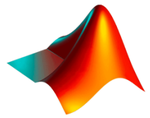
Generic Custom Transposed FIR Filter Avalon IP integrated with NIOS II Processor

By: Youssef Gamal Eldein

Design Flow

1. **FIR Filter Using Matlab** 
2. **FIR Filter RTL Desgin** 

3. **Simulation for the FIR** 
4. **Avalon Wrapper Interface** 

5. **SoC Hardware** 
6. **Soc Software (C code)** 
7. **Simulating the SoC** 
8. **Implementing on FPGA** 

What is a Filter

a filter is by an incredibly broad definition any medium through which a signal passes can be regarded as a filter however we do not usually think about something as a filter unless it can modify the signal in some way

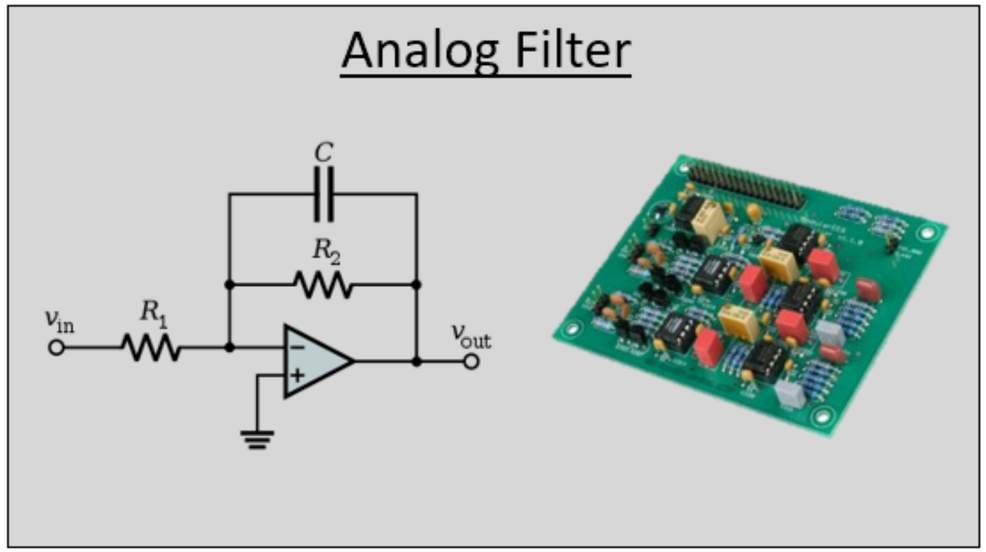
Filters are used in many disciplines. For example, image processing makes heavy use of 2D filters, where the input and output are images. You might use a filter every morning to make your coffee, which filters out solids from liquid. In DSP,

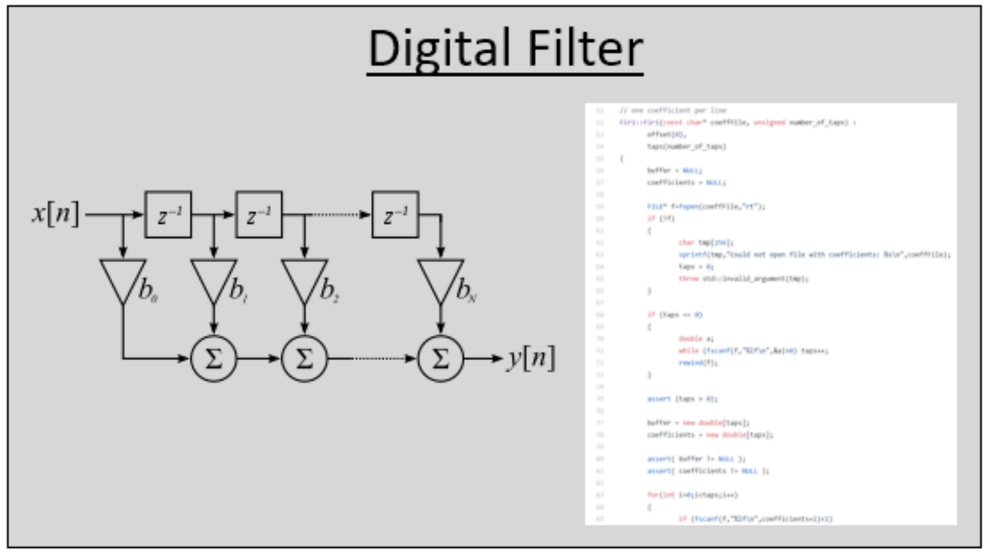
filters are primarily used for:

- **Separation of signals that have been combined (e.g., extracting the signal you want)**
- **Removal of excess noise after receiving a signal**
- **Restoration of signals that have been distorted in some way (e.g., an audio equalizer is a filter)**

What is a Filter

In electronics there are 2 main types of Filters: Analog Filters and Digital Filters



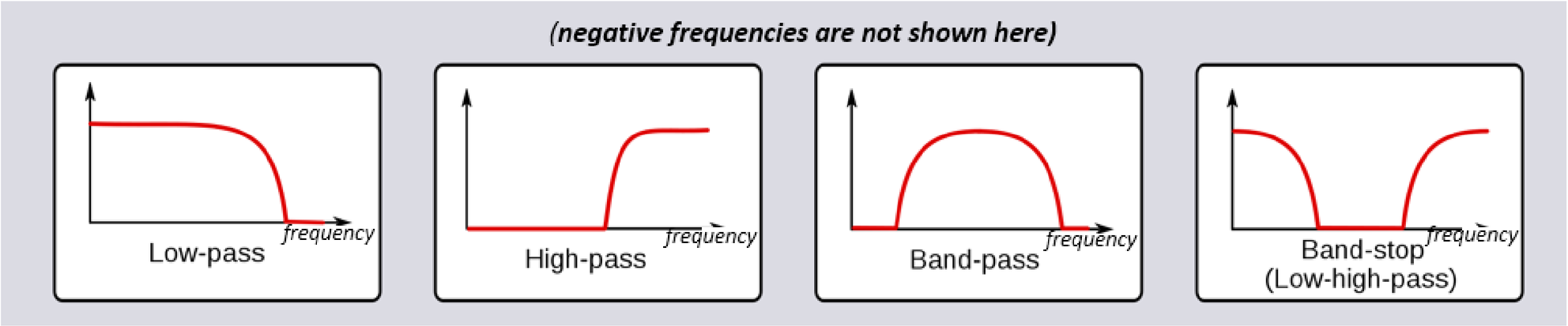


Why use Digital Filters

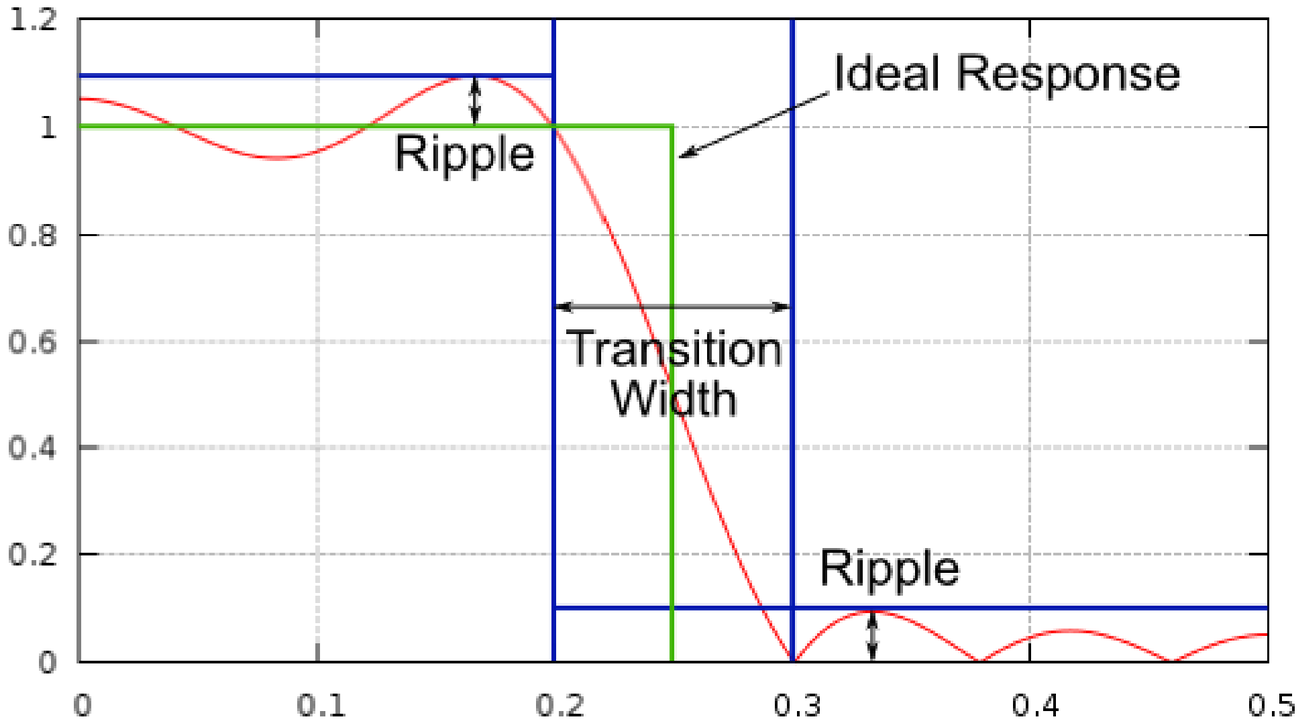
DIGITAL VERSUS ANALOG FILTERING

DIGITAL FILTERS	ANALOG FILTERS
High Accuracy	Less Accuracy - Component Tolerances
Linear Phase (FIR Filters)	Non-Linear Phase
No Drift Due to Component Variations	Drift Due to Component Variations
Flexible, Adaptive Filtering Possible	Adaptive Filters Difficult
Easy to Simulate and Design	Difficult to Simulate and Design
Computation Must be Completed in Sampling Period - Limits Real Time Operation	Analog Filters Required at High Frequencies and for Anti-Aliasing Filters
Requires High Performance ADC, DAC & DSP	No ADC, DAC, or DSP Required

Types of Filters



Filter Characteristics



Each filter permits certain frequencies to remain from a signal while blocking other frequencies. The range of frequencies a filter lets through is known as the “passband”, and “stopband” refers to what is blocked. In the case of the low-pass filter, it passes low frequencies and stops high frequencies, so 0 Hz will always be in the passband. For a high-pass and band-pass filter, 0 Hz will always be in the stopband.

Do not confuse these filtering types with filter algorithmic implementation (e.g., IIR vs FIR).

‘’Transition width’’, also measured in Hz, instructs the filter how quickly it has to go between the passband and stopband since an instant transition is impossible.

Filter Characteristics

Here is an example of a set of filter taps or Coefficients, which define one filter

```
h =  [ 9.92977939e-04  1.08410297e-03  8.51595307e-04  1.64604862e-04
 -1.01714338e-03 -2.46268845e-03 -3.58236429e-03 -3.55412543e-03
 -1.68583512e-03  2.10562324e-03  6.93100252e-03  1.09302641e-02
  1.17766532e-02  7.60955496e-03 -1.90555639e-03 -1.48306750e-02
 -2.69313236e-02 -3.25659606e-02 -2.63400086e-02 -5.04184562e-03
  3.08099470e-02  7.64264738e-02  1.23536693e-01  1.62377258e-01
  1.84320776e-01  1.84320776e-01  1.62377258e-01  1.23536693e-01
  7.64264738e-02  3.08099470e-02 -5.04184562e-03 -2.63400086e-02
 -3.25659606e-02 -2.69313236e-02 -1.48306750e-02 -1.90555639e-03
  7.60955496e-03  1.17766532e-02  1.09302641e-02  6.93100252e-03
  2.10562324e-03 -1.68583512e-03 -3.55412543e-03 -3.58236429e-03
 -2.46268845e-03 -1.01714338e-03  1.64604862e-04  8.51595307e-04
  1.08410297e-03  9.92977939e-04]
```

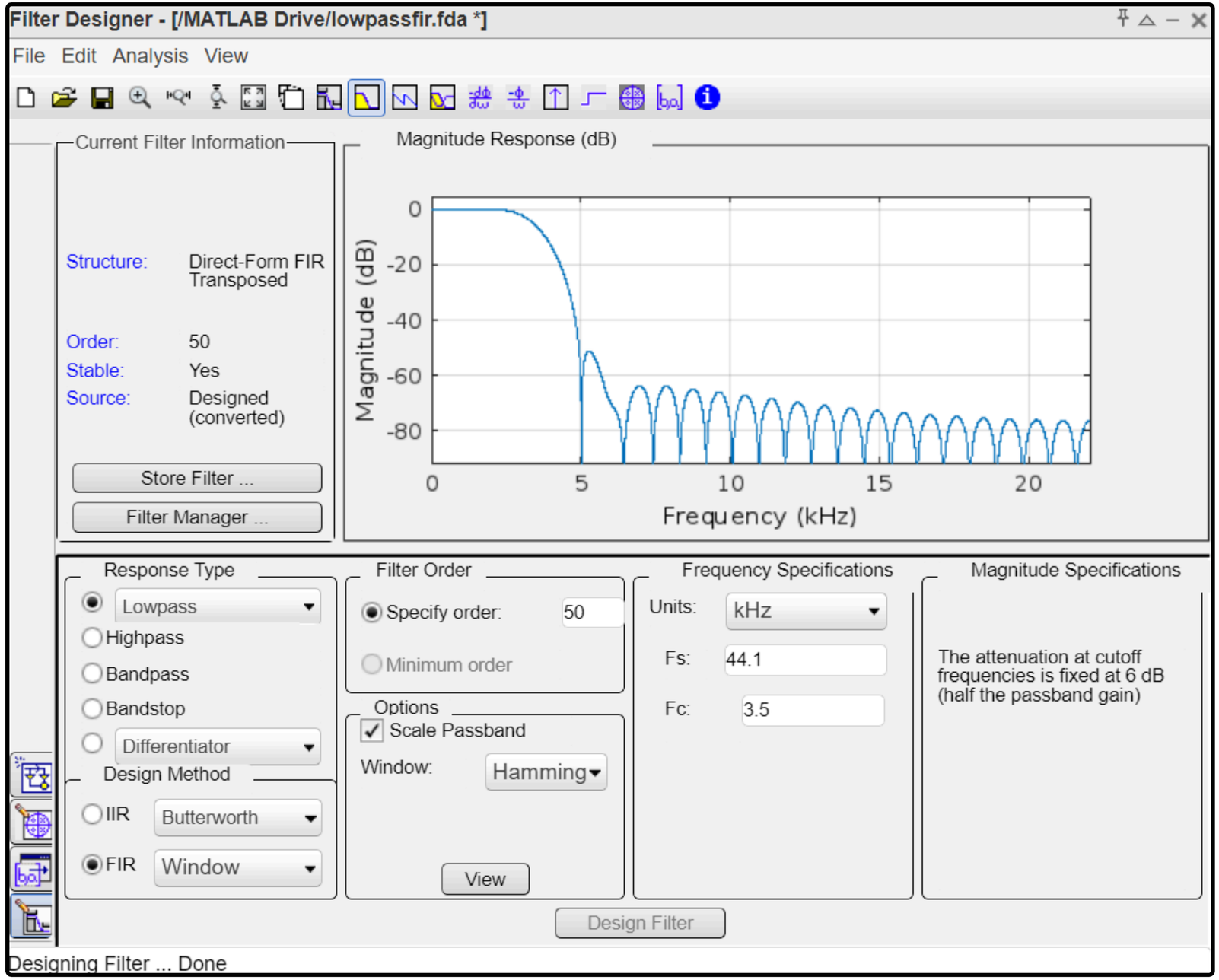
We get these cofficients by using MATLAB Filter Designer

For most filters we will see (known as FIR, or Finite Impulse Response, type filters), we can represent the filter itself with a single array of floats. For filters symmetrical in the frequency domain, these floats will be real (versus complex), and there tends to be an odd number of them. We call this array of floats “filter taps” or Coefficients. We often use ‘h’ as the symbol for filter taps.

Filter Design Usign Matlab

as a start we will design a 50th Order Low pass FIR filter with hamming window

Cutoff Freq. = 3.5KHz
Sampling Freq. = 44.1 KHz
Structure: Transposed
(we will get to the difference btw trasposed and direct form later)



we generate the Coefficients for the filter “File” >> “Export” >> under “Export to” Click Coefficents ASCII >> then Choose Decimal

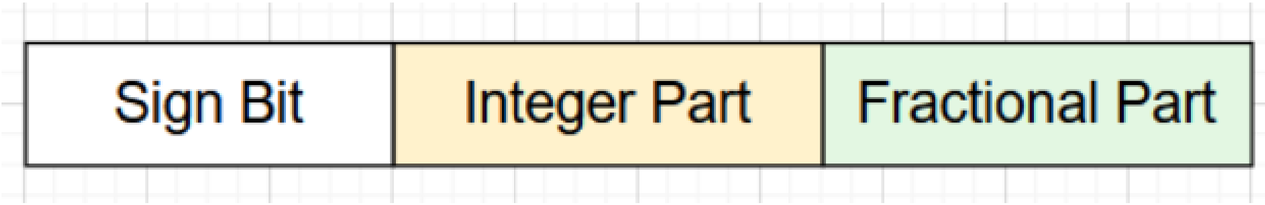
/MATLAB Drive/lowpassfir.fcf

```
% Generated by MATLAB(R) 24.1 and Signal Processing Toolbox 24.1.
% Generated on: 02-May-2024 19:23:31

% Coefficient Format: Decimal

% Discrete-Time FIR Filter (real)
% -------------------------------
% Filter Structure  : Direct-Form FIR Transposed
% Filter Length     : 51
% Stable            : Yes
% Linear Phase      : Yes (Type 1)


Numerator:
-0.000101679756821436263563261659381709023
-0.000626407761087045106794535254124411949
-0.001166205293878934227957966385247345897
-0.001628539759381721812106502689232456 76
-0.001801676391558073827073305217538745637
-0.001396570137053725329384645803543207876
-0.000171358130166342068805354625915526867
 0.001898387965257118019726800817181810999
 0.004473800071620057106080992070928914472
 0.006810811909799314826929883537331988919
```

Small Note: Number of Coefficients or TAPS is higher than the filter order by 1 so 50th order filter will have 51 cofficients and we will see this in the FIR Archeticture

We now want to represent the Floating point Coefficients into binary numbers

we will use Fixed Point Representation and since some of the coeffs are -ve and some are +ve we will use the Signed Version
let the TAP size be 16 bits
we will reserve the MSB for the sign (1 bit)
since all coeffs are less than 1 we dont need any bits for the integer part (0 bit)
and the rest of the bits for the fractional part (15 bits)



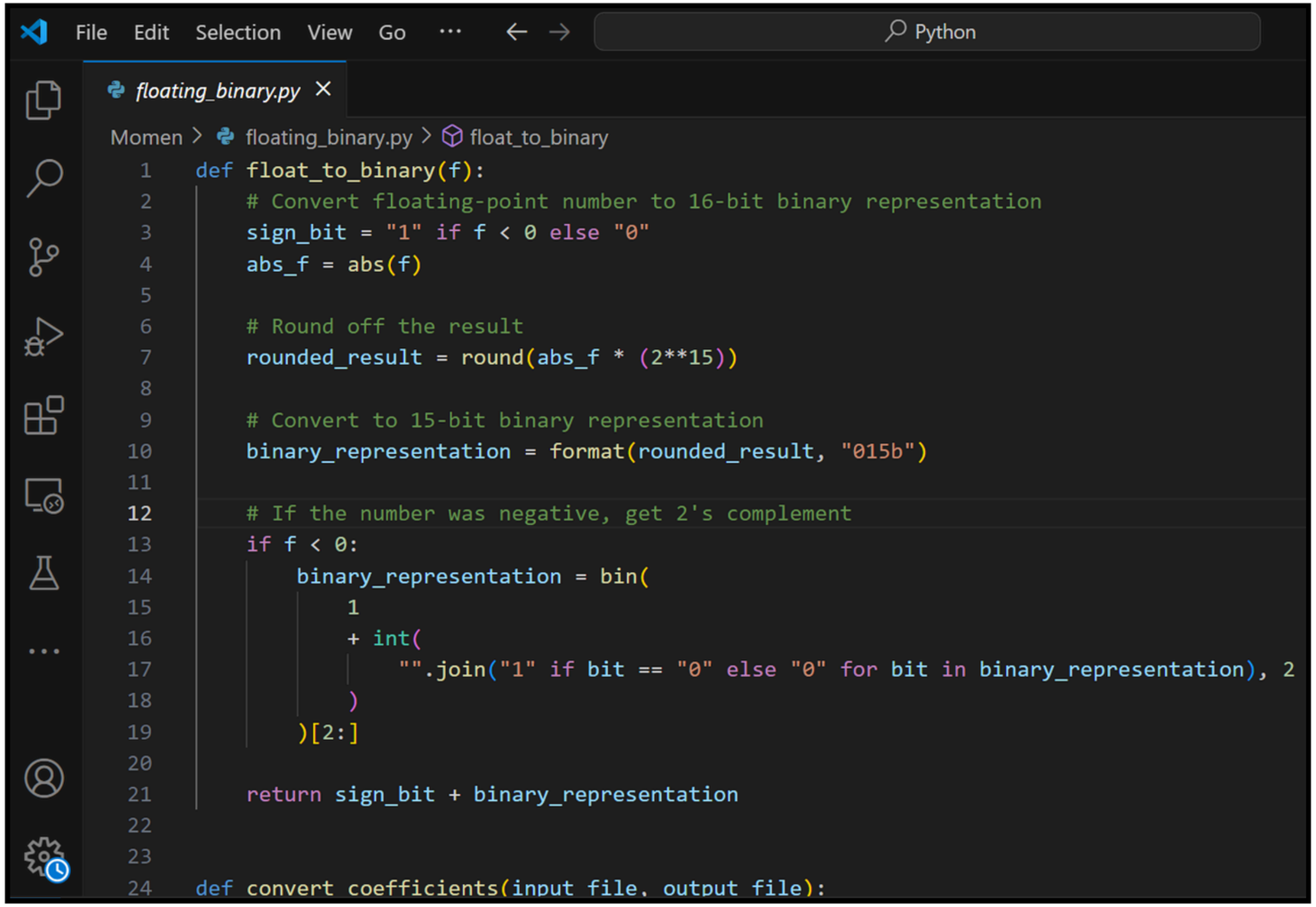
to represent n-bit fractional number, multiply fractional part by 2^n
if +ve >> round the result off
if -ve >> round the result off >> get 2's complement

EX1: -0.000101679 * 2^15 = round off(-3.3318) = -3
3 in binary >> 0000_0000_0000_011
2's complement >> 1111_1111_1111_101

EX2: 0.00189838 * 2^15 = round off(62.206) = 62
62 in binary >> 0000_0000_0111_110

Automating TAPS Conversion

as you saw converting each coefficient from floating to binary takes time so we will make a Python script which take the coeffs file as input and give you the binary equivalent of each coeffs so we can use it in our RTL design



To Check the rest of the code go to GitHub

Momen > lowpassfir_coeffs_float.txt

```
-0.000101679756821436263563261659381709023
-0.000626407761087045106794535254124411949
-0.001166205293878934227957966385247345897
-0.001628539759381721812106502689232456 76
-0.001801676391558073827073305217538745637
-0.001396570137053725329384645803543207876
-0.000171358130166342068805354625915526867
 0.001898387965257118019726800817181810999
 0.004473800071620057106080992070928914472
 0.006810811909799314826929883537331988919
 0.007879282557866029018667219929739076179
 0.006649266756610462680843198768343427218
 0.002486150757723167535606645373036371893
-0.004459324139810905661407414157793027698
-0.013005134512620164857765736599048977951
-0.020955628784920205670561088595604815055
-0.025438394410897253311576449164022051264
-0.023523277420649316149514618246030295268
-0.012992004096052613484890692063800088363
 0.006938941331778051765932779915146966232
 0.035152555766104480461375914046584512107
 0.068584048048719592483024598550400696695
 0.102652032294073464191441757975553628057
 0.132100965524440783216419958989717997611
```

1st Example

2nd Example

Python Script

Momen > lowpassfir_coeffs_binary.txt

```
1111111111111101
1111111111101011
1111111111011010
1111111111001011
1111111111000101
1111111111010010
1111111111111010
0000000000111110
0000000010010011
0000000011011111
0000000100000010
0000000011011010
0000000001010001
1111111101101110
1111111001010110
1111110101010001
1111110010111110
1111110011111101
1111111001010110
0000000011100011
0000010010000000
0000100011000111
0000110100100100
0001000011101001
```

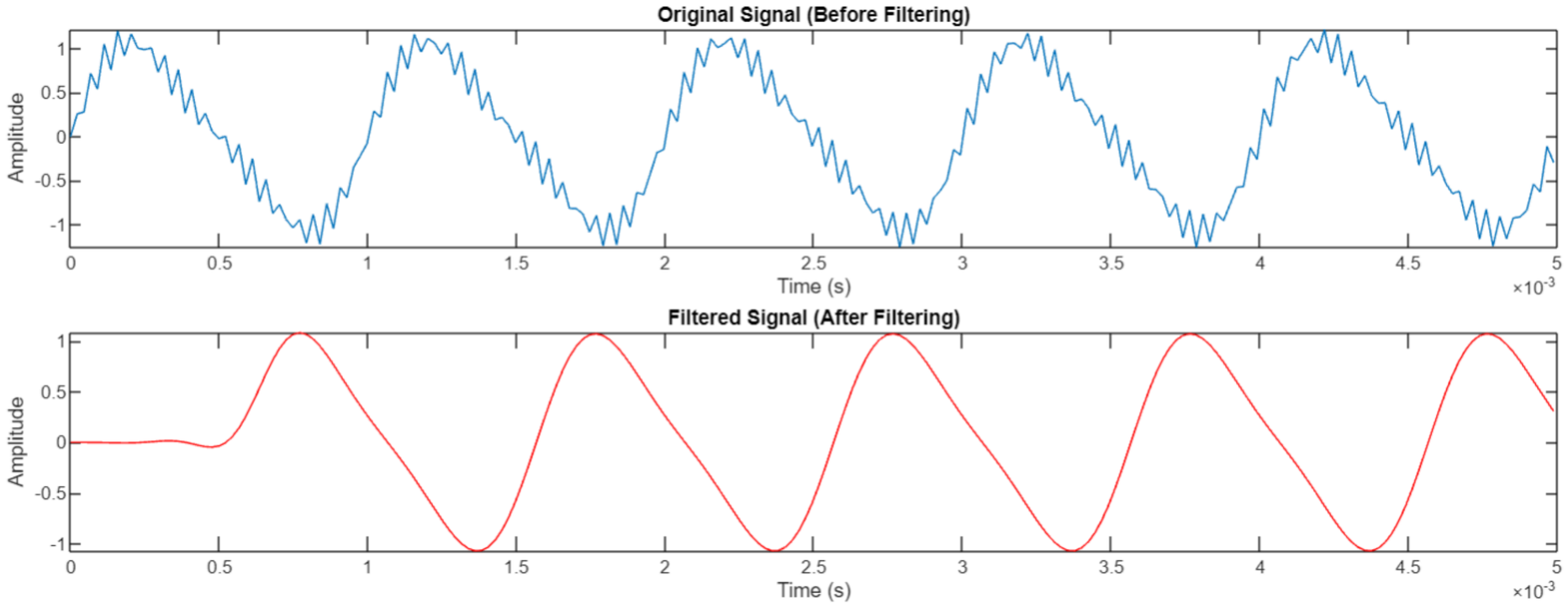
Matlab Script for the fir Filter

After we got the coefficients from matlab we need to write a matlab code to model our FIR fitler to use as our golden model we will compare the output of the RTL to this model later in Verification Stage

```
% FIR Low-pass Filter Design using Hamming Window

% Specifications
order = 50;                         % Filter order
cutoff_frequency = 3.5e3;           % Cutoff frequency in Hz
sampling_rate = 44.1e3;             % Sampling rate in Hz

% Design the FIR filter using Hamming window
fir_coefficients = fir1(order, cutoff_frequency/(sampling_rate/2),
                        'low', hamming(order + 1));

% Display the coefficients
%disp('FIR Coefficients:');
%disp(fir_coefficients);

% Time vector
t = 0:1/sampling_rate:0.005;
```

Testing the Matlab Script

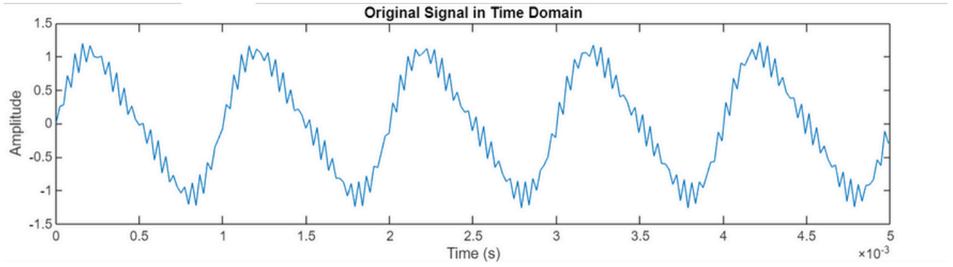
```
% Generate signals
signal = (sin(2*pi*1000*t));
noise = (0.5*sin(2*pi*11000*t)).*(0.8*cos(2*pi*9000*t));
noisy_signal = noise + signal;
```



Matlab Script for Signal Generation

we use the same signal we applied on the filter designed in matlab before

```
% Scale the signal to fit within the range of a 16-bit integer
scaled_signal = int16(choice * (2^15 - 1));

% Convert the scaled signal to binary representation
binary_signal = dec2bin(typecast(scaled_signal, 'uint16'), 16);

%Save each 16-bit binary value on a separate line in the text file
fileID = fopen('1.txt', 'w');
for i = 1:size(binary_signal, 1)
    fprintf(fileID, '%s\n', binary_signal(i, :));
 end
fclose(fileID);

% Plot the original signal in time domain
figure('Position', [100, 200, 2500, 600]);
plot(t, choice);
title('Original Signal in Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
```



/MATLAB Drive/Input_signal.txt

```
1  0000000000000000
2  0010000010111110
3  0010001110101101
4  0101110000001010
5  0100010011000100
6  0111111111111111
7  0110000010111111
8  0111111111111111
9  0111010101000110
10 0111111111111111
11 0111111111111111
12 0111111001011011
13 0111111111111111
14 0101110110110100
15 0111011001000011
16 0011110010011111
17 0110000101110010
18 0010001000110000
19 0100010001001111
20 0001000100110001
21 0010000110101110
```

FIR RTL Design

this is the archeticture of 4th order Direct Form FIR lets break each block quickly

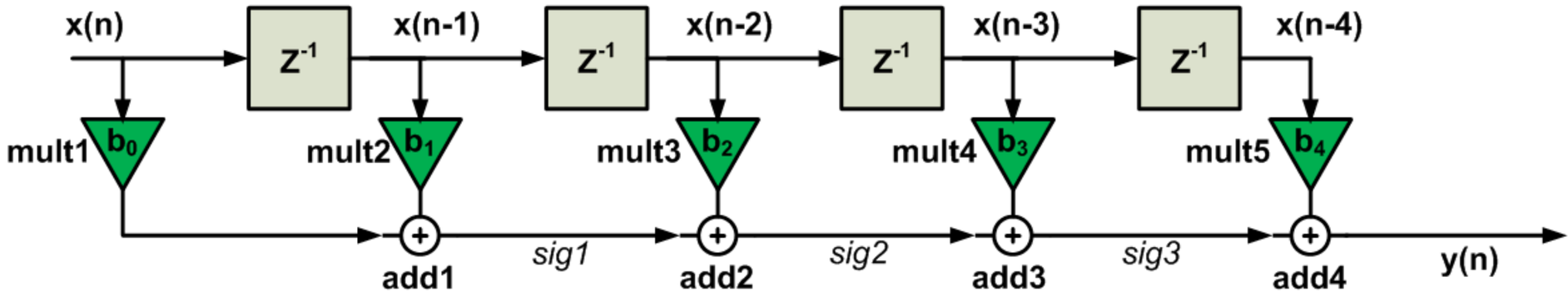
x(n): is just your input signal at the present time

Z^-1: delay block by one cycle you can think of it as a register which delay the input till the next cycle

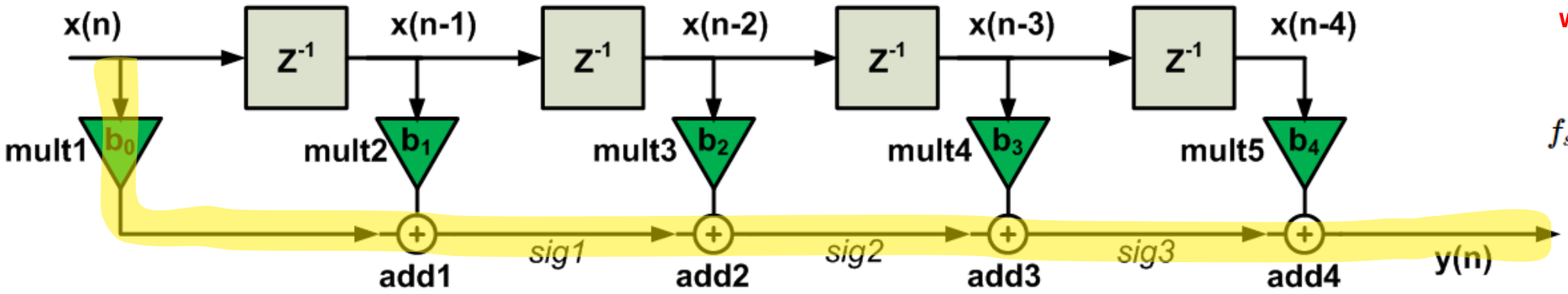
x(n-2): the input signal delayed by 2 cycles

the green triangle is our TAP or a multiplier which multiply your input signal with the coefficient and at the end you have an adder which sums all the result of the multipliers

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

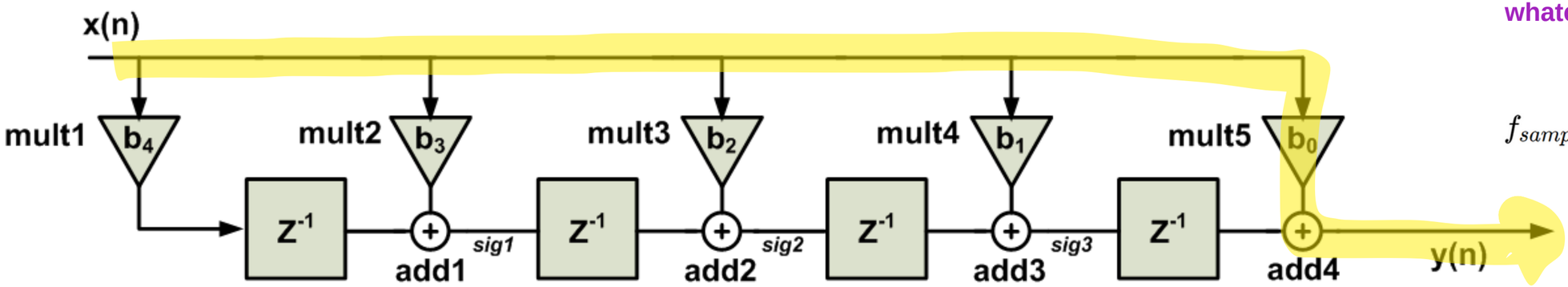


Transposed VS Direct Form



Critical Path = one mult + N*adders
where N equals FIR order

$$f_{sampling} = \frac{1}{T_{mult} + 4T_{add}}$$

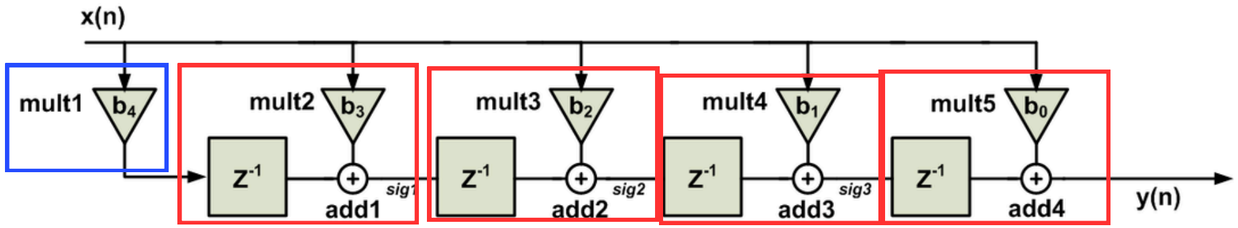


Critical Path = one mult + one adder
whatever the order of the FIR

$$f_{sampling} = \frac{1}{T_{mult} + T_{add}}$$

now you know why we choosed transposed form in matlab filter designer and this is the fir arch that we will write RTL for

Finding a pattern in the FIR Archeitecture



Whatever the order of the filter is it will consist of the same thing

Transposed FIR = one mult(in blue box) + N(Trasnposed Block)

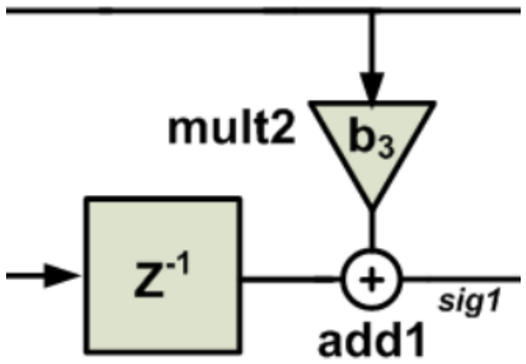
where N is filter order and Trasnposed block is the Block inside the red Box

Now lets Write the RTL for a single Transposed block

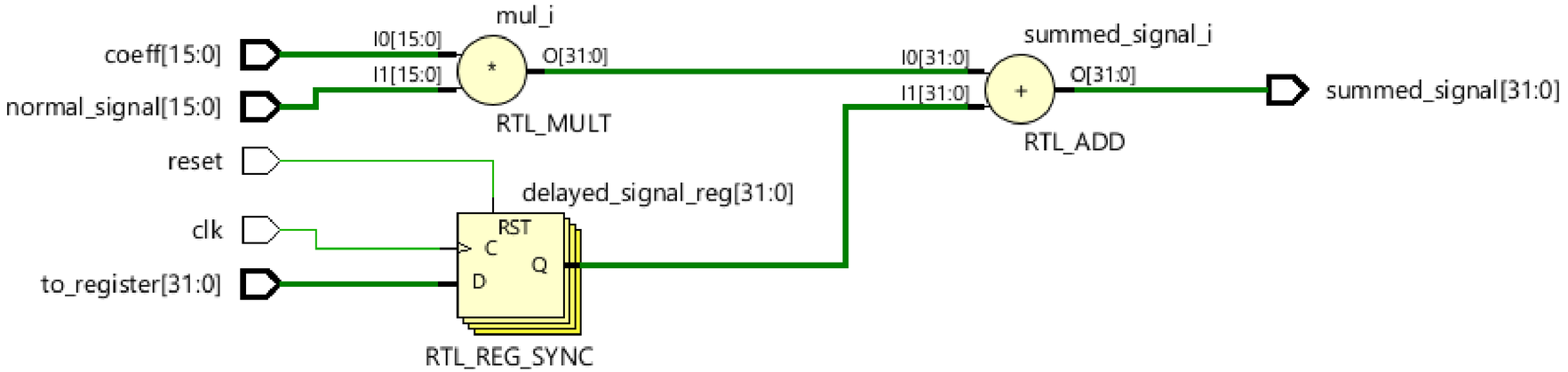
Making a Genric FIR

We can now just write the rtl for the fir we want and accoriding to the fir filter order the amount of code will be determined but since we want to design a genric filter we will make it with another way. we will design fir with order 1 in a seperate module and then in another module replicate it as many times as you want using For Generate Block according to a parameter

RTL Viewer



Transposed_block



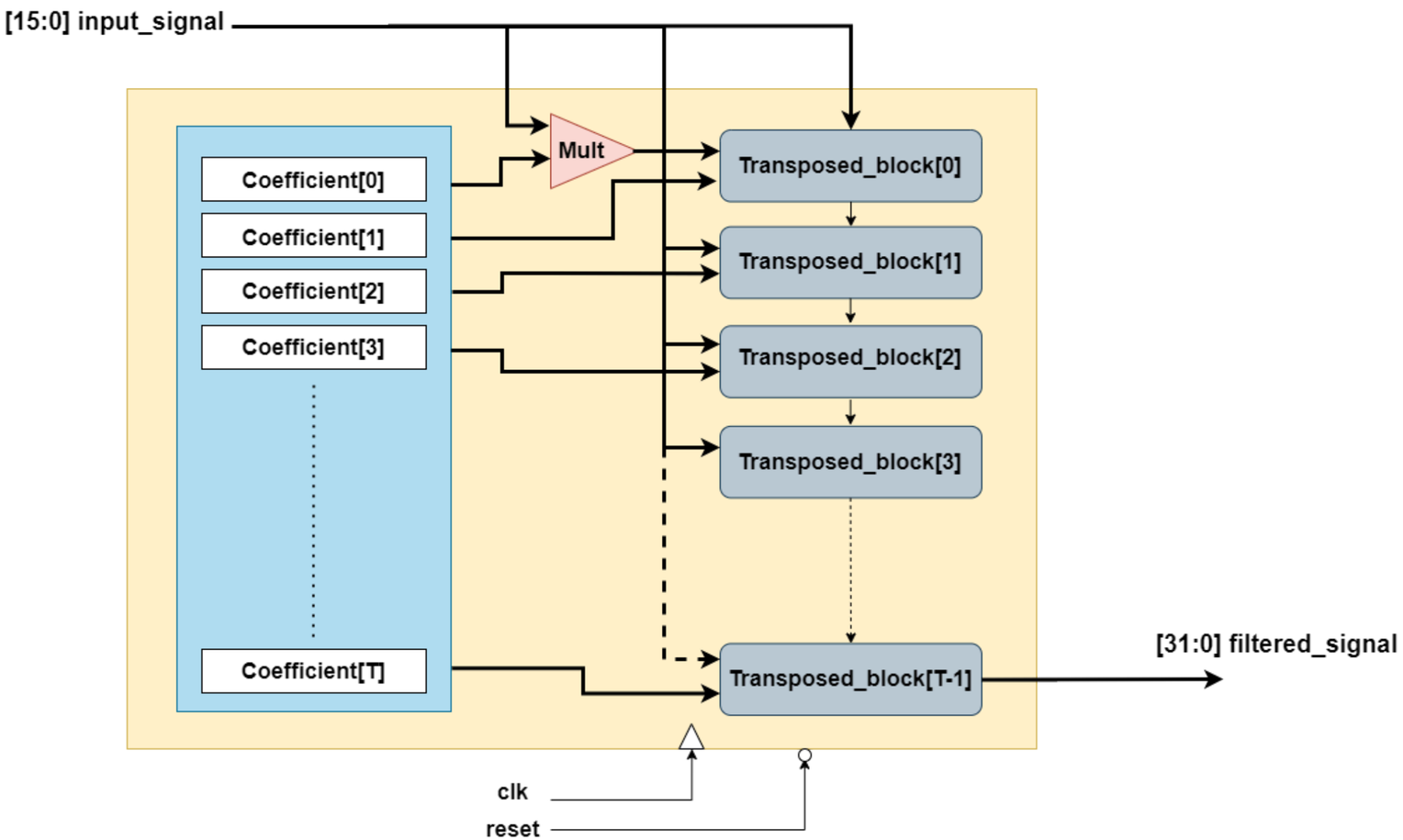
Transposed Block RTL

```
module transposed_block(clk , reset , normal_signal , coeff,to_register, summed_signal);

    //parameters
    parameter                           N = 16;                         // bit resolution

    //input ports
    input                               clk;                            //FPGA clk 50MHz
    input                               reset;                          //active low reset
    input    signed  [N-1:0]            normal_signal;                  //input signal x(n)
    input    signed  [N-1:0]            coeff;                          //coefficient value
    input    signed  [2*N-1:0]          to_register;                    //input signal to Z^-1

    //output ports
    output   signed  [2*N-1:0]          summed_signal;                  //adder output

    //internal signals
    wire     signed  [2*N-1:0]          mul;                            //our TAP (triangle)
    reg      signed  [2*N-1:0]          delayed_signal;                 //output from Z^-1 block


    always@(posedge clk) begin
        if(reset)
            delayed_signal <= 0;

        else
            delayed_signal <= to_register ;
    end

    assign mul = coeff * normal_signal ;
    assign summed_signal = mul + delayed_signal ;
endmodule
```

To Check the rest of the code go to GitHub

Generic FIR Archeitecture

the blue block is a ROM for the coefficients and then the transposed block are replicated using for generate according to a parameter caleed ‘’T’’ which is equivalent to number of TAPS
the TAPS = Filter Order + 1



Generic FIR RTL

Pros:

just by changing the initialization file of the input rom and coeffs rom you can have any filter and any signal without needing to write any extra code

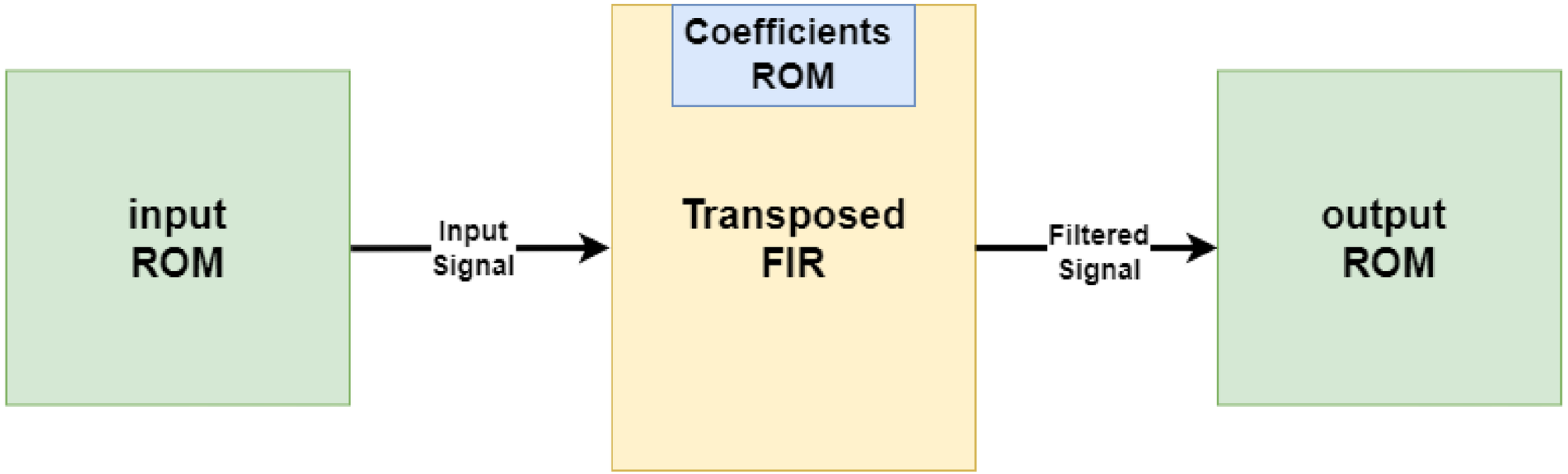
Cons:

You can’t change filter Specs when hardware is programmed

```
genvar i;
generate
  for (i = 0; i < T-1; i = i + 1) begin : gen_block
   transposed_block O_FIR (
      .clk(clk),
      .reset(reset),
      .normal_signal(noisy_signal),
      .coeff(coeff[T-2-i]),
      .to_register(summed_signal[i]),
      .summed_signal(summed_signal[i+1])
    );
  end
endgenerate


assign filtered_signal = summed_signal[T-1];

initial begin
  $readmemb("binary_coefficients2.txt" , coeff);
  end

endmodule
```

Full FIR Archeitecture

Adding 2 memories one for input signal to be stored there and one for output to store the filtered samples
adding those 2 ROMs will help us when we get to SOC development



Simulation of FIR

to simulate the fir now we have to generate a signal to filter it. as I worked through this project I saw many ways to generate the signal I will mention them and we will use the easieset one among them

1) Using Matlab to generate you a signal in form of sampled amplitudes and then store them into the input memory

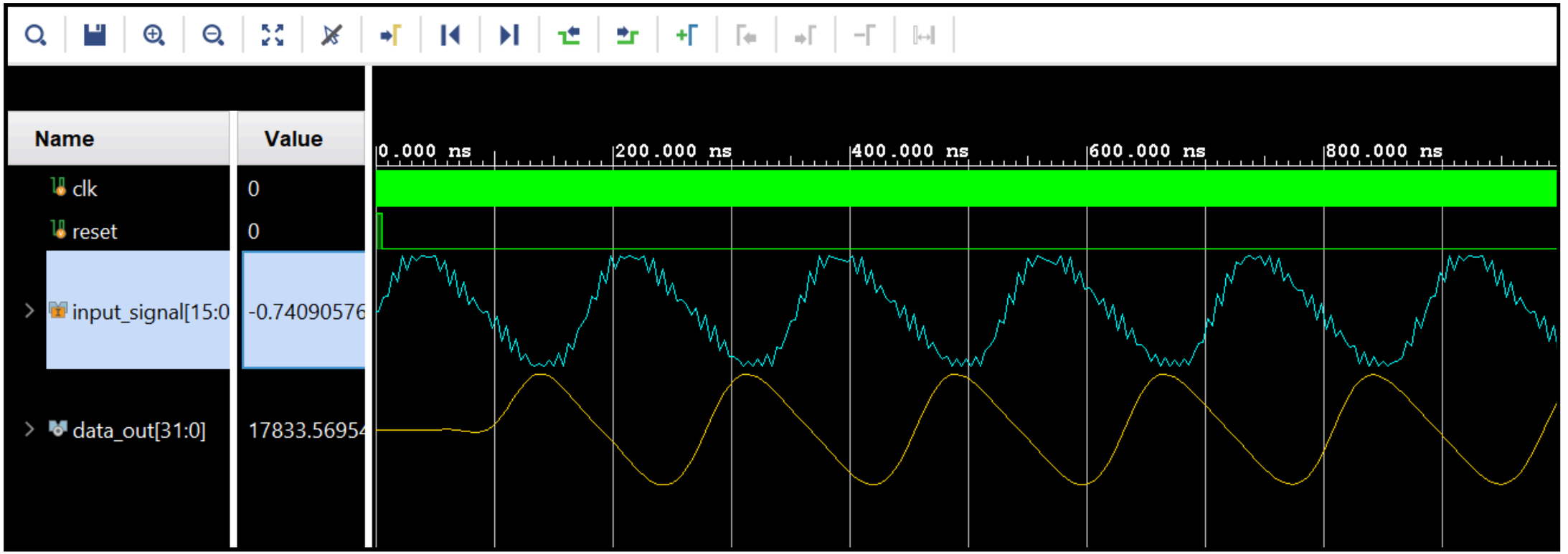
(thats why we added a rom for the input because we will use this method :)

2) using CORDIC algorithm

3) Design a DDS (Direct Digital Synthesizer) module

4) Using a numerically controlled oscillator (NCO) IP

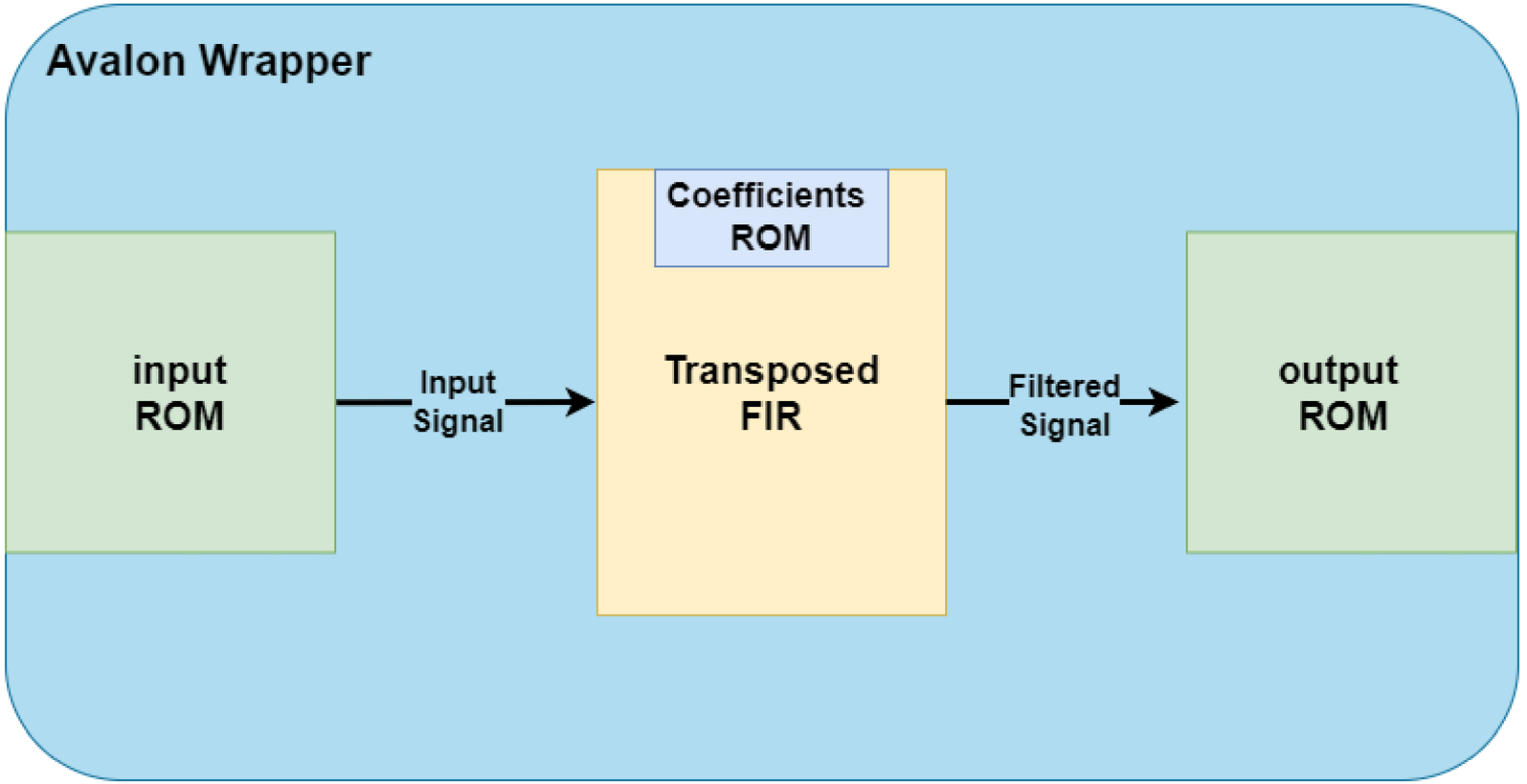
Simulation of FIR



Same as Matlab Ouput

Avalon Wrapper Interface

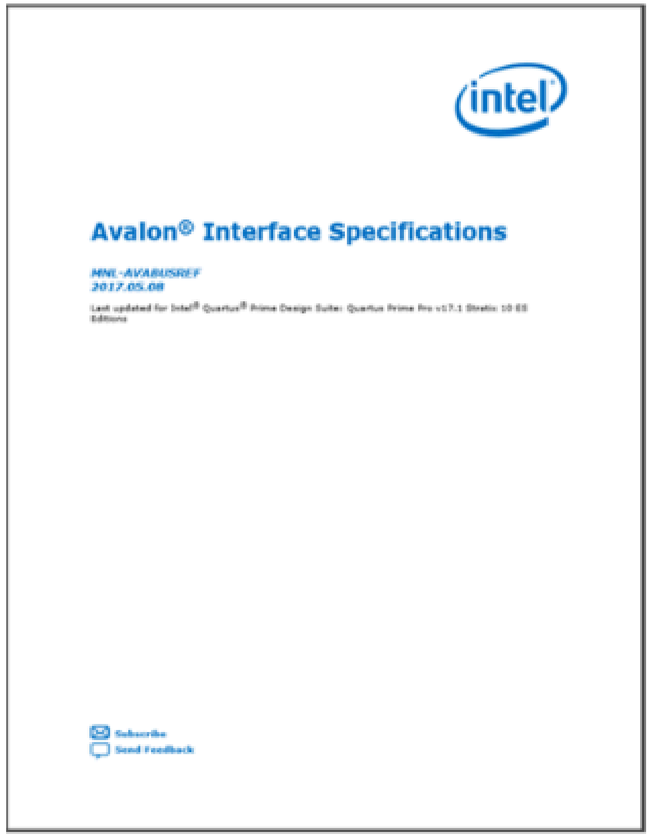
Our NIOS II processor communicates with the peripherals through Avalon interconnect so we need to make our design able to understand the avalon protocol that's why we are going to Wrap our design with avalon interface making it a custom avalon IP there are many interfaces and many prototcols like Avalon Memory-Mapped, Avalon ST, AXI & PCIe



Avalon® Interface Specification

Defines the entire Avalon interface standard

- Provides reference information on additional transfer types
  - Use cases
  - Waveform diagrams



Documentation Link
Avlaon Bus Spec

Avalon MM interface Concepts

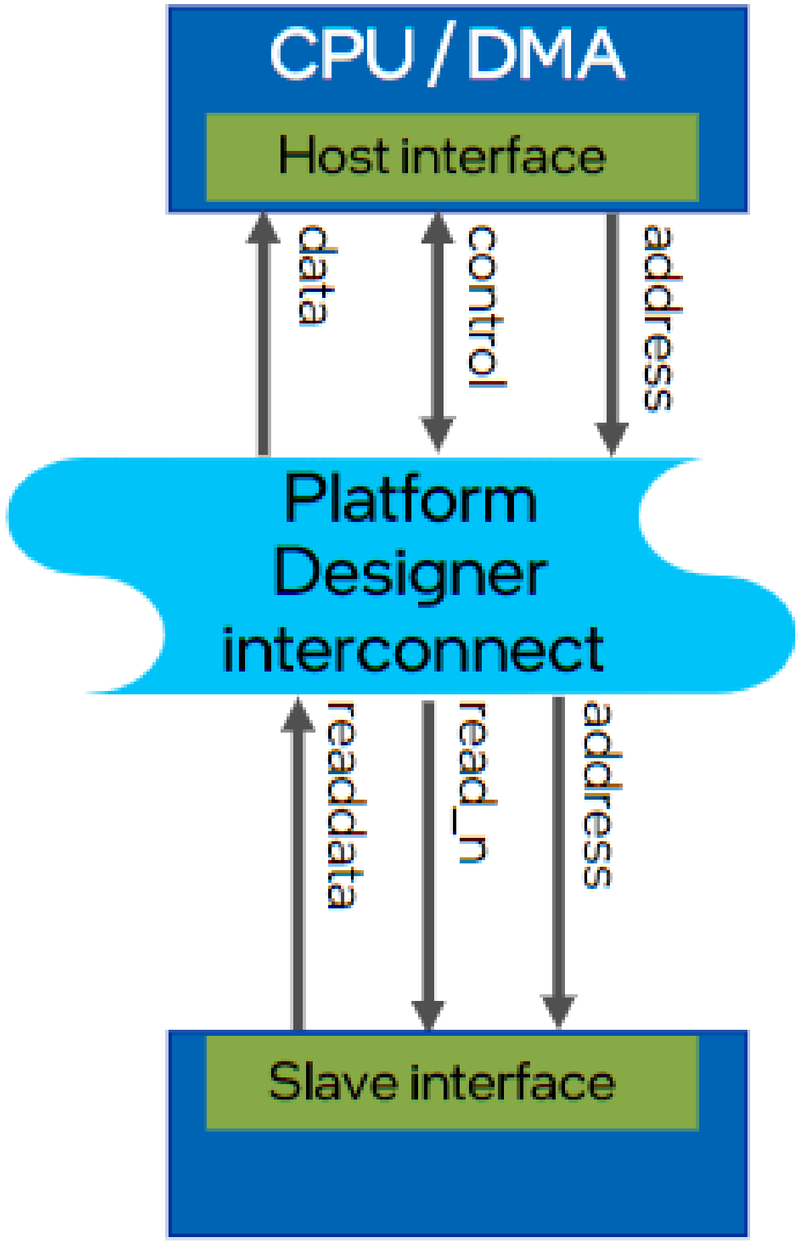
An Address-based protocol that allows components to communicate using read/write transfer

Host (Master) interfaces communicate with specific agents by issuing requests to agent address

Host Interface
initiates read/write transfers with the interconnect targeting address in address space

Agent (Slave) interfaces mapped to a location in address space

Agent interface
accepts and responds to transfer requests from interconnect
interconnect handles decoding of host address a to actual agent interface



Example host/agent connections

Avalon MM interface Concepts

One good thing here is that for slave interfaces all signals are optional meaning we do not have to design a full interface just one with the signals we need

we have 2 slave interfaces one for the input ROM to be able to configure it using NIOS II (Writable Slave)

the other for the output ROM to be able to read the filtered signal from it using NIOS II (Readable Slave)

Basic Avalon®-MM Slave Interface Signals

Signal Type	Width	Direction	Required	Description
address	1-64	Input	N	*Word* address of slave for transfer request *(discussed later)*
waitrequest **waitrequest_n**	1	Output	N	Allows slave to stall transfer until deasserted; other Avalon®-MM interface signals must be held constant
read **read_n**	1	Input	N	Indicates slave should respond to read request
readdata	8, 16, 32, 64, 128, 256, 512, 1024	Output	N	Data provided to Platform Designer interconnect in response to read request
write **write_n**	1	Input	N	Indicates slave should respond to write request
writedata	8, 16, 32, 64, 128, 256, 512, 1024	Input	N	Data from the Platform Designer interconnect for a write request
byteenable **byteenable_n**	2, 4, 8, 16, 32, 64, 128	Input	N	Specifies valid byte lane for readdata or writedata (width = data width / 8)
begintransfer **begintransfer_n**	1	Input	N	Asserts at the beginning (first cycle) of any transfer
response	2	Output	N	Indicates successful transfer or not, as well as whether access is to undefined address location; interconnect provides OK if no slave response

Avalon®-MM Interface Minimum Signal Requirements

Master interface

- Must have **address** and **waitrequest**
- To execute write transfers
  - **write**, **writedata**
- To execute read transfers
  - **read**, **readdata**

Slave interface

- No **address** or **waitrequest** necessary; single address location set in Platform Designer
- For a writable slave
  - **write**, **writedata**
- For a readable slave
  - **readdata**

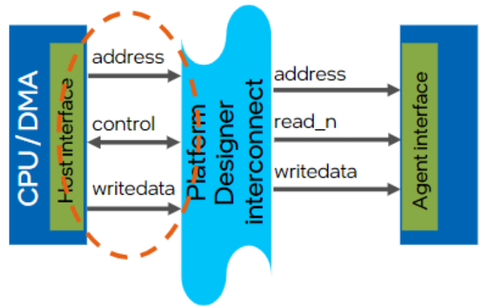
Avalon Wrapper Interface

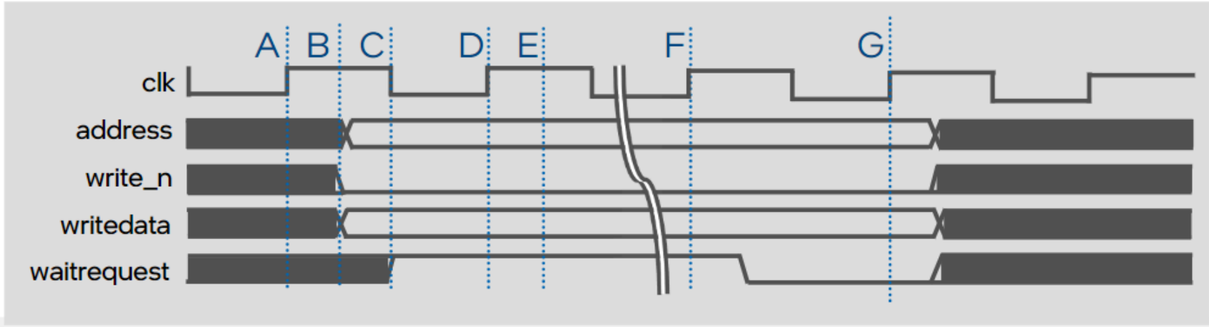
We want to Wrap our input memory with Avalon interface for Write Operation so we can write to it using Nios 2 processor while the reading part we will left it as its because the FIR is the one that reads from the input rom

input
ROM

Fundamental Host Write Transfer

- address, write_n, & writedata asserted
- Wait for waitrequest deassertion
- Transfer ends on next rising edge
- End of transfer





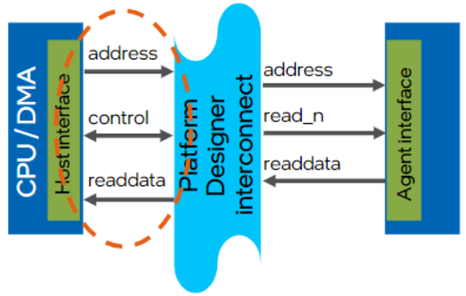
Avalon Wrapper Interface

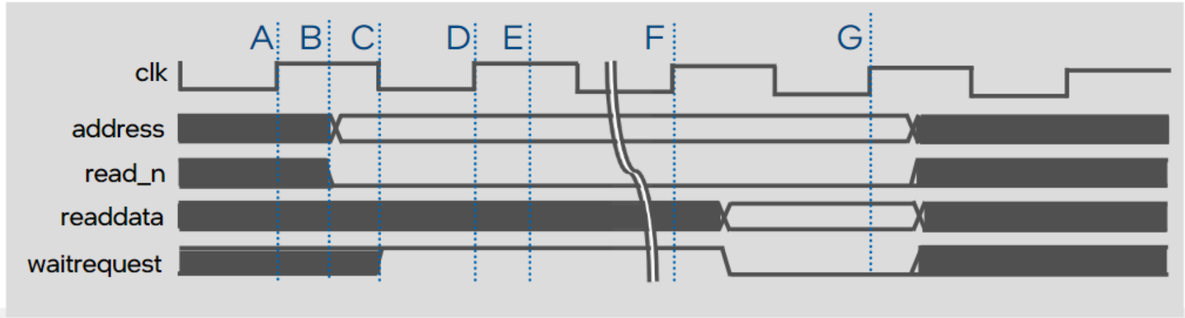
Same here but for Reading

output
ROM

Fundamental Host Read Transfer

- address & read_n asserted
- Wait for waitrequest deassertion
- Host samples readdata on next rising edge
- End of transfer





RTL for the Avalon write & read interface

The input ROM address ranges from 0x1000 to 0x2000
Output ROM ranges from 0x3000 to 0x4000

These addresses are important since in SOC stage we will assign for the FIR same range of address to be used for the Memory mapped interface

Input ROM

Normal Read

```
always@(posedge clk)
begin
  if(!write)
     readdata = ROM[count];
end
```

AvalonWrite

```
always@(posedge clk) begin
    if(write && (address >= 4096 && address <= 8192))
        ROM[address-4096] <= writedata;
end
```

Output ROM

Normal Write

```
always@(posedge clk) begin
  if(!read)
    ROM[count] <= writedata;
end
```

Avalon Read

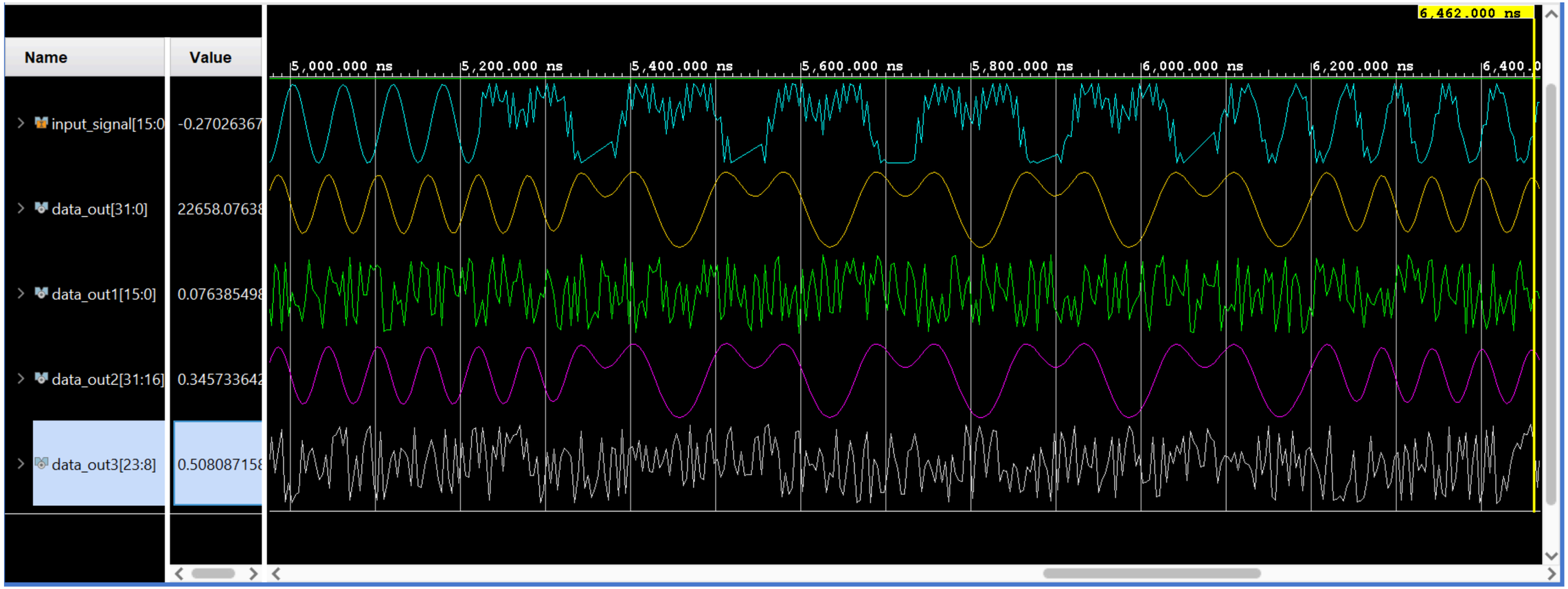
```
always@(posedge clk) begin
    if(read && (address >= 12288 && address <= 16384))
        readdata <= ROM[address-12288];
    end
```

NOTE:

Avalon protocol insists that all Input and output widths be the same
and since our input is 16 bit and our output is 32 bit this is a violation for the avalon protocol so we must either make the output 16 bits like the input and the coeffs or make the input and the coeffs 32 bits

We will make the output 16 bits but to make so we will have to pick which 16 bits out of the 32 bits will be the most accurate and represent our signal like the 32 bits did

Choosing the Most accurate 16 bits



As we see the Most Significant 16 bits are the most accurate ones the rest are just complete mess

Quartus Synthesis Tool Optimization

In our architecture for the FIR, we designed order one of the filters and then replicate it then we take the output from the last adder
but the output of the last Z block is not used
the synthesis tool think that because you didn’t use this output then all the registers assoicated with it are just area for no need so it optimizes and removes them

When we compile the design and look at the resources Utilization

Top-level Entity Name	FIR_transposed
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	98 / 6,272 (2 %)
Total registers	65
Total pins	34 / 92 (37 %)

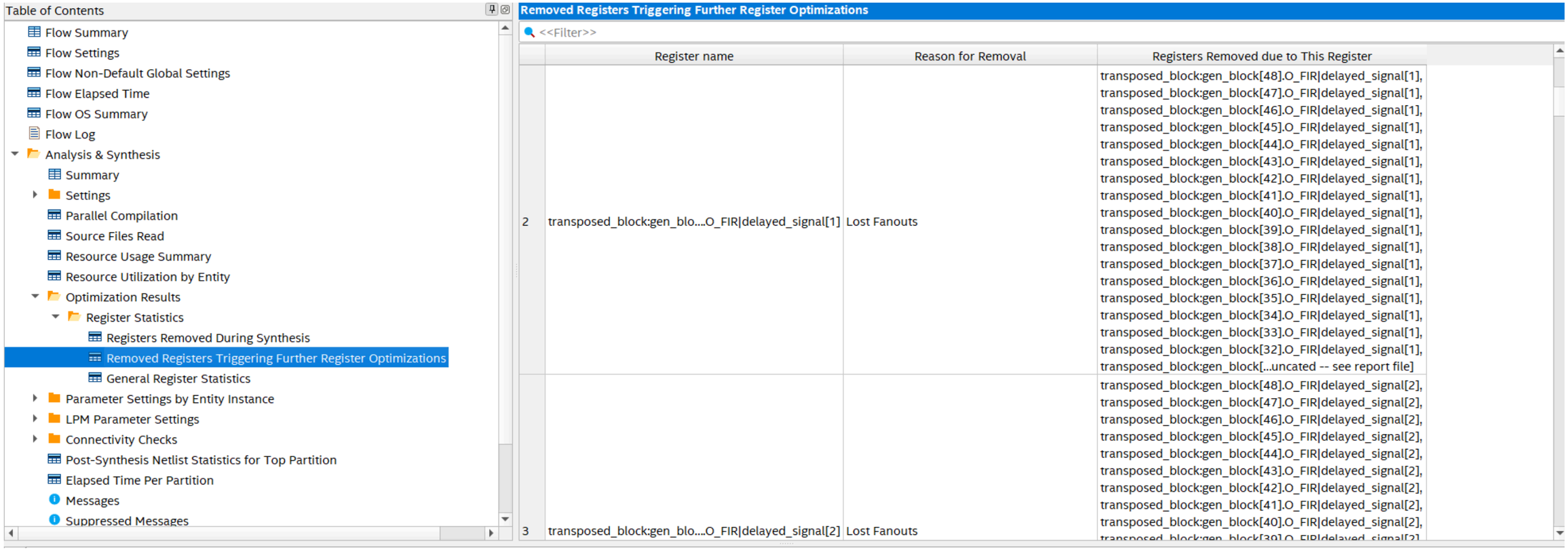
If we want to know what optimizations the synthesis tool did go to ‘’Analysis & Synthesis” Folder >> “Optimization Results” >> “Register Statistics” and then look at Registers removed and the reason for optimization

Registers Removed During Synthesis

<<Filter>>

	Register name	Reason for Removal
...	transposed_block:gen_block[24].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[24].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[25].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[25].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[26].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[26].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[27].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[27].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[28].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[28].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[29].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[29].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[30].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[30].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[31].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[31].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[32].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[32].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[33].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[33].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[34].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[34].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[35].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[35].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[36].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[36].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[37].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[37].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[38].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[38].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[39].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[39].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[40].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[40].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[41].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[41].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[42].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[42].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[43].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[43].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[44].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[44].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[45].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[45].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[46].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[46].O_FIR|delayed_signal[16]
...	transposed_block:gen_block[47].O_FIR|delayed_signal[17..31]	Merged with transposed_block:gen_block[47].O_FIR|delayed_signal[16]
...	Total Number of Removed Registers = 1535	

If we want to know what optimizations the synthesis tool did go to ‘’Analysis & Synthesis” Folder >> “Optimization Results” >> “Register Statistics” and then look at Registers removed and the reason for optimization



As you saw there are more than 1500 Registers that got optimized which is about the whole design when we looked at the reasons we found that some got merged after we picked the most accurate 16 bits and some are lost due to fanout meaning they are connected to output but you didnt use this output and that is logical becuase you use the output of the adder not the Z block

We will solve this issue by forcing the synthesis tool not to optimize these 2 cases by using smth in quartus called **Preserve For Debug**

From toolbox choose “assignments’’ then “assignment editor’’ under To chosse the removed registers and under value give On and under Enabled give Yes

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
81	✔		transposed_block:gen_block[39].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
82	✔		transposed_block:gen_block[40].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
83	✔		transposed_block:gen_block[40].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
84	✔		transposed_block:gen_block[41].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
85	✔		transposed_block:gen_block[41].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
86	✔		transposed_block:gen_block[42].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
87	✔		transposed_block:gen_block[42].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
88	✔		transposed_block:gen_block[43].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
89	✔		transposed_block:gen_block[43].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
90	✔		transposed_block:gen_block[44].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
91	✔		transposed_block:gen_block[44].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
92	✔		transposed_block:gen_block[45].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
93	✔		transposed_block:gen_block[45].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
94	✔		transposed_block:gen_block[45].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
95	✔		transposed_block:gen_block[46].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
96	✔		transposed_block:gen_block[46].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
97	✔		transposed_block:gen_block[46].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
98	✔		transposed_block:gen_block[47].O_FIR|delayed_signal	Disable Register Merging	On	Yes	FIR_transposed		
99	✔		transposed_block:gen_block[47].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
100	✔		transposed_block:gen_block[48].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		
101	✔		transposed_block:gen_block[49].O_FIR|delayed_signal	Preserve Fan-out Free Register Node	On	Yes	FIR_transposed		

When set to On, this option prevents the specified register from merging with other registers, and prevents other registers from merging with the specified register.

Save then Compile Again

as u see Utilization is now more normal then before

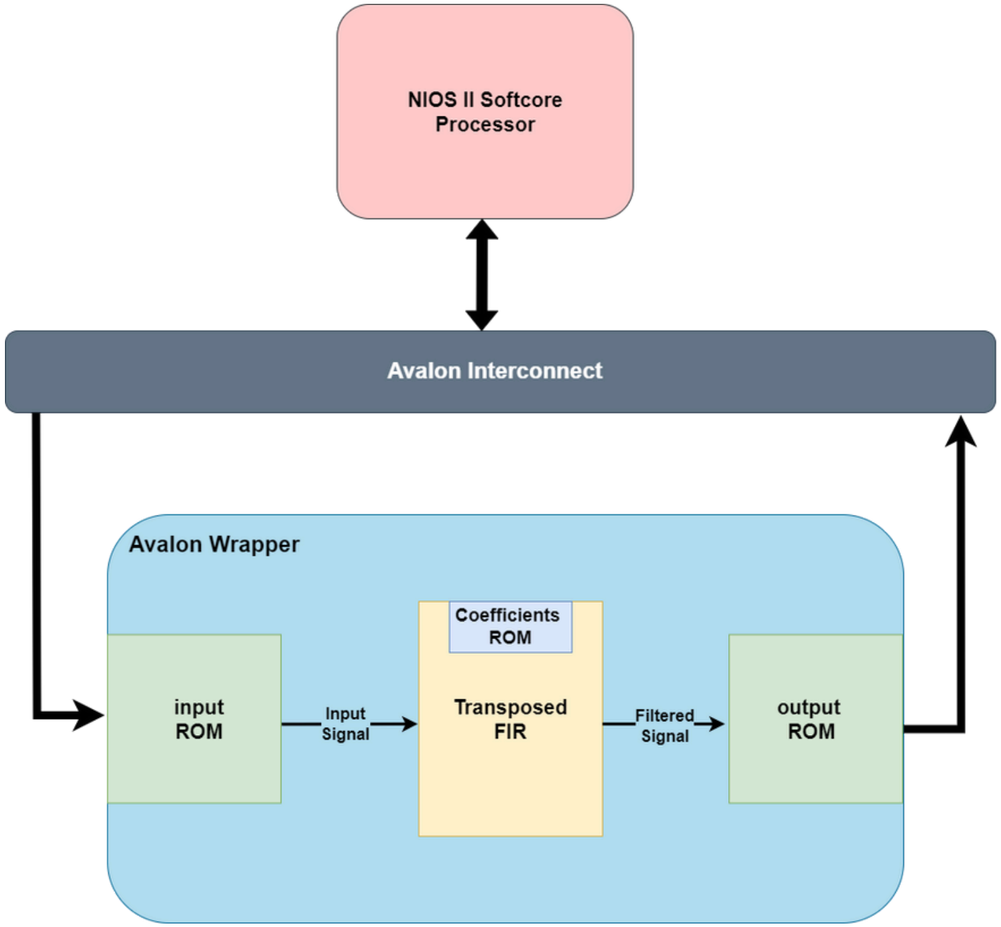
u can also look again at the optimized registers

you will find none :)

Top-level Entity Name	FIR_transposed
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	1,632 / 6,272 (26 %)
Total registers	1600

SoC Hardware

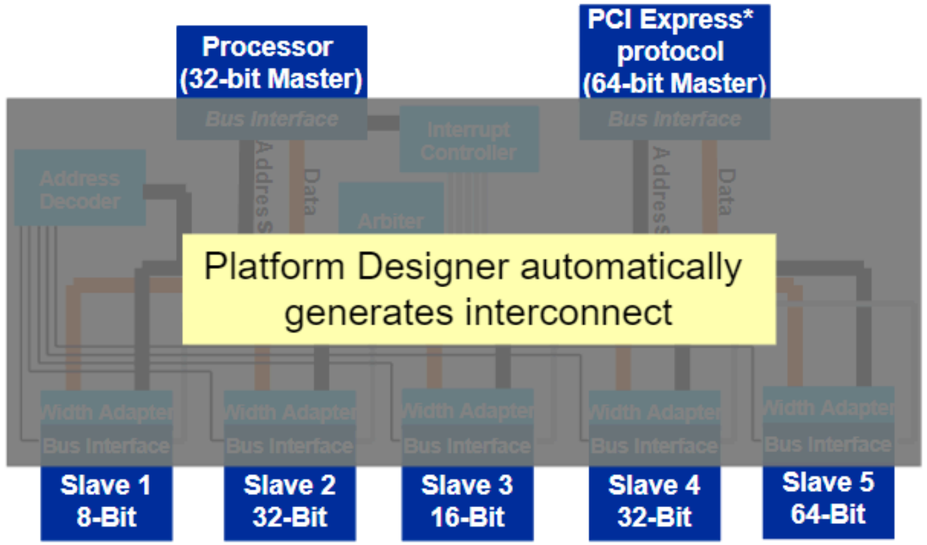
We will now head to Platform Designer in Quartus Prime to Make our SOC by adding our custom avalon IP and connect it with NIOS II also adding On Chip RAM and ROM for data and instruction also a JTAG UART for debugging



Another important thing the Avalon interconnect from the above arch that handles everything is self generated with Platform designer so no need to design it

Automatic Interconnect Generation

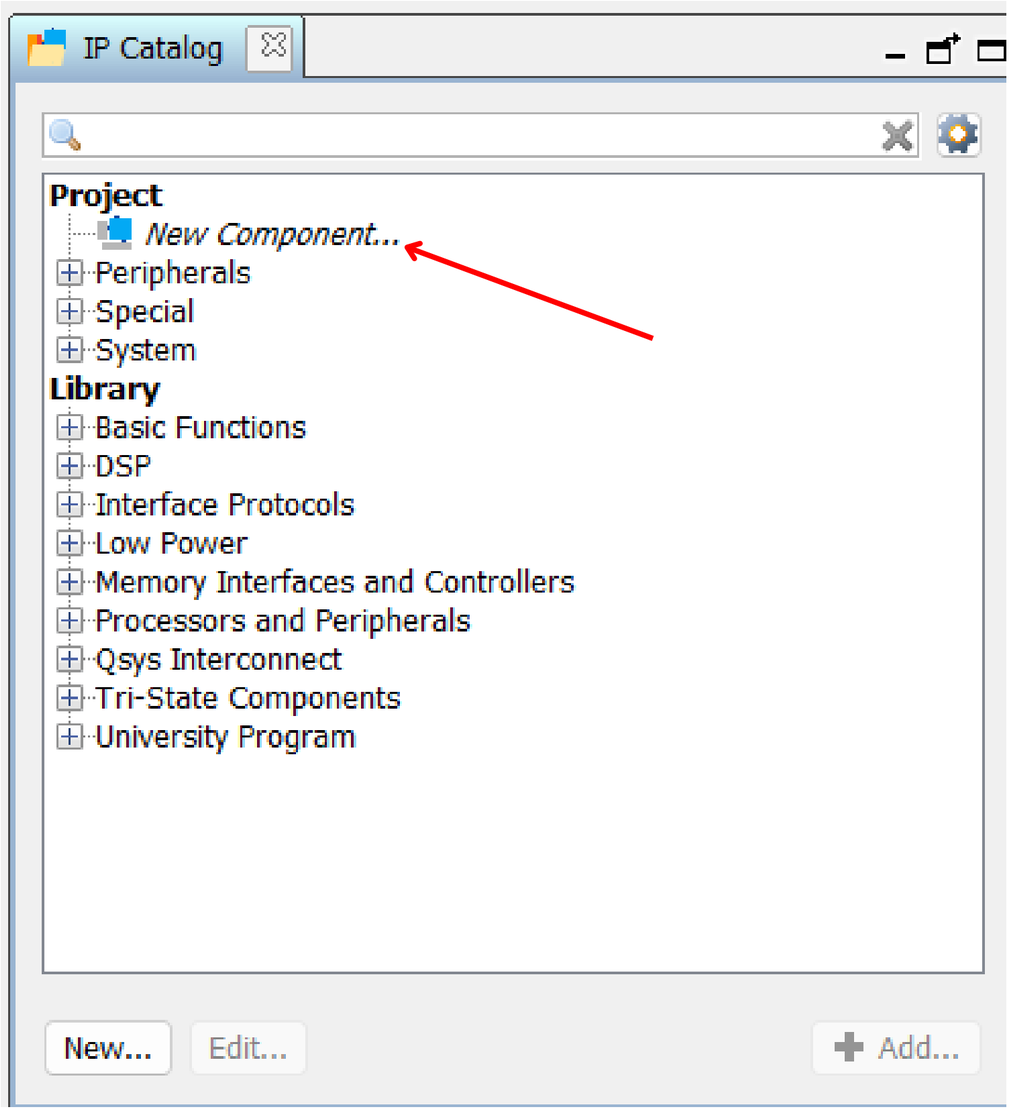
- Avoids error-prone integration
- Saves development time with automatic logic & HDL generation
- Enables you to focus on value-add blocks



Platform Designer improves productivity by automatically generating the system interconnect logic

Adding our Custom IP (1)

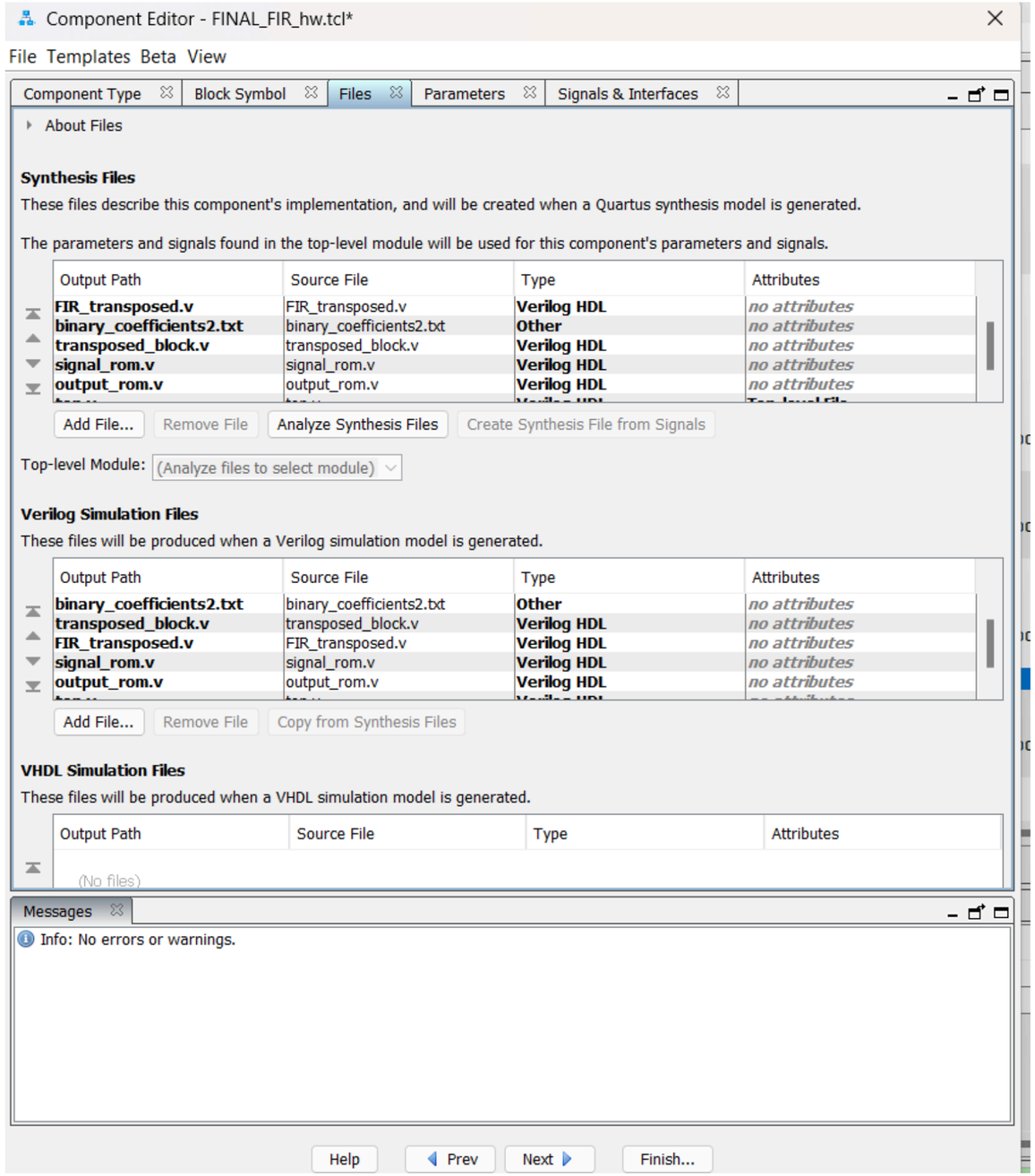
To add a new Custom IP choose ‘’New Component’’



Adding our Custom IP (2)

Here Under Files tab in Synthesis files You add all design files including the coefficients text file

also in simulation files you need to add the same files as in synthesis adding files in simulation is optional but since we are going to simulate the SOC using modelsim then this step is crucial

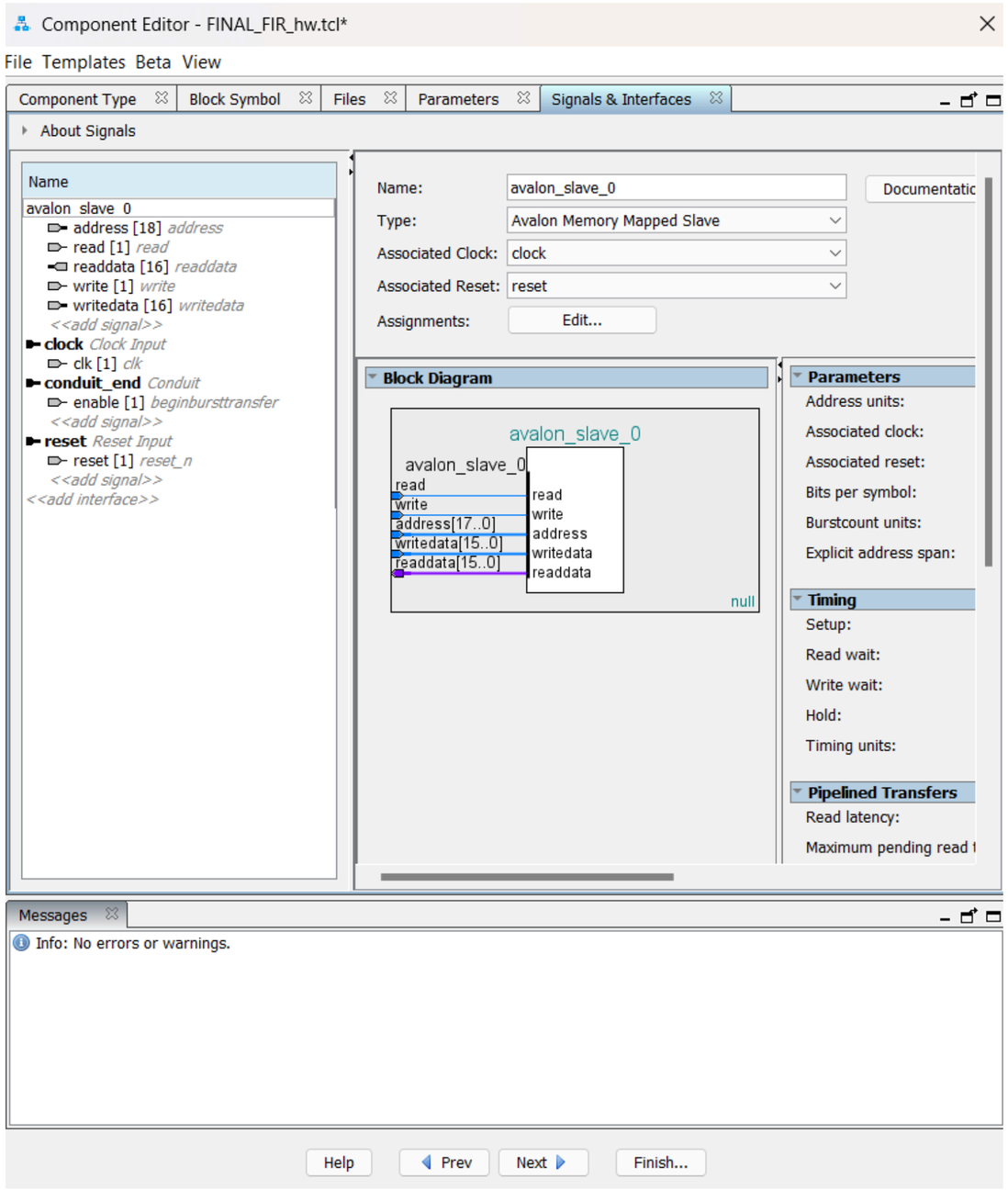


Adding our Custom IP (3)

Under the Signals & interfaces tab, you organize each signal under its right category for address, read , write,readdata and write data all lies under the Avalon slave interface category

clock for clk signal reset for reset and you choose whether it’s active low or high

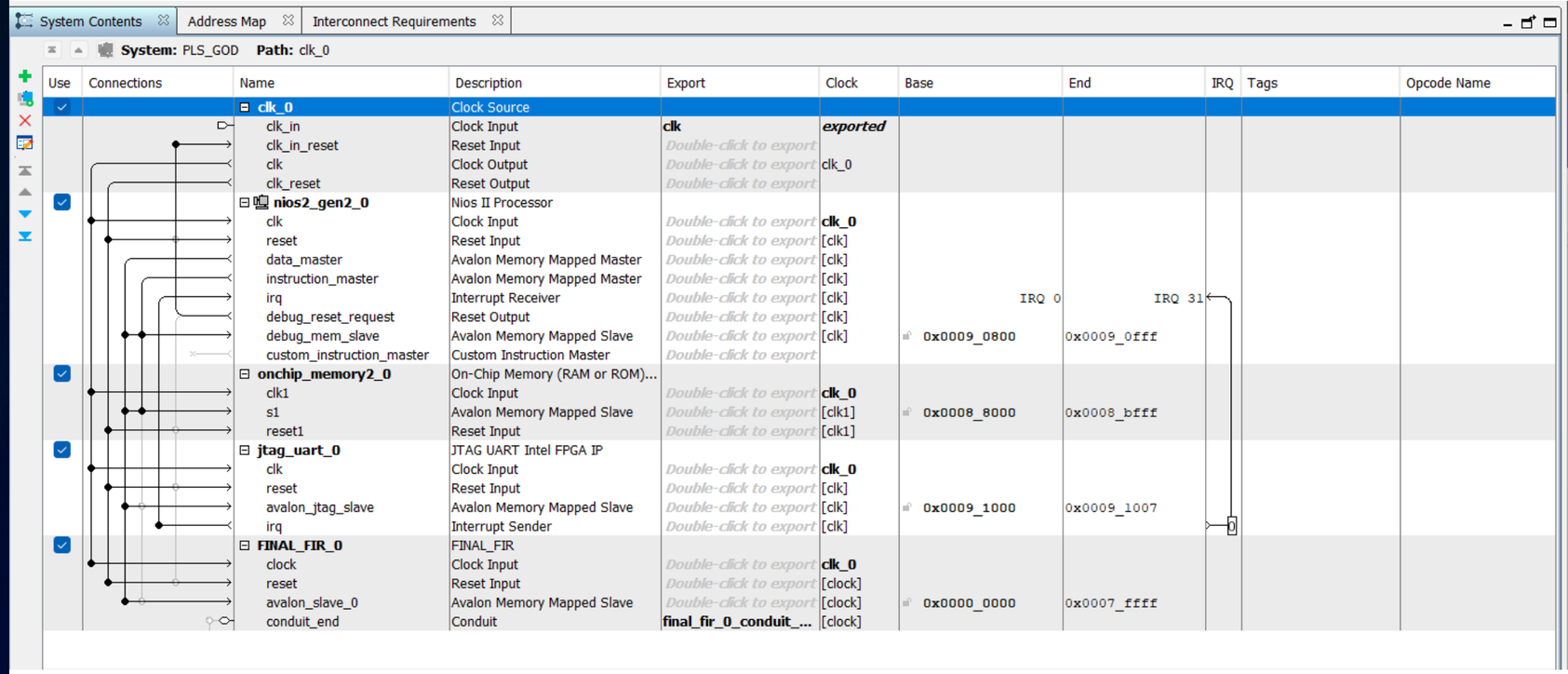
and lastly, the Conduit are the signals you want to export as input or output signal to the design which will be connected to IO of the FPGA board in our case i have an “enable” signal which is an input and i will connect it into a switch on the fpga board to enable the filtering process



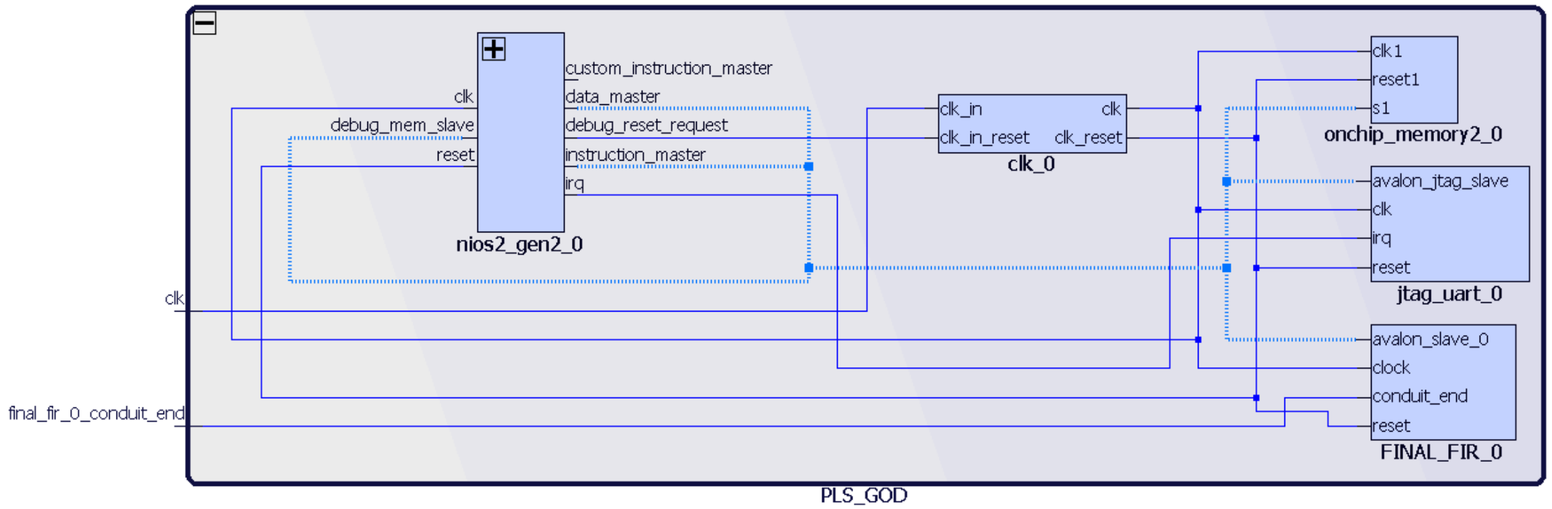
SoC Hardware Development (1)

Now in platform designer we add the components and connect them with each other
also we specify the address range for each component for FIR it took from 0x0000_0000 >>> 0x0007_FFFF

a small reminder we made our avalon interface for the input ROM to start at address 0x1000 >> 0x2000
and for the output ROM 0x3000 >> 0x4000 so both of them lies inside the FIR range so we are good to go

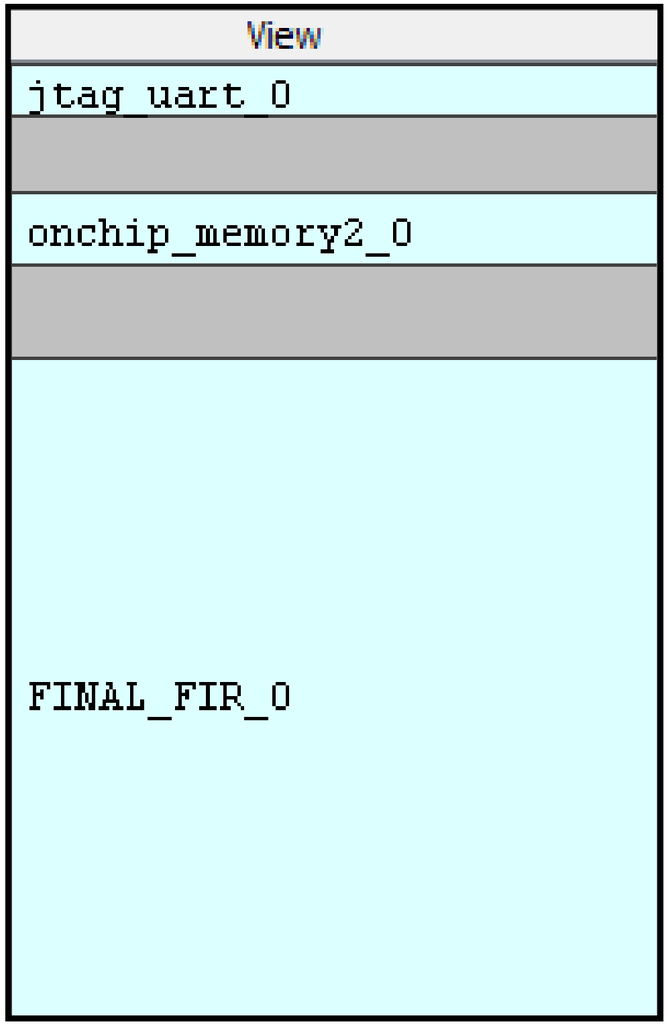


SoC Schematic



SoC Hardware Development (2)

SoC Memory Map



Device	Address Range	Size (bytes)
FINAL_FIR_0	0x00000000 - 0x0007FFFF	524288
onchip_memory2_0	0x00088000 - 0x0008BFFF	16384
jtag_uart_0	0x00091000 - 0x00091007	8

SoC Hardware Development (3)

Resource Utilization

Flow Summary

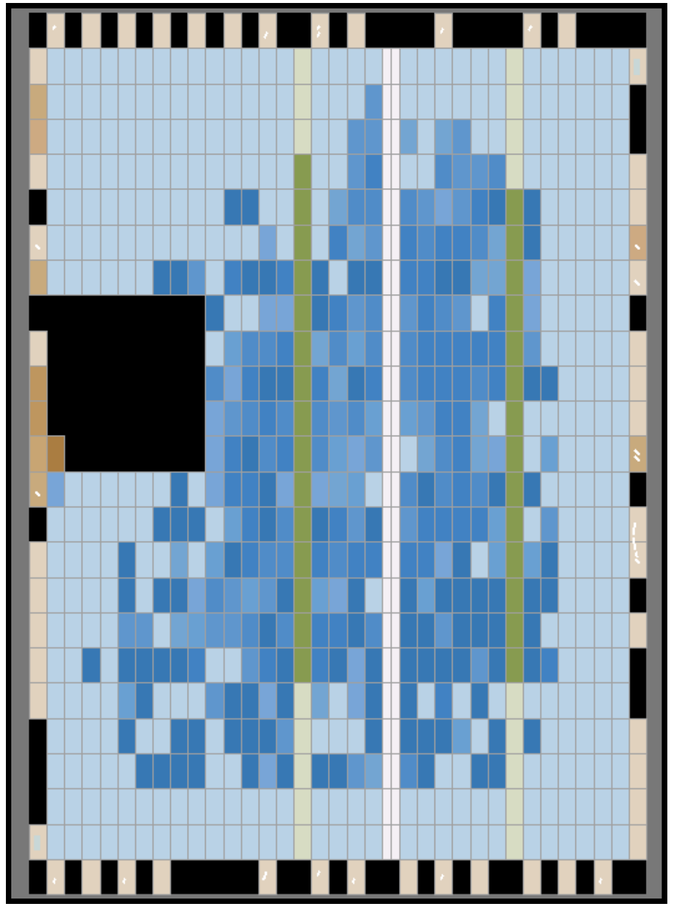
<<Filter>>

Flow Status	Successful - Thu May 16 17:44:59 2024
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	FINAL_SOC_FIR
Top-level Entity Name	PLS_GOD
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	3,538 / 6,272 (56 %)
Total registers	2635
Total pins	2 / 92 (2 %)
Total virtual pins	0
Total memory bits	206,368 / 276,480 (75 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

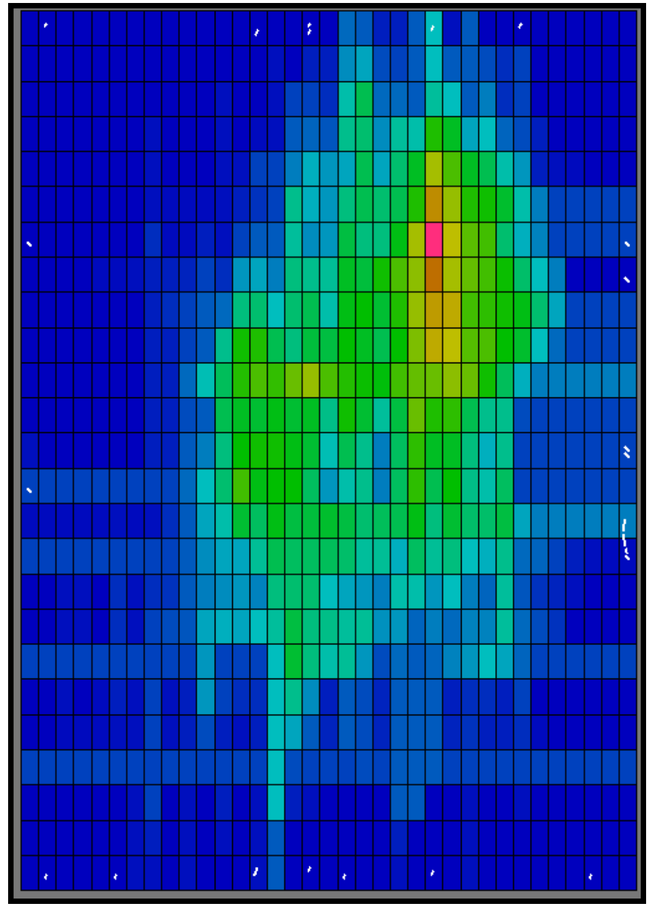
SoC Hardware Development (4)

Chip Planner

Floor Plan



Routing Utlization



SoC Hardware Development (5)

Power Analysis

Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Thu May 16 17:39:27 2024
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	FINAL_SOC_FIR
Top-level Entity Name	PLS_GOD
Family	Cyclone IV E
Device	EP4CE6E22C8
Power Models	Final
Total Thermal Power Dissipation	59.76 mW
Core Dynamic Thermal Power Dissipation	0.65 mW
Core Static Thermal Power Dissipation	43.21 mW
I/O Thermal Power Dissipation	15.90 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

SoC Hardware Development (6)

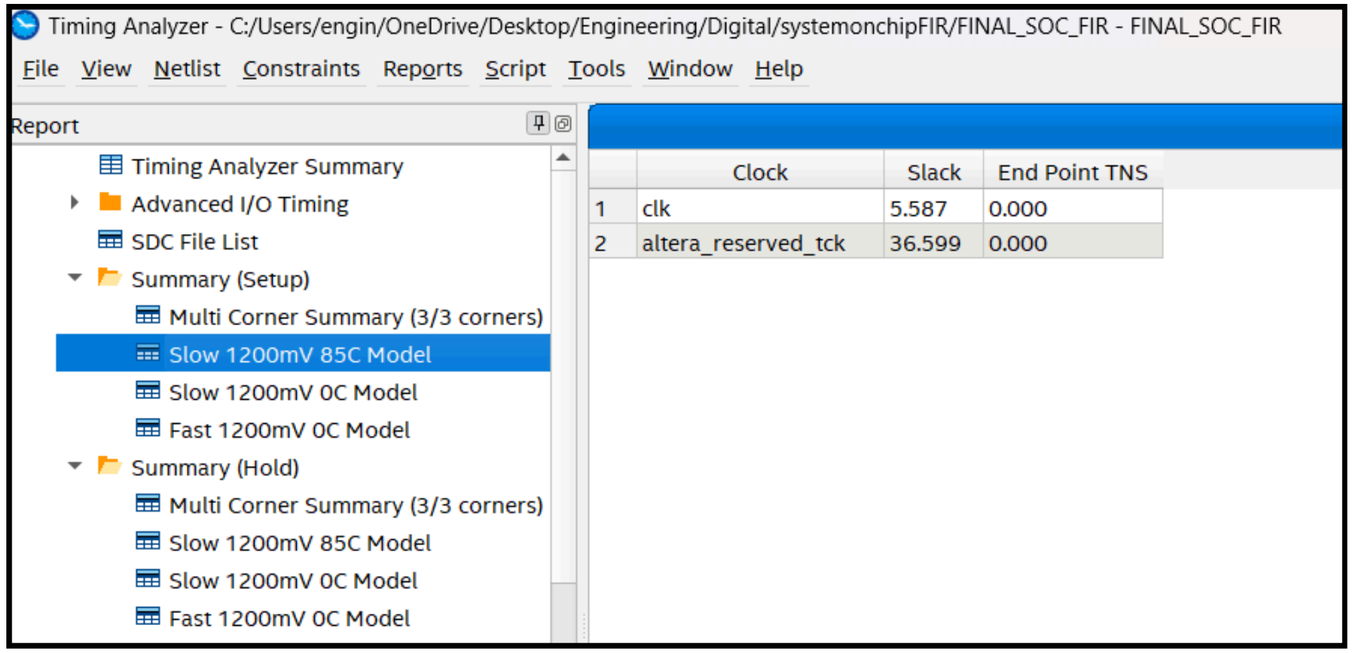
Timing Analysis

We are using the 50 MHz clock of the FPGA while the max is 70 MHZ

Altera_reserved_tck is set to 10 MHz as mentioned by the time quest cookbook

Slow 1200mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	37.31 MHz	37.31 MHz	altera_reserved_tck	
2	69.38 MHz	69.38 MHz	clk	



SoC Software (C code)

Mainly We will have 2 Codes
one to write any signal into the input ROM
and the other Reads the Filtered Signal from the output ROM and compare the filtered signal with the filtered signal with matlab again

Mapping addresses

Our master (NIOS II) is 32 bit and the Slave (FIR) is 16 bits then we must know how to map between these two

If you want to write smth from nios (32 bit) to Fir (16 bit) then one location from nios will be translated into 2 locations from fir and How is this handeld you can look at the next slide or read the documentation for more info

Dynamic Bus Sizing (Narrow Slave)

Platform Designer interconnect handles the local addressing dynamically and automatically

- Remember:
  - Slaves use ***word*** addressing
  - Masters use ***byte*** addressing

C Code (1)

hello_world_small.c | io.h

```c
#include "sys/alt_stdio.h"
#include "system.h"
#include "alt_types.h"
#include "io.h"
#include <stdio.h>



//    while (1) {

#define MEMORY_BASE_ADDRESS 0x00001000
#define MEMORY_BASE_ADDRESS2 0x3000

int main()
{
    alt_u32 samples[] = {
            0b00100000101111100000000000000000,
            0b01011100000010100010001110101101,
            0b01111111111111110100010011000100,
            0b01111111111111111011000010111111,
            0b01111111111111111011110101010000110,
            0b01111110010110110111111111111111,
            0b01011101101101000111111111111111,
            0b00111100100111110111011001000011,
            0b00100010001100000110000101110010,
            0b00010001001100010100010001001111,
            0b00000111101101110010000110101110,
```

This C code is responsible for writing the signal into the input ROM of the FIR

each index of the array is 2 16 bit sample of the input signal combined togehter because NIOS II is 32 bit

Automating Signal Pairs Combination

Since we got hunderds of samples we cant combine each pair of samples together so we will automate this process using a python script one for combining pair of 16 bit samples into one 32 bit and one putting a comma at the end of each line to fit into the C array syntax

Momen > comma.py > ...

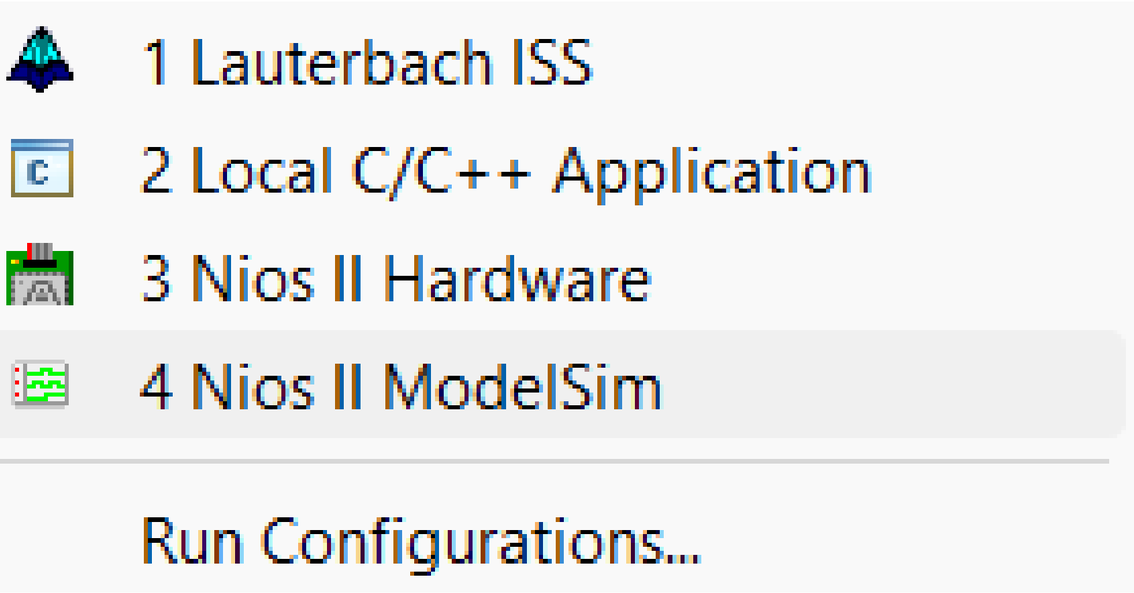
```
1   input_file = "sine_wave_nios_final.txt"
2   output_file = "formatted_sine_wave_final.txt"
3
4   with open(input_file, "r") as f_in, open(output_file, "w") as f_out:
5       for line in f_in:
6
7
8
9
10
11  pr
12
```

bit16_bit32.py > ...

```
def convert_binary_lines(input_file, output_file):
    """Combines pairs of 16-bit binary lines into 32-bit lines in a new file,
    handling any number of lines (up to 2000)."""

    with open(input_file, "r") as infile, open(output_file, "w") as outfile:
        lines = infile.readlines()

        # Handle the case where the number of lines is odd
        if len(lines) % 2 != 0:
            lines.append("0000000000000000")  # Add a padding line for the last odd

        # Process lines in pairs
        for i in range(0, len(lines), 2):
            combined = lines[i + 1].strip() + lines[i].strip()
            outfile.write(combined + "\n")
```

To Check the rest of the code go to GitHub

SoC Simulation Using Modelsim

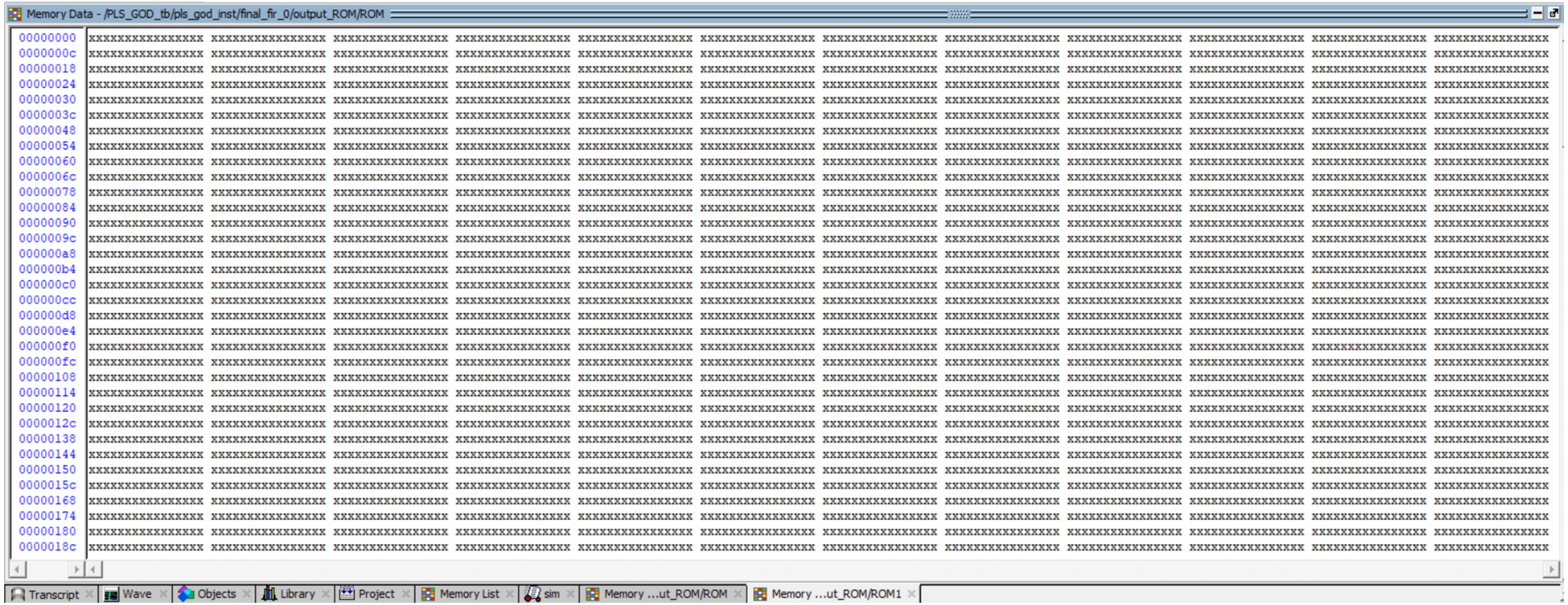


SoC Simulation Using Modelsim (1)

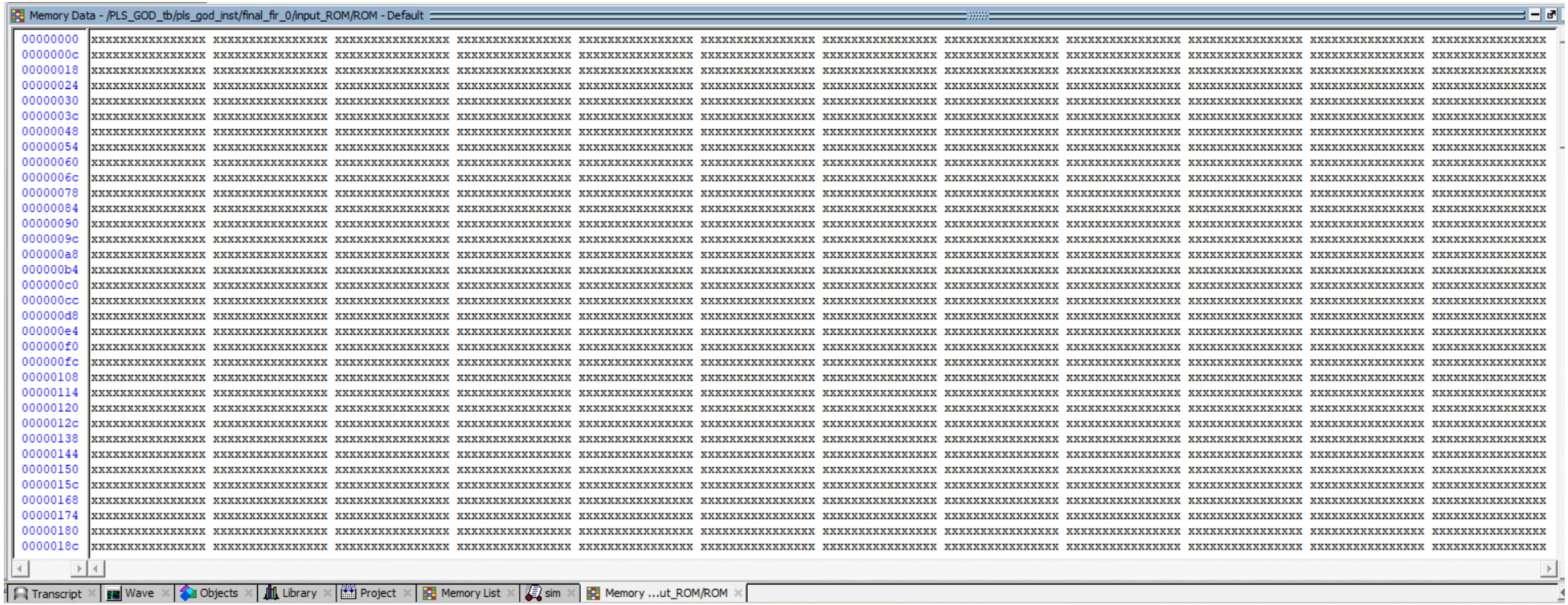
In Eclipse IDE after building project click on “Run” from tool bar then “Run as” you can either run as nios II modelsim meaning to simulate the SOC hardware and software for debugging which we will use now

or NIOS II hardware when the FPGA is programmed with your hardware and you want to upload the code the NIOS ROM

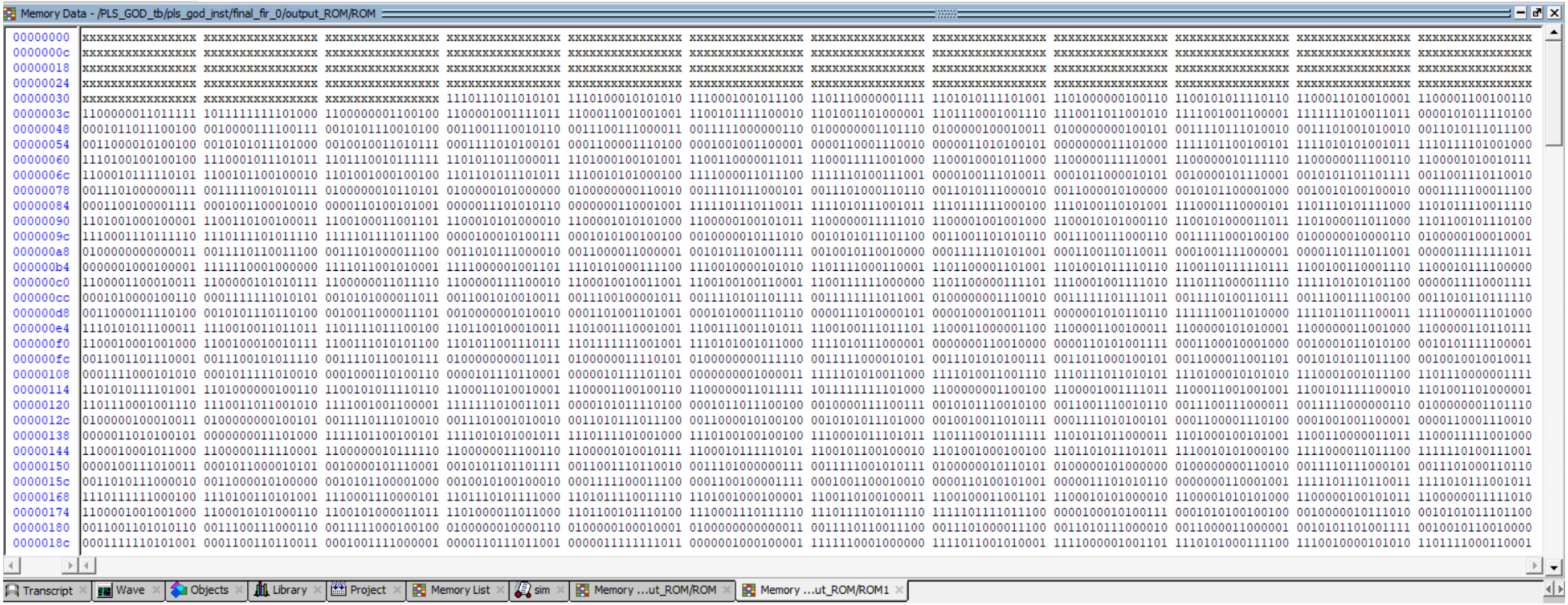
Output memory at start



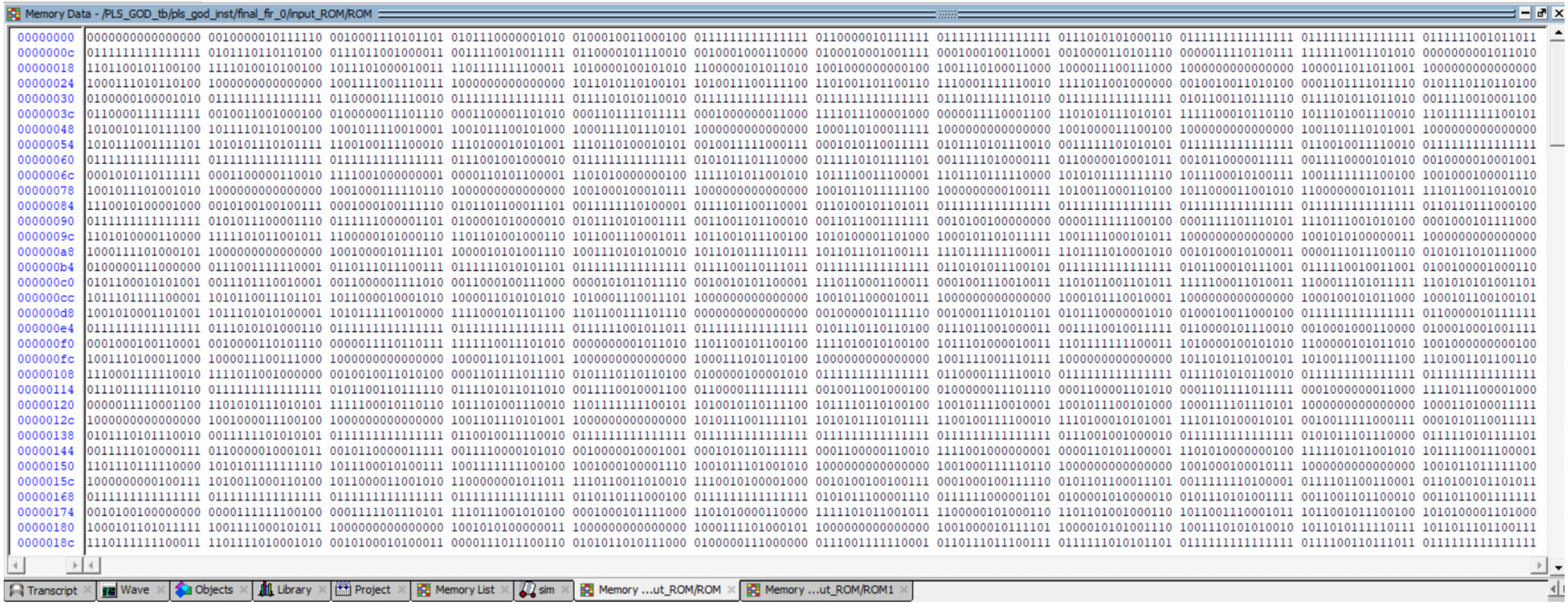
Input memory at start

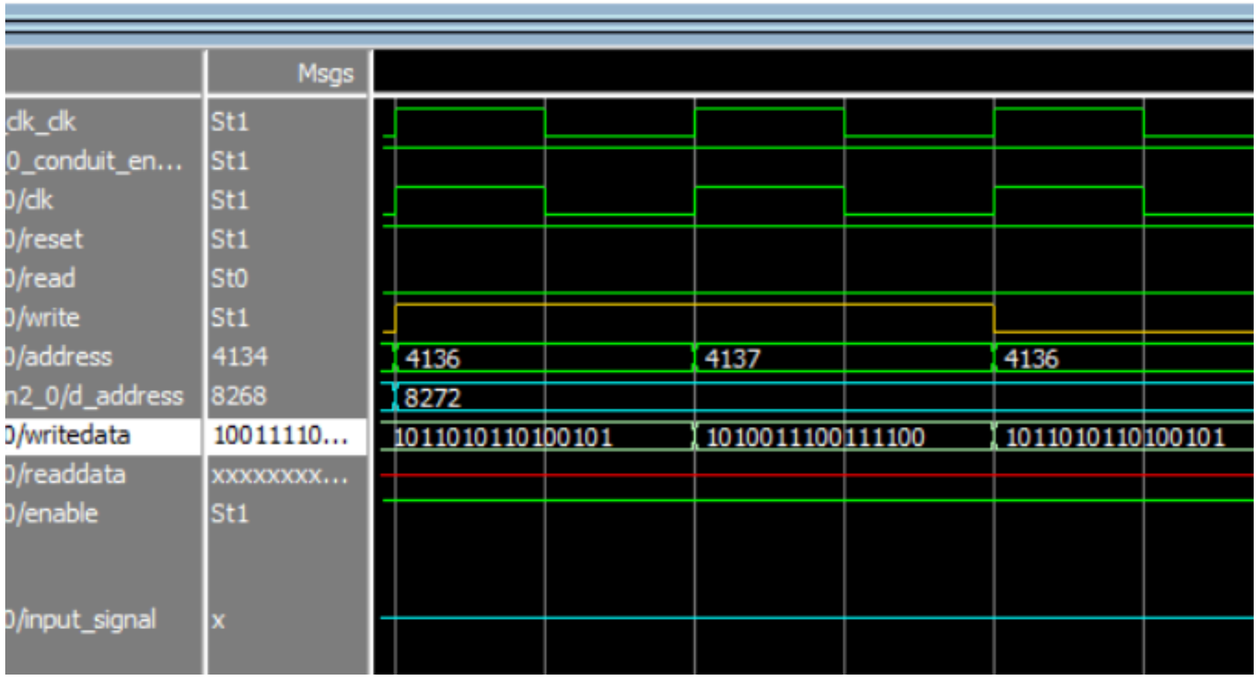


Output memory after excuting the code and filtering the input signal



Input Memory after signal has been written





SoC Simulation Using Modelsim (1)

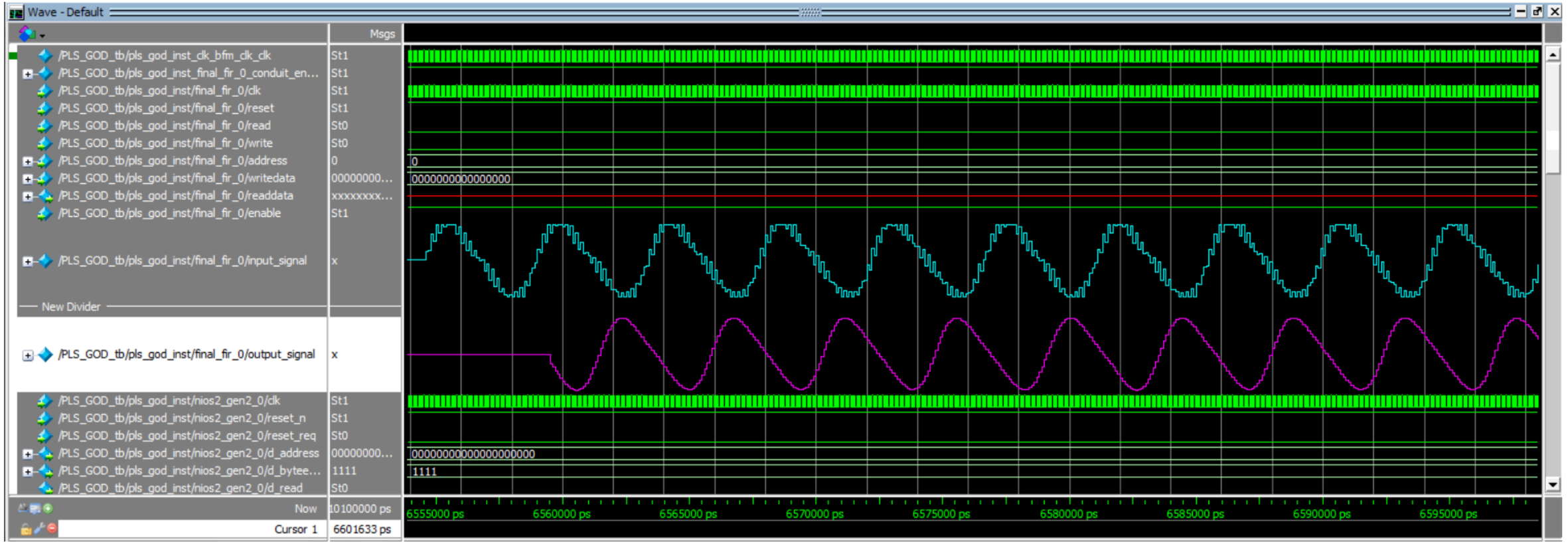
This is how one write operation is made same as we saw in address mapping slide the address issued by nios is divided by 2 that’s your first address and the 2nd is divided by +1
so one write operation for nios is 2 write operation for the FIR

Writing started

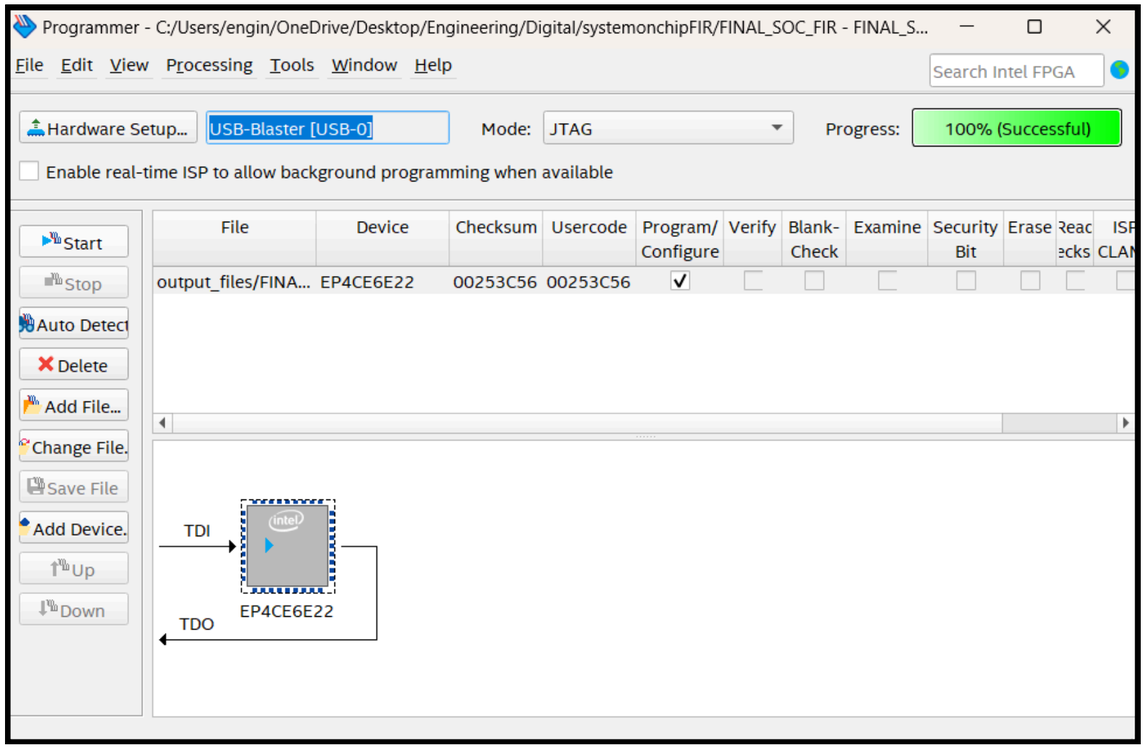
Writing ended

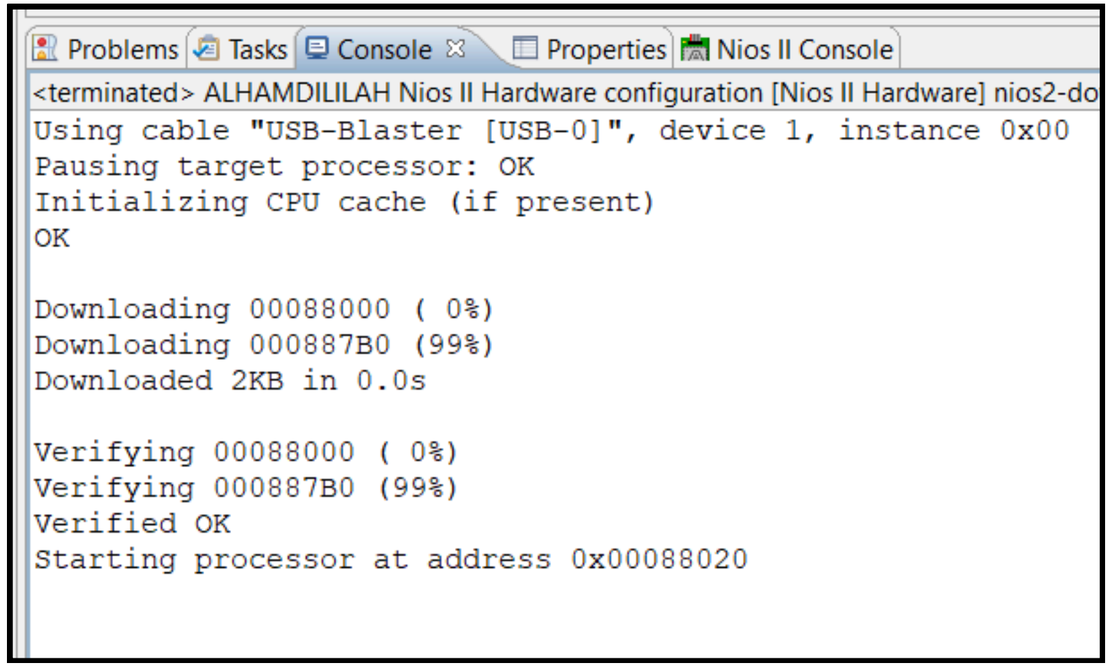
Filtered signal

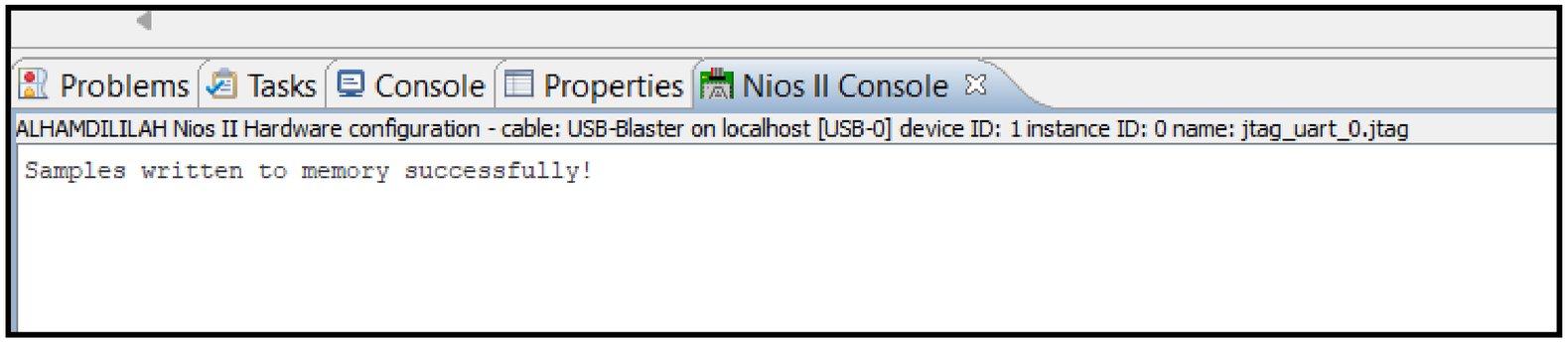
input signal


Wave - Default
Msgs
/PLS_GOD_tb/pls_god_inst_clk_bfm_clk_clk
/PLS_GOD_tb/pls_god_inst_final_fir_0_conduit_en...
/PLS_GOD_tb/pls_god_inst/final_fir_0/clk
/PLS_GOD_tb/pls_god_inst/final_fir_0/reset
/PLS_GOD_tb/pls_god_inst/final_fir_0/read
/PLS_GOD_tb/pls_god_inst/final_fir_0/write
/PLS_GOD_tb/pls_god_inst/final_fir_0/address
/PLS_GOD_tb/pls_god_inst/final_fir_0/writedata
/PLS_GOD_tb/pls_god_inst/final_fir_0/readdata
/PLS_GOD_tb/pls_god_inst/final_fir_0/enable
/PLS_GOD_tb/pls_god_inst/final_fir_0/input_signal
New Divider
/PLS_GOD_tb/pls_god_inst/final_fir_0/output_signal
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/clk
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/reset_n
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/reset_req
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/d_address
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/d_bytee...
/PLS_GOD_tb/pls_god_inst/nios2_gen2_0/d_read
St1
St1
St1
St1
St0
St0
0
00000000...
xxxxxxxx...
St1
x
x
St1
St1
St0
00000000...
1111
St0
0
0000000000000000
00000000000000000000
1111
Now
10100000 ps
Cursor 1
6601633 ps
6555000 ps
6560000 ps
6565000 ps
6570000 ps
6575000 ps
6580000 ps
6585000 ps
6590000 ps
6595000 ps

Programming the FPGA



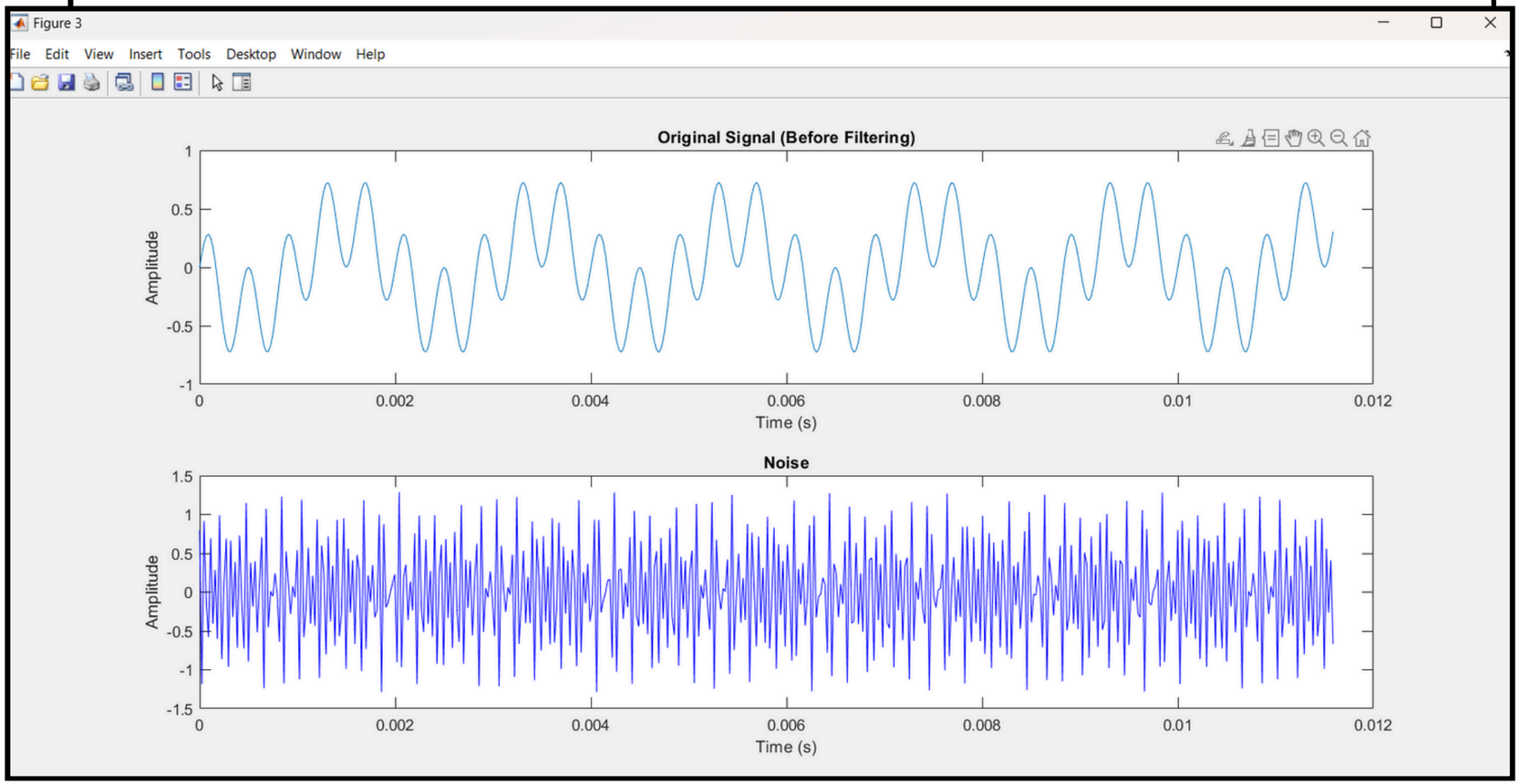


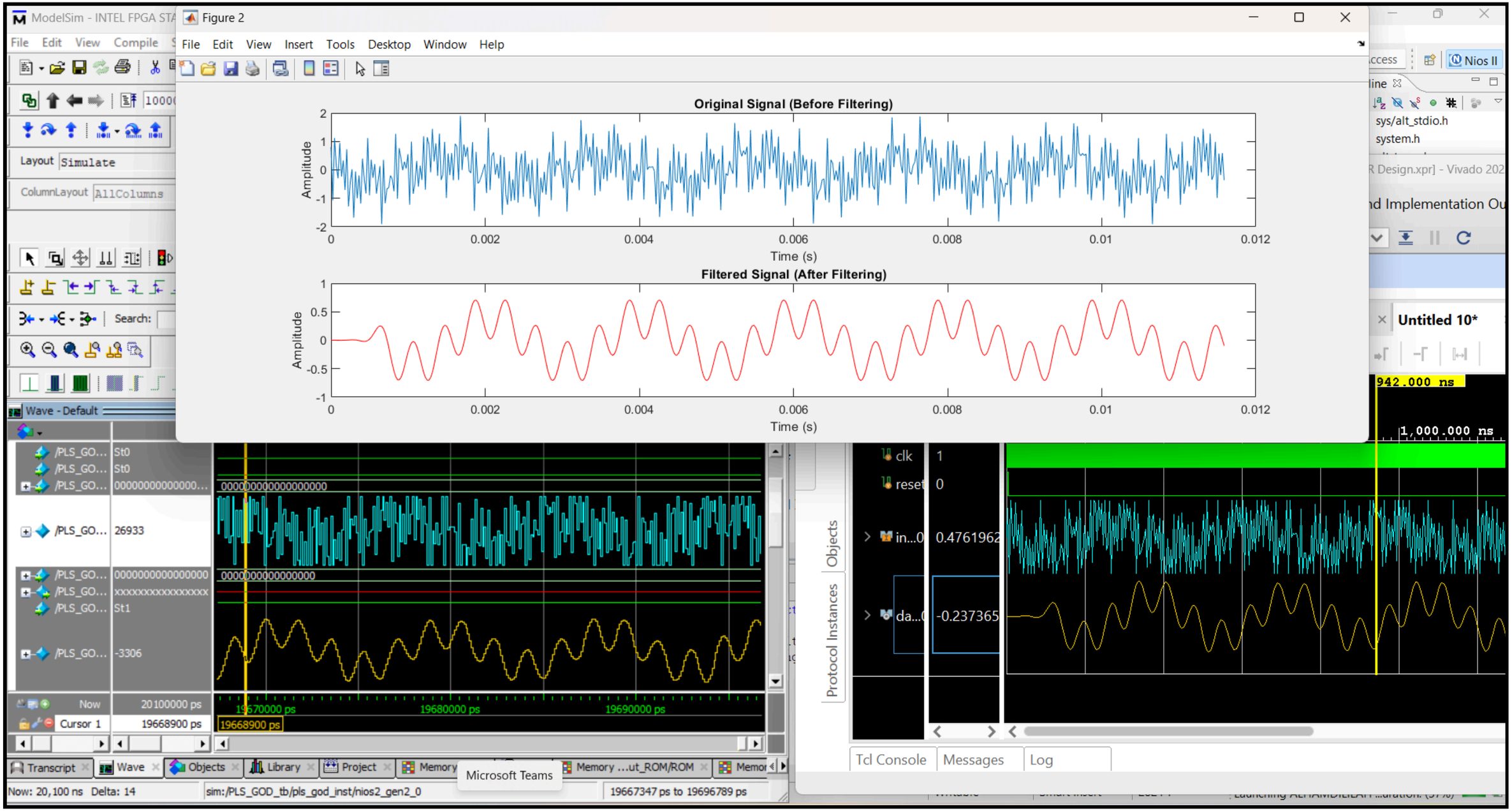


Final Test Case

Matlab Vs FIR Vs SOC

```
% Generate signals
signal = (sin(2*pi*1000*t)).*(0.8*cos(2*pi*1500*t));
noise = (0.5*sin(2*pi*30000*t))+(0.8*cos(2*pi*25000*t));
noisy_signal = noise + signal;
```




ModelSim - INTEL FPGA STA
Figure 2
File Edit View Insert Tools Desktop Window Help
Original Signal (Before Filtering)
Amplitude
Time (s)
Filtered Signal (After Filtering)
Amplitude
Time (s)
Wave - Default
Untitled 10*
942.000 ns
1,000.000 ns
clk
reset
in...0
da...0
0.4761962
-0.237365
Objects
Protocol Instances
Tcl Console
Messages
Log
Transcript
Wave
Objects
Library
Project
Memory
Memory ...ut_ROM/ROM
Microsoft Teams
Now: 20,100 ns Delta: 14
sim:/PLS_GOD_tb/pls_god_inst/nios2_gen2_0
19667347 ps to 19696789 ps
sys/alt_stdio.h
system.h

Refrences

- Eng. Nour and Eng.Yaseen FIR documentation
- Eng. Mohamed Tarek Repo
- Digital Filter Basics -Akash Murthy
- A guide to SDR and DSP using Python - Filters
- FIR design lecture from Zewil DSP course
- Pipelined VS Trasnposed FIR - all about circuits
- Analog Devices, Inc. Digital Filters paper >> ADI Paper
- Intel® Quartus® Prime Pro Edition User Guide: Platform Designer
- Nios II Classic Software Developer’s Handbook
- Avalon® Interface Specifications
- Introduction to Platform Designer - Intel Learning
- Platform Designer Standard Interfaces - Intel Learning
- The Nios® II Processor: Introduction to Developing Software - Intel Learning
- Custom Component Development Using Avalon® and Arm* AMBA* AXI Interfaces - Intel Learning
- Forcing Synthesis tool (Preserve for Debug)
- Preserve for Debug 2

Thank you!

Feel free to reach out to us if you have any questions.

Linkedin

https://www.linkedin.com/in/ygeng026/

Email Address

youssef.gamaleldein@gmail.com

Github

https://github.com/YoussefGamal-007