Department of Electrical and Computer Engineering

Second Semester , 2020/2021

ENCS 3310

<u>Course Final Project Report</u>

****

Student's Name: Mahmoud Samara

Student's No.: 1191602

Date: 31 -5-2021

Section: 1

Instructor: Dr.Abdelatif AbuIssa

****

TABLE OF CONTENTS

I:Introduction:

In this project, we will implement a generic multiplier structurally using two different stages (structures) and then to write a complete code for verification the two stages. So, before talking about my project I will take about the binary multiplier in general. A binary multiplier is an electronic circuit used in digital electronics, such as a computer, to multiply two binary numbers. It is built using binary adders. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

The binary multiplier system has two inputs they are **X** that it will be the (multiplier) with **J** bits (the sign I will use in my code). For the second input it will be **Y** that it will be the (multiplicand) with **K** bits (the sign I will use in my code), and one output will be result that is (J+K-1) bits. After that, both stages will be constructed from little entities which also will be constructed from a minimum entities. We will use these entities to make a complete stage as what we must do.

In my project I will make several basic gates and entities for example in stage 1, I will design a full code for the n bit adder and I use the basic gates to design (and , or , xor) to design half and full adder as It will be shown in my code, then I will make it as a reference for me to use it in the first stage that it depends on the adder and AND gates.

The two stages that we will build in my system: first, using binary bit multiplier and AND gates to make the multiplier which we had learned in digital course, the second stage is using shift register idea that depend on the number we multiply if it is 0 or 1 which we had learned in orga course.

There will be a test bench for each stage that has two registers: Test generator which sends the inputs to our system and send the outputs to the second register and it has a clock input, the second register is the Result analyzer which receives the behavioral output from the test generator and receives the output of the system, it make sure that the two outputs are correct. Finally, We will use (Aldec Active-HDL Student Edition) to simulate my project and test all results and outputs in both stages.

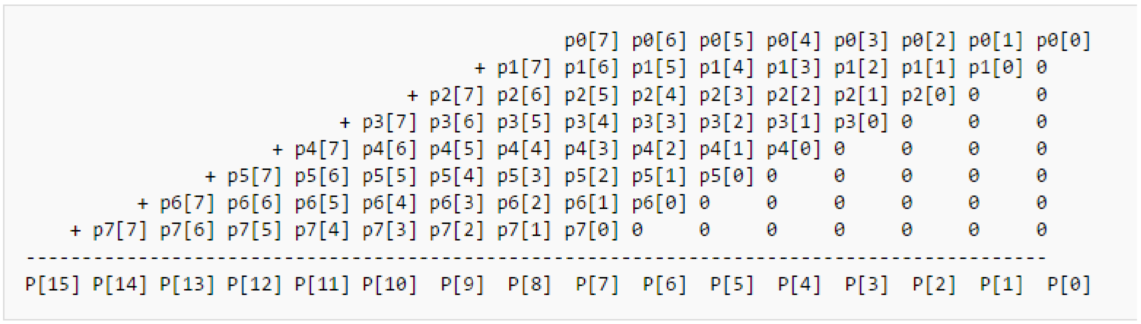
II: Theoretical overview:

In digital systems, a binary multiplier is a combinational logic circuit that performs the multiplication of two binary numbers. These are commonly used to perform various algorithms.

In our project we have two main theories depending on the stages:

1. **Stage (1) : Parallel Binary Multiplier Circuit:**

   Binary encoding is performed by a binary computer. Each long number in binary encoding is multiplied by one digit (either 0 or 1). As a result, multiplying two binary numbers boils down to calculating partial products (which are 0 or the first number), shifting them left, and multiplying them again. For example, suppose we want to multiply a [7:0] and b [7:0] like the following figure.



```
                                                    p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
                                              + p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0] 0
                                        + p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0] 0     0
                                  + p3[7] p3[6] p3[5] p3[4] p3[3] p3[2] p3[1] p3[0] 0     0     0
                            + p4[7] p4[6] p4[5] p4[4] p4[3] p4[2] p4[1] p4[0] 0     0     0     0
                      + p5[7] p5[6] p5[5] p5[4] p5[3] p5[2] p5[1] p5[0] 0     0     0     0     0
                + p6[7] p6[6] p6[5] p6[4] p6[3] p6[2] p6[1] p6[0] 0     0     0     0     0     0
          + p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0] 0     0     0     0     0     0     0
-------------------------------------------------------------------------------------------
P[15] P[14] P[13] P[12] P[11] P[10]  P[9]  P[8]  P[7]  P[6]  P[5]  P[4]  P[3]  P[2]  P[1]  P[0]
```

Figure 1 : Parallel Binary Multiplier example

2. **Stage (2) : Binary Multiplier Using Shift Method:**

   In this stage we are depending on the value of the multiplier and the multiplicand and specially on the multiplicand as the following : If the B0 = 1, the number in the multiplicand (B) is added with the least significant bits of the A register and registers are shifted to the right one bit. If bit B0 equals 0, the combined registers are pushed to the right by one bit without being added. For n bit numbers, this method is performed n times. For example: 4 x 4 multiplier

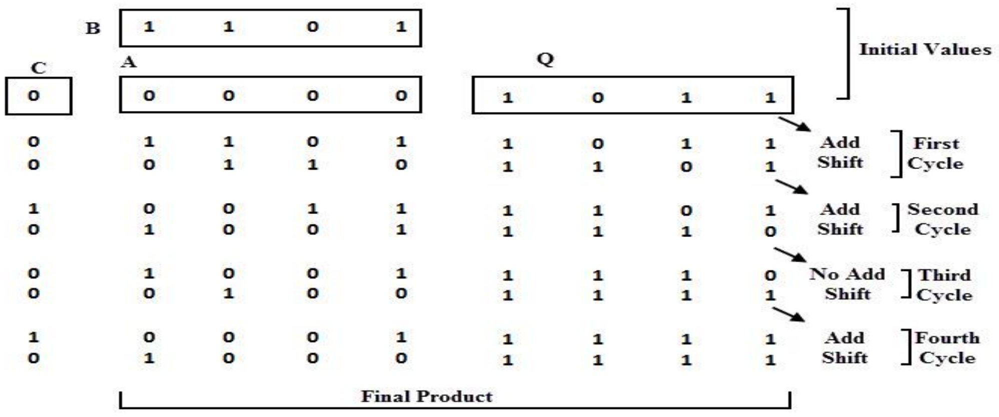
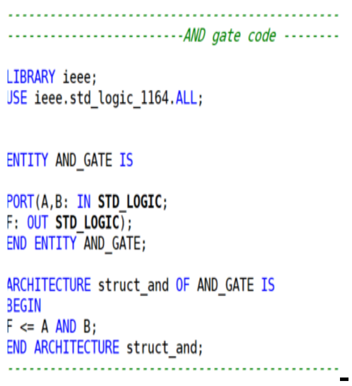


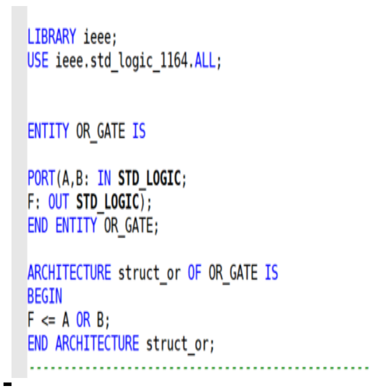
Figure 2 : Binary Multiplier Using Shift Method example

III: Design philosophy:

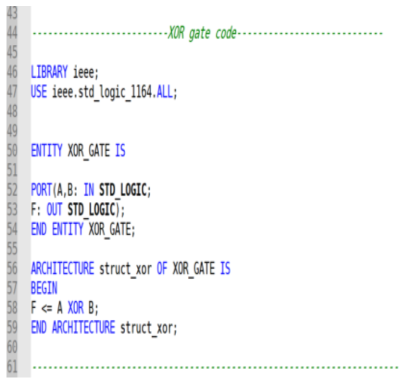
Before starting designing my code for both stages, first I need to make the basic gates that I need for my project. For that, I will design the full code for (and , or , xor) gates and they are as shown in the following figures:



```
--------------------------------------------------
------------------------AND gate code --------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


ENTITY AND_GATE IS

PORT(A,B: IN STD_LOGIC;
F: OUT STD_LOGIC);
END ENTITY AND_GATE;

ARCHITECTURE struct_and OF AND_GATE IS
BEGIN
F <= A AND B;
END ARCHITECTURE struct_and;
--------------------------------------------------
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


ENTITY OR_GATE IS

PORT(A,B: IN STD_LOGIC;
F: OUT STD_LOGIC);
END ENTITY OR_GATE;

ARCHITECTURE struct_or OF OR_GATE IS
BEGIN
F <= A OR B;
END ARCHITECTURE struct_or;
--------------------------------------------------
```

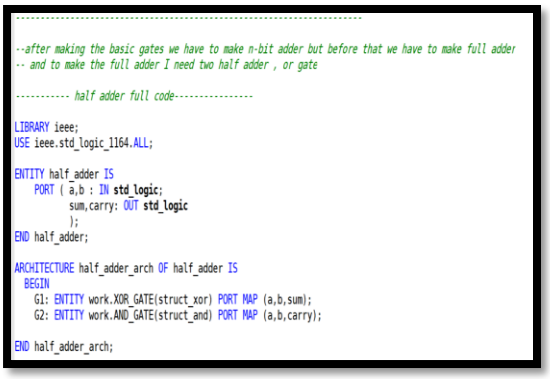


```
-----------------------XOR gate code-----------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


ENTITY XOR_GATE IS

PORT(A,B: IN STD_LOGIC;
F: OUT STD_LOGIC);
END ENTITY XOR_GATE;

ARCHITECTURE struct_xor OF XOR_GATE IS
BEGIN
F <= A XOR B;
END ARCHITECTURE struct_xor;

--------------------------------------------------------------
```

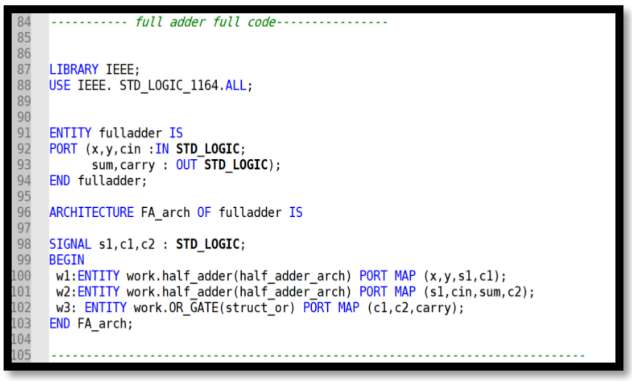
Stage (1):

Half and full adder:

First, I designed a full code for the half adder to use it in the full adder. Then, I made the full adder code using two half adders. The figures below shows the code for (half adder , full adder).



```
------------------------------------------------------------------

--after making the basic gates we have to make n-bit adder but before that we have to make full adder
-- and to make the full adder I need two half adder , or gate

---------- half adder full code---------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY half_adder IS
   PORT ( a,b : IN std_logic;
          sum,carry: OUT std_logic
          );
END half_adder;

ARCHITECTURE half_adder_arch OF half_adder IS
  BEGIN
   G1: ENTITY work.XOR_GATE(struct_xor) PORT MAP (a,b,sum);
   G2: ENTITY work.AND_GATE(struct_and) PORT MAP (a,b,carry);

END half_adder_arch;
```

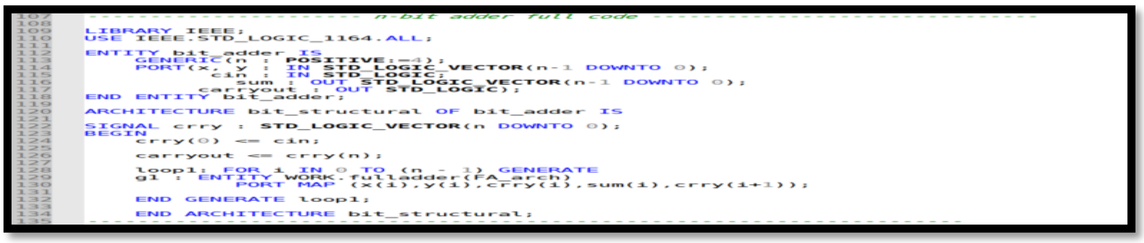


```
---------- full adder full code---------------


LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;


ENTITY fulladder IS
PORT (x,y,cin :IN STD_LOGIC;
      sum,carry : OUT STD_LOGIC);
END fulladder;

ARCHITECTURE FA_arch OF fulladder IS

SIGNAL s1,c1,c2 : STD_LOGIC;
BEGIN
 w1:ENTITY work.half_adder(half_adder_arch) PORT MAP (x,y,s1,c1);
 w2:ENTITY work.half_adder(half_adder_arch) PORT MAP (s1,cin,sum,c2);
 w3: ENTITY work.OR_GATE(struct_or) PORT MAP (c1,c2,carry);
END FA_arch;

-----------------------------------------------------------------------
```

Figure 3 : half adder + full adder

n-bit adder :

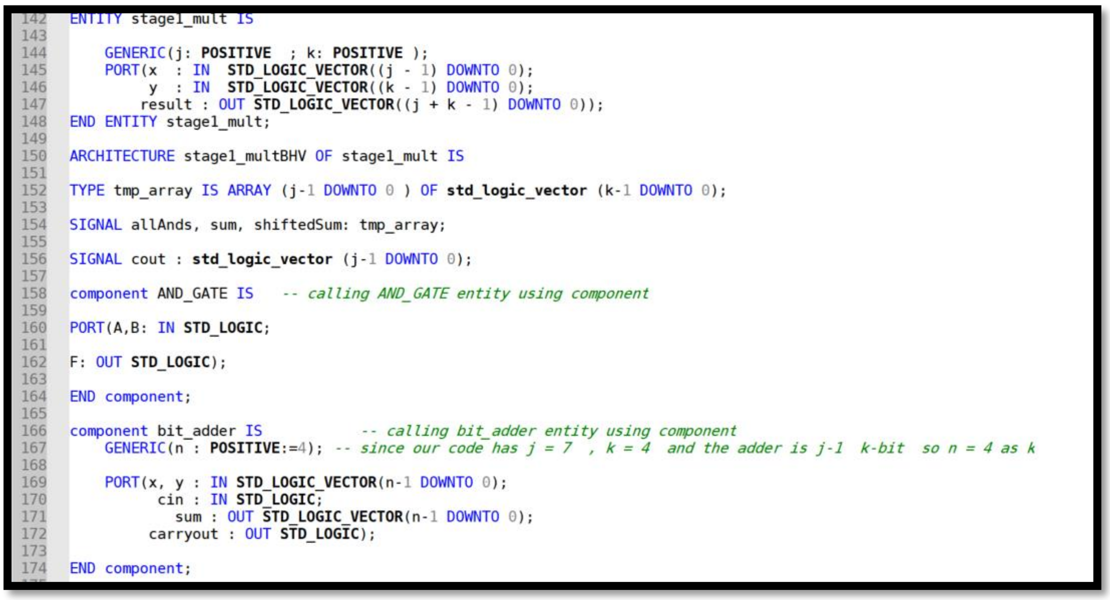
As we have seen in the stage 1 design we can notice that we need n bit adder. The n bit adder will be as following (number of adders equal M-1 , and the adder will be N-bit adder). The following figure shows how I used the full adder library to implement the code for n bit adder.



```
------------------------ n-bit adder full code ----------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bit_adder IS
     GENERIC(n  :  POSITIVE:=4);
     PORT(x,  y  :  IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
           cin  :  IN STD_LOGIC;
              sum  : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
         carryout  :  OUT STD_LOGIC);
END ENTITY bit_adder;

ARCHITECTURE bit_structural OF bit_adder IS

SIGNAL crry : STD_LOGIC_VECTOR(n DOWNTO 0);
BEGIN
     crry(0) <= cin;

     carryout <= crry(n);

     loop1: FOR i IN 0 TO (n - 1) GENERATE
     g1  :  ENTITY WORK.fulladder(FA_arch)
              PORT MAP (x(i),y(i),crry(i),sum(i),crry(i+1));

     END GENERATE loop1;

     END ARCHITECTURE bit_structural;
-------------------------------------------------------------------------
```

STAGE 1 multiplier full code:

As shown in figure (4) that includes the code for stage 1. You can see initially that I used the component keyword to call the n bit adder entity , AND entity and I gave number of bits of the adder is 4, because it is a K. Also both inputs and outputs were generic. I also used the loop strategy because it is the easiest way when everything is in general way.

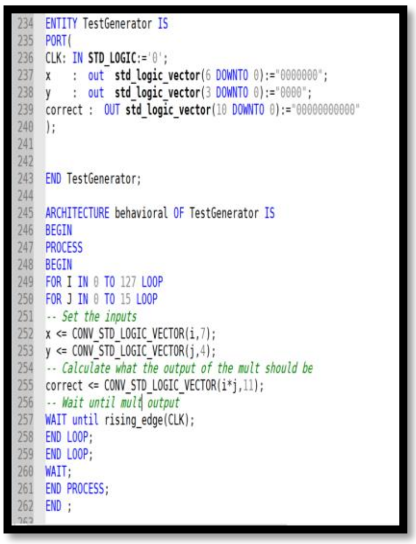


```
ENTITY stage1_mult IS

    GENERIC(j: POSITIVE  ; k: POSITIVE );
    PORT(x  : IN  STD_LOGIC_VECTOR((j - 1) DOWNTO 0);
         y  : IN  STD_LOGIC_VECTOR((k - 1) DOWNTO 0);
        result : OUT STD_LOGIC_VECTOR((j + k - 1) DOWNTO 0));
END ENTITY stage1_mult;

ARCHITECTURE stage1_multBHV OF stage1_mult IS

TYPE tmp_array IS ARRAY (j-1 DOWNTO 0 ) OF std_logic_vector (k-1 DOWNTO 0);

SIGNAL allAnds, sum, shiftedSum: tmp_array;

SIGNAL cout : std_logic_vector (j-1 DOWNTO 0);

component AND_GATE IS   -- calling AND_GATE entity using component

PORT(A,B: IN STD_LOGIC;

F: OUT STD_LOGIC);

END component;

component bit_adder IS           -- calling bit_adder entity using component
    GENERIC(n : POSITIVE:=4); -- since our code has j = 7  , k = 4  and the adder is j-1  k-bit  so n = 4 as k

    PORT(x, y : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
          cin : IN STD_LOGIC;
            sum : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
         carryout : OUT STD_LOGIC);

END component;
```

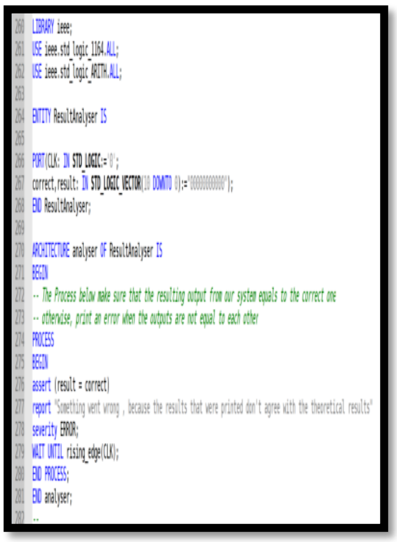
Figure 4 : Stage 1 multiplier

Result checker and test bench for stage 1:

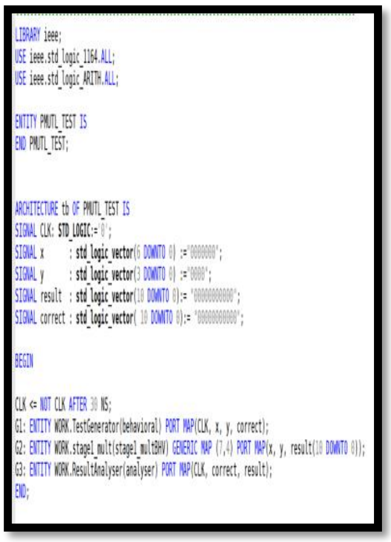
Now for the verification we need test generator , result checker and testbench. In my code the test generator will be built as a loop like what we took in the lectures.The result checker I used in it assert to print the error message as we took in the lectures. Therefore, The Process assert make sure that the resulting output from our system equals to the correct one otherwise, print an error when the outputs are not equal to each other.



```
ENTITY TestGenerator IS
PORT(
CLK: IN STD_LOGIC:='0';
x    :  out  std_logic_vector(6 DOWNTO 0):="0000000";
y    :  out  std_logic_vector(3 DOWNTO 0):="0000";
correct :  OUT std_logic_vector(10 DOWNTO 0):="00000000000"
);


END TestGenerator;

ARCHITECTURE behavioral OF TestGenerator IS
BEGIN
PROCESS
BEGIN
FOR I IN 0 TO 127 LOOP
FOR J IN 0 TO 15 LOOP
-- Set the inputs
x <= CONV_STD_LOGIC_VECTOR(i,7);
y <= CONV_STD_LOGIC_VECTOR(j,4);
-- Calculate what the output of the mult should be
correct <= CONV_STD_LOGIC_VECTOR(i*j,11);
-- Wait until mult output
WAIT until rising_edge(CLK);
END LOOP;
END LOOP;
WAIT;
END PROCESS;
END ;
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_ARITH.ALL;

ENTITY ResultAnalyser IS

PORT(CLK: IN STD_LOGIC:='0';
correct,result: IN STD_LOGIC_VECTOR(10 DOWNTO 0):="00000000000");
END ResultAnalyser;

ARCHITECTURE analyser OF ResultAnalyser IS
BEGIN
-- The Process below make sure that the resulting output from our system equals to the correct one
-- otherwise, print an error when the outputs are not equal to each other
PROCESS
BEGIN
assert (result = correct)
report "Something went wrong , because the results that were printed don't agree with the theoretical results"
severity ERROR;
WAIT UNTIL rising_edge(CLK);
END PROCESS;
END analyser;
--
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_ARITH.ALL;

ENTITY PMUTL_TEST IS
END PMUTL_TEST;


ARCHITECTURE tb OF PMUTL_TEST IS
SIGNAL CLK: STD_LOGIC:='0';
SIGNAL x      : std_logic_vector(6 DOWNTO 0) :="0000000";
SIGNAL y      : std_logic_vector(3 DOWNTO 0) :="0000";
SIGNAL result  : std_logic_vector(10 DOWNTO 0):= "00000000000";
SIGNAL correct : std_logic_vector( 10 DOWNTO 0):= "00000000000";

BEGIN

CLK <= NOT CLK AFTER 30 NS;
G1: ENTITY WORK.TestGenerator(behavioral) PORT MAP(CLK, x, y, correct);
G2: ENTITY WORK.stage1_mult(stage1_multBHV) GENERIC MAP (7,4) PORT MAP(x, y, result(10 DOWNTO 0));
G3: ENTITY WORK.ResultAnalyser(analyser) PORT MAP(CLK, correct, result);
END;
```

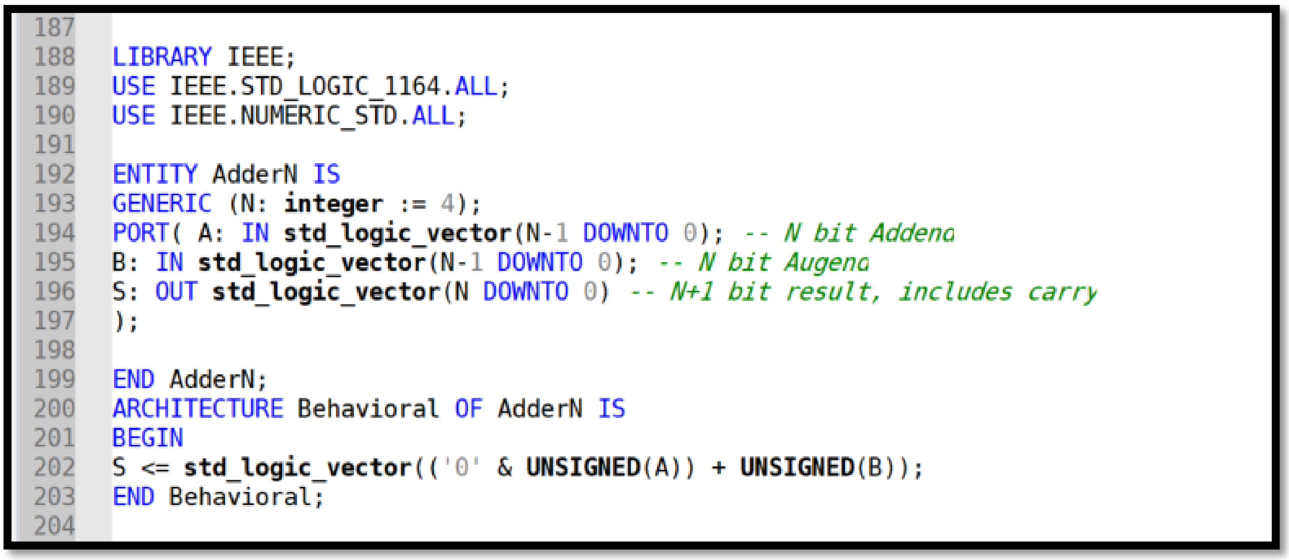
Figure 5 : Stage 1 Test generator, result analyser and test bench

Stage (2):

In stage two as it is shift method and depending on the explaining figure in the project file I designed three circuits to use them in the final code and they are explained as the following:

A) Adder:

In the following figure (7) we can see the full code for the adder that is n bit adder, the output is n+ 1 bit, and I designed it using the behavioral way.

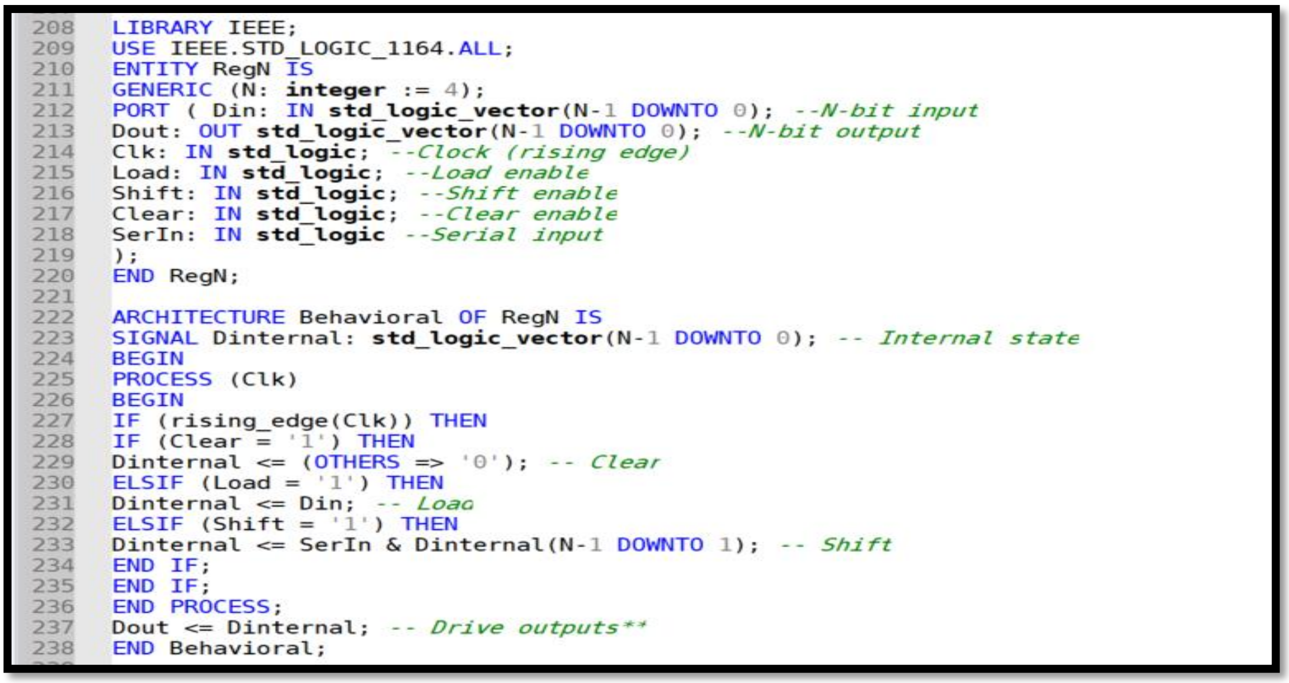


```vhdl

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY AdderN IS
GENERIC (N: integer := 4);
PORT( A: IN std_logic_vector(N-1 DOWNTO 0); -- N bit Addend
B: IN std_logic_vector(N-1 DOWNTO 0); -- N bit Augend
S: OUT std_logic_vector(N DOWNTO 0) -- N+1 bit result, includes carry
);

END AdderN;
ARCHITECTURE Behavioral OF AdderN IS
BEGIN
S <= std_logic_vector(('0' & UNSIGNED(A)) + UNSIGNED(B));
END Behavioral;

```

Figure 7 : Adder

B) Register:

The following figure shows the full code for the shift register and also it used to clear enable. The register has 4 bit input as the value of (N).

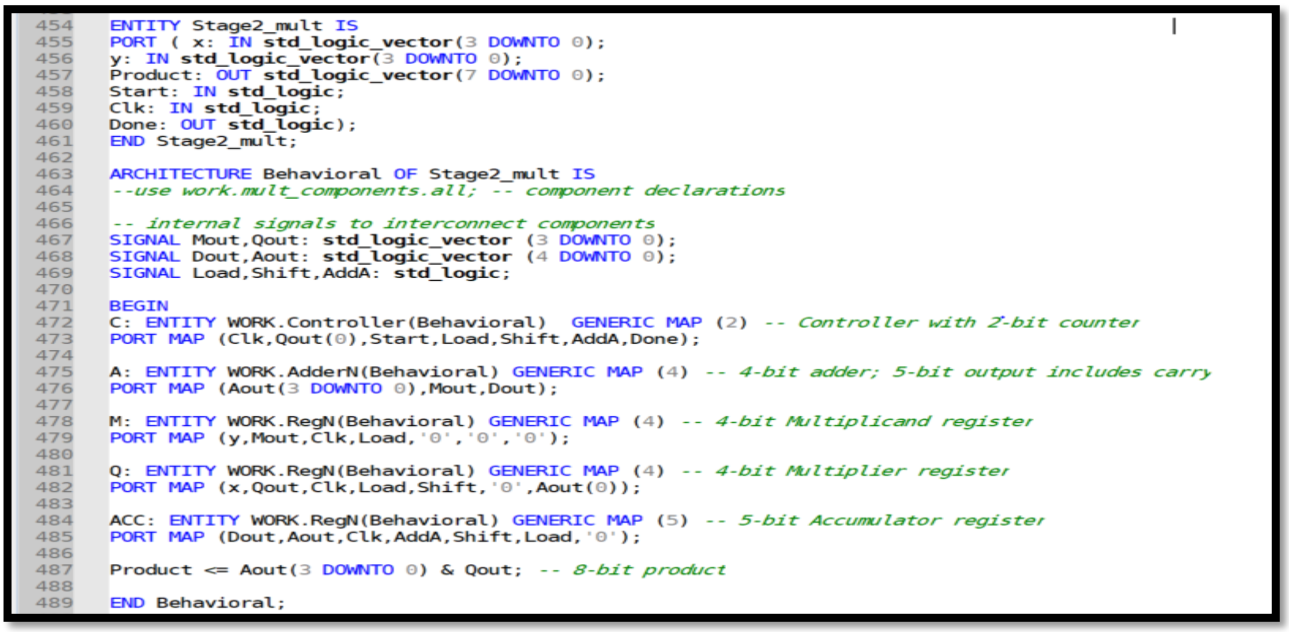


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY RegN IS
GENERIC (N: integer := 4);
PORT ( Din: IN std_logic_vector(N-1 DOWNTO 0); --N-bit input
Dout: OUT std_logic_vector(N-1 DOWNTO 0); --N-bit output
Clk: IN std_logic; --Clock (rising edge)
Load: IN std_logic; --Load enable
Shift: IN std_logic; --Shift enable
Clear: IN std_logic; --Clear enable
SerIn: IN std_logic --Serial input
);
END RegN;

ARCHITECTURE Behavioral OF RegN IS
SIGNAL Dinternal: std_logic_vector(N-1 DOWNTO 0); -- Internal state
BEGIN
PROCESS (Clk)
BEGIN
IF (rising_edge(Clk)) THEN
IF (Clear = '1') THEN
Dinternal <= (OTHERS => '0'); -- Clear
ELSIF (Load = '1') THEN
Dinternal <= Din; -- Load
ELSIF (Shift = '1') THEN
Dinternal <= SerIn & Dinternal(N-1 DOWNTO 1); -- Shift
END IF;
END IF;
END PROCESS;
Dout <= Dinternal; -- Drive outputs**
END Behavioral;
```

Figure 8 : Register

STAGE 2 multiplier full code:

As shown in figure (9) that includes the full code for stage 2. You can see initially that I used the component keyword to call the 3 main circuits that I designed before bit entity.

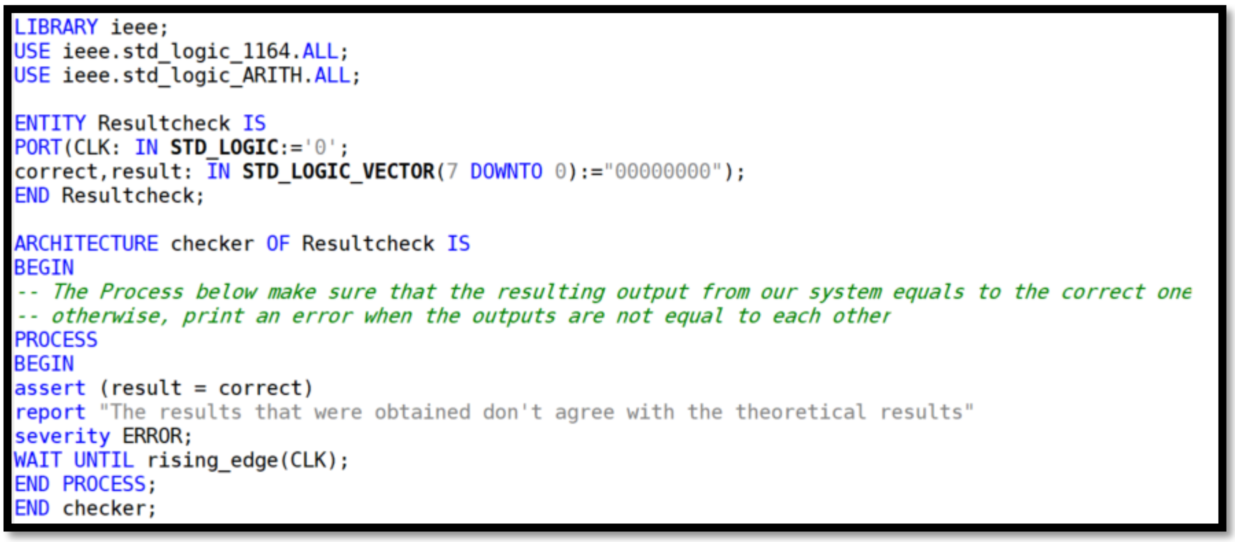


```
ENTITY Stage2_mult IS
PORT ( x: IN std_logic_vector(3 DOWNTO 0);
y: IN std_logic_vector(3 DOWNTO 0);
Product: OUT std_logic_vector(7 DOWNTO 0);
Start: IN std_logic;
Clk: IN std_logic;
Done: OUT std_logic);
END Stage2_mult;

ARCHITECTURE Behavioral OF Stage2_mult IS
--use work.mult_components.all; -- component declarations

-- internal signals to interconnect components
SIGNAL Mout,Qout: std_logic_vector (3 DOWNTO 0);
SIGNAL Dout,Aout: std_logic_vector (4 DOWNTO 0);
SIGNAL Load,Shift,AddA: std_logic;

BEGIN
C: ENTITY WORK.Controller(Behavioral)  GENERIC MAP (2) -- Controller with 2-bit counter
PORT MAP (Clk,Qout(0),Start,Load,Shift,AddA,Done);

A: ENTITY WORK.AdderN(Behavioral) GENERIC MAP (4) -- 4-bit adder; 5-bit output includes carry
PORT MAP (Aout(3 DOWNTO 0),Mout,Dout);

M: ENTITY WORK.RegN(Behavioral) GENERIC MAP (4) -- 4-bit Multiplicand register
PORT MAP (y,Mout,Clk,Load,'0','0','0');

Q: ENTITY WORK.RegN(Behavioral) GENERIC MAP (4) -- 4-bit Multiplier register
PORT MAP (x,Qout,Clk,Load,Shift,'0',Aout(0));

ACC: ENTITY WORK.RegN(Behavioral) GENERIC MAP (5) -- 5-bit Accumulator register
PORT MAP (Dout,Aout,Clk,AddA,Shift,Load,'0');

Product <= Aout(3 DOWNTO 0) & Qout; -- 8-bit product

END Behavioral;
```

Figure 9 : STAGE 2 multiplier

Result checker and test bench for stage 2:

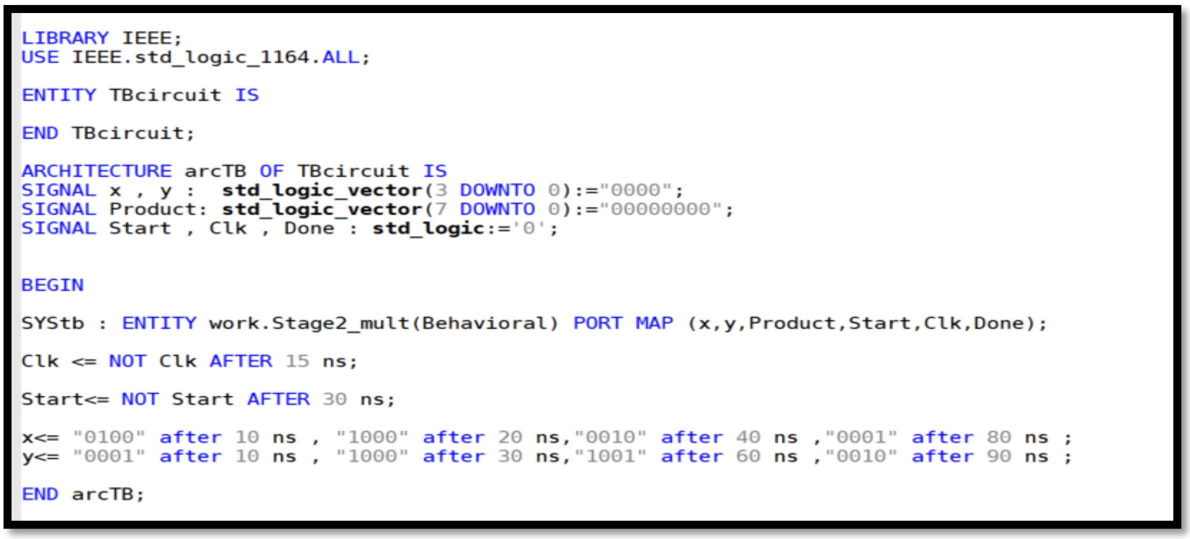
The Result checker below make sure that the resulting output from our system equals to the correct one otherwise, print an error when the outputs are not equal to each other using assert component and typing the error message.



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_ARITH.ALL;

ENTITY Resultcheck IS
PORT(CLK: IN STD_LOGIC:='0';
correct,result: IN STD_LOGIC_VECTOR(7 DOWNTO 0):="00000000");
END Resultcheck;

ARCHITECTURE checker OF Resultcheck IS
BEGIN
-- The Process below make sure that the resulting output from our system equals to the correct one
-- otherwise, print an error when the outputs are not equal to each other
PROCESS
BEGIN
assert (result = correct)
report "The results that were obtained don't agree with the theoretical results"
severity ERROR;
WAIT UNTIL rising_edge(CLK);
END PROCESS;
END checker;
```

Figure 10 : checker

Finally the following figure shows the testbench for stage (2):



```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY TBcircuit IS

END TBcircuit;

ARCHITECTURE arcTB OF TBcircuit IS
SIGNAL x , y :  std_logic_vector(3 DOWNTO 0):="0000";
SIGNAL Product: std_logic_vector(7 DOWNTO 0):="00000000";
SIGNAL Start , Clk , Done : std_logic:='0';


BEGIN

SYStb : ENTITY work.Stage2_mult(Behavioral) PORT MAP (x,y,Product,Start,Clk,Done);

Clk <= NOT Clk AFTER 15 ns;

Start<= NOT Start AFTER 30 ns;

x<= "0100" after 10 ns , "1000" after 20 ns,"0010" after 40 ns ,"0001" after 80 ns ;
y<= "0001" after 10 ns , "1000" after 30 ns,"1001" after 60 ns ,"0010" after 90 ns ;

END arcTB;
```

Figure 11 : TEST BENCH

IV:Simulation and results:

Stage(1):

These are some screenshots for the simulation shows the results of my final code for stage 1. We can conclude in stage 1 that for 7 * 4 mult we need 4-bit adder. In addition, when making the test and analyser we have to use clk as a register to keep changing the value of the inputs.

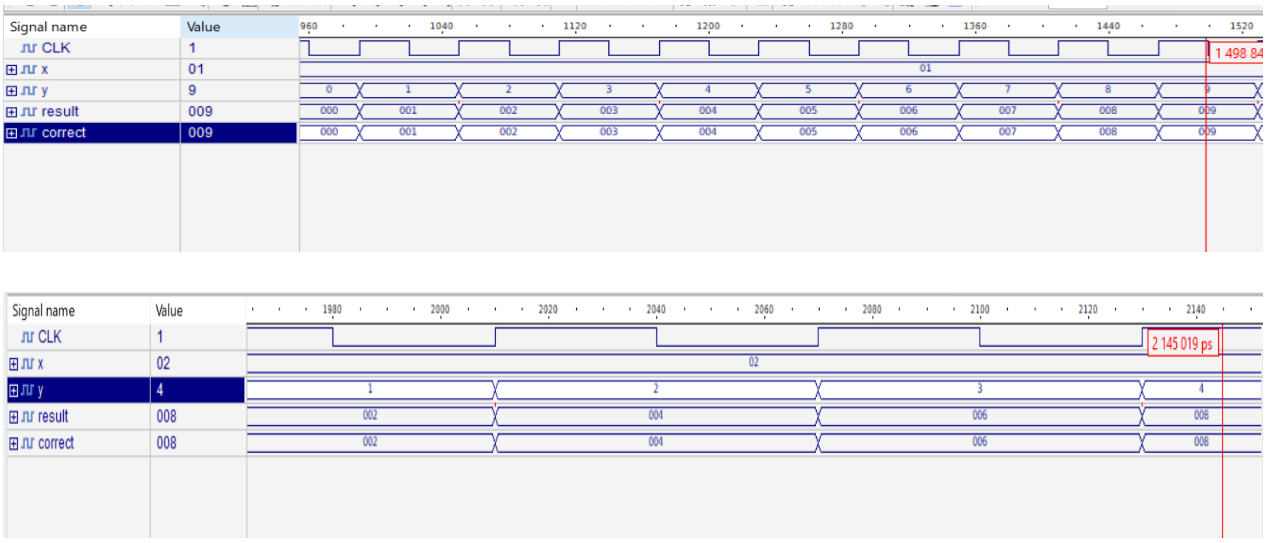


Figure 12 : STAGE 1 simulation

Stage(2):

These are some screenshots for the simulation shows the results of my final code for stage 2. We can conclude in stage 2 that the shift or add depend on what we multiply 0 or 1.

The figures are:

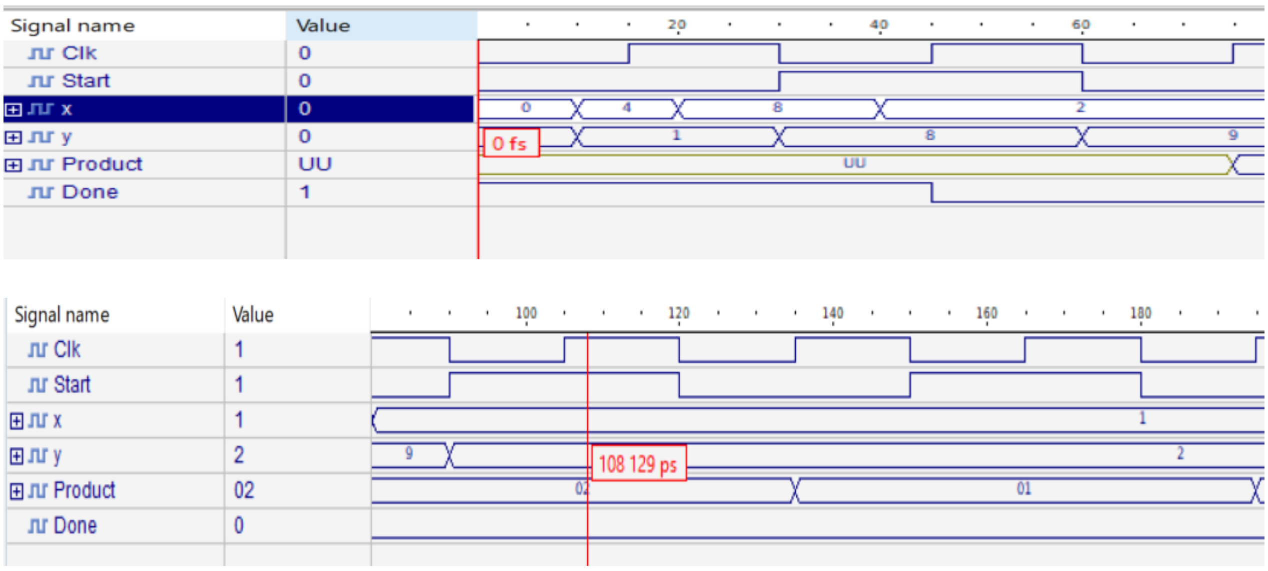


Figure 13 : STAGE 2 simulation

V:Conclusion and Future works:

In conclusion, I am very proud of my work and the results I got even I didn't make stage two generic. On the other hand, the results I got agree with the theoretical especially in case one. Moreover, this project and its stages were very helpful for me because I learned many things, such as: I became now more capable with the VHDL program. In addition, the generic was a big problem for me and I solve it and the same thing with assert. Moreover, I became more familiar with VHDL and how to write commands like how to print an error, testing the systems and building entities in behavioral and structural logics.

I faced some problems with my project which sometimes made me give up, but I continued on the project to the end of these problems are: The general case in both stages were a big problem for me , because I have to make loops so that the lecture notes and youtube videos were my go-to for solving this problem. Furthermore, in the resulting parser when using assertion, I always had an error in the code, then I googled the issue and found that I had to add a library (std_logic_ARITH). In the end, everything were perfect.

I successfully designed a parallel multiplier and another way of multiplier using the shift method. I learnt a lot about types of adders and how to build them from full adder, also I noticed the difference between both stages even they give the same output. In my opinion, I think stage 1 is faster and better than stage 2 maybe because I saw it easier than stage2.

The project and its stages will help me in the future works, because as we know to computer works only with (one or zero) simply just binary numbers. So now I became more familiar with the computer brain and how the arithmetic operations done in it. Finally, this project and previously the assembly and Active HDL now will guide me to make a full calculator using VHDL code and that's what I am planning to. Moreover, during my work I noticed a problem in the parallel multiplier that before sending new data the cycle must end, this leads to very slow throughput, so I think in the future we can use a pipeline as a solution for the problem, which consists of using n-bit registers between an adder and the next one. In this way, new data can be sent while other data are still being operating.

THE END

THANK YOU FOR READING MY REPORT

VI: Appendix:

**** THE FULL CODE WERE ATTACHED IN A SINGLE .vhd FILE ****