

EASY-FILTER: A DESIGN, VERIFICATION, AND VALIDATION TOOL FOR  
FINITE IMPULSE RESPONSE (FIR) FILTER

by

Anshu Kumari, B. Tech

A thesis submitted to the Graduate Council of  
Texas State University in partial fulfillment  
of the requirements for the degree of  
Master of Science  
with a Major in Engineering  
December 2020

Committee Members:

Semih Aslan, Chair

Damian Valles

Dan Tamir

Bill Stapleton

**COPYRIGHT**

by

Anshu Kumari

2020

## **FAIR USE AND AUTHOR'S PERMISSION STATEMENT**

### **Fair Use**

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgment. The use of this material for financial gain without the author's express written permission is not allowed.

### **Duplication Permission**

As the copyright holder of this work, I, Anshu Kumari, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

## **DEDICATION**

My Family

## **ACKNOWLEDGEMENTS**

Firstly, I would like to thank my thesis advisor, Dr. Semih Aslan of the Ingram School of Engineering, at Texas State University, for giving me the excellent opportunity to complete my thesis under his supervision. During my master's thesis, I experienced the birth of my first child. Dr. Semih Aslan was very understanding, and his patience regarding this thesis was much appreciated. His expertise in digital FIR filters and chip verification was invaluable to my thesis. Thank you for all the advice, moral support, and patience in guiding me through this thesis.

I would wish to express my gratitude to my thesis committee member Dr. Damian Valles, Ingram School of Engineering, and Dr. Dan Tamir of the Computer Science Department at Texas State University, who have read through the manuscript. Thank you for your suggestions, which have significantly contributed to the improvement of this thesis. I would like to thank Dr. Bill Stapleton, Ingram School of Engineering at Texas State University, to support this thesis.

I would also like to thank Dr. Vishu Viswanathan, a Graduate Advisor at the Ingram School of Engineering at Texas State University. He was always accessible for all my degree process related questions and enabled me to maintain focus at schoolwork.

The most significant person with an indirect contribution to this work is my father, Mr. Karnail Singh, who has taught me the love of learning. I want to thank him and my mother for constant encouragement. Your devotion, unconditional love, support, patience, optimism, and advice were more valuable than you could ever imagine.

Special thanks to my husband, Arun Singh Kanwar, for continuous support and understanding and for providing valuable input about earlier versions of the thesis and the final preparation of the manuscript. Finally, I would like to thank my wonderful daughter Saanvi Kanwar. Whenever I felt low during this thesis, thinking of Saanvi would always give me the extra bit of inspiration to press on.

## TABLE OF CONTENTS

	<b>Page</b>
ACKNOWLEDGEMENTS.....	v
LIST OF TABLES.....	x
LIST OF FIGURES .....	xi
ABSTRACT.....	xiv
 CHAPTER	
1. INTRODUCTION .....	1
1.1 Problem targeted.....	4
1.2 Proposed solution.....	4
1.3 Hypothesis.....	4
1.4 Research contribution .....	4
1.5 Outline.....	5
2. BACKGROUND .....	7
2.1 Verification vs. validation.....	7
2.2 IC implementation flow .....	8
2.3 Verification .....	13
2.4 Functional verification .....	16
2.5 Filters .....	19
2.5.1 Classification of Filters .....	19
2.6 FIR Filters .....	21
2.6.1 FIR filter's properties.....	21
2.6.2 Parameters of FIR filters .....	22
2.6.3 Types of FIR filters.....	23
2.6.4 Z-transform .....	25
2.6.5 Transform function of discrete-time systems .....	25
2.6.6 Ideal filter approximation .....	27
2.7 Window functions.....	30
2.7.1 Rectangular window .....	31
2.7.2 Hanning window .....	33
2.7.3 Hamming window.....	33

2.7.4	Blackman window .....	34
2.7.5	Kaiser window .....	35
2.8	FIR filter design using window functions.....	37
2.9	FIR filter realization.....	39
2.9.1	Direct realization.....	40
2.9.2	Direct transpose realization.....	40
2.10	Graphical user interface .....	41
2.11	Hardware description language.....	41
2.11.1	Importance of HDLs .....	42
2.11.2	Basics of Verilog .....	43
2.12	Automating the generation.....	44
2.12.1	Python .....	44
2.12.2	Perl.....	45
3.	LITERATURE REVIEW .....	46
3.1	Functional verification .....	46
3.1.1	Design automation .....	47
3.1.2	Testbench automation .....	47
3.1.3	Testcase automation.....	49
3.2	FIR filter automation.....	49
3.3	GUI for FIR filter.....	51
4.	PROPOSED SYSTEM .....	52
4.1	Design methodology .....	52
4.2	Verification methodology .....	55
4.3	Validation methodology.....	56
4.4	Experiment.....	57
4.4.1	FIR filter generation.....	58
4.4.2	FIR filter verification .....	58
4.4.3	FIR filter validation.....	58
5.	EXPERIMENTAL SETUP.....	59
5.1	Experiment set-up .....	59
6.	DESIGN IMPLEMENTATION .....	70
6.1	Filter design .....	70
6.1.1	Low-pass filter design.....	70



6.1.2	High-pass filter design .....	71
6.1.3	Band-pass filter design.....	72
6.1.4	Band-stop filter design.....	73
7.	DESIGN SIMULATION & VALIDATION .....	75
7.1	Design simulation .....	75
7.1.1	Testbench .....	76
7.1.2	Floating-point.....	76
7.1.3	Converting to floating-point .....	78
7.2	Design validation .....	79
8.	RESULT EVALUATION AND COMPARISON.....	81
8.1	Easy-filter versus MATLAB.....	81
8.1.1	Low-pass filter results.....	81
8.1.2	High-pass filter results .....	85
8.1.3	Band-pass filter results.....	89
8.1.4	Band-stop filter results.....	95
9.	CONCLUSION AND FURTHER RESEARCH .....	101
9.1	Further research .....	102
	APPENDIX SECTION.....	103
	REFERENCES .....	126

## LIST OF TABLES

Table	Page
2-1: Frequency response of four standard ideal filters.....	29
2-2: Comparison of window functions .....	31
2-3: Values of parameter $\beta$ .....	36
6-1: Low-pass filter specifications.....	71
6-2: High-pass filter specifications .....	72
6-3: Band-pass filter specifications.....	73
6-4: Band-stop filter specifications .....	74
7-1: An example of floating-point conversion.....	79
8-1: Low-pass filter's coefficients comparison.....	82
8-2: Low-pass filter's Implementation results .....	85
8-3: High-pass filter's coefficients comparison.....	86
8-4: High-pass filter's Implementation results .....	89
8-5: Band-pass filter's coefficients comparison .....	91
8-6: Band-pass filter's Implementation results .....	95
8-7: Band-stop filter's coefficients comparison.....	96
8-8: Band-stop filter's Implementation results .....	100

## LIST OF FIGURES

Figure	Page
1-1: A decreasing trend in the first silicon success rate [3] .....	1
1-2: Types of flaws resulting in silicon re-spin [3] .....	2
1-3: The root cause of logic/functional flaws [3] .....	3
2-1: Verification Vs. Validation .....	7
2-2: Functional Verification Paths .....	8
2-3: Validation of System .....	8
2-4: ASIC Implementation Flow .....	9
2-5: Relative cost of bugs at different stages of the design cycle .....	13
2-6: Type of Verification Methods .....	15
2-7: The design behavior space .....	17
2-8: Design and Verification gaps [12] .....	18
2-9: Filters classification .....	20
2-10: Filter types .....	24
2-11: Block diagram of a linear discrete-time system .....	26
2-12: Transfer functions of four standard ideal filters .....	28
2-13: Rectangular window in the time domain[23] .....	32
2-14: Hanning window in the time domain [23] .....	33
2-15: Hamming window in the time domain [23] .....	34
2-16: Blackman window in the time domain [23] .....	35

2-17: Block diagram of the direct form of FIR filter .....	40
2-18: Block diagram of the direct-transpose form of FIR filter.....	41
4-1: Block diagram of filter design automation.....	54
4-2: The verification methodology .....	55
4-3: An example of .do file .....	56
4-4: The validation methodology.....	57
5-1: Basic block diagram of the experiment setup.....	59
5-2: GUI's user input section.....	60
5-3: The proposed system's graphical user interface.....	62
5-4: FIR filter's coefficients window in text and impulse form .....	64
5-5: Input signal window .....	65
5-6: Verilog options and Verilog code window.....	66
5-7: MATLAB options and MATLAB code .....	67
5-8: ModelSim simulation window .....	68
5-9: Comparison: GUI versus MATLAB coefficients.....	69
7-1: The FIR filter's simulation waveforms .....	75
7-2: The testbench set up of the FIR filter .....	76
7-3: Single precision floating point.....	77
8-1: The low-pass filter's coefficient's simulation waveform.....	83
8-2: The low-pass filter's frequency response .....	84
8-3: Filtered output comparison.....	84

8-4: The high-pass filter's coefficient's simulation waveform.....	87
8-5: The high-pass filter's frequency response .....	87
8-6: Filtered high-pass filter's output comparison.....	88
8-7: The band-pass filter's coefficient's simulation waveform .....	92
8-8: The band-pass filter's frequency response .....	93
8-9: Filtered band-pass filter's output comparison .....	94
8-10: The band-stop filter's coefficient's simulation waveform .....	98
8-11: The band-stop filter's frequency response .....	99
8-12: Filtered band-stop filter's output comparison .....	99

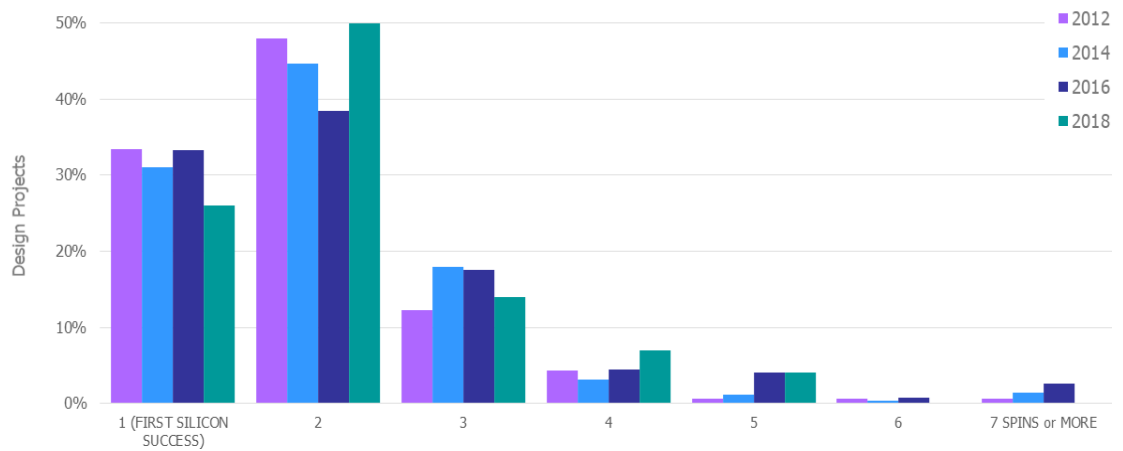
## **ABSTRACT**

Due to increasing hardware design complexity and cutting-edge competition for the short time-to-market requirement, functional verification becomes the primary challenge in the hardware design development project. The essential parts to verify any systems are design code and testbench code. However, it is very labor-intensive to write these codes and prone to manual errors. The idea behind this thesis is to develop a user-friendly graphical user interface (GUI) that helps Verification Engineers to generate design and testbench code. GUI also validate these codes by comparing with MATLAB results more efficiently and in less time. Often, it is challenging to finish the debugging in due time because of obvious reasons such as coming across several design changes at any time. It means that they must rewrite their design and test benches. These changes become a significant issue if they happen right before tape out. The proposed system automates the design implementation of the FIR filter, which engineers can easily modify in less time. The first step is to generate the design and testbench code of the FIR filter and simulate using Modelsim. The second step is to validate this design by comparing its results against already existing well established MATLAB's results. Filters of some sorts are essential to the operation of most electronic circuits. Our proposed system can save a significant amount of time for engineers because it can generate design code, testbench code, and validate it, all in one system. The GUI generated Verilog codes are synthesizable. The simulation results show that GUI based FIR filter's design is fast, convenient, flexible, and error-free.

## 1. INTRODUCTION

Functional verification (FV) is a significant step in the development of today's complex digital designs. In the latest Integrated Circuits (IC) designs, the deep-submicron feature sizes have shifted the emphasis from design to verification [1]. Designers must design ICs with an excess of 50 million equivalent gates and still meet cost and time-to-market constraints. So, verification is the main topic for research and development in the Electronics Design Automation (EDA) industry.

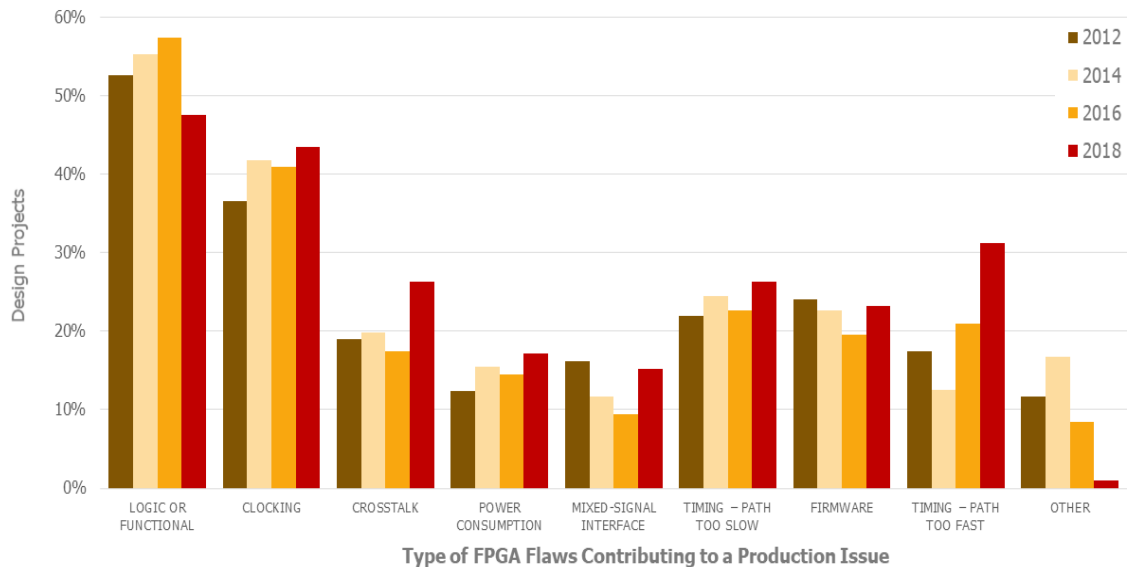
With the continuous increase in design complexity, the probability of Integrated Circuits (IC) failure increases [2]. A study by Wilson Research Group in 2018 shows the rate at which a given IC function satisfactory in first silicon spin is dropping. Figure 1-1 shows that achieving first silicon success is getting worse while achieving second silicon success has improved [3].



**Figure 1-1: A decreasing trend in the first silicon success rate [3]**

The same studies also described the sources of errors in chip design. Chips fail for many reasons like clocking, timing-path issues, power issues, and logic/functional flaws.

As shown in Figure 1-2, logic/functional defects are the most significant cause of flawed silicon that required re-spin.



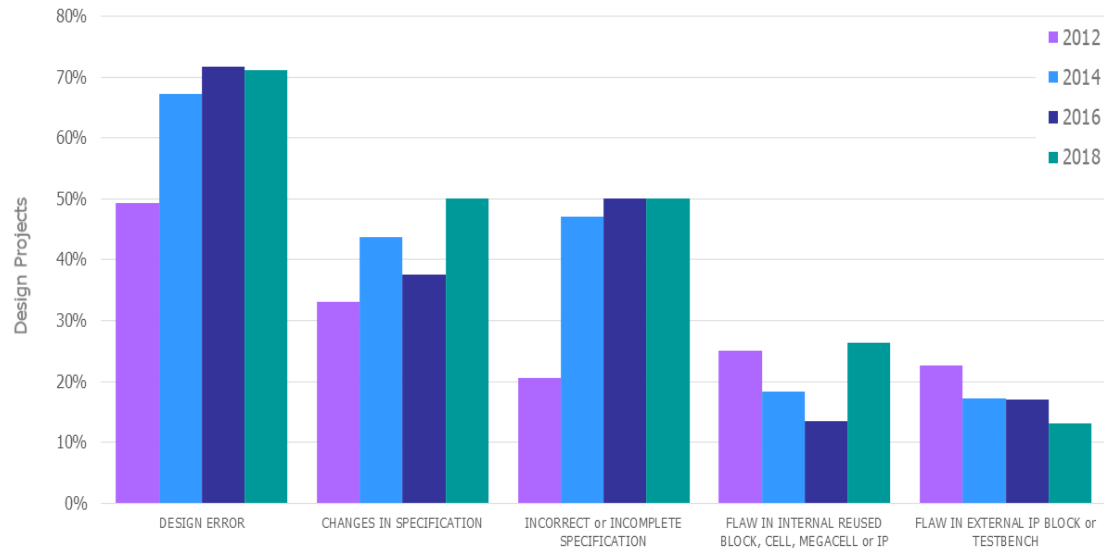
**Figure 1-2: Types of flaws resulting in silicon re-spin [3]**

Since the logic/functional errors were more common than the others, the same research examines the root cause of logic/functional flaws. Design errors were the main reason for functional flaws. The flaws due to changing, incorrect, and incomplete specifications are also typical. According to Figure 1-3, flaws fall in three main categories:

- **Design errors:** About 82% of designs with re-spins resulting from logic/functional flaws had design errors. It means that particular corner cases did not cover during the verification process, and bugs remained hidden in the design flow through tape-out.
- **Specification errors:** About 47% of designs with re-spins resulting from logic/functional flaws had incorrect/incomplete specifications. Moreover, 32% of



designs with re-spins resulting from logic/functional defects had changes in specifications.



**Figure 1-3: The root cause of logic/functional flaws [3]**

- **Reused modules and imported IP:** About 14% of all chips that failed had bugs in reused components or imported IP (Intellectual Property).

So, this data shows that silicon re-spin is very common. Chip re-spin is extremely expensive, and it also requires additional development time [4]. Thus, companies that can control this trend have a considerable advantage over their competitors, both in terms of the subsequent reduction in engineering costs and the business advantage of being to market sooner and with high-quality products. The key to time-to-market success, for many projects, is verification.

## **1.1 Problem targeted**

The problem targeted in this thesis is achieving low cost, reducing manual errors, and reducing project time spent in the design and verification of the FIR filter.

## **1.2 Proposed solution**

The proposed solution is to develop a system that generates the FIR filter's design and testbench, then verifying using ModelSim and validate it against MATLAB results. FIR filters are of four types: low-pass, high-pass, band-pass, and band-stop. For this, we propose to conduct a set of experiments, each of which includes the design, testbench, verification, and validation. The system uses a Graphical User Interface (GUI) for easiness. This interface is suitable for teaching purposes, either at undergraduate or graduate levels.

## **1.3 Hypothesis**

Using the proposed system reduces chip cost and time to market.

## **1.4 Research contribution**

In this thesis, FIR filter designs using Perl and Python languages. Main contributions are summarized as follows:

1. Any user can generate a design and testbench of FIR filter of any type (low-pass, high-pass, band-pass, band-stop) and any order in seconds using easy to use GUI. Along with design and testbench, it verifies and validates the FIR filter that saves time and cost.
2. Easy-filter can design four types of digital filters using the same specifications and GUI in two different languages: Verilog and MATLAB. It reduces human

errors, especially for FIR filters with a large number of coefficients.

3. Easy-filter designed filter is reconfigurable, which means that the filter can be reconfigured anytime by data that are input to the GUI. GUI generated design and testbench Verilog codes run on ModelSim that opens directly from GUI. The filter design was verified by comparing the Python generated coefficients to the MATLAB generated coefficients.
4. Easy-filter catches floating-point to decimal conversion error in two different formats: total average error percentage, the plot of the absolute error in each coefficient
5. Easy to use GUI that does not require any prior knowledge of any programming language, so it is suitable for teaching purposes. GUI can run the FIR filter design automatically on ModelSim and MATLAB software.
6. GUI can compare Python generated coefficients with MATLAB generated coefficients and shows the total average error percentage.
7. Easy-filter design the FIR filter, simulate using ModelSim and validate Python generated coefficients with MATLAB generated coefficients, using a single GUI.

## **1.5 Outline**

This thesis is divided into nine chapters. Following the introduction in Chapter 1, we have Chapter 2, which provides background information about function verification importance and challenges, FIR filter, and graphical user interface. Chapter 3 includes the literature review, which provides information about similar works performed by other people. Chapter 4 gives information about the methodology used in this thesis for design, verification, and validation, and experiments performed as a part of this thesis. Chapter 5

explains the experimental setup, and Chapter 6 explains the design implementation.

Chapter 7 adds the design simulation and validation process, and Chapter 8 compares the results of the experiments performed against MATLAB; it also includes synthesis results.

Chapter 9 includes conclusions and future work recommendations.

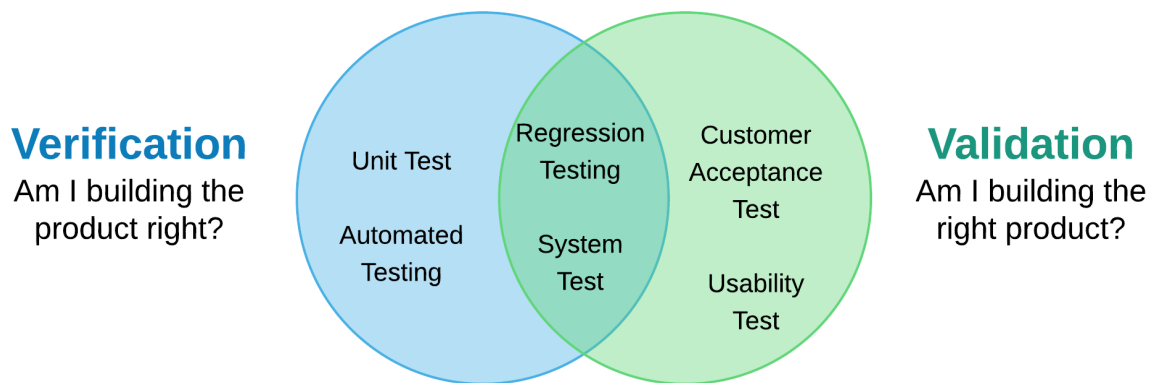
## 2. BACKGROUND

The chapter provides background related to functional verification and FIR filter. The chapter starts with a difference in verification and validation; then, it provides general details about functional verification and FIR filter.

### 2.1 Verification vs. validation

The difference between verification and validation is always confusing.

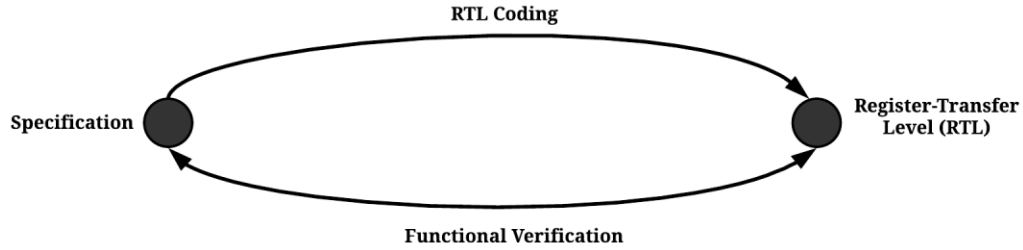
Verification is a test of a system to prove that it meets all its specified requirements at a particular stage of its development. On the other hand, validation is an activity that ensures that an end product meets stakeholder's true needs and expectations. Figure 2-1 shows the difference between verification and validation [5].



**Figure 2-1: Verification Vs. Validation**

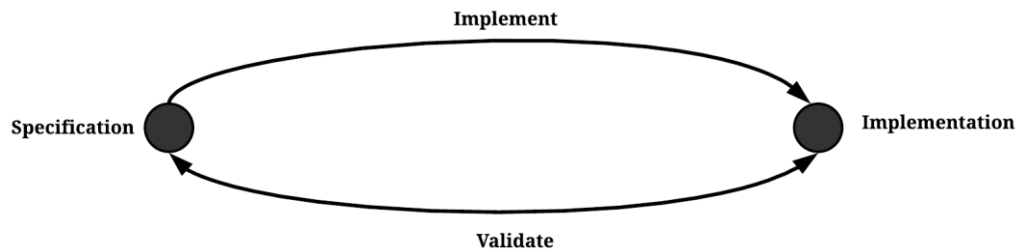
Verification is a process to demonstrate the functional correctness of the design.

The primary purpose of “functional” verification is to ensure that a design meets its functional intent. The convergent path model, as shown in Figure 2-2, functional verification, reconciles a design with its specifications [6].



**Figure 2-2: Functional Verification Paths**

The desired behavior of the system must be known for system validation. The desired behavior description is in specifications. Specifications describe what a system must do; it does not explain how to do it. A system that is supposed to implement the desired behavior is called an implementation. So, validation checks whether an implementation complies with its specification, as shown in Figure 2-3 [6].

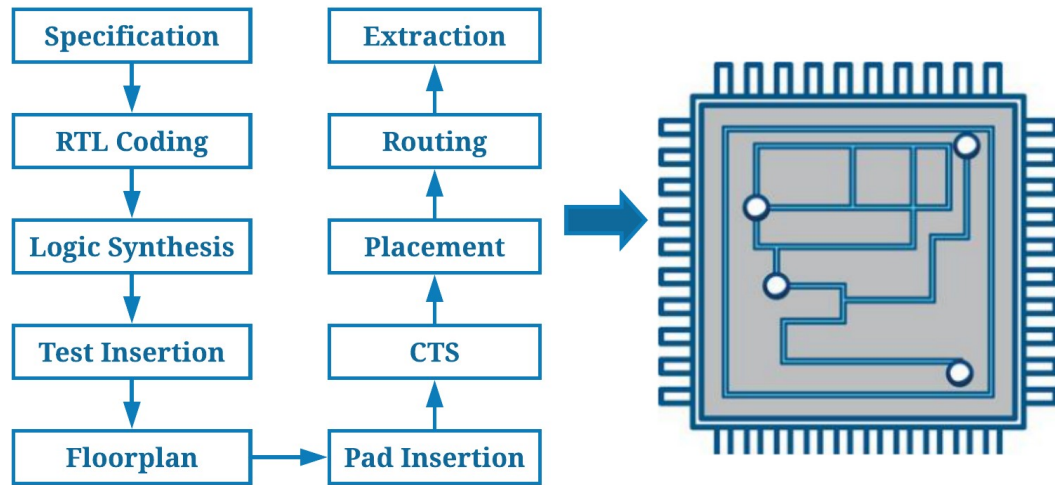


**Figure 2-3: Validation of System**

## 2.2 IC implementation flow

IC development process involves many steps to produce a final circuit. IC development flow depends on the technology used and circuit type (digital, analog, or mixed-signal). The flow also differs if a re-programmable device such as a Field-Programmable Gate Array (FPGA) is used. Figure 2-4 shows a generic flow for a digital Application Specific Integrated Circuit (ASIC). This flow based on standard cell libraries methodology, widely used for developing digital circuits [7]. These circuits usually have enormous complexity, high design, and production cost and benefit from Functional

Verification (FV). Fixing a logical error on these types of circuits is very costly.



**Figure 2-4: ASIC Implementation Flow**

Design goes through some form of transformation on each step shown in Figure 2-4. The process of designing an ASIC (Application Specific Integrated Circuit) is very complex, and it involves many steps. Although the end product is small (in nanometer), but the process of designing is long and challenging.

ASIC design flow includes design conceptualization, chip optimization, logical/physical implementation, and design validation and verification. The overview of each of the steps involved in the process is as follows:

- **Step 1. Specification:** At this step, the engineer defines features, microarchitecture, hardware/software interface, Time, Area, Power, Speed with design guidelines of ASIC. The design specification uses natural language, which then transforms into Register Transform Level (RTL) code. Two different teams involved at this juncture:
  - Design team: Generates RTL code

- Verification team: Generates test bench
- **Step 2. RTL code:** Specifications converted to synthesizable Hardware Description Language (HDL), like Verilog or VHDL. It includes detailed logic implementation of the entire IC. Functional verification is used to ensure the RTL code is according to the specifications.
- **Step 3. Logic synthesis:** The hardware description (RTL) transformed into a gate-level netlist using a synthesis tool like RTL Compiler and Design Compiler. The synthesizer used a standard cell library, constraints, and the RTL code to generate a gate-level netlist. Static timing analysis (STA) calculates the expected circuit timing used to optimize the circuit's optimization.
- **Step 4. Test insertion:** The design includes Design for Testability (DFT) to ensure no bug or fault escape to the production. It uses to determine if the chip function correctly after manufacturing. The design includes the following DFT structures:
- Scan path insertion: It links all registers' elements into one long shift register (scan path).
  - Memory BIST (built-in Self-Test): It uses to check RAMs.
  - ATPG (automation test pattern generation): It generates test vectors or sequential input to check design for faults generated within various circuit elements.
- **Step 5. Floorplan:** This is the first step in the physical design process. In this step, the circuit is organized and structured in a layout form for the first time. It is the process to place blocks in the chip, including block placement, design



portioning, pin placement, and power optimization. The power grid is used to feed the circuit created, and some routing or placement restrictions may be applied.

The overall dimensions and aspect ratio of the chip must be defined as well. A floorplan always takes care of the following:

- Minimize the total chip area
  - Make routing phase easy (routable)
  - Improve signal delays
- **Step 6. Pad insertion:** The chip pads are inserted in the layout according to the design constraints. The pads are the communication channels that the circuit used to communicate with the external environment. Electrostatic Discharge (ESD) should perform at this stage.
- **Step 7. CTS:** Clock tree synthesis is a process to build the clock tree that meets the defined area, timing, and power requirements. It provides the clock connection to the clock pin of a sequential element in the required time and area, with low power consumption. The clock signal of a chip needs to simultaneously reach all the sequential elements; therefore, special optimizations are performed for clock buffering and routing. To avoid massive transition, high power consumption, and increase in delays following structures used for optimizing CTS structure:
- Mesh Structure
  - H-Tree Structure
  - X-Tree Structure
  - Fishbone Structure
  - Hybrid structure

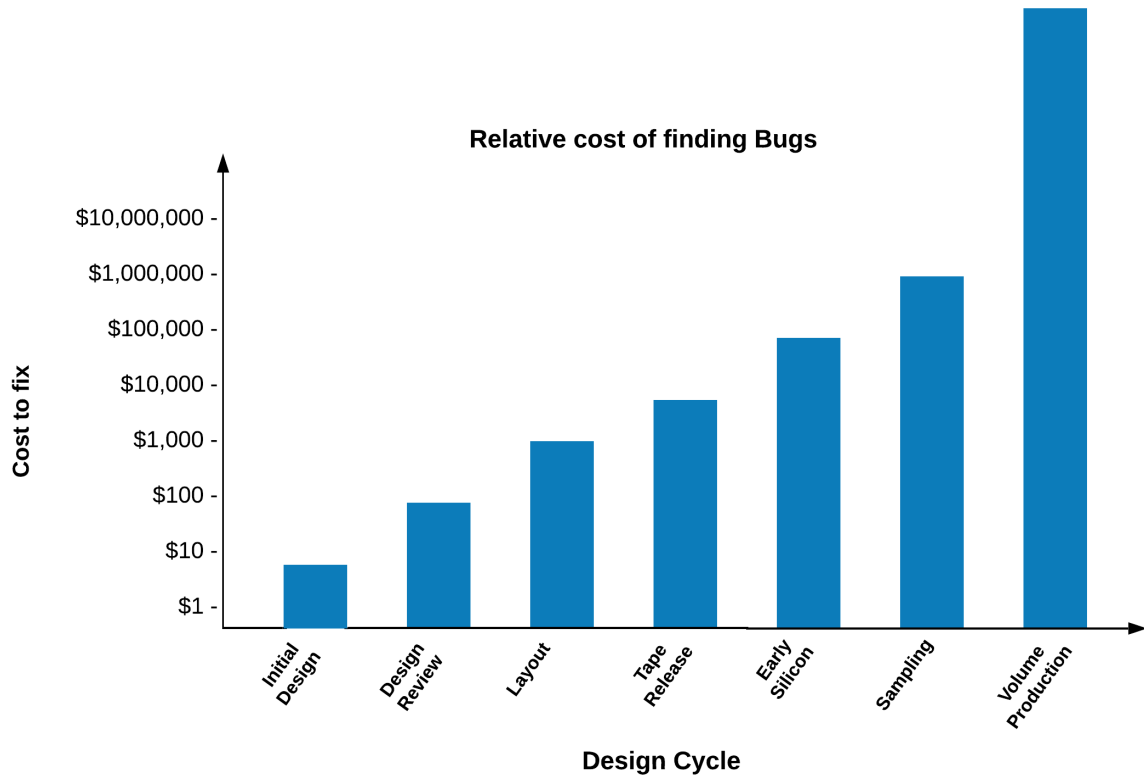
- **Step 8. Placement:** Placement places standard cells in a row. The cells that must be connected must keep close to each other. Elements other than proximity like routing congestion must consider. After the placement, no cell should overlap. A poor placement uses a large area and degrades performance. Various factors, like the timing requirement, the net lengths, and hence the connections of cells, power dissipation should be considered. It removes timing violations.
- **Step 9. Routing:** At this step, physical connections between all cells are established. Furthermore, connect pads and power rings. Routing is of the following types:
  - Global Routing: It uses delays of fan-out of wire to calculate estimated values for each net. Global routing is of two types line routing and maze routing.
  - Detailed Routing: It uses various optimization methods (timing optimization, clock tree synthesis, etc.) to calculate actual delays of wire.
- **Step 10. Extraction:** This step extracts the resistivity and capacitance of the final layout. The extracted data used to perform proper tuning of the previous steps and perform electric simulation for sign-off purposes.

The chip finally becomes ready after all these transformations. Chip goes through all these transformations, so it is not error or misinterpretation proof. So, here verification comes into play. The role of verification is to avoid errors in the design flow. Some verification processes require several resources due to the intensive simulation nature of the verification method. More resources mean high costs. The high cost can also be due to delays in deploying a design that has tight time-to-market. The high cost due to all

these reasons justified with the number of errors caught early in the design cycle.

Because if an error is caught late in the design cycle, it costs more to the company; as shown in Figure 2-5, the cost of errors increases as the design cycle stage increases [8].

Different steps of the process used different verification types; some of them explored in the next section.



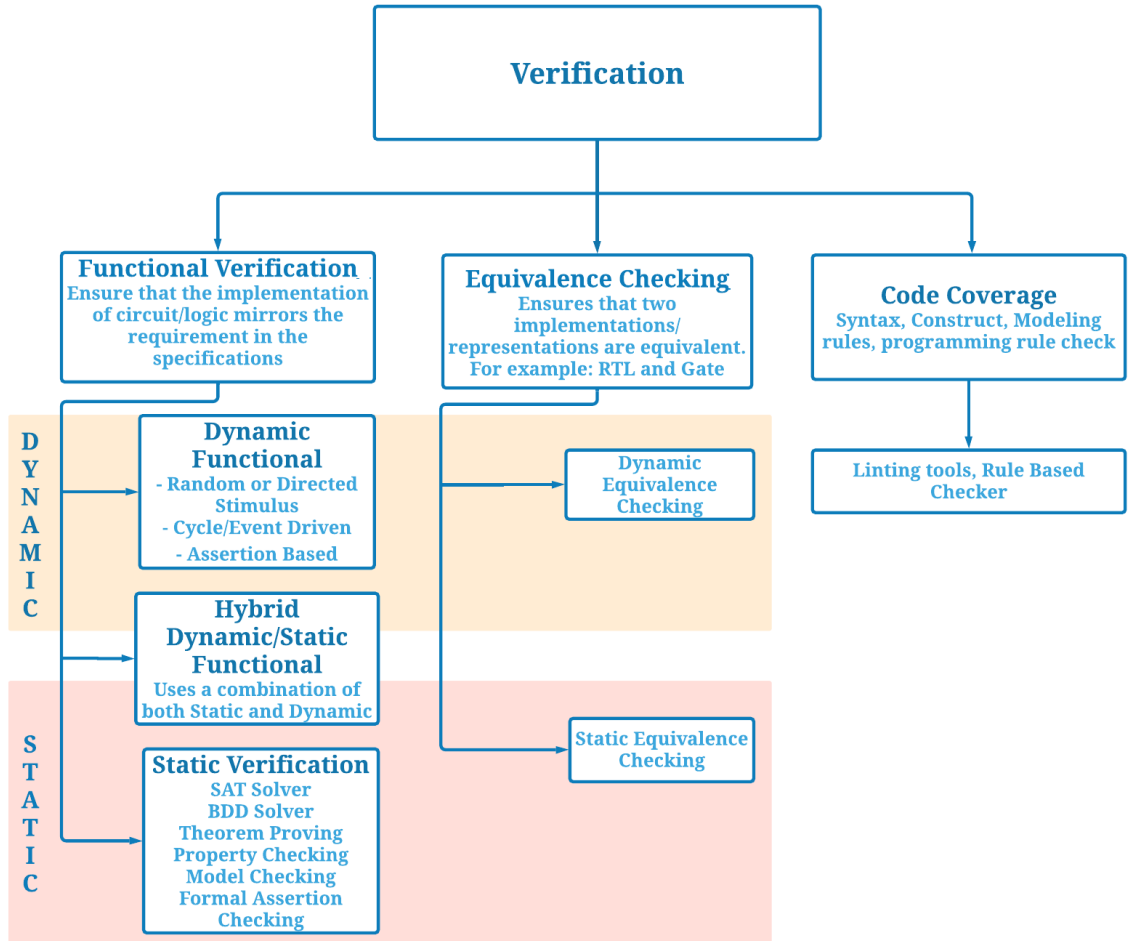
**Figure 2-5: Relative cost of bugs at different stages of the design cycle**

## 2.3 Verification

The previous section explained IC implementation flow. This flow is always executed in parallel with the verification flow. If a company can afford enough resources, two completely different teams work on each flow [1]. So, this approach provides two different views for the same design, which helps in error detection. Design specification works as a communication channel between these two teams.

As explained in the last section, each step is prone to errors or misinterpretations. So, it is essential to carefully execute each step and check the result of each step against design specifications. The first step of transformation is critical from the verification point of view, translating a written document to a hardware description language. It is highly susceptible to errors or misinterpretations. This transformation is explored throughout the design cycle. All other transformations are mechanical and automated, so they are less susceptible to errors or misinterpretations. Figure 2-6 shows various verification methods [9]. Mainly they are of three types: functional verification, equivalence checking, and code coverage. Each verification method is explained below in brief. Functional verification, mainly the dynamic one, is explained in detail in the next section.

- **Functional verification:** This is the favored method of FV and its dynamic type. It is dynamic because input patterns/stimuli are generated and applied to the design over several clock cycles. The corresponding results were collected and compared against a reference/golden model and check if it conforms with the specifications. The static FV is also called formal verification, performs the same comparison. However, it uses some sort of mathematical proof instead of simulation.
- **Equivalence checking:** It compares and checks if the two representations of the same design are equivalent or not. This type of checking is useful, especially after logic synthesis, i.e., comparing gate-level netlists against the design's RTL representation.



**Figure 2-6: Type of Verification Methods**

- **Code coverage:** It checks and reports the code lines that visited(covered) during the simulation. It is easy to collect, and it is an indirect metric to check the overall verification progress.

Some methods are shared with different techniques (e.g., assertions), so they can be misleading in different literature works. This thesis uses the following convention: a technique is a collection of methods used in conjunction. A method is an approach to prove a particular statement or property regarding a design. So, Figure 2-6 shows only methods classified by type.

The verification of design, in general, accomplished mainly two techniques:

formal and functional verification [6]. FV refers to a collection of methods: assertions, random or directed stimulus, coverage, and dynamic simulation. On the other hand, formal verification also refers to collecting methods that include property checking, theorem proving, formal assertion checking, etc. FV is a simulation-based technique and is most commonly used in the chip industry. Even though new methodologies have been proposed that can benefit from formal or semi-formal methods and even adopted in the industry; still, these methods are limited [10].

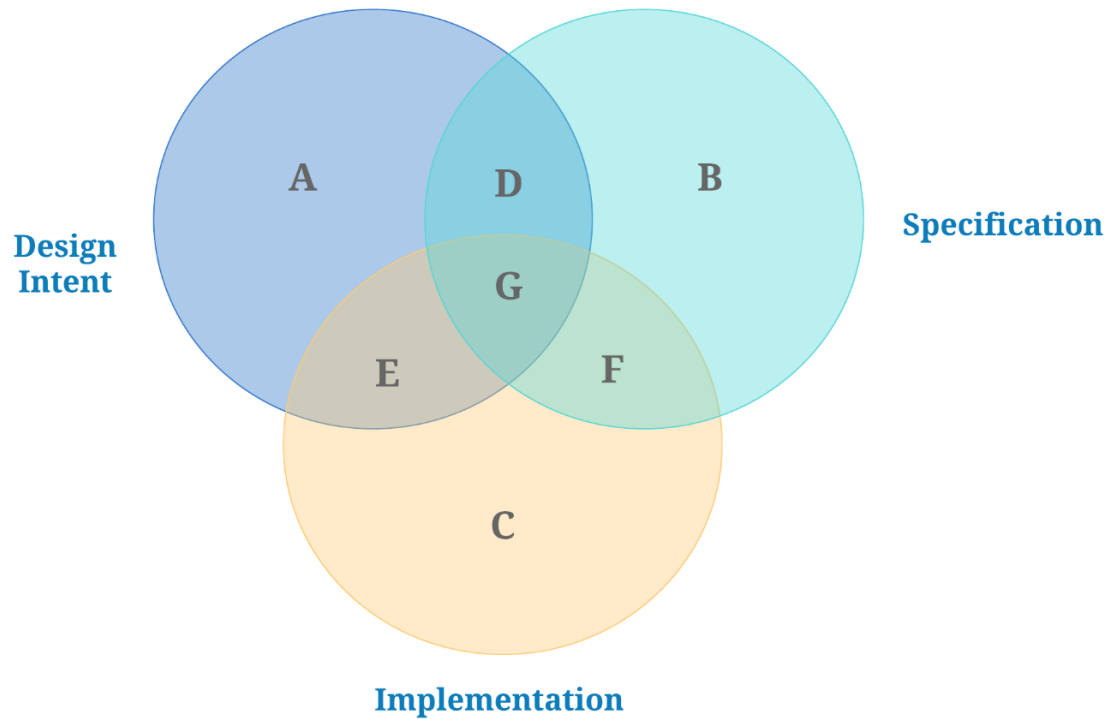
## **2.4 Functional verification**

The goal of functional verification is to prove that a design work as intended. The following are the main steps to achieve this goal [6]:

- Determine the intent.
- Determine what the design does.
- Compare the two to ensure that they match.
- Estimate the level of confidence in the verification effort.

Figure 2-7 shows how design intent, design specifications, and RTL code are related to composing space of design behavior [11]. In Figure 2-7, each circle represents a set of behaviors. The design intent is a set that includes design requirements. The system architect, along with the customer, defined the design requirements. It is an abstract of architect and customer's expectations from a particular design's functionality. The specification is a written document that tries to enumerate those functionalities exhaustively. Engineers follow the specification to do coding. The implementation is the actual intent that is coded in the RTL code. The space that is not covered by any of these circles represents the unspecified, unintended, and unimplemented behavior. The

verification tries to meet these three circles and try to bring them in coincidence.

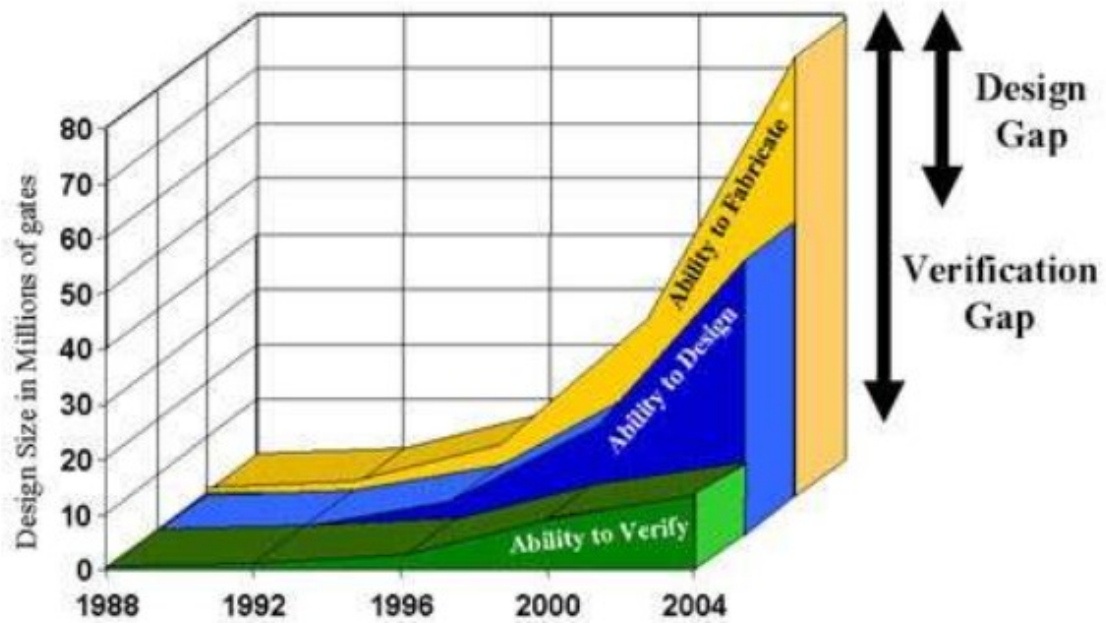


**Figure 2-7: The design behavior space**

When verification tries to match these circles from Figure 2-7, these three circles usually do not coincide and generate very definite results. Region G is the best scenario for any design as a particular intent is defined, specified, and implemented. The goal of functional verification is to maximize this region. In region D, the desired design's intent was specified but could not be implemented for some reason. There may be some functionality specified and implemented in this region, but that was not the design's intent. Region F represents it. So, these scenarios waste resources and time.

So, verification is a must for today's complicated digital design development. Verification complexity increases as hardware's complexity increases. Figure 2-8 shows that verification technology falls behind design and fabrication capability, which widens

the verification gap [12].



**Figure 2-8: Design and Verification gaps [12]**

So, in other words, the capability of the industry's current processes to fill a chip with complex logic is pretty high. However, it cannot guarantee that this logic works appropriately. Hardware Verification Languages (HVL) like *e* language, the Property Specification Language (PSL), and the SystemVerilog language are used to deal with verification complexity. The verification process is essential for the design and very hard to accomplish. Verification is considered complete if each possible scenario is applied to the Design Under Verification (DUV). Each possible output shows the design intended and specified value at every point in time [13]. Measuring verification completeness is not easy, so verification engineers use indirect metrics to measure progress.

These indirect metrics also called coverage metrics. The verification quality is obtained from coverage, either functional or structural coverage. Structural coverage is



also called code coverage because it is directly related to the RTL code. On the other hand, functional coverage is related to design functionalities. The critical element of the verification process is functional coverage—the number of transistors per chip increases, decreasing validation effectiveness. The test cases used for simulation are becoming more complex, resulting in expensive simulation and less coverage [14].

## 2.5 Filters

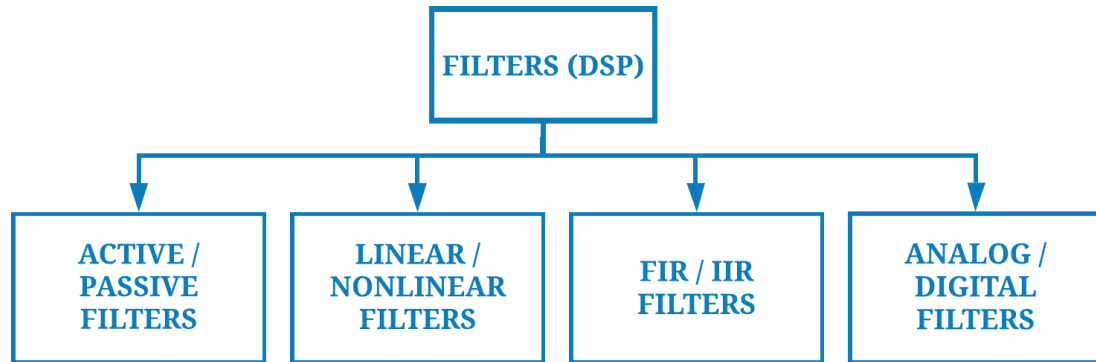
Electronics filters either separate the desirable signal frequencies from undesirable frequencies or change the frequency content that changes the signal waveform. Filters of some form are essential for most of the operations of the electronic circuits. Filters are used for two purposes to separate signals and to restore signals. A filter is used for signal separation when the signal contains interference, noise, or other undesirable signals. On the other hand, a filter is used as signal restoration when the signal gets distorted somehow [15]. The filter's primary purpose is to pass signals in a particular frequency range and reject other frequency ranges in electronic systems.

### 2.5.1 *Classification of Filters*

The broad classification of the filter is given below in Figure 2-9.

- **Analog/Digital filters:** Filters are classified as analog and digital filters based on the incoming signal. Digital filters perform mathematical operations on the discrete-time signal that reduce or enhance some aspects of the signal. On the other hand, analog filters operate on continuous-time analog signals. In digital filters, the analog signal is first processed by digital filter by digitizing and representing a sequence of numbers, then manipulating mathematically, and

finally reconstructing it as a new analog signal. In analog filters, the circuit "directly" manipulates the incoming signal [16].



**Figure 2-9: Filters classification**

- **Active/Passive filters:** Filters are classified as active and passive based on components used. Active filters use active components like amplifiers and passive components like resistors and capacitors, and it is a type of analog filter. On the other hand, passive filters use passive components like resistors, capacitors, and inductors [17].
- **FIR/IIR filters:** Digital filters classified as FIR and IIR based on impulse response. FIR filters have finite impulse response, and IIR filters have infinite impulse responses [18].
- **Linear/Nonlinear filters:** Filters are classified as linear and nonlinear based on the output signal's dependency on the input signal. Linear filters produce the output signal in the time domain resulting from processing time-varying input signal, which is subject to linearity constraint. These results are composed solely of components that have a linear response. In contrast, nonlinear filters produce an output signal that is not a function of its input signal [19].

## 2.6 FIR Filters

Finite Impulse Response (FIR) filter settles to zero in finite time. Hence, its impulse response (or response for any finite duration input signal) is finite. Its filter structure is such that it can be used to implement any sort of frequency response digitally. The FIR filter's primary characteristics are stability, linear phase, and high filter order (more complex circuits) [20]. To implement an FIR filter, it uses a series of delays, adders, and multipliers. Other names of FIR filters are non-recursive or feed-forward, or transversal filters.

### 2.6.1 *FIR filter's properties*

The following properties are characteristics of FIR filters [21]:

- It does not need any feedback for its operation.
- FIR filter has excellent delay characteristics, so it requires more memory.
- Higher-order filters use the FIR filter for tapping.
- Magnitude shaping is flexible, convenient, and implementation is easy and dependent on linear phase characteristics.
- All poles are located at the origin; it means they are located in the unit circle (a requirement for stability in Z transform), which is due to no feedback requirement; thus, it is a stable filter.
- It has only zeros (no poles), so it is also called the all-zero filter.
- To design FIR filter as a linear phase, make symmetric coefficient sequence; linear phase; change phase proportional to frequency, which corresponds to equal delay at all frequencies.

### 2.6.2 *Parameters of FIR filters*

- **Ripple:** specifies the peak to peak level in decibels. It describes the filter's amplitude variation within a band. A smaller ripple is always preferable as it represents a more consistent response. Passband ripple should be as low as possible.
- **Bandwidth:** defines the frequency width of the filter's passband. The bandwidth is the same as the cut-off frequency in a low-pass filter. In a band-pass filter, it defines the difference between upper and lower frequencies at -3dB points.
- **Attenuation:** input signal acquires amplitude loss after passing through a digital filter; it is measured in dB. It defines as a ratio of amplitudes, at a given frequency, the filter's output signal over the filter's input signal.
- **Passband edge frequency:** defines the start of the passband, the signal fully passed in this region without any attenuation.
- **Stopband edge frequency:** defines the start of stopband, the signal attenuated in this region without passing any signal—minimum attenuation in the stopband reached at stopband frequency ( $F_s$ ).
- **Filter coefficients:** the set of constant or tap weights multiply with delayed signal sample value in digital filters. Digital filter design needs to exercise to determine its coefficients to get the desired frequency response. The coefficients for the FIR filter, by definition, are the impulse response of the filter.
- **Filter order:** describes a number that is the highest exponent either in the numerator or in the denominator of a digital filter's z-domain transfer function. There is no denominator in the transfer function in FIR filters, so filter order is

just the number of taps used in the structure. The large filter order is preferable, as it provides a better frequency magnitude response performance of the filter.

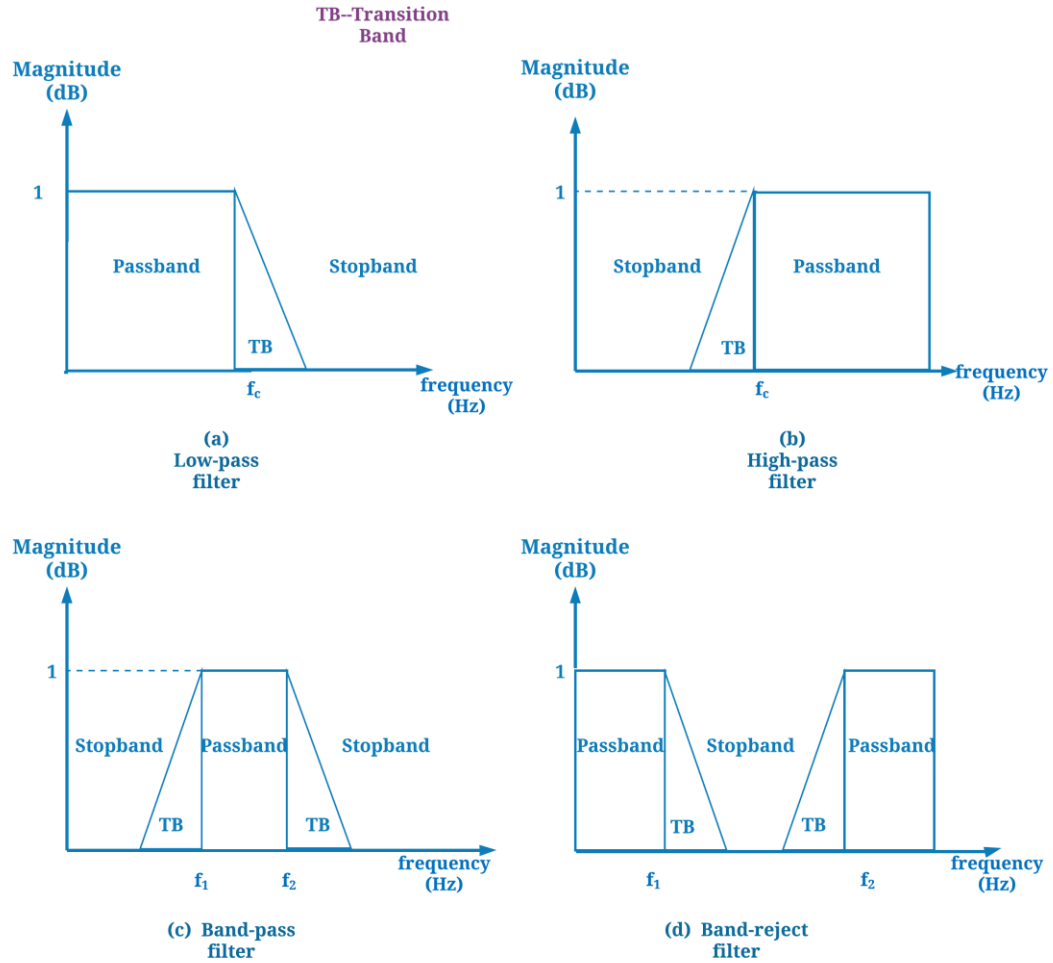
- **Transition region:** describes a frequency range that is between the passband and stopband of the digital filter. It is also called the transition band.
- **Frequency magnitude response:** describes in frequency domain how a filter interacts with the input signal. It is a curve that represents filter attenuation (in dB) versus frequency.

### 2.6.3 *Types of FIR filters*

As mentioned earlier, the primary purpose of a filter is to differentiate between different frequency bands, so the most common filter classification method is frequency selectivity. Based on frequency selectivity, filters are classified as low-pass, high-pass, band-pass, and band-stop [21]. Figure 2-10 shows the frequency response of these four types of filters.

#### 2.6.3.1 Low-pass filter

A low-pass filter passes signals with low frequency than the cut-off frequency and attenuates (reduces amplitude); all signals have the frequency above the cut-off frequency. How much a signal attenuates varies from filter to filter. There are different applications of low-pass filters: electronic circuits (for example, hiss filter that is used in audio), image blurring, acoustic barriers, anti-aliasing filters that condition signals before analog-to-digital conversion, and digital filters that smooth sets of data. An ideal filter does not have a transition band, whereas there is a transition band for practical filters between passband and stopband.



**Figure 2-10: Filter types**

### 2.6.3.2 High-pass filter

A high-pass filter passes signals that have a frequency above the cut-off frequency and attenuates (reduces amplitude) all signals that have the frequency below the cut-off frequency. Its amplitude response increases with the frequency above the cut-off frequency.

#### 2.6.3.3 Band-pass filter

A band-pass filter passes signals that have frequencies within a specific range and attenuates (reduces amplitude) all other frequencies outside that range.

#### 2.6.3.4 Band-stop filter

A band-stop filter does not pass any signal that has frequencies within a specific range and passes all other frequencies outside that range.

### 2.6.4 Z-transform

The Z-transform is derived from the Fourier discrete time-domain transformation, and it is a necessary operation in the digital filter design process. It is performed upon discrete-time signals, which convert it into frequency-domain representation, which is very useful for analyzing discrete-time signals and systems. The Z-transform defined as shown in equation 2-1 [22].

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \quad 2-1$$

where,

$z$  = complex number

### 2.6.5 Transform function of discrete-time systems

The Z-transform is used to find the transfer function of linear discrete-time systems necessary for zeros and poles in the  $z$ -plane. The transfer function of the discrete-time system is defined in equation 2-2 [22].

$$H(z) = \frac{\sum_{i=0}^{M-1} b_i z^{-i}}{\sum_{j=0}^{N-1} a_j z^{-j}} = H_0 \frac{\prod_{i=0}^{M-1} (1 - q_i z^{-i})}{\prod_{j=0}^{N-1} (1 - p_j z^{-j})} \quad 2-2$$

where,

$b_i$  = feedforward filter coefficients (non-recursive part)

$a_j$  = feedback filter coefficients (recursive part)

$H_0$  = constant

$q_i$  = zeros of transfer function

$p_j$  = poles of transfer function

The recursive part of transfer function is a feedback of discrete-time system. FIR filters do not have recursive part of the transfer function, so equation 2-3 shows the simplified form of equation 2-2.

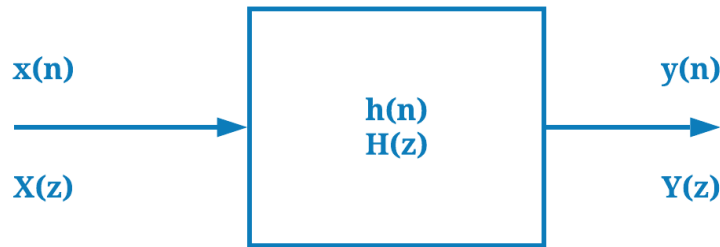
$$H(z) = \sum_{i=0}^{M-1} b_i z^{-i} = H_0 \prod_{i=0}^{M-1} (1 - q_i z^{-i}) \quad 2-3$$

The inverse Z-transform of the transfer function gives the impulse response of a discrete-time system. In other words, the discrete-time system's transfer function is the Z-transform of the impulse response, as shown in equation 2-4.

$$H(z) = \sum_{n=-\infty}^{\infty} h(n) z^{-n} \quad 2-4$$

where,

$h(n)$  = impulse response of discrete-time system.



**Figure 2-11: Block diagram of a linear discrete-time system**



Another representation in the time-domain of the discrete-time system shown in Figure 2-11 is the convolution of the input signal  $x(n)$  with the system's impulse response  $h(n)$ . Equation 2-5 shows the time-domain representation of the discrete-time system. This representation of a discrete-time system is very suitable for software implementation.

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \quad 2-5$$

On the other hand, in the frequency domain, the discrete-time system shown in Figure 2-11 is the Z-transformed input signal  $X(z)$  with the transfer function  $H(z)$  the system. Equation 2-6 shows the frequency domain representation of the discrete-time system.

$$Y(z) = X(z)H(z) \quad 2-6$$

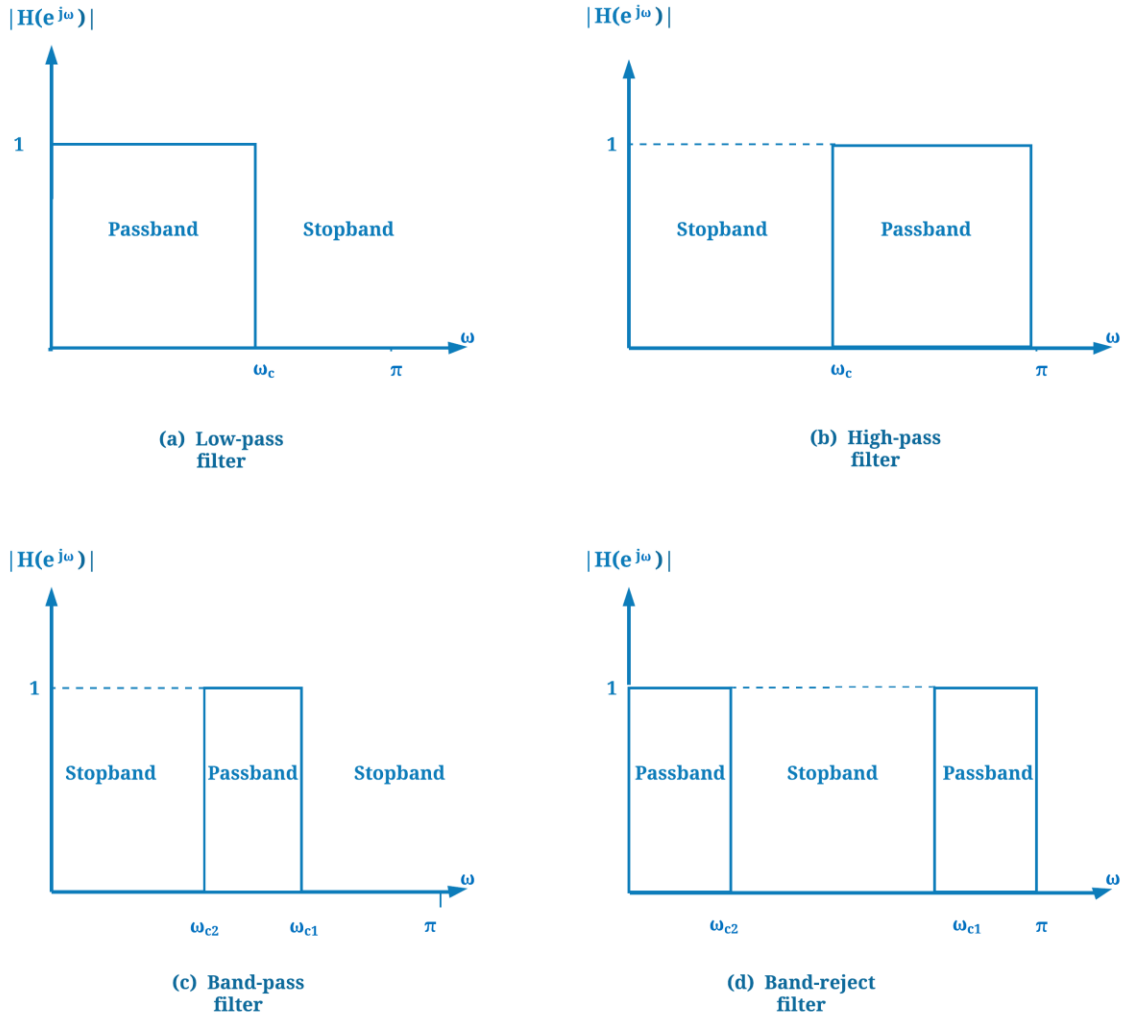
Rearrange Equation 2-6 to get transfer function  $H(z)$ , as shown in equation 2-7.

$$H(z) = \frac{Y(z)}{X(z)} \quad 2-7$$

This representation of the discrete-time system is suitable for hardware implementation, analysis, and synthesis.

### **2.6.6 Ideal filter approximation**

When the FIR filter is designed using the window function, an ideal frequency response must compute the ideal filter samples. As the FIR filter has a finite impulse response, so ideal filter frequency sampling has a finite number of points. As we know, the frequency response of an ideal filter is infinite, so the chances of sampling errors are high if the filter order is small. The sampling errors decrease with an increase in filter order. Figure 2-12 shows the transfer function of the four standard ideal filters [22].



**Figure 2-12: Transfer functions of four standard ideal filters**

Inverse Fourier transform is used to calculate the ideal frequency response. Table 2-1 shows the standard ideal filter frequency responses.

where,

$n$  = variable ranges between 0 and  $N$

$N$  = filter order

$N+1$  = number of ideal frequency response samples

$M=N/2$

**Table 2-1: Frequency response of four standard ideal filters**

Type of filter	Frequency response $h_d[n]$
Low-pass filter	$h_d[n] = \begin{cases} \frac{\sin[\omega_c(n-M)]}{\pi(n-M)} & ; n \neq M \\ \frac{\omega_c}{\pi} & ; n = M \end{cases}$
High-pass filter	$h_d[n] = \begin{cases} 1 - \frac{\omega_c}{\pi} & ; n \neq M \\ -\frac{\sin(\omega_c(n-M))}{\pi(n-M)} & ; n = M \end{cases}$
Band-pass filter	$h_d[n] = \begin{cases} \frac{\sin[\omega_{c2}(n-M)]}{\pi(n-M)} - \frac{\sin[\omega_{c1}(n-M)]}{\pi(n-M)} & ; n \neq M \\ \frac{\omega_{c2} - \omega_{c1}}{\pi} & ; n = M \end{cases}$
Band-stop filter	$h_d[n] = \begin{cases} \frac{\sin[\omega_{c1}(n-M)]}{\pi(n-M)} - \frac{\sin[\omega_{c2}(n-M)]}{\pi(n-M)} & ; n \neq M \\ 1 - \frac{\omega_{c2} - \omega_{c1}}{\pi} & ; n = M \end{cases}$

If filter order  $N$  is even, the constant  $M$  is an integer, but this is not the case with odd-order filters. If  $M$  is an integer (even filter order), the ideal filter frequency response becomes symmetric around its  $M^{\text{th}}$  sample found via expression shown in Table 2-1 [22]. The ideal filter frequency response remains symmetric even if  $M$  is not an integer, but not around any frequency response sample. Equation 2-8 shows the expression to calculate the frequency response of a non-standard ideal filter for inverse Fourier transform.

$$h_d[n] = \frac{1}{\pi} \int_0^\pi e^{j\omega(n-M)} d\omega \quad 2-8$$

## 2.7 Window functions

The window function is a popular method for FIR filter design due to its simplicity. A window, which is a finite array, consists of coefficients selected to satisfy desired requirements. It is necessary to specify the following points when designing a digital FIR filter using a window function:

- It uses a window function.
- The filter's order according to specifications (stopband attenuation, selectivity).

The above requirements are interrelated. Each function uses the following two requirements to choose a filter based on specification:

- High selectivity means a narrow transition region.
- High suppression of undesirable signals means high stopband attenuation.

Table 2-2 shows all window functions mentioned in this thesis and briefly compares their stopband attenuation and selectivity [23].

As shown in Table 2-2, the minimum attenuation of window function and the filter designed using that function are not the same. This difference is due to additional attenuation added during the filter design process that uses the window function. Due to this, stopband attenuation increases, which is desirable.

This method's drawback is that it has fixed minimum stopband attenuation for each function except the Kaiser window. These windows have fixed stopband attenuation, so the only way to affect the transition region is by increasing the filter order. So, stopband attenuation is used to select appropriate window functions for the design

process. A window function with the least attenuation and fulfills the given requirement is always preferred, which gives a narrow transition region to a designed filter. The next step is to compute the filter order. It uses normalized cut-off frequencies of the transition region.

**Table 2-2: Comparison of window functions**

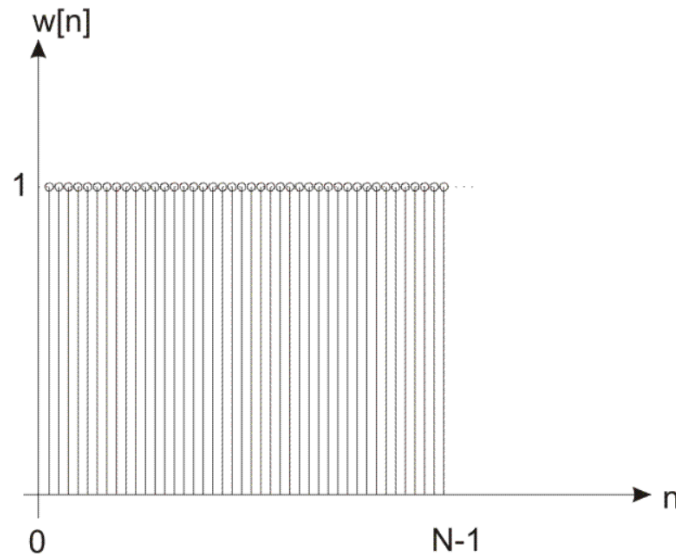
<b>Window Function</b>	<b>Normalized Transition width [Hz]</b>	<b>Passband Ripple [dB]</b>	<b>Minimum stopband attenuation of the window function</b>	<b>Minimum stopband attenuation of the designed filter</b>
Rectangular	$0.9/N$	0.7416	13 dB	21 dB
Hanning	$3.1/N$	0.0546	31 dB	44 dB
Hamming	$3.3/N$	0.0194	41 dB	53 dB
Blackman	$5.5/N$	0.0017	58 dB	75 dB
Kaiser	$4.32/N(\beta=6.76)$	0.00275		70
	$5.71/N(\beta=8.96)$	0.000275		90
	$2.93/N(\beta=4.54)$	0.0274		50

### **2.7.1 Rectangular window**

The rectangular window has low stopband attenuation, which makes it less attractive for most of the filters. Finding rectangular window coefficients is very easy; all coefficients between 0 and (N-1) (N-filter order) equals 1. Equation 2-9 shows the mathematical representation of the rectangular window.

$$w[n] = 1; \quad 0 \leq n \leq N - 1 \quad 2-9$$

The rectangular window only selects  $N$  samples from an input sequence, but it does not perform sample scaling. Figure 2-13 shows a rectangular window's coefficients in time-domain:



**Figure 2-13: Rectangular window in the time domain[23]**

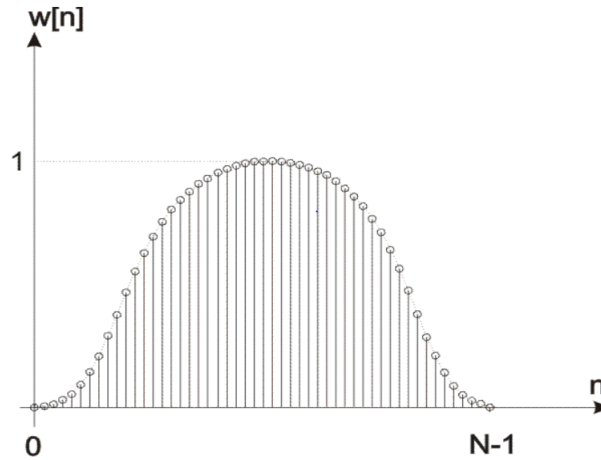
It is not a preferable window for digital filter design due to its reduced stopband attenuation. The reason for its reduced attenuation is the cut-off samples within a window. All sampled frequencies up to a zero sample (from which sampling starts) are equal to zero. The first sample suddenly jumped to a non-zero value, which produces relatively sharp high-frequency components and reduces the stopband attenuation.

Its attenuation increases as the cut-off sample's sharpness decreases, which reduces filter selectivity, which means a wide transition region. The digital filter has predefined requirements, and as the rectangular window has low selectivity, so to get a narrow transition region, the only way is to increase the filter order. The transition region is inversely proportional to the filter order  $N$ , so as the filter order increases, the transition region decreases. As the filter order increases, the filter's complexity also increases,

which require more time to process samples. So, it is essential to choose a window function and filter order carefully [23].

### 2.7.2 Hanning window

The Hanning window minimizes the adverse effects on the final samples' frequency characteristics of the filtered signal. The stopband attenuation of the posterior lobes relatively increases sharply. This window has higher attenuation than a rectangular window.



**Figure 2-14: Hanning window in the time domain [23]**

Figure 2-14 shows the Hanning window's coefficients in the time-domain, and Equation 2-10 shows the Hanning window's mathematical representation.

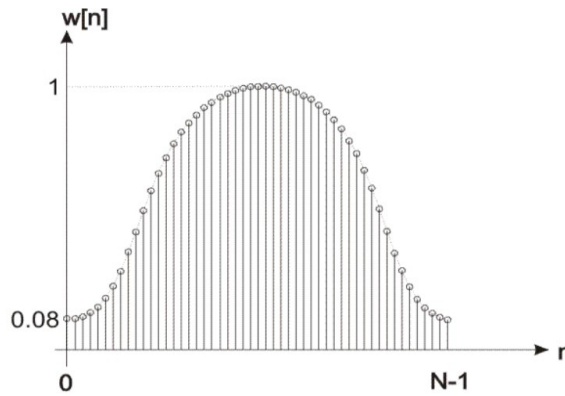
$$w[n] = \frac{1}{2} \left[ 1 - \cos \left( \frac{2\pi n}{N-1} \right) \right] ; \quad 0 \leq n \leq N-1 \quad \text{2-10}$$

### 2.7.3 Hamming window

The Hamming window is the most popular and commonly used. The filter that uses a Hamming window has 53dB minimum stopband attenuation, suitable for most

digital filter implementation. It has a wide transition region as compare to the Hanning and also has high stopband attenuation. With an increase in filter order, the transition region narrows, whereas there is no effect on stopband attenuation. Figure 2-15 shows the coefficients of the Hamming window in the time-domain. Equation 2-11 shows the mathematical representation of the Hamming window [23].

$$w[n] = 0.54 - 0.46 \left[ 1 - \cos \left( \frac{2\pi n}{N-1} \right) \right] ; \quad 0 \leq n \leq N-1 \quad 2-11$$



**Figure 2-15: Hamming window in the time domain [23]**

#### **2.7.4 Blackman window**

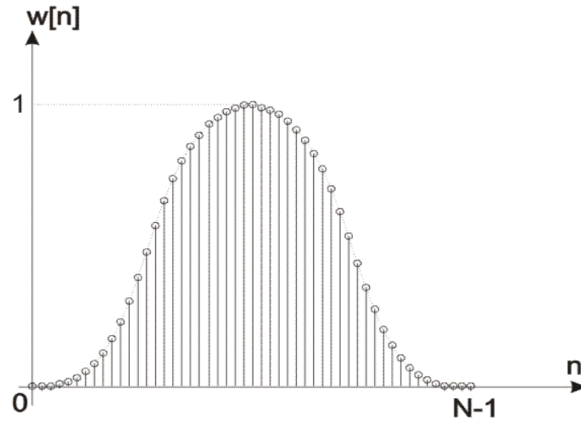
The Blackman window is another popular and commonly used window. It is very convenient for many applications due to its high attenuation and has minimum stopband attenuation of 75dB of a designed filter. Figure 2-16 shows the coefficients of the Blackman window in the time-domain. Equation 2-12 shows the mathematical representation of the Blackman window.

$$w[n] = 0.42 - 0.5 \cos \left[ \frac{2\pi n}{N-1} \right] + 0.08 \cos \left[ \frac{4\pi n}{N-1} \right] ; \quad 0 \leq n \leq N-1 \quad 2-12$$

The Blackman window has an almost similar frequency response as a Hanning window. The only difference is that it has a wide main lobe, and its first side lobe's



attenuation is 51dB. There is additional stopband attenuation due to the side lobes following the first one [23].



**Figure 2-16: Blackman window in the time domain [23]**

### 2.7.5 Kaiser window

There is always a compromise between a narrow transition region (high selectivity) and high stopband attenuation in all the windows described above, so that means these windows are not optimal. The optimal window is described as a function with maximum attenuation per the main lobe's given width. It is also called the Kaiser window. Equation 2-13 shows the mathematical representation of Kaiser window coefficients.

$$w[n] = \frac{I_0 \left[ \beta \cdot \sqrt{1 - \left( \frac{n - \alpha}{\alpha} \right)^2} \right]}{I_0(\beta)} ; \quad 0 \leq n \leq N - 1 \quad 2-13$$

where,

$$\alpha = \frac{N - 1}{2} \quad 2-14$$

$$N = \frac{A - 8}{4.57 \Delta\omega} + 1 \quad 2-15$$

$\beta$  = shape parameter

where,

$A$  = minimum required stopband attenuation

$\Delta\omega$  = width of the desired normalized transition region

Multiply by 2 to get the order of band-pass and band-stop filters. Table 2-3 below provides the value of  $\beta$ .

**Table 2-3: Values of parameter  $\beta$**

<b>A</b>	<b><math>\beta</math></b>
$A < 21$	0
$21 \leq A \leq 50$	$0.5842(A - 21)^{0.4} + 0.07886(A - 21)$
$A > 50$	$0.1102(A - 8.7)$

$I_0$  = modified zero-order Bessel function, which approximated as shown in equation 2-16:

$$I_0(x) = 1 + \lim_{K \rightarrow \infty} \sum_{k=1}^K \left( \frac{x^2}{4} \right)^k (k!)^{-2} \quad 2-16$$

Choose a decent value for K for accuracy. K=20 works for most of the cases.

From all the expressions mentioned above, it is clear that to design an optimal Kaiser filter, the knowledge of normalized transition region's width and desirable minimum stopband attenuation is a must [23].

## 2.8 FIR filter design using window functions

The process to design an FIR filter using window function involves steps [22]:

- Define the filter specification.
- According to the filter specification, specify the window function.
- Compute the filter order required for the given specifications.
- Compute window function coefficients.
- Compute ideal filter coefficients according to filter order.
- Compute FIR filter coefficients according to the obtained window function and ideal filter coefficients.
- If the transition region is too wide or too narrow of the resulted filter, then change the filter order, and steps 4,5, and 6 iterated as needed.

The filter specifications define the desired transition width, normalized frequencies ( $\omega_c, \omega_{c1}, \omega_{c2}$ ) and stopband attenuation. The filter order and window function computed based on these specifications. The window function is selected such that it satisfies the given specifications. After selecting the window function, the filter's order is computed according to the given set of specifications. After these steps, window function coefficients  $w[n]$  is computed using a formula based on the window function selected. The next step is to find frequency samples of the ideal filter using formulas explained in section 2.6.6. This step gives coefficients  $h_d[n]$ . The  $w[n]$  and  $h_d[n]$  have an equal number of elements. And then designed filter's frequency response  $h[n]$  computed using equation 2-17.

$$h[n] = w[n].h_d[n] \quad 2-17$$

The final step is to compute the designed filter's transfer function by transforming

impulse response via Fourier transform, as shown in equation 2-18.

$$H(e^{j\omega}) = \sum_{n=0}^N h[n] \cdot e^{-jn\omega} \quad 2-18$$

Alternatively, via Z-transform, as shown in equation 2-19.

$$H(z) = \sum_{n=0}^N h[n]z^{-n} \quad 2-19$$

Use the following steps if the designed filter has a wide transition region than required:

- Increase filter order.
- Recompute coefficients of the window function.
- Recompute frequency samples of the ideal filter.
- Multiply them to get the desired filter's frequency response.
- Recompute transfer function.

Follow the below steps if the designed filter has a narrow transition region than required:

- Decrease filter order to optimize hardware and software resources.
- Recompute coefficients of the window function.
- Recompute frequency samples of the ideal filter.
- Multiply them to get the desired filter's frequency response.
- Recompute transfer function.

## 2.9 FIR filter realization

Equation 2-20 shows the FIR filter's transfer function.

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{n=0}^{N-1} h[n].z^{-n} \quad 2-20$$

Equation 2-21 used to compute the FIR filter's output samples:

$$y[n] = \sum_{k=0}^{N-1} h[k].x[n-k] \quad 2-21$$

where,

$x[k]$  = FIR filter's input samples

$h[k]$  = the coefficients of FIR filter frequency response

$y[n]$  = FIR filter's output samples

FIR filter realization is of the following types:

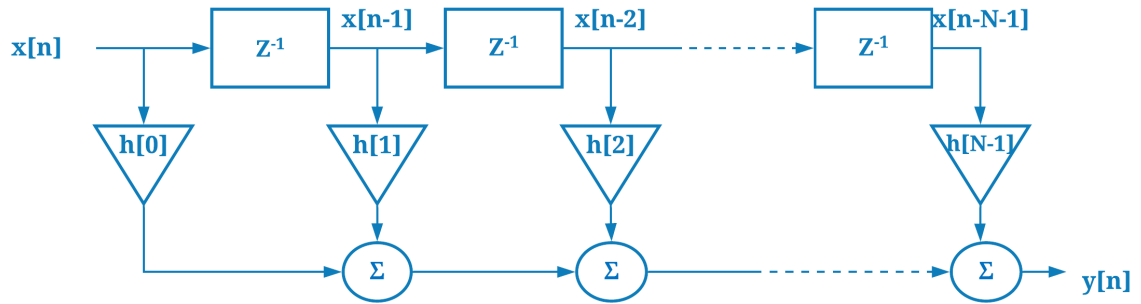
- Direct
- Direct transpose
- Cascade
- Optimized

For hardware implementation, direct, direct transpose, and cascade realization are convenient. However, for software implementation, direct and optimized are good. This thesis explained only direct and direct transpose form.

### 2.9.1 Direct realization

In block diagram representation, the real filter coefficients appear as multipliers in a digital filter's direct structures. Direct realization directly implements equations 2-22.

$$y[n] = \sum_{k=0}^{N-1} h[k].x[n - k] \quad 2-22$$

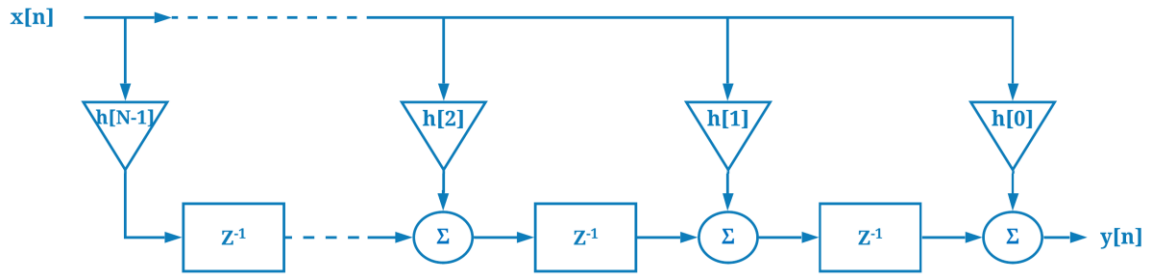


**Figure 2-17: Block diagram of the direct form of FIR filter**

The transversal filter is another name of direct realization. Based on the above expression, to produce an output point, it needs the current sample along with  $N - 1$  previous samples. Based on Figure 2-17, which shows a block diagram of direct realization, it needs  $N$  multipliers for  $(N - 1)^{th}$  order FIR filter [22].

### 2.9.2 Direct transpose realization

In many ways, direct transpose realization is similar to direct realization. Both structures use the same number of delay elements, the same number of multipliers, and the same coefficients. Figure 2-18 shows the block diagram of the direct transpose form of the FIR filter. This thesis uses the direct transpose FIR filter realization.



**Figure 2-18: Block diagram of the direct-transpose form of FIR filter**

## 2.10 Graphical user interface

With GUI's help, the user can easily communicate with Python code, ModelSim, and MATLAB. GUI design instructions are easy to use by any user, even without the knowledge of Python, ModelSim, or MATLAB coding. In GUI, input and data modification is easy and convenient, and it has a fast and intuitive output. The advantages of using GUI for FIR filter design are as follow:

- Reconfiguration is easy.
- Length and variable width can adjust at design-time easily.
- Coefficients are adjustable at run time.
- It generally provides the user with immediate visual feedback about the effect of each action.

## 2.11 Hardware description language

The interconnected transistors are the basic blocks of any digital circuit. Digital circuits can easily design and analyze with a hierarchical structure, which uses interconnected diagrams to represent it. This approach has a limitation that it is impractical to use for large circuits. Another approach to describe these circuits is a textual language used to clearly and concisely capture the digital design's defined

features. These languages are called hardware description language (HDL) [24].

Hardware description language, as the name tells, is used to describe a circuit layout or hardware application.

HDL used for this system is Verilog, which can describe electronic circuits and systems in textual format. It can be used for verification through simulation, provides timing analysis, and can also use for logic synthesis. The Verilog HDL is an IEEE standard. With the help of HDL, the circuit's representation uses words and symbols; then, development software converts textual description into configuration data that loads into FPGA to implement the desired functionality [24].

#### ***2.11.1 Importance of HDLs***

HDLs have the following advantages as compared to traditional schematic-based design [25]:

- By using HDLs, the design is described at a very abstract level. Designers do not need to choose specific fabrication technology to write RTL description. Design can automatically convert to any fabrication technology by using logic synthesis. There is no need to redesign a circuit if a new technology emerges. Designers simply use RTL description as input to logic synthesis and generate new gate-level netlists using new fabrication technology. The logic synthesis tool automatically optimizes the circuit's area and timing for new technology.
- Functional verification of the design performs early in the design cycle if designers use HDLs. Designers work at the RTL level, so they can optimize and modify the RTL description until the design meets the desired functionality. Most design bugs are eliminated at this point; this reduces the design cycle time



because it reduces the probability of finding a functional bug at a later stage in the gate-level netlist or physical layout.

- It uses a textual description with comments, so it is easy to develop and debug circuits. As compare to gate-level schematics, it provides a concise representation of the design. For complex designs, gate-level schematics are almost incomprehensible.

### ***2.11.2 Basics of Verilog***

The basics of Verilog HDL is as follow [25]:

- Verilog HDL is a general-purpose HDL which is easy to learn and easy to use. Its syntax is similar to the C programming language.
- The same model can mix different abstraction levels to define the hardware model in terms of switches, gates, RTL, or behavioral code. For stimulus and hierarchical design, designers have to learn only one language.
- It is mostly the designer's choice because it is the language that most popular logic synthesis tools support.
- For post logic synthesis simulation, Verilog's libraries are provided by all fabrication vendors. So, it provides a wide choice of vendors for chip designing.
- Using the Programming Language Interface (PLI) feature, the designer can write a custom C code that can interact with internal Verilog's structures. With the help of PLI, it is easy to customize a Verilog HDL simulator according to the designer's need.

## **2.12 Automating the generation**

The tools discussed in section 3.2 are either too complicated or lack ASIC support. Some tools can generate design but not testbench, some can generate design and testbench, but the user must have coding knowledge. Another critical point is the tool availability. Some tools have publications about the FIR filter generators. However, tools do not become available freely or a license required for commercial usage. The companies spend money carefully for a license, mainly if it is just for a simple filter, and for any engineering change requirement, it needs permission or license. So, due to all these reasons, this thesis considered the automation of the FIR filter as a case study. The scripts developed for FIR filter using mainly Python language and a portion of it developed using Perl; these languages have many advantages discussed in the following sections. Python and Perl are used to develop scripts that can automatically generate design and testbench code of FIR filter and simulate it using ModelSim. A brief explanation of both is as follow:

### **2.12.1 Python**

Python is useful for verification speed up. It is a boon for the project that has time and resources constrained. Python is easy to learn, dynamic, object-oriented programming language suited for large and complex projects with changing requirements. There are many reasons to use Python script; the following are the main reason to use it for this thesis [26]:

- Easy to use, read, and flexible.
- No compilation is necessary.
- It has an open-source license.

- Python is mature with much support.
- Language can run on multiple systems (for example, Mac, Linux, Windows) but retain its similar interface. Its design does not change a lot with each operating system.
- Program reusability with already available packages and modules with a standard library.
- Easy to connect to other languages (like C++, Perl).
- Software like MATLAB, ModelSim can run smoothly from its script.

### **2.12.2 Perl**

Perl is a general-purpose high level interpreted and dynamic programming language. Perl supports both procedural and Object-Oriented programming. Perl has a similarity to C syntactical, so it is easy to use. The following are the advantages of Perl [27]:

- Easy to learn
- Text-processing
- Contains features of different languages
- Free and open-source
- Supports open-source modules
- Provides support for cross platforms

### **3. LITERATURE REVIEW**

The literature review mainly focuses its attention on functional verification automation, particularly design and testbench automation. The automation of FIR filter implementation and the graphical user interface were also studied and analyzed.

#### **3.1 Functional verification**

Due to the design's heterogeneous nature, functional complexity increases; for example, co-existing hardware and software, analog and digital. There is a requirement of higher system reliability, which pushes verification tasks to ensure that chip-level functions perform satisfactorily in the system environment, especially if chip-level defects have a multiplicative effect. As complexity continuously increases, new verification languages are introduced to verify complicated designs at various abstraction levels. The new tools and technologies are also created to support these new languages.

A. Molina and O. Cadenas provided a quick survey of functional verification to make it easy to choose the technique for the hardware's design cycle to take full advantage of these tools and techniques for verification projects. It is necessary to decide on them as early as possible. They provided an overview of FV, described bottlenecks of the verification, challenges of FV, and explained current FV technologies and trends. The paper described functional verification as the art of combining hardware, software, and communication skills with creative strategies to understand design and its usage to ensure that its quality and delivery schedule are successful. Many problems associated with today's functional verification methodologies are due to the absence of effective automation and growth in the design's size and complexity. Verification is primarily a manual process. The most notorious design and verification problem is the lack of a

useful metric to measure its progress. These are vital points to help design and verification engineers, verification automation, and comprehensive methodology [9].

### ***3.1.1 Design automation***

Mehdi Dehbashi et al. presented efficient automation of the debugging procedure, which reduces the debugging time and increases the diagnosis accuracy. This procedure used the integrated Boolean Satisfiability (SAT) based debugging with testbench based verification. The diagnosis accuracy increased by iterating debugging and counterexample generation that means the total number of fault candidates decreased. Its experimental data shows that this procedure was accurate as actual formal debugging in 71% of the experiments. This paper proposed three techniques to generate diagnostic traces for high-quality counterexamples to enhance diagnosis accuracy. Local Branch activation (LBA) activates the local branches of each fault candidate. Minimization of Sensitized Path Intersection (MSPI) looks for sensitized paths that include a minimum number of fault candidates. Limited Minimization, followed by Branch Activation (LMBA), combines the advantages of both techniques [28].

### ***3.1.2 Testbench automation***

Srikant Kumar Mohanty, Suchismita Sengupta, and S K Mohapatra proposed a test bench automation solution that verifies the completeness and correctness of data as it passes through interconnect fabric. It automatically creates authenticated infrastructure, stimulus vector, and coverage model to support all exchanges between masters and slaves within a System-On-Chip (SOC), reducing verification efforts. It uses a protocol-independent scoreboard to check data integrity and verify different data path transactions

to and from each bus fabric port. The proposed solution saved 40% in the verification cycle compared to various bus matrix testing [29].

Isaac Maia et al. generated a semi-automatic testbench tool called eTBc (Easy Testbench Creator). Furthermore, a methodology called VeriSC ( which allows testbench simulation before RTL without writing any additional code). These resources are used together in IC-development flow to enhance productivity in verification tasks by automatically generating testbench prototypes. In VeriSC methodology, eTBc is used in all functional verification steps. They created TLN (Transaction Level Netlist) using eDL (eTBc Design Language) language and randomly generated stimuli for testbench. They compared results with using eTBc and without using it and concluded that by using eTBc, the production profit was 83.33% higher than the manual process. So, this tool can speed up the functional verification process. This approach's disadvantage is that the user must know eTL (eTBc Template Language) to use this approach. This tool's next steps to develop a graphical user interface and a specific template for more verification methodologies [30].

M. Lajolo et al. proposed an approach for simulation-based validation that generates input sequences for testbench. If current stimuli poorly exercise a part of the design, then designers develop new stimuli to address that part of the design. Developing new stimuli is a very time consuming and laborious task because the designer must understand all the design details to generate a new input sequence. This paper proposed an automated approach that assists designers in generating a test bench for system-level design. It is also suitable for simulation-based validation environments and focuses on integration rather than replacing current manual simulation. Its results show that this

method increased the quality of the validation process [31].

### **3.1.3 Testcase automation**

Samuel Nascimento et al. describe a tool suite named Veasy, which contains four modules to perform linting, simulation, coverage collection, and test case generation, which are vital challenges of functional verification. A Graphical User Interface is used to integrate all modules. This tool is used for test case automation based on layers, capable of generating complex scenarios using drag and drop operations. The tool's capability and performance are compared with commercial and academic functional verification tools, which shows it takes less overall simulation time than other commercially available tools and algorithms used in this tool capable of coverage collection with lower simulation overhead [32].

## **3.2 FIR filter automation**

Verma and Chien developed a generator, which generates efficient decimating filters. The optimized logic is such that there is no need to calculate those values ignored by the decimator. It uses canonical signed digit (CSD) format to save silicon area [33].

Many FPGA projects use FIR generators by Xilinx that create a design that is a good match for the FPGA platform, but the resulting HDL is very difficult to read and understand. Xilinx Compiler v6.3 supports interpolators, decimators, half-band filters, and resource sharing. It generates distributed arithmetic filters, so it supports multiple coefficient banks [34]. Xilinx VHDL is encrypted, so it is difficult to get ideas for implementation.

HDL Coder generates synthesizable Verilog and VHDL code from MATLAB

functions, Simulink models, and State flow charts. The code is suitable for FPGA programming, ASIC prototyping, and design. It can develop and verify FIR filter designs at a high-level of abstraction and automatically generate a synthesizable RTL code that targets FPGA, ASIC, or SOC devices. However, they are quite complicated for simple filters [35].

Bogdan Sbarcea and Dan Nicula developed a tool called Sim2HDL, which automatically translate Simulink models into a hardware description language and drastically reduce the project time. It generates VHDL or Verilog language at a behavioral level description, and it can implement in FPGA using commercial synthesizers. It uses a limited set of Simulink blocks from the original Simulink libraries, and it offers support for Altera and System Generator and various ASIC technologies. It also offers support for MATLAB variables from the workspace. It does not have any bus limitations, unlike Altera [36].

As a part of the Master's thesis, Kevin Camera developed a tool called SF2VHD, which translates state machines in the Stateflow graphical language into VHDL capable of synthesizing into hardware. Stateflow is a vibrant graphical and textual language-based strictly on the original StateCharts language defined by Harel. First, Stateflow data types are converted to bit-accurate VHDL data types to generate a VHDL code. Then the Stateflow expression syntax and operators converted to VHDL equivalents on a line-by-line basis to implement the state machine's functional behavior [37].



### **3.3 GUI for FIR filter**

Myagmardorj Bayasgalan and Xiang-E Sun developed a graphical user interface for the FIR filter. The GUI has used MATLAB's computing power and GUI platform to complete the GUI design and implementation of the FIR digital filter through programming. The GUI achieved the program to design and implement various amplitude characteristics of the FIR digital filter. By setting global shortcut keys, the user, after all, parameters and options, confirmed that it achieved the filter design's rapid implementation [38].

Zhang XueMin introduced a method to simulate the FIR filter-based GUI. This method depends on the MATLAB code and uses controls to generate GUI. The simulation results confirmed that the design based on GUI is convenient, fast, intuitive, and flexible. This method used MATLAB for simulation [39].

Rosa et al. provide a complete optimized FIR filter's design flow from the transfer function to synthesizable VHDL. The generator uses common subexpression sharing (CSE). The tool also modifies the coefficients to make the hardware implementation simple while maintaining the filter's desired design constraints [40].

## 4. PROPOSED SYSTEM

Automation through the graphical user interface (GUI) is the main feature of the Easy-filter. Using GUI, the user can design and implement the FIR filter, generate Verilog code, simulate, and validate generated code. The examples of such operations are explained in chapter 5. Easy-filter uses ModelSim for simulation, uses MATLAB for comparison and validation, and produces Verilog (design and testbench of FIR filter) and MATLAB code as output. The methodology allowed automation (of design and testbench) using Python and Perl language. This chapter explains the design, verification, and validation methodologies.

### 4.1 Design methodology

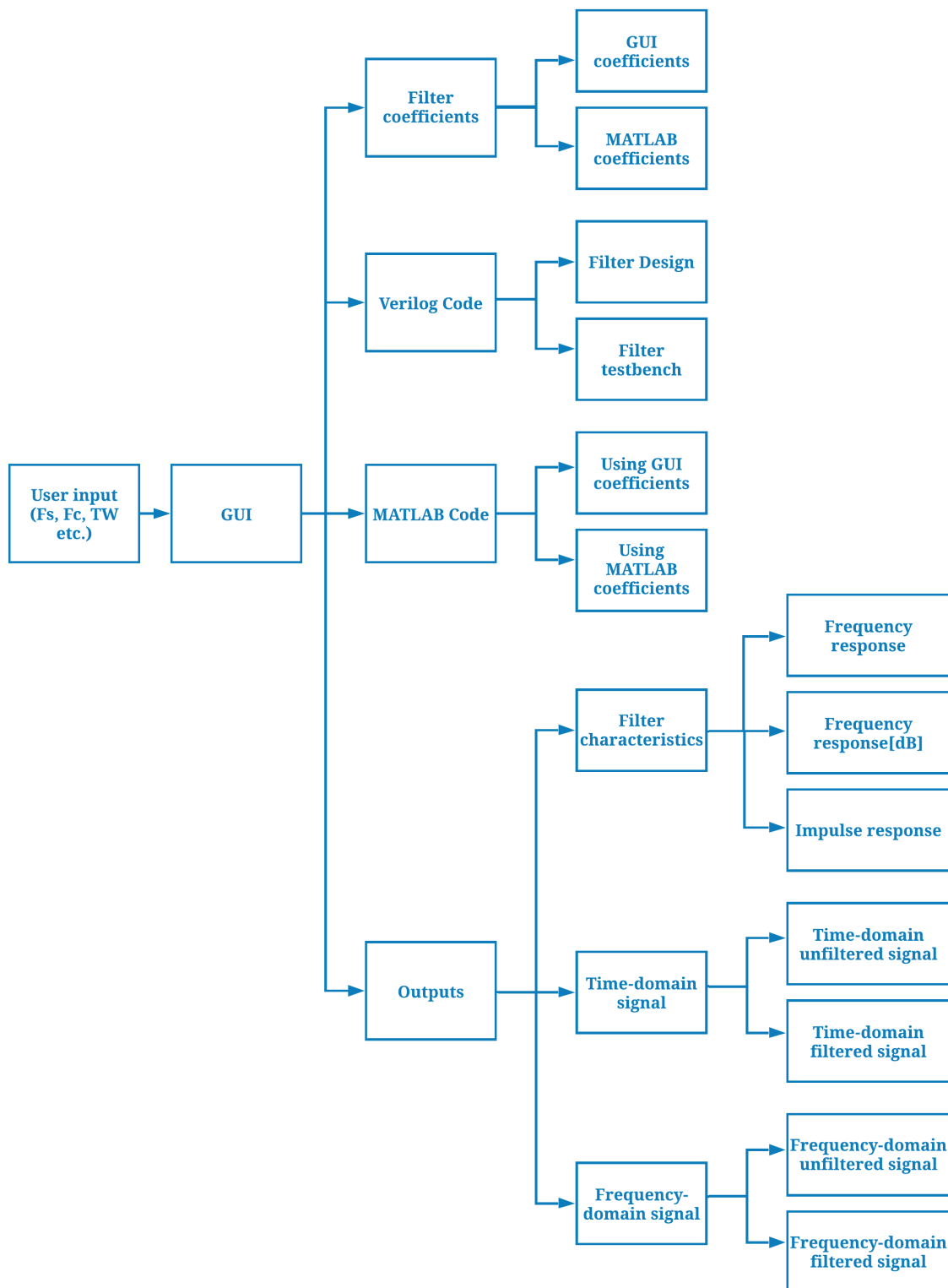
Easy-filter uses Python and Perl languages to automatically generates the following:

- Filter coefficients
  - Python generated coefficients
  - MATLAB generated coefficients
- Verilog code
  - Filter design code
  - Filter testbench code
- MATLAB code
  - Filter code using Python generated coefficients
  - Filter code that uses MATLAB generated coefficients
- Waveform Outputs
  - Filter characteristics

- Impulse response
- Frequency response
- Frequency response in dB
- Time-domain representation of input signal and filtered signal
- The frequency-domain representation of the input signal and filtered signal

The user provides inputs through GUI based on the required filter's specifications.

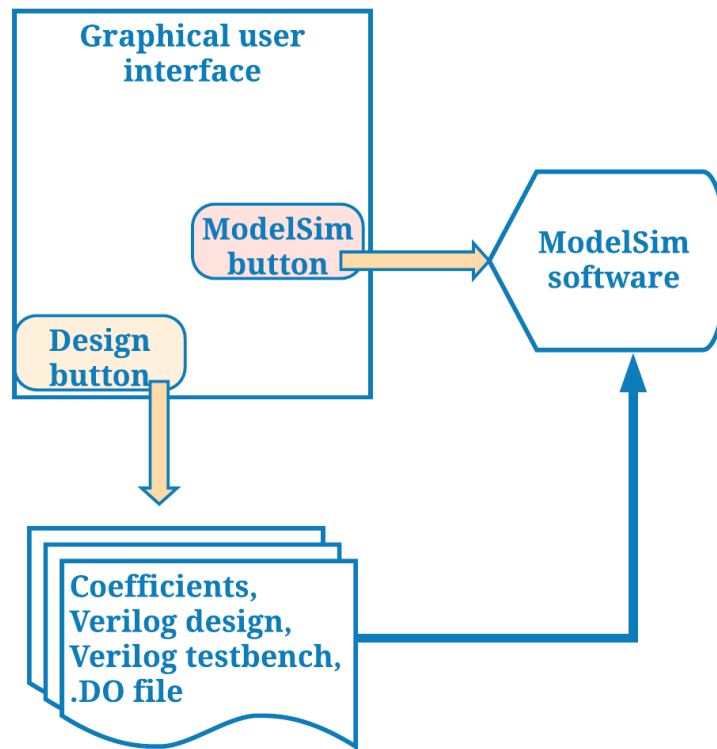
The scripts written in Python and Perl use this input information to generate filter coefficients, Verilog design, and testbench code; and generate MATLAB codes. GUI uses different scripts to generate outputs. A Python script generates the FIR filter's coefficients. Perl script reads these coefficients and converts them into floating-point binary; this floating-point binary representation of coefficients used by another Perl script to generate Verilog design and testbench. Another Python script uses this information to generate impulse response, frequency responses, and filter status. A different Python script uses user input to generate MATLAB codes. One MATLAB code uses the coefficients generated by the proposed system's Python code, and another code uses MATLAB generated coefficients. One Python script connects GUI to MATLAB and shows the unfiltered and filtered output in time-domain and frequency-domain. The user can select between MATLAB and Python generated coefficients to filter the unfiltered signal. The user can also choose four frequencies for the input signal (unfiltered). Figure 4-1 shows the block diagram of filter design automation.



**Figure 4-1: Block diagram of filter design automation**

## 4.2 Verification methodology

Easy-filter uses GUI generated Verilog design and testbench code for filter's verification. The GUI's *Modelsim* button uses to launch ModelSim software. Figure 4-2 shows the verification methodology.



**Figure 4-2: The verification methodology**

A Python script uses user input and another script's generated coefficients to generate .do file and automatically opens and runs this .do file. This .do file is a script that can execute many commands at once. To create .do file, simply type a set of commands in a text file. In this thesis, we created a .do file that loads a design, adds signals to the wave window, provides stimulus to those signals, and runs the simulation. Figure 4-3 shows an example of .do file.

```

vlib work
vmap work ./work
vlog fir_20tap.v
vsim -gui work.tb_fir
add wave -position insertpoint \
sim:/tb_fir/C1k \
sim:/tb_fir/Xin \
sim:/tb_fir/Yout
run 10000 ns

```

Figure 4-3: An example of .do file

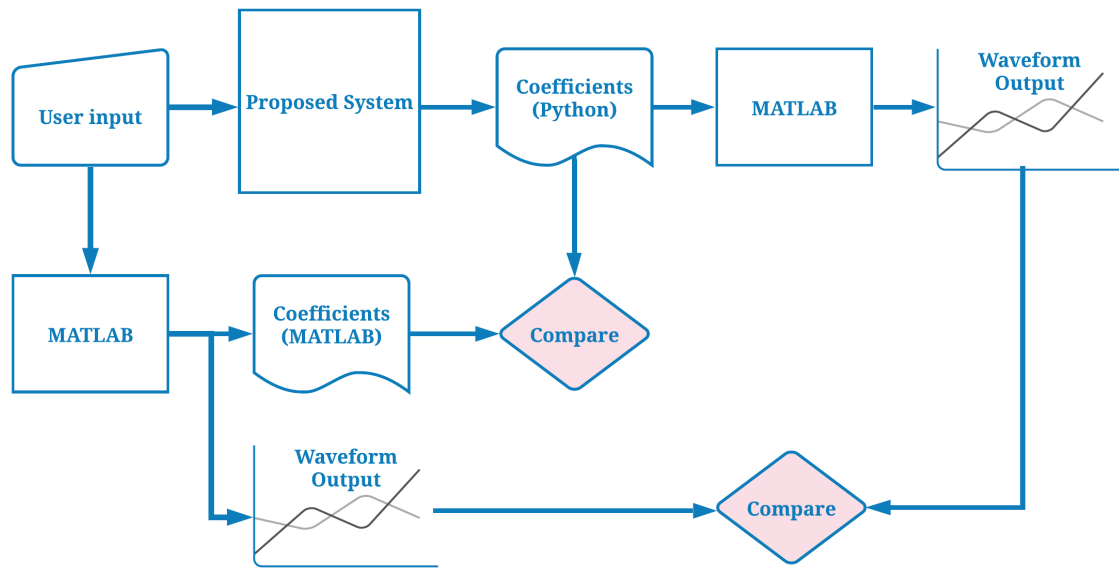
ModelSim software provides a visual representation of filter coefficients, the input signal that needs to be filtered, and filtered output. It uses the Verilog testbench to simulate the input signal to generate the output signal.

### 4.3 Validation methodology

Easy-filter also automatically validates the designed filter by using well-established MATLAB software. For proposed system validation, we provide the same user input to MATLAB software as well. For validation, we use the following GUI's buttons:

- *Design* : It generates filter coefficients using Python script.
- *Code – Python coeff*: It generates MATLAB code using the coefficients generated by the proposed system's Python script.
- *Code – MATLAB coeff*: It generates MATLAB code that generates filter coefficients by MATLAB software.
- *MATLAB*: This button uses the above MATLAB codes and generates coefficients and filtered outputs.
- *Validate*: This button uses Easy-filter's generated coefficients and MATLAB generated coefficients and compares and produces percentage error.

For validation, we provide the same user input to MATLAB software; it generates filter coefficients and compared them against Python script generated coefficients using *Validate* button. Easy-filter also compares the filtered signal in time-domain and frequency-domain. Easy-filter provides Python generated coefficients and user-input to MATLAB software and generates time-domain and frequency-domain representation of the input signal to compare the filtered signal. And then provides the same user-input to MATLAB software, and it calculates its coefficients and generates time-domain and frequency-domain representation of the input signal. A block diagram of the validation methodology is shown in Figure 4-4.



**Figure 4-4: The validation methodology**

#### 4.4 Experiment

In this thesis, Easy-filter's GUI can create four types of filters: low-pass, high-pass, band-pass, and band-stop. The following experiments were carried out as a part of this thesis.

#### **4.4.1 FIR filter generation**

With the help of GUI, four different types of filters were created. As user inputs vary, filter specifications also vary. The filter's output is checked in the form of the waveforms and checked against the filter status window.

#### **4.4.2 FIR filter verification**

To verify, it uses Python generated Verilog testbench, which simulates the inputs and observes outputs. It is often a critical way to verify whether the design's functionality is correct before feeding it to an FPGA. GUI has a button called *ModelSim*. By clicking this button, the Python script automatically generates a *.do file* and runs it on ModelSim software, which shows all the filter coefficients, input, and output waveforms. The coefficients compare against the text file generated by a Python script. ModelSim uses testbench to generate output waveform, which contains simulation information of the design and plotted against time to provide a graphical representation of the simulation.

#### **4.4.3 FIR filter validation**

For filter validation, the filter coefficients generated by Python script using GUI compare against the filter coefficients generated by MATLAB software for the same specifications. The filtered output waveforms of the generated filters are compared against the MATLAB filtered output.

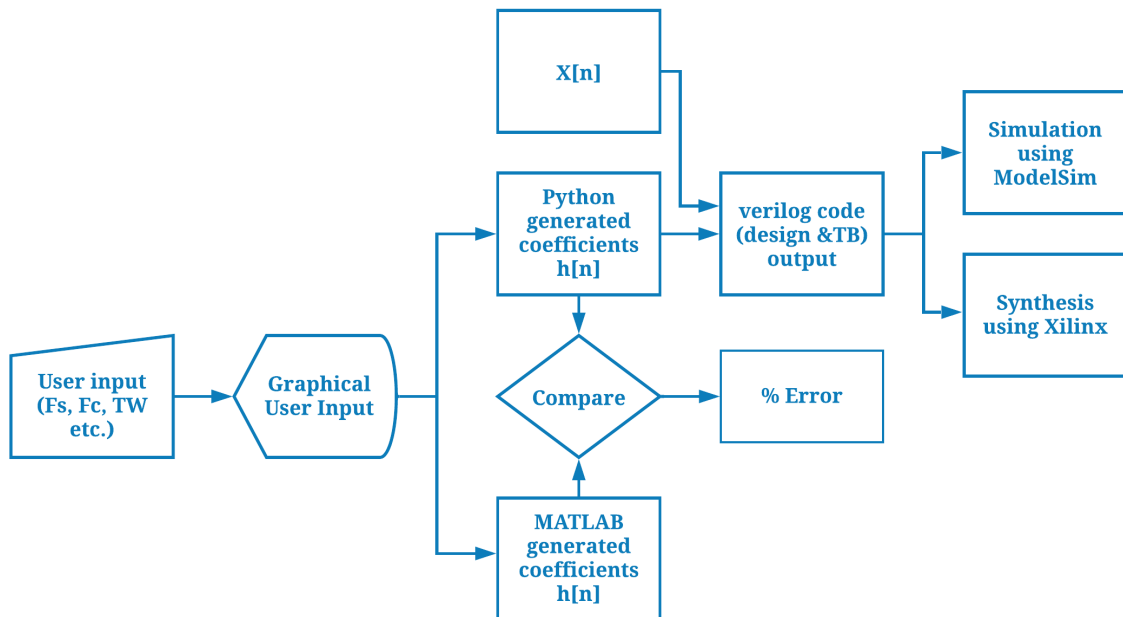


## 5. EXPERIMENTAL SETUP

This chapter explained the experimental setup used in this thesis for the FIR filter's design, verification, and validation automation. Figure 5-1 shows the basic block diagram of the experiment setup.

### 5.1 Experiment set-up

Figure 5-1 provides the complete block diagram of the experiment setup; this section explains each experiment setup step.



**Figure 5-1: Basic block diagram of the experiment setup**

- **User input:** The user provides the information for filter design according to the requirement using the GUI's user input section, as shown in Figure 5-2, to the proposed system.

### Filter Settings

#### Filter Type

☐ Low-Pass
 ☐ High-Pass
 ☒ Band-Pass
 ☐ Band-Reject

#### Window Type

☐ Rectangular (A < 21)
 ☐ Hann (A < 44)
 ☐ Hamming (A < 55)
 ☐ Blackman (A < 75)
 ☒ Kaiser (A < 80)

### Frequency Specifications

Enter Sampling frequency [Hz]:

Enter lower-cutoff frequency ( $f_L$ ) between 200.0 to 9800.0 (Hz):

Enter higher-cutoff frequency ( $f_H$ ) between 5000.0 to 9800.0 (Hz):

Enter Transition bandwidth (TW) between 200.0 to 9800.0 (Hz):

Enter stopband attenuation (A) between 21 to 80 [dB]:

### Input signal frequencies

Enter input signal frequencies between 0.0 to 9999.0 (Hz):

f1 =

f2 =

f3 =

f4 =

### Filter Order

☐ Quantity
 ☒ Quality

N=

Figure 5-2: GUI's user input section

The user provides the following information:

- Select filter type
- Select window type
- Enter sampling frequency
- Enter cutoff frequency
- Enter transition bandwidth
- Enter stopband attenuation for Kaiser window
- Select between quality and quantity
- Enter four frequencies for the input signal that needs to be filtered

The Python script behind Easy-filter's GUI has been written to automatically enable or disable the cut-off frequencies according to the filter selection. The range of cut-off frequencies and transition width varies automatically according to the user's sampling frequency. The stopband attenuation only enables if the user selected the Kaiser window. The transition width disables if the user wants to design a filter with quantity. For quantity, the user has to enter N (required filter order), and the system automatically calculates transition width based on N entered by the user. If the user selected quality, then transition width enables, and the system calculates filter order and the number of coefficients automatically based on the user input.

- **Easy-filter:** The proposed system is a graphical user interface. GUI window has the following windows: filter settings, the filter's impulse response, frequency response, the frequency response in dB, filter's coefficients, Verilog and MATLAB code, ModelSim options, MATLAB options, and filter's status

windows. It has four buttons: *Design*, *Validate*, *MATLAB*, and *Modelsim*. After filter design, the user has options: *Verilog design*, *Verilog TB*, *Code – Python coeff*, and *Code – MATLAB coeff*. Using GUI, as shown in Figure 5-3, users enter all the information according to the required filter's specification.

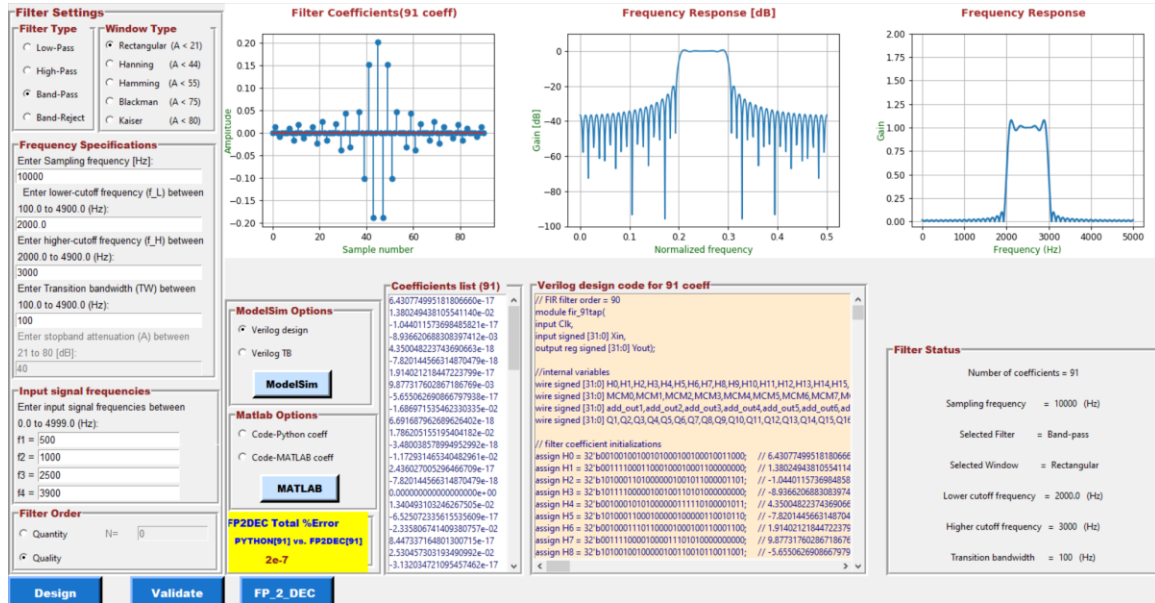


Figure 5-3: The proposed system's graphical user interface

The user follows the following steps to enter the filter's specifications using GUI:

- Select filter type
- Select window type
- Enter sampling frequency
- Enter cut-off frequency (or frequencies for band-pass and band-stop filter) within the specified range
- Enter transition bandwidth within a specified range
- Enter stopband attenuation if Kaiser window is selected

- Choose between quality and quantity
- If quantity selected, then enter the value of  $N$  (filter order)
- Press *Design* button to design filter according to entered filter's specifications
- Select *Verilog* to generate filter's Verilog design code
- Press *Verilog TB* to generate filter's Verilog testbench code
- Press *Modelsim* button to launch the ModelSim simulator
- Select *Code – Python coeff* to generate MATLAB code that uses Python generated coefficients
- Select *Code – MATLAB coeff* to generate MATLAB code that uses MATLAB generated coefficients
- Press *MATLAB* button to launch MATLAB software to generate time-domain and frequency-domain responses for filtered and unfiltered signals
- Press *Validate* button to compare Python generated coefficients against MATLAB generated coefficients and generate a percentage error if there is any difference

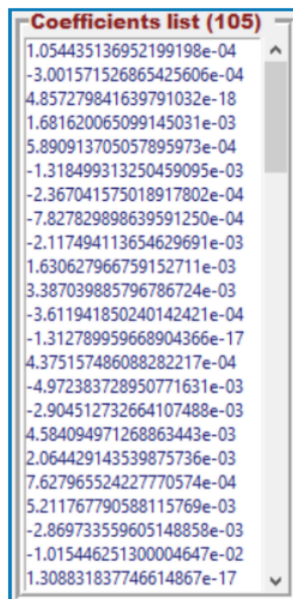
The Python and Perl scripts use all the information provided by the user and generate the following based on the above information:

- Filter coefficients
- Impulse response
- Frequency response in dB
- Verilog design and testbench code
- MATLAB code for Python generated coefficients as well as MATLAB's

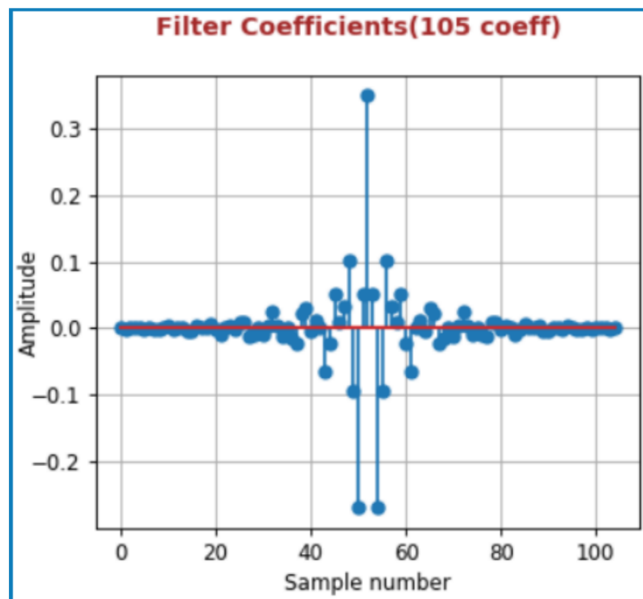
coefficients

- Compare the proposed system's coefficients against MATLAB's coefficients for the same user input and produce a percentage error
- Floating point to a decimal conversion error
- Also, shows filter status

➤ **Coefficients:** The Python script generates filter coefficients based on user input, mainly filter type, window function, and transition width. Perl script converts these coefficients to normalized coefficients and then converts them into binary. As shown in Figure 5-4, GUI shows these coefficients in impulse form and text form. Figure 5-4 (a) shows FIR filter's coefficients in text form, and Figure 5-4(b) shows them in impulse form. On the other hand, MATLAB uses the same user inputs and generates the filter's coefficients.



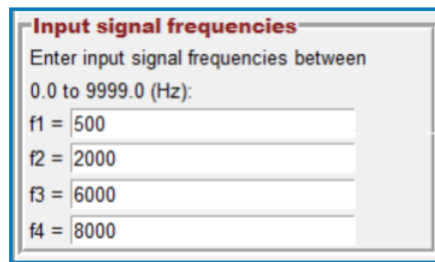
(a) FIR filter's coefficients



(b) FiR filter's impulse response

**Figure 5-4: FIR filter's coefficients window in text and impulse form**

- **X[n]:** Input signal that the user wants to filter. GUI has four options to enter frequencies to combine them to produce an input signal that needs to be filtered. As shown in Figure 5-5, the input signal's frequencies should be within the specified range



The image shows a GUI window titled "Input signal frequencies". Inside the window, there is a text label "Enter input signal frequencies between 0.0 to 9999.0 (Hz):". Below this label, there are four input fields, each preceded by a label: "f1 =", "f2 =", "f3 =", and "f4 =". The values entered in these fields are 500, 2000, 6000, and 8000 respectively.

Frequency Label	Value (Hz)
f1	500
f2	2000
f3	6000
f4	8000

**Figure 5-5: Input signal window**

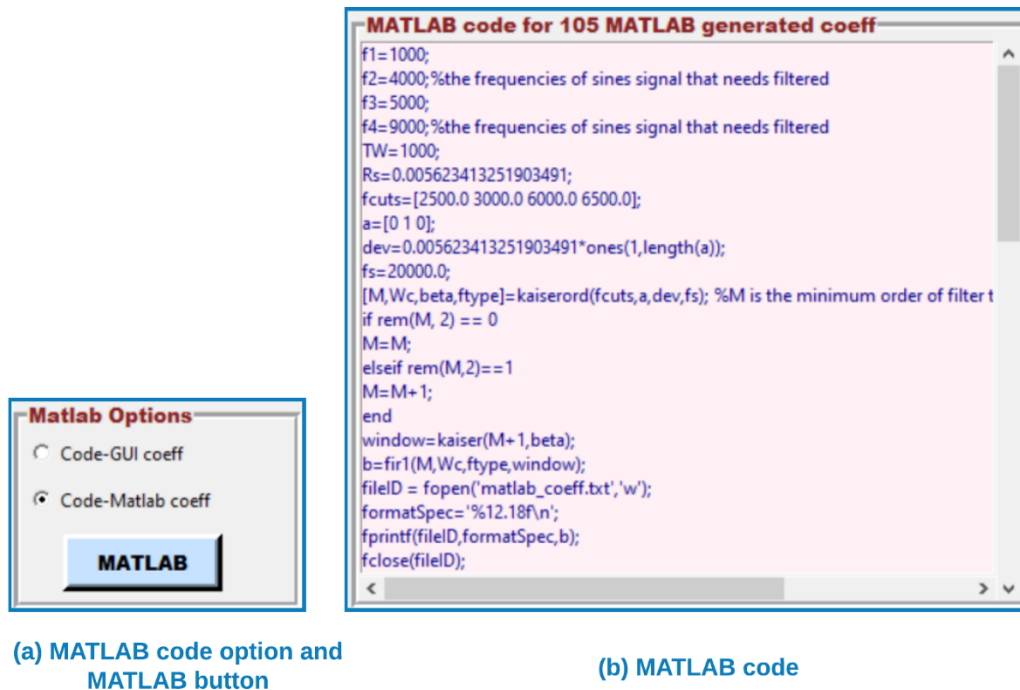
The Python script calculates the range based on sampling frequency. Figure 5-5 shows the GUI's input signal window that the user can use to change the input signal's frequencies.

- **Verilog code:** The proposed system generates the Verilog code of filter design and testbench. After entering all the GUI specifications, when the user selects *Verilog* button, it generates Verilog's design code according to the specifications, and it uses *Verilog TB* to generate the Verilog testbench. *Verilog TB* button uses Perl and Python scripts. Perl script reads the coefficients file to generate Verilog code, and Python script uses user inputs for input frequencies and generates input signal values that Verilog testbench code uses. The user can use *ModelSim* button to simulate a designed filter on ModelSim software and analyze the filter's coefficients. The Perl script generated Verilog design and testbench code simulates the FIR filter coefficients, input, and output signal on ModelSim software. Figure 5-6(a) shows the Verilog code options and *ModelSim* button,

[illegible]

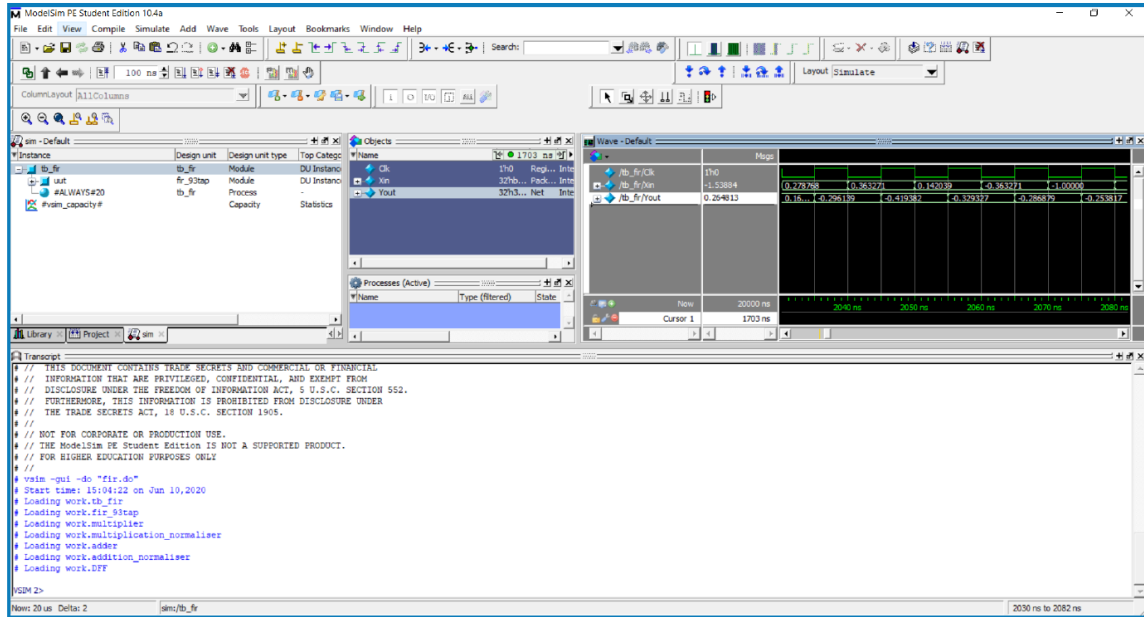
➤ **MATLAB code:** The proposed system can generate two types of MATLAB codes. If the user selects *Code – Python coeff* button, then the Python script automatically uses user input to generate MATLAB code by using Python script-generated coefficients. Furthermore, if the user chooses *Code – MATLAB coeff* option, then the Python script automatically generates MATLAB code with MATLAB calculated coefficients. By pressing *MATLAB* button, it runs the last code option selected by the user using MATLAB software. It generates a time-domain and frequency-domain representation of the input signal (unfiltered) and the output signal(filtered). Figure 5-7(a) shows the options and MATLAB button, and Figure 5-7(b) shows the MATLAB code window.





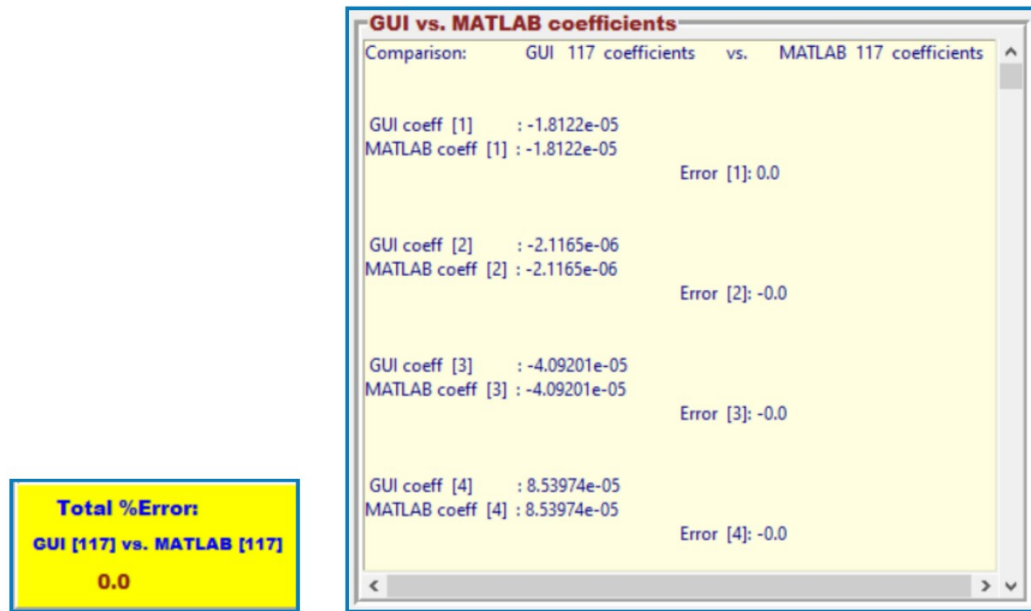
**Figure 5-7: MATLAB options and MATLAB code**

- **Simulation:** GUI has *Modelsim* button to launch the ModelSim simulator. When the user press *Modelsim* button on GUI, it automatically creates *.do file* and runs it on ModelSim. *.do file* is a script file with commands to compile the Hardware Description Language (HDL) files, load the design, give stimulus, and simulate your design in a single *.do file*. So, it saves engineers time. ModelSim generates waveforms that are very good to analyze the filter's signals. Figure 5-8 shows the ModelSim window that automatically opens by clicking the on *Modelsim* button and simulates the FIR filter design.



**Figure 5-8: ModelSim simulation window**

- **Synthesis:** The Verilog code generated by GUI is synthesizable. Xilinx software is used to synthesize this FIR filter design using Artix-7. It converts HDL-based logic of FIR filter design to low-level implementable circuits called Netlist, consisting of a combination of transistors representing digital logic gates. The design's synthesis provides utilization and power report.
- **Validation:** The proposed system's output coefficients and filtered output are compared against well-established MATLAB's coefficients and filtered output for the same inputs. The GUI window shows the filter's coefficients comparison and percentage error, as shown in Figure 5-9 as the user presses *Validate* button. As the user press *Validate* button, the Python script automatically reads GUI generated coefficients and MATLAB generated coefficients, compares them, and generates a total percentage difference and displays it.



(a) Total percentage error      (b) Comparison: GUI versus MATLAB coefficients

**Figure 5-9: Comparison: GUI versus MATLAB coefficients**

- **Conversion error:** The proposed system catches floating-point to a decimal conversion error. As the user presses *FP\_2\_DEC* button, the python script automatically reads floating-point to decimal converted binary coefficients. Then, convert them back to floating-point binary coefficients, compare them, and generates an absolute percentage difference and display it as a total percentage error.

## 6. DESIGN IMPLEMENTATION

This chapter explains the different filters that we have designed using the proposed system's GUI. The filter's specifications varied to design different filters. There are four types of filters in this system, and by varying user input, users can design a different kind of filters.

### 6.1 Filter design

The filter design depends on many parameters, particularly filter type, window type, sampling frequency, cut-off frequencies, transition bandwidth; by varying any of these parameters, the filter design changes. Using Easy-filter's GUI, users can design different filters by varying any of these parameters. This thesis included one example of each design. Here are the filter designs that this thesis included:

- Low-pass filter
- High-pass filter
- Band-pass filter
- Band-stop filter

Each of the above filters, redesign by varying sampling frequency, window type, cut-off frequencies, transition bandwidth, and switching between quality and quantity.

#### 6.1.1 *Low-pass filter design*

Although the user can consider any filter specification, this thesis considered a quality-based low-pass filter design that uses filter specifications, as shown in Table 6-1.

**Table 6-1: Low-pass filter specifications**

Parameters	Values
Passband edge frequency	500 Hz
Sampling frequency	3500 Hz
Transition bandwidth	800 Hz
Stopband attenuation	>70 dB

As the required stopband attenuation of this filter is >70 dB, so filter order calculated using the Blackman window. The designed filter's coefficients provided in section 8.1.1, converted to binary form by the Python script and used directly in Verilog code. This Verilog code is simulated using ModelSim. For the Blackman window number of filter, coefficients are 27. Equation 6-1 shows the formula to calculate filter coefficients for the Blackman window.

$$N = 5.98 * \left( \frac{\text{sampling frequency}}{\text{transition bandwidth}} \right) = 5.98 * \left( \frac{3500 \text{ Hz}}{800 \text{ Hz}} \right) \approx 27 \quad 6-1$$

### **6.1.2 High-pass filter design**

Although the user can consider any filter specification, this thesis considered a quality-based high-pass filter design that uses filter specifications, as described in Table 6-2. As the required stopband attenuation of this filter is >50 dB, so Hamming, Blackman, and Kaiser window are eligible to use for this filter design, and this thesis used the Hamming window to calculate filter order for this filter. The designed filter's coefficients provided in section 8.1.2, converted to binary form by the Python script and

used directly in Verilog code. This Verilog code is simulated using ModelSim. The Hamming window's coefficients calculated using equation 6-2.

$$N = 3.44 * \left( \frac{\text{sampling frequency}}{\text{transition bandwidth}} \right) = 3.44 * \left( \frac{5000 \text{ Hz}}{650 \text{ Hz}} \right) \approx 27 \quad 6-2$$

**Table 6-2: High-pass filter specifications**

Parameters	Values
Passband edge frequency	1500 Hz
Sampling frequency	5000 Hz
Transition Bandwidth	650 Hz
Stopband attenuation	>50 dB

### 6.1.3 Band-pass filter design

Although the user can consider any filter specification, this thesis considered a quality-based band-pass filter design that uses filter specifications, as described in Table 6-3. As the required stopband attenuation of this filter is 40 dB, Hanning, Hamming, Blackman, and Kaiser's window are eligible for this filter design. This thesis used the Kaiser window to calculate the order for this filter. The designed filter's coefficients provided in section 8.1.3, converted to binary form by the Python script and used directly in Verilog code. This Verilog code is simulated using ModelSim. Equation 6-3 shows the formula used to calculate beta for the Kaiser window:

$$\text{beta} = 0.5842 * (A - 21)^{\frac{2}{5}} + 0.07886 * (A - 21) \quad 6-3$$

$$\beta = 0.5842 * (40 - 21)^{\frac{2}{5}} + 0.07886 * (40 - 21) = 3.3953$$

**Table 6-3: Band-pass filter specifications**

Parameters	Values
Passband edge frequencies	1300 Hz, 2650 Hz
Sampling frequency	8000 Hz
Transition Bandwidth	496 Hz
Stopband attenuation	40 dB
Window	Kaiser

The Kaiser window's coefficients calculated using equation 6-4.

$$N = \left( \frac{(A - 8) * \text{sampling frequency}}{2.285 * 2 * \pi * \text{transition bandwidth}} \right) + 1 \quad 6-4$$

$$= \left( \frac{(40 - 8) * 8000}{2.285 * 2 * \pi * 496} \right) + 1 \approx 36$$

The coefficients are even in number, so to make the coefficients odd, add 1 to it, so total coefficients equal to  $36 + 1 = 37$

#### **6.1.4 Band-stop filter design**

Although the user can consider any filter specification, this thesis considered a quality-based band-stop filter design that uses filter specifications, as described in Table 6-4.

**Table 6-4: Band-stop filter specifications**

Parameters	Values
Passband edge frequencies	2000 Hz, 4000 Hz
Sampling frequency	10000 Hz
Transition Bandwidth	900 Hz
Stopband attenuation	>40 dB

As the required stopband attenuation of this filter is >40 dB, so Hanning, Hamming, Blackman, and Kaiser's window are eligible to use for this filter design, and this thesis used the Hanning window to calculate filter order for this filter. The designed filter's coefficients provided in section 8.1.4, converted to binary form by the Python script and used directly in Verilog code. This Verilog code is simulated using ModelSim. Equation 6-5 shows the formula used to calculate coefficients for the Hanning window:

$$N = 3.32 * \left( \frac{\text{sampling frequency}}{\text{transition bandwidth}} \right) = 3.32 * \left( \frac{10000}{900} \right) \approx 37 \quad 6-5$$



## 7. DESIGN SIMULATION & VALIDATION

This chapter provides the FIR filter design's verification and validation. The FIR filter's coefficients calculated, as mentioned in Chapter 6, are used to generate Verilog design and testbench codes automatically. The Verilog design and testbench simulate using ModelSim; it provides a visual representation of FIR filter's coefficients.

### 7.1 Design simulation

It uses ModelSim software to simulate the generated FIR filter's design. The proposed system generates many outputs for design simulation. After generating a custom FIR filter, it can simulate using ModelSim software. Easy-filter uses Verilog testbench to simulate the FIR filter design in ModelSim software. It uses *.do file* to load, simulate, add signals to the waveform, and terminate the simulation. The simulation of the FIR filter mainly checks if the filter coefficients, input signal, and output signal converted duly to binary form or not. APPENDIX A shows the Verilog design and testbench code of the low-pass filter.

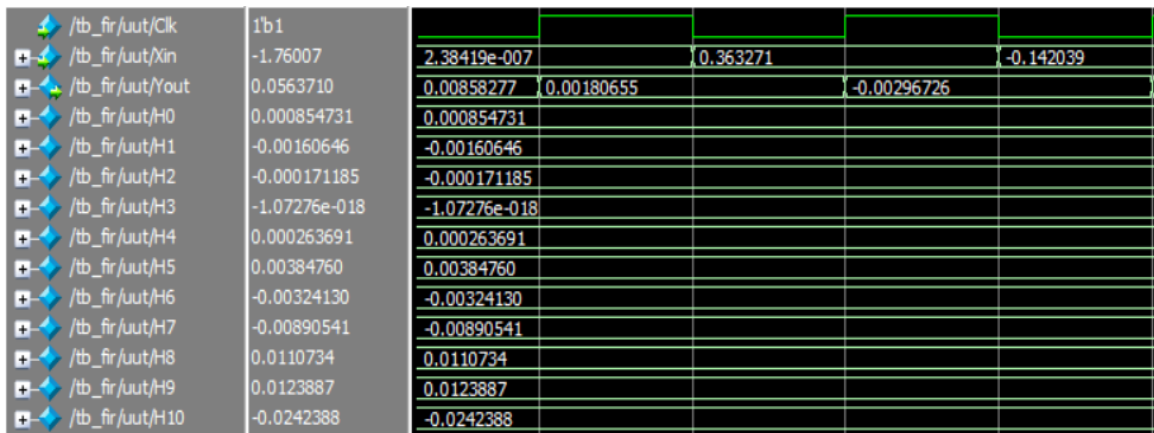


Figure 7-1: The FIR filter's simulation waveforms

Figure 7-1 shows the snippet of coefficients, input, and output waveform automatically generated by Easy-filter after the FIR filter simulation.

### 7.1.1 Testbench

In this thesis, the Verilog language is used to develop a testbench for the FIR filter. So, the FIR filter is the design under test (DUT). The testbench is used to stimulate the inputs to the design and observe the outputs. Its outputs are compared against the text file that contains Python generated FIR filter's coefficients. If the testbench generated coefficients are equal to the text file coefficients, then its test pass; otherwise, it fails. It checks that the coefficients do not change during floating-point to binary conversion, and all the signals behave according to the specifications. Figure 7-2 shows the testbench setup of the FIR filter.

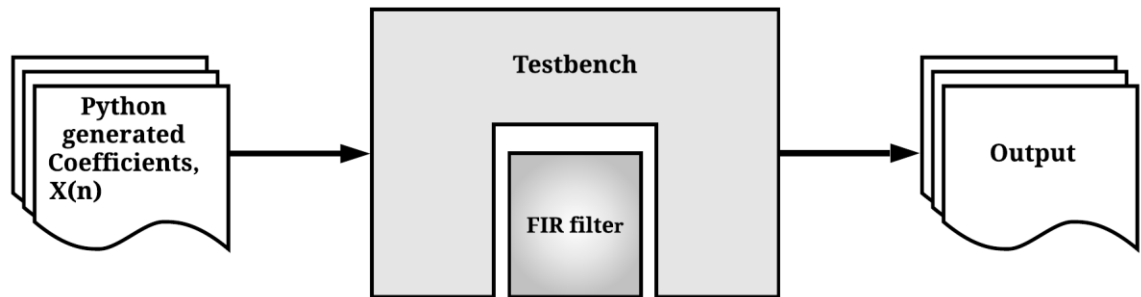


Figure 7-2: The testbench set up of the FIR filter

### 7.1.2 Floating-point

The standard specifies the following formats for the floating-point numbers.

➤ **Single precision:** It uses 32-bits, as shown in Figure 7-3, and has the following format:

- 1 bit- sign (0-positive,1-negative)
- 8 bit- exponent

- 23 bit- mantissa

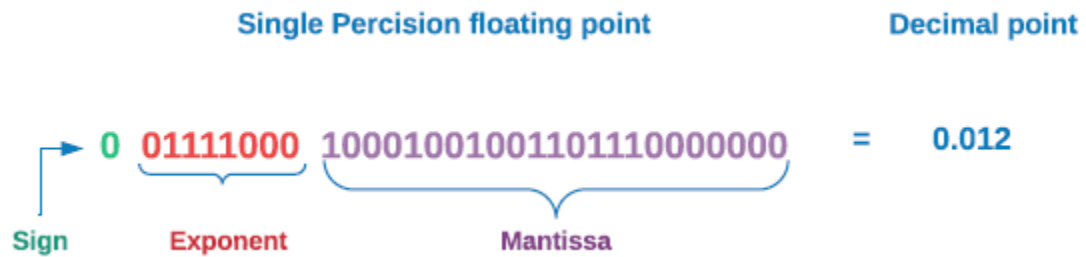


Figure 7-3: Single precision floating point

- **The sign bit:** The first bit in the floating-point number is the sign bit. If the number is positive, then set the sign bit to 0, and if the given number is negative, then set it to 1.
- **The exponent:** For large numbers, the exponent is positive, and it is negative for small numbers (fraction). The unsigned binary with 8 bits can represent numbers 0 through 255, and to represent negative numbers in floating-point, add 127 to the exponent. With this, it can represent numbers between 128 to -127. The number 128 is reserved to represent special numbers.

Example 1:

Required exponent	=	7
Floating-point exponent calculation	=	7+127 = 134
Binary representation	=	10000110

For exponent, if the leftmost bit is 1, then it is positive exponent means it is a large number, and if it is 0, then it is a negative number means a small number(fraction).

- **The mantissa:** The point move in scientific notation such that there is only a single(non-zero) digit to the left of it. In binary representation, this digit is 1 as

there is no other non-zero digit. So, after converting the given number to binary and before storing its mantissa, it drops 1 to store an extra bit in the mantissa.

Example 1:

Number to store = 100.01101110

Scientific notation = 1.0001101110

Mantissa to be stored = 0001101110

Example 2:

Number to store = 0.0001111110

Scientific notation = 1.111110

Mantissa to be stored = 111110

➤ **Special cases:** These cases include zero, infinity, not a number (division by zero, or the square root of the negative number).

- Zero: set sign bit 1 or 0 and all other bits to 0
- Infinity: for positive infinity, set sign bit to 0, and for negative set it to 1, set mantissa to all 1 and exponent to all 0
- Not a number: set sign bit either 0 or 1, set exponent to all 1, and set mantissa to a combination of 1 and 0

### 7.1.3 *Converting to floating-point*

Follow the following steps to convert a number to floating-point [41]:

- If the given number is positive, set the sign bit to 0, and if the number is negative, then set it to 1.
- Divide the number into two parts- the whole part and the fraction part.
- Convert these two parts into binary and then join them by a decimal point.

- Count how many spaces the binary point needs to move so that there is only one 1 to the left. If a binary point moves towards the left, then it is a positive number, and if it moves to the right, then it is a negative number; add 127 to the count and convert it to binary.
- Format the mantissa by dropping the first 1 and store the remaining 23 bits.

**Table 7-1: An example of floating-point conversion**

Parameters	Values
Decimal number	-61.6
Sign bit	<b>1</b>
Convert whole part to binary	111101
Convert fraction part to binary	0.1001100110011001100110
Join both parts together	111101.100110011001100110
Number of spaces to move binary point	5
Add 127 to get the exponent	132
Convert exponent to binary	<b>10000100</b>
Adjust the mantissa	1.11101100110011001100110
Remove leading 1	<b>11101100110011001100110</b>
<b>Result in binary: 1 10000100 11101100110011001100110</b>	

## 7.2 Design validation

For design validation, it uses MATLAB software. The Python script generates the FIR filter's coefficients and saves them in text format. Easy-filter validation has two parts.

- The same specifications fed to Python script and MATLAB software both generate FIR filter's coefficients and compare these coefficients
- The text file is generated by a Python script that contains FIR filter's coefficients

read by MATLAB code and generates filtered output and compares this output with the MATLAB generated filtered output with the same filter specifications. APPENDIX-B shows the MATLAB code for the low-pass filter that used to generate coefficients and filtered output.

## 8. RESULT EVALUATION AND COMPARISON

This chapter shows the results along with a comparative analysis of the experiment's coefficients and filtered output. Easy-filter's coefficients and filtered result compared against MATLAB generated coefficients and filtered output. It also shows the simulation results produced by ModelSim software using Easy-filter's generated Verilog design and testbench codes.

### 8.1 Easy-filter versus MATLAB

The proposed system uses its GUI to enter the user input. To enter user input for MATLAB, one should have basic knowledge of the MATLAB coding. To modify an existing code, one should have to understand it first to modify it according to requirement changes. On the other hand, the proposed system uses GUI to accommodate any specification changes; it is easy and less time-consuming. The user who does not have any coding background can also modify and design it without any problem. The following sections show the results of the experiments included in this thesis.

#### 8.1.1 *Low-pass filter results*

Table 8-1 shows Easy-filter's GUI generated low-pass filter's coefficients and MATLAB's generated low-pass filter's coefficients. The results show that the proposed system generates the same filter coefficients for the same user inputs. So, engineers can rely on this system as much as they rely on the MATLAB system to generate FIR filters. For this experiment, the input signal has 50 Hz, 200 Hz, 1000Hz, and 1500 Hz frequencies. The designed filter's cut-off frequency is 500 Hz, which passes only 50 Hz and 200 Hz frequencies and rejects 1000 Hz and 1500 frequencies.

**Table 8-1: Low-pass filter's coefficients comparison**

<b>Coefficient No.</b>	<b>Easy-filter GUI</b>	<b>MATLAB</b>	<b>%Error</b>
h1=h27	2.6561E-19	0.0000E+00	2.6561E-21
h2=h26	-1.3873E-04	-1.3873E-04	0.0000E+00
h3=h25	-2.8516E-04	-2.8516E-04	0.0000E+00
h4=h24	7.6479E-04	7.6479E-04	0.0000E+00
h5=h23	3.7093E-03	3.7093E-03	0.0000E+00
h6=h22	5.6859E-03	5.6859E-03	0.0000E+00
h7=h21	-3.1408E-18	-3.1408E-18	0.0000E+00
h8=h20	-1.6695E-02	-1.6695E-02	0.0000E+00
h9=h19	-3.3348E-02	-3.3348E-02	0.0000E+00
h10=h18	-2.3324E-02	-2.3324E-02	0.0000E+00
h11=h17	3.7001E-02	3.7001E-02	0.0000E+00
h12=h16	1.4089E-01	1.4089E-01	0.0000E+00
h13=h15	2.4292E-01	2.4292E-01	0.0000E+00
h14	2.8565E-01	2.8565E-01	0.0000E+00

The Verilog code uses Easy-filter's generated coefficients to create design and testbench code, further used by ModelSim to simulate the FIR filter. Figure 8-1 shows the snippet of the simulation waveform generated by the ModelSim simulator that shows the filter's coefficients.

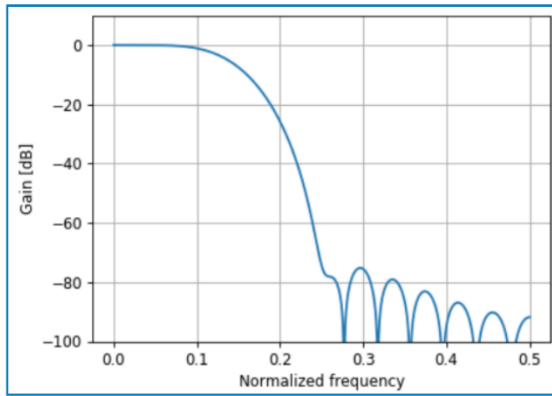


+ ◆ /tb_fir/uut/H0	2.65612e-019	2.65612e-019			
+ ◆ /tb_fir/uut/H1	-0.000138521	-0.000138521			
+ ◆ /tb_fir/uut/H2	-0.000285149	-0.000285149			
+ ◆ /tb_fir/uut/H3	0.000764608	0.000764608			
+ ◆ /tb_fir/uut/H4	0.00370932	0.00370932			
+ ◆ /tb_fir/uut/H5	0.00568581	0.00568581			
+ ◆ /tb_fir/uut/H6	-3.14078e-018	-3.14078e-018			
+ ◆ /tb_fir/uut/H7	-0.0166948	-0.0166948			
+ ◆ /tb_fir/uut/H8	-0.0333483	-0.0333483			
+ ◆ /tb_fir/uut/H9	-0.0233238	-0.0233238			
+ ◆ /tb_fir/uut/H10	0.0370007	0.0370007			
+ ◆ /tb_fir/uut/H11	0.140886	0.140886			
+ ◆ /tb_fir/uut/H12	0.242917	0.242917			
+ ◆ /tb_fir/uut/H13	0.285654	0.285654			
+ ◆ /tb_fir/uut/H14	0.242917	0.242917			
+ ◆ /tb_fir/uut/H15	0.140886	0.140886			
+ ◆ /tb_fir/uut/H16	0.0370007	0.0370007			
+ ◆ /tb_fir/uut/H17	-0.0233238	-0.0233238			
+ ◆ /tb_fir/uut/H18	-0.0333483	-0.0333483			
+ ◆ /tb_fir/uut/H19	-0.0166948	-0.0166948			
+ ◆ /tb_fir/uut/H20	-3.14078e-018	-3.14078e-018			
+ ◆ /tb_fir/uut/H21	0.00568581	0.00568581			
+ ◆ /tb_fir/uut/H22	0.00370932	0.00370932			
+ ◆ /tb_fir/uut/H23	0.000764608	0.000764608			
+ ◆ /tb_fir/uut/H24	-0.000285149	-0.000285149			
+ ◆ /tb_fir/uut/H25	-0.000138521	-0.000138521			
+ ◆ /tb_fir/uut/H26	2.65612e-019	2.65612e-019			

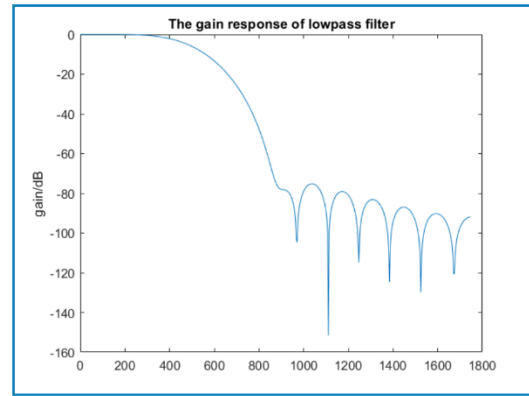
**Figure 8-1: The low-pass filter's coefficient's simulation waveform**

Figure 8-2 shows the frequency response generated by GUI and MATLAB. Both have the same frequency response and attenuation. When a pure sinusoidal input signal passes through a time-variant filter, then the output signal is also sinusoidal at the same frequency, but its magnitude and phase could have changed.

In the second part of the validation, the GUI generated coefficients are stored in a text file and fed to MATLAB code instead of generating its own coefficients.

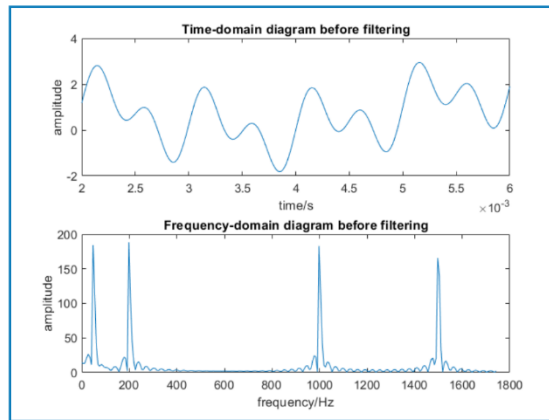


(a) The proposed GUI generated frequency response

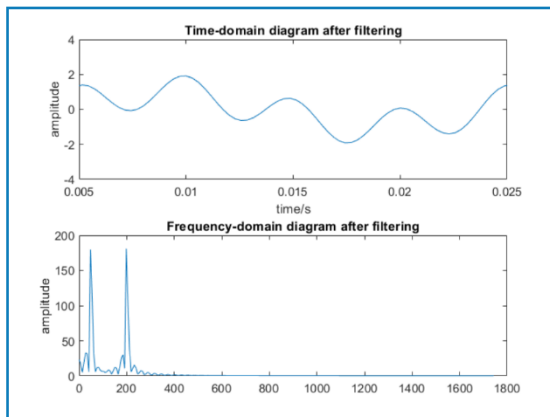


(b) MATLAB generated frequency response

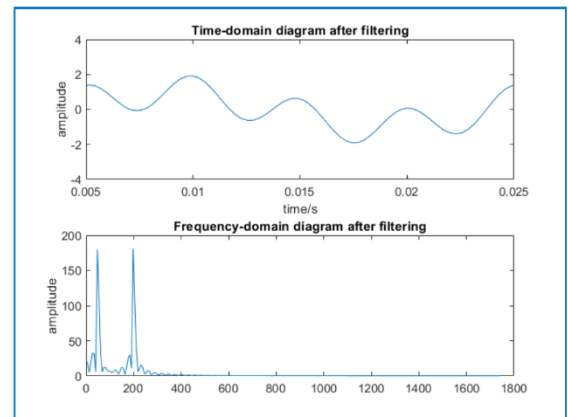
Figure 8-2: The low-pass filter's frequency response



(a) Input signal



(b) Filtered signal generated by MATLAB



(c) Filtered signal generated by proposed GUI

Figure 8-3: Filtered output comparison

Figure 8-3 shows the comparison of MATLAB generated outputs ( using coefficients generated by MATLAB software) and MATLAB generated output with coefficients fed from the GUI system. Easy-filter's GUI generates the same filtered output as MATLAB.

The Verilog design code generated by the Easy-filter GUI is used to generate the circuit using the Xilinx synthesis tool for low-pass filter design. Table 8-2 shows implementation results for the low-pass filter using sampling frequency 20000 Hz, cutoff frequency 5000 Hz, and transition width 1000 Hz. Artix-7's part number 'xc7a200tsbv484-1' is used. This device's breaking point is 190 coefficients, and 454.545 MHz is the highest frequency obtained using this device.

**Table 8-2: Low-pass filter's Implementation results**

	<b>Rectangular</b>	<b>Hanning</b>	<b>Hamming</b>	<b>Blackman</b>	<b>Kaiser</b>	<b>Kaiser</b>
Coefficients	19	67	69	121	75	95
Total On-chip Power	1.167W	4.595W	4.607W	7.903W	5.062W	6.362W
LUT	9.34%	34.46%	35.67%	63.00%	38.84%	49.30%
FF	1.19%	4.40%	4.57%	8.03%	4.97%	6.30%
DSP	2.16%	9.19%	9.46%	16.49%	10.27%	12.97%
IO	22.81%	22.81%	22.81%	22.81%	22.81%	22.81%

### **8.1.2 High-pass filter results**

Table 8-3 shows that Easy-filter generated high-pass filter's coefficients, and MATLAB's generated high-pass filter's coefficients.

**Table 8-3: High-pass filter's coefficients comparison**

<b>Coefficient No.</b>	<b>Easy-filter GUI</b>	<b>MATLAB</b>	<b>%Error</b>
h1=h27	1.151E-03	1.151E-03	0.0000
h2=h26	1.455E-03	1.455E-03	0.0000
h3=h25	-3.650E-03	-3.650E-03	0.0000
h4=h24	-3.050E-18	-3.050E-18	0.0000
h5=h23	9.369E-03	9.369E-03	0.0000
h6=h22	-8.810E-03	-8.809E-03	0.0000
h7=h21	-1.294E-02	-1.294E-02	0.0000
h8=h20	3.003E-02	3.003E-02	0.0000
h9=h19	1.096E-17	1.096E-17	0.0000
h10=h18	-6.061E-02	-6.061E-02	0.0000
h11=h17	5.512E-02	5.512E-02	0.0000
h12=h16	8.857E-02	8.857E-02	0.0000
h13=h15	-2.985E-01	-2.985E-01	0.0000
h14	3.998E-01	3.998E-01	0.0000

The results show that Easy-filter generates the same filter coefficients for the same user inputs. So, engineers can rely on this system as much as they rely on the MATLAB system to generate FIR filters. For this experiment, the input signal has 500 Hz and 1000 Hz, 2000 Hz, and 2300 Hz frequencies. The designed filter's cut-off frequency is 1500 Hz, which passes only 2000 Hz and 2300 Hz frequencies and rejects 500 Hz and 1000 Hz frequencies.

+ ◆ /tb_fir/uut/H0	0.00115061	0.00115061		
+ ◆ /tb_fir/uut/H1	0.00145483	0.00145483		
+ ◆ /tb_fir/uut/H2	-0.00364971	-0.00364971		
+ ◆ /tb_fir/uut/H3	-3.04961e-018	-3.04961e-018		
+ ◆ /tb_fir/uut/H4	0.00936913	0.00936913		
+ ◆ /tb_fir/uut/H5	-0.00880933	-0.00880933		
+ ◆ /tb_fir/uut/H6	-0.0129442	-0.0129442		
+ ◆ /tb_fir/uut/H7	0.0300269	0.0300269		
+ ◆ /tb_fir/uut/H8	1.09576e-017	1.09576e-017		
+ ◆ /tb_fir/uut/H9	-0.0606124	-0.0606124		
+ ◆ /tb_fir/uut/H10	0.0551212	0.0551212		
+ ◆ /tb_fir/uut/H11	0.0885718	0.0885718		
+ ◆ /tb_fir/uut/H12	-0.298523	-0.298523		
+ ◆ /tb_fir/uut/H13	0.399784	0.399784		
+ ◆ /tb_fir/uut/H14	-0.298523	-0.298523		
+ ◆ /tb_fir/uut/H15	0.0885718	0.0885718		
+ ◆ /tb_fir/uut/H16	0.0551212	0.0551212		
+ ◆ /tb_fir/uut/H17	-0.0606124	-0.0606124		
+ ◆ /tb_fir/uut/H18	1.09576e-017	1.09576e-017		
+ ◆ /tb_fir/uut/H19	0.0300269	0.0300269		
+ ◆ /tb_fir/uut/H20	-0.0129442	-0.0129442		
+ ◆ /tb_fir/uut/H21	-0.00880933	-0.00880933		
+ ◆ /tb_fir/uut/H22	0.00936913	0.00936913		
+ ◆ /tb_fir/uut/H23	-3.04961e-018	-3.04961e-018		
+ ◆ /tb_fir/uut/H24	-0.00364971	-0.00364971		
+ ◆ /tb_fir/uut/H25	0.00145483	0.00145483		
+ ◆ /tb_fir/uut/H26	0.00115061	0.00115061		

Figure 8-4: The high-pass filter's coefficient's simulation waveform

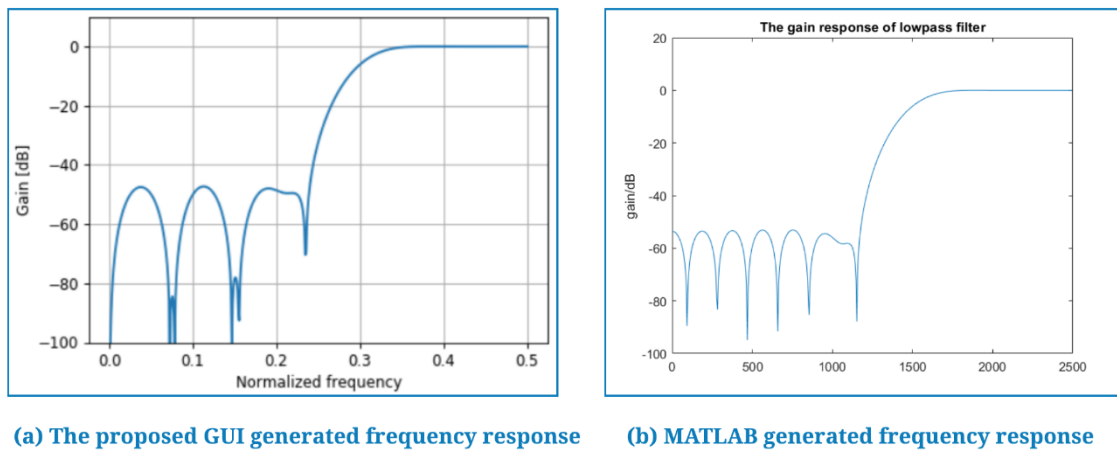
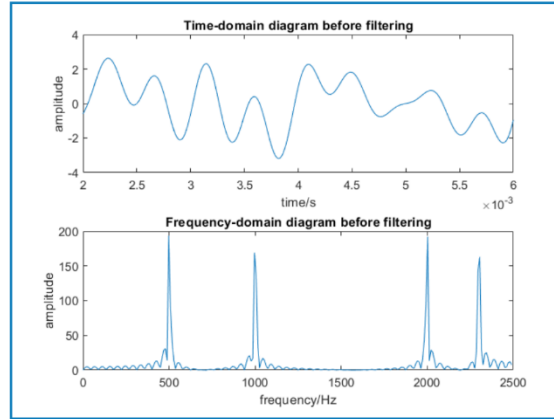
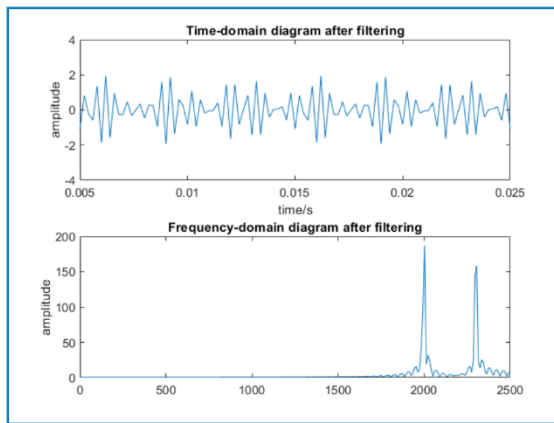


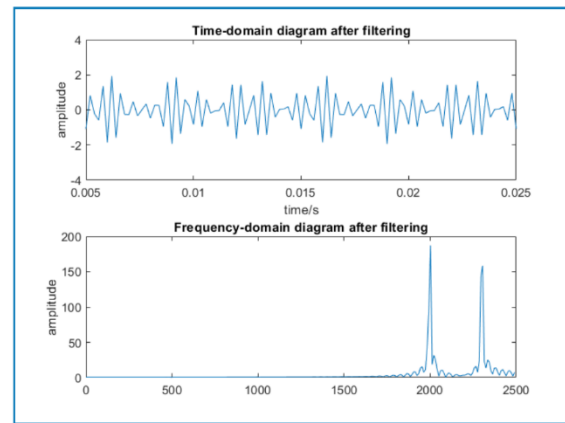
Figure 8-5: The high-pass filter's frequency response



(a) Input signal



(b) Filtered signal generated by MATLAB



(c) Filtered signal generated by proposed GUI

**Figure 8-6: Filtered high-pass filter's output comparison**

The Verilog code uses the GUI Easy-filter's generated coefficients to create design and testbench code, further used by ModelSim to simulate the FIR filter. Figure 8-5 shows the snippet of the simulation waveform that shows the filter's coefficients, frequency response generated by a Python script, and MATLAB. Both have the same frequency response and attenuation.

In the second part of the validation, Python generated coefficients are stored in a text file and fed to MATLAB code instead of generating its own coefficients. Figure 8-6 compares MATLAB generated outputs and MATLAB generated output with coefficients

fed from the GUI system. Easy-filter GUI generates the same filtered output as MATLAB. When a pure sinusoidal input signal passes through a time-variant filter, then the output signal is also sinusoidal at the same frequency. However, its magnitude and phase could have changed; the same is the case with the proposed GUI filter's output.

The Verilog design code generated by Easy-filter GUI was used to generate the circuit using the Xilinx synthesis tool for low-pass filter design. Table 8-4 shows the high-pass filter implementation results using sampling frequency 20000 Hz, cutoff frequency 4000 Hz, and transition width 800 Hz. Artix-7's part number 'xc7a200tsbv484-1' is used. This device's breaking point is 190 coefficients, and 454.545 MHz is the highest frequency obtained using this device.

**Table 8-4: High-pass filter's Implementation results**

	<b>Rectangular</b>	<b>Hanning</b>	<b>Hamming</b>	<b>Blackman</b>	<b>Kaiser with A=40</b>
Coefficients	23	83	87	151	57
Total On-chip Power	1.521W	5.396W	5.663W	9.597W	3.725W
LUT	11.72%	43.01%	45.15%	78.75%	29.39%
FF	1.52%	5.49%	5.76%	10.03%	3.77%
DSP	3.24%	11.35%	11.89%	20.54%	7.84%
IO	22.81%	22.81%	22.81%	22.81%	22.81%

### **8.1.3 Band-pass filter results**

Table 8-5 shows Easy-filter's GUI generated band-pass filter's coefficients and MATLAB generated band-pass filter's coefficients. The results show that Easy-filter generates the same filter coefficients for the same user inputs. So, engineers can rely on

this system as much as they rely on the MATLAB system to generate FIR filters. For this experiment, the input signal has 500 Hz, 1500 Hz, 2000 Hz, and 3800 Hz frequencies. The designed filter's cut-off frequencies are 1300 Hz and 2650 Hz, so it passes only 1500 Hz and 2000 Hz frequencies and rejects 500 Hz and 3800 Hz frequencies as they are not in the pass-band range.

In MATLAB code, beta  $N$  and transition bandwidth are calculated automatically by a *fir1* function, and the given stopband attenuation is 0.01. On the other hand, the proposed Python code used a formula mentioned in section 6.1.3 to calculate beta and  $N$ . The code used to calculate stopband attenuation for the Easy-filter is shown in equation 8-1.

$$A = R_s * \frac{\text{sampling frequency}}{2} = 0.01 * \frac{8000}{2} = 40 \text{ dB} \quad 8-1$$

However, the formula shown in equation 8-2 is used to calculate transition bandwidth.

$$36 = \left( \frac{(A - 8) * \text{sampling frequency}}{2.285 * 2 * \pi * \text{transition bandwidth}} \right) + 1 \quad 8-2$$

$$36 = \left( \frac{(40 - 8) * 8000}{2.285 * 2 * \pi * \text{transition bandwidth}} \right) + 1$$

$$\text{transition bandwidth} \approx 496$$



**Table 8-5: Band-pass filter's coefficients comparison**

<b>Coefficient No.</b>	<b>Easy-filter GUI</b>	<b>MATLAB</b>	<b>%Error</b>
h1=h37	5.7317E-04	5.7317E-04	0.00E+00
h2=h36	9.6570E-04	9.6570E-04	0.00E+00
h3=h35	7.6609E-03	7.6609E-03	0.00E+00
h4=h34	-3.7741E-03	-3.7741E-03	0.00E+00
h5=h33	-1.4642E-02	-1.4642E-02	0.00E+00
h6=h32	3.0479E-03	3.0479E-03	0.00E+00
h7=h31	2.0035E-03	2.0035E-03	0.00E+00
h8=h30	3.0263E-03	3.0263E-03	0.00E+00
h9=h29	3.2308E-02	3.2308E-02	0.00E+00
h10=h28	-8.4859E-03	-8.4859E-03	0.00E+00
h11=h27	-5.1956E-02	-5.1956E-02	0.00E+00
h12=h26	5.3487E-03	5.3487E-03	0.00E+00
h13=h25	3.4973E-03	3.4973E-03	0.00E+00
h14=h24	5.2287E-03	5.2287E-03	0.00E+00
h15=h23	1.2519E-01	1.2519E-01	0.00E+00
h16=h22	-1.1924E-02	-1.1924E-02	0.00E+00
h17=h21	-2.7084E-01	-2.7084E-01	0.00E+00
h18=h20	6.2501E-03	6.2501E-03	0.00E+00
h19	3.3522E-01	3.3522E-01	0.00E+00

+ /tb_fir/uut/H0	0.000573158	0.000573158		
+ /tb_fir/uut/H1	0.000965595	0.000965595		
+ /tb_fir/uut/H2	0.00766087	0.00766087		
+ /tb_fir/uut/H3	-0.00377393	-0.00377393		
+ /tb_fir/uut/H4	-0.0146420	-0.0146420		
+ /tb_fir/uut/H5	0.00304770	0.00304770		
+ /tb_fir/uut/H6	0.00200343	0.00200343		
+ /tb_fir/uut/H7	0.00302625	0.00302625		
+ /tb_fir/uut/H8	0.0323079	0.0323079		
+ /tb_fir/uut/H9	-0.00848579	-0.00848579		
+ /tb_fir/uut/H10	-0.0519557	-0.0519557		
+ /tb_fir/uut/H11	0.00534844	0.00534844		
+ /tb_fir/uut/H12	0.00349712	0.00349712		
+ /tb_fir/uut/H13	0.00522852	0.00522852		
+ /tb_fir/uut/H14	0.125193	0.125193		
+ /tb_fir/uut/H15	-0.0119238	-0.0119238		
+ /tb_fir/uut/H16	-0.270836	-0.270836		
+ /tb_fir/uut/H17	0.00624990	0.00624990		
+ /tb_fir/uut/H18	0.335215	0.335215		
+ /tb_fir/uut/H19	0.00624990	0.00624990		
+ /tb_fir/uut/H20	-0.270836	-0.270836		
+ /tb_fir/uut/H21	-0.0119238	-0.0119238		
+ /tb_fir/uut/H22	0.125193	0.125193		
+ /tb_fir/uut/H23	0.00522852	0.00522852		
+ /tb_fir/uut/H24	0.00349712	0.00349712		
+ /tb_fir/uut/H25	0.00534844	0.00534844		
+ /tb_fir/uut/H26	-0.0519557	-0.0519557		
+ /tb_fir/uut/H27	-0.00848579	-0.00848579		
+ /tb_fir/uut/H28	0.0323079	0.0323079		
+ /tb_fir/uut/H29	0.00302625	0.00302625		
+ /tb_fir/uut/H30	0.00200343	0.00200343		
+ /tb_fir/uut/H31	0.00304770	0.00304770		
+ /tb_fir/uut/H32	-0.0146420	-0.0146420		
+ /tb_fir/uut/H33	-0.00377393	-0.00377393		

**Figure 8-7: The band-pass filter's coefficient's simulation waveform**

The normalized cut-off frequencies calculated based on  $f_{cuts}$  frequencies are 0.3250 and 0.6625. Equation 8-3 and 8-4 show the formulas used to calculate the normalized cut-off frequencies for the Easy-filter.

$$f_L(\text{normalized}) = f_L * \frac{(\text{sampling frequency})}{2} = 1300 * \frac{2}{8000} = 0.3250 \text{ Hz} \quad 8-3$$

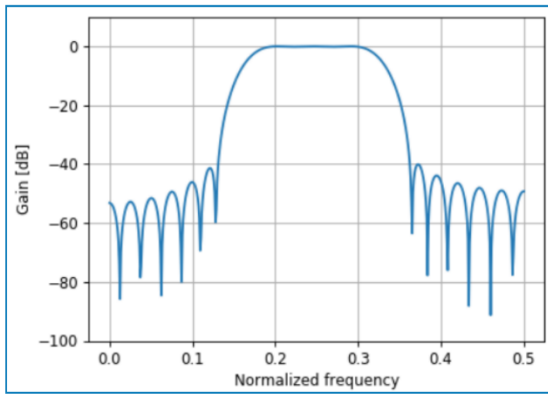
And,

$$f_H(\text{normalized}) = f_H * \frac{(\text{sampling frequency})}{2} = 2650 * \frac{2}{8000} = 0.6625 \text{ Hz} \quad 8-4$$

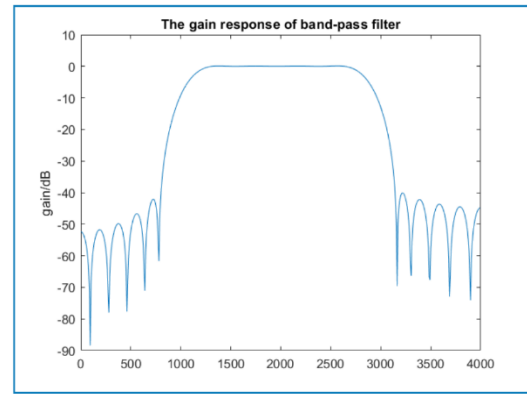
The Verilog code uses Python generated coefficients to create design and testbench code, further used by ModelSim to simulate the FIR filter. Figure 8-7 shows the snippet of the simulation waveform that shows the filter's coefficients.

Figure 8-8 shows the frequency response generated by GUI and MATLAB. Both have the same frequency response and attenuation.

In the second part of the validation, the Python generated coefficients are stored in a text file and fed to MATLAB code instead of generating its own coefficients.

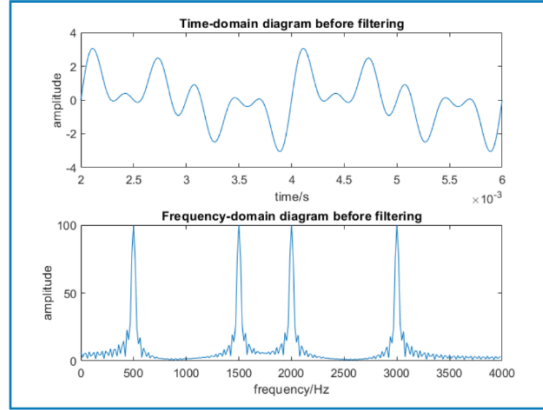


(a) The proposed GUI generated frequency response

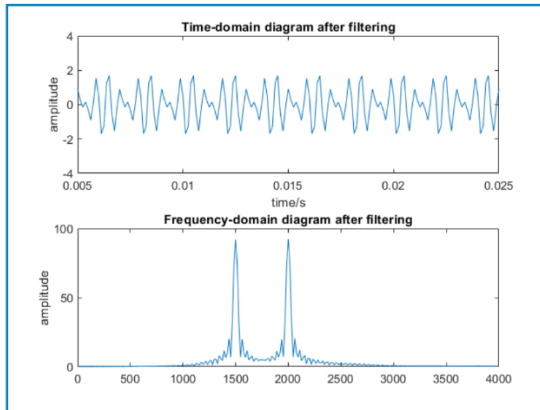


(b) MATLAB generated frequency response

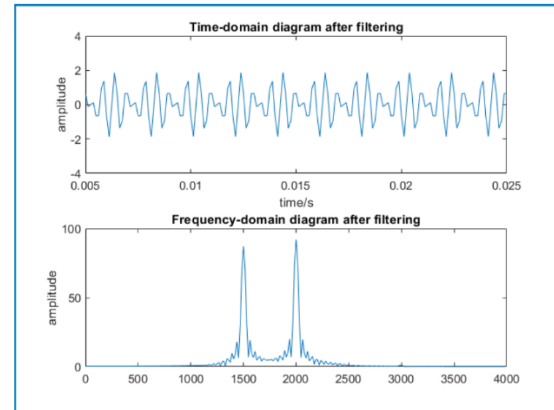
**Figure 8-8: The band-pass filter's frequency response**



(a) Input signal



(b) Filtered signal generated by MATLAB



(c) Filtered signal generated by proposed GUI

**Figure 8-9: Filtered band-pass filter's output comparison**

Figure 8-9 compares MATLAB generated outputs and MATLAB generated output with Python generated coefficients. Easy-filter generates the same filtered output as MATLAB. When a pure sinusoidal input signal passes through a time-variant filter, then the output signal is also sinusoidal at the same frequency. However, its magnitude and phase could have changed; the same is the case with the proposed GUI filter's output.

The Verilog design code generated by Easy-filter was used to generate the circuit using the Xilinx synthesis tool for low-pass filter design. Table 8-6 shows implementation results for the band-pass filter using sampling frequency 20000 Hz,

cutoff frequencies 2500 Hz and 8000 Hz, and transition width 900 Hz. Artix-7's part number 'xc7a200tsbv484-1' is used. This device's breaking point is 190 coefficients, and 454.545 MHz is the highest frequency obtained using this device.

**Table 8-6: Band-pass filter's Implementation results**

	<b>Rectangular</b>	<b>Hanning</b>	<b>Hamming</b>	<b>Blackman</b>	<b>Kaiser with A=68</b>
Coefficients	21	75	77	133	187
Total On-chip Power	1.439 W	4.869 W	4.991 W	8.373 W	12.121 W
LUT	10.69%	38.83%	39.91%	69.28%	95.58%
FF	1.38%	4.96%	5.10%	8.82%	12.39%
DSP	2.97%	10.27%	10.54%	18.11%	25.14%
IO	22.81%	22.81%	22.81%	22.81%	22.81%

#### **8.1.4 Band-stop filter results**

Table 8-7 shows Easy-filter's GUI generated band-stop filter's coefficients and MATLAB's generated band-stop filter's coefficients. The results show that the Easy-filter generates the same filter coefficients for the same user inputs. So, engineers can rely on this system as much as they rely on the MATLAB system to generate FIR filters. For this experiment, the input signal has 500 Hz, 2500 Hz, 3200 Hz, and 4500 Hz frequencies.

**Table 8-7: Band-stop filter's coefficients comparison**

<b>Coefficient No.</b>	<b>Easy-filter GUI</b>	<b>MATLAB</b>	<b>%Error</b>
h1=h37	0.0000E+00	0.0000E+00	0.0000
h2=h36	2.1889E-04	2.1892E-04	0.0000
h3=h35	2.1806E-04	2.1798E-04	0.0000
h4=h34	1.0442E-18	8.7082E-18	0.0000
h5=h33	-9.6679E-04	-9.6642E-04	0.0000
h6=h32	-6.7303E-03	-6.7314E-03	0.0000
h7=h31	1.0206E-02	1.0207E-02	0.0000
h8=h30	3.4606E-03	3.4592E-03	0.0000
h9=h29	6.4409E-18	-9.6662E-18	0.0000
h10=h28	-6.4281E-03	-6.4256E-03	0.0000
h11=h27	-3.5933E-02	-3.5939E-02	0.0000
h12=h26	4.6958E-02	4.6966E-02	0.0000
h13=h25	1.4463E-02	1.4458E-02	0.0000
h14=h24	1.2804E-17	4.4838E-17	0.0000
h15=h23	-2.5543E-02	-2.5533E-02	0.0000
h16=h22	-1.5235E-01	-1.5238E-01	0.0000
h17=h21	2.3755E-01	2.3759E-01	0.0000
h18=h20	1.1483E-01	1.1478E-01	0.0000
h19	6.0010E-01	6.0015E-01	0.0000

The designed filter's cut-off frequencies are 2000 Hz and 4000 Hz, so it passes only 500 Hz and 4500 Hz frequencies and rejects 2500 Hz and 3200 Hz frequencies as they are in the band-stop range.

Easy-filter used a formula mentioned in section 6.1.3 to calculate  $N$ . The normalized cut-off frequencies calculated using equations 8-5 and 8-6.

$$f_L = 2000 * \frac{2}{(\text{sampling frequency})} = 2000 * \frac{2}{10000} = 0.4 \text{ Hz} \quad 8-5$$

And,

$$f_H = 4000 * \frac{2}{(\text{sampling frequency})} = 4000 * \frac{2}{10000} = 0.8 \text{ Hz} \quad 8-6$$

The Verilog code uses the GUI system's generated coefficients to create design and testbench code, further used by ModelSim to simulate the FIR filter. Figure 8-10 shows the snippet of the simulation waveform that shows the filter's coefficients.

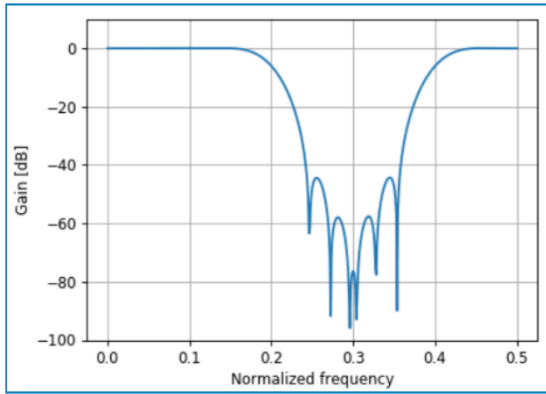
Figure 8-11 shows the frequency response generated by Python script and MATLAB. Both have the same frequency response and attenuation. In the second part of the validation, the GUI generated coefficients are stored in a text file and fed to MATLAB code instead of generating its own coefficients. Figure 8-12 compares MATLAB generated outputs and MATLAB generated output with coefficients fed from the GUI system. Easy-filter generates the same filtered output as MATLAB.

+ /tb_fir/uut/H0	+0	+0
+ /tb_fir/uut/H1	0.000218868	0.000218868
+ /tb_fir/uut/H2	0.000217915	0.000217915
+ /tb_fir/uut/H3	1.04424e-018	1.04424e-018
+ /tb_fir/uut/H4	-0.000966787	-0.000966787
+ /tb_fir/uut/H5	-0.00673008	-0.00673008
+ /tb_fir/uut/H6	0.0102055	0.0102055
+ /tb_fir/uut/H7	0.00346041	0.00346041
+ /tb_fir/uut/H8	6.44086e-018	6.44086e-018
+ /tb_fir/uut/H9	-0.00642800	-0.00642800
+ /tb_fir/uut/H10	-0.0359333	-0.0359333
+ /tb_fir/uut/H11	0.0469582	0.0469582
+ /tb_fir/uut/H12	0.0144632	0.0144632
+ /tb_fir/uut/H13	1.28044e-017	1.28044e-017
+ /tb_fir/uut/H14	-0.0255427	-0.0255427
+ /tb_fir/uut/H15	-0.152352	-0.152352
+ /tb_fir/uut/H16	0.237549	0.237549
+ /tb_fir/uut/H17	0.114828	0.114828
+ /tb_fir/uut/H18	0.600103	0.600103
+ /tb_fir/uut/H19	0.114828	0.114828
+ /tb_fir/uut/H20	0.237549	0.237549
+ /tb_fir/uut/H21	-0.152352	-0.152352
+ /tb_fir/uut/H22	-0.0255427	-0.0255427
+ /tb_fir/uut/H23	1.28044e-017	1.28044e-017
+ /tb_fir/uut/H24	0.0144632	0.0144632
+ /tb_fir/uut/H25	0.0469582	0.0469582
+ /tb_fir/uut/H26	-0.0359333	-0.0359333
+ /tb_fir/uut/H27	-0.00642800	-0.00642800
+ /tb_fir/uut/H28	6.44086e-018	6.44086e-018
+ /tb_fir/uut/H29	0.00346041	0.00346041
+ /tb_fir/uut/H30	0.0102055	0.0102055
+ /tb_fir/uut/H31	-0.00673008	-0.00673008
+ /tb_fir/uut/H32	-0.000966787	-0.000966787
+ /tb_fir/uut/H33	1.04424e-018	1.04424e-018

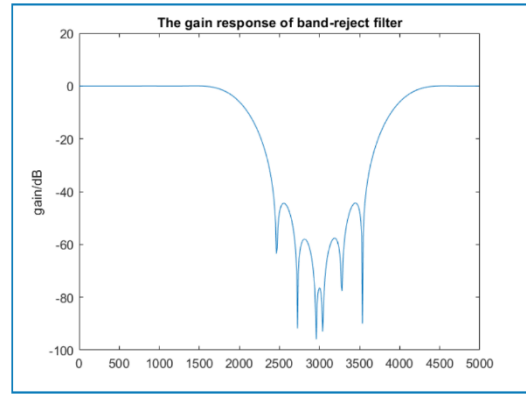
**Figure 8-10: The band-stop filter's coefficient's simulation waveform**

The Verilog design code generated by easy-filter was used to generate the circuit using the Xilinx synthesis tool for low-pass filter design. Table 8-8 shows implementation results for the band-stop filter using sampling frequency 20000 Hz, cutoff frequencies 3000 Hz and 7000 Hz, and transition width 1100 Hz. Artix-7's part number 'xc7a200tsbv484-1' is used. This device's breaking point is 190 coefficients, and 454.545 MHz is the highest frequency obtained using this device.



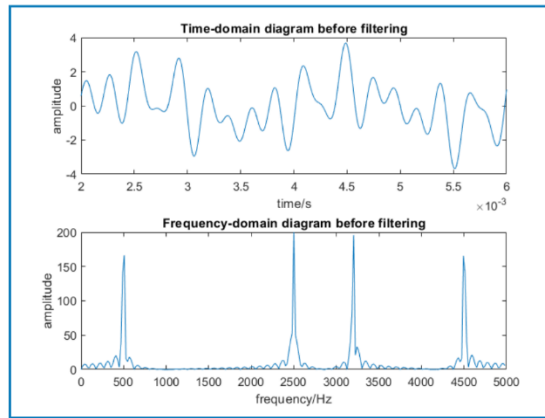


(a) The proposed GUI generated frequency response

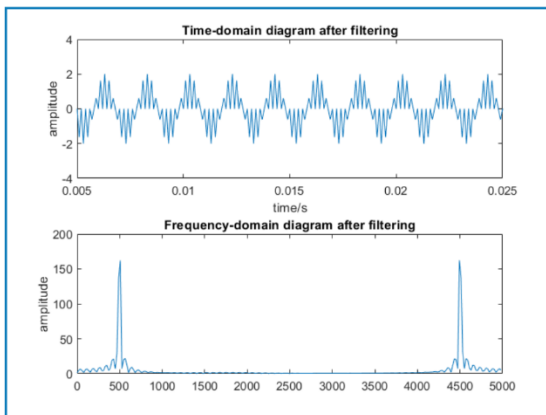


(b) MATLAB generated frequency response

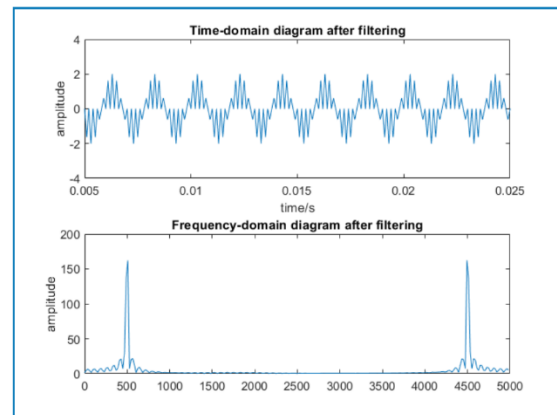
Figure 8-11: The band-stop filter's frequency response



(a) Input signal



(b) Filtered signal generated by MATLAB



(c) Filtered signal generated by proposed GUI

Figure 8-12: Filtered band-stop filter's output comparison

**Table 8-8: Band-stop filter's Implementation results**

	<b>Rectangular</b>	<b>Hanning</b>	<b>Hamming</b>	<b>Blackman</b>	<b>Kaiser with A=55</b>
Coefficients	17	61	63	109	121
Total On-chip Power	1.208 W	3.886 W	3.975 W	7.189 W	7.632 W
LUT	8.57%	31.50%	32.58%	56.68%	62.92%
FF	1.12%	4.03%	4.17%	7.23%	8%
DSP	2.43%	8.38%	8.65%	14.86%	16.22%
IO	22.81%	22.81%	22.81%	22.81%	22.81%

## 9. CONCLUSION AND FURTHER RESEARCH

Easy-filter automatically generates Verilog and MATLAB code for the FIR filter with user-defined specifications. It shortens the development time, increases the efficiency of Verilog coding, and decrease the staff-hour. This method helps even those users who do not know Verilog or MATLAB coding. Additionally, GUI makes it convenient, fast, intuitive, and flexible.

This thesis presented an automated generation, verification, and validation of the most used electronics component, the FIR filters. According to some experienced designers, it takes approximately two days to design and verify a filter. This automated flow reduces the design time to a few minutes. A complete chip project usually has dozens of filters so that this flow can save many hours at the project level.

This flow guarantees the design's timing closure, as it saves time in design and verification and reduces the back-end team's work. It requires less effort in documentation to describe a filter and saves the time that it takes to study, develop, and documentation.

This system can fulfill the original filter's requirements without depending on high-level synthesis tools, which keeps complexity and license costs low. The user can freely choose the filter length and coefficients.

The code quality and coefficients generated by Easy-filter are checked with both ModelSim and MATLAB software.

However, it still needs some improvements.

## 9.1 Further research

Below is the list of recommendations for further research on Python script-controlled GUI used for automation:

- GUI for IIR filter
- Verilog design and testbench automation of other essential electronics components like CPU (Central Processing Unit)
- Accessing Xilinx from GUI
- The optimization of the design
- Replacing multiplier with shift and add

## APPENDIX SECTION

### APPENDIX A.1 Verilog design: Low-pass filter

```
// FIR filter order = 26

module fir_27tap(
    input Clk,
    input signed [31:0] Xin,
    output reg signed [31:0] Yout);

//internal variables

wire signed[31:0]
H0,H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,H11,H12,H13,H14,H15,H16,H17,H18,H19,H20,H21,
H22,H23,H24,H25,H26;

wire
signed[31:0]MCM0,MCM1,MCM2,MCM3,MCM4,MCM5,MCM6,MCM7,MCM8,MCM9,MCM10,MCM
11,
MCM12,MCM13,MCM14,MCM15,MCM16,MCM17,MCM18,MCM19,MCM20,MCM21,MCM22,MCM
23,MCM24, MCM25,MCM26;

wire signed [31:0]add_out1,add_out2,add_out3,add_out4,add_out5,add_out6,
add_out7,add_out8,add_out9,add_out10,add_out11,add_out12,add_out13,add_out14,add_out15,add_out16
,add_out17,add_out18,add_out19,add_out20,add_out21,add_out22,add_out23,add_out24,add_out25,add_o
ut26;

wire signed [31:0]Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,
Q18,Q19,Q20,Q21,Q22,Q23,Q24,Q25,Q26;

// filter coefficient initializations

assign H0 = 32'b00100000100111001100101000110100; // 2.656123646688299902e-19
assign H1 = 32'b10111001000100010100000000000000; // -1.387284829302638301e-04
assign H2 = 32'b10111001100101011000000000000000; // -2.851629516083550408e-04
```

```

assign H3 = 32'b00111010010010000111000000000000; // 7.647929107363890056e-04
assign H4 = 32'b00111011011100110001100000000000; // 3.709346677506889216e-03
assign H5 = 32'b00111011101110100101000000000000; // 5.685874479165271313e-03
assign H6 = 32'b10100010011001111011111110010001; // -3.140775377194302274e-18
assign H7 = 32'b10111100100010001100001110000000; // -1.669499760510593719e-02
assign H8 = 32'b10111101000010001001100001000000; // -3.334846494083928276e-02
assign H9 = 32'b10111100101111110001000110000000; // -2.332396573602673120e-02
assign H10 = 32'b00111101000101111000111000000000; // 3.700083096764832102e-02
assign H11 = 32'b00111110000100000100010010000000; // 1.408865162512743796e-01
assign H12 = 32'b00111110011110001011111100110000; // 2.429170367675183018e-01
assign H13 = 32'b00111110100100100100000100111000; // 2.856538433253219544e-01
assign H14 = 32'b00111110011110001011111100110000; // 2.429170367675183295e-01
assign H15 = 32'b00111110000100000100010010000000; // 1.408865162512743796e-01
assign H16 = 32'b00111101000101111000111000000000; // 3.700083096764832796e-02
assign H17 = 32'b10111100101111110001000110000000; // -2.332396573602673120e-02
assign H18 = 32'b10111101000010001001100001000000; // -3.334846494083929663e-02
assign H19 = 32'b10111100100010001100001110000000; // -1.669499760510594760e-02
assign H20 = 32'b10100010011001111011111110010001; // -3.140775377194304585e-18
assign H21 = 32'b00111011101110100101000000000000; // 5.685874479165271313e-03
assign H22 = 32'b00111011011100110001100000000000; // 3.709346677506889216e-03
assign H23 = 32'b00111010010010000111000000000000; // 7.647929107363904151e-04
assign H24 = 32'b10111001100101011000000000000000; // -2.851629516083556371e-04
assign H25 = 32'b10111001000100010100000000000000; // -1.387284829302649143e-04
assign H26 = 32'b00100000100111001100101000110100; // 2.656123646688299902e-19

//Multiple constant multiplications

multiplier fp1 (.a_1(H26),.b_1(Xin),.out_1(MCM26));

multiplier fp2 (.a_1(H25),.b_1(Xin),.out_1(MCM25));

multiplier fp3 (.a_1(H24),.b_1(Xin),.out_1(MCM24));

```

```

multiplier fp4 (.a_1(H23),.b_1(Xin),.out_1(MCM23));
multiplier fp5 (.a_1(H22),.b_1(Xin),.out_1(MCM22));
multiplier fp6 (.a_1(H21),.b_1(Xin),.out_1(MCM21));
multiplier fp7 (.a_1(H20),.b_1(Xin),.out_1(MCM20));
multiplier fp8 (.a_1(H19),.b_1(Xin),.out_1(MCM19));
multiplier fp9 (.a_1(H18),.b_1(Xin),.out_1(MCM18));
multiplier fp10 (.a_1(H17),.b_1(Xin),.out_1(MCM17));
multiplier fp12 (.a_1(H16),.b_1(Xin),.out_1(MCM16));
multiplier fp12 (.a_1(H15),.b_1(Xin),.out_1(MCM15));
multiplier fp13 (.a_1(H14),.b_1(Xin),.out_1(MCM14));
multiplier fp14 (.a_1(H13),.b_1(Xin),.out_1(MCM13));
multiplier fp15 (.a_1(H12),.b_1(Xin),.out_1(MCM12));
multiplier fp16 (.a_1(H11),.b_1(Xin),.out_1(MCM11));
multiplier fp17 (.a_1(H10),.b_1(Xin),.out_1(MCM10));
multiplier fp18 (.a_1(H9),.b_1(Xin),.out_1(MCM9));
multiplier fp19 (.a_1(H8),.b_1(Xin),.out_1(MCM8));
multiplier fp20 (.a_1(H7),.b_1(Xin),.out_1(MCM7));
multiplier fp21 (.a_1(H6),.b_1(Xin),.out_1(MCM6));
multiplier fp22 (.a_1(H5),.b_1(Xin),.out_1(MCM5));
multiplier fp23 (.a_1(H4),.b_1(Xin),.out_1(MCM4));
multiplier fp24 (.a_1(H3),.b_1(Xin),.out_1(MCM3));
multiplier fp25 (.a_1(H2),.b_1(Xin),.out_1(MCM2));
multiplier fp26 (.a_1(H1),.b_1(Xin),.out_1(MCM1));
multiplier fp27 (.a_1(H0),.b_1(Xin),.out_1(MCM0));

```

*//adders*

```

adder fadd1 (.a(Q1),.b(MCM25),.out(add_out1));
adder fadd2 (.a(Q2),.b(MCM24),.out(add_out2));

```

```

adder fadd3 (.a(Q3),.b(MCM23),.out(add_out3));
adder fadd4 (.a(Q4),.b(MCM22),.out(add_out4));
adder fadd5 (.a(Q5),.b(MCM21),.out(add_out5));
adder fadd6 (.a(Q6),.b(MCM20),.out(add_out6));
adder fadd7 (.a(Q7),.b(MCM19),.out(add_out7));
adder fadd8 (.a(Q8),.b(MCM18),.out(add_out8));
adder fadd9 (.a(Q9),.b(MCM17),.out(add_out9));
adder fadd10 (.a(Q10),.b(MCM16),.out(add_out10));
adder fadd11 (.a(Q11),.b(MCM15),.out(add_out11));
adder fadd12 (.a(Q12),.b(MCM14),.out(add_out12));
adder fadd13 (.a(Q13),.b(MCM13),.out(add_out13));
adder fadd14 (.a(Q14),.b(MCM12),.out(add_out14));
adder fadd15 (.a(Q15),.b(MCM11),.out(add_out15));
adder fadd16 (.a(Q16),.b(MCM10),.out(add_out16));
adder fadd17 (.a(Q17),.b(MCM9),.out(add_out17));
adder fadd18 (.a(Q18),.b(MCM8),.out(add_out18));
adder fadd19 (.a(Q19),.b(MCM7),.out(add_out19));
adder fadd20 (.a(Q20),.b(MCM6),.out(add_out20));
adder fadd21 (.a(Q21),.b(MCM5),.out(add_out21));
adder fadd22 (.a(Q22),.b(MCM4),.out(add_out22));
adder fadd23 (.a(Q23),.b(MCM3),.out(add_out23));
adder fadd24 (.a(Q24),.b(MCM2),.out(add_out24));
adder fadd25 (.a(Q25),.b(MCM1),.out(add_out25));
adder fadd26 (.a(Q26),.b(MCM0),.out(add_out26));

//Flipflop instantiation (for introducing delay)
DFF dff1 (.Clk(Clk),.D(MCM26),.Q(Q1));
DFF dff2(.Clk(Clk),.D(add_out1),.Q(Q2));

```



```

DFF dff3(.Clk(Clk),.D(add_out2),.Q(Q3));
DFF dff4(.Clk(Clk),.D(add_out3),.Q(Q4));
DFF dff5(.Clk(Clk),.D(add_out4),.Q(Q5));
DFF dff6(.Clk(Clk),.D(add_out5),.Q(Q6));
DFF dff7(.Clk(Clk),.D(add_out6),.Q(Q7));
DFF dff8(.Clk(Clk),.D(add_out7),.Q(Q8));
DFF dff9(.Clk(Clk),.D(add_out8),.Q(Q9));
DFF dff10(.Clk(Clk),.D(add_out9),.Q(Q10));
DFF dff11(.Clk(Clk),.D(add_out10),.Q(Q11));
DFF dff12(.Clk(Clk),.D(add_out11),.Q(Q12));
DFF dff13(.Clk(Clk),.D(add_out12),.Q(Q13));
DFF dff14(.Clk(Clk),.D(add_out13),.Q(Q14));
DFF dff15(.Clk(Clk),.D(add_out14),.Q(Q15));
DFF dff16(.Clk(Clk),.D(add_out15),.Q(Q16));
DFF dff17(.Clk(Clk),.D(add_out16),.Q(Q17));
DFF dff18(.Clk(Clk),.D(add_out17),.Q(Q18));
DFF dff19(.Clk(Clk),.D(add_out18),.Q(Q19));
DFF dff20(.Clk(Clk),.D(add_out19),.Q(Q20));
DFF dff21(.Clk(Clk),.D(add_out20),.Q(Q21));
DFF dff22(.Clk(Clk),.D(add_out21),.Q(Q22));
DFF dff23(.Clk(Clk),.D(add_out22),.Q(Q23));
DFF dff24(.Clk(Clk),.D(add_out23),.Q(Q24));
DFF dff25(.Clk(Clk),.D(add_out24),.Q(Q25));
DFF dff26(.Clk(Clk),.D(add_out25),.Q(Q26));

```

*//Assign the last adder output to the final output*

```
always@ (posedge Clk)
```

```
    Yout <= add_out26;
```

```
endmodule
```

## APPENDIX A.2 Verilog testbench: Low-pass filter

```
// ##### Testbench for the FIR filter: #####

module tb_fir;

    // Inputs

    reg Clk;

    reg signed [31:0] Xin;

    // Outputs

    wire signed [31:0] Yout;

    // Instantiate the Unit Under Test (UUT)

    fir_27tap uut (
        .Clk(Clk),
        .Xin(Xin),
        .Yout(Yout));

    // Generate a clock with a 10ns clock period

    initial Clk = 0;

    always #5 Clk = ~Clk;

    // Initialize and apply the inputs

    initial begin

        Xin = 32'b10100110101100111100100110111011; #270; // -1.24753e-15

        Xin = 32'b001111110001001101010110001100000; #10; // 0.162767

        Xin = 32'b001111110101001011001100011100000; #10; // 0.323432

        Xin = 32'b001111110111101011010100111001000; #10; // 0.479811

        Xin = 32'b001111111001000010011100101100100; #10; // 0.629782

        Xin = 32'b001111111010001010111010100000100; #10; // 0.771317

        Xin = 32'b001111111011001110000101100010100; #10; // 0.902513

        Xin = 32'b001111111100000101100010001110000; #10; // 1.02162
```

Xin =32'b00111111100100000100001100101100; #10;// 1.12705  
 Xin =32'b00111111100110111101010100010010; #10;// 1.21744  
 Xin =32'b00111111101001010101010000100000; #10;// 1.29163  
 Xin =32'b00111111101011001010000100110110; #10;// 1.34867  
 Xin =32'b00111111101100011010011010110100; #10;// 1.3879  
 Xin =32'b00111111101101000101011010000000; #10;// 1.40889  
 Xin =32'b00111111101101001010101001100100; #10;// 1.41145  
 Xin =32'b00111111101100101010011001001100; #10;// 1.3957  
 Xin =32'b00111111101011100101010010110100; #10;// 1.36196  
 Xin =32'b00111111101001111100100111101110; #10;// 1.31085  
 Xin =32'b00111111100111110010000010000100; #10;// 1.24318  
 Xin =32'b00111111100101000111101110001000; #10;// 1.16002  
 Xin =32'b001111111000100000000010010010110; #10;// 1.06264  
 Xin =32'b001111111011100111101010100100000; #10;// 0.952471  
 Xin =32'b001111111010101001100010100000000; #10;// 0.831131  
 Xin =32'b001111111001100110100101011101100; #10;// 0.700362  
 Xin =32'b001111111000011111110000000000100; #10;// 0.562012  
 Xin =32'b00111110110101100000010101101000; #10;// 0.41801  
 Xin =32'b00111110100010100110100010110000; #10;// 0.27033  
 Xin =32'b00111101111101111011100111100000; #10;// 0.12096  
 Xin =32'b10111100111001100110100110000000; #10;// -0.0281265  
 Xin =32'b10111110001100110011001110110000; #10;// -0.175002  
 Xin =32'b10111110101000101011100010100000; #10;// -0.317815  
 Xin =32'b10111110111010001101111001101000; #10;// -0.454822  
 Xin =32'b1011111000101011001101110100000; #10;// -0.584406  
 Xin =32'b1011111001101001000001000010100; #10;// -0.70511  
 Xin =32'b1011111010100001100111010110000; #10;// -0.815654  
 Xin =32'b1011111011010100011101001001000; #10;// -0.914952

```

Xin =32'b10111111100000000100010111001010; #10;//-1.00213
Xin =32'b10111111100010011100110001100010; #10;//-1.07655
Xin =32'b10111111100100011010001011000110; #10;//-1.13778
Xin =32'b10111111100101111100001101100000; #10;//-1.18565
Xin =32'b10111111100111000011000001111110; #10;//-1.22023
Xin =32'b10111111100111101111001011111000; #10;//-1.24179
Xin =32'b10111111101000000001110000101110; #10;//-1.25086
Xin =32'b10111111100111111100010010110000; #10;//-1.24819
Xin =32'b10111111100111100000101011111010; #10;//-1.23471
Xin =32'b10111111100110110001001110111110; #10;//-1.21154
Xin =32'b10111111100101110000100110010100; #10;//-1.17998
Xin =32'b10111111100100100001101100001000; #10;//-1.14145
Xin =32'b10111111100011000111101110001000; #10;//-1.09752
Xin =32'b1011111110000110010111111011000; #10;//-1.0498
Xin =32'b10111111100000000000000000000000; #10;//-1
Xin =32'b101111111011100110010100100011000; #10;//-0.949846
Xin =32'b101111111011001101010110000010000; #10;//-0.901063
Xin =32'b101111111010110101111100000010100; #10;//-0.855348
Xin =32'b101111111010100000111100010000100; #10;//-0.814339
Xin =32'b101111111010001111001001011010000; #10;//-0.779584
Xin =32'b101111111010000001010010011100000; #10;//-0.752516
Xin =32'b101111111001111000000001101000100; #10;//-0.734425
Xin =32'b101111111001110011111011111101000; #10;//-0.726439
Xin =32'b101111111001110101100000010010000; #10;//-0.729501
Xin =32'b10111111100111101000110110100100; #10;//-0.744349
Xin =32'b101111111010001011000000110001100; #10;//-0.771508
Xin =32'b101111111010011111010111110010100; #10;//-0.811273
Xin =32'b101111111010111010001101111010100; #10;//-0.863706

```

```

Xin =32'b101111110110110111011011000000; #10;// -0.928631
Xin =32'b10111111100000001011100011001110; #10;// -1.00564
Xin =32'b10111111100011000000101011010000; #10;// -1.09408
Xin =32'b10111111100110001011011011011000; #10;// -1.19308
Xin =32'b10111111101001101001101000101100; #10;// -1.30158
Xin =32'b10111111101101011000101000110010; #10;// -1.41828
Xin =32'b10111111110001010101011110111100; #10;// -1.54174
Xin =32'b10111111110101011100110110110010; #10;// -1.67034
Xin =32'b1011111111001101011001110111010; #10;// -1.80236
Xin =32'b1011111111101111100110010001100; #10;// -1.93593
Xin =32'b1100000000001000110110011001010; #10;// -2.06914
Xin =32'b1100000000011001100110100100000; #10;// -2.20002
Xin =32'b1100000000101001110011011011001; #10;// -2.32659
Xin =32'b1100000000111001001100111011000; #10;// -2.44689
Xin =32'b11000000001000111100011001010100; #10;// -2.55898
Xin =32'b11000000001010100100111001010000; #10;// -2.66103
Xin =32'b1100000000110000001010100100010; #10;// -2.75129
Xin =32'b11000000001101010000000011100110; #10;// -2.82818
Xin =32'b11000000001110001111100111011011; #10;// -2.89025
Xin =32'b11000000001110111110101101011011; #10;// -2.93624
Xin =32'b11000000001111011100010000110010; #10;// -2.9651
Xin =32'b11000000001111100111011011110010; #10;// -2.97601
Xin =32'b11000000001111011111100111110000; #10;// -2.96838
Xin =32'b11000000001111000100011110011001; #10;// -2.94187
Xin =32'b11000000001110010101111011001000; #10;// -2.89641
Xin =32'b11000000001101010100001000011100; #10;// -2.83216
Xin =32'b11000000001011111111100011001010; #10;// -2.74956
Xin =32'b11000000001010011000111001110101; #10;// -2.64932

```

```

Xin =32'b11000000001000100001001000000101; #10;// -2.53235
Xin =32'b11000000000110011001011100100100; #10;// -2.39985
Xin =32'b11000000000100000011010000011001; #10;// -2.25318
Xin =32'b11000000000001100000001100011100; #10;// -2.09394
Xin =32'b10111111111101100100001000000110; #10;// -1.92389
Xin =32'b10111111110111110101101010000100; #10;// -1.74495
Xin =32'b10111111110001111001001000111010; #10;// -1.55915
Xin =32'b10111111101011110010111101000100; #10;// -1.36863
Xin =32'b10111111100101100111100100010010; #10;// -1.17557
Xin =32'b10111111011110110111001001001100; #10;// -0.982213
Xin =32'b10111111010010100111000011000000; #10;// -0.790783
Xin =32'b10111111000110100111110101010100; #10;// -0.603475
Xin =32'b10111110110110000100011110110000; #10;// -0.422422
Xin =32'b10111110011111111010011110100000; #10;// -0.249663
Xin =32'b10111101101100100110011110000000; #10;// -0.0871116
Xin =32'b00111101100000011111101111000000; #10;// 0.0634686
Xin =32'b00111110010011010100110101000000; #10;// 0.20049
Xin =32'b00111110101001010010011011111000; #10;// 0.322563
Xin =32'b00111110110110110110011001010000; #10;// 0.428515
Xin =32'b00111111000001000111010011101000; #10;// 0.517409
Xin =32'b00111111000101101010101100110100; #10;// 0.58855
Xin =32'b00111111001001000011100101011000; #10;// 0.6415
Xin =32'b00111111001011010001001110010100; #10;// 0.67608
Xin =32'b00111111001100010011111011110100; #10;// 0.692367
Xin =32'b00111111001100001101000110010100; #10;// 0.690698
Xin =32'b00111111001010111111000111000100; #10;// 0.671658
Xin =32'b00111111001000101101010101111000; #10;// 0.63607
Xin =32'b00111111000101011100000100101100; #10;// 0.584979

```

```

Xin =32'b00111111000001010000011100001100; #10;// 0.519639
Xin =32'b00111110111000100000101001010000; #10;// 0.441485
Xin =32'b00111110101101000100100010001000; #10;// 0.352116
Xin =32'b00111110100000011010101111110000; #10;// 0.253265
Xin =32'b00111110000101100100101011010000; #10;// 0.14677
Xin =32'b00111101000011011000001010000000; #10;// 0.0345485
Xin =32'b10111101101001101100011111100000; #10;// -0.081436
Xin =32'b10111110010010111111110001100000; #10;// -0.199205
Xin =32'b10111110101000100011001011110000; #10;// -0.316795
Xin =32'b10111110110111010101010011111000; #10;// -0.432289
Xin =32'b10111111000010110011100110001100; #10;// -0.543847
Xin =32'b10111111001001100101010011100100; #10;// -0.649733
Xin =32'b10111111001111111001001101101000; #10;// -0.748343
Xin =32'b10111111010101101001011000011100; #10;// -0.838228
Xin =32'b10111111011010110000100110110100; #10;// -0.918117
Xin =32'b10111111011111001010011111000100; #10;// -0.986935
Xin =32'b10111111100001011001101111100100; #10;// -1.04382
Xin =32'b10111111100010110100011111011000; #10;// -1.08813
Xin =32'b10111111100011110100101011001010; #10;// -1.11947
Xin =32'b10111111100100011001111011010110; #10;// -1.13766
Xin =32'b10111111100100100100011001001000; #10;// -1.14277
Xin =32'b10111111100100010100101011110100; #10;// -1.1351
Xin =32'b10111111100011101011111000110110; #10;// -1.11518
Xin =32'b10111111100010101011100010100100; #10;// -1.08376
Xin =32'b10111111100001010101100101011110; #10;// -1.04179
Xin =32'b10111111011111011000101011111100; #10;// -0.990402
Xin =32'b10111111011011100100111100010000; #10;// -0.930894
Xin =32'b10111111010111010101110101011100; #10;// -0.864706

```

Xin =32'b10111111010010110001101111101100; #10;// -0.793395  
Xin =32'b10111111001101111111011001111100; #10;// -0.718605  
Xin =32'b10111111001001000101110010111000; #10;// -0.64204  
Xin =32'b10111111000100001100000001001000; #10;// -0.565434  
Xin =32'b101111101111110110010010110110000; #10;// -0.490522  
Xin =32'b10111110110101101000100001011000; #10;// -0.419009  
Xin =32'b1011111010110100011111111111000; #10;// -0.352539  
Xin =32'b10111110100101011101100001110000; #10;// -0.292667  
Xin =32'b10111110011101101001110100110000; #10;// -0.240834  
Xin =32'b10111110010010110001100011010000; #10;// -0.198337  
Xin =32'b10111110001010100100110101100000; #10;// -0.166311  
Xin =32'b10111110000101010011001101100000; #10;// -0.145704  
Xin =32'b10111110000011001000111001100000; #10;// -0.137262  
Xin =32'b10111110000100001110100110010000; #10;// -0.141516  
Xin =32'b10111110001000101001001111010000; #10;// -0.158767  
Xin =32'b10111110010000011001111010110000; #10;// -0.189082  
Xin =32'b10111110011011011101111000110000; #10;// -0.232293  
Xin =32'b10111110100100110111001110101000; #10;// -0.287992  
Xin =32'b10111110101101100000100100110000; #10;// -0.355539  
Xin =32'b10111110110111100011111011010000; #10;// -0.434073  
Xin =32'b10111111000001011100001110101100; #10;// -0.522517  
Xin =32'b10111111000111101001111000011000; #10;// -0.6196  
Xin =32'b10111111001110010100111111001100; #10;// -0.723874  
Xin =32'b10111111010101010110111110000100; #10;// -0.833733  
Xin =32'b10111111011100101000101101101100; #10;// -0.94744  
Xin =32'b1011111100010000001010110100000; #10;// -1.06316  
Xin =32'b1011111100101101110100000101000; #10;// -1.17896  
Xin =32'b1011111101001010111110101101010; #10;// -1.29289



```

Xin =32'b10111111101100111001010010000100; #10;// -1.40297
Xin =32'b10111111110000001110110011101000; #10;// -1.50723
Xin =32'b10111111110011010100100000000000; #10;// -1.60376
Xin =32'b10111111110110000110100100101110; #10;// -1.69071
Xin =32'b10111111111000100001100000010100; #10;// -1.76636
Xin =32'b10111111111010100001111110011110; #10;// -1.82909
Xin =32'b1011111111100000101000101000010; #10;// -1.87748
Xin =32'b10111111111101001000010000001110; #10;// -1.91028
Xin =32'b10111111111101101001010101000010; #10;// -1.92643
Xin =32'b10111111111101100110101000000000; #10;// -1.92511
Xin =32'b10111111111100111110111011110100; #10;// -1.90573
Xin =32'b101111111111011110001100110100100; #10;// -1.86797
Xin =32'b101111111111001111110011011000100; #10;// -1.81173
Xin =32'b101111111110111100101110010010000; #10;// -1.7372
Xin =32'b101111111110100101000100101110110; #10;// -1.64482
Xin =32'b101111111110001001000010000001110; #10;// -1.53528
Xin =32'b101111111101101000110101011010010; #10;// -1.40951
Xin =32'b101111111101000100110010001101110; #10;// -1.26869
Xin =32'b10111111100011101001111000011010; #10;// -1.1142
Xin =32'b101111111011100101001011111110000; #10;// -0.947631
Xin =32'b101111111010001010100111110111100; #10;// -0.770748
Xin =32'b101111111000101011110000011010100; #10;// -0.585462
Xin =32'b1011111110110010011010000110011000; #10;// -0.393811
Xin =32'b10111110010010101010110011010000; #10;// -0.197925
Xin =32'b10100111010110111100100110110100; #10;// -3.05017e-15
Xin =32'b00111110010010100111101110010000; #10;// 0.197737
Xin =32'b00111110110010010011111100101000; #10;// 0.39306
Xin =32'b00111111000101010111001000100100; #10;// 0.583773

```

Xin =32'b00111111010001001000101100000000; #10;// 0.767746  
Xin =32'b00111111011100010110010010110100; #10;// 0.942943  
Xin =32'b00111111100011011100000011101010; #10;// 1.10745  
Xin =32'b00111111101000010011011110011110; #10;// 1.25951  
Xin =32'b00111111101100101110001001000010; #10;// 1.39753  
Xin =32'b00111111110000101001001101001010; #10;// 1.52012  
Xin =32'b00111111110100000010010100000110; #10;// 1.62613  
Xin =32'b00111111110110110111100001010110; #10;// 1.71461  
Xin =32'b00111111111001000111011010011110; #10;// 1.78487  
Xin =32'b00111111111010110001000101110010; #10;// 1.83647  
Xin =32'b00111111111011110100001110010100; #10;// 1.86925  
Xin =32'b0011111111100010000111011111100; #10;// 1.88327  
Xin =32'b00111111111100000111111101110110; #10;// 1.87889  
Xin =32'b00111111111011011010100000000100; #10;// 1.85669  
Xin =32'b00111111111010001010001111010110; #10;// 1.8175  
Xin =32'b00111111111000011001010100000010; #10;// 1.76236  
Xin =32'b00111111110110001010010100100110; #10;// 1.69254  
Xin =32'b00111111110011100000001100011100; #10;// 1.60947  
Xin =32'b00111111110000011110001111111010; #10;// 1.51477  
Xin =32'b00111111110110100011111111001010; #10;// 1.41015  
Xin =32'b001111111101001100001001111010010; #10;// 1.29748  
Xin =32'b001111111100101101101111010101000; #10;// 1.17867  
Xin =32'b001111111100001110010000011011000; #10;// 1.05569  
Xin =32'b00111111011011100011011011100000; #10;// 0.930525  
Xin =32'b00111111010011100001110111001000; #10;// 0.805142  
Xin =32'b00111111001011100111010010001100; #10;// 0.681466  
Xin =32'b00111111000011111011010000001000; #10;// 0.561341  
Xin =32'b001111101111001001001110100010000; #10;// 0.446511

```

Xin =32'b00111110101011010101100000; #10;// 0.338584
Xin =32'b00111110011101001100000011010000; #10;// 0.239017
Xin =32'b00111110000110001010101000000000; #10;// 0.149086
Xin =32'b00111101100011110001100101000000; #10;// 0.0698726
Xin =32'b00111011000100110011000000000000; #10;// 0.00224591
Xin =32'b10111101010110011011001001000000; #10;// -0.0531486
Xin =32'b10111101110001000110011000000000; #10;// -0.0958977
Xin =32'b10111110000000001101100001000000; #10;// -0.125825
Xin =32'b10111110000100100110110000110000; #10;// -0.142991
Xin =32'b10111110000101110011110010010000; #10;// -0.147692
Xin =32'b10111110000011111101001000010000; #10;// -0.14045
Xin =32'b10111101111111001110111001010000; #10;// -0.122003
Xin =32'b10111101101111110001000101000000; #10;// -0.0932948
Xin =32'b10111101011000110001111110000000; #10;// -0.0554502
Xin =32'b10111100000111111101011000000000; #10;// -0.00976067
Xin =32'b00111101001011010110111001000000; #10;// 0.0423417
Xin =32'b00111101110010110110001001100000; #10;// 0.0993089
Xin =32'b00111110001000110101010100000000; #10;// 0.159504
Xin =32'b00111110011000101000101010100000; #10;// 0.221232
Xin =32'b00111110100100001100011001110000; #10;// 0.282764
Xin =32'b00111110101011110100101111101000; #10;// 0.342376
Xin =32'b00111110110010111111011101001000; #10;// 0.398371
Xin =32'b00111110111001011111001000000000; #10;// 0.449112
Xin =32'b00111110111111000111000101110000; #10;// 0.493053
Xin =32'b00111111000001110101110011100000; #10;// 0.528761
Xin =32'b00111111000011100001000100010000; #10;// 0.554948
Xin =32'b00111111000100100000101101101100; #10;// 0.570487
Xin =32'b00111111000100110000111001011100; #10;// 0.574438

```

```

Xin =32'b00111111000100001110100110100000; #10;// 0.566065
Xin =32'b00111111000010110111101011110100; #10;// 0.544845
Xin =32'b00111111000000101010111100110100; #10;// 0.510486
Xin =32'b0011111011101101010000010011100000; #10;// 0.462928
Xin =32'b00111110110011100000000011010000; #10;// 0.40235
Xin =32'b00111110101010001000100010010000; #10;// 0.329167
Xin =32'b00111110011110011110001010110000; #10;// 0.244029
Xin =32'b00111110000101110101101000110000; #10;// 0.147805
Xin =32'b00111101001010100101000101000000; #10;// 0.0415815
Xin =32'b10111101100101100011111100100000; #10;// -0.0733626
Xin =32'b10111110010010000100001100110000; #10;// -0.195569
Xin =32'b10111110101001011001011111010000; #10;// -0.323424
Xin =32'b10111110111010010000110110011000; #10;// -0.455182
Xin =32'b10111111000101101100100001001100; #10;// -0.588994
Xin =32'b10111111001110010001001000100000; #10;// -0.722933
Xin =32'b10111111010110101110001011111000; #10;// -0.855026
Xin =32'b10111111011110111011100010100000; #10;// -0.983286
Xin =32'b10111111100011011000100011100010; #10;// -1.10574
Xin =32'b10111111100111000011100000001000; #10;// -1.22046
Xin =32'b10111111101010011010110011101110; #10;// -1.32559
Xin =32'b10111111101101011010111100111010; #10;// -1.41941
Xin =32'b10111111110000000001001100000000; #10;// -1.50029
Xin =32'b10111111110010001000110010010010; #10;// -1.56679
Xin =32'b10111111110011110000111101111010; #10;// -1.61766
Xin =32'b10111111110100110110111101101010; #10;// -1.65182
Xin =32'b10111111110101011001000000011000; #10;// -1.66846
Xin =32'b10111111110101010101111110011000; #10;// -1.66698
Xin =32'b10111111110100101101000111100000; #10;// -1.64703

```

```

Xin =32'b10111111110011011110010001001110; #10;// -1.60853
Xin =32'b10111111110001101001110000100010; #10;// -1.55164
Xin =32'b101111111101111010000011101110100; #10;// -1.47679
Xin =32'b101111111101100010011110100110000; #10;// -1.38468
Xin =32'b101111111101000110101101100101100; #10;// -1.27622
Xin =32'b101111111100100111000011110111100; #10;// -1.15258
Xin =32'b101111111100000011111000000011010; #10;// -1.01514
Xin =32'b1011111111010111011001000001011100; #10;// -0.865484
Xin =32'b1011111111001101001001001010001000; #10;// -0.705361
Xin =32'b1011111111000010010110001110111000; #10;// -0.536678
Xin =32'b10111111110101110010001000110101000; #10;// -0.361463
Xin =32'b10111111110001110100011010010000000; #10;// -0.181841
Xin =32'b1010011101111111111110100100111110; #10;// -3.55148e-15
Xin =32'b001111110001110100011010010000000; #10;// 0.181841
Xin =32'b001111110101110010001000110101000; #10;// 0.361463
Xin =32'b001111111000010010110001110111000; #10;// 0.536678
Xin =32'b001111111001101001001001010001000; #10;// 0.705361
Xin =32'b001111111010111011001000001011100; #10;// 0.865484
Xin =32'b001111111100000011111000000011010; #10;// 1.01514
Xin =32'b001111111100100111000011110111100; #10;// 1.15258
Xin =32'b001111111101000110101101100101100; #10;// 1.27622
Xin =32'b001111111101100010011110100110000; #10;// 1.38468
Xin =32'b001111111101111010000011101110100; #10;// 1.47679
Xin =32'b00111111110001101001110000100010; #10;// 1.55164
Xin =32'b00111111110011011110010001001110; #10;// 1.60853
Xin =32'b00111111110100101101000111100000; #10;// 1.64703
Xin =32'b00111111110101010101111110011000; #10;// 1.66698
Xin =32'b00111111110101011001000000011000; #10;// 1.66846

```

```

Xin =32'b00111111110100110110111011010110; #10;// 1.65182
Xin =32'b00111111110011110000111101111010; #10;// 1.61766
Xin =32'b00111111110010001000110010010010; #10;// 1.56679
Xin =32'b00111111110000000000100110000000; #10;// 1.50029
Xin =32'b001111111101101011010111100111010; #10;// 1.41941
Xin =32'b001111111101010011010110011101110; #10;// 1.32559
Xin =32'b001111111100111000011100000001000; #10;// 1.22046
Xin =32'b001111111100011011000100011100010; #10;// 1.10574
Xin =32'b0011111111011110111011100010100000; #10;// 0.983286
Xin =32'b0011111111010110101110001011111000; #10;// 0.855026
Xin =32'b0011111111001110010001001000100000; #10;// 0.722933
Xin =32'b0011111111000101101100100001001100; #10;// 0.588994
Xin =32'b001111111101111010010000110110011000; #10;// 0.455182
Xin =32'b00111111110101001011001011111010000; #10;// 0.323424
Xin =32'b00111111110010010000100001100110000; #10;// 0.195569
Xin =32'b001111111101100101100011111100100000; #10;// 0.0733626
Xin =32'b1011111010010101001010001010000000; #10;// -0.0415815
Xin =32'b10111110000101110101101000110000; #10;// -0.147805
Xin =32'b10111110011110011110001010110000; #10;// -0.244029
Xin =32'b10111110101010001000100010010000; #10;// -0.329167
Xin =32'b10111110110011100000000011010000; #10;// -0.40235
Xin =32'b10111110111011011010000010011100000; #10;// -0.462928
Xin =32'b10111111000000101010111100110100; #10;// -0.510486
Xin =32'b10111111000010110111101011110100; #10;// -0.544845
Xin =32'b10111111000100001110100110100000; #10;// -0.566065
Xin =32'b10111111000100110000111001011100; #10;// -0.574438
Xin =32'b10111111000100100000101101101100; #10;// -0.570487
Xin =32'b10111111000011100001000100010000; #10;// -0.554948

```

Xin =32'b10111111000001110101110011100000; #10;// -0.528761  
Xin =32'b10111110111111000111000101110000; #10;// -0.493053  
Xin =32'b10111110111001011111001000000000; #10;// -0.449112  
Xin =32'b10111110110010111111011101001000; #10;// -0.398371  
Xin =32'b10111110101011110100101111101000; #10;// -0.342376  
Xin =32'b10111110100100001100011001110000; #10;// -0.282764  
Xin =32'b10111110011000101000101010100000; #10;// -0.221232  
Xin =32'b10111110001000110101010100000000; #10;// -0.159504  
Xin =32'b10111101110010110110001001100000; #10;// -0.0993089  
Xin =32'b10111101001011010110111001000000; #10;// -0.0423417  
Xin =32'b00111100000111111110101100000000; #10;// 0.00976067  
Xin =32'b00111101011000110001111110000000; #10;// 0.0554502  
Xin =32'b00111101101111110001000101000000; #10;// 0.0932948  
Xin =32'b00111101111111001110111001010000; #10;// 0.122003  
Xin =32'b001111100000111111101001000010000; #10;// 0.14045  
Xin =32'b00111110000101110011110010010000; #10;// 0.147692  
Xin =32'b00111110000100100110110000110000; #10;// 0.142991  
Xin =32'b00111110000000001101100001000000; #10;// 0.125825  
Xin =32'b00111101110001000110011000000000; #10;// 0.0958977  
Xin =32'b00111101010110011011001001000000; #10;// 0.0531486  
Xin =32'b10111011000100110011000000000000; #10;// -0.00224591  
Xin =32'b10111011000111100011001010000000; #10;// -0.0698726  
Xin =32'b10111100001100010101010000000000; #10;// -0.149086  
Xin =32'b1011110011101001100000011010000; #10;// -0.239017  
Xin =32'b1011110101011010101101011100000; #10;// -0.338584  
Xin =32'b1011110111001001001110100010000; #10;// -0.446511  
Xin =32'b1011111000011111011010000001000; #10;// -0.561341  
Xin =32'b1011111001011100111010010001100; #10;// -0.681466

```

Xin =32'b10111111010011100001110111001000; #10;//-0.805142
Xin =32'b10111111011011100011011011100000; #10;//-0.930525
Xin =32'b10111111100001110010000011011000; #10;//-1.05569
Xin =32'b10111111100101101101111010101000; #10;//-1.17867
Xin =32'b10111111101001100001001111010010; #10;//-1.29748
Xin =32'b1011111110110100011111111001010; #10;//-1.41015
Xin =32'b10111111110000011110001111111010; #10;//-1.51477
Xin =32'b10111111110011100000001100011100; #10;//-1.60947
Xin =32'b10111111110110001010010100100110; #10;//-1.69254
Xin =32'b10111111111000011001010100000010; #10;//-1.76236
Xin =32'b10111111111010001010001111010110; #10;//-1.8175
Xin =32'b10111111111011011010100000000100; #10;//-1.85669
Xin =32'b1011111111100000111111101110110; #10;//-1.87889
Xin =32'b1011111111100010000111011111100; #10;//-1.88327
Xin =32'b10111111111011110100001110010100; #10;//-1.86925
Xin =32'b10111111111010110001000101110010; #10;//-1.83647
Xin =32'b10111111111001000111011010011110; #10;//-1.78487
Xin =32'b10111111110110110111100001010110; #10;//-1.71461
Xin =32'b10111111110100000010010100000110; #10;//-1.62613
Xin =32'b10111111110000101001001101001010; #10;//-1.52012
Xin =32'b101111111101100101110001001000010; #10;//-1.39753
Xin =32'b101111111101000010011011110011110; #10;//-1.25951
Xin =32'b101111111100011011100000011101010; #10;//-1.10745
Xin =32'b10111111011100010110010010110100; #10;//-0.942943
Xin =32'b10111111010001001000101100000000; #10;//-0.767746
Xin =32'b10111111000101010111001000100100; #10;//-0.583773
Xin =32'b101111110110010010011111100101000; #10;//-0.39306
Xin =32'b101111110010010100111101110010000; #10;//-0.197737

```



```

        Xin =32'b1010011111001111011010111101011011; #10;//-4.40873e-15

    end

endmodule

```

## APPENDIX B.1: MATLAB Code: Low-pass filter

```

f1=200;

f2=300;                                %the frequencies of sines signal that needs filtered

f3=1000;

f4=2000;

fc=500;

TW=800;

fs=3500.0;

M=ceil(5.98*(fs/TW));                  %define the window length

if rem(M, 2) == 1

M=M;

else

M=M+1;

end

N=M-1;                                %define the order of filter

b = fir1(N,fc/(fs/2),'low',blackman(M));

[h,f]=freqz(b,1,512);

figure(1)

plot(f*fs/(2*pi),20*log10(abs(h)))      %frequency and amplitude parameters respectively

xlabel('frequency/Hz');

ylabel('gain/dB');

title('The gain response of low-pass filter');

t=(0:400)/fs;                          %time domain and the step length

t1=(0.002:0.00001:0.006);

```

```

s=sin(2*pi*f1*t)+sin(2*pi*f2*t)+sin(2*pi*f3*t)+sin(2*pi*f4*t);           %unfiltered signal
s1=sin(2*f1*pi*t1)+sin(2*f2*pi*t1)+sin(2*f3*pi*t1)+sin(2*f4*pi*t1);

sf=filter(b,1,s);

figure(2)
subplot(211)
plot(t1,s1);
xlabel('time/s');
ylabel('amplitude');
title('Time-domain diagram before filtering');
subplot(212)
Fs=fft(s,512);                                                            %transform the signal to frequency domain
AFs=abs(Fs);                                                                %take the amplitude
f=(0:255)*fs/512;                                                            %frequency sampling
plot(f,AFs(1:256));                                                        %plot the frequency domain diagram before filtering
xlabel('frequency/Hz');
ylabel('amplitude');
title('Frequency-domain diagram before filtering');
figure(3)
subplot(211)
plot(t,sf)                                                                  %plot the signal graph after filtering
xlabel('time/s');
ylabel('amplitude');
title('Time-domain diagram after filtering');
axis([0.005 0.025 -4 4]);
subplot(212)
Fsf=fft(sf,512);                                                            %frequency-domain diagram after filtering
AFsf=abs(Fsf);                                                                %the amplitude
f=(0:255)*fs/512;                                                            %frequency sampling

```

```
plot(f,AFsf(1:256))           %plot the frequency domain diagram after filtering  
xlabel('frequency/Hz');  
ylabel('amplitude');  
title('Frequency-domain diagram after filtering');
```

## REFERENCES

- [1] V. Berman, "Two approaches to standardization and language design," *IEEE*, vol. 22, no. 3, pp. 283-285, 2005.
- [2] M. Cheng, "Foundry Future: challenges in the 21st century," *IEEE*, pp. 18-23, 2007.
- [3] Foster H., "The 2018 Wilson Research Group Functional Verification Study," Mentor Graphics Corporation, 2018.
- [4] E. Tronci, "Special Section on Recent Advances in Hardware Verification: introductory paper," *International Journal on Software Tools for Technology Transfer*, vol. 8, pp. 355-358, 2006.
- [5] D. Packer, "Verification vs. Validation: Do You know the Difference? " 7 April 2019. [Online]. Available: <https://www.plutora.com/blog/verification-vs-validation>. [Accessed 25 March, 2020].
- [6] J. Bergeron, Writing testbenches: functional verification of HDL models, Norwell, MA: Kluwer Academic Publishers, 2000.
- [7] N. H. Weste and D. M. Harris, "Cell-Based Design," in *CMOS VLSI Design: a circuits and systems perspective*, USA, Addison-Wesley, 2010, pp. 632-634.
- [8] D. Josephson and B. Gottlieb, "Silicon Debug," in *Advances in Electronic Testing: Challenges and Methodologies*, Springer, 2006, pp. 77-79.
- [9] A. Molina and O. Cadenas, "Functional Verification: approaches and challenges," *Latin American applied research*, vol. 37, 2007.

- [10] M. S. Abadir, K. L. Albin, J. Havlicek, N. Krishnamurthy, and A. K. Martin, "Formal Verification Successes at Motorola," *Formal Methods in System Design*, vol. 2, no. 22, pp. 117-123, 2003.
- [11] A. Piziali, "Design Intent Diagram," in *Functional Verification Coverage Measurement and Analysis*, New York, USA, Springer, 2004, pp. 16-18.
- [12] B. Bailey, "A new vision of 'scalable' verification," 19 March 2004. [Online]. Available: <https://www.eetimes.com/a-new-vision-of-scalable-verification/#>. [Accessed 14 May, 2020].
- [13] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore and F. Bruno, "Complete Formal Verification of TriCore2 and Other Processors," in *Design and Verification Conference and Exhibition*, San Jose, 2007.
- [14] C. Yan and K. Jones, "Efficient Simulation-Based Verification by Reordering," in *Design and Verification Conference and Exhibition*, San Jose, CA, 2010.
- [15] S. W. Smith, "Filter Basics," in *The Scientist and Engineer's Guide to Digital Signal Processing*, San Diego, California, California Technical Publishing, 1999, pp. 261-265.
- [16] J. Heath, "Analog filters vs. digital filters," 3 October 2016. [Online]. Available: <https://www.analogictips.com/using-analog-filters-vs-digital-filters/>. [Accessed 26 May, 2020].
- [17] N. Davis, "An introduction to filters," 31 July 2017. [Online]. Available: <https://www.allaboutcircuits.com/technical-articles/an-introduction-to-filters/>. [Accessed 26 May, 2020].

- [18] Siemens, "Introduction to Filters: FIR versus IIR," 6 April 2020. [Online]. Available: <https://community.sw.siemens.com/s/article/introduction-to-filters-fir-versus-iir>. [Accessed 26 May, 2020].
- [19] J. R. Raol, G. Gopalratnam, and B. Twala, "Dynamic System Models and Basic Concepts," in *Nonlinear Filtering Concepts and Engineering Applications*, Boca Raton, CRC Press, 2017, pp. 3-14.
- [20] S. Mokhatab and W. A. Poe, "Finite Impulse Response Models," in *Process Modeling in the Natural Gas Processing Industry*, Waltham, MA, Gulf Professional Publishing, 2012, pp. 533-534.
- [21] N. Kumar, "Optimal Design of FIR and IIR Filters using some Evolutionary Algorithms," Durgapur, INDIA, 2013.
- [22] Z. Milivojević, "Finite impulse response (FIR) filter design methods," in *Digital Filter Design*, mikroElektronika, 2009.
- [23] Z. Milivojevic, "Window functions," in *Digital Filter Design*, mikroElektronika, 2009.
- [24] R. Keim, "What Is a Hardware Description Language (HDL)?," 11 March 2020. [Online]. Available: <https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/>. [Accessed 3 June, 2020].
- [25] S. Palnitkar, "Overview of Digital Design with Verilog HDL," in *Verilog HDL: a guide to digital design and synthesis*, Palo Alto, California, USA, Sun Microsystems Press, 2003, pp. 3-8.

- [26] J. W. Tantra, "Experiences in Building Python Automation Framework for Verification and Data Collections," in *Proceedings of PyCon Asia-Pacific*, 2010.
- [27] A. Aggarwal, "Introduction to Perl," [Online]. Available: <https://www.geeksforgeeks.org/introduction-to-perl/>. [Accessed 7 June, 2020].
- [28] M. Dehbashi, A. Sülflow, and G. Fey, "Automated Design Debugging in a Testbench-Based," in *14th Euromicro Conference on Digital System Design*, Oulu, Finland, 2011.
- [29] S. Kumar, Mohanty, S. Sengupta, and S. K. Mohapatra, "Test Bench Automation to overcome Verification Challenge of SOC Interconnect," in *2015 International Conference on Man and Machine Interfacing (MAMI)*, Bhubaneswar, 2015.
- [30] I. Maia, K. R. G. d. Silva, L. Max, and R. C. P. Camara, "eTBc: A Semi-Automatic Testbench Generation Tool," in *IP SOC*, French, 2007.
- [31] M. Lajolo, L. Lavagno, and M. Rebaudengo, "Automatic Test Bench Generation for Simulation-based Validation," in *M. Lajolo, L. Lavagno, and M. Rebaudengo, "Automatic test bench generProceedings of the Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, USA, 2000.
- [32] S. N. Pagliarini and F. L. Kastensmidt, *VEasy: a Tool Suite for Teaching Functional Verification*, Germany: LAP LAMBERT Academic Publishing, 2012.

- [33] V. Verma and C. Chien, "A VHDL based Functional Compiler for Optimum Architecture Generation of FIR filters," *IEEE International Symposium*, vol. 4, pp. 564-567, 1996.
- [34] Xilinx Inc., "LogiCORE IP FIR Compiler v6.3," 19 October 2011. [Online]. Available:  
  
[https://www.xilinx.com/support/documentation/ip\\_documentation/fir\\_compiler/v6\\_3/ds795\\_fir\\_compiler.pdf](https://www.xilinx.com/support/documentation/ip_documentation/fir_compiler/v6_3/ds795_fir_compiler.pdf). [Accessed 21 June, 2020].
- [35] MathWorks Inc., "HDL Coder user's guide," March 2020. [Online]. Available:  
  
[https://ww2.mathworks.cn/help/pdf\\_doc/hdlcoder/hdlcoder\\_ug.pdf](https://ww2.mathworks.cn/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf). [Accessed 21 June, 2020].
- [36] D. N. Bogdan Sbarcea, "Automatic Conversion of MatLab/Simulink Models to HDL Models," [Online]. Available: <http://fcd.co.il/doc/optim2004.pdf>. [Accessed 21 June, 2020].
- [37] K. Camera, "SF2VHD: A Stateflow to VHDL Translator," 2001.
- [38] M. Bayasgalan and X.-E. Sun, "Graphical User Interface Design of FIR Filter," *Open Access Library*, vol. 5, 2018.
- [39] Z. XueMin, "Design and Simulation of FIR Filter Based on GUI," in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, Taiyuan, China, 2010.
- [40] V. S. Rosa, F. F. Daitx, E. Costa, and S. Bampi, "Design Flow for the Generation of Optimized FIR Filters," in *16th IEEE International Conference*, Banff, Canada, 2009.



- [41] R. Chadwick, "Binary Tutorial - 5. Binary Fractions and Floating Point," 2020.  
[Online]. Available: <https://ryanstutorials.net/binary-tutorial/binary-floating-point.php>. [Accessed 11 June, 2020].