Computer Systems Engineering Department

ADVANCED DIGITAL SYSTEMS DESIGN

ENCS533

Report for the project

Instructor: Dr. Abdallatif Abuissa

Student name and number:

Ahmad Nidal Hamad 1180060

Date: 17-12-2020

Abstract

Adders are one of the most recently used combinational circuits in most of the ICs.

In this project we are going to design a two-bit BCD adder which takes two numbers represented in the BCD format and calculates the sum of them in the BCD format also.

First of all, I will be giving a brief introduction to the circuit used in the project.

Secondly, I will explain the design procedure starting with the main ideas and ending with a complete code.

Thirdly I am going to discuss the simulation result including faulty case.

Finally, I will give my conclusion about the whole project.

Contents

Figures

Introduction

In this introduction I will introduce briefly all the circuits used in this project.

Adders

Half Adder:

Half adder is a combinational circuit that used to perform addition of two bits by giving two values the sum of the two bits and the carry out. .[1]

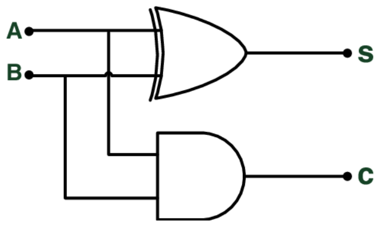


Figure 1:Half adder circuit

Full Adder:

Full Adder is the adder which adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM. .[1]

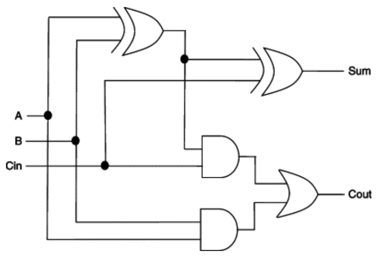


Figure 2:Full adder circuit

4-Bits full adder:

A 4-Bits binary adder is usually built using four full adders by connecting them together in a way that every bit of each number goes to one adder and the input carry is taken from the output carry from the previous adder (the input carry of the first adder is usually set to be zero)

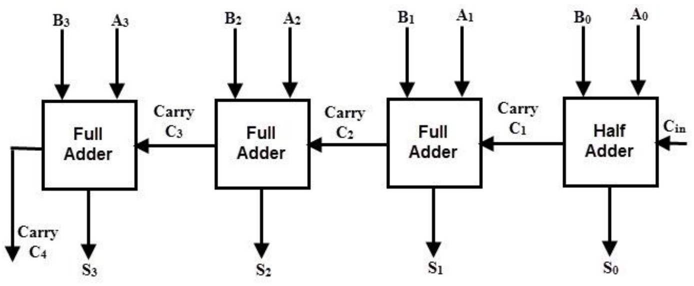


Figure 3:4-Bits full adder circuit

Look ahead adder

In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So, there will be a considerable time delay which is carry propagation delay.

A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic. Let us discuss the design in detail.

The carry output Boolean function of each stage in a 4 stage carry look-ahead adder can be expressed as

$$C_1 = G_0 + P_0C_{in}$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_{in}$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{in}$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{in}$$

From the above Boolean equations, we can observe that C4 does not have to wait for C3 and C2to propagate but actually is propagated at the same time C3 as C2 and. Since the Boolean expression for each carry output is the sum of products so these can be implemented with one level of AND gates followed by an OR gate.

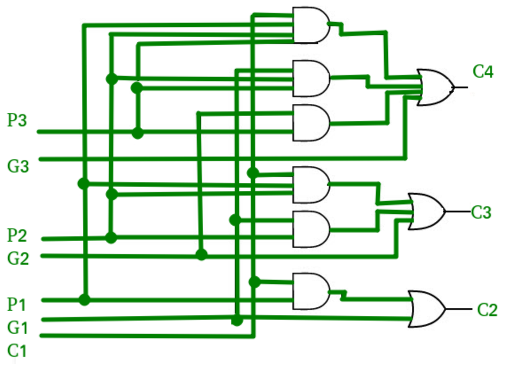


Figure 4:Carry generator circuit

1-Bit BCD adder

Since the BCD number is a normal binary number, to add two BCD number we can use the 4-Bits binary adder. But the only difference that when adding two BCD digits the answer can be in correct so we need to modify it by adding 6 using the 4-Bits binary adder again.

The output of adding two BCD digits is incorrect if the answer exceeds 9 (it is known that the maximum value of a one-digit BCD is 9), so by looking at the output we can notice that the answer is incorrect if there is a carry or if the fourth bit is set to 1 with either the second or the third bit.

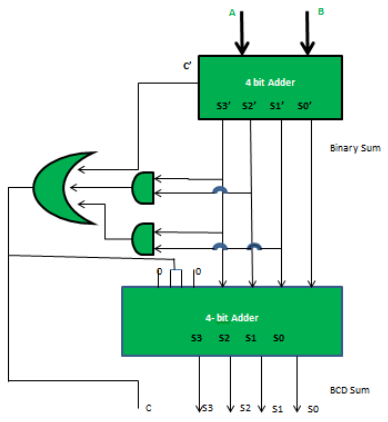


Figure 5:1-Bit BCD adder circuit

2-Bit BCD adder

In the same way a 2-bit BCD adder can be built structurally using two of the 1-bit BCD adders by Appling the first digit of each number to first BCD adder and applying the other digits with the output carry to the second one.

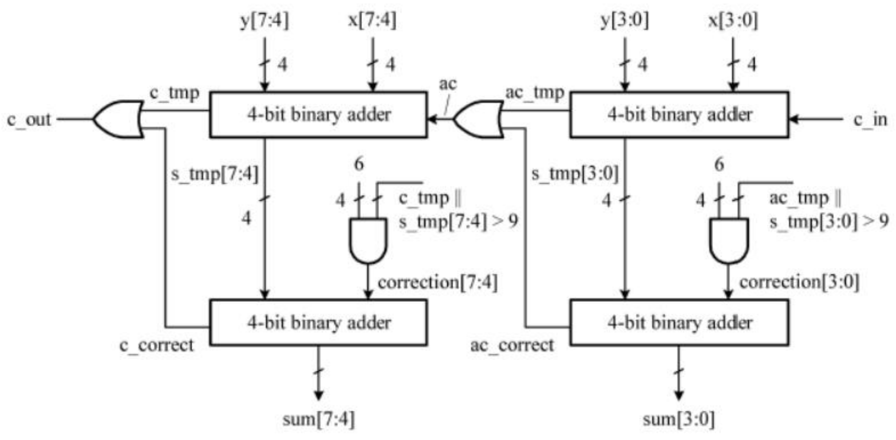


Figure 6:2-Bits BCD adder circuit

Registers

The register is the only sequential circuit used in this project, which means that it needs a clock to operate.

Register in a sequential circuit that is used to store a data from the input whenever at the rising or the falling edge of the clock depending on the type of the register.

Design philosophy

Building the gates

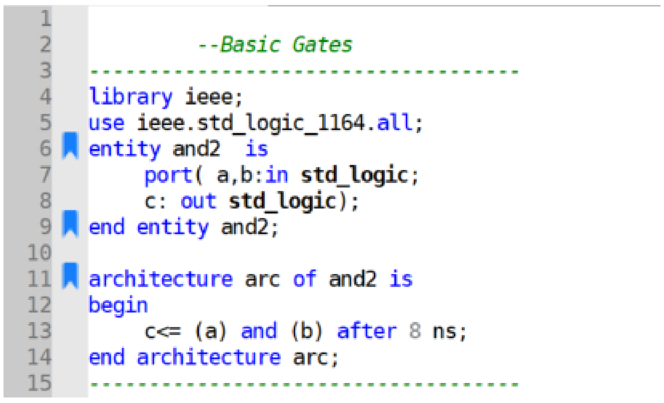

```
        --Basic Gates
-----------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity and2  is
    port( a,b:in std_logic;
    c: out std_logic);
end entity and2;

architecture arc of and2 is
begin
    c<= (a) and (b) after 8 ns;
end architecture arc;
-----------------------------------
```

Figure 7:And gate code

As shown in the above code, the and gate is being built with its suitable delay. Also, all the other gates were built is the same way.

Building the full adder

The full adder was built structurally using the predefined gates as follows:

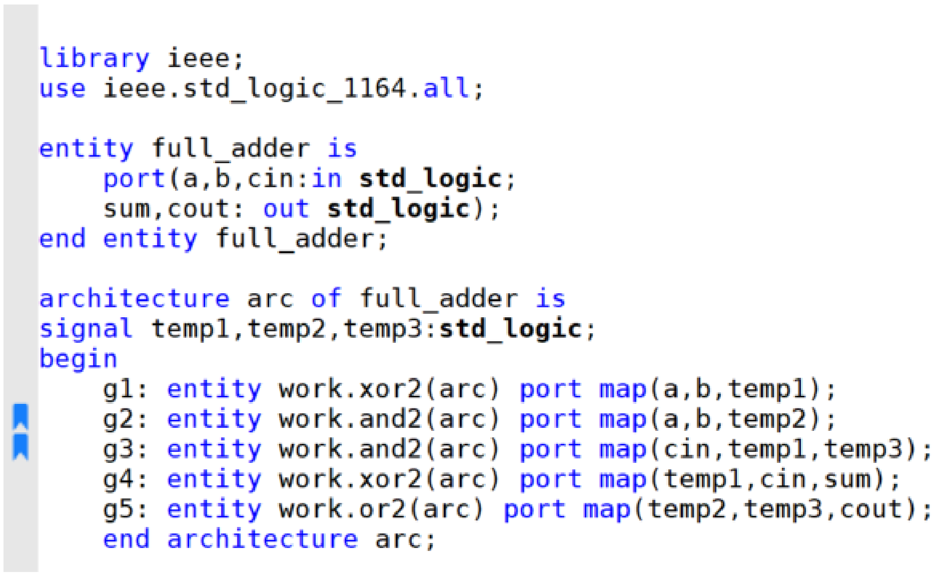
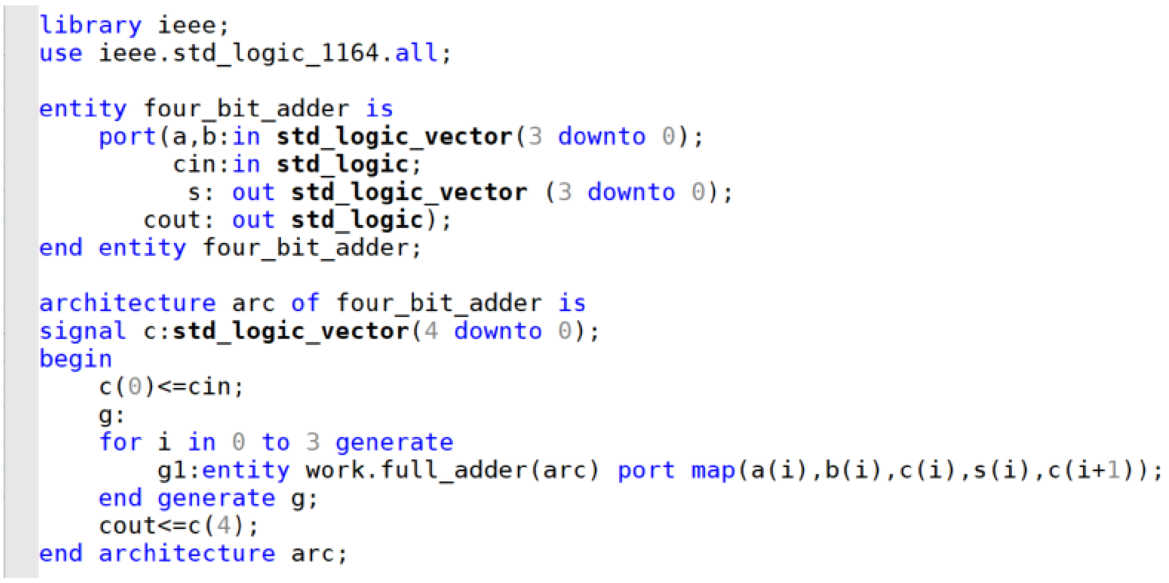

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
    port(a,b,cin:in std_logic;
    sum,cout: out std_logic);
end entity full_adder;

architecture arc of full_adder is
signal temp1,temp2,temp3:std_logic;
begin
    g1: entity work.xor2(arc) port map(a,b,temp1);
    g2: entity work.and2(arc) port map(a,b,temp2);
    g3: entity work.and2(arc) port map(cin,temp1,temp3);
    g4: entity work.xor2(arc) port map(temp1,cin,sum);
    g5: entity work.or2(arc) port map(temp2,temp3,cout);
    end architecture arc;
```

Figure 8:Full adder code

As we can see from the figure the adder is built structurally as discussed earlier in the theoretical part.

Building the 4-Bits ripple adder

The 4-Bits ripple adder was built structurally using four full adders as follows:



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

entity four_bit_adder is
    port(a,b:in std_logic_vector(3 downto 0);
         cin:in std_logic;
          s: out std_logic_vector (3 downto 0);
       cout: out std_logic);
end entity four_bit_adder;

architecture arc of four_bit_adder is
signal c:std_logic_vector(4 downto 0);
begin
    c(0)<=cin;
    g:
    for i in 0 to 3 generate
        g1:entity work.full_adder(arc) port map(a(i),b(i),c(i),s(i),c(i+1));
    end generate g;
    cout<=c(4);
end architecture arc;
```

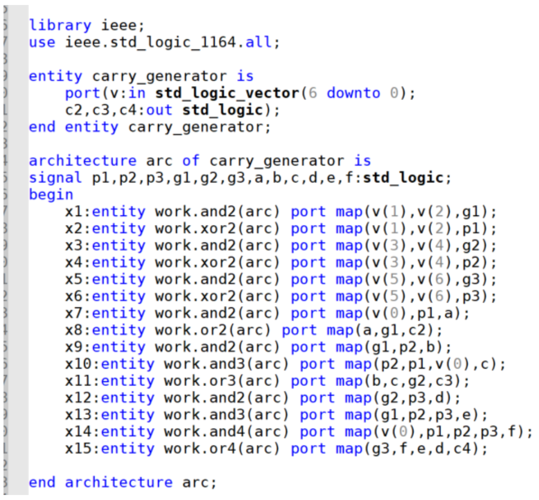
Figure 9:4-Bits ripple adder code

As shown in the code, the four-bit adder is built structurally using four full adders also as shown in the theoretical part. Generate statement is used to simplify the code and reducing the number of lines needed for each 1-bit adder

Building the 4 bits look ahead adder

Building the carry generator

The carry generator was built structurally using the pre-defend half adder and the basic gates.



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

entity carry_generator is
    port(v:in std_logic_vector(6 downto 0);
    c2,c3,c4:out std_logic);
end entity carry_generator;

architecture arc of carry_generator is
signal p1,p2,p3,g1,g2,g3,a,b,c,d,e,f:std_logic;
begin
    x1:entity work.and2(arc) port map(v(1),v(2),g1);
    x2:entity work.xor2(arc) port map(v(1),v(2),p1);
    x3:entity work.and2(arc) port map(v(3),v(4),g2);
    x4:entity work.xor2(arc) port map(v(3),v(4),p2);
    x5:entity work.and2(arc) port map(v(5),v(6),g3);
    x6:entity work.xor2(arc) port map(v(5),v(6),p3);
    x7:entity work.and2(arc) port map(v(0),p1,a);
    x8:entity work.or2(arc) port map(a,g1,c2);
    x9:entity work.and2(arc) port map(g1,p2,b);
    x10:entity work.and3(arc) port map(p2,p1,v(0),c);
    x11:entity work.or3(arc) port map(b,c,g2,c3);
    x12:entity work.and2(arc) port map(g2,p3,d);
    x13:entity work.and3(arc) port map(g1,p2,p3,e);
    x14:entity work.and4(arc) port map(v(0),p1,p2,p3,f);
    x15:entity work.or4(arc) port map(g3,f,e,d,c4);

end architecture arc;
```

Figure 10:Carry generator code

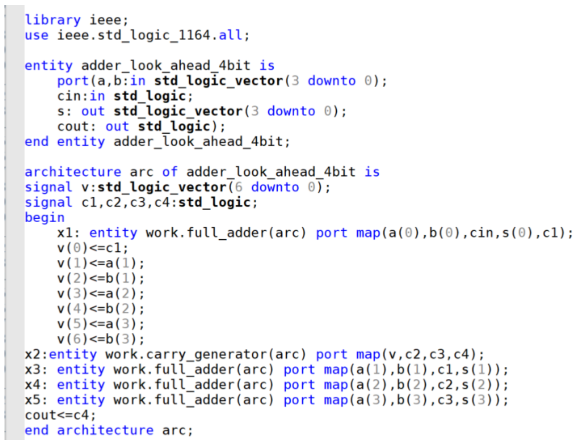
In this step, the carry generator is built structurally using the basic gates.

The vector used in the code will be shown in details when talking about the full system for the look ahead adder.

The variables a b c d and e are used to store the output of each and gate temporarily in order to use it as an input for the or gates later.

Budling the 4-Bits look ahead adder

Finally, the 4 bits look ahead adder was built structurally using the carry generator and the basic gates.



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

entity adder_look_ahead_4bit is
    port(a,b:in std_logic_vector(3 downto 0);
    cin:in std_logic;
    s: out std_logic_vector(3 downto 0);
    cout: out std_logic);
end entity adder_look_ahead_4bit;

architecture arc of adder_look_ahead_4bit is
signal v:std_logic_vector(6 downto 0);
signal c1,c2,c3,c4:std_logic;
begin
    x1: entity work.full_adder(arc) port map(a(0),b(0),cin,s(0),c1);
    v(0)<=c1;
    v(1)<=a(1);
    v(2)<=b(1);
    v(3)<=a(2);
    v(4)<=b(2);
    v(5)<=a(3);
    v(6)<=b(3);
x2:entity work.carry_generator(arc) port map(v,c2,c3,c4);
x3: entity work.full_adder(arc) port map(a(1),b(1),c1,s(1));
x4: entity work.full_adder(arc) port map(a(2),b(2),c2,s(2));
x5: entity work.full_adder(arc) port map(a(3),b(3),c3,s(3));
cout<=c4;
end architecture arc;
```

Figure 11:4-Bits look ahead adder code

Here it is obvious that the 4-Bits adder is built structurally using the pre-implemented carry generator.

Also, for the simplicity, the first half adder is invoked out of the carry generator and then all the bits of each variable with the carry got from the first half adder was copied to a vector to reduce the size of signals needed in the carry generator entity as follows:

V (0) contains the first carry

V (1) contains the second bit of the first number

V (2) contains the second bit of the second number

V (3) contains the third bit of the first number

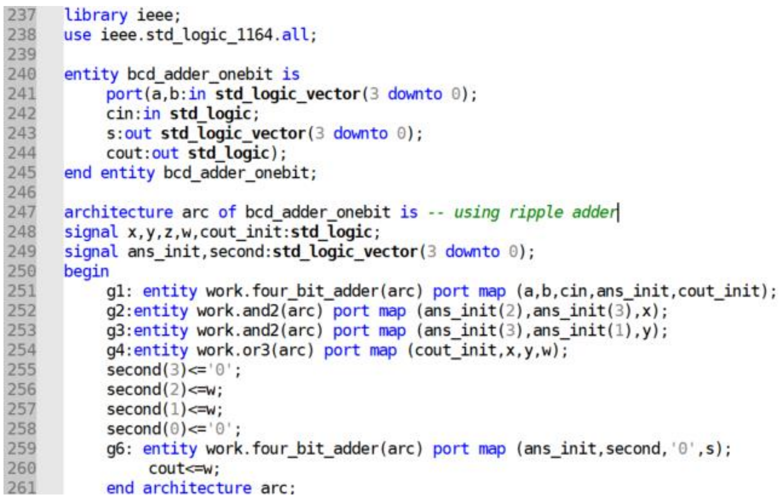
V (4) contains the third bit of the second number

V (5) contains the fourth bit of the first number

V (6) contains the fourth bit of the second number

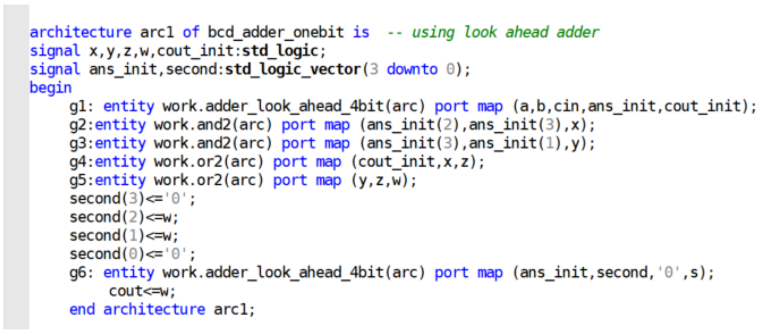
Building the 1-digit BCD adder

The 1-digit BCD adder was built at the first structurally using the four-bit ripple adder and using the four bits look ahead adder in second time as follows:



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

entity bcd_adder_onebit is
    port(a,b:in std_logic_vector(3 downto 0);
    cin:in std_logic;
    s:out std_logic_vector(3 downto 0);
    cout:out std_logic);
end entity bcd_adder_onebit;

architecture arc of bcd_adder_onebit is -- using ripple adder
signal x,y,z,w,cout_init:std_logic;
signal ans_init,second:std_logic_vector(3 downto 0);
begin
    g1: entity work.four_bit_adder(arc) port map (a,b,cin,ans_init,cout_init);
    g2:entity work.and2(arc) port map (ans_init(2),ans_init(3),x);
    g3:entity work.and2(arc) port map (ans_init(3),ans_init(1),y);
    g4:entity work.or3(arc) port map (cout_init,x,y,w);
    second(3)<='0';
    second(2)<=w;
    second(1)<=w;
    second(0)<='0';
    g6: entity work.four_bit_adder(arc) port map (ans_init,second,'0',s);
        cout<=w;
    end architecture arc;
```

Figure 12:1-Digit BCD adder code using ripple adder



```vhdl
architecture arc1 of bcd_adder_onebit is  -- using look ahead adder
signal x,y,z,w,cout_init:std_logic;
signal ans_init,second:std_logic_vector(3 downto 0);
begin
    g1: entity work.adder_look_ahead_4bit(arc) port map (a,b,cin,ans_init,cout_init);
    g2:entity work.and2(arc) port map (ans_init(2),ans_init(3),x);
    g3:entity work.and2(arc) port map (ans_init(3),ans_init(1),y);
    g4:entity work.or2(arc) port map (cout_init,x,z);
    g5:entity work.or2(arc) port map (y,z,w);
    second(3)<='0';
    second(2)<=w;
    second(1)<=w;
    second(0)<='0';
    g6: entity work.adder_look_ahead_4bit(arc) port map (ans_init,second,'0',s);
        cout<=w;
    end architecture arc1;
```

Figure 13:1-Digit BCD adder code using look ahead adder

As we can see, the one-digit BCD adder is implanted using the 4-Bits ripple adder and the 4 bits look ahead adder.

Regarding to the 4-Bits ripple adder, first of all the first 4 least bits from each number were added together using the pre-implemented ripple adder, then the result was checked if It is more than 9 and if so six is added to it again using the same adder.

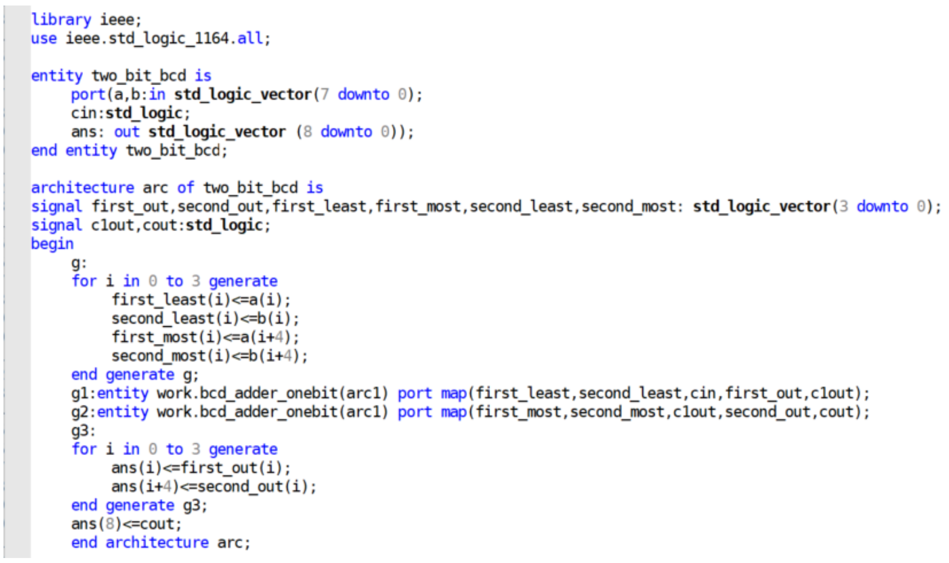
The BCD adder using the look ahead adder is implemented in the same way.

The signal used in the architecture were:

1- Signal x: to store the result of the first and gate that is used to detect if the number is larger than 9.
2- Signal y: to store the result of the second and gate that is used to detect if the number is larger than 9.
3- Signal ans_init: to store the initial answer of the adder before checking if the number is more than 9 and resolving it.
4- Signal w: this is used to store ones indicating that the number is more than 9 and then adding 6 to it instead of 0.

Building the 2-digits BCD adder

The 2-digit BCD adder was built structurally using the 1-digit BCD adder as follows:



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

entity two_bit_bcd is
     port(a,b:in std_logic_vector(7 downto 0);
     cin:std_logic;
     ans: out std_logic_vector (8 downto 0));
end entity two_bit_bcd;

architecture arc of two_bit_bcd is
signal first_out,second_out,first_least,first_most,second_least,second_most: std_logic_vector(3 downto 0);
signal clout,cout:std_logic;
begin
     g:
     for i in 0 to 3 generate
          first_least(i)<=a(i);
          second_least(i)<=b(i);
          first_most(i)<=a(i+4);
          second_most(i)<=b(i+4);
     end generate g;
     g1:entity work.bcd_adder_onebit(arc1) port map(first_least,second_least,cin,first_out,clout);
     g2:entity work.bcd_adder_onebit(arc1) port map(first_most,second_most,clout,second_out,cout);
     g3:
     for i in 0 to 3 generate
          ans(i)<=first_out(i);
          ans(i+4)<=second_out(i);
     end generate g3;
     ans(8)<=cout;
     end architecture arc;
```

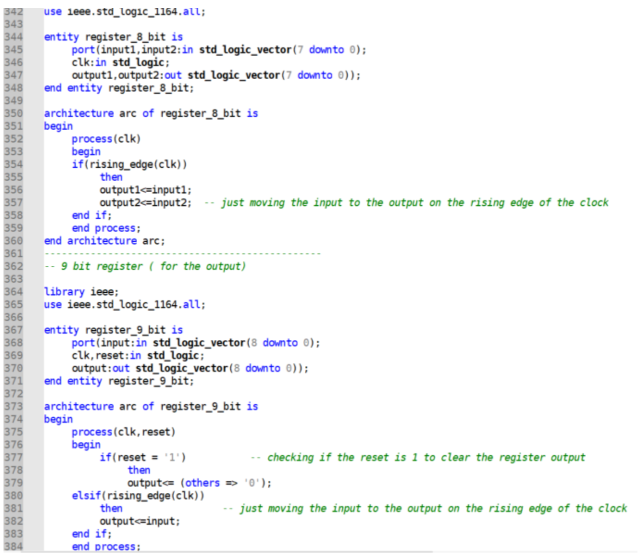
Figure 14:2-Digits BCD adder code

First of all, each number was divided into two parts (most significant 4 bits and least significant 4 bits). Then the least bits were added using the 1-digit BCD adder, after that the most significant bits were added with the carry of the least bits.

Finally, the final result was stored into the answer by merging the results of adding the least bits and the most bits using the generate statement for simplicity.

Building the registers

The registers were built behaviorally as follows:



```vhdl
use ieee.std_logic_1164.all;

entity register_8_bit is
     port(input1,input2:in std_logic_vector(7 downto 0);
     clk:in std_logic;
     output1,output2:out std_logic_vector(7 downto 0));
end entity register_8_bit;

architecture arc of register_8_bit is
begin
     process(clk)
     begin
     if(rising_edge(clk))
          then
          output1<=input1;
          output2<=input2;  -- just moving the input to the output on the rising edge of the clock
     end if;
     end process;
end architecture arc;
-------------------------------------------------
-- 9 bit register ( for the output)

library ieee;
use ieee.std_logic_1164.all;

entity register_9_bit is
     port(input:in std_logic_vector(8 downto 0);
     clk,reset:in std_logic;
     output:out std_logic_vector(8 downto 0));
end entity register_9_bit;

architecture arc of register_9_bit is
begin
     process(clk,reset)
     begin
          if(reset = '1')          -- checking if the reset is 1 to clear the register output
               then
               output<= (others => '0');
     elsif(rising_edge(clk))
          then                 -- just moving the input to the output on the rising edge of the clock
          output<=input;
     end if;
     end process;
```

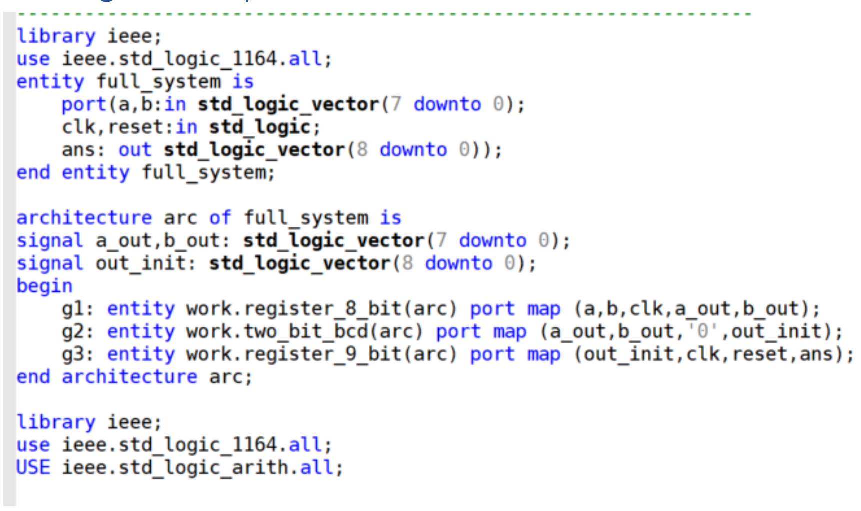
Figure 15:Registers code

Now the registers were built behaviorally using a simple condition that checks if the reset is one then the output is zero and otherwise if it is a rising edge of the clock then the input in transferred to the output.

The first register is an 8-bit input register that takes the 8 bits as two parts to simplify the addition operation later.

The second register is a 9-bit output register that is also built behaviorally, the register is 9 bits not 8 because the result of adding two BCD numbers is up to 9 bits.

Building the full system



```
----------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity full_system is
    port(a,b:in std_logic_vector(7 downto 0);
    clk,reset:in std_logic;
    ans: out std_logic_vector(8 downto 0));
end entity full_system;

architecture arc of full_system is
signal a_out,b_out: std_logic_vector(7 downto 0);
signal out_init: std_logic_vector(8 downto 0);
begin
    g1: entity work.register_8_bit(arc) port map (a,b,clk,a_out,b_out);
    g2: entity work.two_bit_bcd(arc) port map (a_out,b_out,'0',out_init);
    g3: entity work.register_9_bit(arc) port map (out_init,clk,reset,ans);
end architecture arc;

library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

Figure 16:Full system code

As shown in the code, the full system is built structurally using all the pre-implemented components.

There are many signals used in the architecture which are:

1- Signal a_out: to store the first number after passing through the input register
2- Signal b_out: to store the second number after passing through the input register.
3- Out_init: to store the output of the adder before passing the output register.

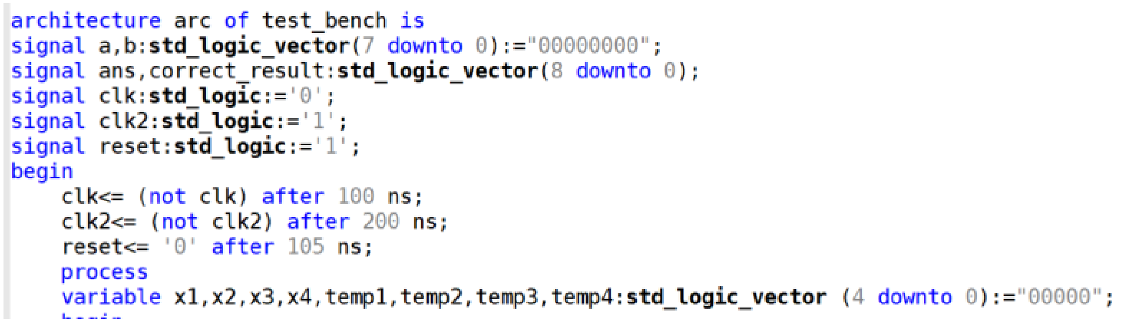
Building the test bench

The following test bench will work as a test bench and a test generator also; by generating all the possible test cases and check if the expected result is as same as the one taken from the adder through the result analyzer.

I think this part is the most complicated part of the project, so I will divide it into parts.

Part1 building the entity which as an empty entity with no inputs or outputs.

Part 2 defining the signals that will be used in testing the system.

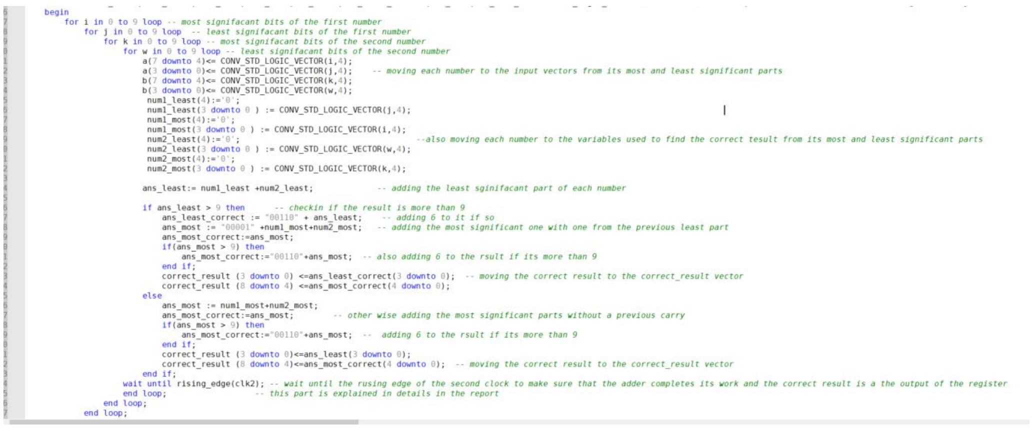


```
architecture arc of test_bench is
signal a,b:std_logic_vector(7 downto 0):="00000000";
signal ans,correct_result:std_logic_vector(8 downto 0);
signal clk:std_logic:='0';
signal clk2:std_logic:='1';
signal reset:std_logic:='1';
begin
    clk<= (not clk) after 100 ns;
    clk2<= (not clk2) after 200 ns;
    reset<= '0' after 105 ns;
    process
    variable x1,x2,x3,x4,temp1,temp2,temp3,temp4:std_logic_vector (4 downto 0):="00000";
```

Figure 17:Test bench code part 1

The signals are:

1- A and b which represent the inputs for the adders
2- Ans to store the output of the adder and correct_result to store the expected output.
3- Clk1: which represents the clock in which all registers will depend on
4- Clk2: **this clock is a little bit complicated so I will explain it in details:**
Since every one operation of the adder needs two clock cycles (one for passing the Inputs and one for passing the output while calculating the result between them) we need another clock that will be used for the result analyzer to check the result and for the test bench to generate the next test case as follows:
Suppose the first clock has a time of 100 ns (worst case) then the first input will be in at 100 ns and the output will be out at 300 ns, now I decided to make benefit from the falling edge of the first clock at 400 ns and use this time to check if the result is correct or not since the next test case won’t be available until 500 ns, like this I have initialized the second clock with 1 and negate it every 200 ns, as a result at every positive edge of this clock I will check if the result is correct or not.
Notes:
1- I decided to initialize the second clock with 1 to make benefit of the rising edge but we can also initialize it with 0 and check the result every falling edge.
2- we can use only one clock with two wait statements before generating the next test case, but I preferred this way because it is more obvious.
5- Reset: that will be used to reset the register.
6- The next variables will be discussed when explaining the operation of calculating the expected result.



```vhdl
begin
    for i in 0 to 9 loop -- most significant bits of the first number
        for j in 0 to 9 loop  -- least significant bits of the first number
            for k in 0 to 9 loop -- most significant bits of the second number
                for w in 0 to 9 loop -- least significant bits of the second number
                    a(7 downto 4)<= CONV_STD_LOGIC_VECTOR(i,4);
                    a(3 downto 0)<= CONV_STD_LOGIC_VECTOR(j,4);    -- moving each number to the input vectors from its most and least significant parts
                    b(7 downto 4)<= CONV_STD_LOGIC_VECTOR(k,4);
                    b(3 downto 0)<= CONV_STD_LOGIC_VECTOR(w,4);
                     num1_least(4):='0';
                     num1_least(3 downto 0 ) := CONV_STD_LOGIC_VECTOR(j,4);
                     num1_most(4):='0';
                     num1_most(3 downto 0 ) := CONV_STD_LOGIC_VECTOR(i,4);
                     num2_least(4):='0';                                  --also moving each number to the variables used to find the correct tesult from its most and least significant parts
                     num2_least(3 downto 0 ) := CONV_STD_LOGIC_VECTOR(w,4);
                     num2_most(4):='0';
                     num2_most(3 downto 0 ) := CONV_STD_LOGIC_VECTOR(k,4);

                    ans_least:= num1_least +num2_least;            -- adding the least sginifacant part of each number

                    if ans_least > 9 then      -- checkin if the result is more than 9
                        ans_least_correct := "00110" + ans_least;    -- adding 6 to it if so
                        ans_most := "00001" +num1_most+num2_most;   -- adding the most significant one with one from the previous least part
                        ans_most_correct:=ans_most;
                        if(ans_most > 9) then
                            ans_most_correct:="00110"+ans_most;  -- also adding 6 to the rsult if its more than 9
                        end if;
                        correct_result (3 downto 0) <=ans_least_correct(3 downto 0);  -- moving the correct result to the correct_result vector
                        correct_result (8 downto 4) <=ans_most_correct(4 downto 0);
                    else
                        ans_most := num1_most+num2_most;
                        ans_most_correct:=ans_most;        -- other wise adding the most significant parts without a previous carry
                        if(ans_most > 9) then
                            ans_most_correct:="00110"+ans_most;  --  adding 6 to the rsult if its more than 9
                        end if;
                        correct_result (3 downto 0)<=ans_least(3 downto 0);
                        correct_result (8 downto 4)<=ans_most_correct(4 downto 0);  -- moving the correct result to the correct_result vector
                    end if;
                wait until rising_edge(clk2); -- wait until the rusing edge of the second clock to make sure that the adder completes its work and the correct result is a the output of the register
                end loop;              -- this part is explained in details in the report
            end loop;
        end loop;
```

Figure 18:Test bench code part 2

Generating the test cases and finding the expected results

First of all four nested loops are implemented which indicate the following:

The variable i indicates the most significant 4 bits of the first number

The variable j indicates the least significant 4 bits of the first number

The variable k indicates the most significant 4 bits of the second number

The variable w indicates the least significant 4 bits of the second number

So, in the beginning of the loop, we move the bits of each number of the inputs a and b to them using the conversion function. Also, the two numbers are moved to two temp variables num1_most, num1_least, num2_most and num2_least that will be used to calculate the expected result. We can see that the variables are 5 bits not four because the result can be up to 5 bits and the VHDL gives a warning if the answer vector does not have the same number of bits as the two added vectors.

For finding the expected result, I will explain the process in a few points:

1- Now the two input numbers are in the signals that will be used as an input for the system and also in the variables num1_most, num1_least, num2_most and num2_least.
2- We will start calculating the expected result by adding the least 4 bit of each number which are stored in num1_least and nim2_least and store the result in another variable ans_least.
3- After that we will check if the variable ans_least is more than 9 and if so, we will add 6 to it again and store the result in a variable called ans_least_correct, also we will add 1 when adding the most significant bits (because we are sure that there will be a carry) and check if the result is more than 9 in the same way and store the final result of the most significant bits in ans_most_correct.
4- Otherwise, if the result of adding the least significant bits is less than or equal to 9, we will add the most ones without adding an extra carry and check in the same way if the result is acceptable or not and store the final result in ans_most_correct.

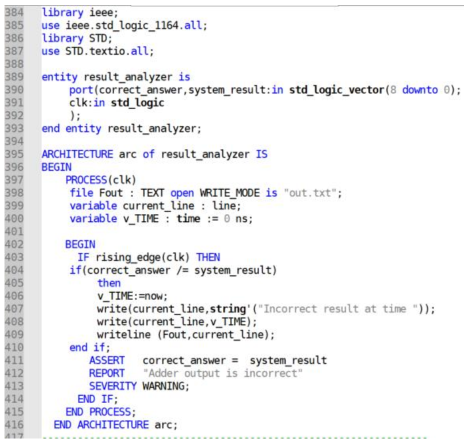
5- Then we transfer the expected result to the correct _answer vector to compare it with the adder result.

Finally, we will call the adder entity and the result analyzer entity to check if the result is correct or not.

- **Very important note:**

  To change between the ripple adder and the look ahead adder, just change the name of the full system architecture to ripple or look_ahead.

Budling the result analyzer



```
library ieee;
use ieee.std_logic_1164.all;
library STD;
use STD.textio.all;

entity result_analyzer is
     port(correct_answer,system_result:in std_logic_vector(8 downto 0);
     clk:in std_logic
     );
end entity result_analyzer;

ARCHITECTURE arc of result_analyzer IS
BEGIN
    PROCESS(clk)
     file Fout : TEXT open WRITE_MODE is "out.txt";
     variable current_line : line;
     variable v_TIME : time := 0 ns;

    BEGIN
      IF rising_edge(clk) THEN
     if(correct_answer /= system_result)
          then
          v_TIME:=now;
          write(current_line,string'("Incorrect result at time "));
          write(current_line,v_TIME);
          writeline (Fout,current_line);
     end if;
        ASSERT   correct_answer =  system_result
        REPORT   "Adder output is incorrect"
        SEVERITY WARNING;
      END IF;
    END PROCESS;
  END ARCHITECTURE arc;
```

Figure 19:Result analyzer code

The design is very simple, it takes the expected and the actual results and the clock, and if it is a rising edge of the clock it compares the two result and assert that they are the same with a warning severity if not. Also, the comparison is done using the simple if statement to write the output into an output file, using the file and line variables and the now variable to indicate at which time we are now.

Calculating the required clock period and maximum latency for each adder

The 2-digit BCD adder using the ripple 4-Bits adder

After testing the circuit under some various options for the clock I have found that the minimum clock needed to let the adder finish its work for all test cases is about 92 ns (frequency of about 10.86 MH), so we can realize that the maximum latency of the BCD adder using the ripple 4-Bits adder is about 92 ns or a little bit less.

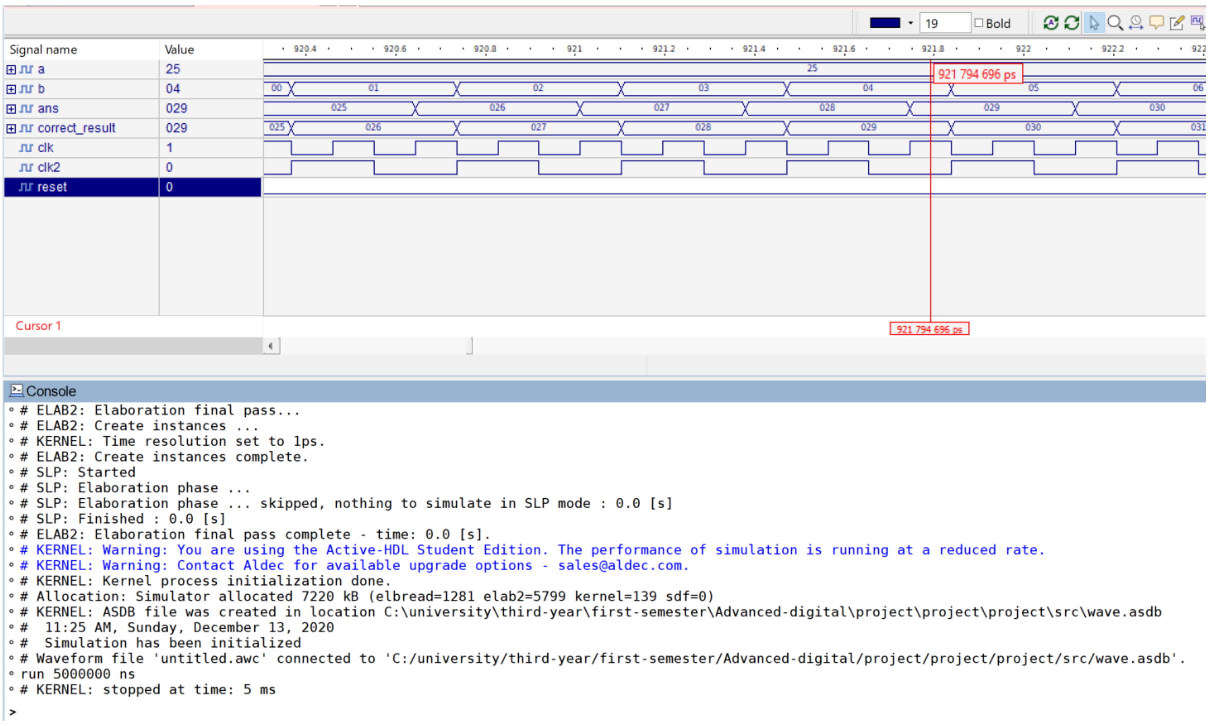
The 2-digit BCD adder using the look ahead 4-Bits adder

After testing the circuit under some various options for the clock I have found that the minimum clock needed to let the adder finish its work for all test cases is about 85 ns (frequency of about 11.76 MH), so we can realize that the maximum latency of the BCD adder using the look ahead 4-Bits adder is about 85 ns or a little bit less.

Simulation results and discussion

Simulation using the ripple adder

Simulation using a clock with time period = 92 ns

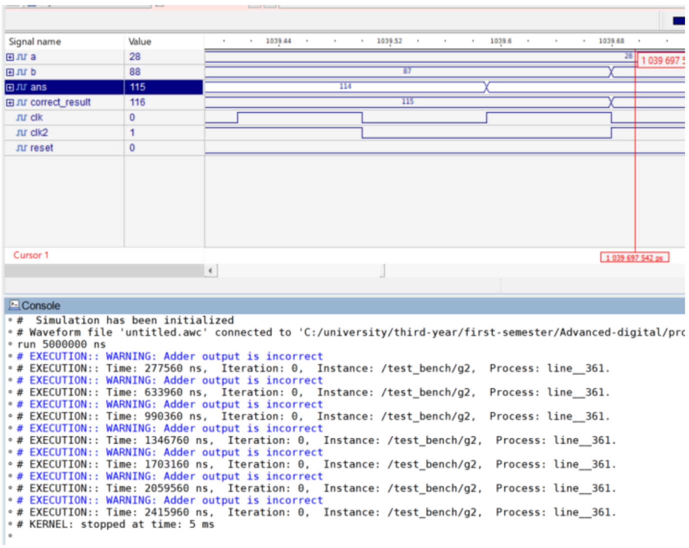


```
# ELAB2: Elaboration final pass...
# ELAB2: Create instances ...
# KERNEL: Time resolution set to 1ps.
# ELAB2: Create instances complete.
# SLP: Started
# SLP: Elaboration phase ...
# SLP: Elaboration phase ... skipped, nothing to simulate in SLP mode : 0.0 [s]
# SLP: Finished : 0.0 [s]
# ELAB2: Elaboration final pass complete - time: 0.0 [s].
# KERNEL: Warning: You are using the Active-HDL Student Edition. The performance of simulation is running at a reduced rate.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 7220 kB (elbread=1281 elab2=5799 kernel=139 sdf=0)
# KERNEL: ASDB file was created in location C:\university\third-year\first-semester\Advanced-digital\project\project\project\src\wave.asdb
#  11:25 AM, Sunday, December 13, 2020
#  Simulation has been initialized
# Waveform file 'untitled.awc' connected to 'C:/university/third-year/first-semester/Advanced-digital/project/project/project/src/wave.asdb'.
run 5000000 ns
# KERNEL: stopped at time: 5 ms
>
```

Figure 20:Ripple adder simulation result 1

As we can see from the simulation result there is no warnings indicating that the result is incorrect.

Simulation using a clock with a time period = 90 ns



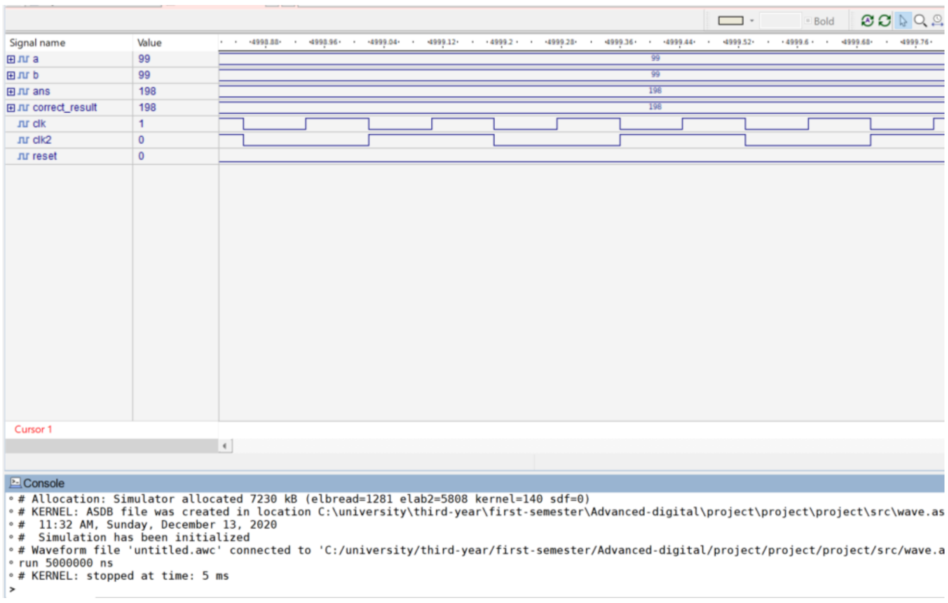
```
#  Simulation has been initialized
# Waveform file 'untitled.awc' connected to 'C:/university/third-year/first-semester/Advanced-digital/pro
run 5000000 ns
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 277560 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 633960 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 990360 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 1346760 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 1703160 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 2059560 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 2415960 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# KERNEL: stopped at time: 5 ms
```

Figure 21:Ripple adder simulation result 2

As we can see here the result analyzer indicated that the result of the adder is not equal to the expected one, that's because the clock is less than the maximum latency of the adder.

Simulation using the look ahead adder

Simulation using a clock with time period = 85 ns

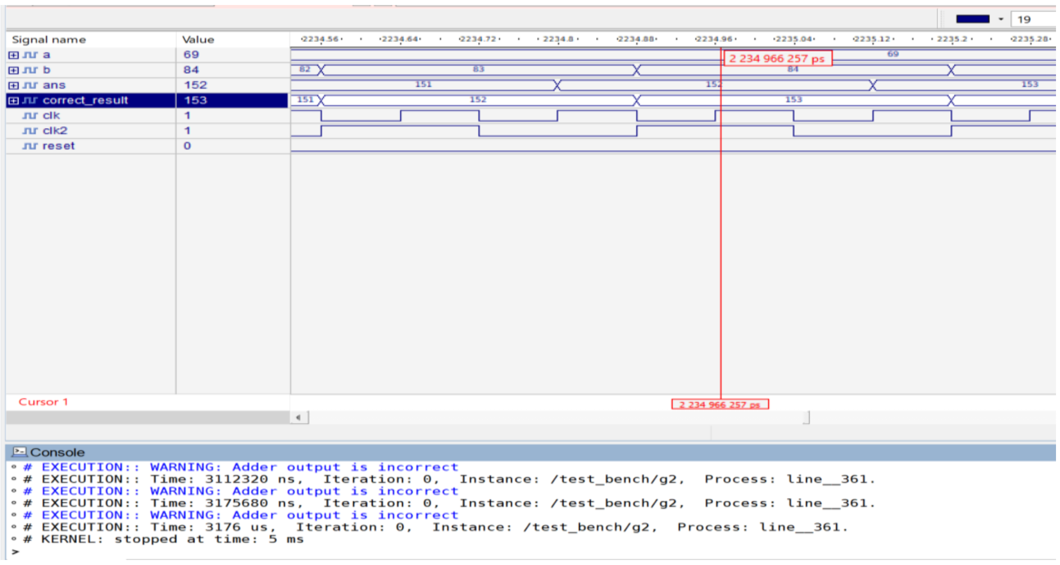


```
# Allocation: Simulator allocated 7230 kB (elbread=1281 elab2=5808 kernel=140 sdf=0)
# KERNEL: ASDB file was created in location C:\university\third-year\first-semester\Advanced-digital\project\project\project\src\wave.as
#  11:32 AM, Sunday, December 13, 2020
#  Simulation has been initialized
# Waveform file 'untitled.awc' connected to 'C:/university/third-year/first-semester/Advanced-digital/project/project/project/src/wave.a
run 5000000 ns
# KERNEL: stopped at time: 5 ms
>
```

Figure 22:look ahead adder simulation result 1

As we can see from the simulation result there is no warnings indicating that the result is incorrect.

Simulation using a clock with a time period = 80 ns



```
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 3112320 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 3175680 ns,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# EXECUTION:: WARNING: Adder output is incorrect
# EXECUTION:: Time: 3176 us,  Iteration: 0,  Instance: /test_bench/g2,  Process: line__361.
# KERNEL: stopped at time: 5 ms
>
```

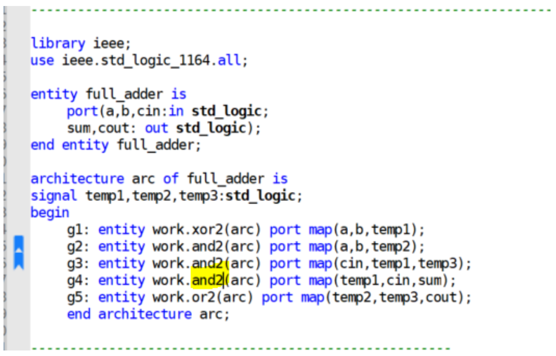
Figure 23:look ahead adder simulation result 2

As we can see here the result analyzer indicated that the result of the adder is not equal to the expected one, that's because the clock is less than the maximum latency of the adder.

Simulation with a fault in the circuit

In this part I am going to run the simulation again but with changing a random gate to a different one in order to get wrong results.

For example, I will change the XOR gate in the full adder to and gate as follows

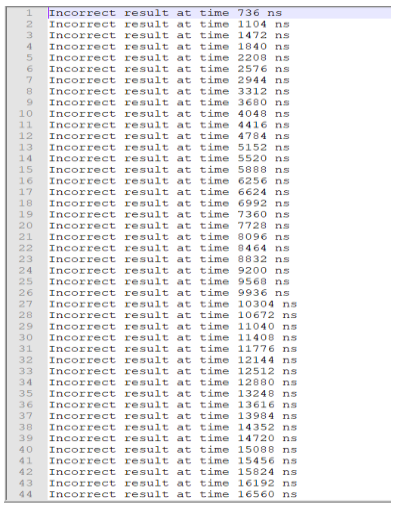


```
---------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
     port(a,b,cin:in std_logic;
     sum,cout: out std_logic);
end entity full_adder;

architecture arc of full_adder is
signal temp1,temp2,temp3:std_logic;
begin
     g1: entity work.xor2(arc) port map(a,b,temp1);
     g2: entity work.and2(arc) port map(a,b,temp2);
     g3: entity work.and2(arc) port map(cin,temp1,temp3);
     g4: entity work.and2(arc) port map(temp1,cin,sum);
     g5: entity work.or2(arc) port map(temp2,temp3,cout);
     end architecture arc;

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

Figure 24:Full adder code with a fault

As we can see from the above code, the highlighted gate should be a XOR gate but to make the circuit faulty I will make it and gate. Also, the clock is set to be more than the worst case; to make sure that the warning is due to the wrong design in the circuit and no due to the clock.

Since in the previous simulations I took the results from the default console, here I will take them from the output file.



```
Incorrect result at time 736 ns
Incorrect result at time 1104 ns
Incorrect result at time 1472 ns
Incorrect result at time 1840 ns
Incorrect result at time 2208 ns
Incorrect result at time 2576 ns
Incorrect result at time 2944 ns
Incorrect result at time 3312 ns
Incorrect result at time 3680 ns
Incorrect result at time 4048 ns
Incorrect result at time 4416 ns
Incorrect result at time 4784 ns
Incorrect result at time 5152 ns
Incorrect result at time 5520 ns
Incorrect result at time 5888 ns
Incorrect result at time 6256 ns
Incorrect result at time 6624 ns
Incorrect result at time 6992 ns
Incorrect result at time 7360 ns
Incorrect result at time 7728 ns
Incorrect result at time 8096 ns
Incorrect result at time 8464 ns
Incorrect result at time 8832 ns
Incorrect result at time 9200 ns
Incorrect result at time 9568 ns
Incorrect result at time 9936 ns
Incorrect result at time 10304 ns
Incorrect result at time 10672 ns
Incorrect result at time 11040 ns
Incorrect result at time 11408 ns
Incorrect result at time 11776 ns
Incorrect result at time 12144 ns
Incorrect result at time 12512 ns
Incorrect result at time 12880 ns
Incorrect result at time 13248 ns
Incorrect result at time 13616 ns
Incorrect result at time 13984 ns
Incorrect result at time 14352 ns
Incorrect result at time 14720 ns
Incorrect result at time 15088 ns
Incorrect result at time 15456 ns
Incorrect result at time 15824 ns
Incorrect result at time 16192 ns
Incorrect result at time 16560 ns
```

Figure 25:Faulty circuit simulation result

As shown in the file a lot of outputs were wrong due to the new added fault.

Very important note about the simulation:

We can see that at some moments the output of the adder is not as same as the expected one and there is no warning. That's not a problem because this wrong result is because the adder result is not ready yet, for that reason the clock is chosen in a way to make sure that the result analyzer won't check unless the adder has completed the process (for the worst case).

Conclusion

After ending the project, I can say that it was a very interesting one that took me from the theoretical part of the course to an application close enough to what is done in the real life.

Also, I should emphasize on some points:

1- When designing a circuit, we must simulate it using a hardware description language like VHDL to reduce the size of losses.

2- When designing circuits virtually we should pay attention to code styling, like for example every component should be in a separate entity to reduce the time needed in debugging and finding the problem if an error occurs.

3- As we can see from the previous test bench, we have tested all the possible test cases for the inputs; because the number of possibilities for them is not very large, but in other applications this can't be done due to the very huge space on possibilities for the inputs, so we need to find smarter design verification techniques.

4- Finally, I have to draw your attention to the vital role of the result analyzer clock time period that should be chosen precisely because the full system may work fine but the time in which the result analyzer checks the result is not enough for the system to complete its work.

References

1- https://www.geeksforgeeks.org/carry-look-ahead-adder/

Appendix

Full code

```
 -- All the signals and variables were explained in details in the report



        --Basic Gates
-----------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity and2  is
            port( a,b:in std_logic;
            c: out std_logic);
end entity and2;
                                                                                       -- 2 inputs and
gate
architecture arc of and2 is
begin
            c<= (a) and (b) after 8 ns;
end architecture arc;
-----------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity or2  is
            port( a,b:in std_logic;
            c: out std_logic);
end entity or2;                                                                        -- 2 inputs or
gate
```

```
architecture arc of or2 is
begin
        c<= (a) or (b) after 8 ns;
end architecture arc;
------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity xor2  is
        port( a,b:in std_logic;
        c: out std_logic);                              -- 2 inputs xor gate
end entity xor2;

architecture arc of xor2 is
begin
        c<= (a) xor (b) after 12 ns;
end architecture arc;
--------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity and3 is
        port(a,b,c:in std_logic;
        f:    out std_logic);                       -- 3 inputs and gate
end entity and3;

architecture arc of and3 is
begin
        f<= (a) and (b) and (c) after 8 ns;
```

```
            end architecture arc;


------------------------------------


library ieee;

use ieee.std_logic_1164.all;


entity and4 is

            port(a,b,c,d:in std_logic;

            f:    out std_logic);                                              -- 4 inputs and
gate

end entity and4;


architecture arc of and4 is

begin

            f<= (a) and (b) and (c) and (d) after 8ns;

            end architecture arc;


----------------------------------------------


library ieee;

use ieee.std_logic_1164.all;


entity or3 is

            port(a,b,c:in std_logic;

            f:    out std_logic);

end entity or3;                                                                    -- 3 inputs
or gate


architecture arc of or3 is
```

```
signal x:std_logic;

begin

            f<= (a) or (b) or (c) after 8ns;

            end architecture arc;


-----------------------------------------------------


library ieee;

use ieee.std_logic_1164.all;


entity or4 is

            port(a,b,c,d:in std_logic;                                                              -- 4
inputs or gate

            f:    out std_logic);

end entity or4;


architecture arc of or4 is

begin

            f<= (a) or (b) or (c) or (d) after 8ns;

            end architecture arc;


-------------------------------------------------


                --Adders

---------------------------------------------------------------------


-- full adder structurly using gates


library ieee;

use ieee.std_logic_1164.all;
```

```
entity full_adder is
            port(a,b,cin:in std_logic;
            sum,cout: out std_logic);
end entity full_adder;


architecture arc of full_adder is
signal temp1,temp2,temp3:std_logic;
begin
            g1: entity work.xor2(arc) port map(a,b,temp1); -- calling the xor gate to find the sum of the first two bits
            g2: entity work.and2(arc) port map(a,b,temp2); -- calling the first and gate to find the first input for the or gate to get the carry
            g3: entity work.and2(arc) port map(cin,temp1,temp3);  -- calling the second and gate to find the second input for the or gate to get the carry
            g4: entity work.xor2(arc) port map(temp1,cin,sum);  -- calling the xor gate to find the sum of the three bits
            g5: entity work.or2(arc) port map(temp2,temp3,cout); -- calling the or gate to find the final carry
            end architecture arc;

------------------------------------------------------


-- 4-bit full adder structurly using 1-bit full adder


library ieee;
use ieee.std_logic_1164.all;


entity four_bit_adder is
            port(a,b:in std_logic_vector(3 downto 0);
              cin:in std_logic;

               s: out std_logic_vector (3 downto 0);


             cout: out std_logic);
end entity four_bit_adder;


architecture arc of four_bit_adder is
signal c:std_logic_vector(4 downto 0);
begin
        c(0)<=cin; -- moving the first carry to the carry vector
        g:
        for i in 0 to 3 generate            -- generate statement to reduce the size of the code
                g1:entity work.full_adder(arc) port map(a(i),b(i),c(i),s(i),c(i+1)); -- calling the
full adder entity for all the bits with the previous carry
        end generate g;
        cout<=c(4);        -- copying the last carry to the final carry of the system
end architecture arc;


-----------------------------------------


-- carry generator for the four bit look ahead adder


library ieee;
use ieee.std_logic_1164.all;


entity carry_generator is
        port(v:in std_logic_vector(6 downto 0);
        c2,c3,c4:out std_logic);
end entity carry_generator;
```

```
architecture arc of carry_generator is

signal p1,p2,p3,g1,g2,g3,a,b,c,d,e,f:std_logic;           -- every pi stands for the sum from the
ith half adder and gi stands for the carry from the ith half adder

begin

            x1:entity work.and2(arc) port map(v(1),v(2),g1);

            x2:entity work.xor2(arc) port map(v(1),v(2),p1);

            x3:entity work.and2(arc) port map(v(3),v(4),g2);

            x4:entity work.xor2(arc) port map(v(3),v(4),p2);

            x5:entity work.and2(arc) port map(v(5),v(6),g3);

            x6:entity work.xor2(arc) port map(v(5),v(6),p3);

            x7:entity work.and2(arc) port map(v(0),p1,a);

            x8:entity work.or2(arc) port map(a,g1,c2);                                        -- this part
works as same as discussed in the introduction part

            x9:entity work.and2(arc) port map(g1,p2,b);

            x10:entity work.and3(arc) port map(p2,p1,v(0),c);

            x11:entity work.or3(arc) port map(b,c,g2,c3);

            x12:entity work.and2(arc) port map(g2,p3,d);

            x13:entity work.and3(arc) port map(g1,p2,p3,e);

            x14:entity work.and4(arc) port map(v(0),p1,p2,p3,f);

   x15:entity work.or4(arc) port map(g3,f,e,d,c4);


end architecture arc;


-----------------------------------------------------------------


-- 4-bit look ahead adder structurly using carry generator



library ieee;

use ieee.std_logic_1164.all;
```

```
entity adder_look_ahead_4bit is
        port(a,b:in std_logic_vector(3 downto 0);
        cin:in std_logic;
        s: out std_logic_vector(3 downto 0);
        cout: out std_logic);
end entity adder_look_ahead_4bit;


architecture arc of adder_look_ahead_4bit is
signal v:std_logic_vector(6 downto 0);
signal c1,c2,c3,c4:std_logic;
begin
        x1: entity work.full_adder(arc) port map(a(0),b(0),cin,s(0),c1);      -- calling the first
adder to get the first carry
        v(0)<=c1;
        v(1)<=a(1);
        v(2)<=b(1);
        v(3)<=a(2);                        -- initializing the vector that will be sent to the carry
genrator
        v(4)<=b(2);
        v(5)<=a(3);
        v(6)<=b(3);
x2:entity work.carry_generator(arc) port map(v,c2,c3,c4); -- calling the carry generator to
get all the needed carries
x3: entity work.full_adder(arc) port map(a(1),b(1),c1,s(1));   -- calling the second full adder
x4: entity work.full_adder(arc) port map(a(2),b(2),c2,s(2));   -- calling the third full adder
x5: entity work.full_adder(arc) port map(a(3),b(3),c3,s(3));  -- calling the fourth full adder
cout<=c4;        -- moving the last carry to the output carry of the system
end architecture arc;
-----------------------------------------------------------------
```

```
-- 1 bit bcd adder using ripple and look ahead adders



library ieee;
use ieee.std_logic_1164.all;


entity bcd_adder_onebit is
        port(a,b:in std_logic_vector(3 downto 0);
        cin:in std_logic;
        s:out std_logic_vector(3 downto 0);
        cout:out std_logic);
end entity bcd_adder_onebit;


architecture ripple of bcd_adder_onebit is -- using ripple adder
signal x,y,w,cout_init:std_logic;   -- x and y are temp signals to store the output of the first
and second and gates that checks if the number is more than 9
signal ans_init,second:std_logic_vector(3 downto 0);-- vector to store the initail result
before correcting it if necessary
begin
        g1: entity work.four_bit_adder(arc) port map (a,b,cin,ans_init,cout_init);   -- calling
the four bit ripple adder by sending the 4 bits of each bcd digit
        g2:entity work.and2(arc) port map (ans_init(2),ans_init(3),x);          -- checking if the
output if more than 9
        g3:entity work.and2(arc) port map (ans_init(3),ans_init(1),y);          -- checking if the
output is more than 9
        g4:entity work.or3(arc) port map (cout_init,x,y,w);              -- determining if the
output is more than 9 and storing 1 in w if so
        second(3)<='0';

        second(2)<=w;   -- inistializing the vector that will be added to the output to
recorrect it
```

```
            second(1)<=w;

            second(0)<='0';

            g6: entity work.four_bit_adder(arc) port map (ans_init,second,'0',s);          --
correcting the output by adding 6 to if if it is more than 9 and 0 otherwise

                        cout<=w;          -- moving one tto the output carry if the number was
corrected by adding 6 to it

            end architecture ripple;


architecture look_ahead of bcd_adder_onebit is       -- using look ahead adder

signal x,y,z,w,cout_init:std_logic;

signal ans_init,second:std_logic_vector(3 downto 0);

begin

            g1: entity work.adder_look_ahead_4bit(arc) port map (a,b,cin,ans_init,cout_init);

            g2:entity work.and2(arc) port map (ans_init(2),ans_init(3),x);

            g3:entity work.and2(arc) port map (ans_init(3),ans_init(1),y);

            g4:entity work.or2(arc) port map (cout_init,x,z);

            g5:entity work.or2(arc) port map (y,z,w);                                              --
this is as same as the previous one but with changing the type of the 4-bits full adder

            second(3)<='0';

            second(2)<=w;

            second(1)<=w;

            second(0)<='0';

            g6: entity work.adder_look_ahead_4bit(arc) port map (ans_init,second,'0',s);

                        cout<=w;

            end architecture look_ahead;


-------------------------------------------------------------------

-- 2 bits bcd adder using 1-bit bcd adder and ripple adder

library ieee;

use ieee.std_logic_1164.all;
```

```
entity two_bit_bcd is
        port(a,b:in std_logic_vector(7 downto 0);
        cin:std_logic;
        ans: out std_logic_vector (8 downto 0));
end entity two_bit_bcd;


architecture ripple of two_bit_bcd is
signal first_out,second_out,first_least,first_most,second_least,second_most:
std_logic_vector(3 downto 0);-- vectors to store the most and least significant parts of each
bcd number and for the outputs of each adder
signal c1out,cout:std_logic;
begin
        g:
        for i in 0 to 3 generate


                first_least(i)<=a(i);


                second_least(i)<=b(i);    -- moving the least and most bits of each number to
their vectors
                first_most(i)<=a(i+4);
                second_most(i)<=b(i+4);
        end generate g;
        g1:entity work.bcd_adder_onebit(ripple) port
map(first_least,second_least,cin,first_out,c1out);      -- calling the first 1-digit bcd adder for
the least bytes
        g2:entity work.bcd_adder_onebit(ripple) port
map(first_most,second_most,c1out,second_out,cout);        -- calling the second 1-digit bcd
adder for the least bytes
        g3:
        for i in 0 to 3 generate
```

```
                    ans(i)<=first_out(i);                -- moving the final result to the final answer
vector
                    ans(i+4)<=second_out(i);
          end generate g3;
          ans(8)<=cout;               -- adding the last carry for the final answer
          end architecture ripple;

---------------------------------------------------------------------------------------------------


-- 2 bits bcd adder using 1-bit bcd adder and look ahead adder



library ieee;
use ieee.std_logic_1164.all;

entity two_bit_bcd is
          port(a,b:in std_logic_vector(7 downto 0);
          cin:std_logic;
          ans: out std_logic_vector (8 downto 0));
end entity two_bit_bcd;

architecture look_ahead of two_bit_bcd is
signal first_out,second_out,first_least,first_most,second_least,second_most:
std_logic_vector(3 downto 0);
signal c1out,cout:std_logic;
begin
          g:
          for i in 0 to 3 generate
```

```
                    first_least(i)<=a(i);

                    second_least(i)<=b(i);

                    first_most(i)<=a(i+4);
                                                            -- this is as same as the previous one but with
changing the type of the 4-bits full adder

                    second_most(i)<=b(i+4);

          end generate g;

          g1:entity work.bcd_adder_onebit(look_ahead) port
map(first_least,second_least,cin,first_out,c1out);

          g2:entity work.bcd_adder_onebit(look_ahead) port
map(first_most,second_most,c1out,second_out,cout);

          g3:

          for i in 0 to 3 generate

                    ans(i)<=first_out(i);

                    ans(i+4)<=second_out(i);

          end generate g3;

          ans(8)<=cout;

          end architecture look_ahead;


---------------------------------------------------


                                        --Registers

---------------------------------------------------


-- 8 bit register ( for the input )


library ieee;

use ieee.std_logic_1164.all;


entity register_8_bit is
```

```
	port(input1,input2:in std_logic_vector(7 downto 0);
	clk:in std_logic;
	output1,output2:out std_logic_vector(7 downto 0));
end entity register_8_bit;

architecture arc of register_8_bit is
begin
	process(clk)
	begin
	if(rising_edge(clk))
		then
		output1<=input1;
		output2<=input2;  -- just moving the input to the output on the rising edge of
the clock
	end if;
	end process;
end architecture arc;
-----------------------------------------------
-- 9 bit register ( for the output)

library ieee;
use ieee.std_logic_1164.all;

entity register_9_bit is
	port(input:in std_logic_vector(8 downto 0);
	clk,reset:in std_logic;
	output:out std_logic_vector(8 downto 0));
end entity register_9_bit;

architecture arc of register_9_bit is
```

```
begin

        process(clk,reset)
        begin
                if(reset = '1')         -- checking if the reset is 1 to clear the register output
                        then
                        output<= (others => '0');
        elsif(rising_edge(clk))
                then                            -- just moving the input to the output on the
rising edge of the clock
                output<=input;
        end if;
        end process;
end architecture arc;
---------------------------------------------------------------

                                        --Result Analyzer
---------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
library STD;
use STD.textio.all;

entity result_analyzer is
        port(correct_answer,system_result:in std_logic_vector(8 downto 0);
        clk:in std_logic
        );
end entity result_analyzer;

ARCHITECTURE arc of result_analyzer IS
```

```
BEGIN
   PROCESS(clk)
           file Fout : TEXT open WRITE_MODE is "out.txt"; -- initializing the output file pointer
           variable current_line : line;                                   -- initializng the line that will be
used to write into the file
           variable v_TIME : time := 0 ns;                                         -- initializing a time
variable to get the current time


   BEGIN
     IF rising_edge(clk) THEN
           if(correct_answer /= system_result)     -- chceking the result is incorrect
                     then
                     v_TIME:=now;                        -- making the time variable equals the current
simulation time
                     write(current_line,string'("Incorrect result at time "));  -- writng the warning
to the line
                     write(current_line,v_TIME);
                     writeline (Fout,current_line);               -- writing the line on the file
           end if;
      ASSERT   correct_answer =  system_result
      REPORT   "Adder output is incorrect"        -- asserting again if the result is correct with a
warning serverity
      SEVERITY WARNING;
     END IF;
   END PROCESS;
  END ARCHITECTURE arc;
----------------------------------------------------------------
--Full system


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

```
entity full_system is

            port(a,b:in std_logic_vector(7 downto 0);

            clk,reset:in std_logic;

            ans: out std_logic_vector(8 downto 0));

end entity full_system;


architecture ripple of full_system is

signal a_out,b_out: std_logic_vector(7 downto 0);
              -- full system using ripple adder and all previous components structurly

signal out_init: std_logic_vector(8 downto 0);

begin

            g1:       entity work.register_8_bit(arc) port map (a,b,clk,a_out,b_out);        -- calling
the input register

            g2: entity work.two_bit_bcd(ripple) port map (a_out,b_out,'0',out_init); -- calling the
2 digits bcd adder

            g3:       entity work.register_9_bit(arc) port map (out_init,clk,reset,ans);-- calling the
output register

end architecture ripple;


architecture look_ahead of full_system is

signal a_out,b_out: std_logic_vector(7 downto 0);
              -- full system using look ahead adder and all previous components structurly

signal out_init: std_logic_vector(8 downto 0);

begin

            g1:       entity work.register_8_bit(arc) port map (a,b,clk,a_out,b_out);

            g2: entity work.two_bit_bcd(look_ahead) port map (a_out,b_out,'0',out_init);

            g3:       entity work.register_9_bit(arc) port map (out_init,clk,reset,ans);

end architecture look_ahead;


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

```
                                --Test Bench
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity test_bench is
end entity test_bench;


architecture arc of test_bench is
signal a,b:std_logic_vector(7 downto 0):="00000000";
signal ans,correct_result:std_logic_vector(8 downto 0);
signal clk:std_logic:='0';
signal clk2:std_logic:='1';
signal reset:std_logic:='1';
begin
        clk<= (not clk) after 92 ns;            -- here we set the clock time to the worst case
as shown in the report
        clk2<= (not clk2) after 184 ns;                 -- here we set the clock time to double
the worst case also as shown in the report //this clock is explained in deatails in the report
        reset<= '0' after 105 ns;
        process
        variable
num1_least,num2_least,num1_most,num2_most,ans_least,ans_least_correct,ans_most,ans
_most_correct:std_logic_vector (4 downto 0):="00000"; -- variables will be shown during
disucssing the code
        begin
                for i in 0 to 9 loop -- most signifacant bits of the first number
                        for j in 0 to 9 loop  -- least signifacant bits of the first number
```

```
                                        for k in 0 to 9 loop -- most signifacant bits of the second
number
                                                  for w in 0 to 9 loop -- least signifacant bits of the
second number
                                                            a(7 downto 4)<=
CONV_STD_LOGIC_VECTOR(i,4);
                                                            a(3 downto 0)<=
CONV_STD_LOGIC_VECTOR(j,4);          -- moving each number to the input vectors from its
most and least significant parts
                                                            b(7 downto 4)<=
CONV_STD_LOGIC_VECTOR(k,4);
                                                            b(3 downto 0)<=
CONV_STD_LOGIC_VECTOR(w,4);
                                                             num1_least(4):='0';
                                                             num1_least(3 downto 0 ) :=
CONV_STD_LOGIC_VECTOR(j,4);
                                                             num1_most(4):='0';
                                                             num1_most(3 downto 0 ) :=
CONV_STD_LOGIC_VECTOR(i,4);
                                                             num2_least(4):='0';
                                                  --also moving each number to the variables used to
find the correct tesult from its most and least significant parts
                                                             num2_least(3 downto 0 ) :=
CONV_STD_LOGIC_VECTOR(w,4);
                                                             num2_most(4):='0';
                                                             num2_most(3 downto 0 ) :=
CONV_STD_LOGIC_VECTOR(k,4);


                                                  ans_least:= num1_least +num2_least;
                    -- adding the least sginifacant part of each number



                                                            if ans_least > 9 then      -- checkin if the result is
more than 9
                                                                      ans_least_correct := "00110" +
ans_least;        -- adding 6 to it if so
```

```
                                                              ans_most := "00001"
+num1_most+num2_most;    -- adding the most significant one with one from the previous
least part
                                                              ans_most_correct:=ans_most;
                                                              if(ans_most > 9) then

         ans_most_correct:="00110"+ans_most;        -- also adding 6 to the rsult if its more
than 9
                                                              end if;
                                                              correct_result (3 downto 0)
<=ans_least_correct(3 downto 0);  -- moving the correct result to the correct_result vector
                                                              correct_result (8 downto 4)
<=ans_most_correct(4 downto 0);
                                                     else
                                                              ans_most := num1_most+num2_most;
                                                              ans_most_correct:=ans_most;
           -- other wise adding the most significant parts without a previous carry
                                                              if(ans_most > 9) then

         ans_most_correct:="00110"+ans_most;  --  adding 6 to the rsult if its more than 9
                                                              end if;
                                                              correct_result (3 downto
0)<=ans_least(3 downto 0);
                                                              correct_result (8 downto
4)<=ans_most_correct(4 downto 0);  -- moving the correct result to the correct_result vector
                                                     end if;
                                             wait until rising_edge(clk2); -- wait until the rusing
edge of the second clock to make sure that the adder completes its work and the correct
result is a the output of the register
                                             end loop;                                  -- this part is
explained in details in the report
                                     end loop;
                             end loop;
                     end loop;
```

```
			wait;
		end process;
		g1:entity work.full_system (ripple) port map (a,b,clk,reset,ans);--calling the full
system to fin the sum ( you can use the look ahead adder by changing the ARchitecture to
look_ahead and clock 1 to 85ns and clock 2 to 170ns )
		g2:entity work.result_analyzer (arc) port map (correct_result,ans,clk2); -- calling the
result analyzer to check the result
END ARchitecture arc;
```