BIRZEIT UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ADVANCED DIGITAL DESIGN ENCS3310

COURSE PROJECT

Prepared By : Zakiya AbuMurra

ID : 1191636

Instructor : D. Abdallatif Abuissa

Section : 1

BIRZEIT

June 12, 2022

Introduction

The aim of this project to implement two digit BCD adder structural in two ways. One way is a ripple adder that a present stage wait carry out of previous stage, another ones is a Carry look a head that no dependently in carry out . Gradually , we built one bit full adder , four bit full adder , one digit BCD adder dawn to two digit BCD adder .

BCD adder is add two in input in binary number and check if the output greater than 9 or not to add 6 .In other word . to convert the result of operation to decimal that can people more familiar with the system ,

Then , the design put under the verification to confirm the results .

Contents

Table of figures :

1. Background

1.1 Binary Adder

A binary adder is a digital circuit that performs the arithmetic addition of two binary values of arbitrary length.[1]

1.1.1 Ripple Adder

The ripple adder is a combinational circuit which add two n-bit binary number by using n-full adder block . this type is a parallel circuit because the adder block depend to the previous carry that cause a long propagation delay . This carry propagation causes a considerable time delay when n is large. [2]

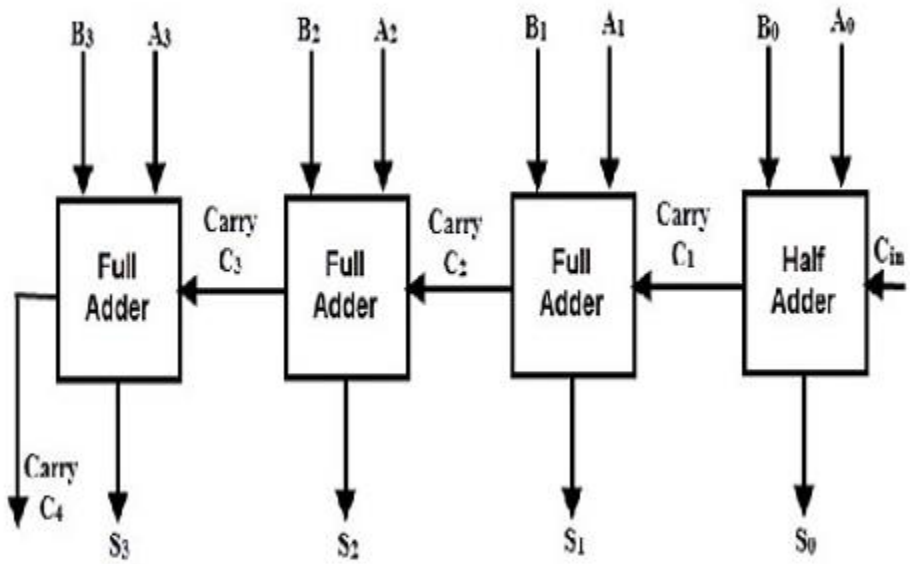


Figure 1 : 4-bit Ripple adder

1.1.2 Look Ahead Adder

we use this type of adder to reduce the propagation delay that occur during the addition operation by use the complex hardware circuitry .The carry output at any stage is dependent only on the initial carry bit of the beginning stage

From the truth table of the look-ahead adder , the Boolean expressions are :

Using the Gi and Pi terms the Sum Si and Carry Ci+1 are given as below where Gi ($Ai \oplus B$) is a Carry generate and Pi ($Ai \,.\, B$) is a carry propagate .

$$Si = Pi \oplus Gi.$$

$$Ci+1 = Ci.Pi + Gi.$$

Then ,the Carry Out for each full adder block is calculated as :

- C1 = C0.P0+G0.
- C2 = C1.P1+G1 = (C0.P0+G0).P1+G1.
- C3 = C2.P2+G2 = (C1.P1+G1).P2+G2.
- C4 = C3.P3+G3 = C0.P0.P1.P2.P3 + P3.P2.P1.G0 + P3.P2.G1 + G2.P3 + G3.[3]

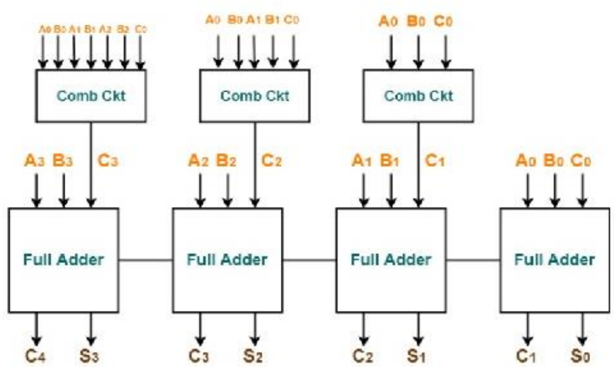


Figure 2 : 4-bit-Carry-Look-ahead-Adder-Logic-Diagram

1.2 BCD Adder

Although the binary number system is the most natural for a computer since it is easily represented in today's electronic technology, the decimal system is more familiar to most people. Converting decimal numbers to binary, performing all arithmetic calculations in binary, and then converting the binary outputs back to decimal is one technique to address this difference. This method demands the storage of decimal numbers in the computer in order to convert them to binary. Because the computer can only handle binary numbers, we must represent decimal digits using a code consisting of 1s and 0s. When decimal numbers are stored in a computer in coded form, it is also possible to perform arithmetic operations directly on them. [4]

the addition of two decimal digits in BCD, as well as the possibility of a carry from a prior pair of digits that were less significant. Because each digit cannot exceed 9, the total cannot exceed 9 + 9 + 1 = 19, with the 1 representing a previous carry

For example , if we want add (184 + 576) the result equal to 760 , we can do it in BCD adder as bellow :

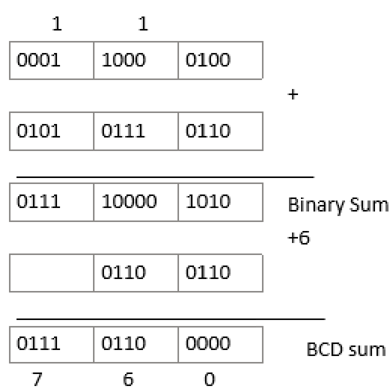


Figure 3: BCD summation

Note that if the result of Binary summation greater than 9 , then add $(0110)_2$ to get in BCD sum .

2 Design

2.1 Stage One using Ripple Adder

2.1.1 One bit Full adder

In this part , we built the one bit full adder from AND , XOR and OR gates as shown in figure ( ) There is three inputs (X , Y and Cin) and two output (s , Cout) where s is a summation result and Cout the Carry out from operation . We connect the gates together using wire that it wired output of gate to input of another gate. Each gate has a delay time, for two AND gates and one OR gate need respectively 8ns, 8ns, 8ns to get the carry out (Cout) result. Also, two OR gates need 24ns where each OR gate has a 12 ns time delay. Hence, all time delay for one bit full adder is 24ns. Where this value use later in Four bit Ripple adder .

Go to appendix No (1) to see code.

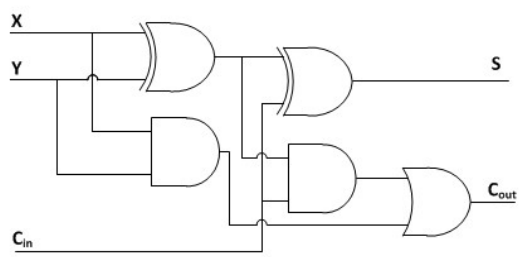


Figure 4 : One Bit Full adder design

2.1.2 Four bit Ripple adder

In this part , we have 4 one-bit full adder blocks. Each blocks has 3-bit Xi , Yi , Ci inputs and 2-bit Si , Coi output , then Call the module One bit full adder four times and connected together by wires . As we mentioned below , one block has 24ns time delay to get the correct answer so we add this value when call the blocks in code

You can see code in appendix No(2)

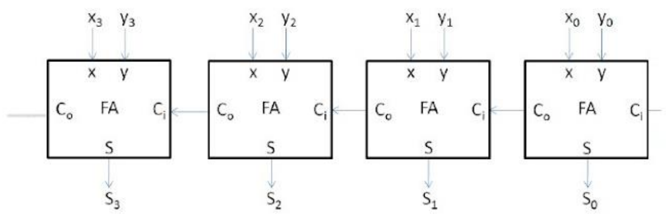


Figure 5 : 4-bit Ripple Adder desgin

2.1.3 1-Digit BCD adder using Ripple adder

In this part , we have a two input , each of them has a four bit , then these inputs enter 4-bit Ripple adder block, the output of this block enter as a input of anther 4-bit ripple adder block to convert the result form binary sum to BCD sum . Also, we can note the Combinational circuit content the two AND gates and OR gate which check if the binary sum in BCD sum form or not by add a (0, twice of carry out form combinational circuit) . In case the check is not the circuit will add (0110) to the binary sum.

You can see the code in appendix No (3).

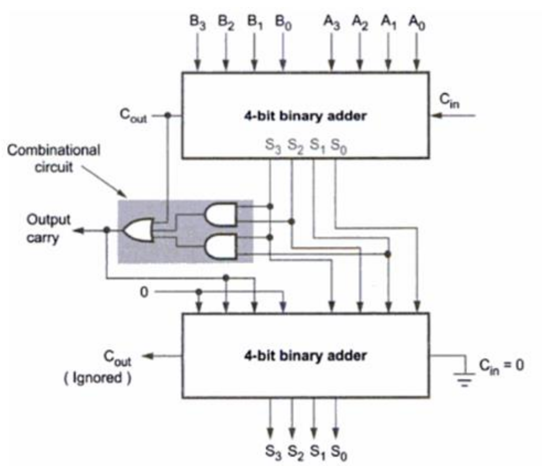


Figure 6 : 1-digit BCD using Ripple adder design

2.1.4 2-Digit BCD adder using Ripple adder

In this part, we built the two digit BCD adder using Ripple adder. Where have two inputs each of them has 8-bit. However, the input concatenate to be one input then enters to the register that implemented as a D-flip flop. As we know, the register has one input, clock and rest and output which aim of register is store the data.

In the code we implement one module for registers and we use the parameter to allow use it more than one in different number of bits.

You can see the code in appendix No (4).

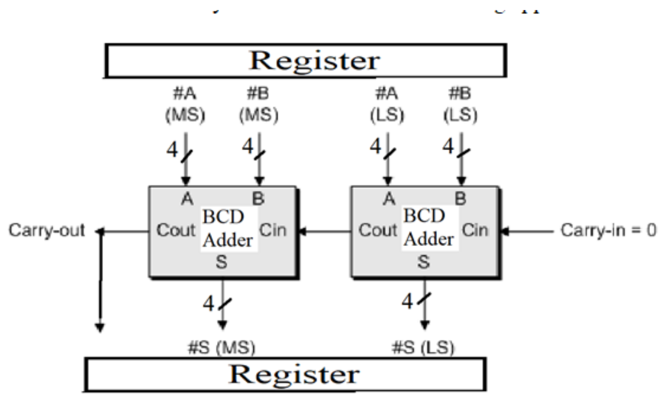


Figure 7 : 2-digit BCD using Ripple adder design

2.2 Stage two using carry look a head

2.2.1 full adder for carry look ahead

The carry look ahead adder was built using modified the full-adder. They both provide the same inputs . The Cout output in full adder has been replaced with carry propagate and generate in carry look ahead since the carry-out output isn't necessary, but the carry propagate and generate outputs are

The carry-in determines overall production, although P and G are created separately. . So , the propagation delay was reduced .

You can see the code in appendix No(7).

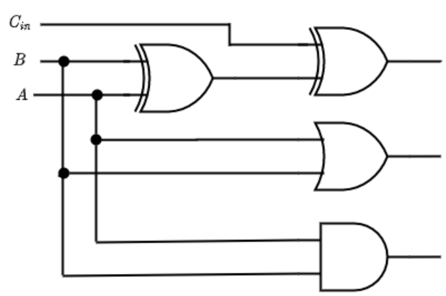


Figure 8 : full adder for carry look a head design

2.2.2 4-bit Carry look ahead adder

In this part , because of complexity for the circuit gate level , we implemented the summation , carry out , carry generate and carry propagate using gates based on the Boolean expressions that derived from truth table that mentioned in 1.1.2 . In addition . the total delay for all system using carry look ahead adder is 128ns , that reduce the propagation delay when use ripple adder .

You can see the code in appendix no(8).

2.2.3 1-digit BCD using carry look ahead :

This part explain same as 2.1.4 , but the different in implementation use a carry look ahead adder in binary adder .

You can see the code in appendix no (9)

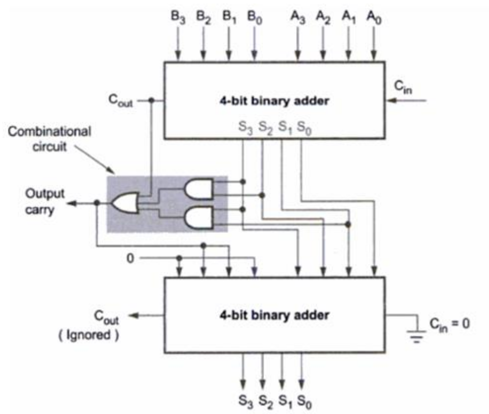


Figure 9 : 1-digit BCD using carry look ahead adder design

2.2.4 2-digit BCD using carry look ahead adder

All performance for the system will improved because the carry look ahead adder reduce the propagation delay to 128ns .

You can see the code in appendix no(10)

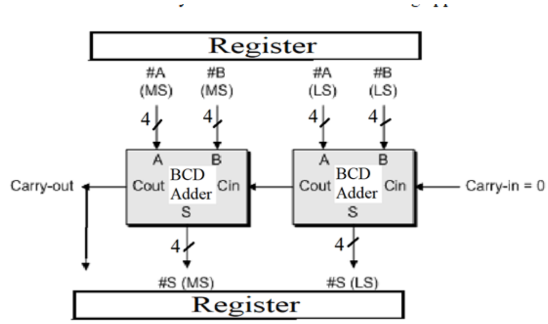


Figure 10: 2-digit BCD using carry look ahead adder design

2.3 Design Verification

In Design Verification , test inputs were generated on rising edges, and the predicted outputs were built behaviorally in the test generator to verify the design. On every clock cycle, the circuit outputs are compared to the predicted ones for a result analyzer, and if there is a mismatch, a warning message is presented. A test bench is used to give a clock the appropriate delay for system latency, as well as inputs generated, which are then passed to the circuit, and outputs are then passed to the result analyzer, from which both behavior and structure are derived.

2.3.1 Test generator

In this part , the code built behavior that has inputs and outputs . we add 6 if the result from or and AND operation equals to 1 . you can see all detail in appendix no (11)

2.3.2 Analyzer

In this part, correct values are compared with output values, if they are not equal then an error is asserted, and an error message is printed.
You can see code in appendix no(12).

2.3.3 Test bench

All of the components are connected here by creating test inputs, passing them to the comparator circuit, and comparing the results to the expected ones to determine whether there is an error.

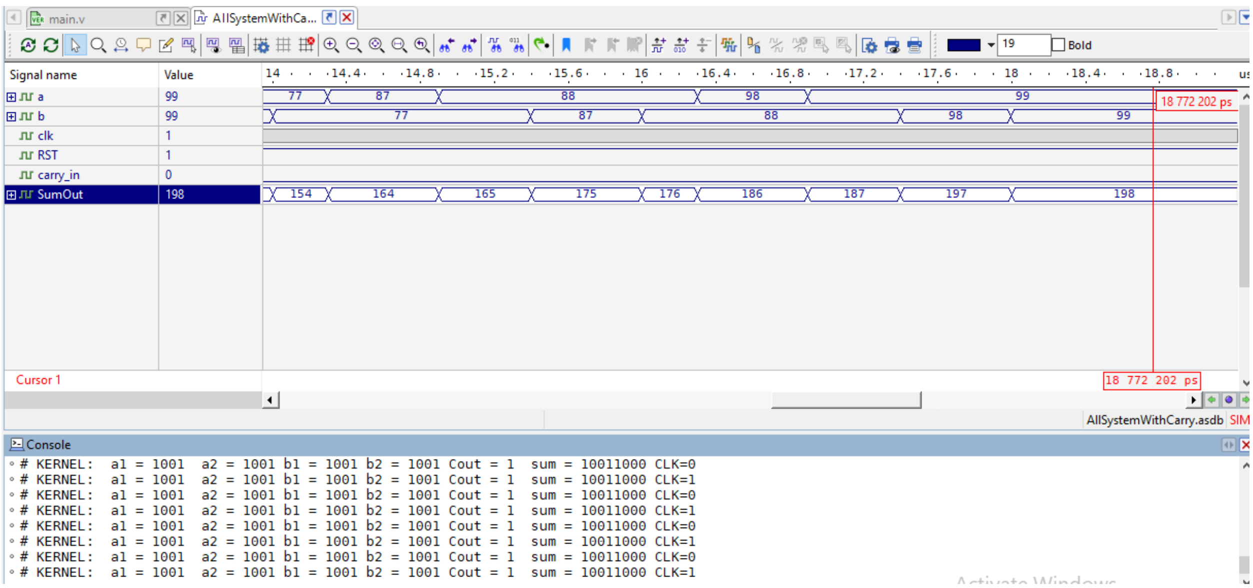
You can see test bench for stage one using the ripple adder in appendix no(13).
And for stage two using the look ahead carry in appendix no(14).

3 Result

3.1 Stage One using Ripple adder

Using test bench code then simulate the module . taking the example , if a(input) = 99 in decimal that equal (1001 1001) in binary and b(input) = 99 in decimal that equal (1001 1001) in binary . the output equal to 198 in decimal that represent $(0001\ 1001\ 1000)_2$ where the carry out from the addition is (0001) . the results confirm what we explain in part 1.2 .

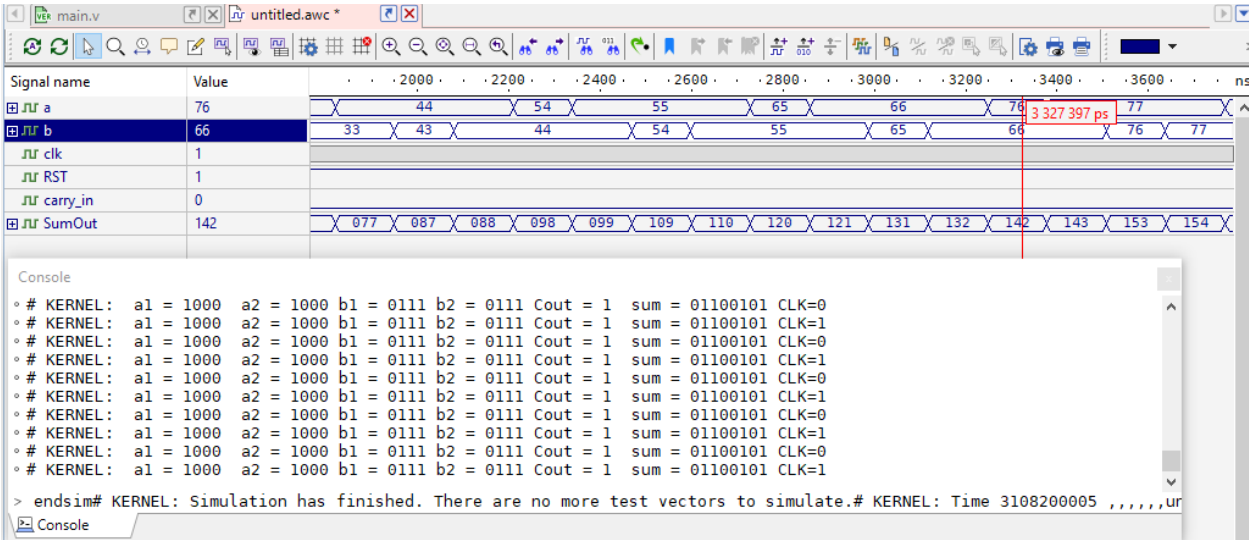
You can see the test bench code in appendix No(6).



3.2 Stage two using Carry look a head adder

Using test bench code then simulate the module . taking the example , if a(input) = 76 in decimal that equal (0111 0110) in binary and b(input) = 66 in decimal that equal (0110 0110) in binary . the output equal to 142 in decimal that represent $(0001\ 0100\ 0010)_2$ where the carry out from the addition is (0001) . the results confirm what we explain in part 1.2 .

You can see the test bench code in appendix no(15) :



3.2 Verification testing

3.2.1 Testing without error for stage one :

We note all is correct .

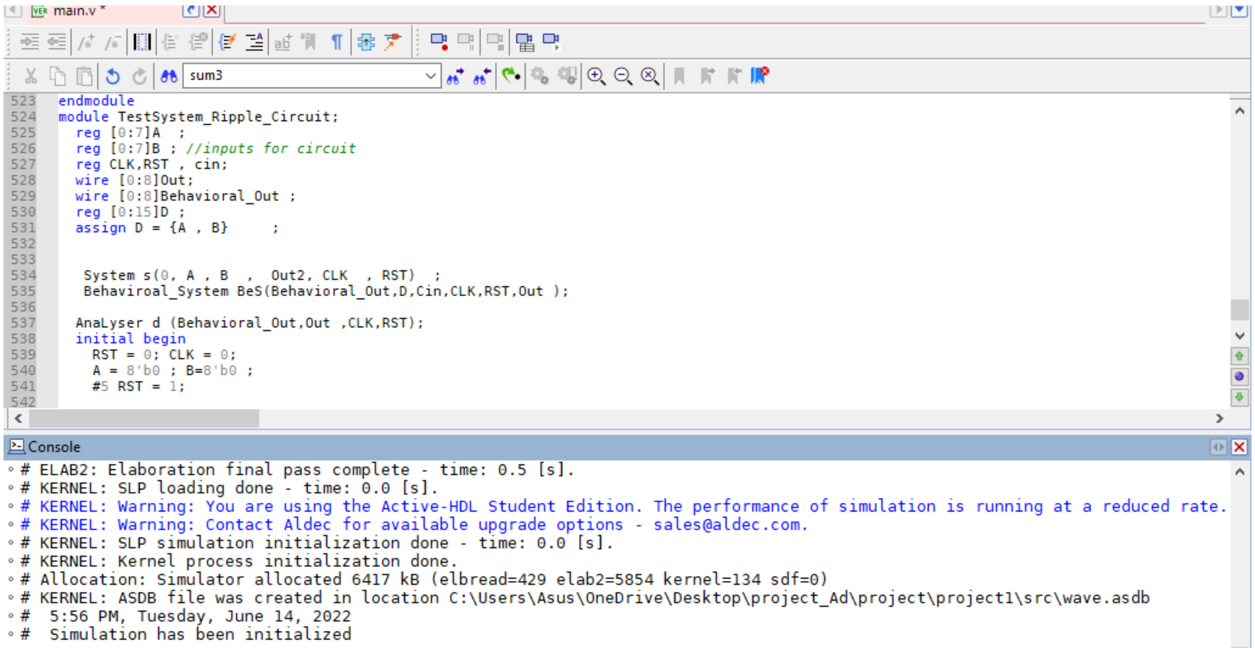


Figure 11 : design verification for ripple BCD adder without error

3.2.2 Testing with error for stage one :

If we change any gates from code that caused the result is error . the analyzer will be print error as shown in figure (12).

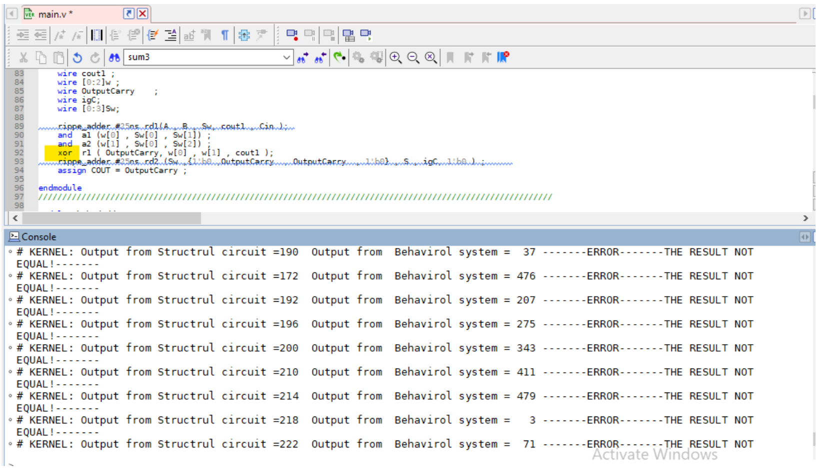


Figure 12: design verification for ripple BCD adder with error

3.2.3 Testing without error for stage two :

No massages error as shown below in figure 13.

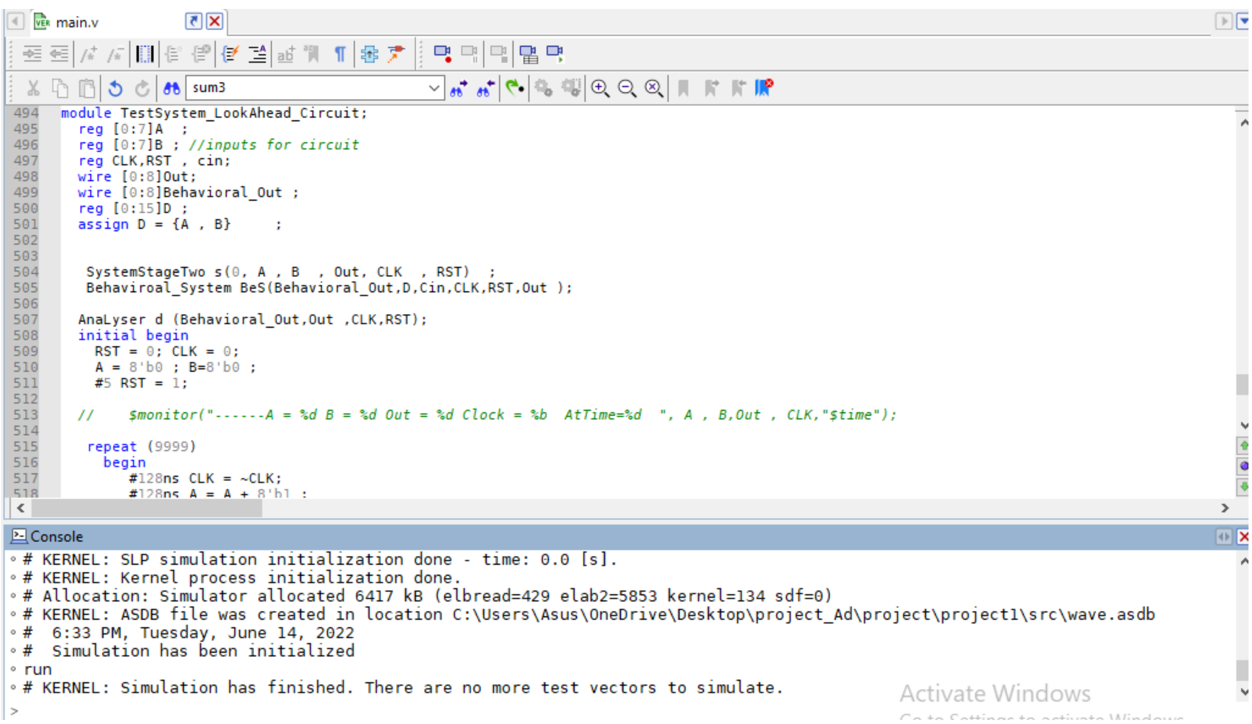


Figure 13 : design verification for Carry look ahead BCD adder without error

3.2.4 Testing with error

If we change any gates from code that caused the result is error . the analyzer will be print error as shown in figure (14).

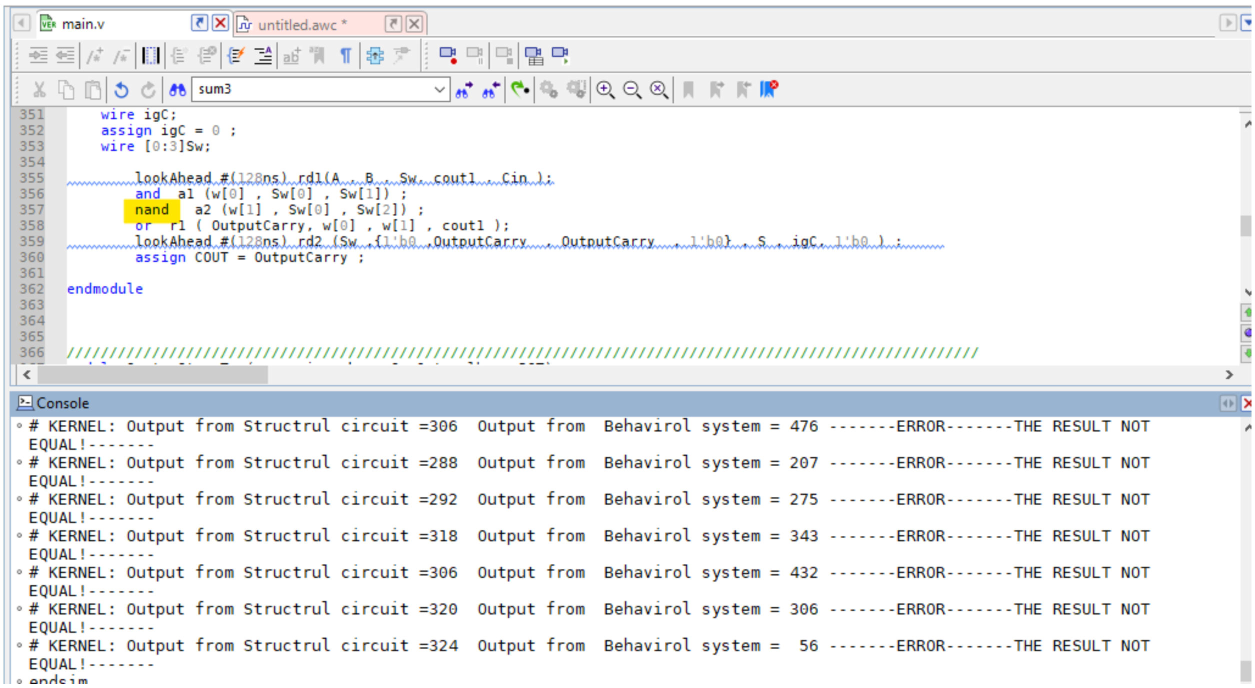


Figure 14 : design verification for Carry look ahead BCD adder with error

3 Conclusion and future Work

In Our project , the Two digit BCD adder was built in two ways . Using ripple adder and carry ahead adder . In each way we implement the design structural using gate level .Which takes me to be able handle the correct delay time for each circuit. In addition , if I had to choose only one approach for constructing the BCD ADDER circuit, I would choose the carry look ahead adder because it gives the results quickly .

The future work can be resolve several problem as redundant the code . for example , the full adder can using in ripple and carry ahead if the architecture for it have the mux to choose the specific gates .

This project served as a springboard for us to enter the domains of hardware design and verification. I had a great time working on it, and I'd like to do more research and stay up with new developments. Because I believe that 'improvements, not excuses,' is the key.

4 References

1. https://www.tutorialspoint.com/what-is-binary-adder [accessed 13 June 2022 at 6:00pm]
2. https://www.gatevidyalay.com/ripple-carry-adder/ [accessed June 13, 2022 at 6:15pm]
3. https://www.elprocus.com/carry-look-ahead-adder/ [accessed June 13, 2022 at 5:00pm]
4. https://drive.google.com/file/d/115Otrw7LO2JhUWrH4GYEqvk_MBqQTYb_/view[accessed June 13, 2022 at 10:00pm]

5 Appendix

5.2 One bit adder :

```
//////////////////////////////////////////////////////////////////////
module fulladder_oneBit(X, Y, Ci, SUM, COUT);

  input X, Y, Ci;
  output SUM, COUT;
  wire w1,w2,w3;

  xor (w1, X, Y);
  xor (SUM, w1, Ci);
  and (w2, w1, Ci);
  and (w3, X, Y);
  or  (COUT, w2, w3);

endmodule

//////////////////////////////////////////////////////////////////////////////
```

5.3 four 1-bit ripple adder :

```
//////////////////////////////////////////////////////////////////////
module rippe_adder(X, Y, S, Co , Cin);
 input [0:3] X;
 input [0:3]Y;// Two 4-bit inputs
 input Cin ;
 output [0:3] S;  //sum
 output Co;      // carry out
 wire w1, w2, w3;
 //calling 4 1-bit full adders to design 4-bit ripple adder
         fulladder_oneBit #25ns u1(X[3], Y[3], Cin, S[3], w1);
         fulladder_oneBit #25ns u2(X[2], Y[2], w1, S[2], w2);
         fulladder_oneBit #25ns u3(X[1], Y[1], w2, S[1], w3);
         fulladder_oneBit #25ns u4(X[0], Y[0], w3, S[0], Co);
endmodule
```

5.4 1-digit BCD adder :

```
////////////////////////////////////////////////////////////////////////////////////////////////////

module BCD (A , B ,Cin ,  COUT , S ) ;
    input [0:3]A ;
    input [0:3]B ;
    input Cin ;
    output COUT  ;
    output [0:3]S ;
    wire cout1 ;
    wire [0:2]w ;
    wire OutputCarry    ;
    wire igC;
    wire [0:3]Sw;

    rippe_adder #25ns rd1(A , B , Sw, cout1 , Cin );
    and  a1 (w[0] , Sw[0] , Sw[1]) ;
    and  a2 (w[1] , Sw[0] , Sw[2]) ;
    or   r1 ( OutputCarry, w[0] , w[1] , cout1 );
    rippe_adder #25ns rd2 (Sw ,{1'b0 ,OutputCarry  , OutputCarry  , 1'b0} , S , igC, 1'b0 ) ;
    assign COUT = OutputCarry ;

endmodule
//////////////////////////////////////////////////////////////////////////////////////////////////////////
```

5.5 2-digit BCD adder using ripple adder :

```
module Reg2(D,clk,rest , Out2);
    parameter n = 1;

    input [n-1:0]D;
    input clk , rest ;
    output reg [n-1:0] Out2;

     always @ (posedge clk or negedge rest)
      if (!rest)
         Out2 <= 0;
      else
         Out2 <= D;

endmodule

/////////////////////////////////////////////////////

module System(carry_in,a,b ,  SumOut, clk  , RST);
    input carry_in;
    input [0:7] a;
    input [0:7] b;
    input clk   ,RST  ;
    output [0:8]SumOut ;
    wire  Carry_out;
    wire [0:15]q, R ;
    wire  [0:7]out;
    wire cout;
    wire [0:8]CarryWithout ;

    assign q ={a,b};

    Reg2 reg1(q,clk, RST,R);
    defparam reg1.n = 16;

    BCD b1(R[4:7], R[12:15] , 1'b0 ,  cout , out[4:7] ) ;
    BCD b2(R[0:3], R[8:11] ,cout ,  Carry_out , out[0:3] ) ;
    assign CarryWithout =   { Carry_out , out  };
    Reg2 #180ns reg2 ( CarryWithout , clk , RST, SumOut )  ;
    defparam reg2.n = 9;


endmodule
```

5.6 Test Bench for Stage One :

```
endmodule


module tb_system ;

    reg [0:7]a ;
    reg [0:7]b;
    reg clk ;
    reg RST ;
    reg carry_in ;

    wire [0:8]SumOut;

    System s (carry_in,a,b ,  SumOut, clk  , RST);

    always #600 clk = ~ clk ;

    initial
        begin
            clk =0 ;   RST =1 ; a =0 ; b =0 ;   carry_in=0 ;


            $monitor (" a1 = %b  a2 = %b b1 = %b b2 = %b Cout = %b  sum = %b CLK=%b" , a[0:3], a[4:7] , b[0:3] , b[4:7], SumOut[0],SumOut[1:8] , clk );
        repeat(9)
            begin

                 #300ns a[0:3] = a[0:3] + 1 ;
                 #600ns a[4:7] = a[4:7] + 1 ;
                 #500ns b[0:3] = b[0:3] + 1;
                 #600ns b[4:7] = b[4:7] + 1;
           end
    end
```


5.7 full adder for carry look ahead

```
module FaLook(a,b,cin,s,p,g);
    input a,b,cin;
    output  s,p,g;
    wire wxor;
    xor(wxor,a,b);
    or (p,a,b);
    and (g,a,b);
    xor(s,wxor,cin);
endmodule
```

5.8 4-bit Carry look ahead adder

```
////////////////////////////////////////////////////////////////////////////////////

module lookAhead(a,b,sum , cout , cin);
    input [0:3]a;
    input [0:3]b;
    input cin;
    output cout;
    output [0:3]sum;
    wire[0:21] wires;


    FaLook FA0 (a[3],b[3],cin,sum[3],wires[21],wires[20]);
    and(wires[19],cin,wires[21]);
    or(wires[18],wires[19],wires[20]);
    FaLook FA1(a[2],b[2],wires[18],sum[2],wires[17],wires[16]);
    and(wires[15],cin,wires[21],wires[17]);
    and(wires[14],wires[20],wires[17]);
    or(wires[13],wires[16],wires[14],wires[15]);
    FaLook FA2(a[1],b[1],wires[13],sum[1],wires[12],wires[11]);
    and(wires[10],cin,wires[21],wires[17],wires[12]);
    and(wires[9],wires[20],wires[17],wires[12]);
    and(wires[8], wires[16],wires[12]);
    or(wires[7],wires[11],wires[10],wires[9],wires[8]);
    FaLook FA3(a[0],b[0],wires[7],sum[0],wires[6],wires[5]);
    and (wires[4],cin,wires[21],wires[17],wires[12],wires[6]);
    and(wires[3],wires[20],wires[17],wires[12],wires[6]);
    and(wires[2],wires[16],wires[12],wires[6]);
    and(wires[1],wires[11],wires[6]);
    or(wires[0],wires[1],wires[2],wires[3],wires[4]);
    assign cout = wires[0];
endmodule
```

5.9 1-digit BCD using carry look ahead

```
///LOOK AHEAD BCD ADDER FOR STAGE TWO

module BCDStageTwo (A , B ,Cin ,  COUT , S ) ;
    input [0:3]A ;
    input [0:3]B ;
    input Cin ;
    output COUT  ;
    output [0:3]S ;
    wire cout1 ;
    wire [0:2]w ;
    wire OutputCarry    ;


    wire igC;
    assign igC = 0 ;
    wire [0:3]Sw;

        lookAhead #(128ns) rd1(A , B , Sw, cout1 , Cin );
        and  a1 (w[0] , Sw[0] , Sw[1]) ;
        and   a2 (w[1] , Sw[0] , Sw[2]) ;
        or  r1 ( OutputCarry, w[0] , w[1] , cout1 );
        lookAhead #(128ns) rd2 (Sw ,{1'b0 ,OutputCarry  , OutputCarry  , 1'b0} , S , igC, 1'b0 ) ;
        assign COUT = OutputCarry ;

endmodule
```

5.10 2-digit BCD using carry look ahead adder

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module SystemStageTwo(carry_in,a,b ,  SumOut, clk  , RST);
    input carry_in;
    input [0:7] a;
    input [0:7] b;
    input clk  ,RST  ;
    output [0:8]SumOut ;
    wire  Carry_out;
    wire [0:15]q ;
    wire [0:15]R ;
    wire  [0:7]out;
    wire cout;
    wire [0:8]CarryWithout ;

    assign q ={a,b};

    Reg2 reg1(q,clk, RST,R);
    defparam reg1.n = 16;

    BCDStageTwo b1(R[4:7], R[12:15] , 1'b0 ,  cout , out[4:7] ) ;
    BCDStageTwo b2(R[0:3], R[8:11] ,cout ,  Carry_out , out[0:3] ) ;
    assign CarryWithout =   { Carry_out , out  }    ;

    Reg2 #128ns reg21 ( CarryWithout , clk , RST, SumOut )  ;
    defparam reg1.n = 8 ;



endmodule
```

5.11 Test Generator

```
 Behaviroal_System(Out1,X,Cin,CLK,RST,Out2);

tput reg [0:8]Out1 ;
tput Out2 ;
put   [0:15] X ;
put Cin,CLK,RST ;
g [0:3]SumOut1,SumOut2,SumOut3,SumOut4;
g carryOut1,carryOut2,carryOut3,carryOut4,carryOut5;


ways @(posedge CLK or negedge RST)
  begin
      {carryOut1,SumOut1} = X[0:3]+X[4:7];
       carryOut2= (SumOut1[3] & SumOut1[2]) | (SumOut1[3] & SumOut1[1]) | carryOut1 ;
      if(carryOut2)
          begin

          SumOut2 = SumOut1+4'b0110 ;
          Out1[0:3] = SumOut2;

          end
      else
          Out1[0:3] = SumOut1;

      {carryOut3,SumOut3} = X[8:11]+X[12:15]+carryOut2;
      carryOut4 = (SumOut3[3] & SumOut3[2]) | (SumOut3[3] & SumOut3[1]) | carryOut3 ;
      if(carryOut4)
          begin

          SumOut4= SumOut3+4'b0110 ;
          Out1[4:7] = SumOut4;

          end
      else
          Out1[4:7] = SumOut3;
          Out1[8] = carryOut4;



  end
```

5.12 Analyzer

```
module AnaLyser(Behavioral_Out,Structural_Out,CLK,RST);
    input[0:8]Behavioral_Out,Structural_Out;
    input CLK,RST ;
    always @(posedge CLK or negedge RST)
        begin:beginAlways
            if(Behavioral_Out != Structural_Out)
                begin
                 $display("Output from Structrul circuit =%d  Output from  Behavirol system = %d " , Structural_Out,Behavioral_Out);
                 end
            end
endmodule
```

5.13 Test Bench for ripple system in Design verification

```
///////////////////////////////////////////////////////////////////////////////////////.
endmodule
module TestSystem_Ripple_Circuit;
  reg [0:7]A  ;
  reg [0:7]B ; //inputs for circuit
  reg CLK,RST , cin;
  wire [0:8]Out;
  wire [0:8]Behavioral_Out ;
  reg [0:15]D ;
  assign D = {A , B}     ;


   System s(0, A , B  ,  Out2, CLK  , RST)  ;
   Behaviroal_System BeS(Behavioral_Out,D,Cin,CLK,RST,Out );

  AnaLyser d (Behavioral_Out,Out ,CLK,RST);
  initial begin
    RST = 0; CLK = 0;
    A = 8'b0 ; B=8'b0 ;
    #5 RST = 1;

    $monitor("A = %d B = %d Out = %d Clock = %b  AtTime ", A , B,Out , CLK,$time);

   repeat (9999)
     begin
        #128ns CLK = ~CLK;
        #128ns A = A + 8'b1 ;
        #128ns B = B + 8'b1 ;
    end
  end

endmodule
```

5.14 Test Bench for look ahead carry system in Design verification

```
module TestSystem_LookAhead_Circuit;
  reg [0:7]A  ;
  reg [0:7]B ; //inputs for circuit
  reg CLK,RST , cin;
  wire [0:8]Out;
  wire [0:8]Behavioral_Out ;
  reg [0:15]D ;
  assign D = {A , B}     ;


   SystemStageTwo s(0, A , B  , Out, CLK  , RST)  ;
   Behaviroal_System BeS(Behavioral_Out,D,Cin,CLK,RST,Out );

  AnaLyser d (Behavioral_Out,Out ,CLK,RST);
  initial begin
    RST = 0; CLK = 0;
    A = 8'b0 ; B=8'b0 ;
    #5 RST = 1;

    $monitor("------A = %d B = %d Out = %d Clock = %b  AtTime=%d -------ERROR-------THE RESULT NOT EQUAL!------- ", A , B,Out , CLK,"$time");

   repeat (9999)
     begin
        #128ns CLK = ~CLK;
        #128ns A = A + 8'b1 ;
        #128ns B = B  + 8'b1 ;
    end
  end
/////////////////////////////////////////////////////////////////////////////////////////////
```

5.15 Test bench for Stage Two :

```
module tb_systemStageTwo ;

    reg [0:7]a ;
    reg [0:7]b;
    reg clk ;
    reg RST ;
    reg carry_in ;

    wire [0:8]SumOut;

    SystemStage s (carry_in,a,b ,  SumOut, clk  , RST);
    always #128 clk = ~ clk ;

    initial
        begin
            clk =0 ;   RST =1 ; a =0 ; b =0 ;
            carry_in=0 ;

            $monitor (" a1 = %b  a2 = %b b1 = %b b2 = %b Cout = %b  sum = %b CLK=%b" , a[0:3], a[4:7] , b[0:3] , b[4:7], SumOut[0],SumOut[1:8] , clk );
        //always #600 clk = ~ clk ;
        repeat(9) begin

           #128ns a[0:3] = a[0:3] + 1 ;
           #128ns a[4:7] = a[4:7] + 1 ;
           #240s b[0:3] = b[0:3] + 1;
            #240ns b[4:7] = b[4:7] + 1;
           end
    end

    endmodule
```