ECSE 325 - Lab Report 3

Group 42

Michael Frajman 260863814
Shi Tong Li 260857759

April 14, 2020

Introduction

The goal of this lab is to design a synchronous finite impulse response filter (FIR) with reset, input line and an output. The sizes of the input and coefficient of the VHDL code are 16 bits, and the output will be 17 bits. A script (similar to the one used in lab2) will convert both the given input and the coefficient to from decimal to fixed point representation. Once the code for both the script and FIR are written and the FIR code is compiled, the compilation report, chip planner, timing summaries and RTL viewer designs outputted by Quartus are to be observed and analyzed. The functionality of the FIR is tested in ModelSim using a testbench file which will take in the fixed point representation file from the script as data. This lab has the additional requirement of performing a timing analysis on the component in order to determine the maximum frequency that the design can attain. The timing analysis helps in illustrating the effects of pipelining and the placement of combinational logic components. Using feedback from the timing analysis, we can redesign our FIR in the so called “broadcasting form” to allow it to become more efficient.

Script and input files

Since we were provided with the word and fractional length for the fixed point representation of our data in the lab assignment document (with inputs and weights being 16 bit words with 15 bit fractional portions and the output needing a 17 bit word with 15 bit fraction), we opted to create a shorter script using prebuilt functions in Matlab as suppose to reusing our lab 2 Python scripts. Our Matlab script primarily utilized the built in “fi” function which translates regular decimal numbers into fixed point representation. After running the input file data through the “fi” function, the output was translated to a string of bits and printed to a text file. These text files of our inputs and weights could then be fed into our test bench code. Overall the ability to use Matlab over Python for creating our script due to having more information provided to us allowed for us to write shorter code in less time while still achieving the same amount of functionality.

```
%% Convert the coefficient/weight values from floats to fixed point

%read file, write contents to an array and close
coefFile = fopen("lab3-coef.txt");
coefArray = fscanf(coefFile, "%f");
fclose(coefFile);

%create a new array which runs contents of input array through fixed point
%representation function and binary form function
coefFixedPoint = bin(fi(coefArray,1,16,15)); %signed, 16bit word, 15bit fractional

%create a new output file and give it write permission, print the array to
%file and close

%contents of print statement allow array to be printed column wise
coefOutFile = fopen("lab3-coef-fixedOutput.txt", "w");
fprintf(coefOutFile, [repmat('%c',1,size(coefFixedPoint,2)) '\r\n'], transpose(coefFixedPoint));
fclose(coefOutFile);
```

Figure 1: Example of our Matlab code for translating input data to fixed point binary.

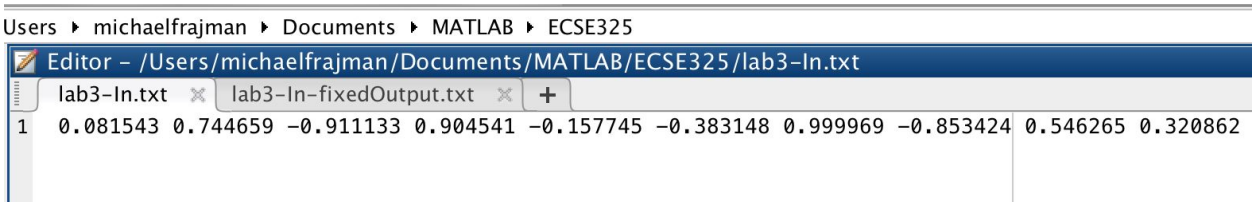

Figure 2: The first 10 values of the data input file as provided in the lab assignment.

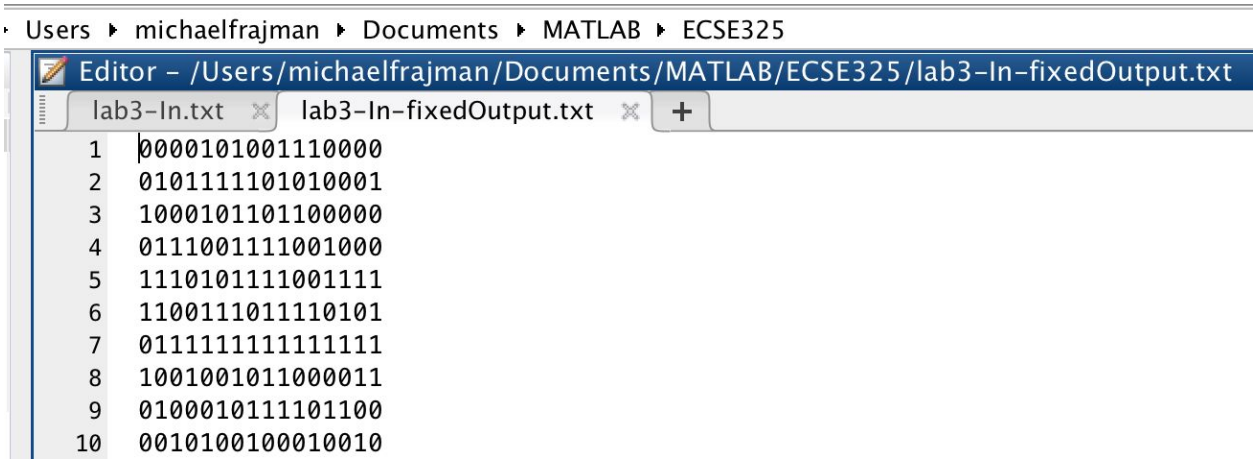

Figure 3: The first 10 values of the input data translated to fixed point binary by out script. Each number is a 16 bit word with a 15 bit fractional portion.

VHDL code for regular FIR

The direct layout of the FIR filter follows the discrete convolution $y(n) = \Sigma_{i=0}^{N} b_i * x(n-i)$.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   -- finite inpulse response filtre
6   entity g42_FIR is
7   port( x     :in std_logic_vector(15 downto 0); --input singla
8         clk   :in std_logic; --clock
9         rst   :in std_logic; --asynchronous active-high reset
10        y     :out std_logic_vector(16 downto 0) --output signal
11  );
12  end g42_FIR;
13
14  architecture a0 of g42_FIR is
15  type arr is array(24 downto 0) of signed(15 downto 0);
16  signal coeff: arr;
17  signal input: arr := (others => "0000000000000000");
18
19  begin
20  --values of the coeff
21  coeff(0) <= "0000001001110011";
22  coeff(1) <= "0000000000010001";
23  coeff(2) <= "1111111111010010";
24  coeff(3) <= "1111111011011101";
25  coeff(4) <= "0000001100011010";
26  coeff(5) <= "1111110110100111";
27  coeff(6) <= "1111110000001101";
28  coeff(7) <= "0000110110111101";
29  coeff(8) <= "1110110001110010";
30  coeff(9) <= "0000110111111000";
31  coeff(10) <= "0000001100001000";
32  coeff(11) <= "1110101000001010";
33  coeff(12) <= "0001111000110100";
34  coeff(13) <= "1110101000001010";
35  coeff(14) <= "0000001100001000";
36  coeff(15) <= "0000110111111000";
37  coeff(16) <= "1110110001110010";
38  coeff(17) <= "0000110110111101";
39  coeff(18) <= "1111110000001101";
40  coeff(19) <= "1111110110100111";
41  coeff(20) <= "0000001100011010";
42  coeff(21) <= "1111111011011101";
43  coeff(22) <= "1111111111010010";
44  coeff(23) <= "0000000000010001";
45  coeff(24) <= "0000001001110011";
46
```

Image 4: VHDL code for our direct FIR part 1.

```
process(clk, rst)
    variable temp: signed(31 downto 0) := (others => '0');
    variable output: signed(16 downto 0) := (others => '0');
 begin
    --for reset, reset all values
    if rst = '1' then
       temp := (others => '0');
       output := (others => '0');
       y <= (others => '0');
    --rising clock tick
    elsif rising_edge(clk) then
       output := (others => '0');
       --shifting input
       for i in 0 to 23 loop
          input(1+i) <= input(i);
       end loop;
       input(0) <= signed(x);

       --take the total sum of all the multiplication of coeff and delayed input
       for i in 0 to 24 loop
          temp := coeff(24-i)*input(i);
          output := output + temp(31 downto 15);
       end loop;
       --output
       y <= std_logic_vector(output);
    end if;
 end process;


 end a0;
```

Image 5: VHDL code for our direct FIR part 2.

For simplicity, the 25 coefficient values were not read from the file, but rather put into an array in our VHDL code for ease of access. We had to implement the formula given by the lab3 instructions, which was essentially to multiply the shifted input array by the corresponding coefficient and the result is fed to the output.

VHDL code for broadcasting FIR

The broadcasting method uses a transposed layout when compared to the direct form of the FIR. It follows the formula: $y(n) = \Sigma_{i=0}^{N} b_{25-i} * x(n)$.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity g42_FIRb is
port( x     :in std_logic_vector(15 downto 0); --input singla
      clk   :in std_logic; --clock
      rst   :in std_logic; --asynchronous active-high reset
      y     :out std_logic_vector(16 downto 0) --output signal
);
end g42_FIRb;

architecture a0 of g42_FIRb is
type arr is array(24 downto 0) of signed(15 downto 0);
signal coeff: arr;
type arrtemp is array(24 downto 0) of signed(31 downto 0);
signal arrout: arrtemp;

begin
--values of the coeff
coeff(0) <= "0000001001110011";
coeff(1) <= "0000000000010001";
coeff(2) <= "1111111111010010";
coeff(3) <= "1111111011011101";
coeff(4) <= "0000001100011010";
coeff(5) <= "1111110110100111";
coeff(6) <= "1111110000001101";
coeff(7) <= "0000110110111101";
coeff(8) <= "1110110001110010";
coeff(9) <= "0000110111111000";
coeff(10) <= "0000001100001000";
coeff(11) <= "1110101000001010";
coeff(12) <= "0001111000110100";
coeff(13) <= "1110101000001010";
coeff(14) <= "0000001100001000";
coeff(15) <= "0000110111111000";
coeff(16) <= "1110110001110010";
coeff(17) <= "0000110110111101";
coeff(18) <= "1111110000001101";
coeff(19) <= "1111110110100111";
coeff(20) <= "0000001100011010";
coeff(21) <= "1111111011011101";
coeff(22) <= "1111111111010010";
coeff(23) <= "0000000000010001";
coeff(24) <= "0000001001110011";
```

Image 6: VHDL code for our broadcast/transpose FIR part 1.

```
process(clk, rst)
    variable temp: signed(31 downto 0) := (others => '0');
    variable output: signed(16 downto 0) := (others => '0');
 begin
    --for reset, reset all values
    if rst = '1' then
       temp := (others => '0');
       output := (others => '0');
       y <= (others => '0');
    --rising clock tick
    elsif rising_edge(clk) then
       output := (others => '0');
       --do multiplication and put in out array
       for i in 0 to 24 loop
          arrout(i) <= coeff(i)*signed(x);
       end loop;

       --take the total sum of all the multiplication of coeff and input
       for i in 0 to 24 loop
          temp := arrout(i);
          output := output + temp(31 downto 15);
       end loop;
       --output
       y <= std_logic_vector(output);
    end if;
 end process;


 end a0;
```

Image 7: VHDL code for our broadcast/transpose FIR part 2.

The broadcasting FIR filter differs from the normal one due to its ability to multiply first, then shift, as opposed to shift then multiply. For the VHDL code, we have the same entity set up as the normal FIR. Then, we have an array arrout to store the multiplication results. At the rising clock edge, the results of the multiplication of the coefficient and the unshifted input will be stored in the arrout array. Finally, a loop will take into account the shift and add these multiplication together.

Compilation report

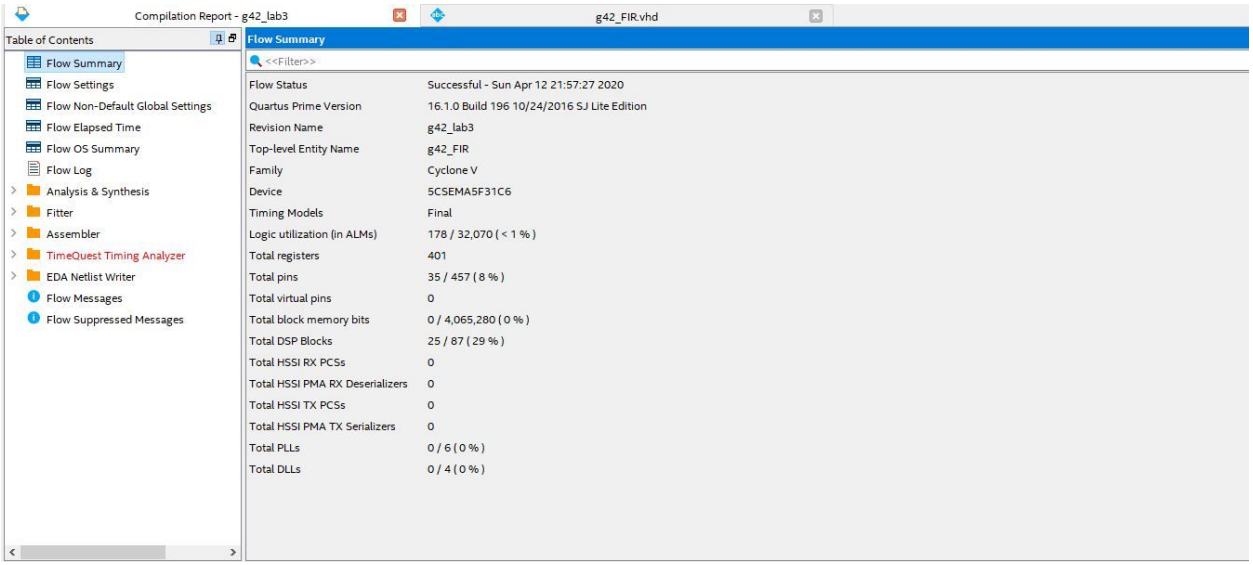

Image 8: Compilation report/flow summary of our counter design.

Resource utilization

According to the compilation report, image 6, there are 401 registers used for a 25-tap FIR filter (16 bits for the input and coefficient, 17 for the output, 1 for clock, 1 for ready and 1 for reset). Since the number of registers used is directly proportional to the number of bits of the input and output, we can expect a linear increase or decrease in register utilization when adding or removing bit in the size input or the output.

Chip planner

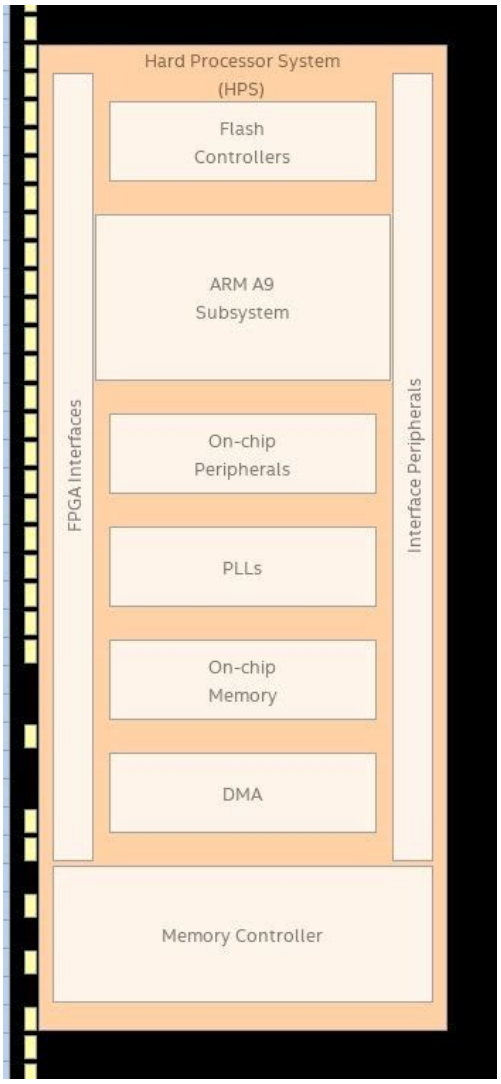

Image 9: Closeup of the processor of the FPGA board from the chip planner.

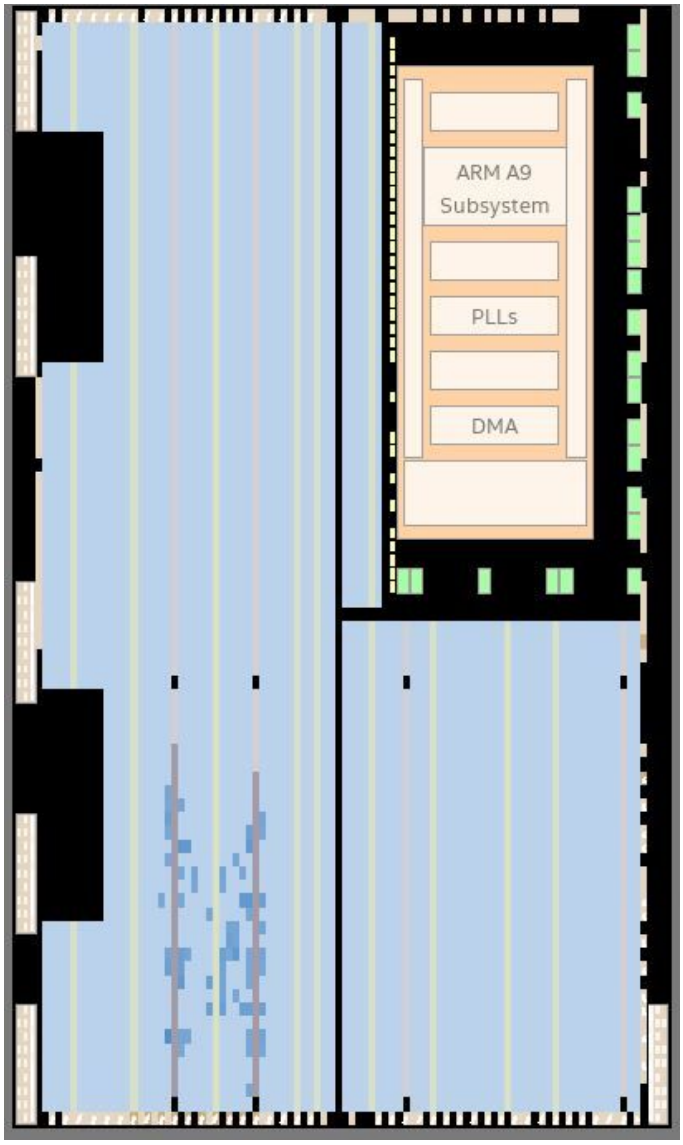

Image 10: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue rectangles, namely the large cluster in the bottom left.

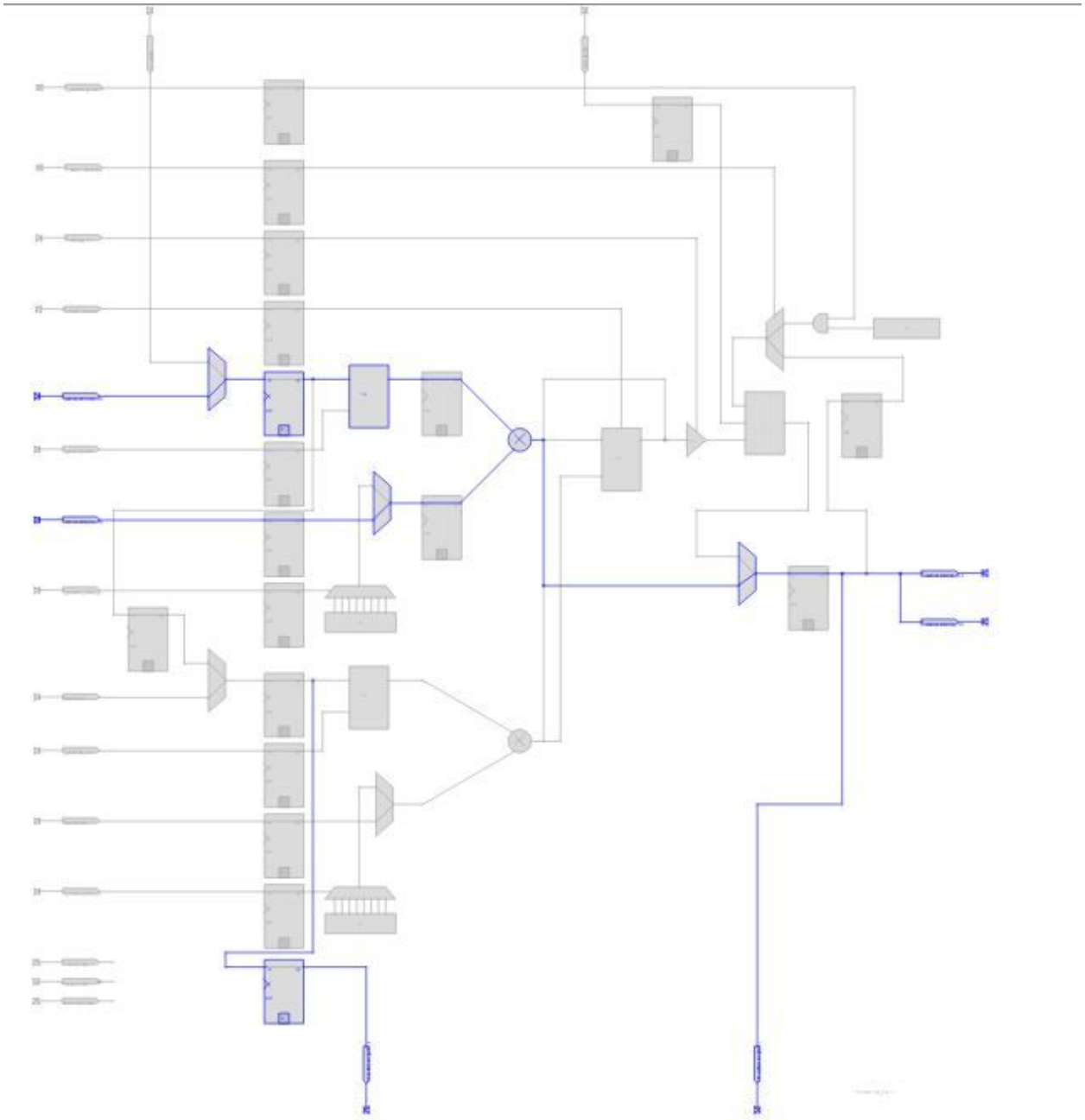

Image 11: Logic circuit layout of a look-up-table being used by the FPGA board to produce our direct FIR. This image is from the chip planner as well.

RTL view

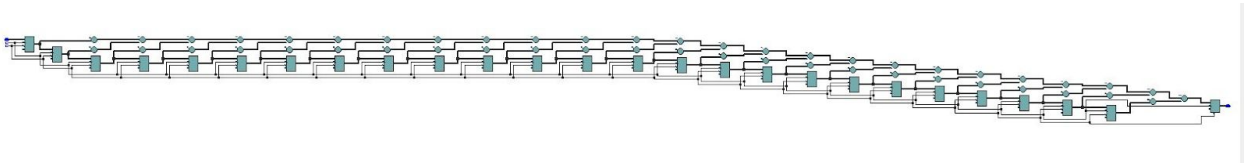

Image 12: The RTL/logic diagram of our direct FIR design.

From the RTL viewer screenshot, it can be seen that we are using 26 registers, 24 multipliers, and 26 adders. The control signals on the first register are used to implement the ready and reset.The bottom register at the end is used to store the 17-bit number of the FIR filter in order to output it.

Testbench code

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

entity g42_FIR_tb is
end g42_FIR_tb;

architecture testbench of g42_FIR_tb is



   component g42_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 g42_FIR;

   file file_IN      : text;
   file file_RESULTS    : text;

   constant clk_PERIOD : time := 60 ns;

   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
      g42_FIR_test : g42_FIR
      port map (
         x  => x_in,
         clk   => clk_in,
         rst   => rst_in,
         y     => y_out
      );

--CLOCK GENERATION
   clk_gen : process
       begin
           clk_in <= '1';
           wait for clk_PERIOD / 2;
           clk_in <= '0';
           wait for clk_PERIOD / 2;
   end process clk_gen;
```

Image 13: The code of the FIR testbench file part 1.

```
   feeding_instr : process is
      variable v_lline1 : line;
      variable v_lline2 : line;
      variable v_oline  : line;
      variable v_x_in   : std_logic_vector(15 downto 0);

   begin

      rst_in <= '1';
      wait until rising_edge(clk_in);
      wait until rising_edge(clk_in);
      rst_in <= '0';

      file_open(file_IN, "C:\Users\sli196.CAMPUS\ECSE325\scripts\lab3\lab3-In-fixedOutput.txt", read_mode);
      file_open(file_RESULTS, "C:\Users\sli196.CAMPUS\ECSE325\scripts\lab3\lab3-out.txt", write_mode);

      while not endfile (file_IN) loop
         readline(file_IN, v_lline1);
         read(v_lline1, v_x_in);
         x_in <= v_x_in;
         write(v_oline, y_out);
         writeline(file_RESULTS, v_oline);
         wait until rising_edge(clk_in);
       end loop;
       wait until rising_edge(clk_in);
       wait until rising_edge(clk_in);
       wait;
   end process feeding_instr;
end testbench;
```

Image 14: The code of the FIR testbench file part 2.

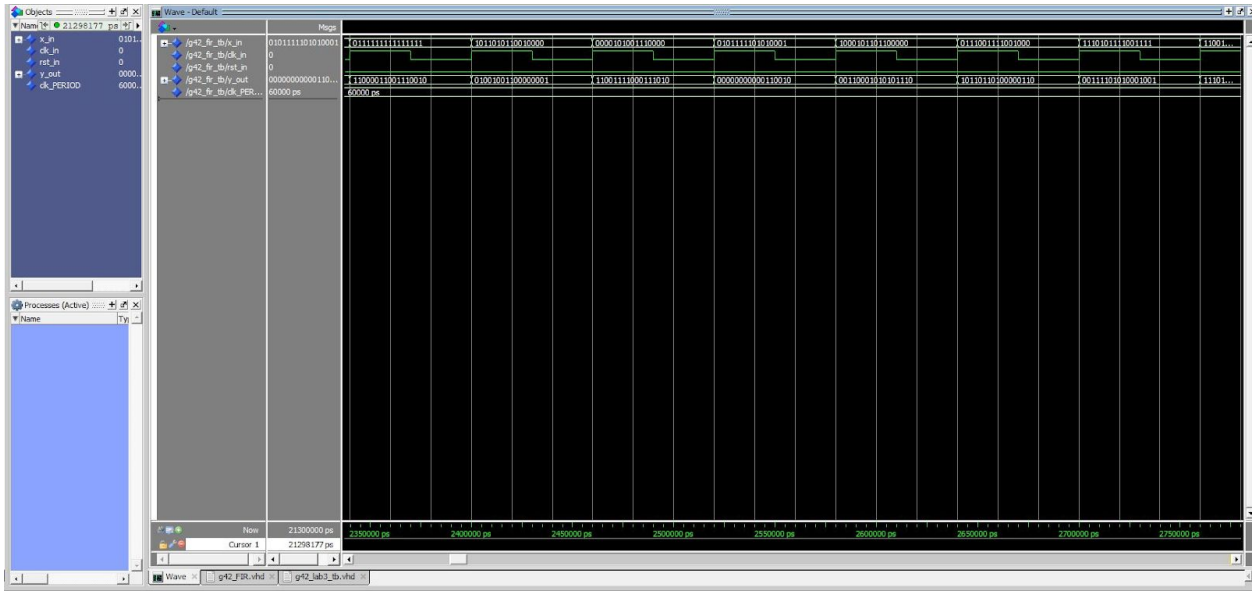

Image 15: The simulation results of the FIR testbench file.

Overall our testbench code was mostly drawn from that provided in the lab2 assignment document with us only modifying the paths to our data input file in our file directory. Our testbench code was able to take in our input data and provide outputs.

Users ▸ michaelfrajman ▸ Documents ▸ MATLAB ▸ ECSE325

Editor - /Users/michaelfrajman/Documents/MATLAB/ECSE325/lab3-theoretical-output.txt

lab3-theoretical-output.txt

```
1  00000000000110011
2  00000000111010011
3  11111110111001101
4  00000000111101011
5  11111111100111100
6  00000001000001000
7  11111110010000010
8  00000000101010101
9  00000101100100010
10 11110000101111111
```

Image 16: The fixed point representation of the first 10 values of the provided outputs generated by our Matlab script.

```
0000000000110011
0000000111010011
1111111011100 1100
0000000111101100
1111111100111100
0000001000001000
1111111001000 0011
0000000101010100
0000101100100100
1111000010111 1110
```

Image 17: The output of the first 10 values of our testbench for the direct FIR.

SDC

```
1  create_clock -period 20 [get_ports clk]
```

	Fmax	Restricted Fmax	Clock Name	Note
1	84.45 MHz	84.45 MHz	clk	

Image 18: SDC outputs for direct FIR.

Timing for regular FIR: the timing analysis for the regular FIR shows no sign of failing paths, and the maximum F_{max} of this filter is 84.45MHz.

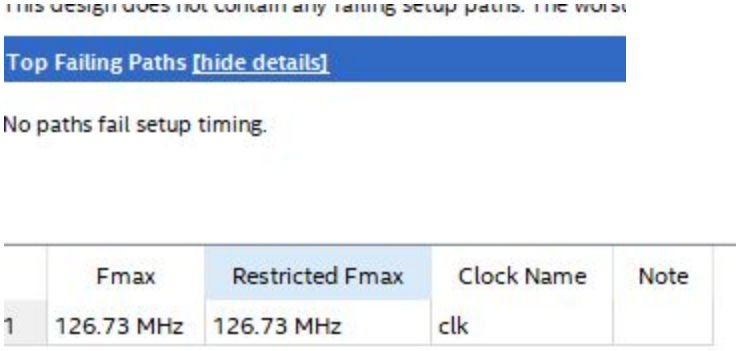

Image 19: SDC outputs for transpose/broadcast FIR.

Timing for broadcast FIR: the timing analysis for the broadcast FIR shows no sign of failing paths, and the maximum F_{max} of this filter is 126.73MHz.

Conclusion

This lab provided us with an experience to learn the process of pipelining and component placement and the effects it will have on a circuits timing. Both our direct and broadcast filtres possessed no timing constraint violations. Our broadcast form FIR also demonstrated how variations in placement can lead to a faster signal when compared to direct form. In theory both the direct FIR and its transpose, the broadcast FIR, would produce similar results in terms of timing as input goes through a similar delay for each weight (for example the input would experience two units of delay as it goes through weight b_0). However this timing difference is not trivial, as images 18 and 19 suggest, with the broadcast from being over 40MHz faster.

In terms of the correctness of the output, our design seems to be effective. Given images 16 and 17, the fixed point output from both our testbench and those from our script generated from the provided assignment files are one-to-one.

For our script for generating fixed point numbers, as discussed at the start of the lab, we opted to create a new one as opposed to reusing that from lab2. This decision was a good one in retrospect as we achieved the same results with shorter and more readable code.