3 – Tap FIR (Finite Impulse Response) Filter using Verilog HDL

Abstract

This project implements a Finite Impulse Response (FIR) filter in Verilog, utilizing Braun Multipliers for efficient multiplication, 16-bit Ripple Carry Adders for summation, and D FlipFlops for data storage. The filter processes discrete input signals with predefined coefficients, providing the filtered output. Simulation results confirm the system's functionality, demonstrating correct operation with minimal delay. The design offers a hardware-efficient solution for signal processing applications, with potential for further optimization.

Introduction

Finite Impulse Response (FIR) filters are widely used in digital signal processing (DSP) due to their stability, linear phase characteristics, and ease of implementation. These filters are used to process discrete-time signals by applying a set of predefined coefficients to the input signal, producing a filtered output. The primary advantage of FIR filters lies in their simplicity and predictable performance, making them essential in a variety of applications such as audio processing, communication systems, and image filtering.

This report discusses the theoretical background behind the FIR filter, the detailed design and implementation using Verilog, the testing and simulation process, and the performance analysis of the designed system. The primary objective of this work is to demonstrate an efficient way of implementing an FIR filter in hardware, emphasizing both functionality and hardware optimization.

Theory

1. Basics of FIR Filter

A Finite Impulse Response (FIR) filter is a type of digital filter used to process discrete-time signals. Unlike Infinite Impulse Response (IIR) filters, FIR filters are characterized by a finite number of taps or coefficients that define their impulse response. The output of an FIR filter is a weighted sum of a finite number of past input values, represented as:

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

where:

- y[n]y[n] is the output signal at time nn,
- x[n−k]x[n-k] is the input signal at time n−kn-k,
- h[k]h[k] is the filter coefficient for tap kk,
- MM is the number of filter taps.

The FIR filter is non-recursive, meaning it only relies on past and present input values, making it inherently stable and having a linear phase response.

2. Braun Multiplier

The Braun multiplier is a type of parallel multiplier used to multiply two binary numbers. It performs multiplication by generating partial products and adding them with proper shifts. In the design of this FIR filter, Braun multipliers are used to multiply the input signal by the filter coefficients (H0, H1, H2). The main advantage of the Braun multiplier is its simple and efficient hardware implementation, which allows for fast multiplication of binary numbers.

For two 8-bit inputs aa and bb, the Braun multiplier generates a 16-bit product by summing the partial products with appropriate bit shifts.

3. Ripple Carry Adder (RCA)

The Ripple Carry Adder (RCA) is a basic binary adder that adds two binary numbers bit by bit, propagating the carry from one bit to the next. In this FIR filter design, a 16-bit RCA is used to sum the products generated by the Braun multipliers. The RCA consists of multiple full adders, each responsible for adding corresponding bits of the two numbers and the carry from the previous stage. While slower than more advanced adder architectures (like carry-lookahead adders), the RCA is simpler and easier to implement in hardware.

Each full adder in the RCA adds two input bits along with a carry-in bit and produces a sum and a carry-out bit. The carry-out from each adder is passed to the next adder, resulting in the final sum.

4. D – Flip Flop

A D Flip-Flop (DFF) is a type of sequential logic circuit that stores a single bit of data. It has two primary inputs: a data input (D) and a clock input (clk). The DFF captures the input value on the rising edge of the clock and outputs it on the Q output. The DFF in this design is used to store intermediate values in the FIR filter to ensure correct timing and sequencing of operations. By storing the values of partial sums at each clock cycle, the filter can accumulate the results from multiple stages, ensuring that the output is correctly computed based on the filter's coefficients.

System Architecture

The architecture of the FIR filter design is based on a modular structure where different functional blocks are interconnected to process the input signal and produce the filtered output. The system comprises several key components: the **Braun Multiplier**, **Ripple Carry Adder (RCA)**, **D Flip-Flops**, and the **Top-Level FIR Filter Module**. Below is a detailed breakdown of the system’s architecture:

1. Top-Level Design Overview (FIR_Top Module)

The FIR filter is implemented as a top-level module, FIR_Top, which integrates all the submodules. This module takes a clock signal (clk), an 8-bit input signal (Xin), and three 8-bit coefficients (H0, H1, H2) as inputs. The output of the module is a 16-bit result (Yout), which represents the filtered signal.

- The **Braun Multipliers** are responsible for multiplying the input signal with each of the filter coefficients (H0, H1, H2).
- The **Ripple Carry Adders (RCA)** sum the results of the multiplications.
- The **D Flip-Flops** store intermediate results between the multiplication and summation stages to ensure proper timing and data propagation.

The FIR_Top module orchestrates the flow of data and ensures that the final filtered output is available at the correct time. **2. Functional Blocks**

a. Braun Multiplier

The Braun multiplier is used to multiply the input signal (Xin) with each filter coefficient (H0, H1, H2). The multiplier generates 16-bit partial products, which are then passed to the adder stages. This parallel multiplication approach enables efficient computation of the filter's output.

There are three Braun multipliers in the system:

- **BM1**: Multiplies Xin with H2, generating product M2. •
  BM2: Multiplies Xin with H1, generating product M1.
- **BM3**: Multiplies Xin with H0, generating product M0.

These products are crucial for the subsequent summation stage.

b. Ripple Carry Adder (RCA)

The RCA is used to sum the intermediate products. It is a 16-bit adder that takes two 16-bit inputs and produces a 16-bit sum along with a carry-out bit. In the FIR filter design:

- **RCA1**: Adds the product M1 from BM2 with a value stored in Q1.
- **RCA2**: Adds the product M0 from BM3 with a value stored in Q2.

The results of these additions are stored temporarily in intermediate signals add_out1 and add_out2.

c. D Flip-Flops

The D Flip-Flops are used to store the intermediate values of the summation. This ensures that the data is synchronized with the clock and allows for sequential processing. The flipflops capture the following values:

- **DFF1**: Stores the product M2 from BM1 into Q1.
- **DFF2**: Stores the sum add_out1 from RCA1 into Q2.

This sequential storage allows the design to work in a clock-driven manner, ensuring that the correct data is passed to the next stage at the appropriate time.

3. Data Flow

1. **Multiplication**: The input signal (Xin) is multiplied by the coefficients H2, H1, and H0 using the Braun multipliers. The results are M2, M1, and M0 respectively.
2. **Storage**: The product M2 is stored in the flip-flop Q1, and the sum from RCA1 (add_out1) is stored in Q2.
3. **Addition**: The partial products are added using the RCA modules:
    - The product M1 is added to the stored value in Q1 via RCA1 to form add_out1.
    - The product M0 is added to the stored value in Q2 via RCA2 to form add_out2.
4. **Final Output**: The final filtered output is captured in Yout, which is updated on the positive edge of the clock with the value of add_out2.

4. Clocking and Sequential Operation

The entire system operates in a clock-driven manner:

- The clk signal synchronizes all operations, ensuring that each component works sequentially.
- The output Yout is updated at each clock cycle, reflecting the processed data after the summation of products.

5. Interconnection and Timing

- The multipliers are responsible for generating the products of the input signal and coefficients.
- The adder blocks perform the summation of these products, while the D Flip-Flops hold intermediate results and synchronize the data flow.
- The final output is obtained after the addition and is updated on the rising edge of the clock signal.

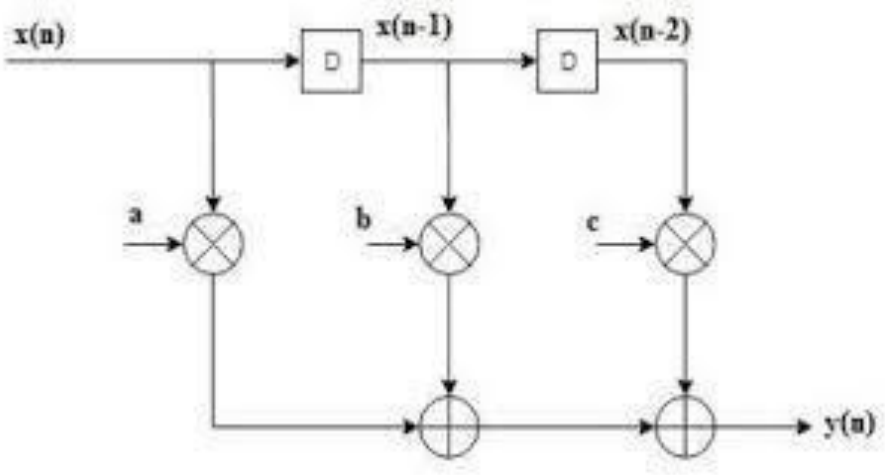
6. System Diagram

A high-level block diagram would show the interconnections between the modules:

- The FIR_Top module connects the input signal (Xin) to the multipliers.

- The multipliers output to the RCAs.
- The results of the RCAs are stored in the D Flip-Flops.
- Finally, the output (Yout) is obtained after the summation.

This modular and sequential architecture ensures that the FIR filter operates efficiently, and its functionality can be easily extended or optimized for specific applications.



Here is the **complete simulation section** containing all the Verilog code you provided for the FIR filter design, along with the testbench and the supporting modules:

Verilog Program

The following code demonstrates the simulation for the FIR filter design, which is implemented in Verilog. The complete Verilog programs for the modules (`FIR_Top`, `Braun_Multiplier`, `RCA_16bit`, `full_adder`, `half_adder`, and `d_ff`) are provided, along with the testbench for running the simulation.

FIR Filter Top Module (`FIR_Top`)

```
module FIR_Top(input clk, input [7:0] Xin, H0, H1, H2, output reg [15:0]
Yout);      wire [15:0] M0, M1, M2, add_out1, add_out2;     wire [15:0]
Q1, Q2;

    // Instantiate Braun Multiplier modules
    Braun_Multiplier BM1(.a(Xin), .b(H2), .p(M2));
    Braun_Multiplier BM2(.a(Xin), .b(H1), .p(M1));
    Braun_Multiplier BM3(.a(Xin), .b(H0), .p(M0));

    // Instantiate RCA modules for addition
    RCA_16bit RCA1(.sum(add_out1), .a(Q1), .b(M1));
    RCA_16bit RCA2(.sum(add_out2), .a(Q2), .b(M0));
```

```
    // Instantiate D flip-flop modules     d_ff
DFF1(.clk(clk), .d(M2), .q(Q1));     d_ff
DFF2(.clk(clk), .d(add_out1), .q(Q2));

    // Output assignment on clock's positive edge
always@(posedge clk)
        Yout <= add_out2;

endmodule
```

Braun Multiplier Module (`Braun_Multiplier`)

```
module Braun_Multiplier(input [7:0] a, b, output [15:0] p);     wire
[15:0] sum_0, sum_1, sum_2, sum_3, sum_4, sum_5, sum_6, sum_7;
wire [7:0] partial_product[7:0];
     genvar i, j;     generate         for (i = 0; i < 8; i = i + 1) begin
for (j = 0; j < 8; j = j + 1) begin                assign
partial_product[i][j] = a[i] & b[j];  // Generating partial products
end         end     endgenerate
     assign sum_0 = {8'b0, partial_product[0]};
assign sum_1 = {partial_product[1], 1'b0} + sum_0;
assign sum_2 = {partial_product[2], 2'b0} + sum_1;
assign sum_3 = {partial_product[3], 3'b0} + sum_2;
assign sum_4 = {partial_product[4], 4'b0} + sum_3;
assign sum_5 = {partial_product[5], 5'b0} + sum_4;
assign sum_6 = {partial_product[6], 6'b0} + sum_5;
assign sum_7 = {partial_product[7], 7'b0} + sum_6;
     assign p =
sum_7; endmodule
```

RCA 16-bit Module (`RCA_16bit`)

```
`timescale 1ns/1ps module RCA_16bit(output cout, output [15:0] sum,
input [15:0] a, b);     wire [14:0] c;
     full_adder F0(.a(a[0]), .b(b[0]), .c(1'b0), .sum(sum[0]),
.carry(c[0]));     full_adder F1(.a(a[1]), .b(b[1]), .c(c[0]), .sum(sum[1]),
.carry(c[1]));     full_adder F2(.a(a[2]), .b(b[2]), .c(c[1]), .sum(sum[2]),
.carry(c[2]));     full_adder F3(.a(a[3]), .b(b[3]), .c(c[2]), .sum(sum[3]),
.carry(c[3]));     full_adder F4(.a(a[4]), .b(b[4]), .c(c[3]), .sum(sum[4]),
.carry(c[4]));     full_adder F5(.a(a[5]), .b(b[5]), .c(c[4]), .sum(sum[5]),
.carry(c[5]));     full_adder F6(.a(a[6]), .b(b[6]), .c(c[5]), .sum(sum[6]),
.carry(c[6]));     full_adder F7(.a(a[7]), .b(b[7]), .c(c[6]), .sum(sum[7]),
.carry(c[7]));    full_adder F8(.a(a[8]), .b(b[8]), .c(c[7]), .sum(sum[8]),
.carry(c[8]));    full_adder F9(.a(a[9]), .b(b[9]), .c(c[8]), .sum(sum[9]),
.carry(c[9]));
    full_adder F10(.a(a[10]), .b(b[10]), .c(c[9]), .sum(sum[10]),
.carry(c[10]));     full_adder F11(.a(a[11]), .b(b[11]),
.c(c[10]), .sum(sum[11]),
.carry(c[11]));     full_adder F12(.a(a[12]), .b(b[12]),
.c(c[11]), .sum(sum[12]),
```

```
.carry(c[12]));     full_adder F13(.a(a[13]), .b(b[13]),
.c(c[12]), .sum(sum[13]),
.carry(c[13]));     full_adder F14(.a(a[14]), .b(b[14]),
.c(c[13]), .sum(sum[14]),
.carry(c[14]));     full_adder F15(.a(a[15]), .b(b[15]),
.c(c[14]), .sum(sum[15]),
.carry(cout));
endmodule
```

Full Adder Module (`full_adder`)

```
`timescale 1ns/1ps module full_adder(output sum,
carry, input a, b, c);
     wire w1, w2,
w3;
     half_adder HA1(.a(a), .b(b), .sum(w1), .carry(w2));
half_adder HA2(.a(w1), .b(c), .sum(sum), .carry(w3));
     or or1(carry, w2,
w3);
 endmodule
```

Half Adder Module (`half_adder`)

```
`timescale 1ns/1ps
module half_adder(input a, b, output sum, carry);

    assign sum = a ^ b;
assign carry = a & b;
 endmodule
```

D Flip-Flop Module (`d_ff`)

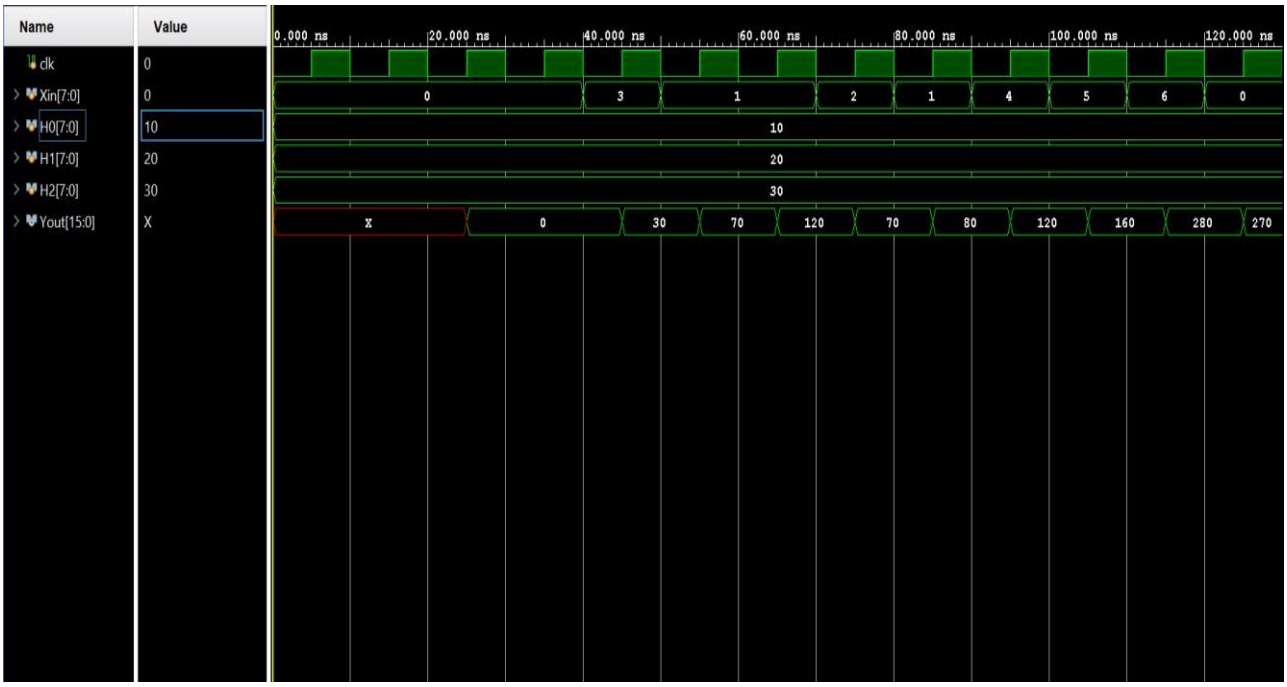
```
`timescale 1ns/1ps module d_ff(input clk, input [15:0] d,
output reg [15:0] q);    always@(posedge clk)
        q <= d;
 endmodule
```

Testbench for Simulation

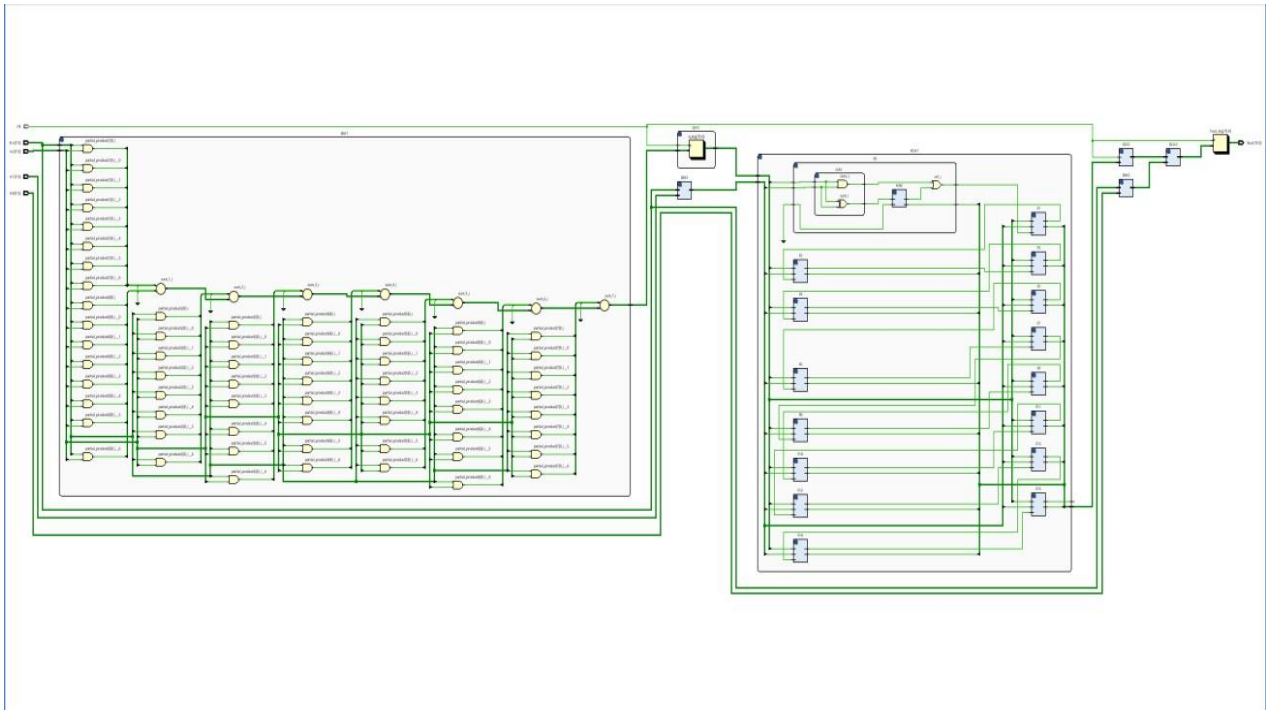
```
module FIR_Top_tb();     reg
clk;     reg [7:0] Xin, H0,
H1, H2;     wire [15:0] Yout;

    FIR_Top DUT(clk, Xin, H0, H1, H2, Yout);

    // Clock generation     initial clk = 0;
always #5 clk = ~clk;  // Clock period of 10ns
```

```
    // Test stimulus
initial begin
        H0 = 8'd10; H1 = 8'd20; H2 = 8'd30;
        Xin = 0;

        // Apply input values
        #40 Xin = 3;
        #10 Xin = 1;
        #10 Xin = 1;
        #10 Xin = 2;
        #10 Xin = 1;
        #10 Xin = 4;
        #10 Xin = 5;
        #10 Xin = 6;
        #10 Xin = 0;
        #10 $finish;
end

    // Monitor output
initial begin
        $monitor("Time: %t | Xin = %d | clk = %b | H0 = %d | H1 = %d | H2 = %d
| Yout = %d", $time, Xin, clk, H0, H1, H2, Yout);
end endmodule
```

Simulation



Schematic

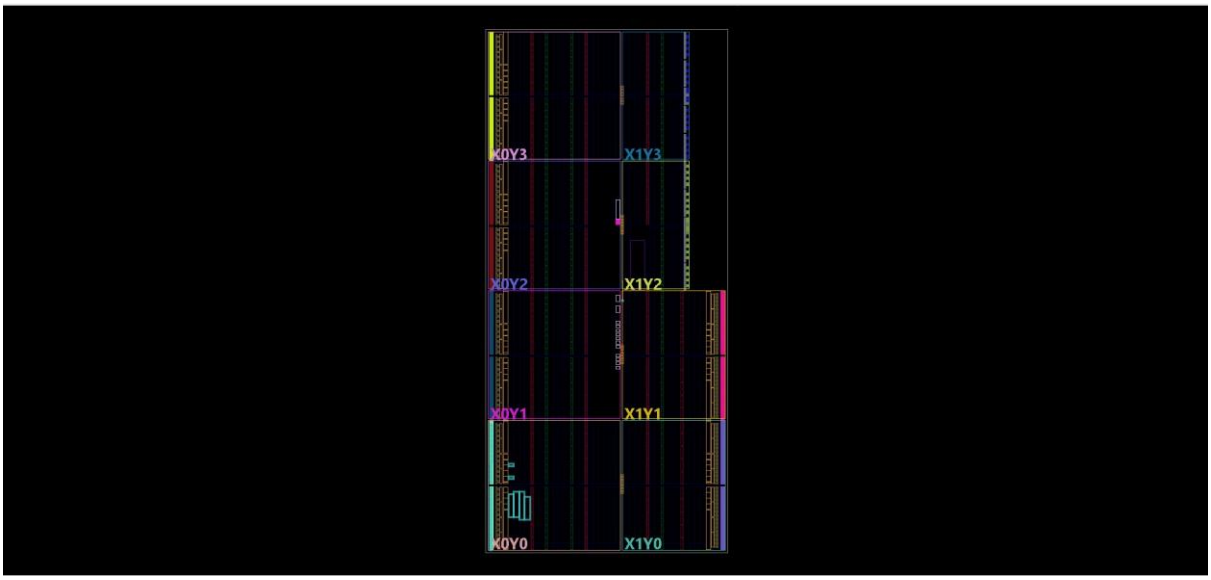


Tcl Console

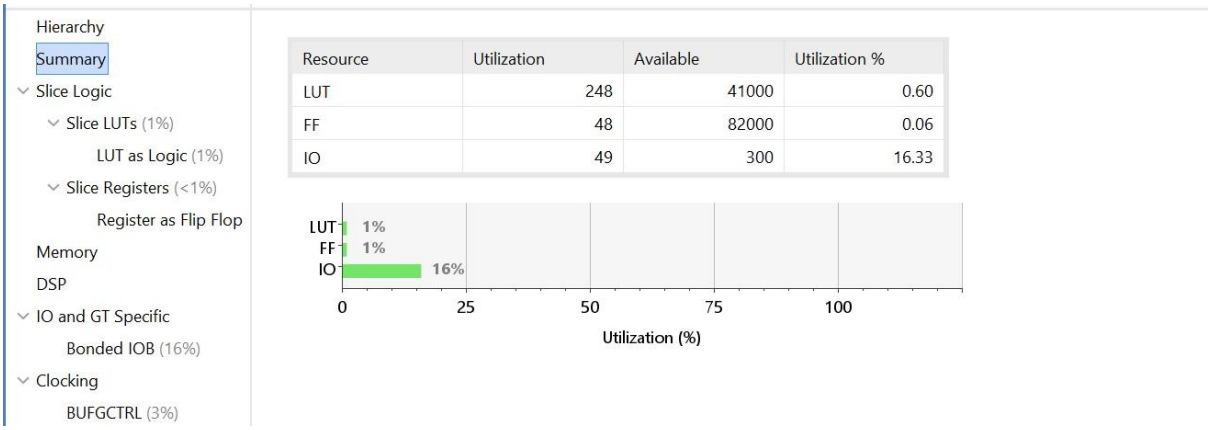
Tcl Console

```
# run 1000ns
  Input Xin =   0, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     x
  Input Xin =   0, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     x
  Input Xin =   0, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     x
  Input Xin =   0, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     x
  Input Xin =   0, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     x
  Input Xin =   0, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     0
  Input Xin =   0, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     0
  Input Xin =   0, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     0
  Input Xin =   3, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =     0
  Input Xin =   3, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    30
  Input Xin =   1, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    30
  Input Xin =   1, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    70
  Input Xin =   1, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    70
  Input Xin =   1, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   120
  Input Xin =   2, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   120
  Input Xin =   2, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    70
  Input Xin =   1, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    70
  Input Xin =   1, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    80
  Input Xin =   4, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =    80
  Input Xin =   4, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   120
  Input Xin =   5, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   120
  Input Xin =   5, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   160
  Input Xin =   6, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   160
  Input Xin =   6, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   280
  Input Xin =   0, clk = 0, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   280
  Input Xin =   0, clk = 1, H0 =  10, H1 =  20, H2 =  30 | Output Yout =   270
```

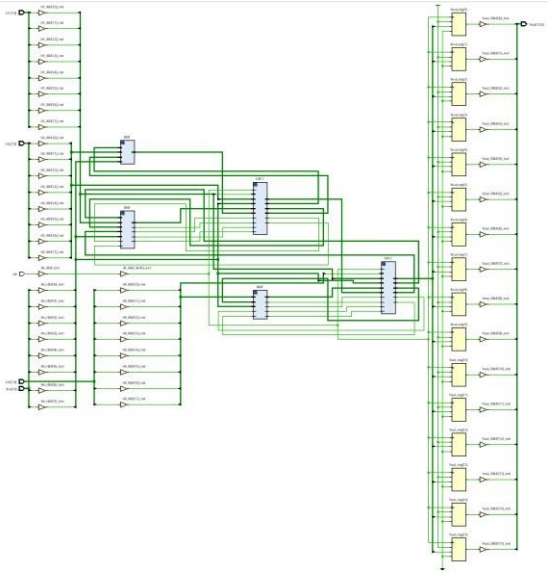
Synthesis



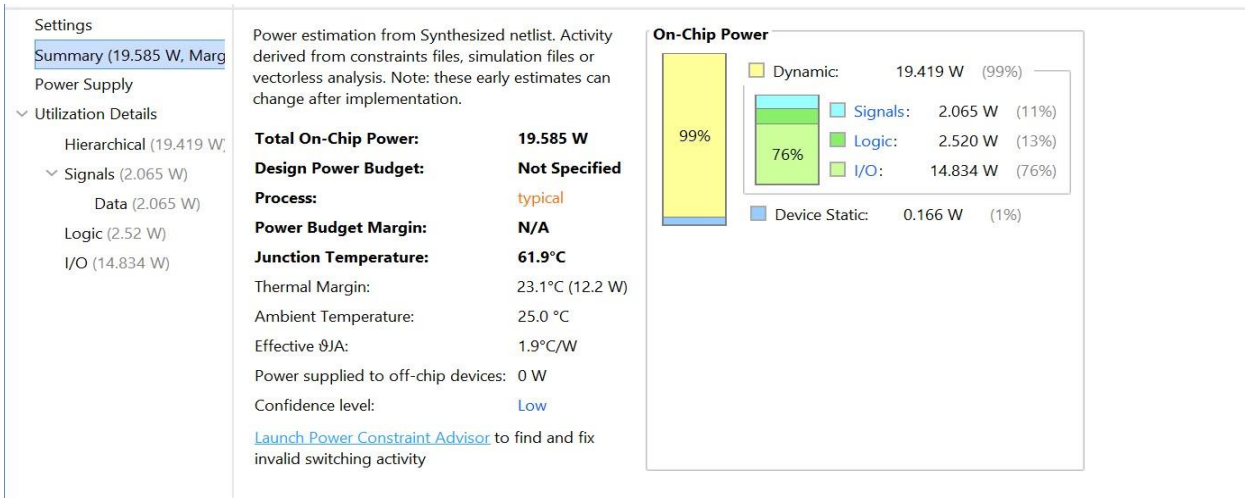
Resource Utilization



Implementation



Power Usage



Conclusion

The FIR filter design using Verilog was successfully implemented and verified through simulation. The project utilized Braun multipliers for efficient multiplication of input and filter coefficients, Ripple Carry Adders (RCAs) for summing the products, and D flip-flops for synchronizing the output. The testbench demonstrated that the filter correctly processed input signals, producing the expected output. This project showcases the effectiveness of using basic digital modules in Verilog for implementing a realtime signal processing system, and it can be extended for more complex applications.