McGill University

ECSE 325 - Lab 3

Timing Constraint Specification and Timing Analysis using TimeQuest

Group 2
Section 005

Yi Zhu 260716006

Mai Zeng 260782174

1. Introduction

The main goal of this lab is to learn how to specify timing constraints and perform static timing analysis of the synthesized circuit using the TimeQuest timing analyzer. In order to do so, we will implement a Finite Impulse Response(FIR) Filter in VHDL and perform the simulations on the FIR by using the ModelSim. An FIR filter is a filter that provides a finite-period response to any finite length input. For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y(n) = \sum_{i=0}^{N} b_i * x(n-i)$$

Figure 1.1 Output Formula for a Causal Discrete-time FIR Filter of Order N

where x(n) is the input signal, y(n) is the output signal and bi denotes the weights. We can see the system flow chart for an FIR filter in *Figure 1.2* below:

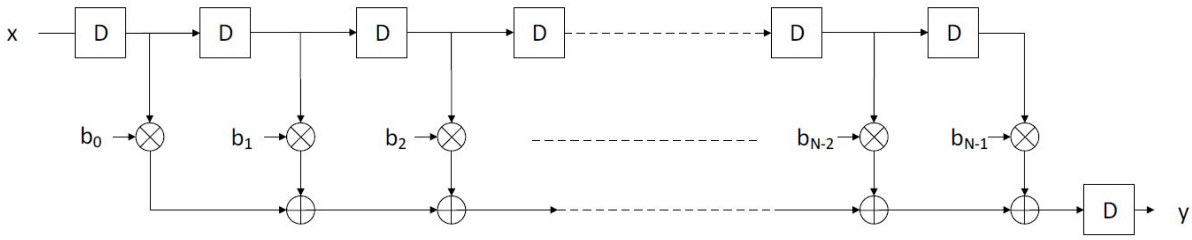


Figure 1.2 Flow Chart of the FIR Filter

Besides on, we will also use the concept of Root Mean Square Error (RMSE), which is the standard deviation of the residuals, in our design optimization. RMSE is a measure of how spread out these residuals are and it tells us how concentrated the data is around the line of best fit. The RMSE can be calculated by:

$$RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(\hat{y}_i - y_i)^2}{N}}$$

Figure 1.3 Formula for Root Mean Square Error

Where $\hat{y}_i$ is the estimated value, y_i is the actual value and N is the number of samples.

Finally, after the compilation is finished, we will apply two common solutions to resolve the timing issues of the violated paths in case of any timing violation. The first solution is to reduce levels of combinational logic for the violated paths and the second solution is to redesign architecture and rearrange registers. We will see the exercise of these two solutions in section 4 of this lab report.

2. VHDL Code

2.1 Finite Impulse Response Filter

In this part, we implement a bandpass FIR filter with order 25 (25-tap FIR filter) in VHDL to restore a sine wave corrupted by white noise where we are provided with filter inputs and weights in floating-point format. The VHDL code for the FIR filter is shown in *Figure 2.1-1 and 2.1-2* in the next two pages.

In our code, we use IEEE as our main library and three sub-libraries, which are std_logic_1164, numeric_std, std_logic_unsigned and std_logic_textio. The std_logic_unsigned is used in this lab since we will deal with the signed number and the std_logic_textio is implemented since we will use standard textio procedures such as READ and WRITE.

In the entity section, there are three inputs x, clk, rst and one output y. The 16-bits-input x represents the input of the sequence and 17-bits-output x represents the output of the finite impulse response filter. Since we will implement a time-sensitive finite impulse response filter, we have clk as an input. Thus, the filter will only be processed at the rising edge of the clock cycle. Besides on, we also have rst as our input since we will include the reset function for the FIR filter. When the rst is high, all the values stored in the signals and the output of the filter are reset to 0. At this moment, the filter will be ready to take in new inputs and generate the outputs.

In the architecture section, we have three array signals, naming COEFF_ARRAY. X_ARRAY and MUL_ARRAY. The COEFF_ARRAY is an array with 25 signed-16-bits elements to hold the coefficients given. The X_ARRAY is an array with 25 signed-16-bits elements to hold the input values. Lastly, the MUL_ARRAY is an array with 25 signed-32-bits elements to hold the product results of the weights and the input values. In the process block, since the FIR filter is time-sensitive and has the reset function, we have clk and rst in the sensitivity list. For the reset function, we have one if statement that when the rst signal is high, we will set all array values to be ‘0’. Then, we followed the concept of FIR filter introduced in the introduction section of this report to implement the filter and collect the output into y. All the calculation process will happen at the rising edge of the clock cycle only.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_signed.ALL;
use ieee.std_logic_textio.all;

entity g02_FIR is
port(
    x : in std_logic_vector (15 downto 0); -- input Signal
    clk : in std_logic; -- clock
    rst : in std_logic; -- asynchronous active-high reset
    y : out std_logic_vector (16 downto 0) -- output signal
);
end g02_FIR;

architecture fir of g02_FIR is
    -- type ARRAY32 is array(24 downto 0) of signed(31 downto 0);
    type ARRAY_G is array(24 downto 0) of signed(15 downto 0);      -- define an array type
    type MULARRAY is array(24 downto 0) of signed(31 downto 0);
    signal COEFF_ARRAY : ARRAY_G;                          -- create array to hold coefficients
    signal X_ARRAY : ARRAY_G;                          -- create array to hold input values
    signal MUL_ARRAY : MULARRAY;
    --signal MULTI : signed(31 downto 0);
    begin
        -- fill input array with initial values of 0
        --INPUT_ARRAY <= (others=>(others=>'0'));
        COEFF_ARRAY(0) <=   "0000001001110011";
        COEFF_ARRAY(1) <=   "0000000000010001";
        COEFF_ARRAY(2) <=   "1111111111010010";
        COEFF_ARRAY(3) <=   "1111111011011101";
        COEFF_ARRAY(4) <=   "0000001100011010";
        COEFF_ARRAY(5) <=   "1111110110100111";
        COEFF_ARRAY(6) <=   "1111110000001101";
        COEFF_ARRAY(7) <=   "0000110110111101";
        COEFF_ARRAY(8) <=   "1110110001110010";
        COEFF_ARRAY(9) <=   "0000110111111000";
        COEFF_ARRAY(10) <=  "0000001100001000";
        COEFF_ARRAY(11) <=  "1110101000001010";
        COEFF_ARRAY(12) <=  "0001111000110100";
        COEFF_ARRAY(13) <=  "1110101000001010";
        COEFF_ARRAY(14) <=  "0000001100001000";
        COEFF_ARRAY(15) <=  "0000110111111000";
        COEFF_ARRAY(16) <=  "1110110001110010";
        COEFF_ARRAY(17) <=  "0000110110111101";
        COEFF_ARRAY(18) <=  "1111110000001101";
        COEFF_ARRAY(19) <=  "1111110110100111";
        COEFF_ARRAY(20) <=  "0000001100011010";
        COEFF_ARRAY(21) <=  "1111111011011101";
        COEFF_ARRAY(22) <=  "1111111111010010";
        COEFF_ARRAY(23) <=  "0000000000010001";
        COEFF_ARRAY(24) <=  "0000001001110011";


        filter : process(rst, clk)
        variable sum : signed (31 downto 0);
        begin
            if rst = '1' then
                -- reset array values to 0
                sum := (others => '0');
                X_ARRAY <= (others=>(others=>'0'));
                MUL_ARRAY <= (others=>(others=>'0'));

```

Figure 2.1-1 VHDL Code for FIR Filter Part 1

```vhdl
        filter : process(rst, clk)
        variable sum : signed (31 downto 0);
        begin
            if rst = '1' then
                -- reset array values to 0
                sum := (others => '0');
                X_ARRAY <= (others=>(others=>'0'));
                MUL_ARRAY <= (others=>(others=>'0'));

            -- Regular
            elsif(rising_edge(clk)) then
                -- reset temporary array
                sum := (others => '0');
                MUL_ARRAY <= (others=>(others=>'0'));

                X_ARRAY(0) <= signed(x);

                for i in 1 to 24 loop
                   X_ARRAY(i) <= X_ARRAY(i-1);
                end loop;

                -- calculate output value
                for i in 0 to 24 loop
                    sum := sum + (X_ARRAY(i) * COEFF_ARRAY(i));
                end loop;

                y <= std_logic_vector(sum(31 downto 15));

            -- Broadcast Filter
            --   MUL_ARRAY(0) <= COEFF_ARRAY(24) * signed(x);
            --   for i in 1 to 24 loop
            --      MUL_ARRAY(i) <= MUL_ARRAY(i-1) + signed(x) * COEFF_ARRAY(24-i);
            --   end loop;
            --   y <= std_logic_vector(MUL_ARRAY(24)(31 downto 15));

            end if;
        end process filter;
end architecture;
```

Figure 2.1-2 VHDL Code for FIR Filter Part 2

2.2 Broadcasting Form of Finite Impulse Response Filter

In this part, we implement the broadcasting form of the FIR filter in VHDL while representing the filter's input, output signals and weights in the fixed-point representations (1,15), (2,15) and (1,15), respectively. The VHDL code of the broadcasting form of the FIR filter is shown in *Figure 2.2-1* and *2.2-2* below:

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_signed.ALL;
use ieee.std_logic_textio.all;

entity g02_FIR is
port(
    x : in std_logic_vector (15 downto 0); -- input Signal
    clk : in std_logic; -- clock
    rst : in std_logic; -- asynchronous active-high reset
    y : out std_logic_vector (16 downto 0) -- output signal
);
end g02_FIR;

architecture fir of g02_FIR is
    -- type ARRAY32 is array(24 downto 0) of signed(31 downto 0);
    type ARRAY_G is array(24 downto 0) of signed(15 downto 0);      -- define an array type
    type MULARRAY is array(24 downto 0) of signed(31 downto 0);
    signal COEFF_ARRAY : ARRAY_G;                          -- create array to hold coefficients
    signal X_ARRAY : ARRAY_G;                          -- create array to hold input values
    signal MUL_ARRAY : MULARRAY;
    --signal MULTI : signed(31 downto 0);
    begin
        -- fill input array with initial values of 0
        --INPUT_ARRAY <= (others=>(others=>'0'));
        COEFF_ARRAY(0) <=   "0000001001110011";
        COEFF_ARRAY(1) <=   "0000000000010001";
        COEFF_ARRAY(2) <=   "1111111111010010";
        COEFF_ARRAY(3) <=   "1111111011011101";
        COEFF_ARRAY(4) <=   "0000001100011010";
        COEFF_ARRAY(5) <=   "1111110110100111";
        COEFF_ARRAY(6) <=   "1111110000001101";
        COEFF_ARRAY(7) <=   "0000110110111101";
        COEFF_ARRAY(8) <=   "1110110001110010";
        COEFF_ARRAY(9) <=   "0000110111111000";
        COEFF_ARRAY(10) <=  "0000001100001000";
        COEFF_ARRAY(11) <=  "1110101000001010";
        COEFF_ARRAY(12) <=  "0001111000110100";
        COEFF_ARRAY(13) <=  "1110101000001010";
        COEFF_ARRAY(14) <=  "0000001100001000";
        COEFF_ARRAY(15) <=  "0000110111111000";
        COEFF_ARRAY(16) <=  "1110110001110010";
        COEFF_ARRAY(17) <=  "0000110110111101";
        COEFF_ARRAY(18) <=  "1111110000001101";
        COEFF_ARRAY(19) <=  "1111110110100111";
        COEFF_ARRAY(20) <=  "0000001100011010";
        COEFF_ARRAY(21) <=  "1111111011011101";
        COEFF_ARRAY(22) <=  "1111111111010010";
        COEFF_ARRAY(23) <=  "0000000000010001";
        COEFF_ARRAY(24) <=  "0000001001110011";


        filter : process(rst, clk)
        variable sum : signed (31 downto 0);
        begin
            if rst = '1' then
                -- reset array values to 0
                sum := (others => '0');
                X_ARRAY <= (others=>(others=>'0'));
                MUL_ARRAY <= (others=>(others=>'0'));

```

Figure 2.2-1 VHDL Code for Broadcast Filter Part 1

```
filter : process(rst, clk)
variable sum : signed (31 downto 0);
begin
    if rst = '1' then
        -- reset array values to 0
        sum := (others => '0');
        X_ARRAY <= (others=>(others=>'0'));
        MUL_ARRAY <= (others=>(others=>'0'));

    -- Regular
    elsif(rising_edge(clk)) then
        -- reset temporary array
        sum := (others => '0');
        MUL_ARRAY <= (others=>(others=>'0'));

        --X_ARRAY(0) <= signed(x);

        --for i in 1 to 24 loop
            --X_ARRAY(i) <= X_ARRAY(i-1);
        --end loop;

        -- calculate output value
        --for i in 0 to 24 loop
        --  sum := sum + (X_ARRAY(i) * COEFF_ARRAY(i));
        --end loop;

        --y <= std_logic_vector(sum(31 downto 15));

    -- Broadcast Filter
         MUL_ARRAY(0) <= COEFF_ARRAY(24) * signed(x);
         for i in 1 to 24 loop
            MUL_ARRAY(i) <= MUL_ARRAY(i-1) + signed(x) * COEFF_ARRAY(24-i);
         end loop;
         y <= std_logic_vector(MUL_ARRAY(24)(31 downto 15));

    end if;
end process filter;
end architecture;
```

Figure 2.2-2 VHDL Code for Broadcast Filter Part 2

For the broadcast filter, the entity section and the first part of the architecture section are the same as for the FIR filter. The only difference is in the process section that instead of using the concept of FIT filter, we followed the system flow chart of the broadcast filter illustrated in *Figure 2.2-3* below. All the calculation process will happen at the rising edge of the clock cycle only as well.

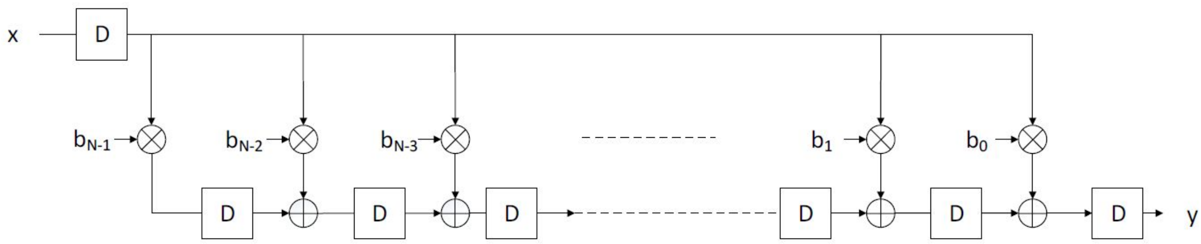


Figure 2.2-3 System Flow Chart for Broadcast Filter

2.3 Testbench

The VHDL code for the testbench of the regular FIR filter is shown in *Figure 2.3-1* and *2.3-2* below:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.ALL;
use STD.textio.all;
use ieee.std_logic_textio.all;

entity G02_FIR_TB is
end G02_FIR_TB;

architecture testbench of G02_FIR_TB is
    -- Define Component
    component G02_FIR is
        port(
            x : in std_logic_vector (15 downto 0); -- input Signal
            clk : in std_logic; -- clock
            rst : in std_logic; -- asynchronous active-high reset
            y : out std_logic_vector (16 downto 0) -- output signal
            );
    end component;

    -- Define testbench internal signal
    file file_vectors_x : text;
    file file_coeff : text;
    file file_results : text;

    -- clock
    constant clk_period : time := 100ns;

    signal x_in : std_logic_vector(15 downto 0);
    signal clk_in : std_logic;
    signal rst_in : std_logic;
    signal y_out : std_logic_vector(16 downto 0);

    begin
    -- instantiate FIR
    G02_FIR_INST : G02_FIR
        port map(
            x => x_in,
            clk => clk_in,
            rst => rst_in,
            y => y_out
        );

    -- clock generation
    clk_generation : process
    begin
        clk_in <= '1';
        wait for clk_period / 2;
        clk_in <= '0';
        wait for clk_period / 2;
    end process clk_generation;

    ---------------------------------------------------------------------------
    -- Providing Inputs
    ---------------------------------------------------------------------------
    feeding_instr : process is
        variable v_Iline1 : line;
        -- variable v_Iline2 : line;
        variable v_Oline : line;
        variable v_x_in : std_logic_vector(15 downto 0);
        variable v_y_in : std_logic_vector(16 downto 0);

```

Figure 2.3-1 VHDL Code for FIR Filter Testbench Part 1

```vhdl
    begin
        --reset the circuit
        rst_in <= '1';
        wait until rising_edge(clk_in);
        wait until rising_edge(clk_in);
        rst_in <= '0';
        file_open(file_VECTORS_X, "P:/McGill/ECSE325/Lab3/lab3-in-converted.txt", read_mode);
        file_open(file_RESULTS, "P:\McGill\ECSE325\Lab3\lab3-out.txt", write_mode);
        --file_open(file_VECTORS_X, "./lab3-in-converted.txt", read_mode);
        --file_open(file_RESULTS, "./lab3-out.txt", write_mode);

        while not endfile(file_VECTORS_X) loop
            readline(file_VECTORS_X, v_Iline1);
            read(v_Iline1, v_x_in);
            x_in <= v_x_in;
            wait until rising_edge(clk_in);
                    wait for 25 ns;
            write(v_Oline, y_out);
            writeline(file_RESULTS, v_Oline);
            --wait until rising_edge(clk_in);
        end loop;
        wait;
    end process;
end architecture;
```

Figure 2.3-2 VHDL Code for FIR Filter Testbench Part 2

A testbench is a special VHDL entity that generates inputs applied to our circuit, to automate the simulation of our circuit and compare the outputs to respond to different inputs. In the port declaration section of this testbench code, we have the same inputs and outputs as in the entity of FIR filter code. We then define three text-type files for the testbench: file_vectors_x, file_corff and file results for inputs, coefficients and results respectively. We also set a constant clk_period for the 100ns time interval of the simulation period. Four internal signals, naming x_in, clk_in, rst_in and y_out, are defined in order to accomplish the port mapping. Then we map the ports to the FIR filter and initiate the clock. Finally, we start our simulation and write the result into the result file. Four variables named v_Iline, v_Oline, v_x_in and v_y_in are created for result file writing purposes. We first reset everything and wait for the clock cycle to reach a rising edge. At each rising edge, we clear the reset and then the input will be taken from the input text file and put into the FIR filter. The output generated by the filter will then be written into the result text file at the rising edge of one clock cycle as well. After so, we will compare the output result in the text file with the correct result.

Here it is noted that in convenience, for all these three testbenches we basically use the same format except the bitwidth changes and also noted that for broadcast design and regular design we use same file because the bitwidth is the same but for reducing bitwidth design we use different testbench but with same format.
The bitwidth changes are shown in *Table 2.3* below:

	X_IN	Y_OUT
Regular Design	15 to 0	16 to 0
Reducing Bitwidth	2 to 0	3 to 0
Broadcast	15 to 0	16 to 0

Table 2.3 Changes of Bitwidth

2.3.1 Testbench Result

Figure 2.3.1-1 to *2.3.1-3* below show the output results from the testbench:

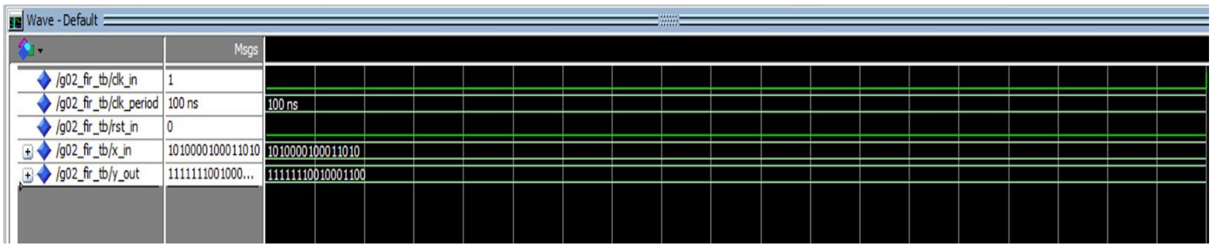


Figure 2.3.1-1 Result from Testbench for Regular Design

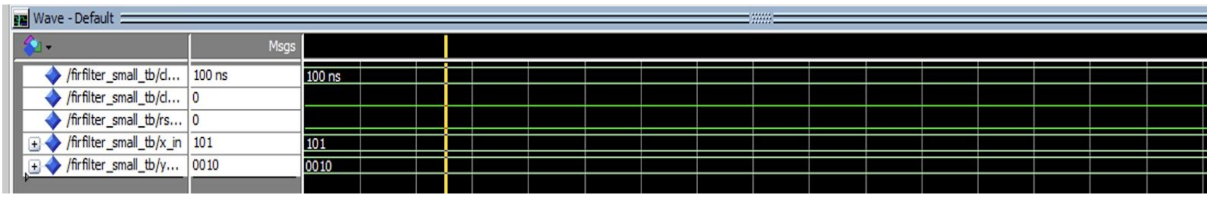


Figure 2.3.1-2 Result from Testbench for Reducing Bitwidth Design

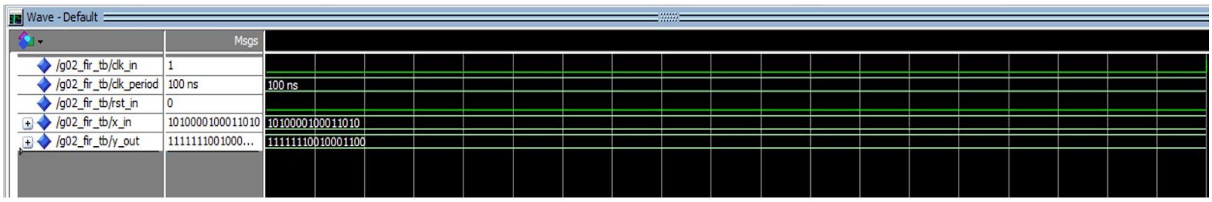


Figure 2.3.1-3 Result from Testbench for Regular Design

3. Resource Utilization

3.1 FIR Filter Regular Implementation

The flow summary of the regular form of the implementation is shown below in *Figure 3.1-1*. The logical utilization is 97/32070 and the total number of registers in use are 424. The registers that needed theoretically are 25 x 16 + 17 = 417 and it is matched with our design. The reason is that there are 25 elements in the x-input array where each element has 16 bits therefore the number of registers for holding the array is 25 x 16 = 400. Moreover, we need 17 registers to store the coefficient number since the coefficient number needs 17 bits for each and in total there would be 417 registers. We may need some other registers for some other reasons but as long as it is about to 424 which is fine. We can also look at the design in the RTL viewer for the regular implementation and it is shown in section 4.1.

Flow Summary

<<Filter>>

Flow Status	Successful - Sun Apr 12 21:16:09 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G02_FIR
Top-level Entity Name	g02_FIR
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	97 / 32,070 (< 1 %)
Total registers	424
Total pins	35 / 457 (8 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	24 / 87 (28 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 3.1-1 Flow Summary for Regular Design

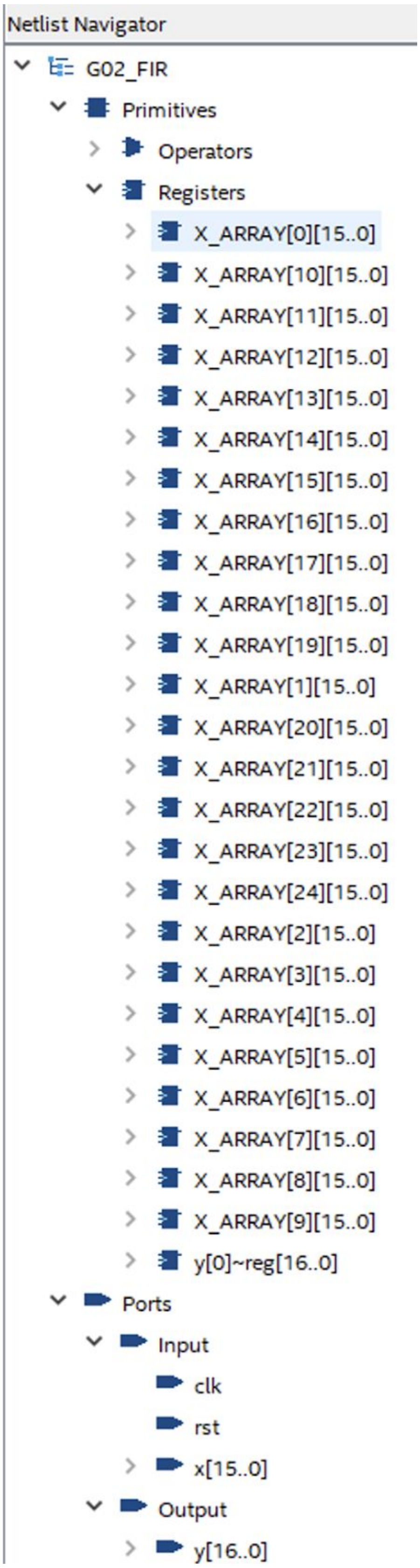


Figure 3.1-2 Registers used for Regular Design

3.2 FIR Filter with Reducing Combinational Logic Levels

The figure below shows the flow summary for the design with reducing bitwidth. Here, we reduced the input bitwidth to 3 bits as well as the weight for the coefficient number. As it is shown in *Figure 3.2-1*, now we only need 79 registers. The previous FIR filter implementation with 16 bits triggers several failing paths regarding the specific timing constraint, in the lab manual, it indicated that one of the strategies to resolve the timing constraint issue is to reduce the levels of combinational logic. In order to do that we reduce the bitwidth of signals and since we reduced the bitwidth of the signals now we see that the registers are less than before (79 < 424). The logic behind this should more likely be we reduced the registers that we need to use in order to reduce the bitwidth signal. And with this technique, now we are having no failing paths and increasing the maximum frequency.

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Apr 13 01:06:21 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G02_FIR
Top-level Entity Name	g02_FIR
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	19 / 32,070 (< 1 %)
Total registers	79
Total pins	9 / 457 (2 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	24 / 87 (28 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 3.2-1 Flow Summary for Design with Reducing Bitwidth

The figure below shows why we need 79 registers. For each X_ARRAY input we need 3 registers since the bitwidth is 3 bits now and we need 25 of them. For the output we need 4 registers since the output is 4 bits. The sum of them is 25 x 3 + 4 = 79.

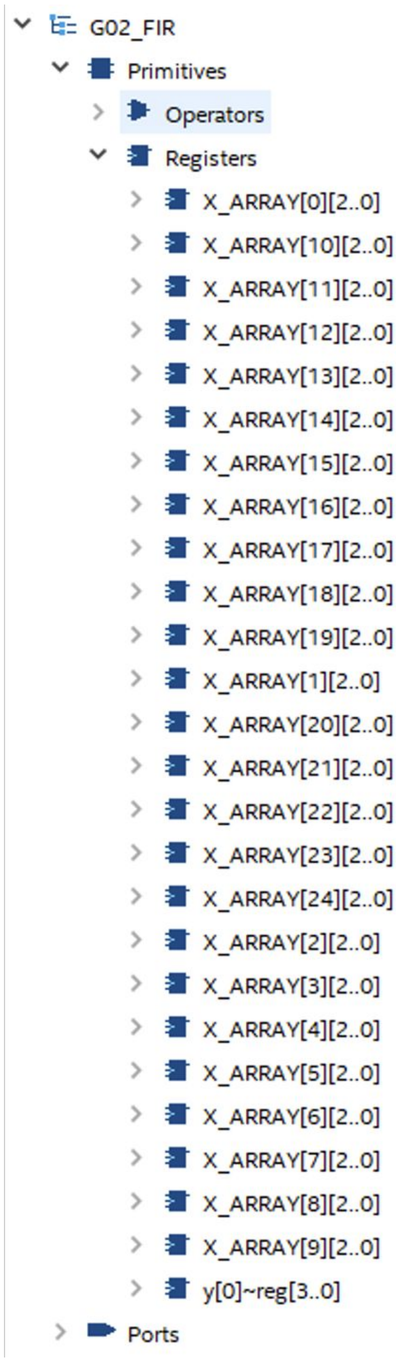


Figure 3.2-2 Registers used for the Design with Reducing Bitwidth

3.3 Broadcast Filter

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Apr 13 02:03:43 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G02_FIR
Top-level Entity Name	g02_FIR
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	389 / 32,070 (1 %)
Total registers	801
Total pins	35 / 457 (8 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	13 / 87 (15 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 3.3-1 Flow Summary for Broadcast Filter

Figure 3.3-1 shows the flow summary for the broadcast filter. The logic Utilization is 380/32070 and the total registers used are 801. The broadcast filter is using an array holding the multiplication result and this array has a 25 array index of 32 bits long. Therefore, we have 25 x 32 = 800 total register numbers. *Figure 3.3-2* below shows the registers that are used for the broadcast filter:

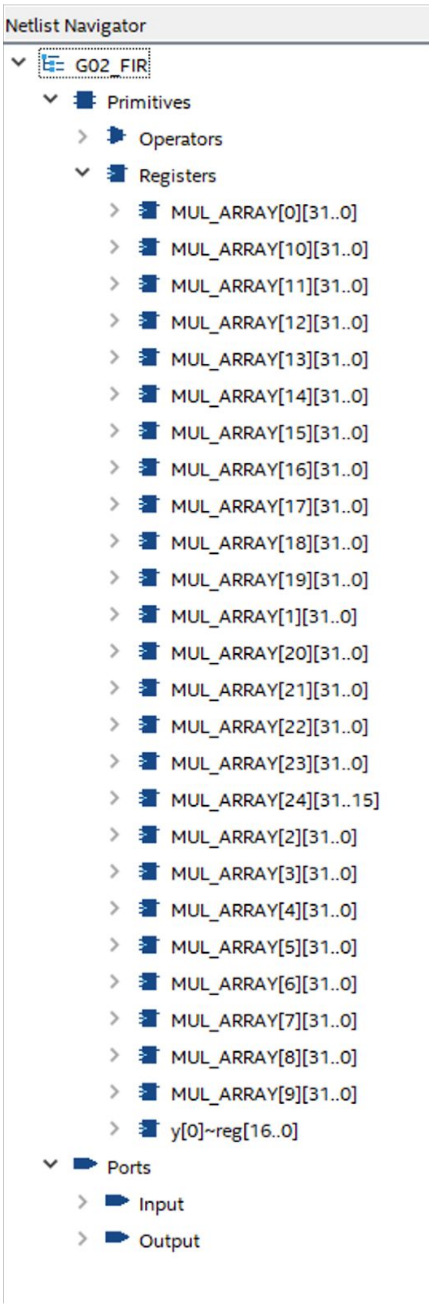


Figure 3.3-2 Registers Used for Broadcast Filter

3.4 Verification

The figures below show how we verify our result is correct and we get the RMSE all in a very accurate value in MATLAB.

```
co = fopen('lab3-out.txt', 'rt');
so = fopen('lab3-sim-out.txt', 'rt');
so2 = fopen('lab3-sim-bf-out.txt', 'rt');
so3 = fopen('lab3-sim-small-out.txt', 'rt');

cv = fscanf(co, '%f');
cv = [0; cv];
cvfp = fi(cv, 1, 17, 15);

q = quantizer([17 15]);

svb = [];
for i = 1:1000
    svb = [svb; fgetl(so)];
end
sv = bin2num(q, svb);

svb2 = [];
for i = 1:1000
    svb2 = [svb2; fgetl(so2)];
end
sv2 = bin2num(q, svb2);

svb3 = [];
for i = 1:1000
    svb3 = [svb3; fgetl(so3)];
end
sv3 = bin2num(quantizer([4 2]), svb3);

rmse1 = sqrt(sum((sv - cv).^2) / 1000);
rmse2 = sqrt(sum((sv2 - cv).^2) / 1000);
rmse3 = sqrt(sum((sv3 - cv).^2) / 1000);
```

Figure 3.4-1 MATLAB Verification Code

rmse1	3.8636e-04
rmse2	3.8636e-04
rmse3	0.1943

Figure 3.4-2 Verification Result

The figure below shows how we find that 3 is the bitwidth which is the minimum bitwidth for letting RMSE < 0.27. Basically, we keep feeding smaller and smaller bitwidth until the RMSE is bigger than 0.27.

```matlab
rmse = [];
for k = 1:15
    cfp = fi(coef, 1, k + 1, k);
    xfp = fi(xv, 1, k + 1, k);
    correct = [];
    for i = 1:length(xfp)
        val = fi(0, 1, k + 2, k);
        if (i <= 25)
            for j = 1:i
                val = val + xfp(i - j + 1) * cfp(j);
            end
        else
            for j = 1:25
                val = val + xfp(i - j + 1) * cfp(j);
            end
        end
        correct = [correct; val];
    end
    correct = fi(correct, 1, k + 2, k);

    rmse = [rmse; sqrt(sum((correct - cvfp).^2) / 1000)];
end


insmall = fopen('lab3-In-fixed-point-small.txt', 'wt');
xfp = fi(xv, 1, 3, 2);
bxfp = bin(xfp);
for i = 1:length(bxfp)
    fprintf(insmall, '%s\n', bxfp(i, :));
end

cout = fopen('lab3-coef-fixed-point-small.txt', 'wt');
cfp = fi(coef, 1, 3, 2);
bcfp = bin(cfp);
for i = 1:length(cfp)
    fprintf(cout, '"%s",\n', bcfp(i, :));
end
```

Figure 3.4-3 Bitwidth Finder Code (MATLAB)

4. Timing Analysis by TimeQuest

During our compilation, we used the same SDC file shown below and this is how we tell the time analyzer that the clock period is 20ns.

```
G02_FIR.vhd    g02_testbed.sdc

1  create_clock -period 20 [get_ports clk]

   create_clock [-add] [-name <clock_name>] -period <value> [-waveform <edge_list>] [<targets>]
   -add: Adds clock to a node with an existing clock
   -name <clock_name>: Clock name of the created clock
   -period <value>: Speed of the clock in terms of clock period
   -waveform <edge_list>: List of edge values
   <targets>: List or collection of targets
```

Figure 4 SDC File

4.1 FIR Filter

The following figures show that there are 5 timing violations in the timing analysis and the maximum frequency is 12.49MHz.

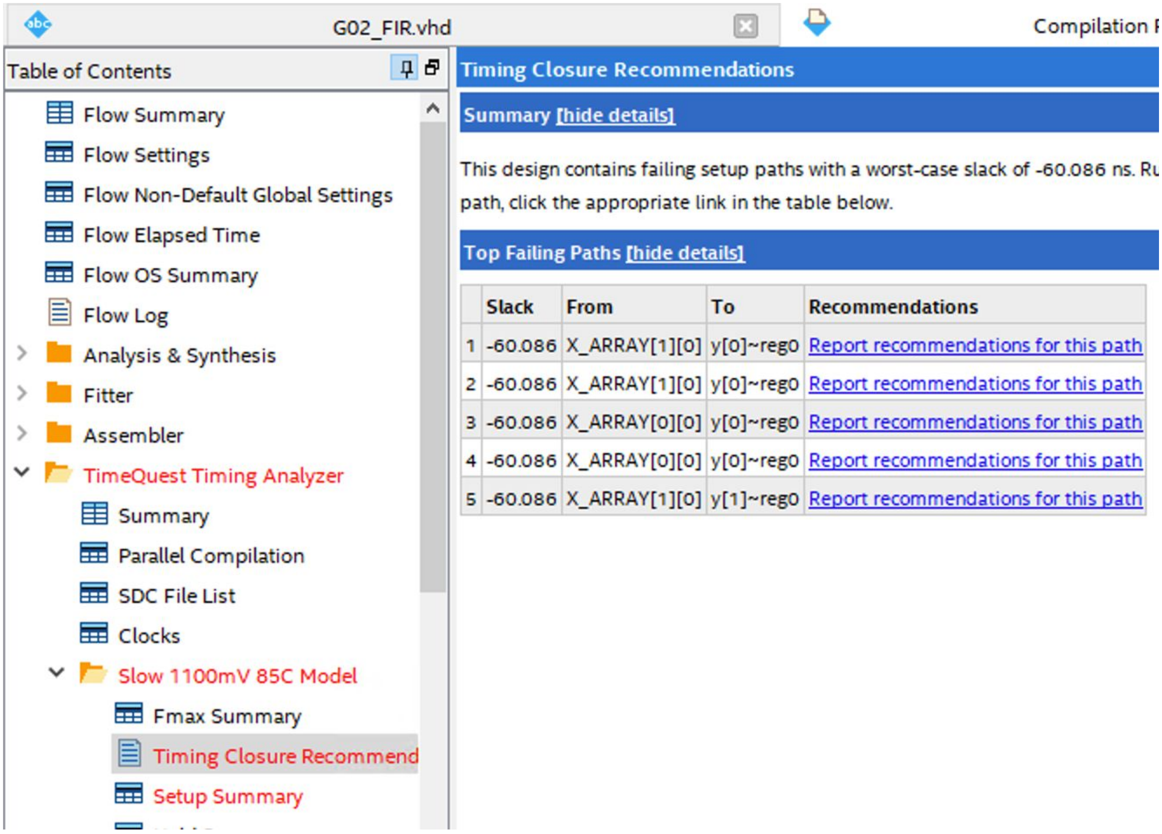


Figure 4.1-1 FIR Filter Timing Analysis

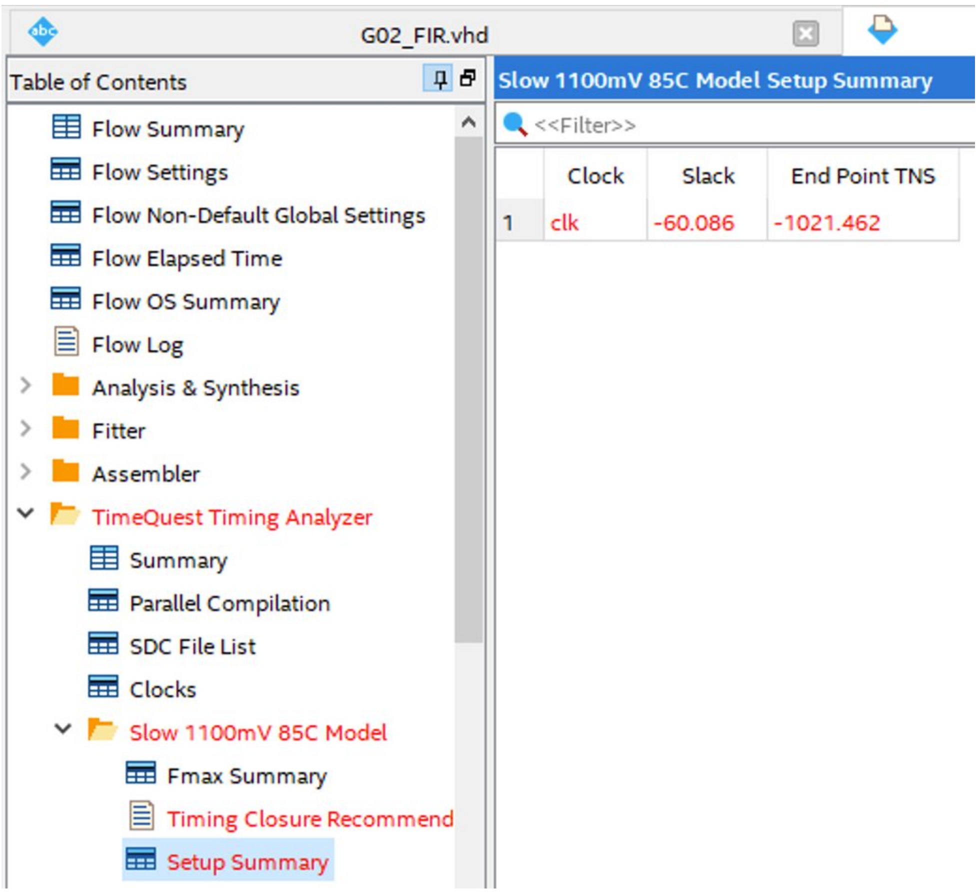


Figure 4.1-2 Setup Summary for FIR Filter

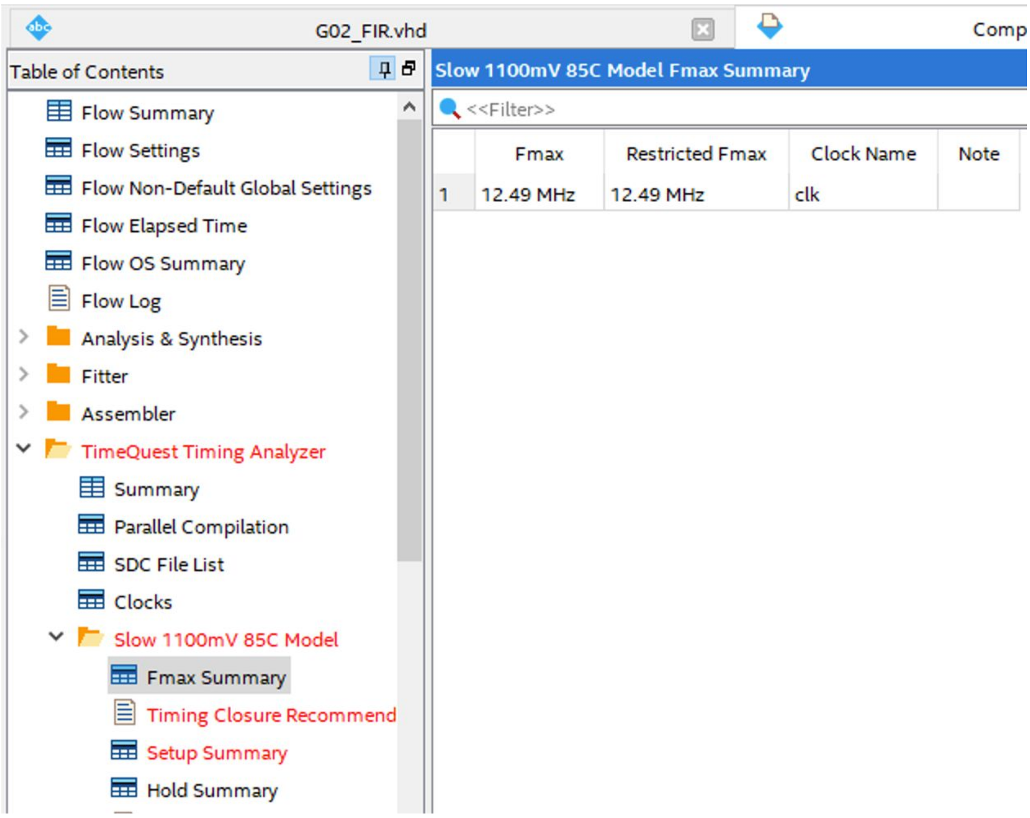


Figure 4.1-3 Fmax for FIR Filter

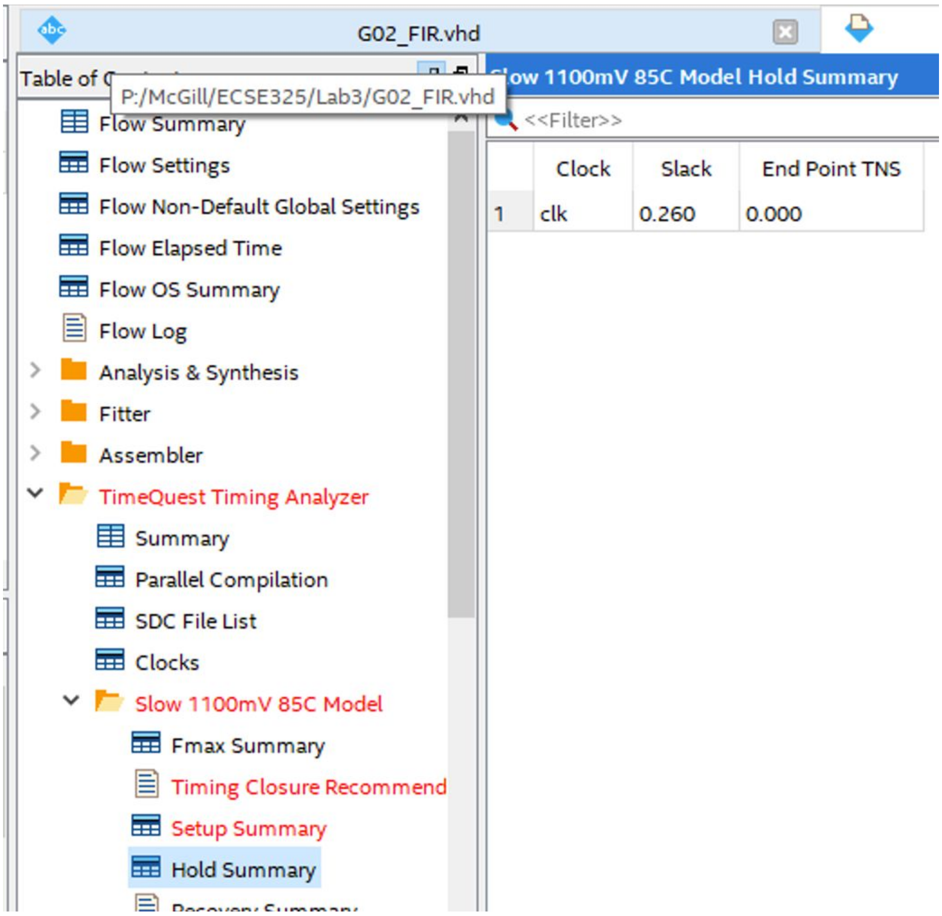
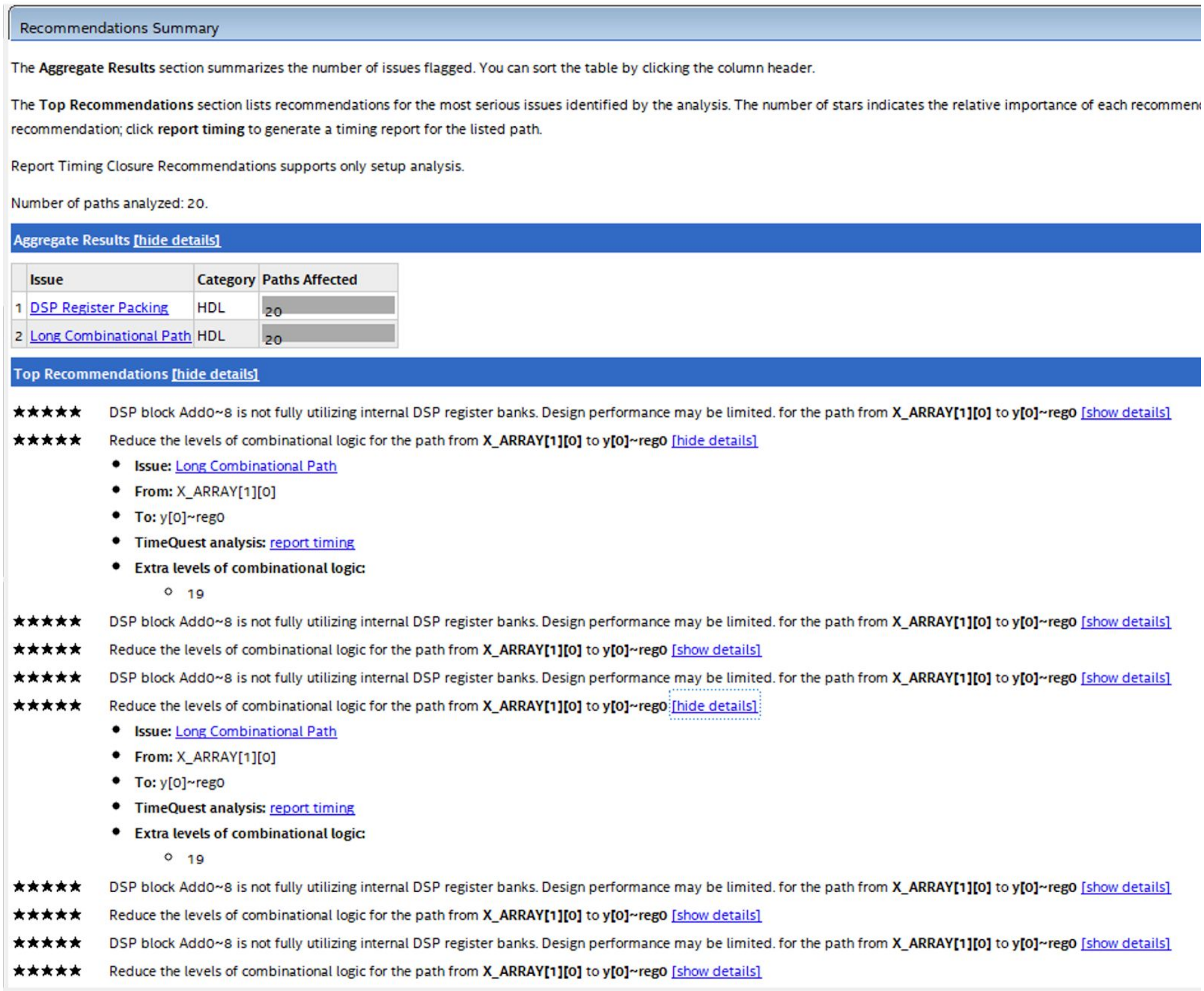


Figure 4.1-4 Hold Summary for FIR Filter



Recommendations Summary

The **Aggregate Results** section summarizes the number of issues flagged. You can sort the table by clicking the column header.

The **Top Recommendations** section lists recommendations for the most serious issues identified by the analysis. The number of stars indicates the relative importance of each recommen
recommendation; click **report timing** to generate a timing report for the listed path.

Report Timing Closure Recommendations supports only setup analysis.

Number of paths analyzed: 20.

Aggregate Results [hide details]

	Issue	Category	Paths Affected
1	DSP Register Packing	HDL	20
2	Long Combinational Path	HDL	20

Top Recommendations [hide details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [hide details]

- **Issue:** Long Combinational Path
- **From:** X_ARRAY[1][0]
- **To:** y[0]~reg0
- **TimeQuest analysis:** report timing
- **Extra levels of combinational logic:**
  - 19

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [hide details]

- **Issue:** Long Combinational Path
- **From:** X_ARRAY[1][0]
- **To:** y[0]~reg0
- **TimeQuest analysis:** report timing
- **Extra levels of combinational logic:**
  - 19

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **X_ARRAY[1][0]** to **y[0]~reg0** [show details]

Figure 4.1-5 Recommendations Summary for FIR Filter

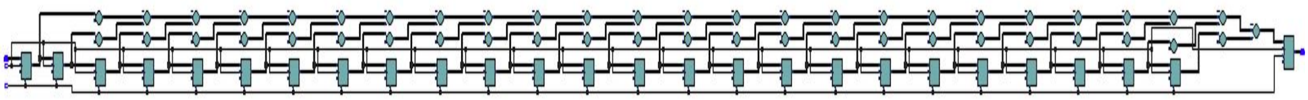


Figure 4.1-6 RTL Viewer for FIR Filter

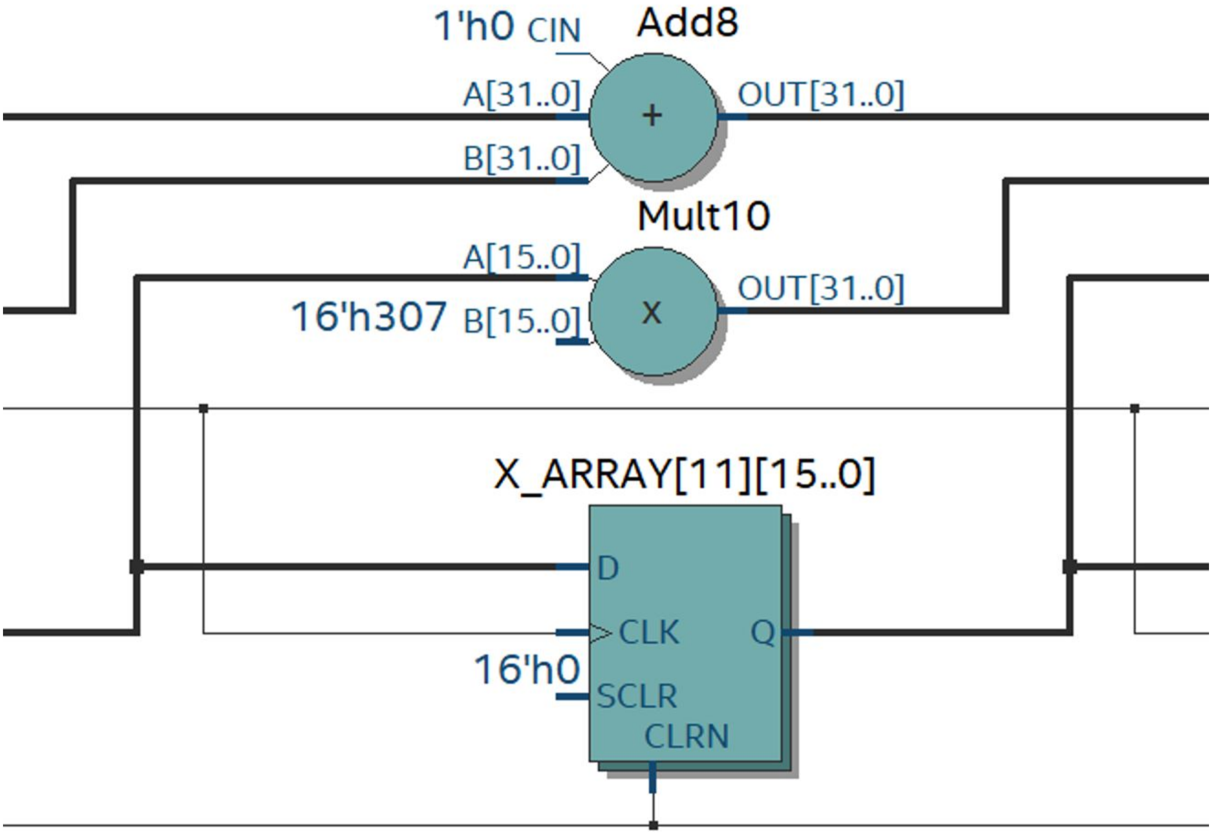


Figure 4.1-7 Detailed RTL Viewer for FIR Filter

The two figures shown above are the RTL viewer of our FIR filter design. *Figure 4.1-6* shows the overview of the RTL viewer, including a 25-times-repeated detailed RTL viewer, which is shown in *Figure 4.1-7*. The detailed RTL viewer shows one basic building block which includes one register, one adder and one multiplier. The register is used to store each number in the input sequence propagated during each clock cycle. The multiplier is used to multiply the output of each register by the corresponding weight while the adder is used to add all the output of each basic building block to calculate the total sum. Overall, this block is repeated 25 times since we are designing a 25-tap FIR filter.

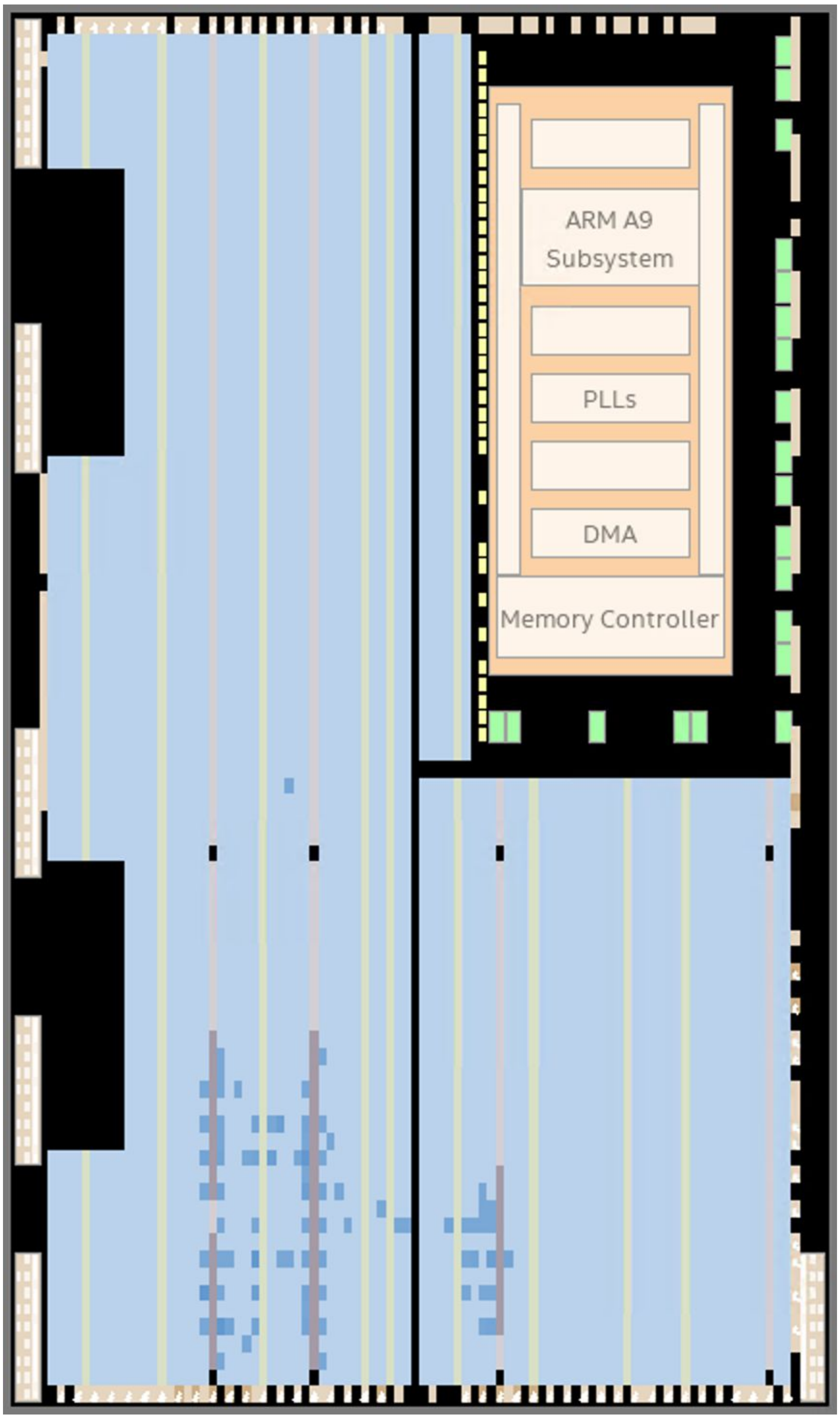


Figure 4.1-8 Chip Planner Overview for FIR Filter

Figure 4.1-8 shows an overview of the chip planner used on our VHDL code for the FIR filter. The chip planner provides a visual display of our post-place-and-route design mapped to the device architecture of our chosen FPGA and allows us to create, move, and delete logic cells and I/O atoms. Areas highlighted in dark illustrate the logical array blocks we used in order to implement our VHDL code.

4.2 FIR Filter Reducing Bitwidth

FIR filter with 3 bits input and 3 bits weight and it still has violation and the maximum frequency is 39.87MHz.

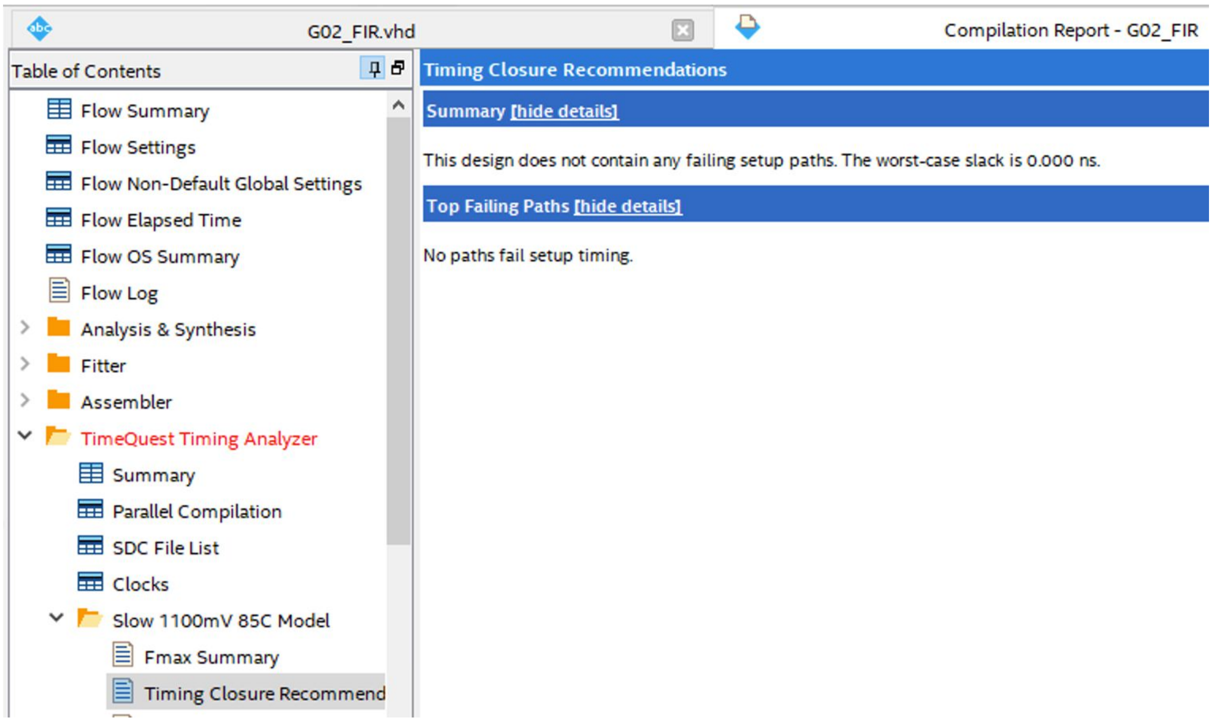


Figure 4.2-1 Timing Closure Recommendations for FIR Filter with Reducing Bitwidth

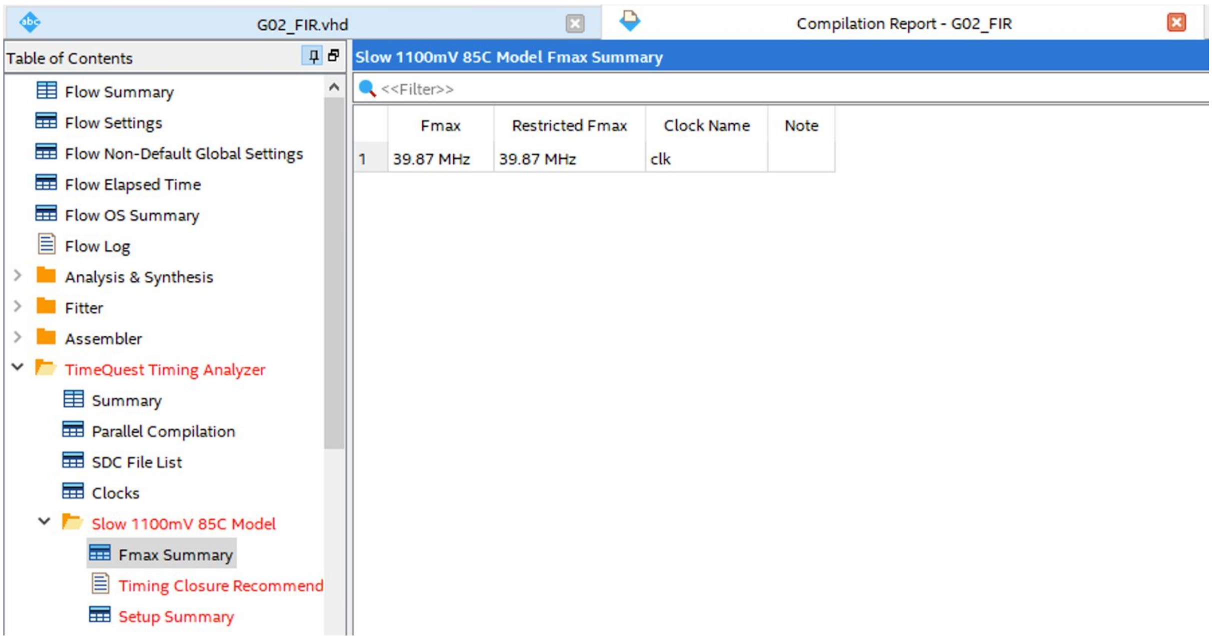


Figure 4.2-2 Fmax Summary for FIR Filter with Reducing Bitwidth

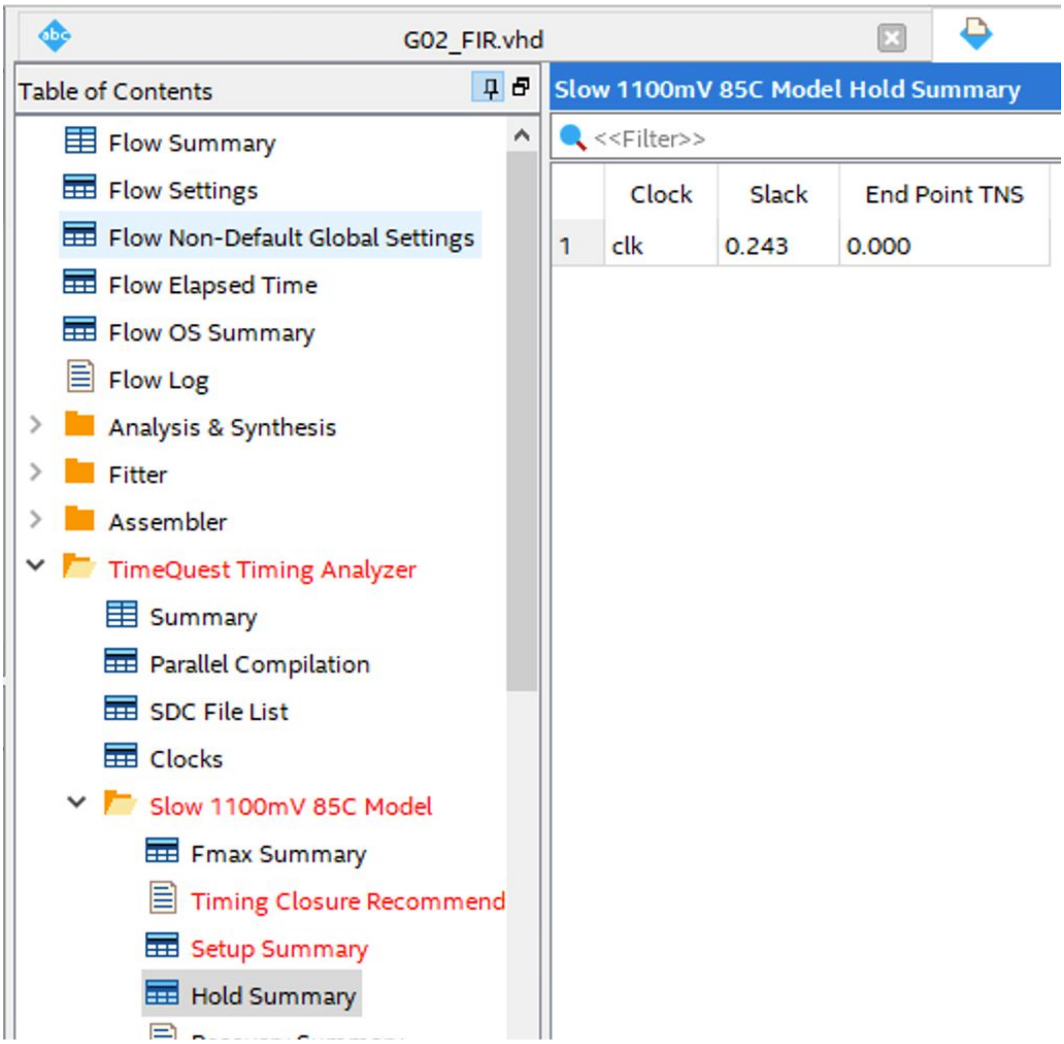


Figure 4.2-3 Hold Summary for FIR Filter with Reducing Bitwidth

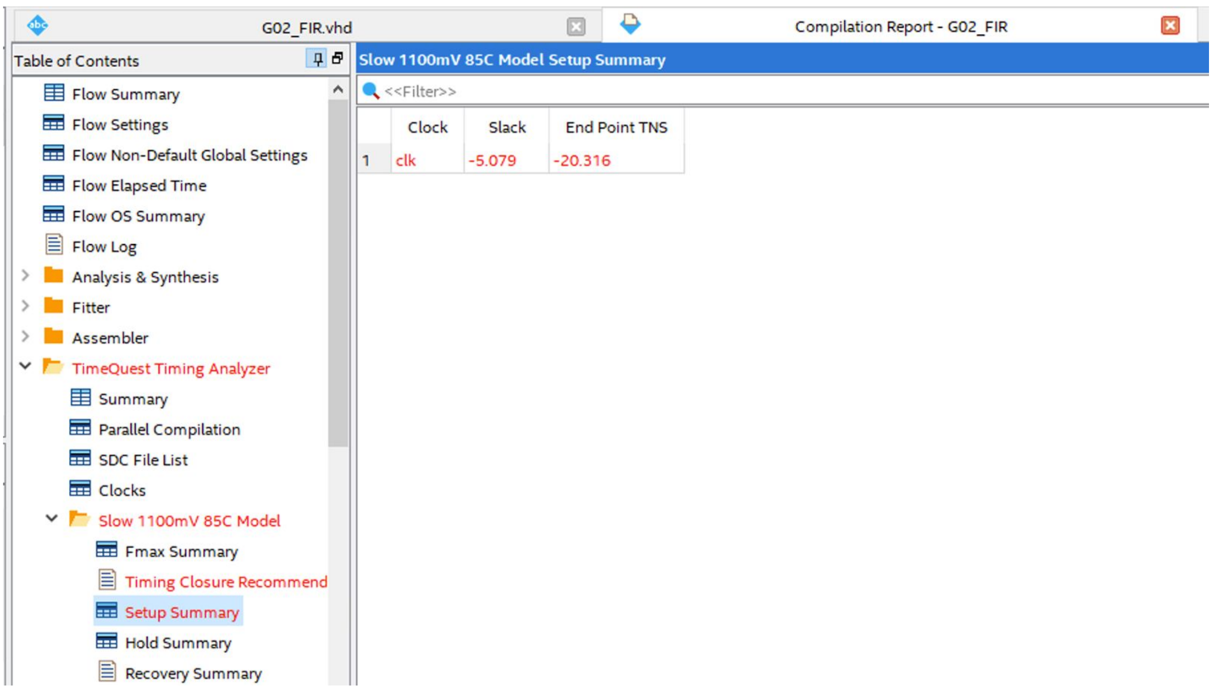


Figure 4.2-4 Setup Summary for FIR Filter with Reducing Bitwidth

4.3 Broadcast Filter

The following figures show that there is no timing violation in the timing analysis for broadcast filters. As a result of register rearrangement and design modification, the computation no longer needs to wait for the accumulated sum to be computed and it is now just computing the weight times input and adding the X_ARRAY(i+1) which is accessible immediately. According to Fmax Summary now the maximum frequency of the circuit is 257.86MHz. The figures of the chip planner and the RTL viewer of the VHDL code for the broadcast filter are also shown below.

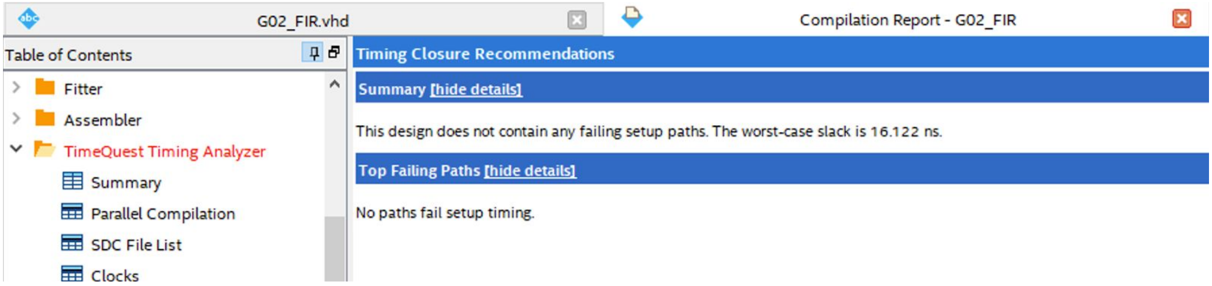


Figure 4.3-1 No Violation for Broadcast Filter

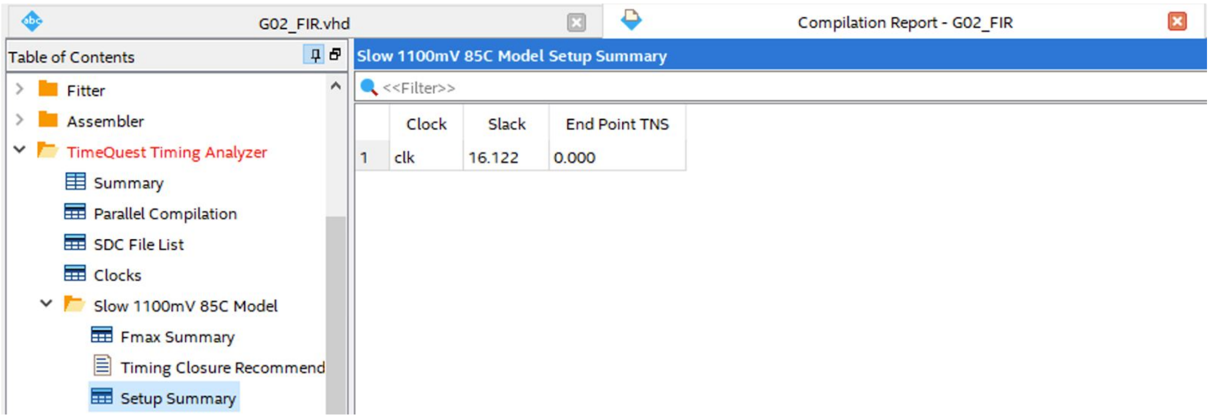


Figure 4.3-2 Broadcast Filter Setup Summary

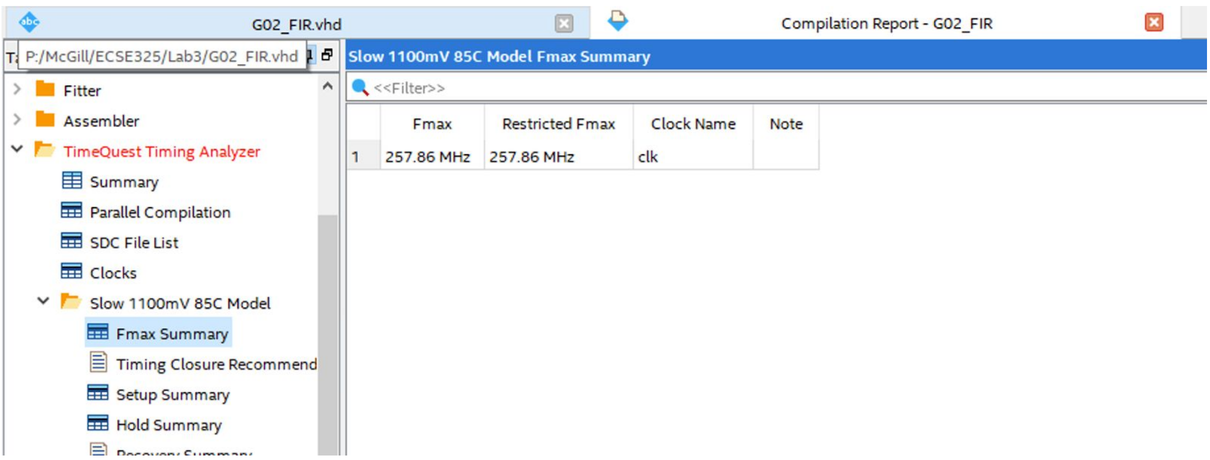


Figure 4.3-3 Broadcast Filter Fmax Summary

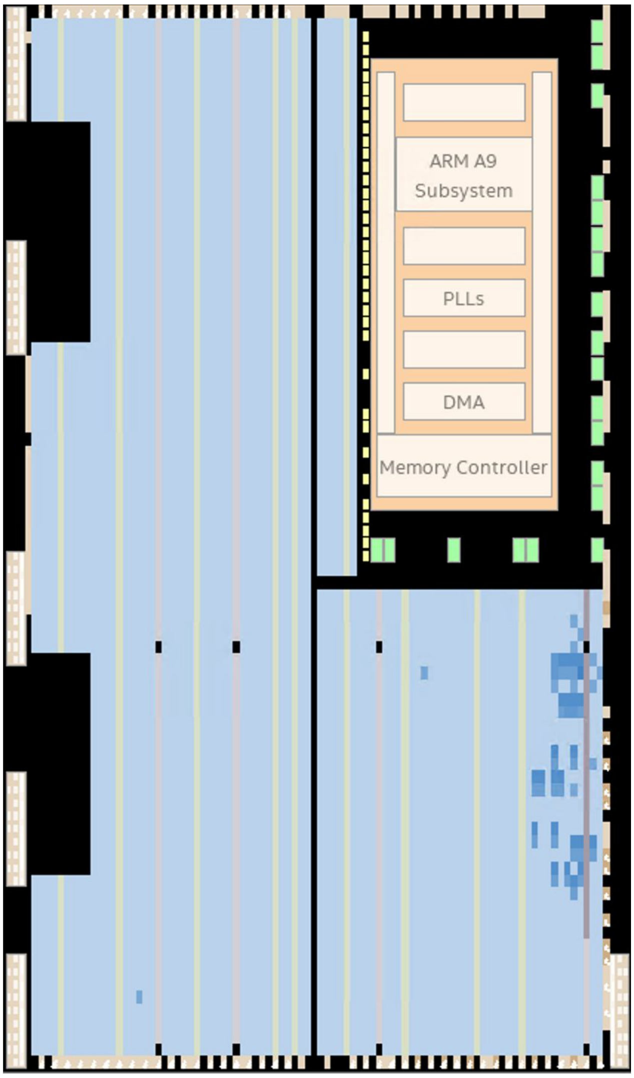


Figure 4.3-4 Chip Planner for Broadcast Filter

Figure 4.3-4 shows the overview of the chip planner used on our VHDL code for the broadcast filter. The areas highlighted in dark illustrate the logical array blocks we used in order to implement our VHDL code. The TRL viewer for the broadcast Filter is shown in *Figure 4.3-5* and *4.3-6* below:

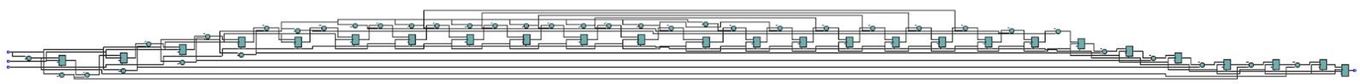


Figure 4.3-5 RTL Viewer for Broadcast Filter

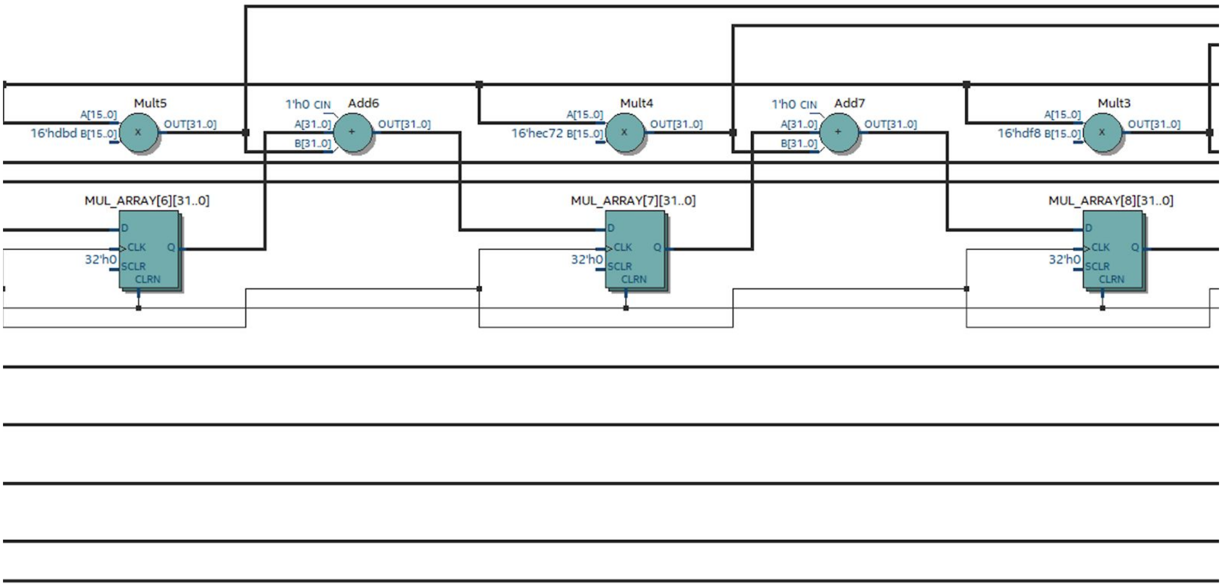


Figure 4.3-6 Detailed RTL Viewer for Broadcast Filter

5. Conclusion

In this laboratory, we learned the basic knowledge of specifying timing constraints and performing static timing analysis of the synthesized circuit. The simulation output result from our testbench complies with our estimation and satisfies the design requirements. Thus, we can conclude that our designs of FIR filter and broadcast filter are successful. This lab is also a useful practice for time-critical circuit design validation and simulation.