Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ADVANCED DIGITAL DESIGN ENCS533
COURSE PROJECT

Prepared by:

Name: Sojood Asfour

ID: 1230298

Instructor: Dr. Abdellatif Abu-Issa

Section: 2

Table of Contents

Table of figures:

Introduction and background:

In present digital systems, arithmetic operators like addition constitute a fundamental part of solving computational problems. BCD addition is largely applied in systems interfacing with human-readable numerical data; digital clocks, calculators, and financial applications are some common examples. Unlike pure binary arithmetic, BCD operations must preserve decimal digit boundaries, thus requiring special circuitry for carry propagation and decimal correction.

This project involved designing and implementing an n-digit BCD adder using two types of binary adders: ripple carry adder and carry look-ahead adder. The adders were structurally described using very few basic logic gates, each having its delay profile.

A BCD adder is a circuit which adds two binary-coded decimal (BCD) numbers, in which each digit is represented by a 4-bit binary value from 0000 to 1001. It contains, therefore, four full-adders in sequential arrangement, each admitting one bit to be added to the previous carry output. Should the output sum be greater than 9 (binary 1001), the circuit will add the binary number 0110 (6) in order to correct it, and the result will thus be a valid BCD number.

The project scope is divided into two major phases having increasing levels of complexity. Stage 1 consists of building a 3-digit BCD adder using ripple carry adders, whereby full functional verification and error detection are encompassed. Subsequently, in Stage 2, the ripple carry adder units are replaced by fast carry look-ahead adders to achieve higher performance, whereby the system is again assessed for its latency and maximum clock frequency.

Design Philosophy:

In completing this project, a modular structure of an n-digit BCD adder was built on primitive logic gates. The functionality embodied the hardware delays in relation to summing BCD digits, as time restrictions needed to be followed in conditions of boundary BCD digit addition. The first step in the design was to grasp BCD multi-digit addition and the constraints of accurately adding BCD bits.

The work breakdown structure is centered on the design of a 1-bit full adder composed of the basic gates AND, OR, XOR and their time delays. Those were merged to produce individual 4-bit adders that are used as components to sum up the BCD digits. Considering the need for performing an adjustment after the value of 9 is attained in BCD addition, a correction circuit was also included to apply the value six (binary 0110) where needed.

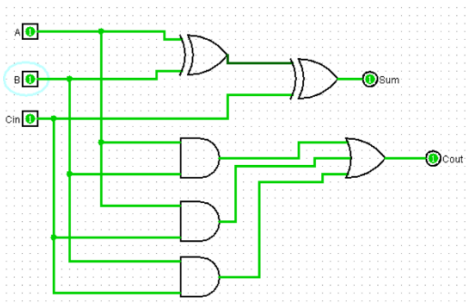


Figure 1: Full adder design using logisim

Ripple adder:

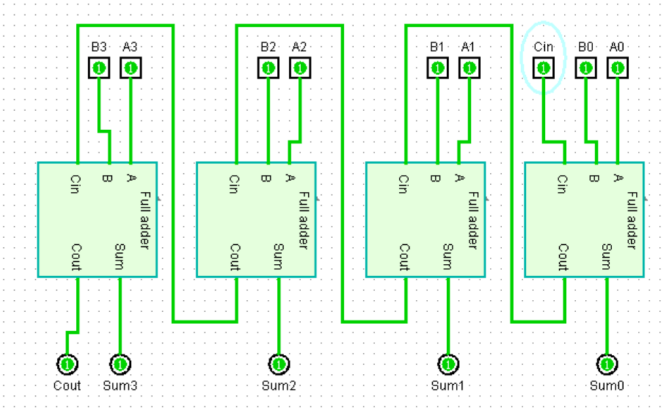


Figure 2: 4-bit Ripple adder design

The initial adder to get done with the job was the Ripple Carry Adder (RCA), It is important to state that the RCA structure allows the linking up of multiple 1-digit BCD adders in a continuous manner, with the carry-out from one digit going to the carry-in of the next.

The ripple carry design is quite simple and does not require many resources, yet the one drawback is that it suffers from a sequential delay which builds up, as the carry has to make its way to each and every digit. This delay is the limit of the clock frequency that the system can handle without having to deal with errors.

Figure 3 shows a circuit that uses two 4-bit adders to add two 1-digit BCD numbers. The lower adder takes in A3–A0 and B3–B0 and gives back a sum S3–S0. The OR gate checks if the result is outside of the valid BCD range (0-9) by looking at Cout, S1&S3, and S2&S3. If it is, it means that the result needs to be fixed by adding 6 to the result. The last Sum3-Sum0 outputs show the corrected BCD sum.

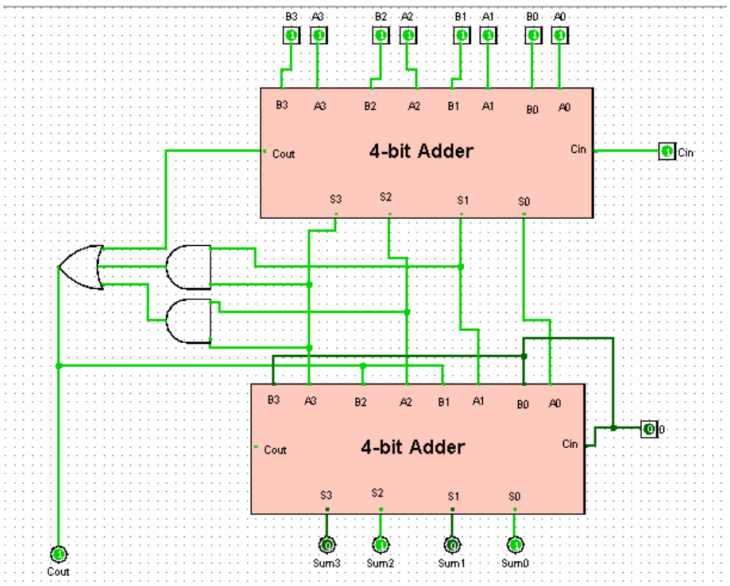


Figure 3: BCD adder 1-digit using ripple adder

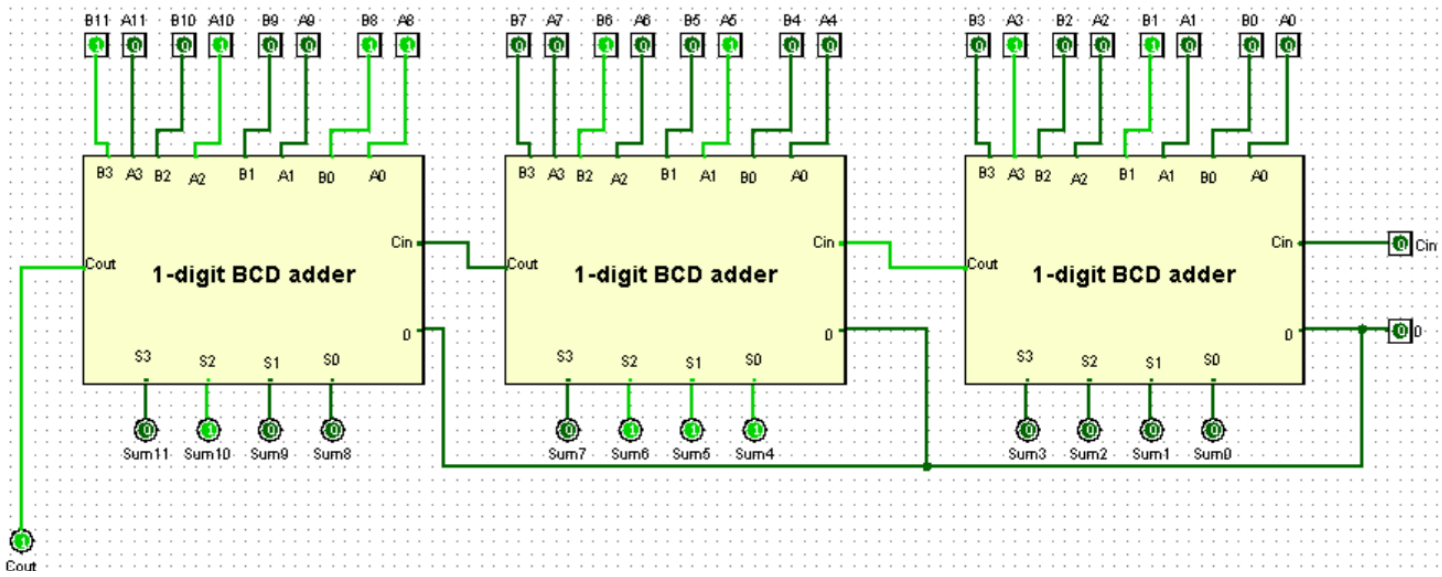
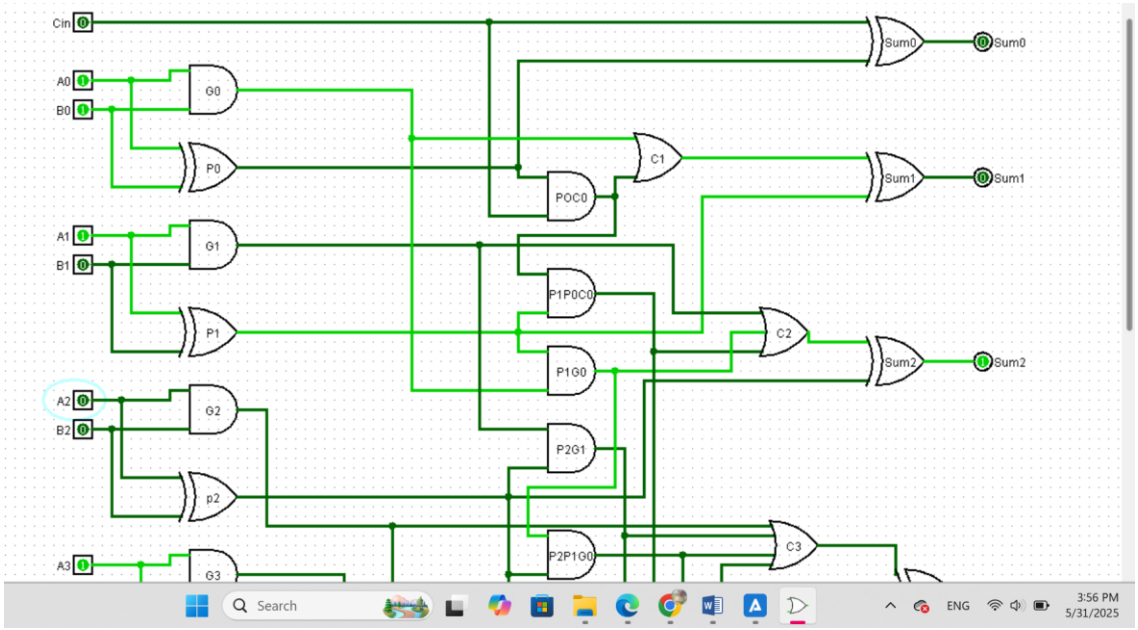


Figure 4: 3-digit BCD adder using Ribble adder

Carry Look-ahead adder (Cla):

In Stage 2 of the project, the design opted to replace the ripple carry adders of Stage 1 with structurally designed carry look-ahead adders to increase the operating speed and to shorten the critical path delay. The CLA was designed without any full adders; instead, it directly computes the carry and sum signals through logic-gates as per the carry look-ahead principle. The Carry_LA_4bit module uses AND and XOR gates to generate the Generate (G) and Propagate (P) signals for each bit, and these signals are used to compute the internal carries C[1] to C[3] and the final carry out Cout in parallel by using hierarchical combinations of AND and OR gates of G and P terms, thus reducing much of the propagation delay, which normally occurs in sequential designs like that of a ripple carry adder.



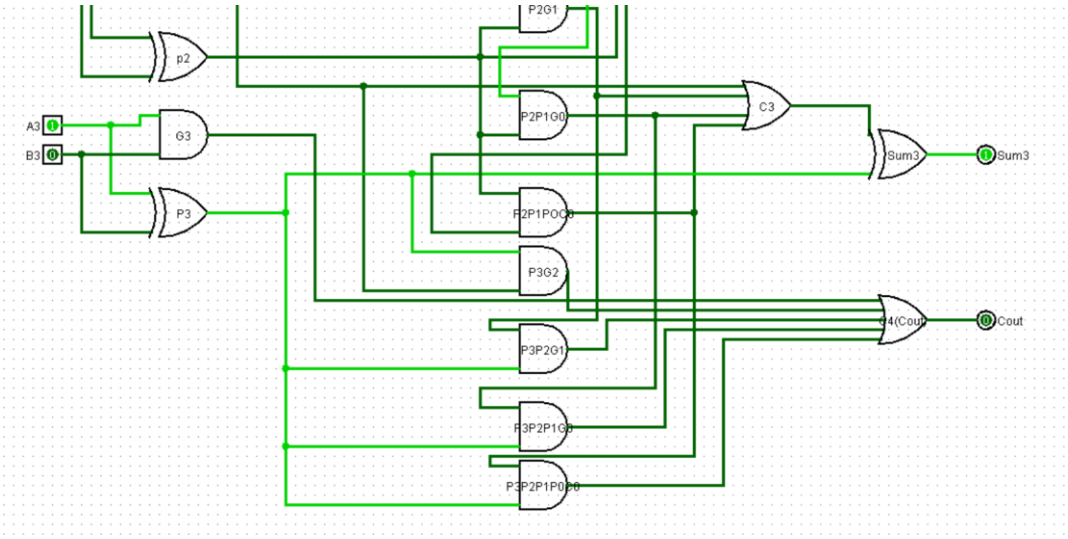


Figure 5: 4-bit CLA design

The CLA adder design was extended to one-digit BCD addition under the name ClA_BCD_1digit, where two 4-bit CLAs are instantiated: The first adder adds the two inputs A and B plus the incoming carry, producing an intermediate 4-bit sum and carry. Based on the BCD rules, there is a correction if the result is greater than 9 or there is an overflow. The correction is done structurally by checking a combination of sum bits and carry through AND and OR gates. If correction is necessary, 6 (4'b0110) is added to the sum by the second CLA instance, resulting in a valid BCD output.

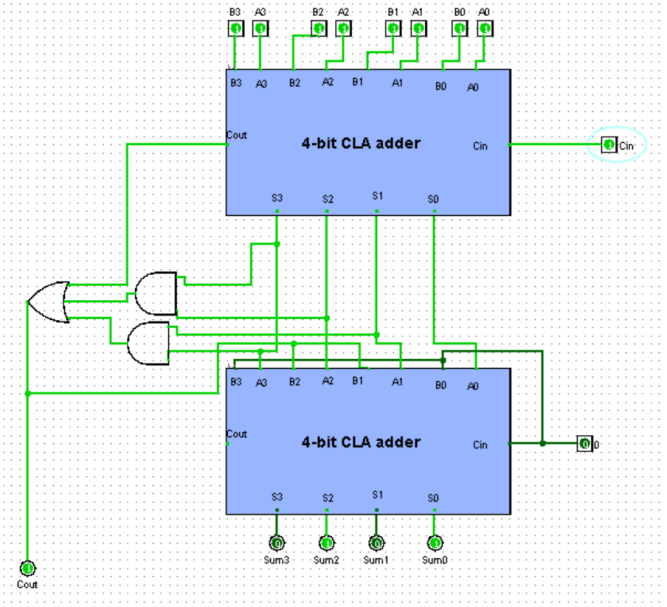


Figure 6: BCD adder 1-digit using CLA adder

To scale this up, the BCD_Adder_ndigit_Cla module was designed to give a 3-digit BCD adder. This design organizes several ClA_BCD_1digit modules chained together structurally in a way that passes the carry from one digit to another. All the inputs and outputs are synchronized by registers clocked and rested. The inputs A, B, and Cin are sampled before computation and the output Sum and Cout are sampled at the end of the clock cycle. This enables reliable and pipelined operation of the system in a synchronous environment

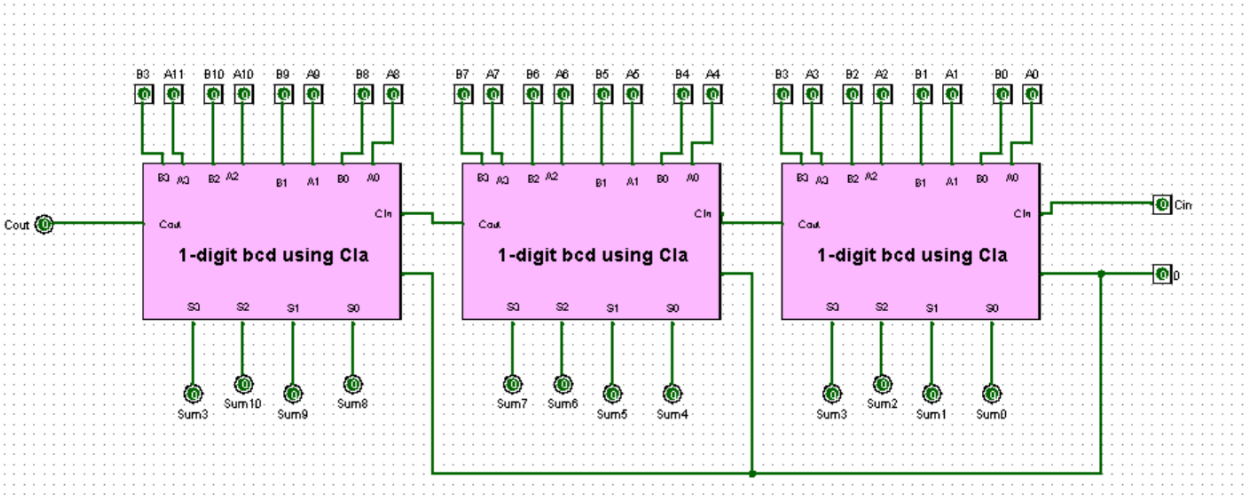


Figure 7: 3-digit BCD adder using CLA adder

Behavioral bcd adder:

The last type of adder to be developed was a Behavioral BCD Adder model that has been made as a golden reference for verification, It is written in a short code of Verilog which is not just another writing language but a way to express concepts clearly and effectively, making it possible to compare the functional correctness of the implemented circuits against the original adder and thereby giving a means to unearth the discrepancies. A testbench is employed in this comparison which checks the output from the ripple carry and CLA implementations along with the behavioral model in terms of various quasi-randomly generated test cases and, in case of errors, it also aids in debugging.

In this BCD adder design, there is a delay for the output, due to the delay of the gates, so the glitches will occur, so to solve this issue, So, the clock will synchronize the inputs with the outputs, to avoid glitches the inputs enter to the registers at a positive edge of the clock, then the output will be stored at their registers at the next positive edge, then the glitches disappear. We need our design to work at the maximum possible frequency without glitches (minimum latency or clock period), by experiment, I found that the minimum clock period without glitches is 20 ns, (e.g. every 10 ns the clock toggled), with reset = 0 at t = 25ns, the maximum possible frequency for the design to avoid glitches is 1 / 20ns = 50000000 HZ.

Results:

In order to verify the correctness of the Ripple-Carry as well as Carry Look-Ahead BCD adders, a behavioral BCD adder was created and used as a reference model. The module is a high-level description BCD adder and performs multi-digit BCD addition. It was presumed to produce correct output. It serves as reference in order to compare the output of the structurally designed adders. For automating and managing this process, a special testbench was created. There is a 20-run random testbench accessed via a generate_valid_bcd function that produces valid 3-digit BCD numbers by randomly selecting between 0 and 9 for each digit. The testbench also produces a random carry-in value of 0 or 1. For each run, the same inputs are presented to the Ripple-Carry, CLA-based, and behavioral BCD adders simultaneously. The output of the structural adders is then compared with the behavioral model prediction. Any differences

are noted, counters are maintained to observe the number of errors per design, and passing cases are displayed for the purpose of visual verification. Such an automatic and random testing method ensures that the structural designs are not just functionally correct, but also robust against a very high number of input combinations. This behavioral-based validation process is critical in creating assurance in the precision and reliability of the entire multi-digit BCD adder system.

The simulation was completed, and all outputs of the console displayed all the Ripple-Carry and Carry Look-Ahead BCD adder test cases passed. Each test case displayed the input operands A and B (in BCD format), the carry-in value, and the resulting sum and carry-out for both adders. During the comparison, every expected output from the behavioral model was identical to both the sum and carry-out from Ripple and CLA adder results for every one of the 20 test cases. There were no errors appearing, and the simulation ended with the following message:

"TEST COMPLETE "

"Ripple Errors: 0"

"CLA Errors: 0"

At this stage, we can conclude that both structural implementations are functionally correct and consistent in the same way as the behavioral model for all tested input combinations.

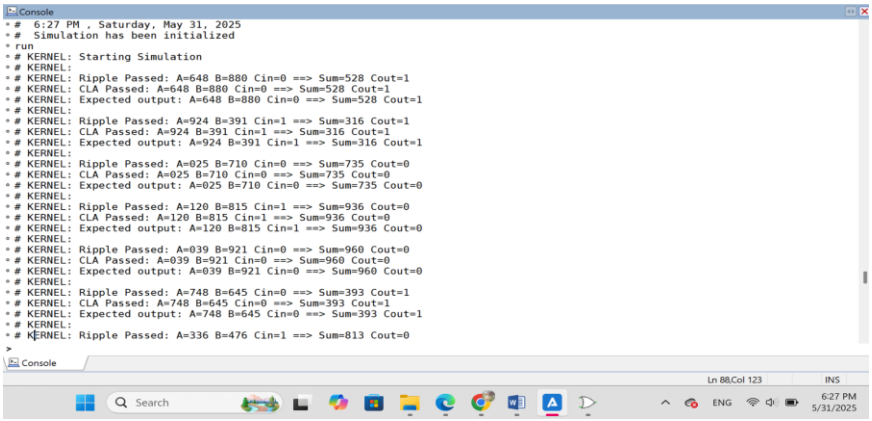


Figure 8: Console result1

Figure 9: console result2

Figures 11 and 12 shows a snapshot of one of the tested cases displayed in the console during simulation. It confirms that the output matched the expected result, and the test case passed successfully.

```
KERNEL:
KERNEL: Ripple Passed: A=120 B=815 Cin=1 ==> Sum=936 Cout=0
KERNEL: CLA Passed: A=120 B=815 Cin=1 ==> Sum=936 Cout=0
KERNEL: Expected output: A=120 B=815 Cin=1 ==> Sum=936 Cout=0
KERNEL:
```

Figure 10: test case

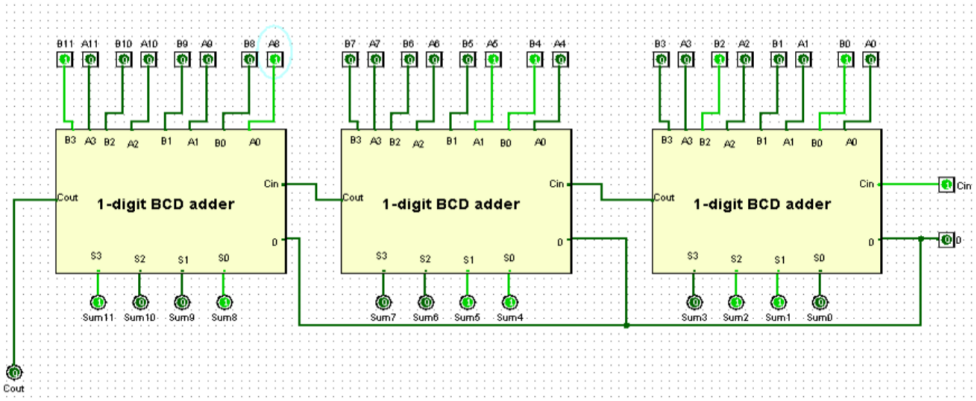


Figure 11: Test 3-digit bcd adder using Ripple adder

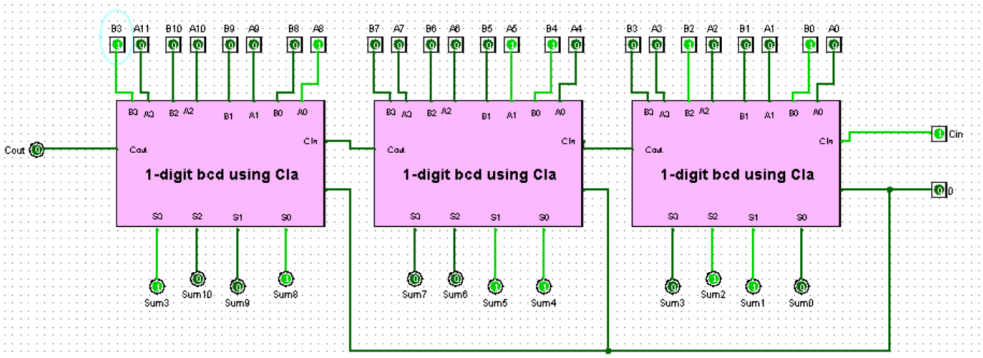


Figure 12: Test 3-digit bcd adder using CLA adder

In Figure 13, the waveform illustrates the results of randomly generated test cases. It visually confirms that the outputs behave correctly over time for various input combinations.

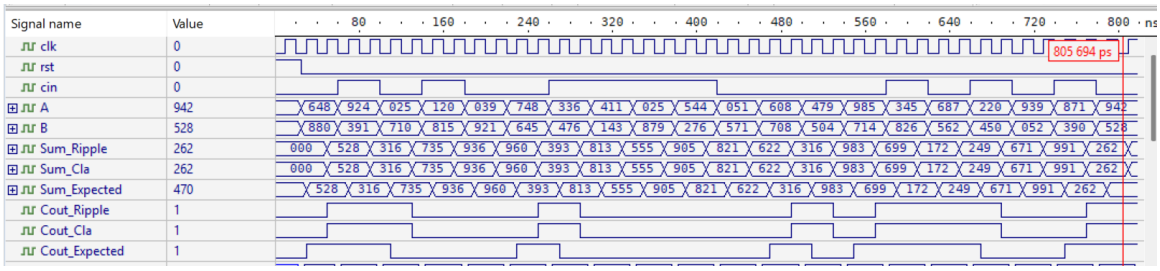


Figure 13: Result waveform

The maximum possible frequency for the design to avoid glitches is 1 / 20ns = 50000000 HZ, and for example with 18 ns clock period (every 9ns the clock toggled), the result:

```
# KERNEL:
# KERNEL: TEST COMPLETE
# KERNEL: Ripple Errors: 0
# KERNEL: CLA Errors   : 0
```

Figure 14: result with minimum frequency

Conclusion & Future works:

In the project, I created a multi-digit Binary-Coded Decimal (BCD) adder using Verilog HDL. The design features two major structural realizations: the Ripple Carry Adder and the Carry Look-Ahead (CLA) Adder. All designs were structurally created using basic logic gates and modular Verilog design. The Ripple Carry BCD adder design was composed of cascaded BCD adders that were each a 1-digit based on 4-bit full adders and correction logic. The CLA-based design used custom logic dedicated to carry look-ahead generation that improved speed, while the correct logic handle decimal overflow. Both forms of the BCD adder were multi-digit capable and could add multi-digit BCD numbers by chaining multiple 1-digit BCD adders together and provide carry propagation.

For accuracy, a third module was built using behavioral Verilog (the reference). A testbench was set up to run random valid BCD inputs and compare the outputs from the Ripple design, the CLA design, and the Behavioral design. The testbench would check for mismatches, and print PASS or error details for each case. Since all designs now had a clock signal, input/output registers were employed to avoid timing hazards or glitches. The waveform results showed proper synchronization and the valid outputs for every one of the random observations (inputs). So, the project achieved total verification success (100%) over all the test cases.

In future development, the project could proceed to include subtraction to facilitate both addition and subtraction of the BCD numbers in the same system using the 10’s complement technique. Increasing the project with the addition of overflow detection schemes would lead to correct operation in edge cases. In addition, the modules of the BCD adder could become part of a more complete set of tools for arithmetic operations in an Arithmetic Logic Unit (ALU). There might also be some exploration on the optimization by area, delay, and power, particularly linked to the Carry Look-Ahead implementation for larger digit sized cases. Finally, adding support for pipelining and high-frequency operation would enhance throughput making the design more suitable for high-frequency application cases such as in digital signal processing or embedded financial systems.

Appendix:

Design Code:

```
//Sojood Asfour 1230298
//sec 2

// 1-bit full adder constructed from basic logic gates
// Computes sum and carry-out for inputs a, b, and carry-in (cin)
module full_adder(input a, b, cin, output sum, cout);
  wire and_ab, and_bcin, and_acin;

  // sum = a XOR b XOR cin
  xor #(15) u1(sum, cin, a, b); // 15 ns

  // carry = (a AND b) OR (b AND cin) OR (a AND cin)
  and #(11) u3(and_ab, a, b);   // 11 ns
  and #(11) u4(and_bcin, b, cin);
  and #(11) u5(and_acin, a, cin);
  or #(11) u6(cout, and_ab, and_bcin, and_acin); // 11 ns
endmodule

// 4-bit adder built using four 1-bit full adders
// Adds two 4-bit inputs x and y with a carry-in and produces a 4-bit sum and carry-out
module adder4_bit(input cin, input [3:0] x, y, output cout, output [3:0] sum);
  wire [4:0] c;
  assign c[0] = cin;
  assign cout = c[4];
  genvar i;
  generate
    for (i = 0; i <= 3; i = i + 1) begin : addbit
```

```
      full_adder adder (.a(x[i]), .b(y[i]), .cin(c[i]), .sum(sum[i]), .cout(c[i+1]));
    end
  endgenerate
endmodule

// 1-digit BCD adder using the 4-bit adder
module BCD1_digit(input cin, input [3:0] A, B, output cout, output [3:0] sum);
  wire [3:0] S, final_sum;
  wire coutadd1, OutputCarry, dummy_cout;

  // First 4-bit addition
  adder4_bit adder1(.cin(cin), .x(A), .y(B), .cout(coutadd1), .sum(S));

  // Check if correction is needed: S > 9 or coutadd1 = 1;
  wire and23, and13;
  and #(11) u1(and13, S[1], S[3]);
  and #(11) u2(and23, S[2], S[3]);
  or #(11) u3(OutputCarry, and13, and23, coutadd1);

  // Add 6 (0110) if correction is needed
  wire [3:0] checkS;
  assign checkS = OutputCarry ? 4'b0110 : 4'b0000;
  adder4_bit adder2(.cin(0), .x(S), .y(checkS), .cout(dummy_cout), .sum(final_sum));

  assign sum = final_sum;
  assign cout = OutputCarry;
endmodule

// n-digit BCD adder (Ripple-Carry based) with clocked input/output registers
// Implements a multi-digit BCD adder by cascading 1-digit BCD adders
// Synchronous design with reset and clock to register inputs and outputs
```

```
module BCD_Adder_ndigit_Ripple #(parameter n = 3)(input clk, rst, cin, input [4*n-1:0] A, B, output reg [4*n-1:0] Sum, output reg Cout);
  reg [4*n-1:0] A_reg, B_reg;
  reg Cin_reg;
  wire [4*n-1:0] sum;
  wire [n:0] Carry;

  assign Carry[0] = Cin_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      A_reg <= 0;
      B_reg <= 0;
      Cin_reg <= 0;
    end else begin
      A_reg <= A;
      B_reg <= B;
      Cin_reg <= cin;
    end
  end

  genvar i;
  generate
    for (i = 0; i < n; i = i + 1) begin : bcd_1digit
      BCD1_digit bcd (.cin(Carry[i]), .A(A_reg[4*i+3:4*i]), .B(B_reg[4*i+3:4*i]), .cout(Carry[i+1]), .sum(sum[4*i+3:4*i]));
    end
  endgenerate

  always @(posedge clk or posedge rst) begin
    if (rst) begin
```

```
      Sum <= 0;
      Cout <= 0;
    end else begin
      Sum <= sum;
      Cout <= Carry[n];
    end
  end
endmodule


// 4-bit Carry Look-Ahead Adder (CLA)
// Computes sum and carry-out faster using generate/propagate logic instead of ripple-carry
// Includes logic for computing intermediate carry signals C1 to C3
module Carry_LA_4bit(input [3:0] A, B, input cin, output [3:0] Sum, output Cout);
  wire [3:0] G, P, C;
  assign C[0] = cin;
  genvar i;
  generate
    for(i = 0; i < 4; i++) begin: GP_signal
      and #(11) (G[i], A[i], B[i]);
      xor #(15) (P[i], A[i], B[i]);
    end
  endgenerate
  wire p0_cin, p1_g0, p1_p0_cin, p2_g1, p2_p1_g0, p2_p1_p0_cin, p3_g2, p3_p2_g1, p3_p2_p1_g0,
p3_p2_p1_p0_cin;
  and #(11) (p0_cin, P[0], cin);
  or #(11) (C[1], G[0], p0_cin);
  and #(11) (p1_g0, P[1], G[0]);
  and #(11) (p1_p0_cin, P[1], P[0], cin);
  or #(11) (C[2], G[1], p1_g0, p1_p0_cin);
  and #(11) (p2_g1, P[2], G[1]);
  and #(11) (p2_p1_g0, P[2], P[1], G[0]);
```

```
  and #(11) (p2_p1_p0_cin, P[2], P[1], P[0], cin);
  or #(11) (C[3], G[2], p2_g1, p2_p1_g0, p2_p1_p0_cin);
  and #(11) (p3_g2, P[3], G[2]);
  and #(11) (p3_p2_g1, P[3], P[2], G[1]);
  and #(11) (p3_p2_p1_g0, P[3], P[2], P[1], G[0]);
  and #(11) (p3_p2_p1_p0_cin, P[3], P[2], P[1], P[0], cin);
  or #(11) (Cout, G[3], p3_g2, p3_p2_g1, p3_p2_p1_g0, p3_p2_p1_p0_cin);
  generate
    for (i = 0; i < 4; i++) begin : generate_sum
      xor #(15) (Sum[i], P[i], C[i]);
    end
  endgenerate
endmodule

// 1-digit BCD adder using Carry Look-Ahead Adder for both normal and correction additions
// Performs initial sum using CLA, applies BCD correction if needed using a second CLA
module ClA_BCD_1digit(input Cin, input [3:0] A, B, output Cout, output [3:0] Sum);
  wire [3:0] sum_adder1, final_sum;
  wire cout_adder1, output_carry, dummy_cout;

  Carry_LA_4bit add1(.A(A), .B(B), .cin(Cin), .Sum(sum_adder1), .Cout(cout_adder1));

  wire and13, and23;
  and #(11)(and13, sum_adder1[1], sum_adder1[3]);
  and #(11)(and23, sum_adder1[2], sum_adder1[3]);
  or #(11)(output_carry, and13, and23, cout_adder1);

  wire [3:0] check_sum1;
  assign check_sum1 = output_carry ? 4'b0110 : 4'b0000;
  Carry_LA_4bit add2(.A(sum_adder1), .B(check_sum1), .cin(1'b0), .Sum(final_sum), .Cout(dummy_cout));
```

```
  assign Sum = final_sum;
  assign Cout = output_carry;
endmodule


// n-digit BCD adder using Carry Look-Ahead Adders
// Structurally builds a multi-digit BCD adder using CLA-based 1-digit BCD adders
// Registers inputs and outputs using clock and reset for synchronous operation
module BCD_Adder_ndigit_Cla #(parameter n = 3)(input clk, rst, cin, input [4*n-1:0] A, B, output reg [4*n-1:0] Sum, output reg Cout);
  reg [4*n-1:0] A_reg, B_reg;
  reg Cin_reg;
  wire [4*n-1:0] sum;
  wire [n:0] Carry;

  assign Carry[0] = Cin_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      A_reg <= 0;
      B_reg <= 0;
      Cin_reg <= 0;
    end else begin
      A_reg <= A;
      B_reg <= B;
      Cin_reg <= cin;
    end
  end

  genvar i;
  generate
    for (i = 0; i < n; i++) begin : bcd_1digit
```

```
    ClA_BCD_1digit bcd (.Cin(Carry[i]), .A(A_reg[4*i+3:4*i]), .B(B_reg[4*i+3:4*i]), .Cout(Carry[i+1]),
.Sum(sum[4*i+3:4*i]));
   end
  endgenerate


  always @(posedge clk or posedge rst) begin
   if (rst) begin
     Sum <= 0;
     Cout <= 0;
   end else begin
     Sum <= sum;
     Cout <= Carry[n];
   end
  end
endmodule


module BCD_Adder_ndigit_Behavioral #(parameter n = 3)(input clk, rst, cin, input [4*n-1:0] A, B, output reg
[4*n-1:0] Sum, output reg Cout);


  integer i;
  reg [4:0] temp;
  reg [3:0] a_digit, b_digit;
  reg carry;
  reg [4*n-1:0] next_Sum;


  always @(posedge clk or posedge rst) begin
   if (rst) begin
     Sum  <= 0;
     Cout <= 0;
   end else begin
     carry = cin;
```

```
    next_Sum = 0;

    for (i = 0; i < n; i = i + 1) begin
      // Extract digit i from A and B using shift
      a_digit = (A >> (i * 4)) & 4'b1111;
      b_digit = (B >> (i * 4)) & 4'b1111;

      temp = a_digit + b_digit + carry;

      if (temp > 9) begin
        temp = temp + 6;
        carry = 1;
      end else begin
        carry = 0;
      end

      // Store result in correct place using shift
      next_Sum = next_Sum | (temp[3:0] << (i * 4));
    end

    Sum  <= next_Sum;
    Cout <= carry;
  end
 end

endmodule
```

Testbench Code:

```
//Sojood Asfour 1230298
//sec 2
```

```
`timescale 1ns/1ps

module BCD_Adder_Test_All;
  reg clk, rst, cin;
  reg [11:0] A, B; // 3-digit BCD input
  wire [11:0] Sum_Ripple, Sum_Cla, Sum_Expected;
  wire Cout_Ripple, Cout_Cla, Cout_Expected;

  integer i, errors_ripple, errors_cla;

  //Instantiateof Ripple-Carry BCD Adder
  BCD_Adder_ndigit_Ripple #(3) ripple_adder (clk, rst , cin, A, B, Sum_Ripple, Cout_Ripple);

  //Instantiate Of CLA BCD Adder
  BCD_Adder_ndigit_Cla #(3) cla_adder (clk, rst , cin, A, B, Sum_Cla, Cout_Cla);

  //Instantiate of Behavioral BCD Adder
  BCD_Adder_ndigit_Behavioral #(3) behavioral_adder (clk, rst , cin, A, B, Sum_Expected,Cout_Expected);

  //Clock generation
  initial clk = 0;
  always #10 clk = ~clk;

  //BCD Generator: each digit 0–9
  function [11:0] generate_valid_bcd;
    reg [3:0] d0, d1, d2;
    begin
      d0 = $urandom_range(0, 9);
      d1 = $urandom_range(0,9);
      d2 = $urandom_range(0,9);
```

```
    generate_valid_bcd = {d2, d1, d0};
  end
endfunction

//Main Test Process
initial begin
  $display("Starting Simulation\n");
  errors_ripple = 0;
  errors_cla = 0;

  rst = 1; cin = 0; A = 0; B = 0;
  #25; rst = 0;

  for (i = 0; i < 20; i = i + 1) begin
    A = generate_valid_bcd();
    B = generate_valid_bcd();
    cin = $urandom_range(0, 1);

    @(posedge clk);
    #30;

    //Check Ripple adder
    if (Sum_Ripple !== Sum_Expected || Cout_Ripple !== Cout_Expected) begin
      $display("Ripple Error @ Test %0d", i+1);
      $display("    A     = %0d%0d%0d",A[11:8], A[7:4], A[3:0]);
      $display("    B     = %0d%0d%0d",B[11:8], B[7:4], B[3:0]);
      $display("    Cin   = %b", cin);
      $display("    Ripple: Sum = %0d, Cout = %b", Sum_Ripple, Cout_Ripple);
      $display("    Expect: Sum = %0d, Cout = %b\n", Sum_Expected, Cout_Expected);
      errors_ripple = errors_ripple + 1;
    end
```

```
        else begin
                $display("Ripple Passed: A=%0d%0d%0d B=%0d%0d%0d Cin=%b ==> Sum=%0d%0d%0d
Cout=%b",
       A[11:8], A[7:4], A[3:0], B[11:8], B[7:4], B[3:0], cin, Sum_Ripple[11:8], Sum_Ripple[7:4],
Sum_Ripple[3:0],  Cout_Ripple);
    end


    //CLA test
    if (Sum_Cla !== Sum_Expected || Cout_Cla !== Cout_Expected) begin
      $display("CLA Error @ Test %0d", i+1);
      $display("    A     = %0d", A);
      $display("    B     = %0d", B);
      $display("    Cin   = %b", cin);
      $display("    CLA   : Sum = %0d, Cout = %b", Sum_Cla, Cout_Cla);
      $display("    Expect: Sum = %0d, Cout = %b\n", Sum_Expected, Cout_Expected);
      errors_cla = errors_cla + 1;
    end
        else begin
      $display("CLA Passed: A=%0d%0d%0d B=%0d%0d%0d Cin=%b ==> Sum=%0d%0d%0d Cout=%b",
                A[11:8], A[7:4], A[3:0], B[11:8], B[7:4], B[3:0], cin, Sum_Cla[11:8], Sum_Cla[7:4],
Sum_Cla[3:0], Cout_Cla);
                $display("Expected output: A=%0d%0d%0d B=%0d%0d%0d Cin=%b ==> Sum=%0d%0d%0d
Cout=%b\n",
       A[11:8], A[7:4], A[3:0], B[11:8], B[7:4], B[3:0], cin, Sum_Expected[11:8], Sum_Expected[7:4],
Sum_Expected[3:0], Cout_Expected);
    end
  end


  $display("\nTEST COMPLETE");
  $display("Ripple Errors: %0d", errors_ripple);
  $display("CLA Errors   : %0d", errors_cla);
```

```
    $finish;
  end
endmodule
```