335
   assign  c[2]=16'b0_000011000100000;                    //0.0478669
   assign  c[3]=16'b1_111110011101111;                    //-0.0239608
   assign  c[4]=16'b1_111010001101010;                    //-0.0905421
   assign  c[5]=16'b1_111110101110100;                    //-0.0199179
   assign  c[6]=16'b0_001100011101101;                    //0.1947536
   assign  c[7]=16'b0_011000111110001;                    //0.3901887
   assign  c[8]=16'b0_011000111110001;                    //0.3901887
   assign  c[9]=16'b0_001100011101101;                    //0.1947536
   assign  c[10]=16'b1_111110101110100;          //-0.0199179
   assign  c[11]=16'b1_111010001101010;          //-0.0905421
   assign  c[12]=16'b1_111110011101111;          //-0.0239608
   assign  c[13]=16'b0_000011000100000;          //0.0478669
   assign  c[14]=16'b0_000010010110011;          //0.0367335
   assign  c[15]=16'b1_111110110001001;          //-0.019268




   // multipliers


   genvar i;
   generate
          for(i=0;i<depth;i=i+1)    begin
          multi mt(am[i],din,c[i]);


          end
   endgenerate


   // adder


   genvar j;
   generate
                 for(j=0;j<depth;j=j+1)    begin
                 assign add_out[j]=bm[j]+q[j];


                        end
   endgenerate


   // latches
```

```
   genvar k;
   generate
                 for(k=0;k<depth-1;k=k+1)    begin
                 DFF dff(clk,reset,add_out[k],q[k+1]);
                 end
   endgenerate

         genvar e;
   generate
                 for(e=0;e<depth;e=e+1)    begin
                 DFF dff(clk,reset,am[e],bm[e]);
                 end
   endgenerate

   // output assignment
   assign dout=add_out[15];

endmodule



// latch
module DFF (clk,reset,d,q);
   input clk,reset;
   input signed  [31:0]d;
   output reg signed  [31:0]q;

   always @(posedge clk) begin
          if(reset)
          q<=31'd0;
          else
          q<=d;
   end
endmodule



// multiplication
module multi (m,a,b);
   input signed  [15:0] a,b;
   output signed [31:0] m;
   assign m=a*b;
endmodule




module testbench();
   parameter SF1=2.0**-8.0;
   parameter SF2=2.0**-23.0;

   reg clk,reset;
   reg signed [15:0]data_in;
   wire signed [31:0]data_out;


   filter_16tap dut(.clk(clk),.reset(reset),.din(data_in),.dout(data_out));

   //  clock Generation
        initial begin
                         clk=1'b0;
                         reset=1'b1;
                         #7 reset=1'b0;
                         #350 $finish;
```

```
            end

      always #5 clk =~clk;

       initial begin
#2 data_in=16'b0000_0000_0000_0000;//0

#7 data_in=16'b0000_0000_0000_0000;//0
#13 data_in=16'b0000_0000_0000_0000;//0
#7 data_in=16'b00000001_00000000;//1
#13 data_in=16'b00000010_00000000;//2
#7 data_in=16'b00000011_00000000;//3
#13 data_in=16'b00000100_00000000;//4
#7 data_in=16'b00000101_00000000;//5
#13 data_in=16'b00000110_00000000;//6
#7 data_in=16'b00000111_00000000;//7
#13 data_in=16'b11111111_00000000;//-1
#7 data_in=16'b11111110_00000000;//-2
#13 data_in=16'b11111101_00000000;//-3
#7 data_in=16'b11111100_00000000;//-4
#13 data_in=16'b11111011_00000000;//-5
#7 data_in=16'b11111010_00000000;//-6
#13 data_in=16'b11111001_00000000;//-7
#7 data_in=16'b11111000_00000000;//-8
#13 data_in=16'b00000000_00000000;//0
#7 data_in=16'b00000000_00000000;//0

end

always @(posedge clk)
$display($time," data_in= %f , data_out= %f",(data_in*SF1),(data_out*SF2));
```

endmodule

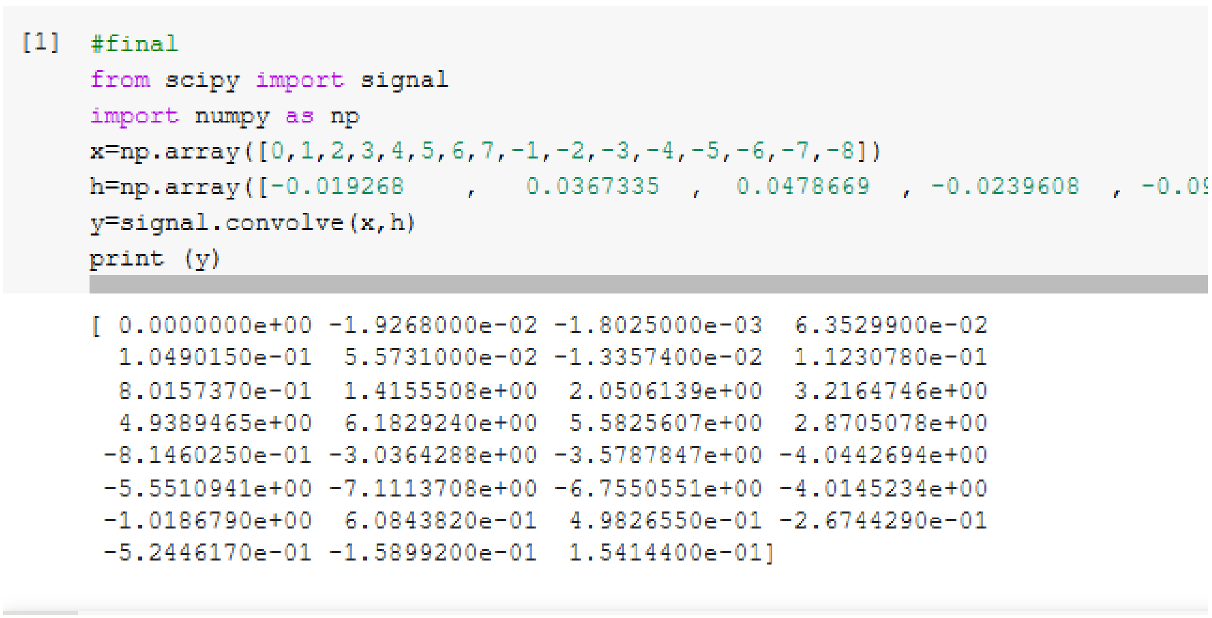
Testing and verification of verilog code:-

The above code will implement the FIR structure and we know that the output of the filter is the convolution on input and filter coefficients. So to cross check we can find the convolution of given input from some other source like using python code and compare it with the output of verilog code.

A python script which used to verify the code is below

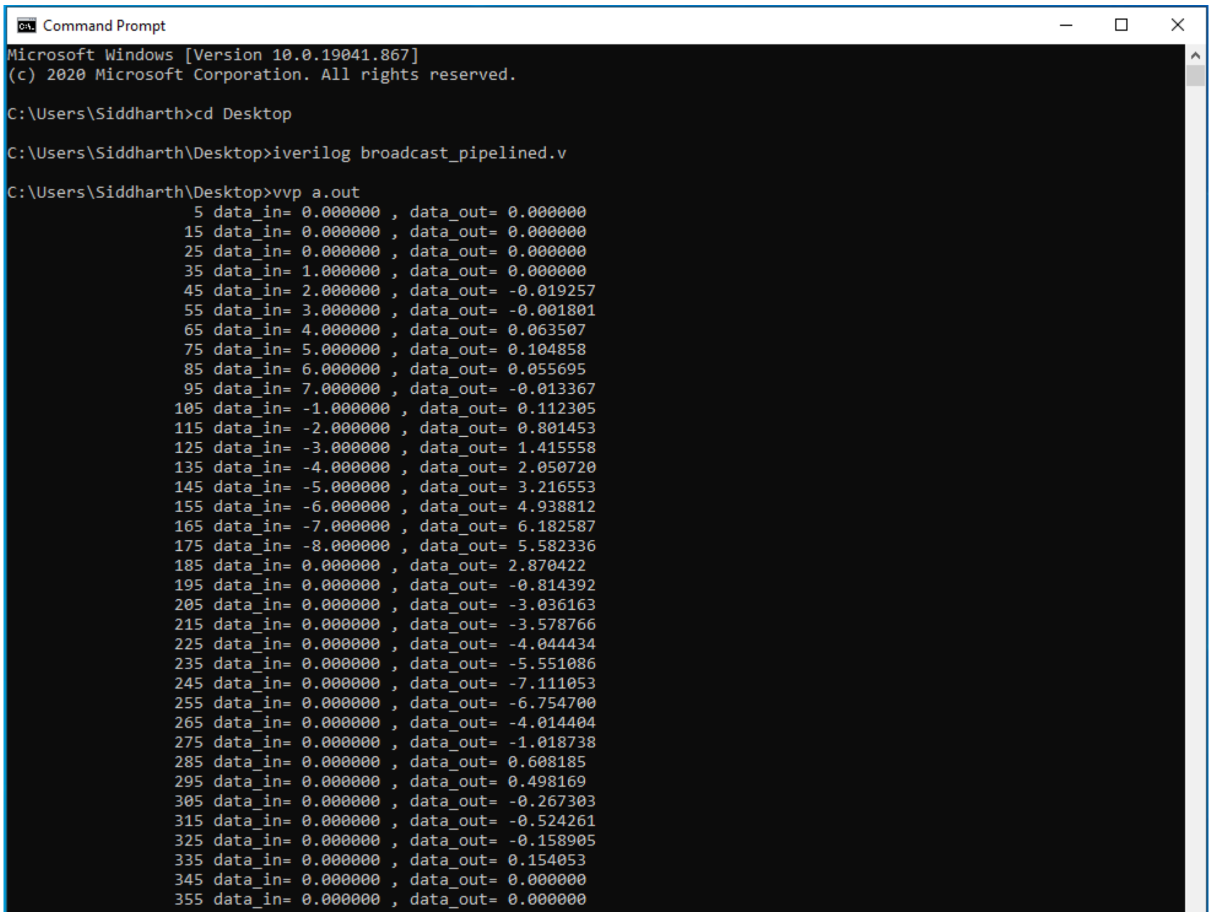
```
#final
from scipy import signal
import numpy as np
x=np.array([0,1,2,3,4,5,6,7,-1,-2,-3,-4,-5,-6,-7,-8])
h=np.array([-0.019268    ,   0.0367335   ,   0.0478669   , -0.0239608   ,
-0.0905421   , -0.0199179   ,   0.1947536   , 0.3901887   ,  0.3901887   ,
0.1947536   ,  -0.0199179   ,  -0.0905421    , -0.0239608    ,  0.0478669
,   0.0367335   ,  -0.019268])
y=signal.convolve(x,h)
print (y)
```

Output of this Py script is below



```
[1]  #final
     from scipy import signal
     import numpy as np
     x=np.array([0,1,2,3,4,5,6,7,-1,-2,-3,-4,-5,-6,-7,-8])
     h=np.array([-0.019268    ,   0.0367335   ,   0.0478669   , -0.0239608   , -0.09
     y=signal.convolve(x,h)
     print (y)

     [ 0.0000000e+00 -1.9268000e-02 -1.8025000e-03  6.3529900e-02
       1.0490150e-01  5.5731000e-02 -1.3357400e-02  1.1230780e-01
       8.0157370e-01  1.4155508e+00  2.0506139e+00  3.2164746e+00
       4.9389465e+00  6.1829240e+00  5.5825607e+00  2.8705078e+00
      -8.1460250e-01 -3.0364288e+00 -3.5787847e+00 -4.0442694e+00
      -5.5510941e+00 -7.1113708e+00 -6.7550551e+00 -4.0145234e+00
      -1.0186790e+00  6.0843820e-01  4.9826550e-01 -2.6744290e-01
      -5.2446170e-01 -1.5899200e-01  1.5414400e-01]
```

The output from verilog code is shown below:

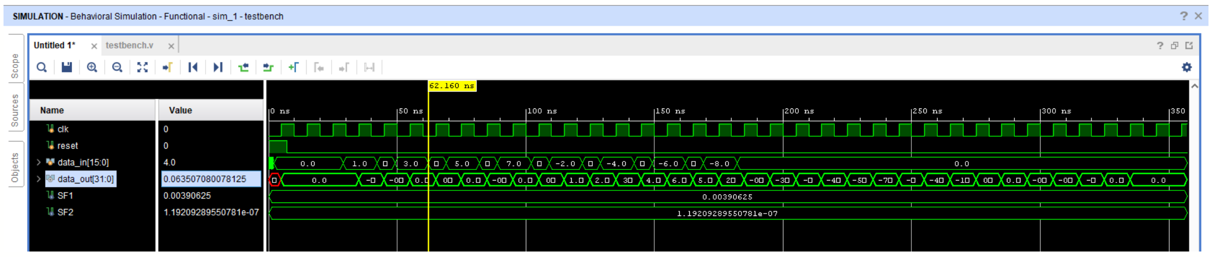


```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Siddharth>cd Desktop

C:\Users\Siddharth\Desktop>iverilog broadcast_pipelined.v

C:\Users\Siddharth\Desktop>vvp a.out
                   5 data_in= 0.000000 , data_out= 0.000000
                  15 data_in= 0.000000 , data_out= 0.000000
                  25 data_in= 0.000000 , data_out= 0.000000
                  35 data_in= 1.000000 , data_out= 0.000000
                  45 data_in= 2.000000 , data_out= -0.019257
                  55 data_in= 3.000000 , data_out= -0.001801
                  65 data_in= 4.000000 , data_out= 0.063507
                  75 data_in= 5.000000 , data_out= 0.104858
                  85 data_in= 6.000000 , data_out= 0.055695
                  95 data_in= 7.000000 , data_out= -0.013367
                 105 data_in= -1.000000 , data_out= 0.112305
                 115 data_in= -2.000000 , data_out= 0.801453
                 125 data_in= -3.000000 , data_out= 1.415558
                 135 data_in= -4.000000 , data_out= 2.050720
                 145 data_in= -5.000000 , data_out= 3.216553
                 155 data_in= -6.000000 , data_out= 4.938812
                 165 data_in= -7.000000 , data_out= 6.182587
                 175 data_in= -8.000000 , data_out= 5.582336
                 185 data_in= 0.000000 , data_out= 2.870422
                 195 data_in= 0.000000 , data_out= -0.814392
                 205 data_in= 0.000000 , data_out= -3.036163
                 215 data_in= 0.000000 , data_out= -3.578766
                 225 data_in= 0.000000 , data_out= -4.044434
                 235 data_in= 0.000000 , data_out= -5.551086
                 245 data_in= 0.000000 , data_out= -7.111053
                 255 data_in= 0.000000 , data_out= -6.754700
                 265 data_in= 0.000000 , data_out= -4.014404
                 275 data_in= 0.000000 , data_out= -1.018738
                 285 data_in= 0.000000 , data_out= 0.608185
                 295 data_in= 0.000000 , data_out= 0.498169
                 305 data_in= 0.000000 , data_out= -0.267303
                 315 data_in= 0.000000 , data_out= -0.524261
                 325 data_in= 0.000000 , data_out= -0.158905
                 335 data_in= 0.000000 , data_out= 0.154053
                 345 data_in= 0.000000 , data_out= 0.000000
                 355 data_in= 0.000000 , data_out= 0.000000
```

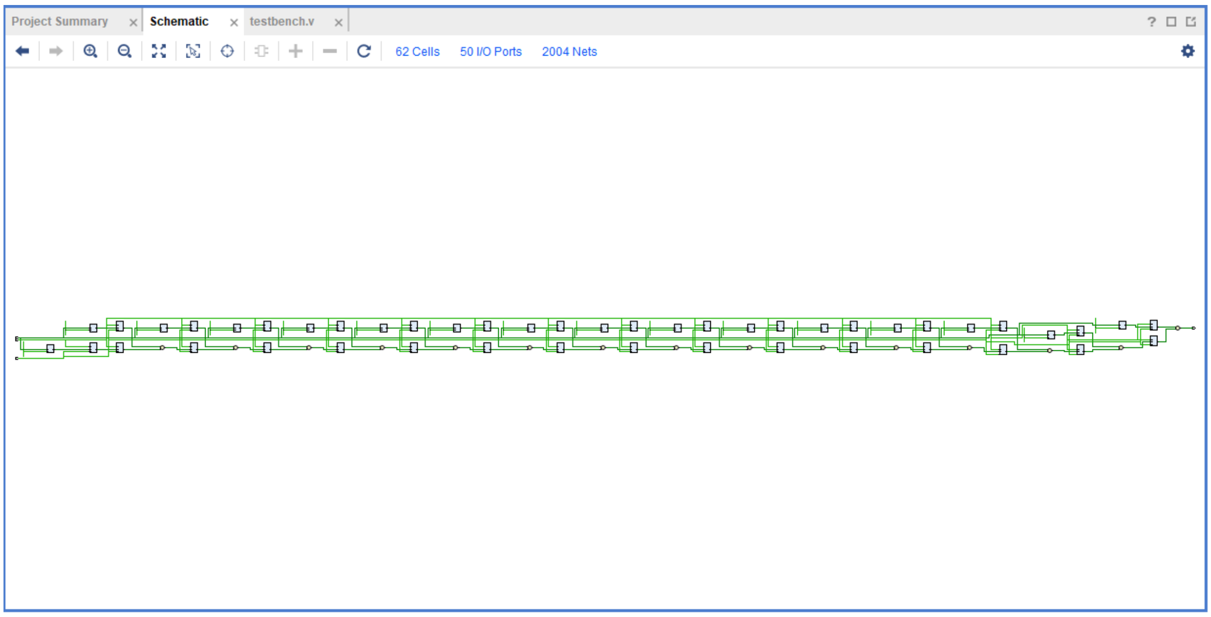
From the comparison of both outputs we can say verilog code is correct and it can be implemented on 7 Series FPGA. We have used 15 bits or fractional parts if we use more bits than precision of the filter can be improved.

Xilinx Vivado Simulation Results:-

Behavioural simulation:-

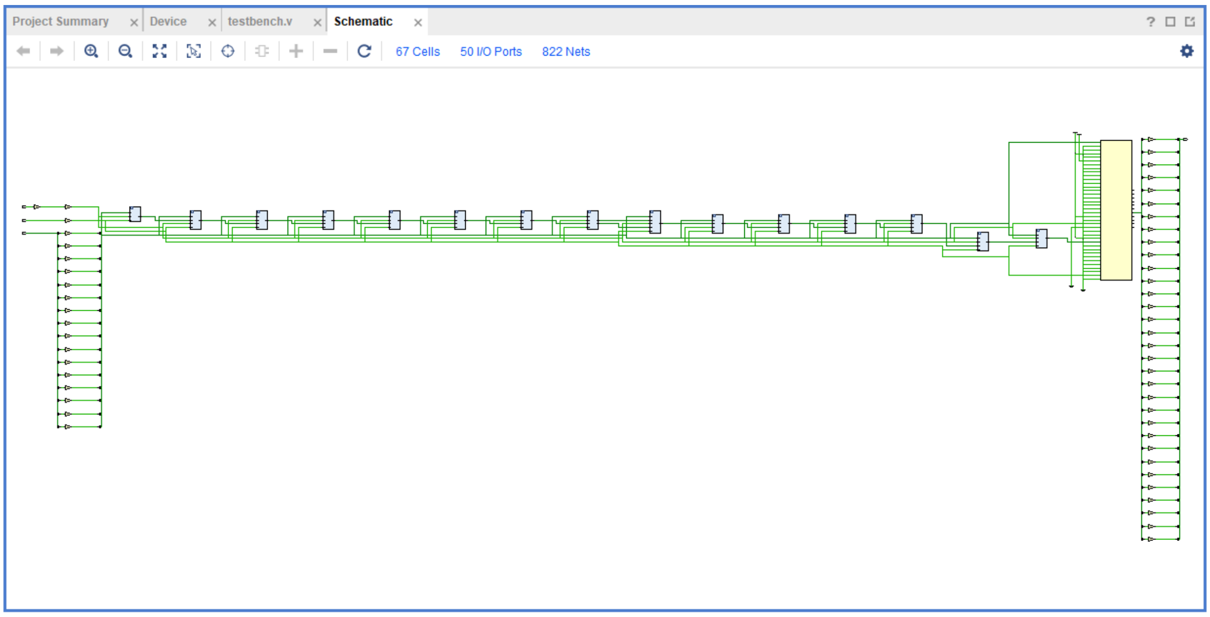


RTL Analysis:-

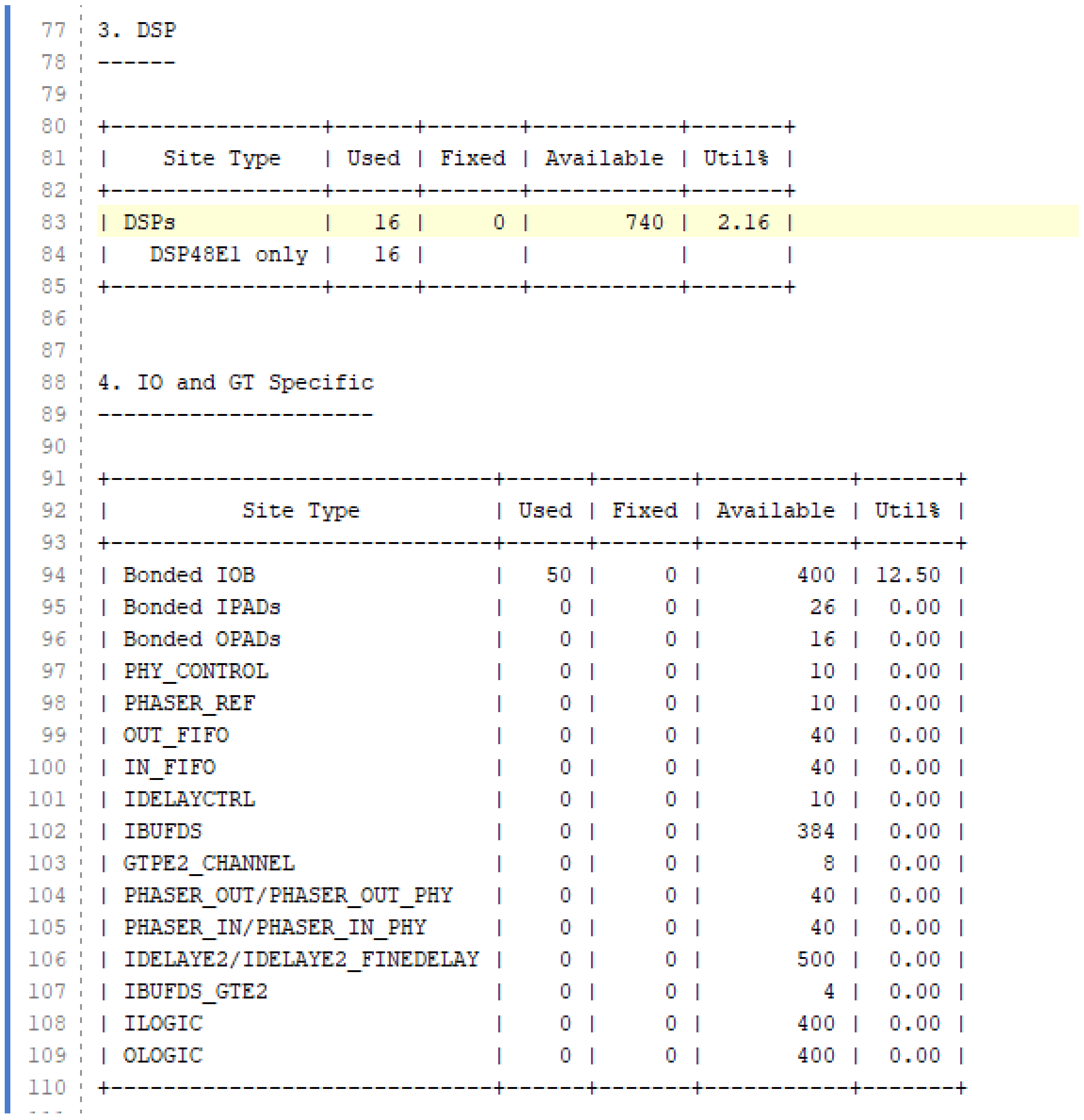


Synthesis : -

Schematic



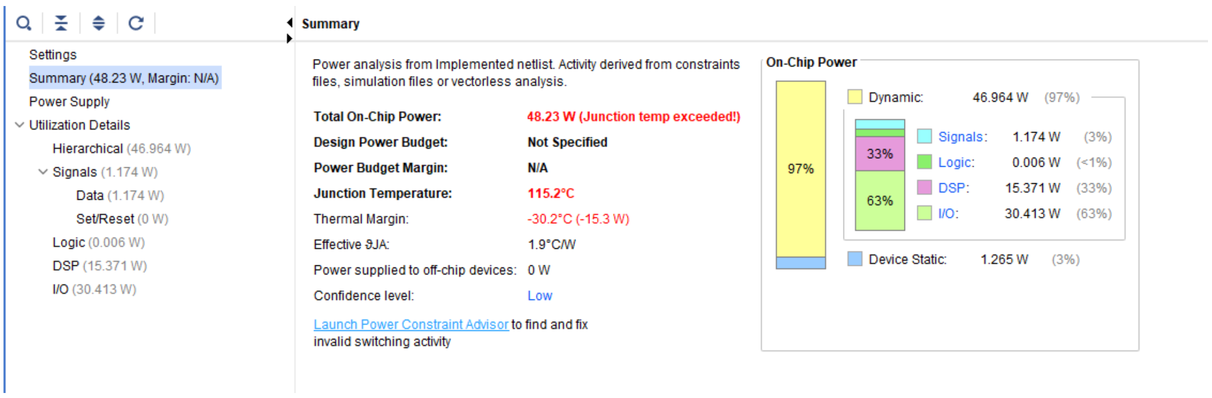
Synthesis Report:-



```
3. DSP
------

+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| DSPs           |   16 |     0 |       740 |  2.16 |
|   DSP48E1 only |   16 |       |           |       |
+----------------+------+-------+-----------+-------+


4. IO and GT Specific
---------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |   50 |     0 |       400 | 12.50 |
| Bonded IPADs                |    0 |     0 |        26 |  0.00 |
| Bonded OPADs                |    0 |     0 |        16 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |        10 |  0.00 |
| PHASER_REF                  |    0 |     0 |        10 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        40 |  0.00 |
| IN_FIFO                     |    0 |     0 |        40 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |        10 |  0.00 |
| IBUFDS                      |    0 |     0 |       384 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |         8 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        40 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        40 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       500 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |         4 |  0.00 |
| ILOGIC                      |    0 |     0 |       400 |  0.00 |
| OLOGIC                      |    0 |     0 |       400 |  0.00 |
+-----------------------------+------+-------+-----------+-------+
```

Implementation:-

Power Report and Timing Report



Summary

- Settings
- Summary (48.23 W, Margin: N/A)
- Power Supply
- Utilization Details
  - Hierarchical (46.964 W)
  - Signals (1.174 W)
    - Data (1.174 W)
    - Set/Reset (0 W)
  - Logic (0.006 W)
  - DSP (15.371 W)
  - I/O (30.413 W)

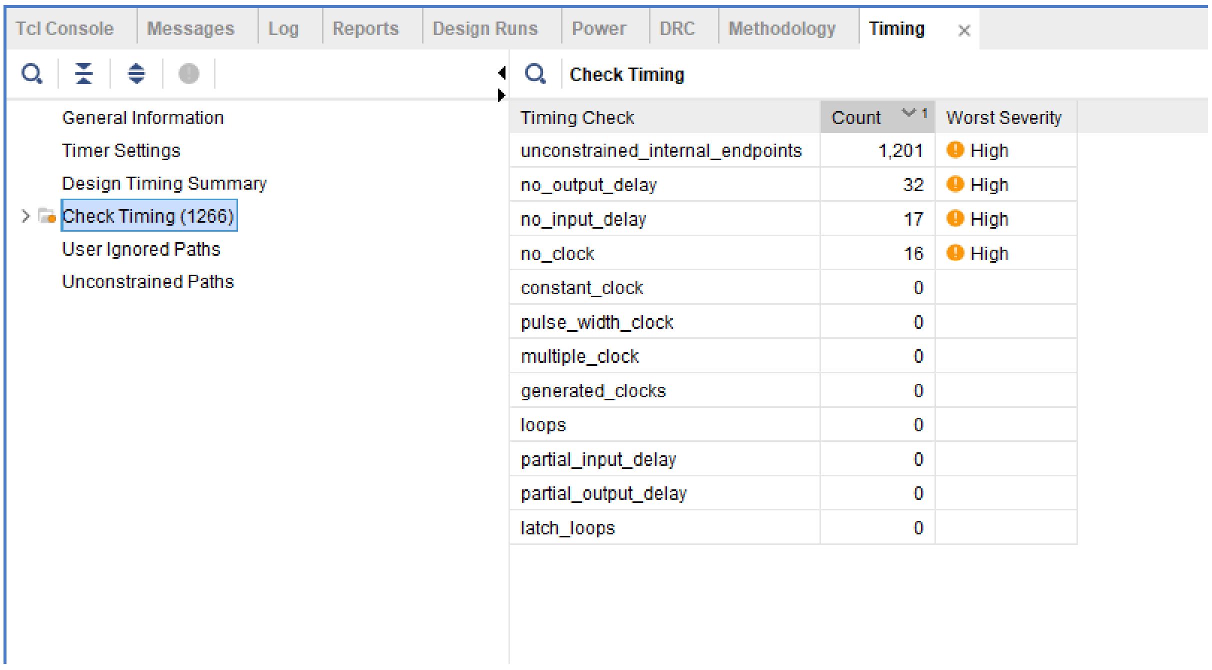
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**48.23 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**115.2°C**
Thermal Margin:	-30.2°C (-15.3 W)
Effective ϑJA:	1.9°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	46.964 W	(97%)	97%
Signals:	1.174 W	(3%)	
Logic:	0.006 W	(<1%)	
DSP:	15.371 W	(33%)	33%
I/O:	30.413 W	(63%)	63%
Device Static:	1.265 W	(3%)	



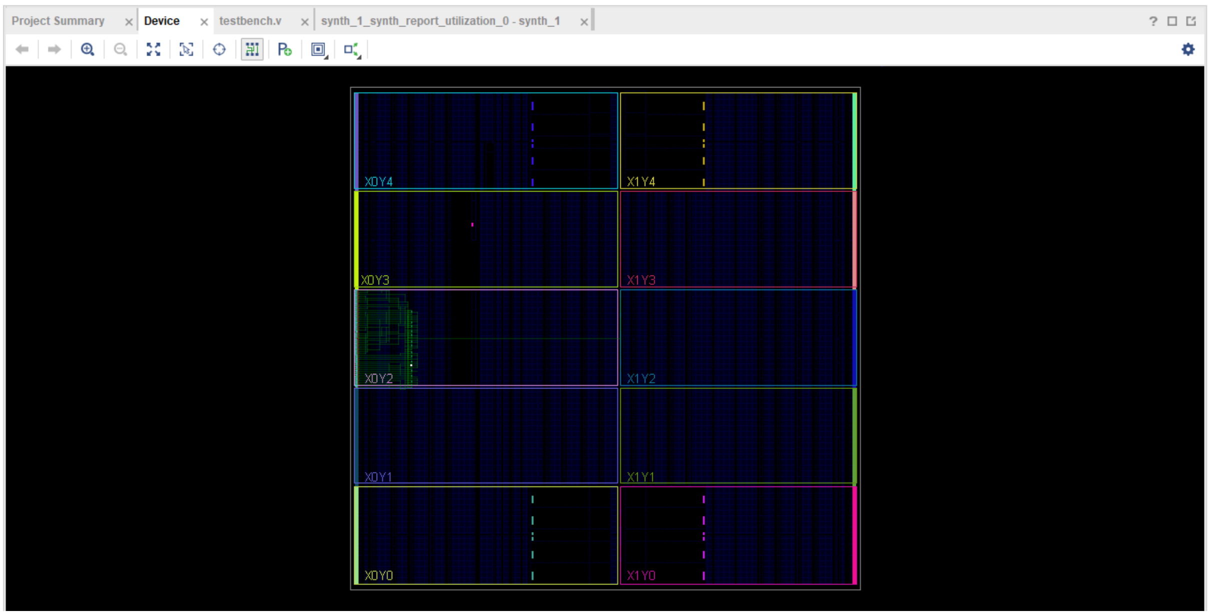
Tcl Console | Messages | Log | Reports | Design Runs | Power | DRC | Methodology | Timing

- General Information
- Timer Settings
- Design Timing Summary
- Check Timing (1266)
- User Ignored Paths
- Unconstrained Paths

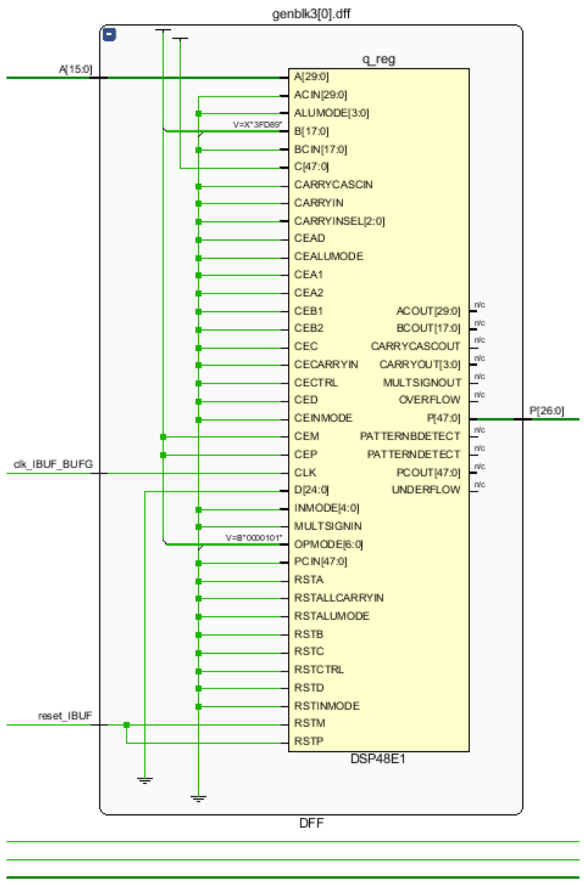
Check Timing

Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	1,201	High
no_output_delay	32	High
no_input_delay	17	High
no_clock	16	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

Device after Implementation



The above Implemented device is using DSPE Slices to implement the 16 Tap Filter.One DSP48E1 Slice is shown below,



Conclusion:-

In conclusion A FIR Filter can be implemented on a 7 Series FPGA. Because of fixed word length in digital filter coefficients have to be rounded off which creates deviation in the performance of the FIR filter implemented on Xilinx Artix-7 FPGA Evaluation Kit. Precision can be increased by using large no. of bits to represent the fractional part of the filter coefficients.

Submitted by:-

Sidharth Yadav

VLSI System Design