Department of Electrical and Computer Engineering Second Semester

2022/2023 ENCS3310 Course Final Project Report

Name: Al- Ayham Maree

ID: 119408

Date: 12/6/2022

Instructor: Dr. Abdellatif Abu Isaa

Subject: Project for 2-digit BCD Adder

Table of Contents:

Introduction:

In this project, what is required is to build 2-digit BCD Adder by write a complete code using Verilog Hardware Description Language and this project is divided into two stages, the first stage must be built using Carry Ripple Adder and the second stage must be built using Carry Look Ahead Adder and the code must be built structurally.

Before delving into the depth of the project, a BCD adder is a 4-bit binary adder that can combine two 4-bit BCD format. The addition produces a BCD-format 4-bit output word that contains the decimal sum of the addend and augend, as well as a carry if the sum exceeds a decimal value of 9. As a result, BCD adders may do decimal addition.

The BCD format is a binary-coded decimals are an easy way to represent decimal values, as each digit is represented by its own 4-bit binary sequence which only has 10 different combinations. By comparison, converting real binary representation to decimal requires arithmetic operations like multiplication and addition.

Conversion to decimal digits for display or printing is simpler, but the resultant circuit required to implement this method is more complicated. There are three decimal digits in the binary coded decimal "1001 0101 0110," which contains three groups of four bits. The resultant decimal value is 956 when ordered from left to right.

The most important component of this project is to validate the design of our designs (Design Verification), and this process is done by designing the circuit you want and testing the generator, test analyzer and test bench for the entire system in your design, the circuit you design works as usual as for the test generator the task is to produce results and give Inputs to the original piece that you build, meaning that it gives an expected value for this design. It is entered into the test analyzer and compared with the value out of the original design. If it is not correct or equal, a warning will be issued to you that there is an error in one of the values, and this is one of the most important elements of building a correct and complete design.

Then, after the system is built, a test bench is created where the basic inputs are made as registers and the outputs are like wires, and their function as a module is to give a value to each input in order to get what the entire system has to be produced, precedes this process and the build process is to define The time delay for each gate you created and the full time to use this system in the test bench. The goal is not to get late results and there is great importance to maintain time consistency in building these designs that many devices in the world rely on and that any error for any design somewhere is possible it can lead to trouble and possibly atrocities

Each stage will have a test bench with two registers: a test generator that sends inputs to our system and outputs to the second register, as well as a clock input; and a result analyzer that receives the behavioral output from the test generator and receives the system's output, ensuring that the two outputs are correct. Finally, we will simulate my project and test all results and outputs in both stages using (Aldec Active-HDL Student Edition).

Theoretical Overview:

BCD Adder is a combinational logic circuit in digital systems that performs the addition of two binary values in BCD format (Binary Coded Decimal). These are frequently employed in the execution of many algorithms. Depending on the stages of our project, we have two major theories:

1. **Stage (1): 2- Digit BCD Adder Using Carry Ripple Adder:**

   To add an N-bit number, several complete adder circuits can be cascaded in simultaneously. There must be N number of complete adder circuits in an N-bit parallel adder. A ripple carry adder is a logic circuit in which each full adder's carry-out is the carry-in of the next most important full adder. Because each carry bit ripples into the following step, it's termed a ripple carry adder. The sum and carry out bits of each half adder stage in a ripple carry adder are not valid until that stage's carry in happens. This is caused by propagation delays inside the logic circuitry.

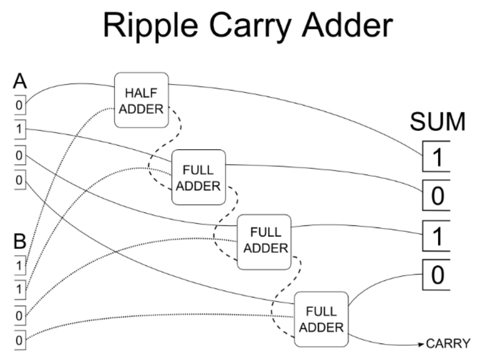


Figure 1: Carry Ripple Adder Concept

   The duration between the application of an input and the occurrence of the matching output is known as propagation delay. When the input is "0," the output is "1," and vice versa. The propagation delay is the time it takes for the NOT gate's output to become "0" after logic "1" is applied to the NOT gate's input. Similarly, the carry propagation delay is the period between when the carry in signal is applied and when the carry out (Cout) signal occurs.

   And we must use two pieces of this design to obtain two BCD Adder numbers, which must be linked to the project scenario, and the design's delay and latency rate, as well as the maximum value of the frequency, must be determined by tracing the inputs and outputs to the structurally built pieces and obtaining the required values, and I built it using structurally Verilog code using (and, or, xor), and in this stage we need two ripple adder to build one BCD adder and then we can connect between to BCD adder to result 2-digit BCD Adder.

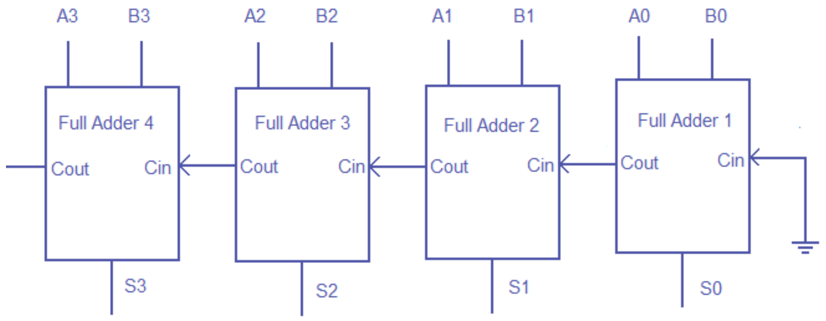


Figure 2: Ripple Carry Adder

2. Stage (2): 2-Digit BCD Adder using Look Ahead Adder:

Carry Look-ahead Adder is the faster adder circuit. It reduces the propagation delay, which occurs during addition, by using more complex hardware circuitry. It is designed by transforming the ripple-carry Adder circuit such that the carry logic of the adder is changed into two-level logic**.**

Carry look ahead adders generate two bits termed Carry Propagate and Carry Generate, which are denoted by the letters Cp and Cg. The Cp bit is passed on to the next step, and the Cg bit is utilized to generate the output carry bit, which is unrelated to the input carry bit. As shown below:

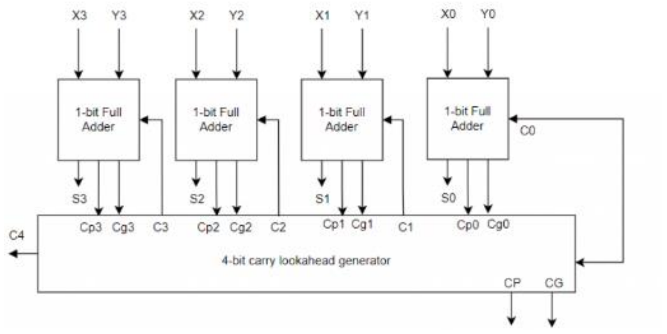


Figure 3: Look Ahead Adder

We'll need two Boolean expressions to build the carry look ahead adder formulas for carry propagate Cp and carry generate Cg.

$Cp_i = X \wedge Y$

$Cg = X. Y$

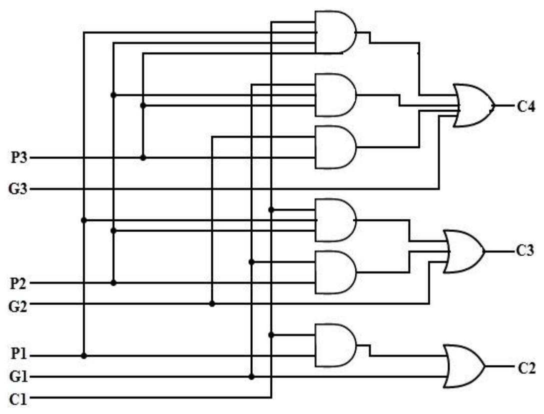


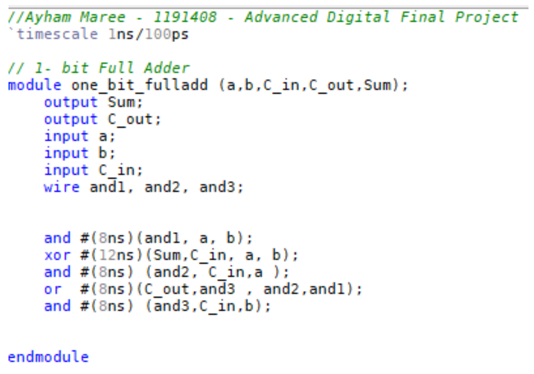
Figure 4: Carry Ahead Adder Using Basic Gates

At this stage, we have to create a two-digit BCD Adder using two Carry Look Ahead Adder as required, as here the result is expected to be faster than the first stage using Carry Ripple Adder, due to what was explained and mentioned above, and I built it structurally Verilog code using (and, or, xor), and in this stage we need two look ahead adder to build one BCD adder and then we can connect between to BCD adder to result 2-digit BCD Adder.

Design philosophy:

1- Stage 1 : using carry ripple adder:

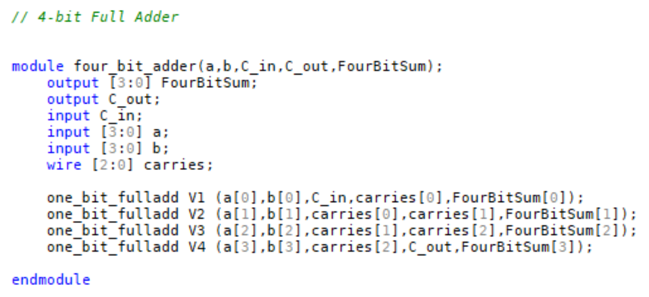
A) 1-Bit Full Adder**:** I designed the 1-bit full adder using basic gates and with delay given for each gate in a full code structurally, as the delay time for this design by theoretical calculation was 16 ns in simulation.



```
//Ayham Maree - 1191408 - Advanced Digital Final Project
`timescale 1ns/100ps

// 1- bit Full Adder
module one_bit_fulladd (a,b,C_in,C_out,Sum);
    output Sum;
    output C_out;
    input a;
    input b;
    input C_in;
    wire and1, and2, and3;


    and #(8ns)(and1, a, b);
    xor #(12ns)(Sum,C_in, a, b);
    and #(8ns) (and2, C_in,a );
    or  #(8ns)(C_out,and3 , and2,and1);
    and #(8ns) (and3,C_in,b);


endmodule
```

Figure 5: 1-bit full adder structural Verilog code

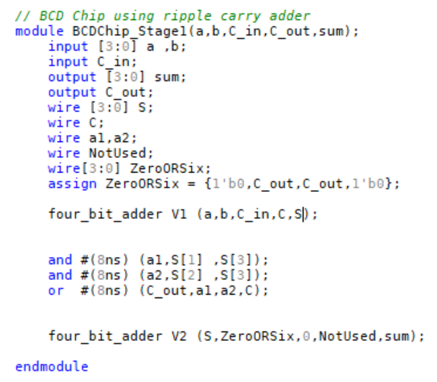
B) 4-bit Full Adder: I designed the 4-bit full adder using 4 pieces from 1-bit full adder in a full code structurally, as the delay time for this design by theoretical calculation was 64 ns and in simulation the same as in theoretical calculation



```
// 4-bit Full Adder

module four_bit_adder(a,b,C_in,C_out,FourBitSum);
    output [3:0] FourBitSum;
    output C_out;
    input C_in;
    input [3:0] a;
    input [3:0] b;
    wire [2:0] carries;

    one_bit_fulladd V1 (a[0],b[0],C_in,carries[0],FourBitSum[0]);
    one_bit_fulladd V2 (a[1],b[1],carries[0],carries[1],FourBitSum[1]);
    one_bit_fulladd V3 (a[2],b[2],carries[1],carries[2],FourBitSum[2]);
    one_bit_fulladd V4 (a[3],b[3],carries[2],C_out,FourBitSum[3]);

endmodule
```

Figure 6: 4-bit full adder using 1-bit full adder structural code

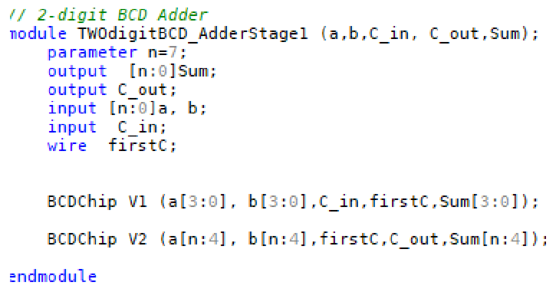
C) 1-digit BCD Adder: I designed the 1-digit BCD Adder using 2 pieces from 4-bit full adder in a full code structurally, as the delay time for this design by theoretical calculation was 140 ns and in simulation the same as in theoretical calculation, and the code as shown below



```
// BCD Chip using ripple carry adder
module BCDChip_Stage1(a,b,C_in,C_out,sum);
    input [3:0] a ,b;
    input C_in;
    output [3:0] sum;
    output C_out;
    wire [3:0] S;
    wire C;
    wire a1,a2;
    wire NotUsed;
    wire[3:0] ZeroORSix;
    assign ZeroORSix = {1'b0,C_out,C_out,1'b0};

    four_bit_adder V1 (a,b,C_in,C,S);


    and #(8ns) (a1,S[1] ,S[3]);
    and #(8ns) (a2,S[2] ,S[3]);
    or  #(8ns) (C_out,a1,a2,C);


    four_bit_adder V2 (S,ZeroORSix,0,NotUsed,sum);

endmodule
```

Figure 7:1-Digit BCD Adder using 4-bit full adder structural code

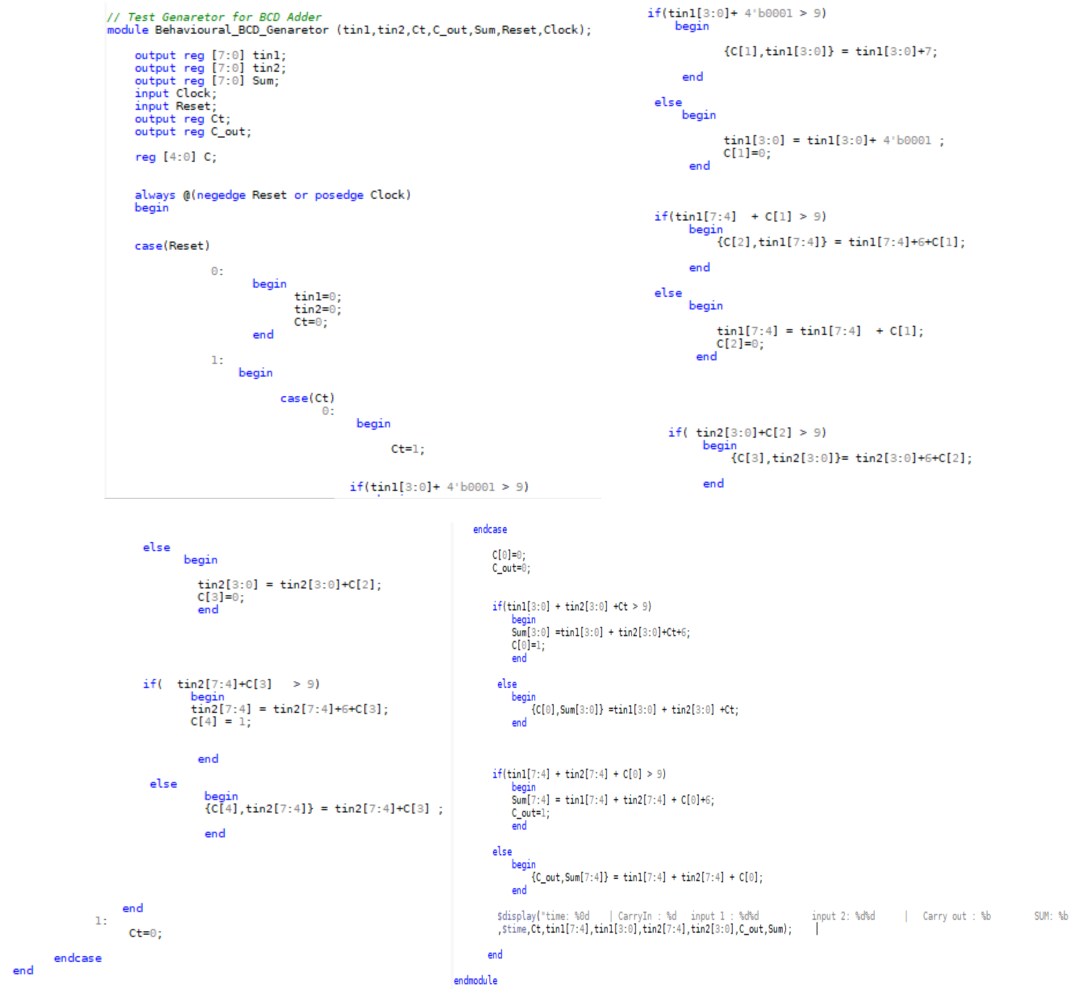
D) 2-digit BCD Adder: The 2-digit BCD Adder was designed using two pieces of 1-digit BCD Adder where the delay was calculated theoretically and was 280 ns and also when simulating it was completely equal to what was calculated theoretically



```
// 2-digit BCD Adder
module TWOdigitBCD_AdderStage1 (a,b,C_in, C_out,Sum);
    parameter n=7;
    output  [n:0]Sum;
    output C_out;
    input [n:0]a, b;
    input  C_in;
    wire  firstC;


    BCDChip V1 (a[3:0], b[3:0],C_in,firstC,Sum[3:0]);

    BCDChip V2 (a[n:4], b[n:4],firstC,C_out,Sum[n:4]);

endmodule
```

Figure 8: 2-digit BCD adder structural code

E) Test Generator: This piece was created to be used for design verification, and this piece works to give input to the piece that you created above and also gives output to the test analyzer where this piece must be built behaviorally and the mechanism and shape of the inputs and outputs and how the master piece works in order to work similarly so that the outputs from base piece as well in test generator, when it is on the positive edge, the output value is taken and changed , but when it is in the negative edge, the output values remain the same, and the delay time of the system is determined for the clock to work on the basis of it, since all cases are (0-99) when a carry 0 and carry 1 , beacues the adder has 8-bit input



```
// Test Genaretor for BCD Adder
module Behavioural_BCD_Genaretor (tin1,tin2,Ct,C_out,Sum,Reset,Clock);

    output reg [7:0] tin1;
    output reg [7:0] tin2;
    output reg [7:0] Sum;
    input Clock;
    input Reset;
    output reg Ct;
    output reg C_out;

    reg [4:0] C;


    always @(negedge Reset or posedge Clock)
    begin


    case(Reset)

                0:
                      begin
                            tin1=0;
                            tin2=0;
                            Ct=0;
                      end

                1:
                    begin

                          case(Ct)
                                0:
                                     begin

                                          Ct=1;


                                    if(tin1[3:0]+ 4'b0001 > 9)
```

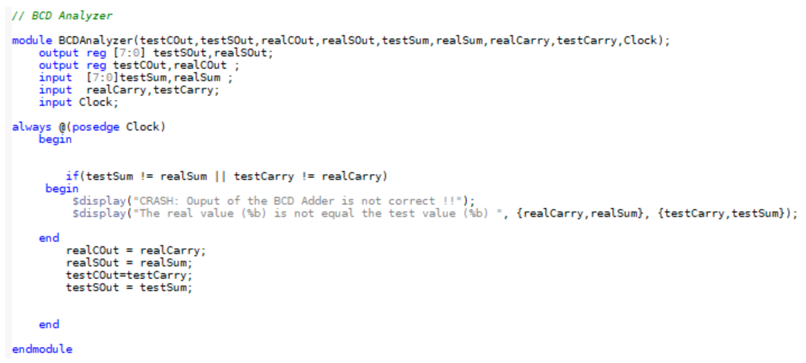
```
if(tin1[3:0]+ 4'b0001 > 9)
    begin

          {C[1],tin1[3:0]} = tin1[3:0]+7;

     end

 else
     begin

          tin1[3:0] = tin1[3:0]+ 4'b0001 ;
          C[1]=0;
      end


 if(tin1[7:4]  + C[1] > 9)
      begin
          {C[2],tin1[7:4]} = tin1[7:4]+6+C[1];

      end

 else
      begin

          tin1[7:4] = tin1[7:4]  + C[1];
          C[2]=0;
       end



   if( tin2[3:0]+C[2] > 9)
        begin
            {C[3],tin2[3:0]}= tin2[3:0]+6+C[2];

        end
```

```
         else
               begin

                 tin2[3:0] = tin2[3:0]+C[2];
                 C[3]=0;
                 end




         if(  tin2[7:4]+C[3]   > 9)
                begin
                tin2[7:4] = tin2[7:4]+6+C[3];
                C[4] = 1;


                 end

          else
                  begin
                  {C[4],tin2[7:4]} = tin2[7:4]+C[3] ;

                  end




            end
        1:
             Ct=0;

      endcase
end
```

```
   endcase

      C[0]=0;
      C_out=0;


      if(tin1[3:0] + tin2[3:0] +Ct > 9)
          begin
          Sum[3:0] =tin1[3:0] + tin2[3:0]+Ct+6;
          C[0]=1;
          end

       else
          begin
              {C[0],Sum[3:0]} =tin1[3:0] + tin2[3:0] +Ct;
          end



      if(tin1[7:4] + tin2[7:4] + C[0] > 9)
          begin
          Sum[7:4] = tin1[7:4] + tin2[7:4] + C[0]+6;
          C_out=1;
          end

      else
          begin
              {C_out,Sum[7:4]} = tin1[7:4] + tin2[7:4] + C[0];
          end

       $display("time: %0d    | CarryIn : %d   input 1 : %d%d          input 2: %d%d      |   Carry out : %b         SUM: %b
       ,$time,Ct,tin1[7:4],tin1[3:0],tin2[7:4],tin2[3:0],C_out,Sum);     |

     end

endmodule
```

Figure 9: Test Generator Behavioral Code

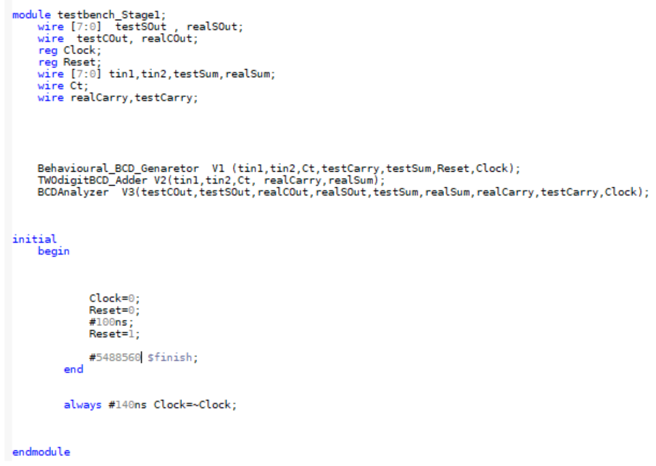
F) Test Analyzer: This piece is designed to compare between the values coming out of test generator and the values coming out of 2-digit BCD Addet where if the values are equal, they will work normally in the system, but if they are not equal, a message should appear showing the user that there is a problem with the outgoing values and they are not equal Where it works by the clock, when it is on the positive edge, the output value is taken and changed, but when it is in the negative edge, the output values remain the same, and the delay time of the system is determined for the clock to work on the basis of it, since all cases are (0-99) when a carry 0 and carry 1 , beacues the adder has 8-bit input



```verilog
// BCD Analyzer

module BCDAnalyzer(testCOut,testSOut,realCOut,realSOut,testSum,realSum,realCarry,testCarry,Clock);
    output reg [7:0] testSOut,realSOut;
    output reg testCOut,realCOut ;
    input  [7:0]testSum,realSum ;
    input  realCarry,testCarry;
    input Clock;

always @(posedge Clock)
    begin


        if(testSum != realSum || testCarry != realCarry)
     begin
         $display("CRASH: Ouput of the BCD Adder is not correct !!");
         $display("The real value (%b) is not equal the test value (%b) ", {realCarry,realSum}, {testCarry,testSum});

    end
        realCOut = realCarry;
        realSOut = realSum;
        testCOut=testCarry;
        testSOut = testSum;


    end

endmodule
```

Figure 10: Test Analyzer Code

G) Test Bench: This piece is designed to take the values for inputs and outputs for each piece in the system as it is called an instance of each piece and works to change its properties to complete the process of this system in general to cover all possible cases, for example changing the value of the clock every certain period and changing the value of the inputs every certain period and it constitutes the complete square of this system , to check whether it works successfully or not, and the mechanism of the process can be ascertained by placing a few cases, and the value of clock cycle will be change every half period of the 2-digit BCD adder 280/2 = 140 ns and then you can show all the possible results where each of the Test Generator , Test Analyzer , 2-Digit BCD Adder, and the value of finish time will be the full delay time * 99 * 99 * 2 = 280 * 99*99*2 = 5488560 ns.

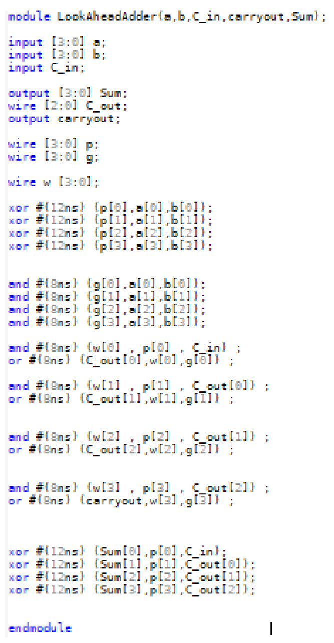


```verilog
module testbench_Stage1;
    wire [7:0]  testSOut , realSOut;
    wire  testCOut, realCOut;
    reg Clock;
    reg Reset;
    wire [7:0] tin1,tin2,testSum,realSum;
    wire Ct;
    wire realCarry,testCarry;




    Behavioural_BCD_Genaretor  V1 (tin1,tin2,Ct,testCarry,testSum,Reset,Clock);
    TWOdigitBCD_Adder V2(tin1,tin2,Ct, realCarry,realSum);
    BCDAnalyzer  V3(testCOut,testSOut,realCOut,realSOut,testSum,realSum,realCarry,testCarry,Clock);


initial
    begin



            Clock=0;
            Reset=0;
            #100ns;
            Reset=1;

            #5488560 $finish;
        end


        always #140ns Clock=~Clock;


endmodule
```

Figure 11: test bench code stage 1

2- Stage 2: Using Carry Look Ahead Adder:

A) Carry Look Ahead Adder: I designed the Look Ahead adder using basic gates and with delay given for each gate in a full code structurally, as the delay time for this design by theoretical calculation was 28 ns in simulation, and this used instead of 4-bit full adder in stage 1, and its own delay is less than the delay in the 4-bit adder (Carry Ripple Adder) and its perform to get the results from the simulation quickly without any problem.

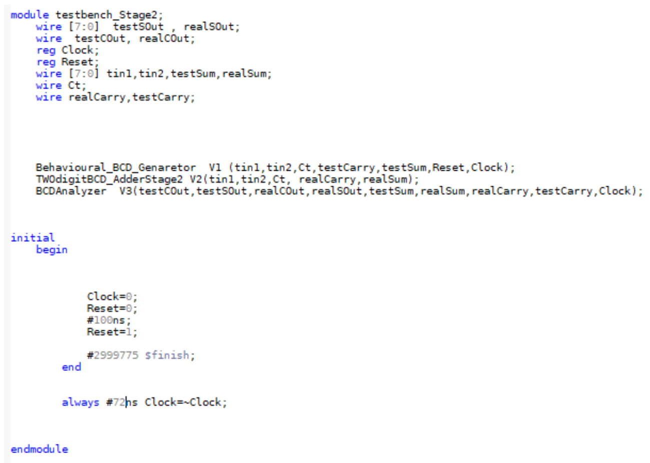


```
module LookAheadAdder(a,b,C_in,carryout,Sum);

input [3:0] a;
input [3:0] b;
input C_in;

output [3:0] Sum;
wire [2:0] C_out;
output carryout;

wire [3:0] p;
wire [3:0] g;

wire w [3:0];

xor #(12ns) (p[0],a[0],b[0]);
xor #(12ns) (p[1],a[1],b[1]);
xor #(12ns) (p[2],a[2],b[2]);
xor #(12ns) (p[3],a[3],b[3]);


and #(8ns) (g[0],a[0],b[0]);
and #(8ns) (g[1],a[1],b[1]);
and #(8ns) (g[2],a[2],b[2]);
and #(8ns) (g[3],a[3],b[3]);

and #(8ns) (w[0] , p[0] , C_in) ;
or #(8ns) (C_out[0],w[0],g[0]) ;

and #(8ns) (w[1] , p[1] , C_out[0]) ;
or #(8ns) (C_out[1],w[1],g[1]) ;


and #(8ns) (w[2] , p[2] , C_out[1]) ;
or #(8ns) (C_out[2],w[2],g[2]) ;


and #(8ns) (w[3] , p[3] , C_out[2]) ;
or #(8ns) (carryout,w[3],g[3]) ;



xor #(12ns) (Sum[0],p[0],C_in);
xor #(12ns) (Sum[1],p[1],C_out[0]);
xor #(12ns) (Sum[2],p[2],C_out[1]);
xor #(12ns) (Sum[3],p[3],C_out[2]);


endmodule
```

Figure 12: Look Ahead Adder structural code

Note : The 1-Digit BCD Adder and 2-Digit BCD Adder and Test Generator and Test Analyzer are the same in stage 1 (Carry Ripple Adder) but the instance was Carry Look Ahead Adder instead of 1-bit and 4-bit full adder.

B) Test Bench: This piece is designed to take the values for inputs and output for each piece in the system as it is called an instance of each piece and works to change its properties to complete the process of this system in general to cover all possible cases, for example changing the value of the clock every certain period and changing the value of the inputs every certain period and give the inputs for zero values because it's possible to see a problem when the value is not zero firstly it constitutes the complete square of this system , to check whether it works successfully or not, and the mechanism of the process can be ascertained by placing a few cases and the value of clock cycle will be change every half period of the 2-digit BCD adder 144/2 = 72 ns ,and then you can show all the possible results where each of the Test Generator , Test Analyzer , 2-Digit BCD Adder, and the value of finish time will be the full delay time * 99 * 99 * 2 = 72 * 99*99*2 = 2999775ns



```
module testbench_Stage2;
    wire [7:0]  testSOut , realSOut;
    wire  testCOut, realCOut;
    reg Clock;
    reg Reset;
    wire [7:0] tin1,tin2,testSum,realSum;
    wire Ct;
    wire realCarry,testCarry;



    Behavioural_BCD_Genaretor  V1 (tin1,tin2,Ct,testCarry,testSum,Reset,Clock);
    TWOdigitBCD_AdderStage2 V2(tin1,tin2,Ct, realCarry,realSum);
    BCDAnalyzer  V3(testCOut,testSOut,realCOut,realSOut,testSum,realSum,realCarry,testCarry,Clock);


initial
    begin


            Clock=0;
            Reset=0;
            #100ns;
            Reset=1;

            #2999775 $finish;
        end


        always #72ns Clock=~Clock;


endmodule
```

Figure 13: test bench code stage 2

Simulations & Results:

1- Stage 1:

These are some simulation pictures that demonstrate the outcomes of my finished code for stage 1. We can deduct from stage 1 that for 2-digit BCD adder utilizing 4-bit adder (Carry Ripple Adder). Furthermore, we must utilize clock as a register to constantly altering the value of the inputs when creating the test generator and analyzer.

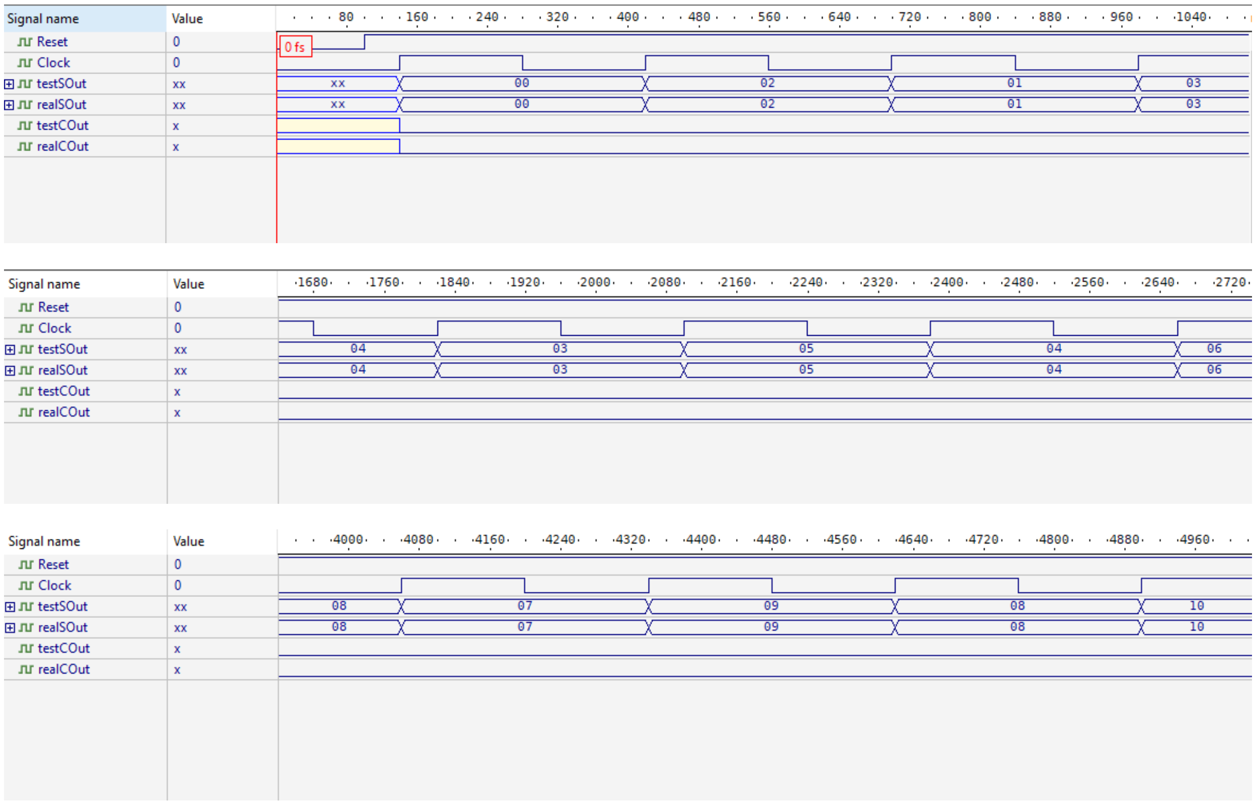


Figure 14: Stage 1 Simulation

2- 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 2-Digit BCD Adder built used Carry Look Ahead Adder.

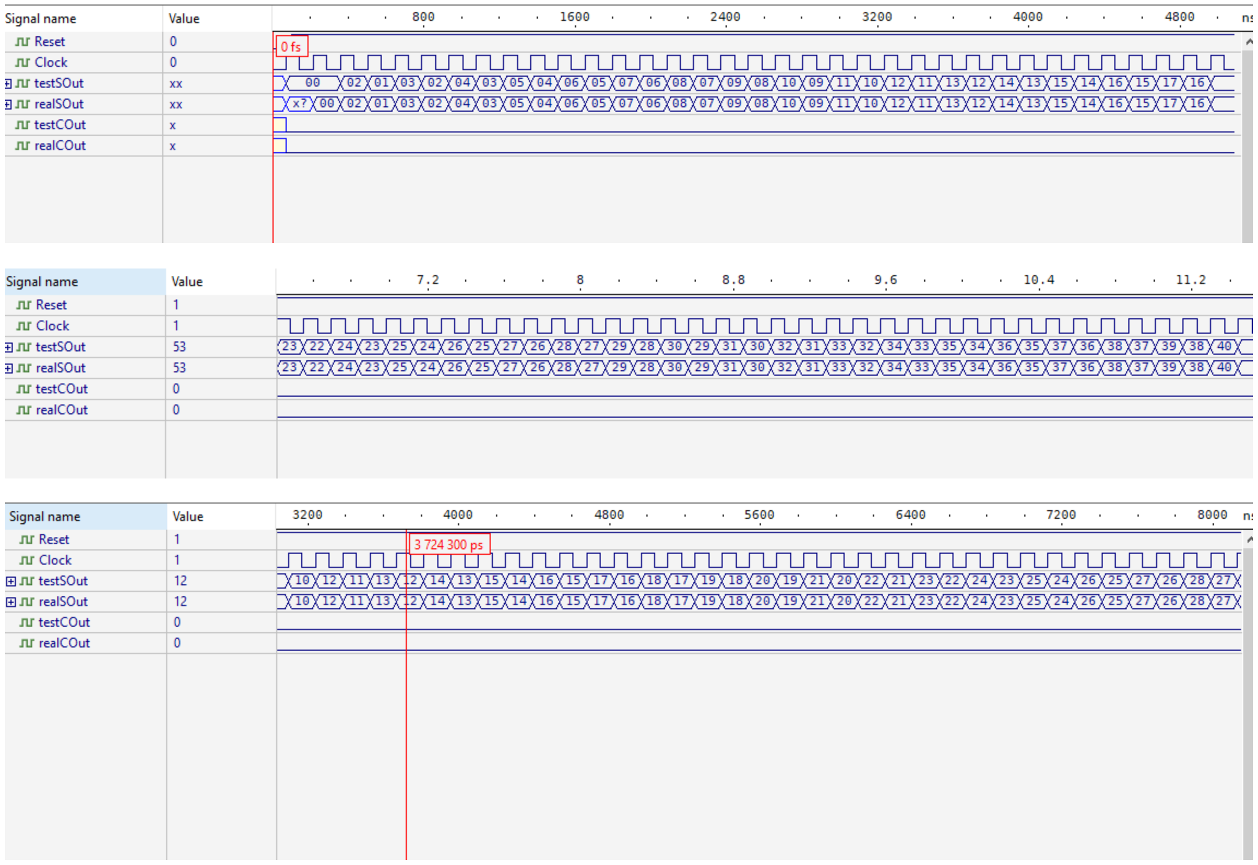


Figure 15: Stage 2 Simulation

In stage 1:

Maximum frequency = 1/Maximum delay = 1/280ns = 3571 kHz

Maximum Latency = 280 ns

In stage 2:

Maximum frequency = 1/Maximum delay = 1/144ns = 6944 kHz

Maximum Latency = 144 ns

Conclusion:

Finally, I am really happy of my effort and the findings I obtained; yet, the outcomes I obtained are consistent with the theoretical. Furthermore, this project and its stages were quite beneficial to me since I learnt a lot of new things, such as how to utilize the Verilog HDL program better. Furthermore, I gained a better understanding of Verilog HDL and how to create commands such as printing an error, testing systems, and creating entities in behavioral and structural logics.

I have successfully built 2-Digit BCD Adder using Ripple and Look Ahead methods. I learned a lot about plugin types and how to create them from the complete Adder and I also noticed the difference between both phases they even give the same result. In my opinion, I think that the second stage is faster and better than the first because of what was mentioned about the look ahead adder that it has a lower propagation delay than the ripple adder, and this improves its speed and effectiveness and gives results faster and more accurately.

I had some problems building some parts, but the adventure to find a solution to the problem was very interesting as I was able to complete the code correctly and completely and cover the required as I noticed the difference in aspects between each system when it was fully built, and there were some glitches but I was able to come up with a way to get rid of This problem by correctly calculating the delay and frequency of each system completely.

The project and its stages will aid me in future work since, as we all know, computers only deal with binary integers (one or zero). As a result, I gained a better understanding of the computer brain and how it performs mathematical operations. Finally, this project, as well as the assembly and Active HDL before it, will lead me to create a whole calculator utilizing Verilog code, which is exactly what I want to do.

During my work in the project, I faced the problem of how to get an out bot and one bot from the same module, and this is what made it difficult for me to finish working on the project in time and increase the pressure. I think that we should intensify learning about these languages and how to deal with them in order to get an excellent experience to go to the labor market in the most appropriate way As this project was for me one of the most important projects, and other than that, it was flexible and easy, and I hope to get similar projects for it in the future due to its flexibility and ease.