ECSE 325 - Lab Report 2

Group 42

Michael Frajman 260863814
Shi Tong Li 260857759

March 30, 2020

Introduction

The goal of this lab is to design a synchronous multiply-accumulate (MAC) unit with reset, ready and two input lines and an output. The sizes of the input and output in the VHDL code are to be determined by the use of a script. The script will parse the provided input files of numbers and determine the number of bits required for these numbers fixed point representation, as well as convert the input file numbers to that fixed point representation and put them in an output file . Once the code for both the script and MAC are written and the MAC code is compiled, the compilation report, chip planner and RTL viewer designs outputted by Quartus are to be observed and analyzed. Finally, the functionality of the MAC is tested in ModelSim using a testbench file which will take in the fixed point representation file from the script as data.

Script and input files

(As an aside our group was part of the Friday lab section so we used the ThuFrilab2-x.txt and ThuFrilab2-y.txt files for our input values for our script.)

Since all groups were free to use the programming language of choice for our script we chose Python do to its easy to understand syntax and our familiarity with the language. Our script fulfilled all requirements as outlined in the assignment document. We determined the binary length of the decimal word by first finding the maximum and minimum whole number value among the inputted data and then determining if those values fall into a certain range of -2^{n-1} to $2^{n-1}-1$. For determining the length of the fractional portion of the input data we used regular expression and treated the input data as a string instead of as floats. The script would then convert all inputs to fixed point representation using the calculated precision and create an output file.

```
/Users/michaelfrajman/PycharmProjects/ecse325lab2/venv/bin/python /Users/michaelfrajman/PycharmProjects/ecse325lab2/main.py
****************FOR X INPUTS****************

Highest value word from list: 3
Lowest value word from list: -3
Lowest number of bits needed for word value range: 3
Longest fractional portion of the value of list (in digits): 7
Output file ready for X inputs.

****************FOR Y INPUTS****************

Highest value word from list: 3
Lowest value word from list: -3
Lowest number of bits needed for word value range: 3
Longest fractional portion of the value of list (in digits): 7
Output file ready for Y inputs.


Process finished with exit code 0
```

Image 1: The console output of our script.

ThuFrilab2-x.txt

```
3.09375 3.984375 3.203125 -3.921875 -3.03125 3.625 -1.40625 2.484375 -0.78125 1.375
```

Image 2: An example of the first 10 digits as provided in the ThuFrilab2-x.txt input file.

lab2-x-binary.txt

```
0110001100
0111111110
0110011010
1000001010
1001111100
0111010000
1101001100
0100111110
1110011100
0010110000
```

Image 3: An example of our script's output, the same first 10 values of the x input file (imag 2) in fixed point representation using 3 bits for the word and 7 bits for the fractional portion as shown in the console output (image 1).

VHDL code for MAC

g42_lab2.vhd | Compilation Report - g42_lab2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity g42_lab2 is
   Port( x: in std_logic_vector(9 downto 0); --the first digit to multiply from the x file, found to be 10 bits from script
         y: in std_logic_vector(9 downto 0); --the second digit to multiply for the y file, found to be 10 bits from script
         N: in std_logic_vector(9 downto 0);    --the total number of numbers in the file (1000 = 0b1111101000)
         clk: in std_logic;                     --clock signal
         rst: in std_logic;                     --reset
         mac: out std_logic_vector(19 downto 0);   --the product
         ready: out std_logic);                 --0 when accessing and accumulating from txt file, 1 when ready for another batch
end g42_lab2;

architecture behavioral of g42_lab2 is
   signal temp: signed(19 downto 0);
   signal i : integer :=0;
   signal readyTemp: std_logic := '0';
   begin
   process(clk, rst)
      begin
      if rst = '1' then
         temp <="00000000000000000000";
      elsif rising_edge(clk) then
         temp <= temp + signed(x) * signed(y); --taken from pseudocode
         i <= i+1;
         if(i = 1000) then
            readyTemp <= '1';
         end if;
      end if;
   end process;
   ready <= readyTemp;
   mac <= std_logic_vector(temp);
end behavioral;
```

Image 4: VHDL code for our MAC.

The above image shows the code for our MAC as specified in the lab 2 assignment.
We have first the entity block given in the lab 2 instructions. The architecture of the MAC consists of a process block that stores each of the multiplications into our “temp” (temporary storage variable) each clock cycle. There is also an asynchronous reset of the “temp” which can reset the count at any time as well as a ready signal which goes high when the mac is not counting. Both of these signals would be needed for batching, that is separating the input data into smaller batches to save on space on registers for fixed point representation, as supposed to processing all data in one file.

Compilation report

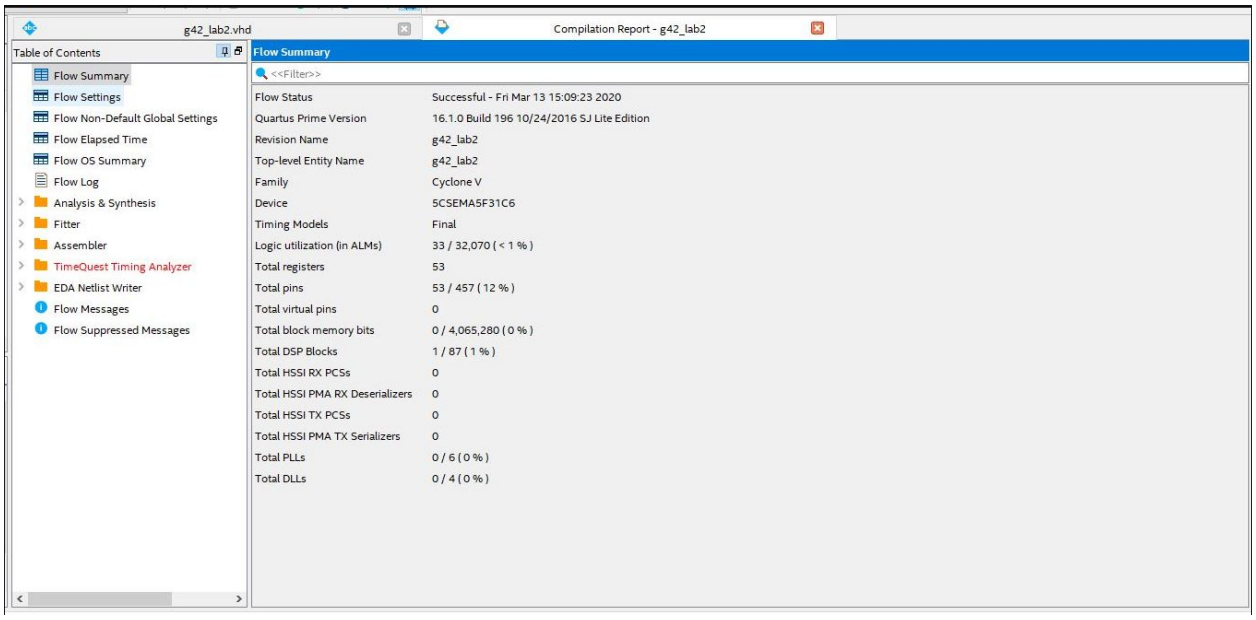

Image 5: Compilation report/flow summary of our counter design.

Resource utilization

According to the compilation report, image 5, there are 53 registers used for a 20-bit output MAC (10 bits for the x input, 10 for the y input, 10 to store the count N, 20 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. One possible means of decreasing the number of bits needed for the input and output is by feeding data to the MAC in optimized batches.

Chip planner

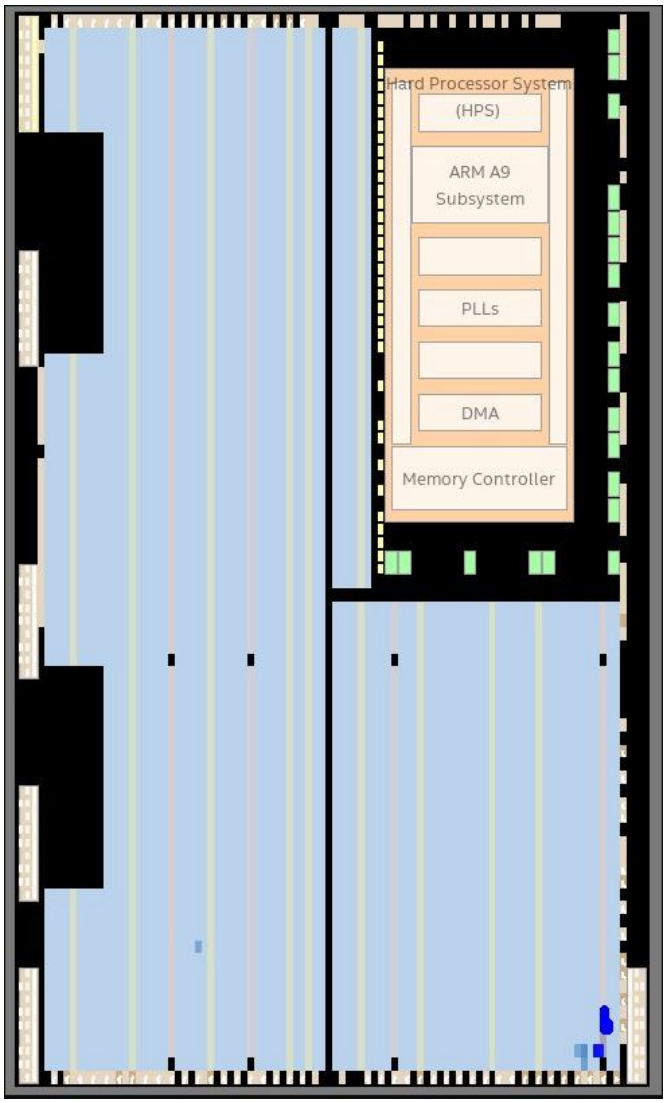

Image 6: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue rectangles (eg. one such rectangle is near the bottom of the chip toward the far left side).

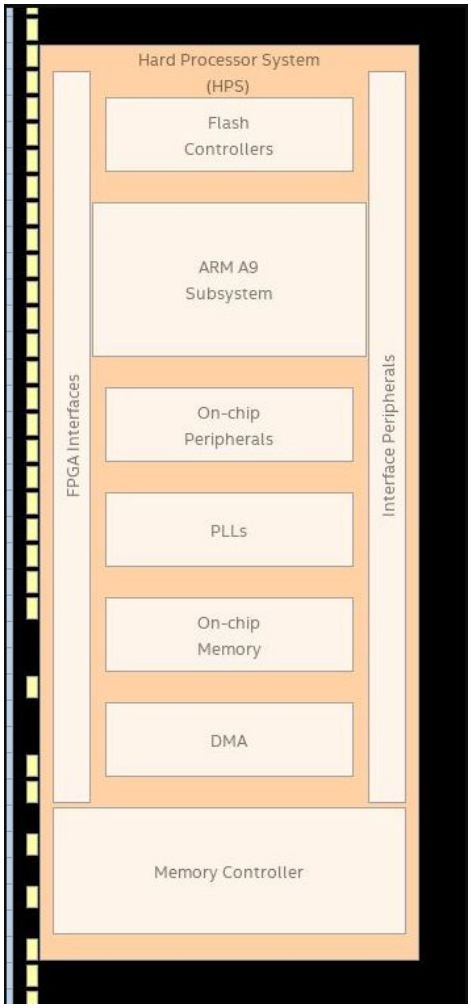

Image 7: Closeup of the processor of the FPGA board from the chip planner.

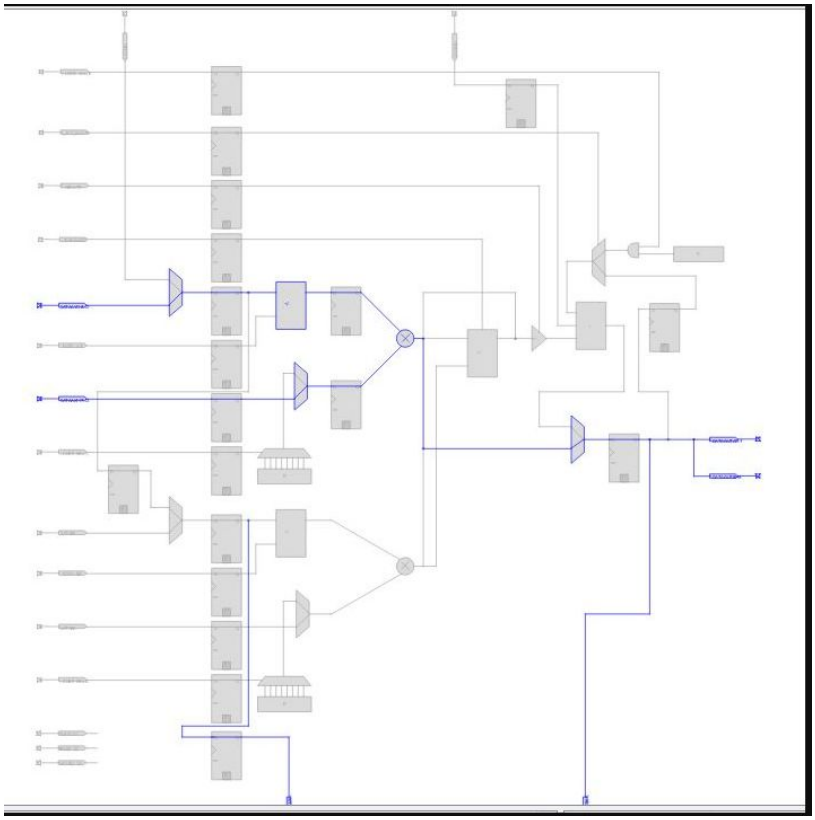

Image 8: Logic circuit layout of a look-up-table being used by the FPGA board to produce our MAC. This image is from the chip planner as well.

RTL view

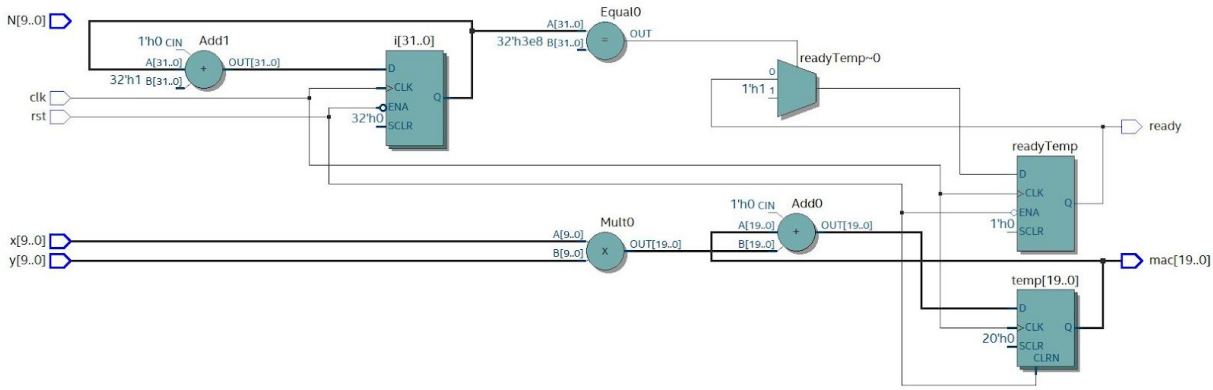

Image 9: The RTL/logic diagram of our MAC design.

From the RTL viewer screenshot, image 9, it can be seen that we are using 2 adders, 1 multiplexer, 1 multiplier, 1 equalizer and 3 registers. The control signals on the multiplexers are used to implement the ready and reset.The bottom register at the end is used to store the 20-bit number of the MAC in order to output it.

Testing

```
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_MAC_tb is
end g42_MAC_tb;

architecture test of g42_MAC_tb is

-- Declare the Component under test
  component g42_MAC is
   port(
      x     : in std_logic_vector(9 downto 0); -- first input
      y     : in std_logic_vector(9 downto 0); -- second input
      N     : in std_logic_vector(9 downto 0); -- total number of inputs
      clk   : in std_logic; --clock
      rst   : in std_logic; --asynchronous active - high reset
      mac   : out std_logic_vector(19 downto 0); -- output of MAC unit
      ready : out std_logic -- denotes the validity of the mac signal
   );
  end component g42_MAC;

  ---- Testbench internal signals
  file file_VECTORS_X : text;
  file file_VECTORS_Y : text;
  file file_RESULTS   : text;

  constant clk_PERIOD : time := 100 ns;

  signal x_in     :  std_logic_vector(9 downto 0);
  signal y_in     :  std_logic_vector(9 downto 0);
  signal N_in     :  std_logic_vector(9 downto 0);
  signal clk_in   :  std_logic;
  signal rst_in   :  std_logic;
  signal mac_out   : std_logic_vector(19 downto 0);
  signal ready_in :  std_logic;

begin

  -- instantiate MAC
   g42_MAC_INST : g42_MAC
   port map (
        x => x_in,
            y => y_in,
            N => N_in,
            clk => clk_in,
            rst => rst_in,
            mac => mac_out,
            ready => ready_in
  );
```

Image 10: The code of the MAC testbench file part 1.

```
  --  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;

  --Feeding Inputs
  feeding_instr : process is
    variable v_lline1 : line;
    variable v_lline2 : line;
    variable v_oline  : line;
    variable v_x_in   : std_logic_vector(9 downto 0);
    variable v_y_in   : std_logic_vector(9 downto 0);
  begin
      --reset the circuit
      N_in <= "1111101000";  --N = 1000
      rst_in <= '1';
      wait until rising_edge(clk_in);
      wait until rising_edge(clk_in);
      rst_in <= '0';

      file_open(file_VECTORS_X, "C:\Users\sli196\ECSE325\scripts\lab2\ecse325lab2\lab2-x-binary.txt", read_mode);
   file_open(file_VECTORS_Y, "C:\Users\sli196\ECSE325\scripts\lab2\ecse325lab2\lab2-y-binary.txt", read_mode);
   file_open(file_RESULTS, "C:\Users\sli196\ECSE325\scripts\lab2\ecse325lab2\lab2-out.txt", write_mode);

      while not endfile (file_VECTORS_X) loop
         readline(file_VECTORS_X, v_lline1);
         read(v_lline1, v_x_in);
         readline(file_VECTORS_Y, v_lline2);
         read(v_lline2, v_y_in);

         x_in <= v_x_in;
         y_in <= v_y_in;

         wait until rising_edge(clk_in);
      end loop;

   if ready_in = '1' then
   write(v_oline, mac_out);
   writeline(file_RESULTS, v_oline);
   wait;
   end if;


      wait;
  end process;
end test;
```

Image 11: The code of the MAC testbench file part 2. (Note: this testbench file was created in accordance to/modified from the lab 2 instructions.)

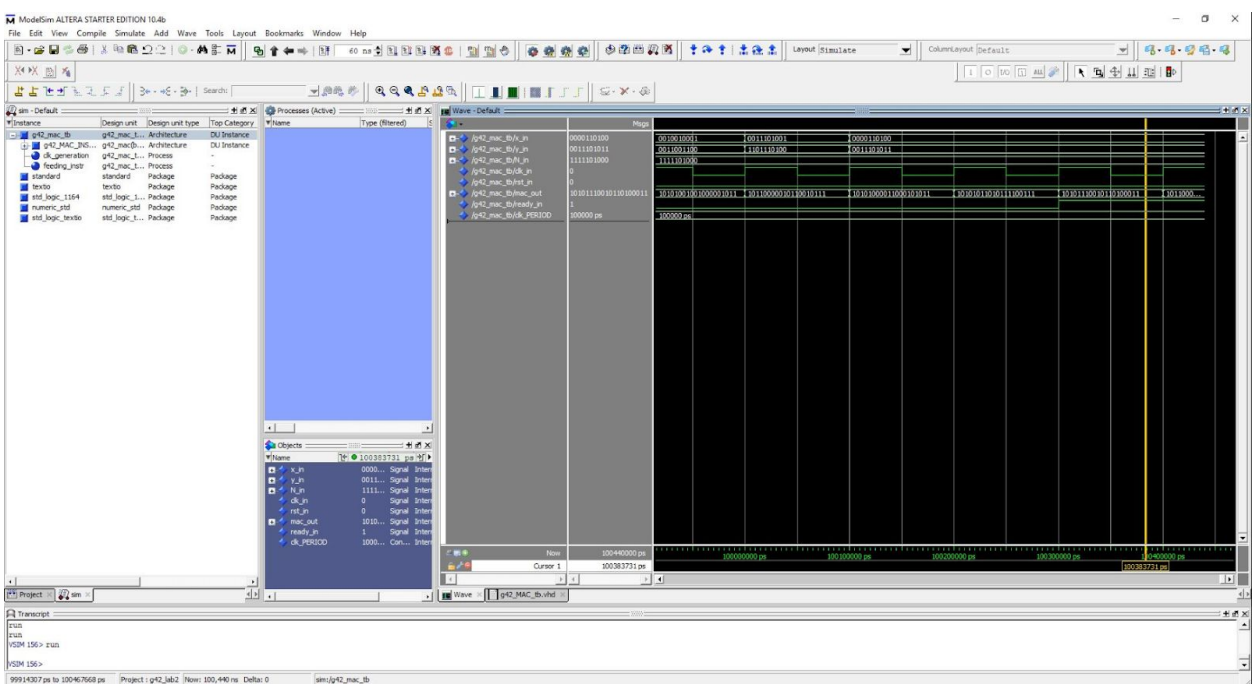

Image 12: The simulation results of the MAC testbench file.

Overall our testbench code was mostly drawn from that provided in the lab 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 an output after all data had been processed and the ready bit had flipped (as can be seen in image 12).

Batching

The idea of batching is to split up the given files of 1000 inputs into smaller pieces and to feed them to the MAC in increments. This has the potential to reduce the number of registers needed to store the inputs and outputs if the batch contains numbers that can be represented with less bits than the number that requires the most. Batching can also serve as another way of executing our testbench as we now have the ability to compare the output of the batched inputs with that of the single larger file to see if they are the same.

Ultimately due to time and technological constraints (in part due to the quarantine) we were unable to fully implement batching for our MAC. As such we were unable to produce any successful tests.

Conclusion

The lab gave us a more realistic scenario to try and find a solution compared to lab 1. We needed to create a script for processing our data while only being provided with a brief description of the desired functionality. We were overall satisfied with the performance of our script as it managed to achieve the desired functionality.

Our VHDL code performed fairly satisfactorily as well. Having followed the testbench example provided in the lab assignment document, our MAC was able to take in the fixed point number files produced by our script and give outputs as can be seen in image 12.

As mentioned in the previous section on batching, we were unable to complete that portion of the lab. However, the idea of how to properly set up a testbench file in order to verify the code was understood.

Overall for the portions of the lab which we were able to complete we were satisfied with our results, from the script performing as desired, to the base of our MAC and testbench file as well.