Project work on

DSP Filter Design and Hardware Implementation on FPGA

SUMMER INTERNSHIP PROJECT

at

G.S. Sanyal School of Telecommunication (GSST)

Indian Institute of Technology Kharagpur (IIT Kharagpur)

(Duration: 13–05–2025 to 25-06–2025)

submitted by

SOUMYADEEP CHAKRABORTY

National Institute of Technology Durgapur

Depatment : Electronics and Communication Engineering

Roll No : 22EC8056 Reg No : 22U10418

Under the supervision and guidance of

Dr. Amit Kumar Dutta (Associate Professor)

(GSSST - IIT Kharagpur)

DECLERATION

I, **SOUMYADEEP CHAKRABORTY, (Roll No: 22EC8056**) **Natioanal Institute of Technology, Durgapur** hereby declare that the work presented in this report entitled **“DSP Filter Design and Hardware Implementation on FPGA”** has been carried out by me as a part of my **Summer Internship Project** at the **G.S. Sanyal School of Telecommunication, Indian Institute of Technology (IIT) Kharagpur**.
This project is an original work conducted under the valuable guidance and supervision of the internship mentor, **Dr. Amit Kumar Dutta,** as an internship project at **IIT Kharagpur**. Throughout the duration of this project, I have made sincere efforts to uphold the values of academic integrity and propriety. All external sources of information, data, tools, or references used in the completion of this work have been appropriately cited at the end. I undertake full responsibility for the authenticity and originality of the content and findings documented in this report.

SOUMYADEEP CHAKRABORTY
ROLL NO: 22EC8056 REG NO: 22U10418
NIT DURGAPUR

CERTIFICATE

This is to certify that **Mr. Manas Mahato** has successfully completed a **Summer Internship** under my guidance and supervision on the topic –

“DSP Filter Design and Hardware Implementation on FPGA”

The internship commenced on **12th May 2025** and was completed on **25th June 2025.** The work contained in this project report was submitted to the **Indian Institute of Technology, Kharagpur**, towards the fulfilment of a Summer Internship Project in the **G.S. Sanyal School of Telecommunication (GSST)**, has been carried out solely by him under my supervision.

I further certify that he has shown dedication, sincerity, and a keen interest throughout the internship period and has completed the assigned work with commendable effort and enthusiasm.

DR. AMIT KUMAR DUTTA (Associate Professor)
G.S. Sanyal School of Telecommunication (GSST)
Indian Institute of Technology, Kharagpur

ACKNOWLEDGEMENT

At the outset, I would like to extend my heartfelt gratitude to **Dr. Amit Kumar Dutta** for granting me the invaluable opportunity to delve into the domains of **Digital Signal Processing, Embedded Systems, Computer Architecture**, and **VLSI** through this internship focused on **Digital Filter Design and Hardware Implementation**. His unwavering support, insightful feedback, and constant encouragement played a pivotal role in shaping the progress and success of this project.

I am also sincerely thankful to my mentors, **Mr. Kunal Thakur** and **Mr. Ashutosh Das**, for their expert guidance, patient mentorship, and continuous support throughout the course of my work. Their technical advice and constructive feedback were instrumental in helping me overcome challenges and refine my approach.

My deep appreciation also goes to the **G.S. Sanyal School of Telecommunication, IIT Kharagpur**, for hosting me and providing the necessary infrastructure and academic environment that facilitated a smooth and productive internship experience.

Lastly, I would like to acknowledge the valuable contributions of my fellow interns, **Manas Mahato** and **Subhagata Kundu**. Their collaborative spirit, constant support, and camaraderie made the internship both intellectually rewarding and personally enjoyable.

Index

Abstraction

This project aims to develop and implement a fully functional **Digital Signal Processing (DSP) system** capable of filtering **complex input signals** based on defined frequency characteristics. The heart of the system is a **high-pass filter**, specifically designed to allow frequencies **above 50 kHz** to pass while effectively suppressing lower frequencies. The design process starts with creating an **elliptical high-pass IIR filter** in MATLAB, based on the desired **cutoff frequency, ripple, and attenuation** specifications. To make the design suitable for hardware implementation, the impulse response of the IIR filter is truncated, resulting in an **FIR filter** that offers stability, linear phase, and hardware efficiency.
The next phase involves building a comprehensive DSP system in **Simulink**, incorporating the FIR filter along with **control logic, address generators, counters, and memory blocks,** simulating the behavior of a practical signal processing unit. The system architecture is designed to read input data from internal memory, process it through the filter, and write the output back to memory under precise control.
The complete Simulink design is then converted into synthesizable **HDL code** and implemented using **Xilinx Vivado**. Through Vivado's IP Integrator, the DSP module is interfaced with the **Zynq UltraScale+ MPSoC**, which handles data transfer, control signals, and synchronization via **AXI interfaces** and **BRAMs**. The processor writes input data to memory, initiates the filtering process, and retrieves the output data for further use or analysis. The hardware design is synthesized, a bitstream is generated, and the solution is deployed on the **FPGA board**. The final system demonstrates successful **real-time filtering of complex fixed-point data,** managed entirely by the processing system with efficient memory-based communication and reliable timing control.

Tools used:

- **MatLab R2024b**
- **Simulink**
- **Xilinx Vivado 2023.1**
- **Vitis IDE 2023.1**
- **FPGA ZCU106 (Hardware)**

Introduction

In many **signal processing** tasks, it becomes essential to suppress certain undesired frequency components while preserving the useful parts of a signal. This is precisely the role of **digital filters**. A **High-pass elliptical filter** is one such specialized filter that allows frequencies **beyond a fixed frequency,** defined by a **cutoff frequency,** to pass while attenuating all frequency components lowar than cutoff frequency. The elliptical filter, also known as the **Cauer filter**, is characterized by its **equiripple behavior** in both the **passband** and the **stopband**, providing a **highly efficient transition** from **pass to stop** region with the **lowest possible order** among standard filter types.
Digital filters are typically analyzed in the **frequency domain** using a **transfer function**, which relates input to output and helps visualize how the system modifies different frequencies. This transfer function can be expressed in terms of **poles and zeros**, where **zeros** correspond to frequencies of complete attenuation (**output zero**), and **poles** represent points where the **gain peaks** sharply. The location of these poles and zeros fundamentally shapes the **behaviour of the filter.** transfer function in pole zero form:

$$H(z)= \frac{(z-q_0)(z-q_1)(z-q_2)(z-q_3).....(z-q_i)}{(z-p_0)(z-p_1)(z-p_2)(z-p_3).....(z-p_j)}$$

where, $p_j = j^{th}$ pole
$q_i = i^{th}$ zero

Another representation of filter transfer function is as

$$H(z)= \frac{b_0+b_1z^{-1}+b_2z^{-2}+b_3z^{-3}+.....}{a_0+a_1z^{-1}+a_2z^{-2}+a_3z^{-3}+.....}$$

where, a = Denominator Coefficient
b = Numeretor Coefficient

Two major classes of digital filters are **IIR (Infinite Impulse Response)** and **FIR (Finite Impulse Response)**. An IIR filter leverages both present and past inputs and outputs, offering sharp frequency selectivity with fewer coefficients. However, due to the presence of poles in the denominator, they carry a risk of instability unless carefully designed. The general form of an IIR filter's transfer function is:

$$H(z)= \frac{b_0+b_1z^{-1}+b_2z^{-2}+b_3z^{-3}+.....}{1+a_1z^{-1}+a_2z^{-2}+a_3z^{-3}+.....}$$

In contrast, **FIR filters** are inherently stable as they depend only on current and previous input values, with no feedback from the output. Their transfer function includes **only zeros**, making the denominator equal to 1.

Filter Design Flow

- **Filter specifications:**
  - **Type**- High Pass
  - **Order** - 2nd
  - **Design** - Elliptical
  - **Cutoff frequency** - 50 KHz
  - **Sampling Frequency** - 200 KHz
  - **Passband Ripple** - 3 dB
  - **Stopband Attenuation** - 40 dB

MATLAB Design

- **Description**

To begin the filter design, a **high-pass elliptical IIR filter** was created in **MATLAB** using predefined specifications such as **sampling frequency, cutoff frequency, passband ripple, stopband attenuation, and filter order**. These parameters defined the desired frequency response of the **high-pass filter.**
Once the **IIR filter** was designed, we applied a unit impulse signal to determine its **impulse response**, which is theoretically **infinite in length**. For practical simulation, the impulse was limited to **100** samples, yielding a finite-length output. As **FIR filters** are preferred in hardware due to their **stability and ease of implementation**, we **truncated** the impulse response to **12 samples**, effectively creating an FIR filter that approximates the original IIR behavior.

$$H_{FIR}(z)=0.2192-0.4744z^{-1}+0.234z^{-2}+0.1598z^{-3}-0.1454z^{-4}-0.0377z^{-5}+0.0755z^{-6}-0.0011z^{-7}-0.0342z^{-8}+0.0088z^{-9}+0.0135z^{-10}-0.0073z^{-11}$$

To evaluate this approximation, we compared the frequency responses of the IIR and FIR filters, particularly around the **cutoff region**. As expected, the FIR filter exhibited some **loss of sharpness** in the transition band due to truncation. The cutoff frequency was estimated using the **standard -3 dB point**, allowing us to assess how well the FIR retained the intended high-pass characteristics.
The FIR filter was then implemented using **complex fixed-point arithmetic** input signals to simulate **real-world hardware conditions**. This validated the filter's performance and accuracy under resource-constrained environments like FPGAs.
Through this process, we successfully converted a **theoretical IIR design** into a **practical FIR implementation**, preserving essential **frequency-domain features** while ensuring compatibility with digital hardware systems.

- **Cutoff frequency comparison through Command Window**

```
Cutoff frequency IIR = 45190.43 Hz
Cutoff frequency FIR = 45166.02 Hz
```

MATLAB Code

- **Code**
- **Explanation**

```
% Design filter (example)
fs = 200e3;           % Sampling frequency
fc = 50e3;            % Cutoff frequency
N = 2;                % Filter order
Rp = 3;               % Passband ripple
Rs = 40;              % Stopband attenuation

[b, a] = ellip(N, Rp, Rs, fc/(fs/2), 'high'); % Elliptical filter

% Plot full freq response using freqz
[H_full, w_full] = freqz(b, a, 4096, fs);  % Full response

% Impulse response h(n) from filter
imp = [1 zeros(1,99)];     % 100-sample impulse
h = filter(b, a, imp);     % Get 20-point impulse response

n=12;            % number of sample points in FIR filter
% Truncate to n samples to create h1(n)
h1 = h(1:n);     % Keep only first n samples

% Compute magnitude response from h(n)
[H_20, w_20] = freqz(h1, 1, 4096, fs);  % h(n) treated as FIR (denominator = 1)
```

Filter specification

Filter generation

Transfer function of IIR filter

Impulse response generation of IIR filter

Generating impulse response of FIR filter by Truncating impulse response of IIR filter

Transfer function of FIR filter

```
n = 10;                                    % Use last 10 points for averaging

% IIR filter
flat_level = mean(H_db(length(H_db) - n + 1 : end));
cutoff_level = flat_level - 3;
idx = find(H_db >= cutoff_level, 1, 'first');     % First index where drop occurs
cutoff_freq = w_full(idx);                        % Corresponding frequency in Hz
fprintf('Cutoff frequency = %.2f Hz\n', cutoff_freq);

% FIR filter
flat_level_t = mean(Ht_db(length(Ht_db) - n + 1 : end));
cutoff_level_t = flat_level_t - 3;
idxt = find(Ht_db >= cutoff_level_t, 1, 'first'); % First index where drop occurs
cutoff_freq_t = w_20(idxt);                       % Corresponding frequency in Hz
fprintf('Cutoff frequency t = %.2f Hz\n', cutoff_freq_t);
```

Calculating cutoff frequency of both IIR and FIR filter

```
%Fixed-point configuration
T = numerictype(1, 8, 6);  % Signed, 8-bit word length, 6-bit fraction
F = fimath('RoundingMethod', 'Floor','OverflowAction', 'Saturate');
N = 100;
real_vals = fi(randn(N, 1), T, F);
imag_vals = fi(randn(N, 1), T, F);

% Combine into complex fixed-point
complex_fxp = complex(real_vals, imag_vals);  % Still fi type

% Create time vector
Ts = 1;  % sampling time (or 0.005, etc.)
t = (0:N-1)' * Ts;

% STEP 5: Create time series
in_ts = timeseries(complex_fxp,t);

x = in_ts.Data;     % input signal values
t = in_ts.Time;     % time values
```

Generating time series complex fixed point input

```
a=1;
h1_fixed = fi(h1, T, F);  % Numerator coefficients
a_fixed = fi(a, T, F);  % Numerator coefficients

y = filter(h1_fixed, a_fixed, x);   % Apply FIR filter to your input
```

Applying the input to FIR filter

- **Illustrations:**

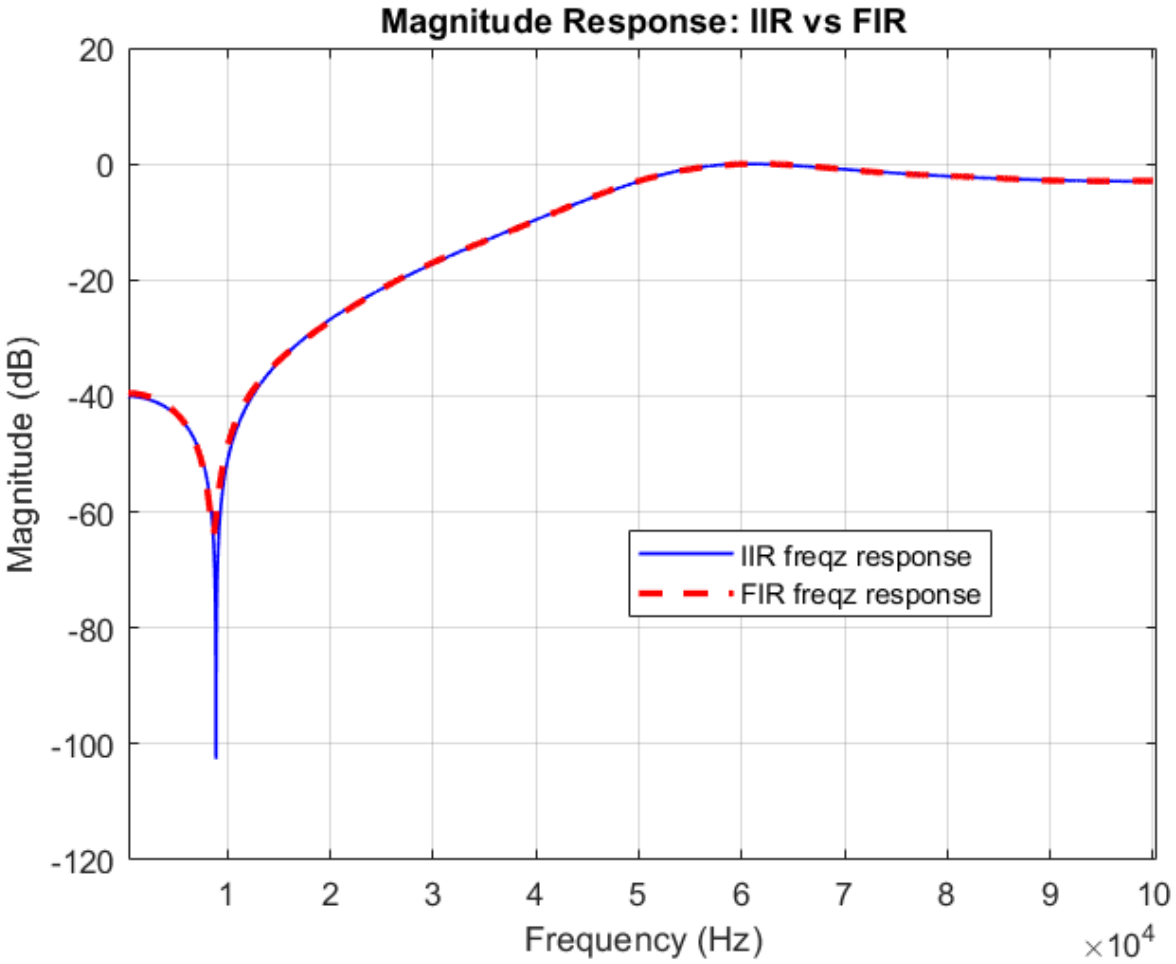


Figure 1: Transfer function comparison of IIR and FIR filter

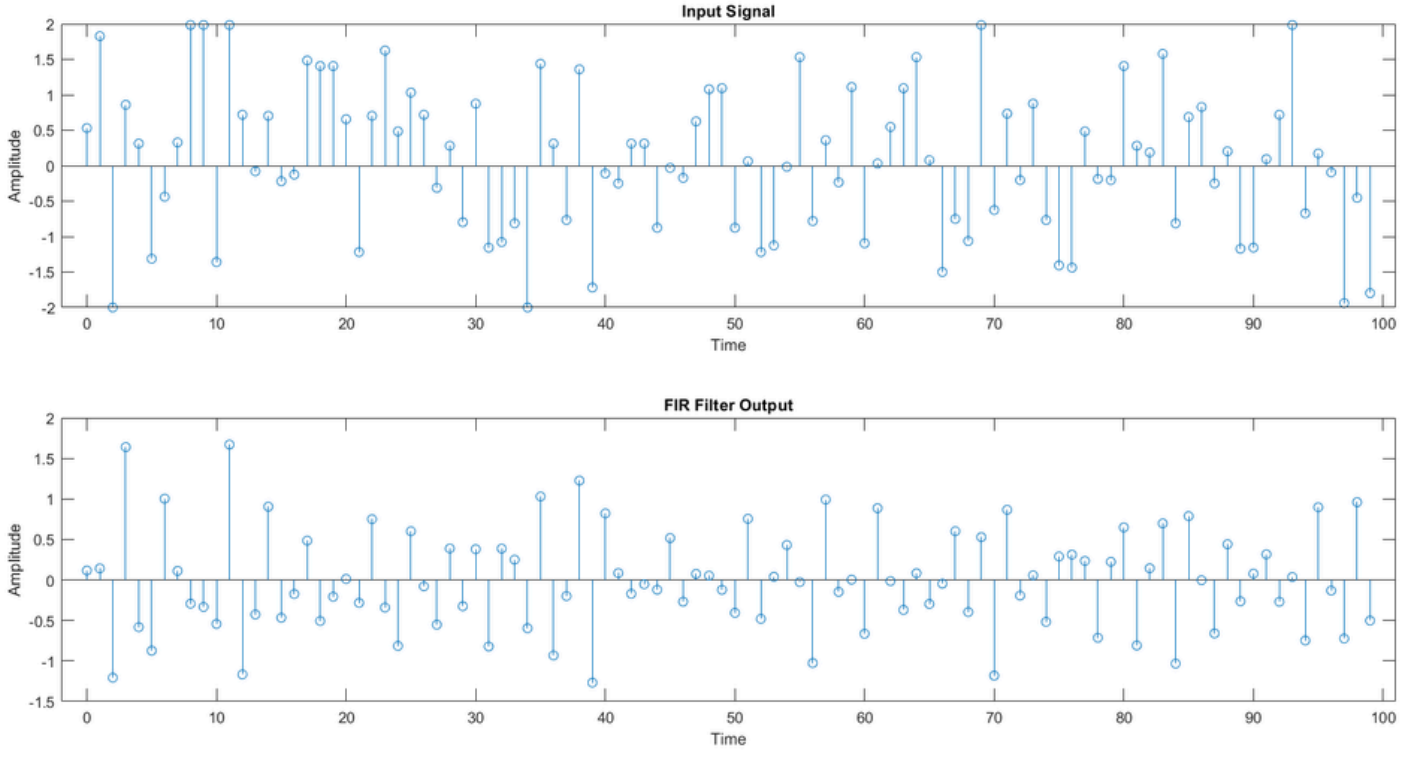


Figure 2: Input and output of the FIR filter

Simulink Design

- **FIR Filter Core: Multiply-Accumulate DSP Network (design diagram 1)**

At the core of the **signal processing system** lies the **FIR filtering network**, designed using a **direct-form FIR architecture** in Simulink. This module performs the core **convolution** operation between the **input signal** and **filter coefficients**, governed by the classical FIR equation:

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

where **y[n]** is the **output**, **h[k]** are the **filter coefficients**, **x[n−k]** are the **delayed inputs**, and **N=12** is the **filter order**. In the Simulink implementation, unit **delay blocks (Z^{-1})** create delayed versions of the input signal, which are then **multiplied** by corresponding **coefficients** using parallel multiplier blocks. These products are **summed** using a **multi-stage adder tree,** generating one filtered output per clock cycle.

Instead of a s**ingle 12-input adder**, we used a **hierarchical adder tree** to **minimise combinational delay** and logic depth, which significantly **improves timing performance** and **reduces latency**. This tree structure also optimizes the usage of **hardware resources** like **DSP slices** and **LUTs**, and is more **scalable** for higher-order filters. It aligns well with **pipelined processing**, making it ideal for **high-speed** FPGA implementation.

The filter coefficients **h[k]** were derived by **truncating** the impulse response of a high-pass elliptical IIR filter designed in MATLAB. This provided a **sharp transition** in frequency while keeping the F**IR length manageable** for hardware. Though some deviation from the IIR was expected, the resulting FIR closely approximated the original spectral behavior.

To further improve timing, we applied the **Cutset Theorem**, introducing **pipeline registers** inside the **adder tree**. These registers **break up long combinational paths** and reduce the critical path delay, enabling high-frequency operation on platforms like the **Zynq UltraScale+ ZCU106**.

A major feature of the design is its ability to process complex fixed-point data, with coefficients represented as:

$$h[k] = h_{Re}[k] + j \cdot h_{Im}[k]$$

Both real and imaginary parts are defined in Simulink Constant blocks, using **signed 8-bit fixed-point format with 6 fractional bits: fi(1,8,6)**. This provides a good trade-off between precision and hardware efficiency.

The design is fully **parallel and streaming-based,** meaning one complex input sample is processed per clock after the pipeline is filled. Overall, this module achieves **high throughput, low latency, and excellent resource optimization,** making it a robust and efficient DSP core for **real-time** FIR filtering on FPGA.

- **Design diagram 1**

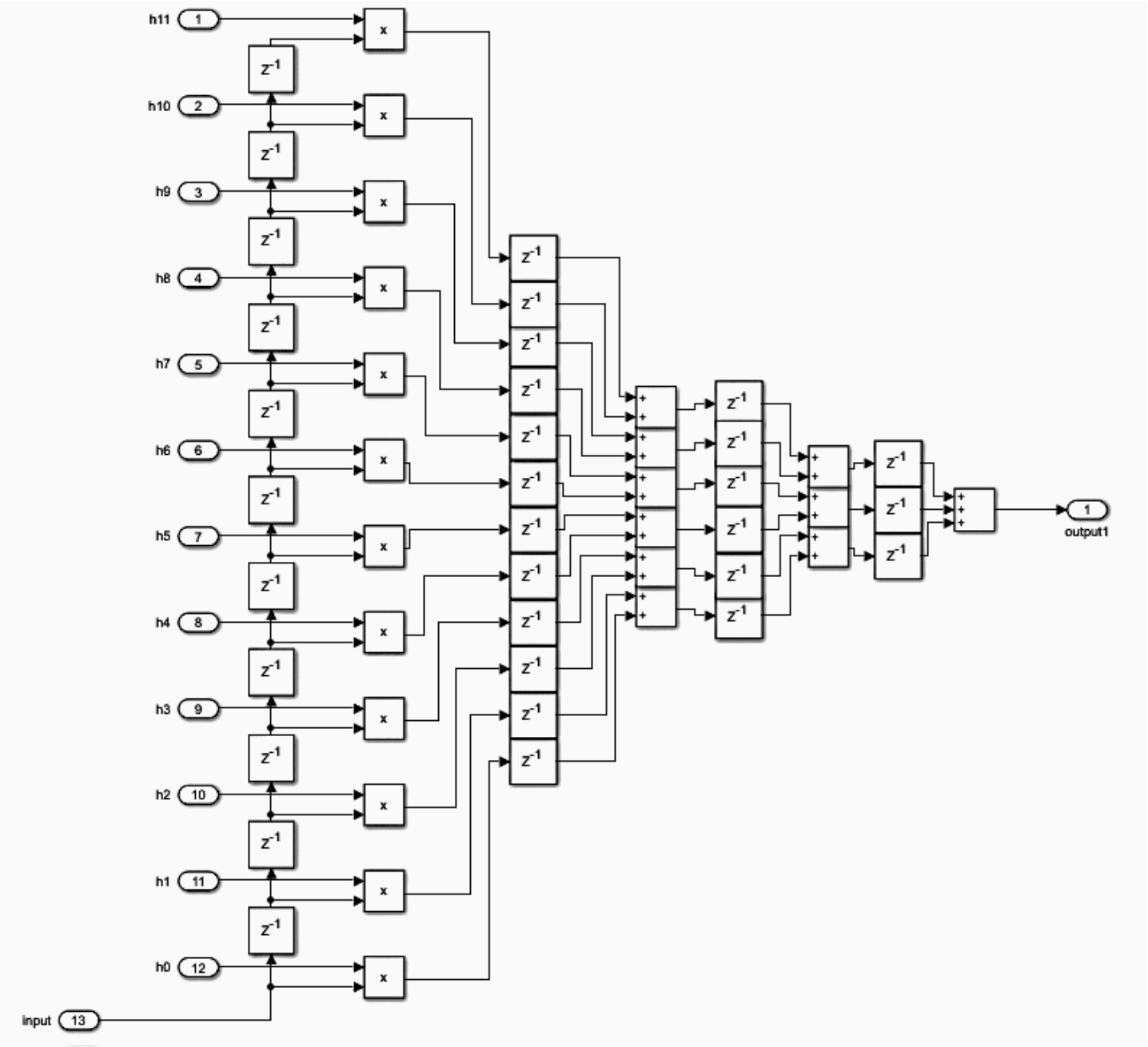


Diagram 1: DSP network

- **FIR Coefficient Loading and Complex Number Support (design diagram 2)**

The coefficient interface layer, represented in the third level of the hierarchy, functions as the **parameter injection point** for the FIR system, where **complex-valued coefficients** are provided to the filtering core. In this block, the set of **FIR coefficients** is statically assigned using **Simulink Constant blocks**, each one corresponding to a filter tap. These real constants are converted to complex by **real-img to complex block** of Simulink.
From a structural standpoint, this block acts as a **wrapper around the FIR MAC network,** feeding the coefficients into the DSP computation path described previously. The input signal, labeled **i1**, is a complex fixed-point stream. The result is output at the **output1** port, which continues the complex fixed-point format to retain the structure and integrity of the input data throughout the entire system.

What makes this module particularly important is its role in **decoupling the signal processing core from coefficient generation and configuration**. By isolating coefficient storage and format adaptation into a dedicated layer, the system becomes significantly more modular and easier to scale or reconfigure for different filter types.

- **Design diagram 2**

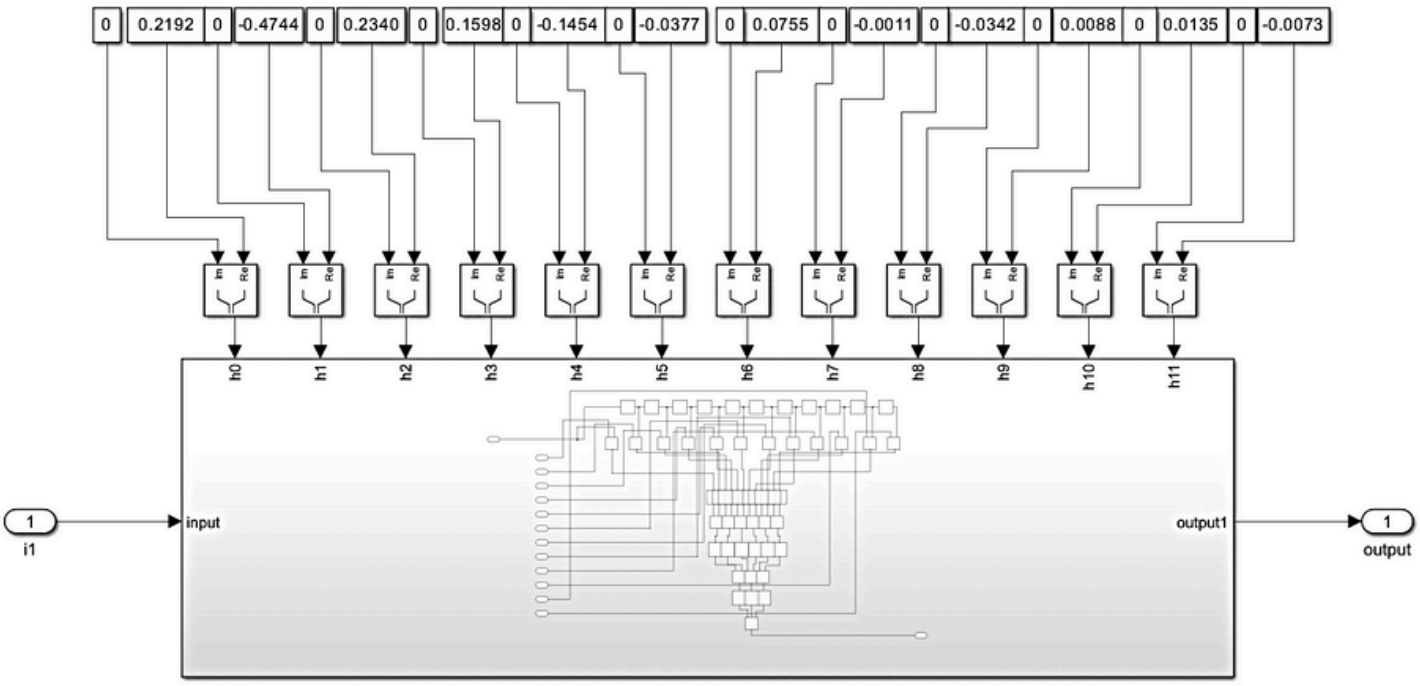


Diagram 2: Filter coefficient feeding network

- **FIR system control and memory logic (design diagram 3)**

The module represented in the first hierarchical level of the Simulink design serves as the **coordination and control infrastructure** for the entire FIR filtering system. It governs how input data is supplied to the filter core, how filtered output is stored, and how various **control, addressing, and handshaking signals** synchronize the system. At this level, the focus transitions from signal-level arithmetic to data management, memory interfacing, and sequential control, ensuring that the core filtering module operates efficiently in an automated and hardware-compatible way.

On the **input side**, raw signal samples are first written into a dedicated input **RAM block (addr_i_int)**, which temporarily buffers the incoming data. The addressing and write-enable signals used for loading this memory are dynamically selected via **multiplexers (mux_addr, mux_we)**. These multiplexers allow the system to switch between **testbench-controlled input** and **internal FSM-controlled input**, based on the value of the global control signal **en**. When **en = 0**, the system is in loading mode, and the address (**addr_i**) and write-enable (**we1**) signals are taken from **external sources**, i.e., the testbench. This enables manual or scripted data injection during simulation or processor-driven operation. When **en = 1**, the FIR system enters active mode, and the address and control signals are generated internally by the **input FSM (fsm_i)** and its associated **counter block**, ensuring automatic feeding of preloaded data into the FIR core.

Once the input data has been consumed, the FIR core processes each sample and

produces filtered output, which is directed toward the output RAM subsystem (addr_o_int). This side of the system mirrors the structure of the input logic. The writing of filtered outputs into memory is controlled by a second **output FSM (fsm_o)**, which manages a counter and issues the appropriate address and write-enable signals. Here again, **multiplexers (mux_addr2, mux_we2)** are used to toggle between internal and external signal sources. However, in this case, the control is governed by a signal named **open**. When **open = 1**, the RAM addressing and writing are handled internally by the output FSM, allowing the FIR system to autonomously store computed values. When **open = 0**, the address and enable signals are taken from the **external testbench**, enabling simulation or hardware logic to read the stored filtered data for verification or downstream processing.
This **dual-mode signal routing**—controlled by en on the input side and open on the output side—adds a layer of **flexibility and controllability** that is crucial for both testing and real-time deployment. It enables the FIR system to operate in **two distinct phases**: first as a **memory-mapped, externally controlled peripheral**, and then as a **self-contained, autonomous DSP engine**. This approach not only supports detailed simulation and debugging but also mirrors real-world embedded processing scenarios, where a processor might preload data into memory and then release control to a dedicated accelerator core.
The rest of the control logic, including **ready signal generation, address counters, and parameterized FSMs**, ensures that the entire filtering operation is conducted in a fully **deterministic and synchronous** fashion. The ready flag is asserted only after the full sequence of outputs has been successfully written, signaling the end of computation. This clean separation between data loading, processing, and unloading allows for **modular design reuse, robust timing closure**, and **easy integration with larger FPGA-based dataflow systems.**
Altogether, this module provides the **intelligent orchestration layer** that surrounds the FIR core. By managing memory access, state transitions, and input/output routing using **configurable FSMs, counters, and mux logic**, it ensures that the filtering operation proceeds seamlessly—whether controlled externally for simulation or internally for real-time FPGA deployment. Its design allows the FIR system to behave predictably under different modes of operation, maintaining both performance and flexibility across various use cases.

- **Input FSM control (fsm_i)**

The **input-side controller**, **fsm_i**, is a finite state machine that manages the logic for writing input data into the RAM . This FSM begins operation in the **no-operation** state, where the write enable signal we is set to true, and the address index counter i1 is initialized to 0. This state remains active as long as the global **en signal remains 0**, meaning the system is waiting for input data from the testbench. Once the en signal becomes 1, indicating that all the input data has been written and the system is ready to begin processing, the FSM transitions to the read state. In this state, **we** is set to **false**, effectively disabling writing from the testbench side and enabling **reading** from the **RAM**, and a counter counts **until all the input data** has been read. Once the full length of data has been read into the system, the FSM enters another **no-operation state**, where **we** is set to **true** again, signaling readiness to receive the next set of data if needed. This design ensures a clean separation between external write operations and internal data processing by enabling write protection as soon as computation starts, preventing unintentional data corruption. The controlled toggling of we and gated transition using the en flag makes this FSM **highly synchronized** and **resource-efficient** for hardware implementation.

- **Output FSM control (fsm_o)**

The **output-side controller**, **fsm_o**, is another finite state machine tasked with writing the filtered **output data** back into the **RAM** after the FIR processing is complete. Initially, the FSM stays in the **no-operation state**, where the signal **we2** is set to **false**, indicating **no writing** activity. It stays in the no-operation till the **output** of the FIR system is **ready**. This acts as a **delay buffer**, allowing **internal pipeline stabilization** before writing starts. There is an additional **3 clock cycles delay** between the input and output of the FIR system. After this, the FSM transitions to the **write state**. In this state, **we2** becomes **true**, allowing the filtered data to be written into the output RAM. This state remains active till all the **output data** is fed to the **output RAM**. This process takes **longer** than the length of the input, because the **length** of the **output** is **greater** than that of the **input** (in our case, the input length is **100** and the number of filter coefficients(**h**) are **12**, thus the length of the output is 100+12-1=**111**). When all the output has been stored in the RAM, the FSM transitions to the **no-operation state**, where **ready** is asserted **true**, signaling to the testbench that all the output data has been written and is ready for **external retrieval**. This design ensures **precise output timing control**, accounting for filter delay and data burst length, and uses clearly defined states for initialization, processing, and completion.

- **Design diagram 3**

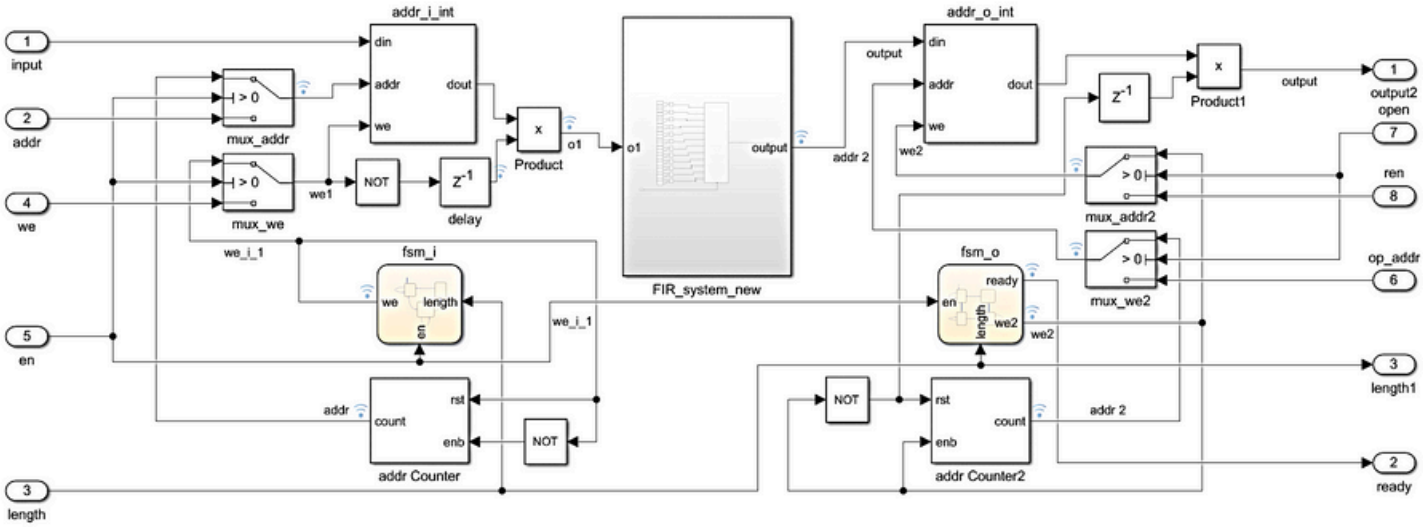


Diagram 3: FIR system control and memory logic network

- **Testbench controller and interface module (design diagram 4)**

At the highest abstraction level, the topmost module functions as a **testbench-controlled input-output interface** for the FIR filtering system. It manages the external feeding of input data, initiation of processing, and the retrieval of output data in a highly controlled and synchronous manner. The core of this module revolves around two primary **address counters** (**addr Counter** and **op addr Counter**) and two **FSMs** (**enable** and **enable2**) that handle the lifecycle of input and output operations, respectively.
On the **input** side, the **addr Counter** generates sequential addresses to write incoming data into the FIR input RAM. The counter is enabled when the signal **en** is **low**, indicating that the system is in the **input loading phase**. The input process is initiated by the assertion of the global control signal **enable_1**, which **activates** the **enable** FSM. This FSM pulls en low, enabling external write access.

With **we = 1** and the counter's **enb** line **high**, **addresses** are incremented, and input samples are written into the RAM. Once the defined number of inputs (e.g., **100**) are written, the FSM sets **en high**, handing over **control** to the **internal logic** of the FIR system and disabling external write access. A constant block sets the data **length**, which is used by both input and output controllers to time the number of samples processed.
The **enable** FSM thus ensures a **safe** and **conflict-free transition** from testbench-controlled data loading to internal computation. Its only task is to control this phase cleanly: **enabling the RAM** and **counter** during data loading and relinquishing control to internal logic for processing.
The **main FIR module** is instantiated at the center of the design. It receives the externally generated **input signals** (**input, addr, we, length, en, ren, open, op_addr**) and produces **outputs** (**output, output2, ready, length1**) that control the post-processing flow. Once the input is complete and **en** is **high**, the FIR core begins its **internal operation**, processing the data and storing the filtered results in the **output RAM**.
When computation concludes, the FIR system asserts the **ready** signal **high**, which is sensed by the **enable2** FSM. In response, the FSM sets **open** to **0**, granting the **testbench access** to the output RAM. It also enables the **op addr Counter** by setting its **enb** signal **high**. This initiates the **readout phase**, where addresses are generated for output retrieval. The **output data** is now sequentially fetched by the testbench using the generated addresses. Once the full output is retrieved, the FSM can optionally **reset** the signals, preparing the system for the next input cycle.
A particularly important aspect of this top-level module is the **multiplexing logic** for **address** and **write-enable** control. Since both internal and external agents may access the RAMs, 2-to-1 multiplexers are used on both input and output paths for addr and we signals. On the **input side**, the MUX selects between testbench and internal sources based on the value of **en**. When **en = 0**, **testbench signals** are routed; when **en = 1**, **internal signals** are used. Similarly, on the **output side**, the **open** signal governs the MUXes. When **open = 0**, the output **address** and **read-enable** come from the **testbench**; when **open = 1**, they switch to the **internal FSM control**. This dynamic arbitration ensures that there is **no conflict during memory access**, maintains proper sequencing, and allows for seamless transition between simulation and real-time processing modes.
In summary, this topmost module provides a **deterministic, synchronized, and testbench-compatible shell for the FIR system**. Through FSM-based gating, address sequencing, and precise control signal transitions, it ensures **safe handoff** between input loading, processing, and output delivery, making the design highly suitable for FPGA-based DSP applications and hardware-in-the-loop simulation.

- **<u>Key points of design:</u>**

- For simulation purposes, we connect the input port to the **from workspace** block to get our required input from **MATLAB workspace**. This has been removed from the final design.
- All the **control signals** (e.g.- en,we etc.) should be of type **boolean**. All the **inputs, output and filter coefficients** are should be defined as the type **complex fixed point(1,8,6)**. Other **parameters** (e.g.-address, length) should be defined as the type **uint8**.
- **Sampling time** should be set to **1** ,not -1(inherit).
- Our main FIR block takes **3 clock cycles** to produce the output.
- As our **input** is of length is **100** and number of **coefficients** are **12** the length of the output is **100+12-1 =111**

- **Design diagram 4**

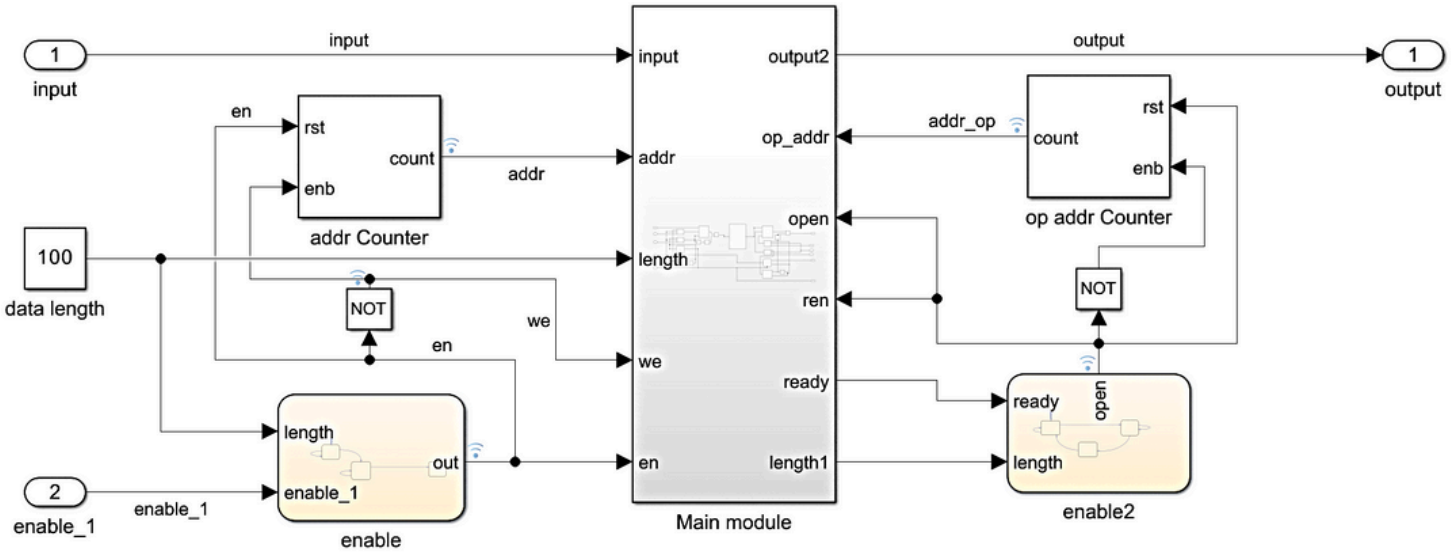
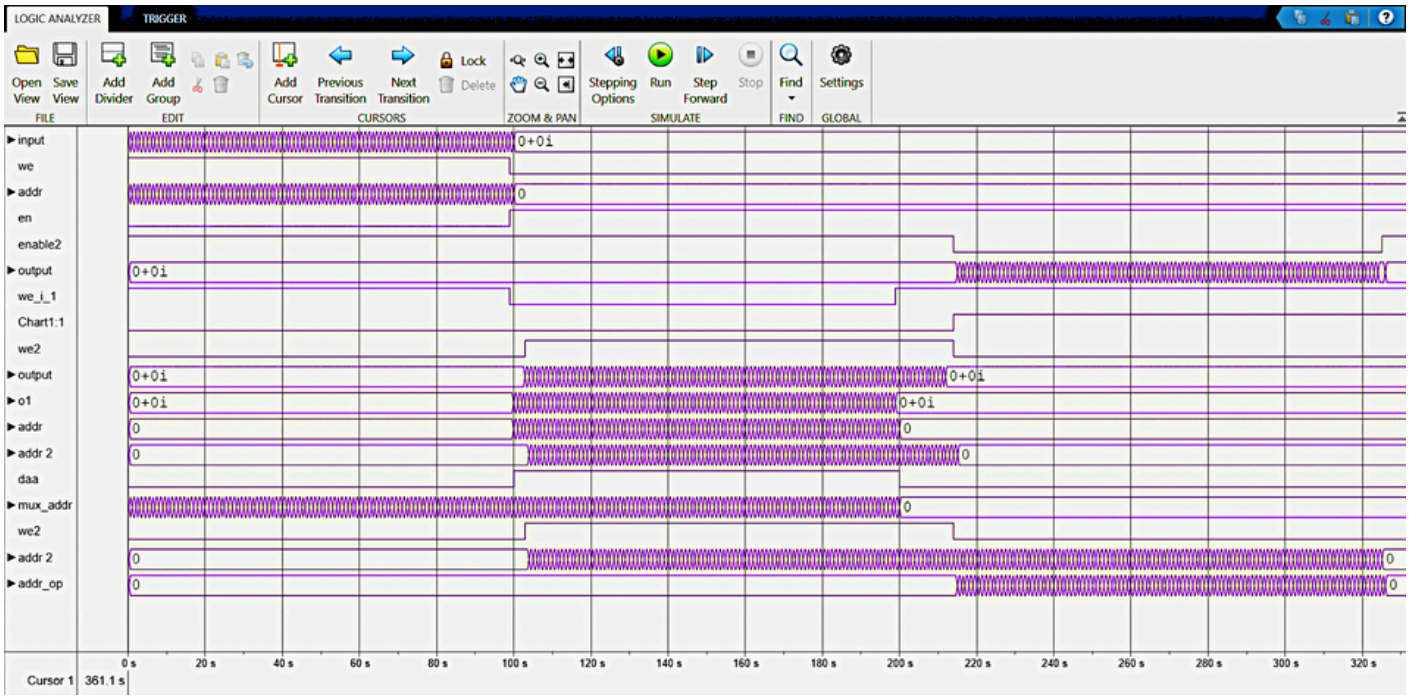


Diagram 4: Testbench controller and interface network

- **Simulation result**



- **Key points of simulation result:**

- From the simulation result, we can verify that during the input feeding cycle from the testbench en=0 and we=1, the address is generated perfectly. After that, en=1.
- During ithe nput data reading cycle, we=1 and the address is generated perfectly.
- The timing estimation between input and output is matched perfectly.
- The output data is stored properly by making we2=1 and giving the address to addr2.
- After the output data is stored in the RAM the ready=1.
- The open=0 indicated tesbench is retrieving the output data correctly.
- the total operation takes almost 350 clock cycles to complete.

• Importing the design to Vivado

After successfully designing, simulating, and validating the complete digital filter system in Simulink including the FIR filter, memory interface, FSM controllers, and control logic—the next step involves converting the Simulink model into synthesizable HDL code. This is done using the **HDL Workflow Advisor**, a built-in tool in MATLAB and Simulink that automates the process of generating VHDL or Verilog code. The tool also supports HDL test bench generation, IP core packaging, and direct integration with FPGA development tools like **Vivado**. This transition marks the beginning of implementing the system on real hardware, enabling high-speed and real-time signal processing directly on an FPGA platform.
OUR SELECTED FPGA BOARD**: AMD Zynq™ UltraScale+™ MPSoC ZCU106**

To create Vivado project from the Simulink design we need to follow these steps:

- The **HDL Workspace advisor** comes under the **HDL Code** tools section.
- Under the HDL code generation **settings,** we need to choose the subsystem and **HDL language**; in our case, it is **Verilog**. We need to select the target device as **ASIC/FPGA**.
- Also, there will be some changes in **parameter configuration** such as- - Solver type - Fixed step , discrete; Reset type Synchronous; default parameter behaviour - inlined etc.
- Within **Set Target** comes the option where the specifications of the FPGA are filled accordingly to that our Code is generated -
  Target workflow: **Generic ASIC/FPGA**, Synthesis tool: **Xilinx Vivado**, Family: **Zynq UltraScale +**,
  Device: **XCZU7EV- FFVC1156-2-e**.
- The Set Target Frequency set at: **500 MHz** Then RUN the complete Set Target.
- Then within P**repare Model for HDL Code Generation check Model settings** → **HDL code advisor** → **Run selected checks** → solve all warnings and errors
- Then inside HDL workflow advisor **HDL code generation → Set HDL options → Generate RTL Code and Testbench → select Generate RTL code → Run all**.
- After Generation of RTL Code, Synthesis and Implementation can be performed within **Simulink** or can be done in **Vivado**. To check all the **timing constraints** we performed that in HDL workflow advisor.
  FPGA Synthesis and Analysis → Perform Synthesis and P/R → Run implementation → uncheck skip this task → Run all .
- After the task finishes, we need to check the **timing slack** under implementation result, which should be **positive.**

This process will create a **vivado project file (.xpr)** inside a folder **vivado_prj** and all the generated **HDL code files (.v files)** will be inside the folder **hdlsrc**.

• Key points during HDL workflow:

- The code would be generated for **subsystem**, not for the whole system. So it is better to make the final design inside a subsystem.
- To meet the timing constarins (**timing slack positive**), we may need to change the design by adding **Flip flop (z^{-1} delay block)** in between **large combinational path** to **reduce the combinational path delay**.
- We have to make sure all the **warnings and errors** are solved during this process, as these can cause **timing violations** and other problems in later stages.

- **HDL workflow window:**

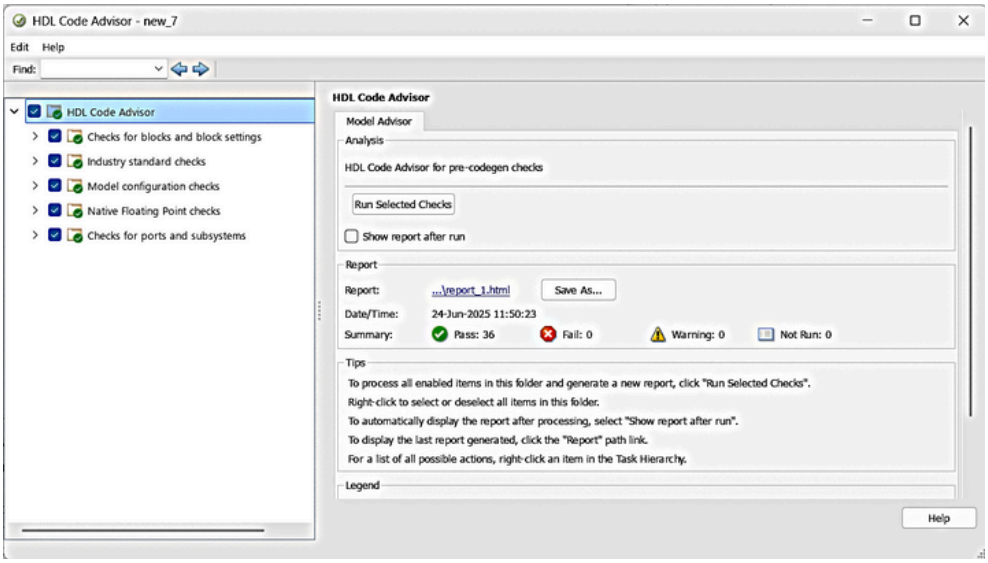


Figure 1:
HDL code advisor window (ran without any error or warning)

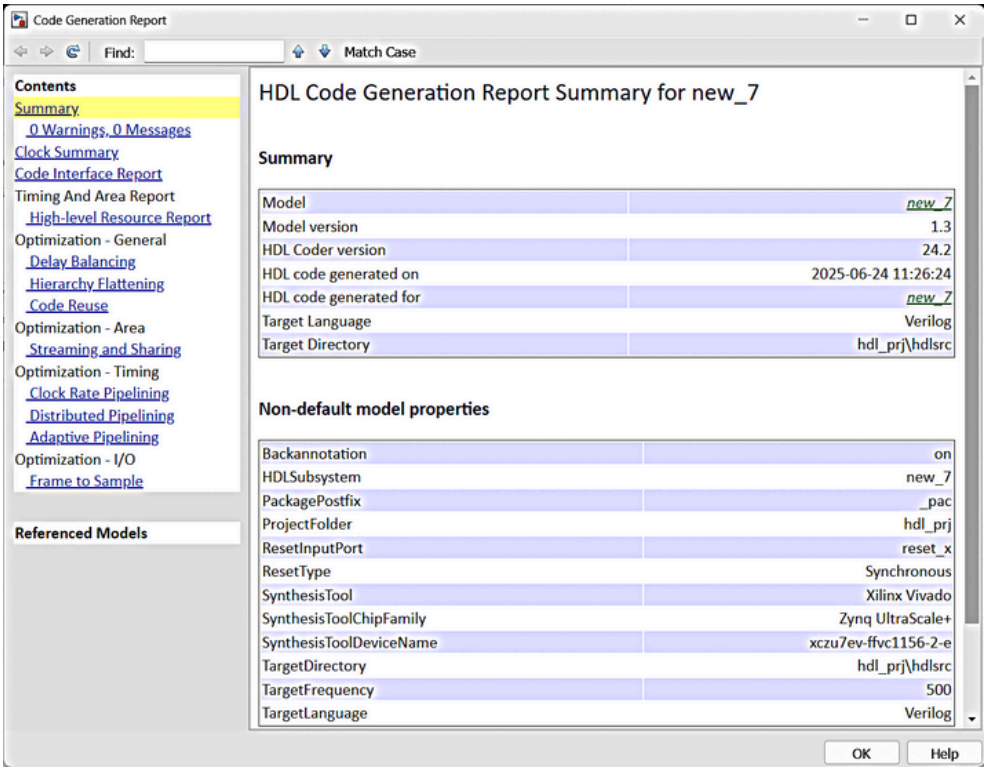


Figure 2:
HDL code generation report window (after successful HDL code generation)

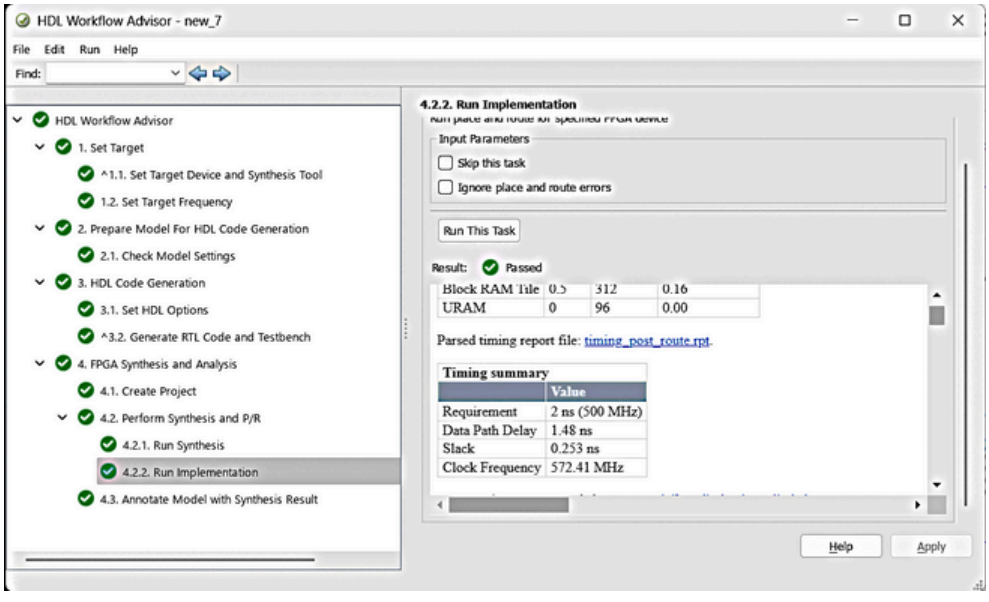


Figure 3:
HDL workflow advisor window (ran implementation with positive time slack)

Xilinx Vivado Design

• Introduction

When transitioning from the Simulink-based FIR filter design to hardware implementation using Xilinx Vivado on the FPGA, **directly mapping the entire system** to the board posed significant limitations. Each complex fixed-point input sample consists of **15-bit real and 15-bit imaginary components**, along with numerous **control signals** for **address generation, enable toggling, data validity, and output synchronization**. The synthesized RTL schematic revealed a requirement of approximately **90 I/O pins**, far **exceeding** the available I/O **resources** of the ZCU106 evaluation board.

Routing such a large number of signals through **physical switches** or connectors is not only **impractical** but also **unreliable**. **Synchronizing** real-time data and control transitions using **manual toggling** becomes **unmanageable**, especially at high clock speeds. Moreover, signal integrity and precise timing, which are critical in fixed-point DSP systems, cannot be ensured through manual inputs.

To address these hardware limitations, we adopted a **block design approach** within **Vivado's IP Integrator**. The **FIR system** was packaged as a **custom IP block** and integrated into a block diagram environment. Input data was fed using **two BRAMs**—one for the **real part** and one for the **imaginary**—while the filtered **output** was stored in **two additional BRAMs,** similarly divided. These BRAMs served as **on-chip memory buffers**, eliminating the need for e**xternal I/O** and ensuring **high-speed access**.

All control signals and synchronization were managed by the **Zynq UltraScale+ MPSoC Processing System**, which communicated with the programmable logic using **AXI interconnects**. The **PS core** could **write inputs to BRAM**, trigger computation via control registers, and **read back the output data** for further use or validation. This architecture provides a reliable, scalable, and **high-performance** interface between the processor and the custom logic, enabling full system verification and deployment without the impracticalities of pin-level interfacing.

• Block design description and system integration

At the heart of this block design is the **Zynq UltraSCALE+ MPSoc Processing System (PS)** block, which functions as the **system controller**. It handles **clock generation, input generation** and all **input-output interfacing**. Through **memory-mapped AXI** transactions, the Zynq PS writes the input samples into memory blocks and later reads the FIR output, providing a seamless bridge between software and hardware.

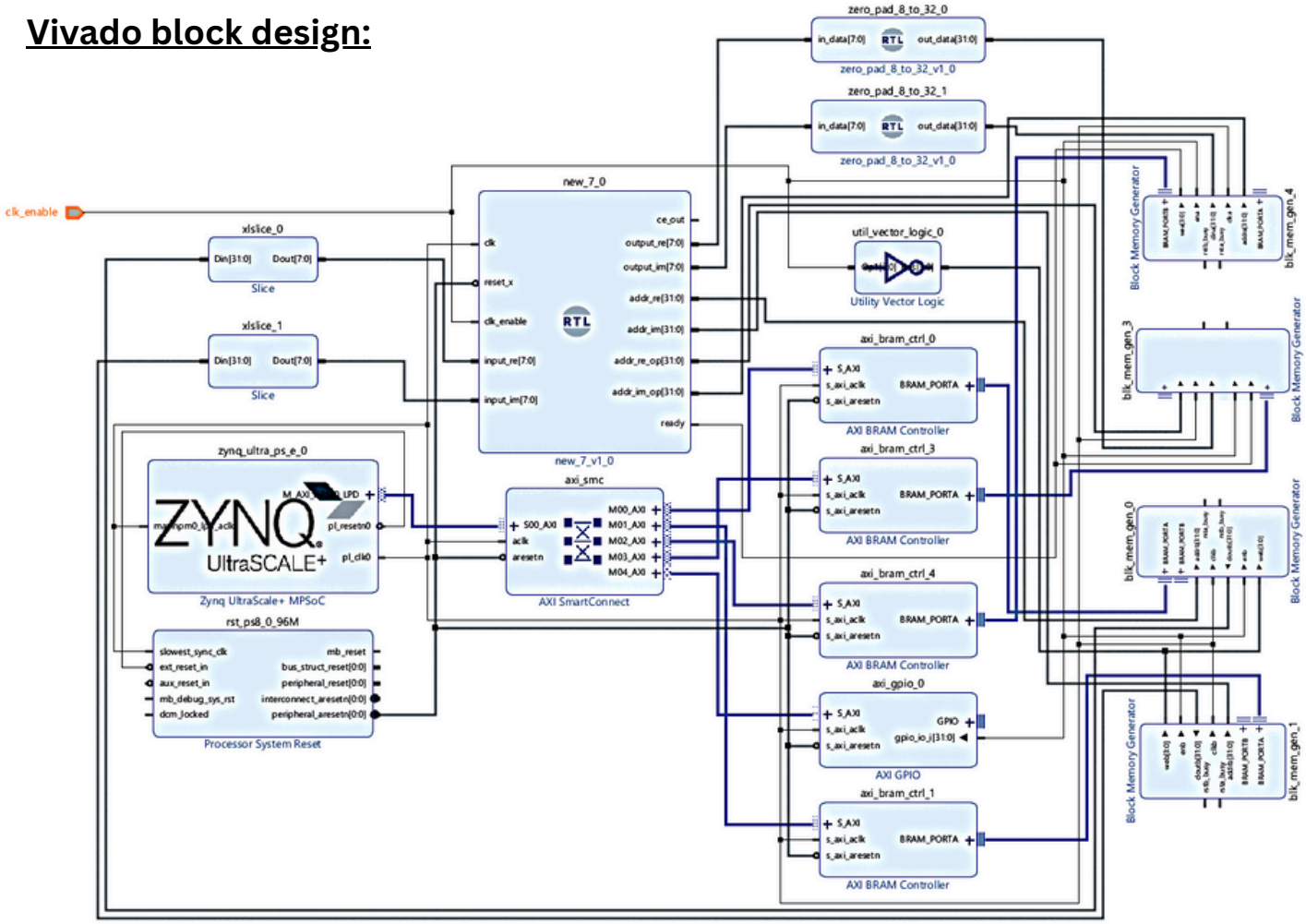
The **AXI SmartConnect** module acts as a **multiport interconnect**, enabling the **single AXI master** interface of the Zynq PS to communicate with **multiple AXI slave devices** such as B**RAM controllers, GPIO** and c**ustom IPs**. It handles **address decoding** and **arbitration**, ensuring proper routing of data transactions between connected peripherals.

To connect memory buffers, the design uses **AXI BRAM Controllers**, which convert **AXI signals** into native **BRAM control signals**. Each **BRAM controller** interfaces with a **Block Memory Generator IP**, configured as a **dual-port Block RAM. Four** such BRAMs are used in total: **two** for storing **input signals (real and imaginary parts)**, and **two** for **output data (also real and imaginary)**. These BRAMs act as **intermediate buffers**, decoupling the Zynq processor from the core RTL design. This is essential because the **RTL FIR block** does not support AXI natively, and thus **cannot interface directly with the Zynq PS.**

The **Slice** blocks are used to **truncate** or extract only the lower **8 bits** from each **32-bit BRAM data** word. Since the system uses 8-bit fixed-point inputs, only this part of the stored data is

relevant. Conversely, the **Zero Padding (zero_pad_8_to_32)** blocks are placed at the output of the RTL module, which outputs **8-bit** results. These are padded to **32 bits** to match the BRAM word width for storage, ensuring alignment and compatibility with AXI interfaces.
To feed the **external control signal clk_enable**, an **AXI GPIO** block is employed. This enables the user to give **external input to PS**, which is routed to the RTL module's **clk_enable input pin**. The **Processor System Reset** module **synchronizes all resets** within the design to a common clock domain, ensuring clean startup behavior across all blocks.
Additional support logic is implemented using **Utility Vector Logic** blocks, which are used to generate logic blocks like **inverter**. These simplify logic routing and remove the need to add such inversions directly inside the RTL.

Vivado block design:



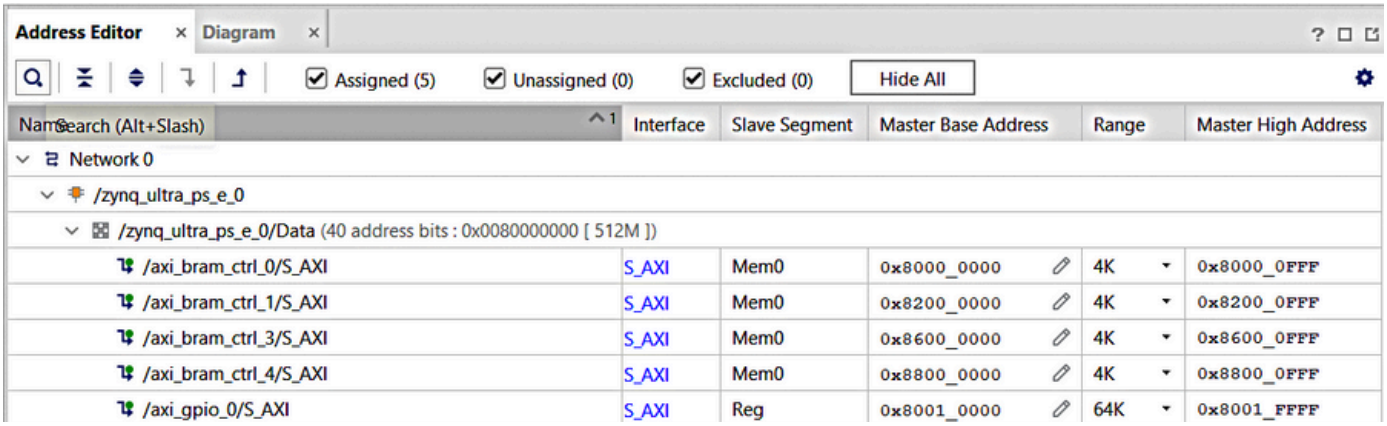
- **Block design flow**

The integration of the design followed a structured and modular flow:

- First, the **RTL core module** of the FIR system was packaged as a **custom IP** and instantiated within the Vivado block design environment. This core accepts complex fixed-point input samples and produces filtered outputs based on internal control logic.
- To provide **inputs** to this module, **two 32bit BRAMs** were instantiated: one for the **real** part and one for the **imaginary** part. **One port** of each BRAM was connected through its own **BRAM controller**. The other output of these BRAMs was passed through **Slice blocks** to extract the relevant 8 bits. These were then routed to the respective **input ports** of the FIR module.
- The output of the FIR module, also **8-bit** wide, was passed through **Zero Padding** blocks to expand the data to **32 bits**. These were connected to the input ports of the output **BRAMs**, again separately for **real** and **imaginary** parts. This setup ensured seamless write-back of results, allowing the processor to later retrieve and analyze the filtered data. Other port of the BRAM is connected to BRAM controller.

- The custom input pin **clk_enable** signal was managed using a **GPIO** pin, which was routed to the Zynq PS through **AXI GPIO** block. It was also routed to the **clk_enable port** of the RTL block.
- The **web** port of the **input BRAM** is connected to the **inverted clk_enable** signal, and that of the output BRAM is connected to **ready** port of the **RTL block**.
- the **enb** ports of all the **BRAM** are connected to the **clk_enable** pin.
- Address buses are connected to the corresponding BRAM addr port.
- After all connections were made manually, **Run Connection Automation** was used to automatically generate the necessary **auxiliary connections for AXI**, including **clock**, **reset**, and **interconnects**. Once this automation step was complete, remaining manual connections were finalized for clocks, resets, and data routing.
- All **clk ports** of blocks are connected to the **clk** line.
- As all the BRAMs have **specific address mapped** in the memory to and from the data has to be stored and fetched respectively. For this we changed our **RTL code of the testbench address generator** for both input and output logic and **replaced the address with the address of the BRAM in FPGA**.

- **Memory address mapping of the BRAMs:**



- After the design we need to **validate the design** from the option available at the top.
- To use this block design in Vivado project we need to right-click on the design and select **create HDL wrapper**.

- **Simulation of Vivado block design:**

To verify the block design correctness we simulated it by **Run simulation**. As there is **no testbench** file and the **PS part is not programmed**, we have to give **forced clock and constant** to **clk, clk_enable,reset_x** signals and give **random input** to check the functionality.

- **Post simulation Vivado design flow**

After the simulation we proceed toward the **bitstream generation** to implement it in FPGA. :

- First we **Run Synthesis**.
- After synthesis, we opened the **schematic**. There, we allocated our **clk_enable** pin to a physical Dip switch on the FPGA.

Name	Direction	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type
clk_enable	IN			A15	☑	67	LVCMOS18	1.800				PULLDOWN

- After saving this option automatically generated an **.xdc file (design.xdc).**

- **Simulation result of Vivado block design:**

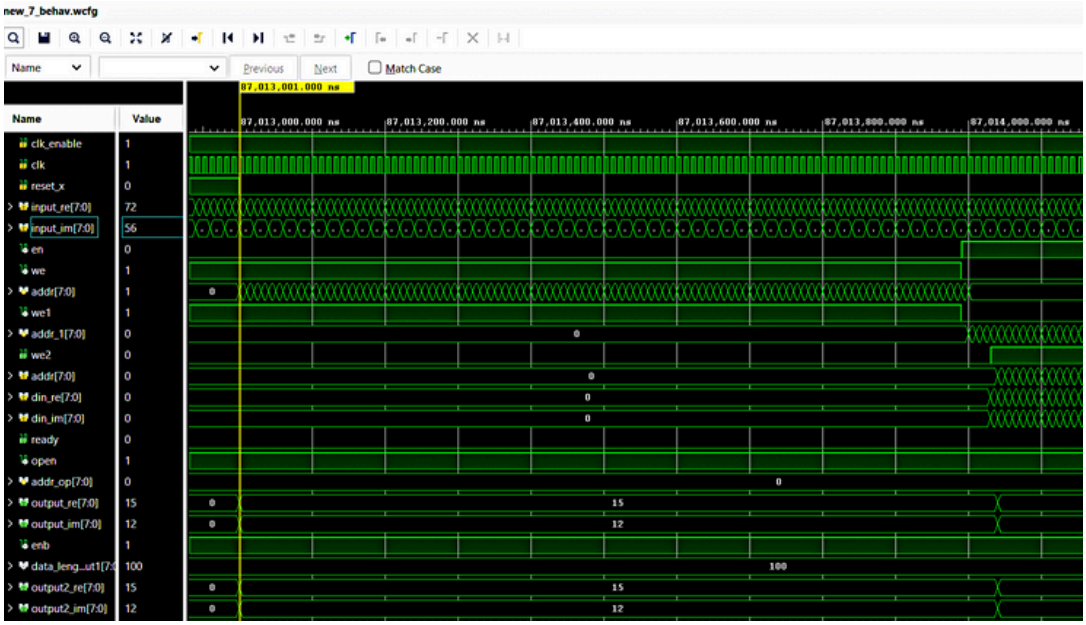


Figure 1: Input data feeding to BRAM and input data fetching from the BRAM

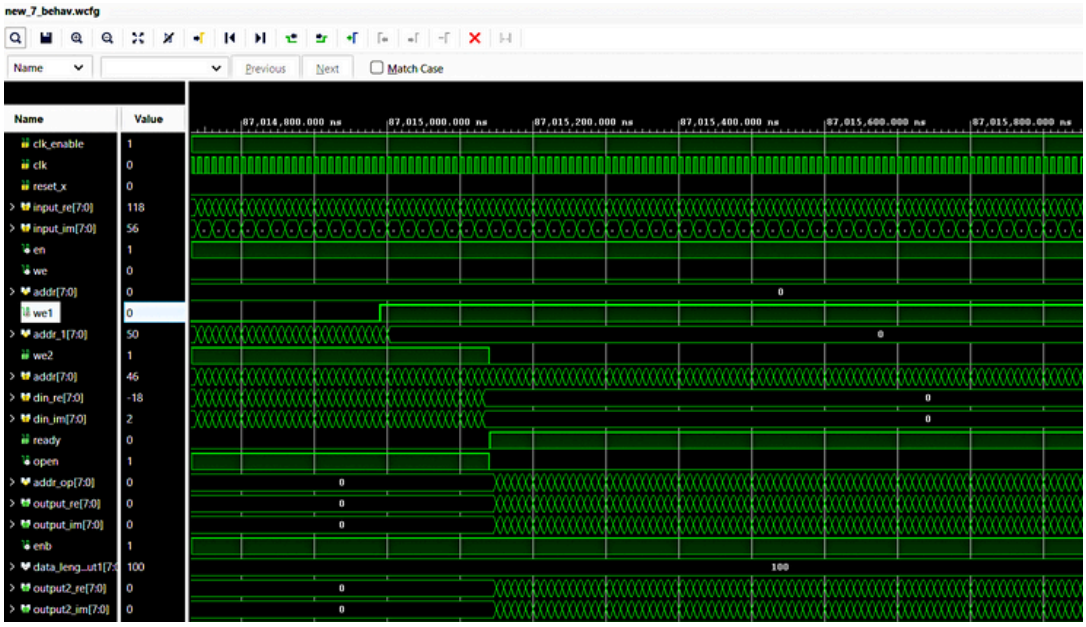


Figure 2: output data feeding to BRAM and output data fetching from the BRAM

- **Synthesised Schematic of Vivado design:**

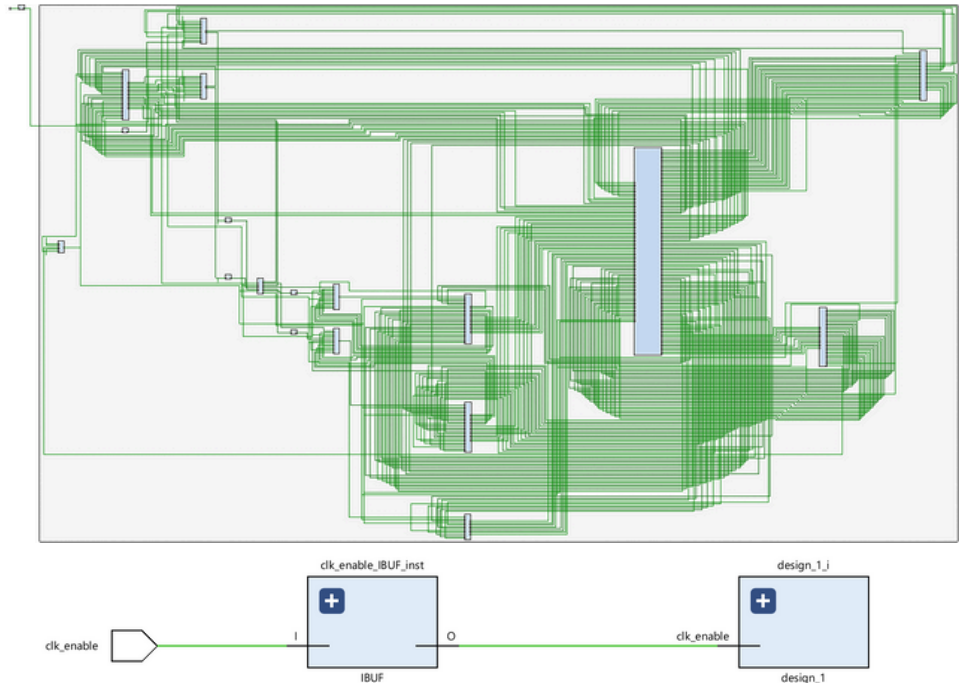


Figure 1: Schematic of Vivado design

- Then we directly run **Generate bitstream**, which automatically run **implementation**. we need to check for any **errors** or **critical warnings**, if any should be corrected.
- After generating bitstream selected **Write Bitstream** to write the bitstream on the hardware.
- after successful write bitstream the Hardware Manager is activated this allows to connect the hardware **ZCU106 FPGA**. Connection over Network can be done using **LAN cable** along with that use **USB to JTAG connections.**
- Select **Open Target → Auto Connect → Programme Device .**

IMPORTANT NOTE:

- Possible Error during write bitstream – **Out of Context (OOC) module error**, where the module are set as OOC mode. Solve this – Go to S**etting→ Synthesis → More Option uncheck the OOC** any option do this for the Implementation and Bitstream also.
- For enabling **UART communication,** we need to modify the **I/O configuration** of the **VIVADO IP design of ZYNQ processor**, **enabling** the **UART 0** option.

Programming in Vitis

After generating the complete bitstream, we need to **export** the design to **Vivado**. The design is exported from Vivado. Go to **File → Export → Export Hardware → Including Bitstream → Give file location** this will generate a **.xsa** file which is the bitstream or the design file.

Generating the Vitis file:

After completing the hardware design in Vivado and successfully generating the **.xsa (Xilinx Support Archive)** file, the next step is to move to the **Vitis IDE**. The .xsa file contains all the necessary hardware information, including the Zynq processor configuration, memory mapping, and peripheral connections. This file is imported into Vitis to create a new platform project, where we can now **develop and program the Zynq processor using C/C++ code**. In this stage, we will write software to send input data to BRAM, trigger the filtering operation, and read the processed output data from the system.

Generating the Vitis file:

- On **Vitis IDE,** a workspace location needs to be provided where all the working files will be stored. IMPORTANT
- Inside IDE, **create Platform** name it →then **browse for our design.xsa** file, operating systems – **Standalone,** Processor – **PSU_cortexaXX, 64 bits**.
- Then we need to create the A**PPLICATION PROJECT** for our processor.
- Under the **Application project → src → create a new file (.c/.c++),** that will be the source code that will run in our **ZYNQ SoftCore.**
- After writing the complete source file **BUILD** the project or click **CTRL + B.** If everything runs well this will generate a **Binaries file** which will have a programmable **.elf** ,which will run on the Zynq processor.
- The output will be displayed on **Vitis terminal**, which will communicate with the ZYNQ PS over **UART** and display our result on the computer system.

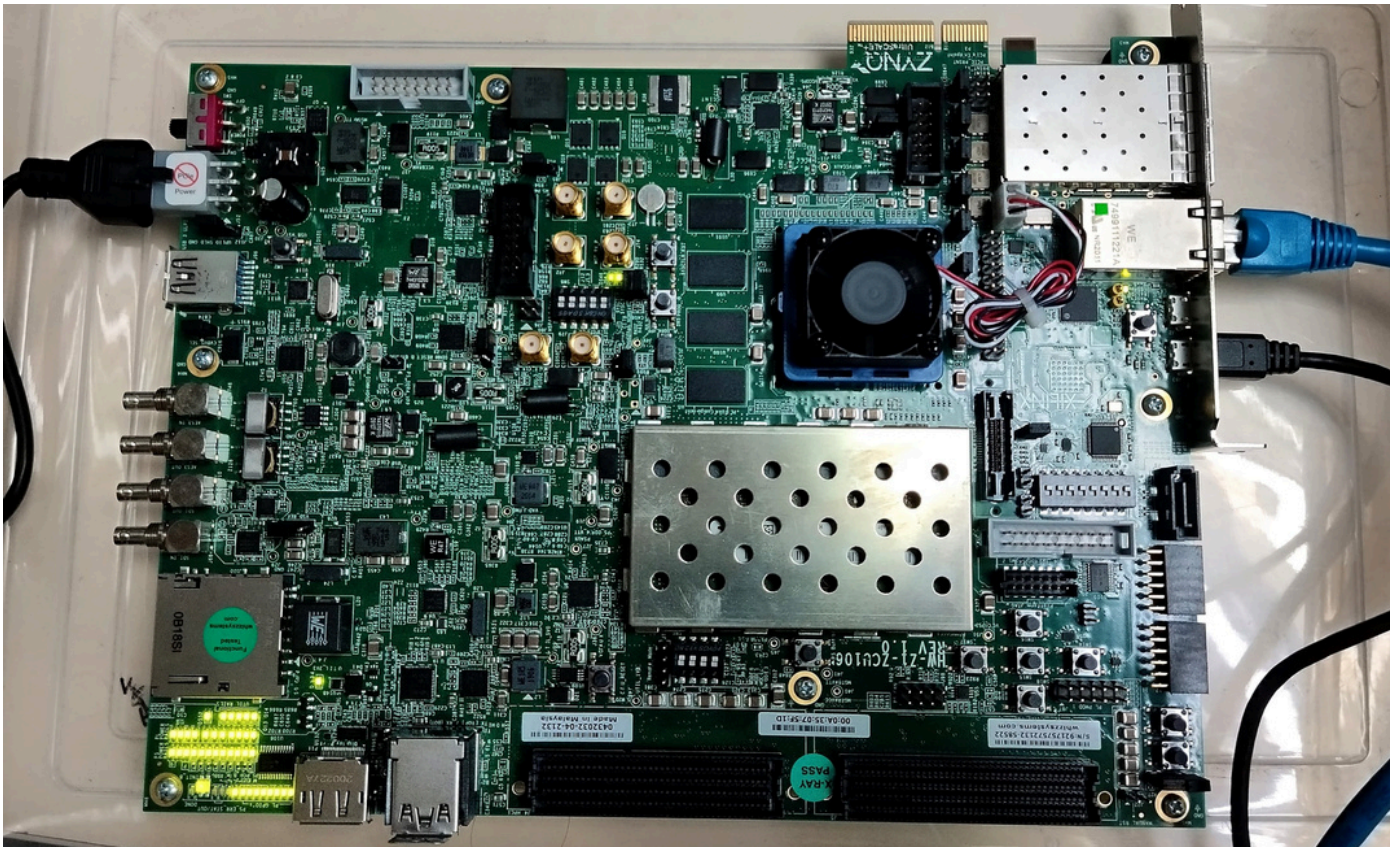
Important Note:

Only **100 samples** of input data should be fed at once. The system needs to be **reset** before next set of input data.

SCOPES OF IMPROVEMENT:

- **Enhancing System Robustness:** The current system processes **100 input** samples, generating approximately **120 output** samples. To avoid interference between consecutive data sets, the next set of input should be provided after at least 120 clock cycles, ensuring the current output completes before new input begins.
- **Improved Input Data Validation:** Present testing is done using **randomly generated input** data. For better reliability, the system should be tested with **real-world sensor data** or actual signal inputs, which would provide a more practical evaluation of system performance.
- **Optimization for Higher Performance:** The design can be further improved by applying **pipelining techniques** within the filter architecture. This would reduce critical path delay, enabling the system to process data **faster** and handle **higher throughput** while maintaining accuracy and stability.

Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC[ZCU106] FPGA working model



Study sources and references:

- MATHWORKS HELP CENTRE for MatLab and SIMULINK: https://www.mathworks.com/help/index.html
- AMD TECHNICAL INFORMATION PORTAL DUCUMENTATIONS: https://docs.amd.com/search
- OTHER SOURCES LIKE: Google, YouTube, GitHub