BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Advanced Digital Design - ENCS533

<u>Course Project Report</u>

Name: Nagham Moutaz Maali

ID: 1212312

Section: 2

Instructor: Dr. Abdellatif Abu-Issa

Contents

1. Introduction

The goal of this project is to design and implement a 3-digit BCD (Binary-Coded Decimal) adder using structural modeling in Verilog HDL, and then verify its functionality and timing using simulation tools. The project is divided into two stages, each exploring a different type of adder: Ripple Carry Adder in Stage 1 and Carry Look-Ahead Adder in Stage 2.

BCD (Binary-Coded Decimal) is a way of representing decimal numbers where each digit is coded using 4 binary bits. For example, the decimal number 529 is represented in BCD as three separate groups: 0101 (5), 0010 (2), and 1001 (9). BCD is especially important in digital systems that interact with decimal data, like calculators, digital displays, and financial systems, where accuracy in representing decimal digits is essential.

In this project, the 3-digit BCD adder receives its inputs and produces outputs through registers (flip-flops). The design is done structurally, meaning each component (like full adders or logic gates) is built from basic gates such as AND, OR, XOR, etc., with specified gate delays. The design follows a bottom-up approach: I start with a 1-bit full adder, then build a 4-bit binary adder, then use that to create a 1-digit BCD adder, and finally combine three of them to make a 3-digit BCD adder.

In Stage 1, a Ripple Carry Adder (RCA) is used. This type of adder is simple to build but has a slower carry propagation time.

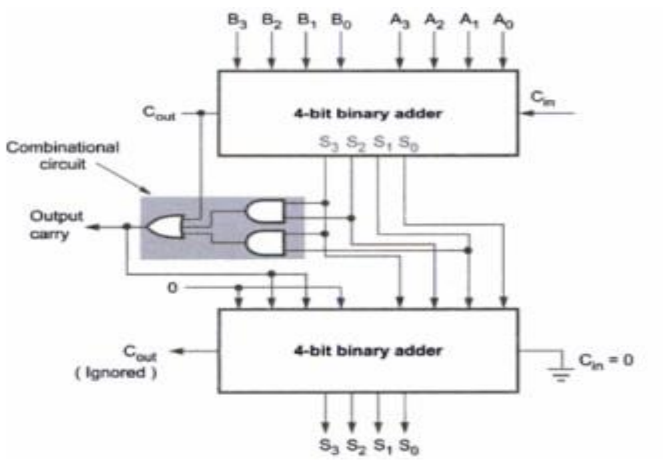
In Stage 2, a Carry Look-Ahead Adder (CLA) is introduced. It speeds up the addition process by reducing the delay caused by carry propagation.

Both stages include not only functional simulation but also timing analysis to find the maximum latency and therefore the maximum operating clock frequency. Additionally, each stage includes a test scenario where a bug is intentionally introduced, and a verification mechanism is used to detect the error and write it to a file.

By the end of this project, I gain hands-on experience in digital design, structural modeling, simulation, and understanding how adder types affect the performance of arithmetic circuits.

2. Design View

This project implements a 3-digit Binary Coded Decimal (BCD) adder, using a hierarchical top-down design approach. The general architecture of the system is shown in Figure 1:



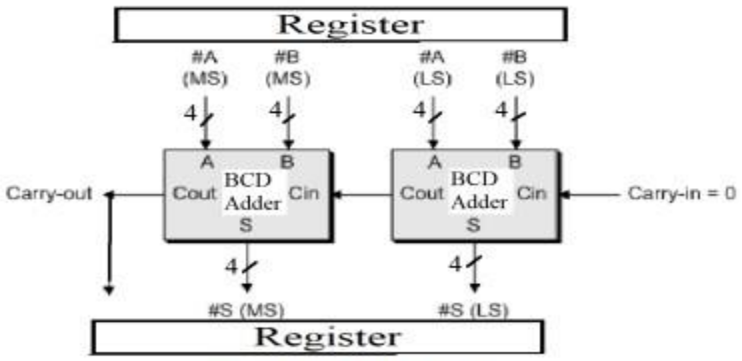
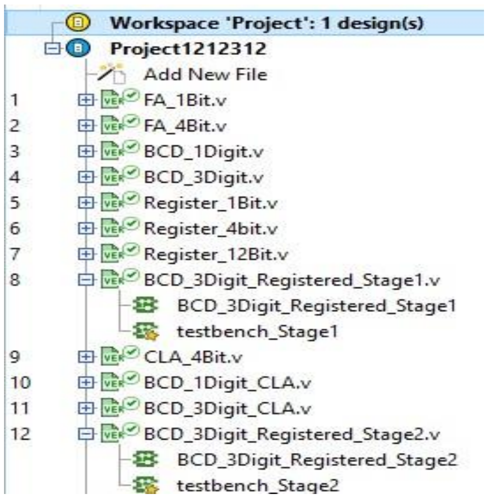


Figure 1: 2-Digit BCD Adder

It illustrates the modular breakdown and connectivity across two design stages.

The system is designed in two stages:



```
Workspace 'Project': 1 design(s)
  Project1212312
    Add New File
1     FA_1Bit.v
2     FA_4Bit.v
3     BCD_1Digit.v
4     BCD_3Digit.v
5     Register_1Bit.v
6     Register_4bit.v
7     Register_12Bit.v
8     BCD_3Digit_Registered_Stage1.v
        BCD_3Digit_Registered_Stage1
        testbench_Stage1
9     CLA_4Bit.v
10    BCD_1Digit_CLA.v
11    BCD_3Digit_CLA.v
12    BCD_3Digit_Registered_Stage2.v
        BCD_3Digit_Registered_Stage2
        testbench_Stage2
```

- Stage 1: Ripple-Carry Based BCD Adder

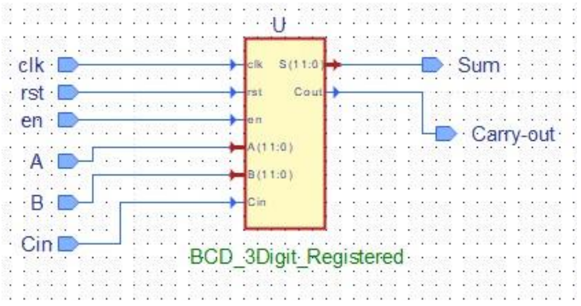
  This stage starts with a basic 1-bit Full Adder, and builds progressively up to the full system:

1. FA_1Bit: Performs binary addition on single bits (A, B, Cin) and produces Sum and Cout.
2. FA_4Bit: A 4-bit adder created by chaining four FA_1Bit modules.
3. BCD_1Digit: Adds two 4-bit BCD values and applies correction (adding 6 = 0110) if the sum exceeds 9 or a carry is generated.
4. BCD_3Digit: Cascades three BCD_1Digit modules to handle 12-bit (3-digit BCD) numbers. Carries are forwarded between digits to maintain correctness across digits.
5. BCD_3Digit_Registered_Stage1: Full 3-digit BCD adder system using ripple-carry adders and BCD correction, with registered inputs and outputs. It adds input/output registers around the BCD adder using D flip-flops for synchronization and pipelining.

- Stage 2: Carry Lookahead Based BCD Adder

  This stage introduces performance optimization using Carry Lookahead logic.

  1. CLA_4Bit: A 4-bit adder using Carry-Lookahead logic, which improves speed compared to the ripple-carry version.
     It reduces addition delay. Instead of waiting for each carry to ripple through, it computes them in parallel using generate (G) and propagate (P) logic.
     This would replace FA_4Bit in the BCD structure, allowing a faster 1-digit BCD adder and subsequently a faster BCD_3Digit adder when scaled.
  2. BCD_3Digit_Registered_Stage2: Full 3-digit BCD adder system using Carry-Lookahead adder and BCD correction, with registered inputs and outputs. It adds input/output registers around the BCD adder using D flip-flops for synchronization and pipelining.



Block

To enable synchronization of inputs and outputs, and to support pipelined processing, the design includes:

1. Register_1Bit (D flip-flop): A single D flip-flop with reset and enable, used to store a 1-bit value synchronously.
2. Register_4Bit: Groups 4 D flip-flops to store 4-bit values.
3. Register_12Bit: Stores the full 12-bit input/output values.

I used D flip-flops because they are simple, predictable, and ideal for data storage. Unlike T or JK flip-flops, which are more suited to toggling or complex logic conditions, D flip-flops store the value of the input D on the rising edge of the clock, making them perfect for synchronous designs where reliable data capture and register-based pipelining are required.

✓ Note: All the modules will be in 📎 Appendices section

3. Adder Architecture

3.1. Stage 1: Ripple Carry Adder

I built a 1-bit full adder using basic gates: Sum = $(A \oplus B) \oplus Cin$ and Cout = (A & B) | (A & Cin) | (B & Cin)

It has two XOR gates for the sum and three AND + two OR gates for the carry.

then I connected four 1-bit full adders in series. The carry-out from each bit goes into the next as carry-in.

In BCD, valid sums are from 0000 to 1001 (0–9). If the sum > 9 or Cout = 1, we need to add 6 (0110) to correct it. This is done by a second adder that adds 6 if needed.

Each FA_1Bit has:

2 XORs × 15 ns = 30 ns

3 ANDs + 2 ORs × 11 ns = 55 ns

Total = 30 + 55 = 85 ns per full adder

For 4-bit RCA:

Latency = 4 × 85 = 340 ns

The maximum frequency = 1/total Latency = 1/ (340 * 10^-9) = 2.94MHz

3.2. Stage 2: Carry Look-Ahead Adder

The CLA is faster because it calculates all carries in parallel instead of waiting. It uses Generate (G) and Propagate (P) logic to find each carry quickly:

G = A & B

P = $A \oplus B$

Then I compute each carry using logic formulas. This reduces delay since carry doesn't ripple.

Similar to the ripple design, I used a second adder to add 6 (0110) if the 4-bit sum > 9 or Cout = 1. CLA improves the speed of both stages.

Each CLA bit still uses logic gates but in parallel:

XORs: 4 × 15 ns = 60 ns

AND/OR logic for carry: approximately 66 gates × 11 ns = 726 ns

But due to parallelism, worst-case delay ≈ carry-out delay = roughly 121 ns

Which is Much better than ripple’s 340 ns

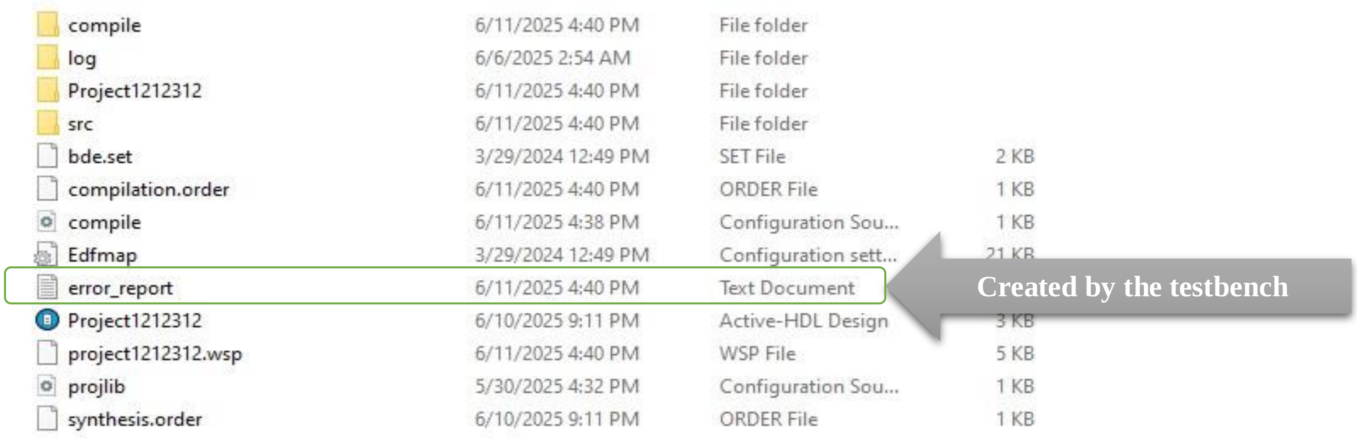
The maximum frequency = 1/total Latency = 1/ (121 * 10^-9) = 8.26MHz

4. Simulation and Verification

For both Stage 1 and Stage 2, I created similar testbenches to verify the correct behavior of the registered 3-digit BCD adder module. Each testbench includes:

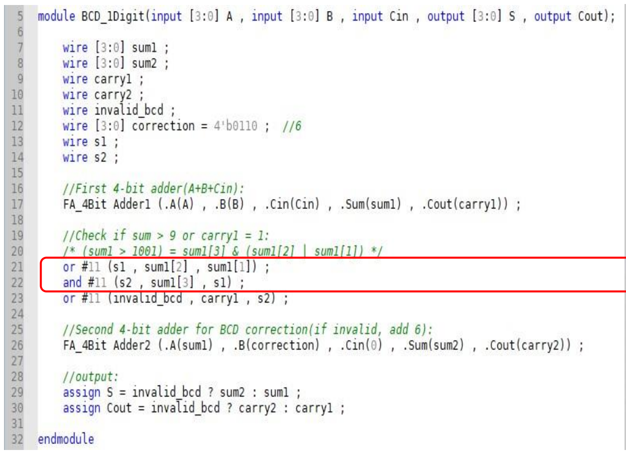
- Clock Generator using initial and forever to create a periodic signal.
- Reset and Enable Logic to initialize and activate the design under test (DUT).
- Input Stimuli: Carefully selected test cases that target corner cases and typical use cases.
- Output Checkers: Use of conditional checks (if, ===) to compare actual and expected results.
- Error Logging: For error detection, the testbench writes to an external file (error_report.txt) if a mismatch is found during testing.

The testbench code simulates real timing behavior using a clock generation block with a 100ns period. A reset and enable mechanism ensures the system starts in a known state, allowing reliable test execution. Input vectors are applied sequentially in different test cases, with two clock cycles of delay to let the outputs stabilize before evaluation. Conditional if statements compare actual and expected outputs: when outputs match, a success message is printed using $display; when mismatches occur—especially during the intentional error injection test—clear error messages are shown. These specific errors are deliberately introduced to verify the robustness of the testbench and the design’s ability to handle faulty scenarios. All mismatches during this phase are also logged into a text file, error_report.txt, using Verilog’s file I/O system. This report provides a persistent, organized summary of failures, including the time, test case number, and the nature of the discrepancy, which is crucial for debugging and validating the system’s error-handling capabilities.

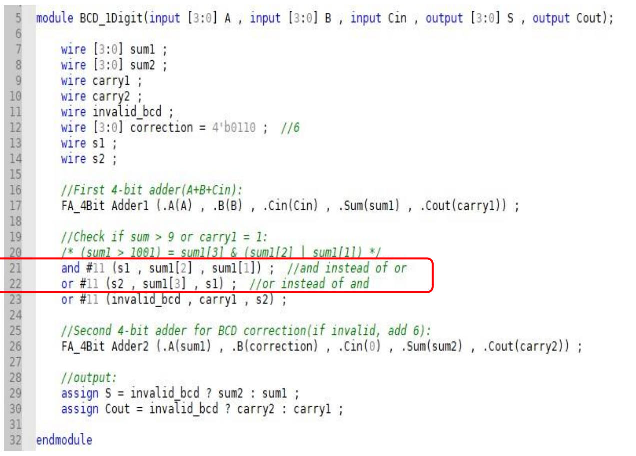


compile	6/11/2025 4:40 PM	File folder	
log	6/6/2025 2:54 AM	File folder	
Project1212312	6/11/2025 4:40 PM	File folder	
src	6/11/2025 4:40 PM	File folder	
bde.set	3/29/2024 12:49 PM	SET File	2 KB
compilation.order	6/11/2025 4:40 PM	ORDER File	1 KB
compile	6/11/2025 4:38 PM	Configuration Sou...	1 KB
Edfmap	3/29/2024 12:49 PM	Configuration sett...	21 KB
error_report	6/11/2025 4:40 PM	Text Document	
Project1212312	6/10/2025 9:11 PM	Active-HDL Design	3 KB
project1212312.wsp	6/11/2025 4:40 PM	WSP File	5 KB
projlib	5/30/2025 4:32 PM	Configuration Sou...	1 KB
synthesis.order	6/10/2025 9:11 PM	ORDER File	1 KB

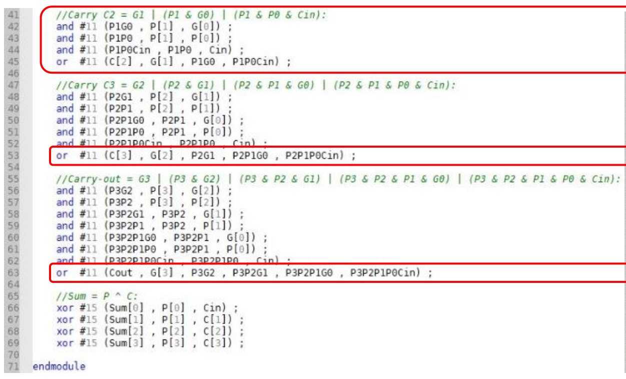
To further evaluate the reliability of the testbench, I intentionally introduced logic gate errors in both BCD_1Digit.v and CLA_4Bit.v. In BCD_1Digit.v, an AND gate was replaced with an OR gate: or #11 (s2 , sum1[3] , s1);, and an OR gate was replaced with an AND: and #11 (s1 , sum1[2] , sum1[1]);. These changes distorted the expected binary-to-BCD conversion behavior. In CLA_4Bit.v, I introduced similar logic flaws: required AND gates were replaced with OR gates like or #11 (P1G0 , P[1] , G[0]);—and critical OR logic was mistakenly substituted with AND gates like and #11 (C[2] , G[1] , P1G0 , P1P0Cin);. These faults disrupt correct carry propagation and result in erroneous final outputs. The testbench was able to identify these faults, confirming that it can detect even subtle internal errors.



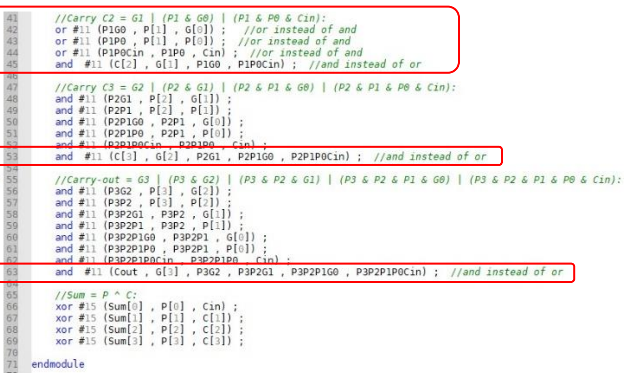
```verilog
module BCD_1Digit(input [3:0] A , input [3:0] B , input Cin , output [3:0] S , output Cout);

    wire [3:0] sum1 ;
    wire [3:0] sum2 ;
    wire carry1 ;
    wire carry2 ;
    wire invalid_bcd ;
    wire [3:0] correction = 4'b0110 ;  //6
    wire s1 ;
    wire s2 ;

    //First 4-bit adder(A+B+Cin):
    FA_4Bit Adder1 (.A(A) , .B(B) , .Cin(Cin) , .Sum(sum1) , .Cout(carry1)) ;

    //Check if sum > 9 or carry1 = 1:
    /* (sum1 > 1001) = sum1[3] & (sum1[2] | sum1[1]) */
    or #11 (s1 , sum1[2] , sum1[1]) ;
    and #11 (s2 , sum1[3] , s1) ;
    or #11 (invalid_bcd , carry1 , s2) ;

    //Second 4-bit adder for BCD correction(if invalid, add 6):
    FA_4Bit Adder2 (.A(sum1) , .B(correction) , .Cin(0) , .Sum(sum2) , .Cout(carry2)) ;

    //output:
    assign S = invalid_bcd ? sum2 : sum1 ;
    assign Cout = invalid_bcd ? carry2 : carry1 ;

endmodule
```



```verilog
module BCD_1Digit(input [3:0] A , input [3:0] B , input Cin , output [3:0] S , output Cout);

    wire [3:0] sum1 ;
    wire [3:0] sum2 ;
    wire carry1 ;
    wire carry2 ;
    wire invalid_bcd ;
    wire [3:0] correction = 4'b0110 ;  //6
    wire s1 ;
    wire s2 ;

    //First 4-bit adder(A+B+Cin):
    FA_4Bit Adder1 (.A(A) , .B(B) , .Cin(Cin) , .Sum(sum1) , .Cout(carry1)) ;

    //Check if sum > 9 or carry1 = 1:
    /* (sum1 > 1001) = sum1[3] & (sum1[2] | sum1[1]) */
    and #11 (s1 , sum1[2] , sum1[1]) ;  //and instead of or
    or #11 (s2 , sum1[3] , s1) ;  //or instead of and
    or #11 (invalid_bcd , carry1 , s2) ;

    //Second 4-bit adder for BCD correction(if invalid, add 6):
    FA_4Bit Adder2 (.A(sum1) , .B(correction) , .Cin(0) , .Sum(sum2) , .Cout(carry2)) ;

    //output:
    assign S = invalid_bcd ? sum2 : sum1 ;
    assign Cout = invalid_bcd ? carry2 : carry1 ;

endmodule
```



```verilog
    //Carry C2 = G1 | (P1 & G0) | (P1 & P0 & Cin):
    and #11 (P1G0 , P[1] , G[0]) ;
    and #11 (P1P0 , P[1] , P[0]) ;
    and #11 (P1P0Cin , P1P0 , Cin) ;
    or  #11 (C[2] , G[1] , P1G0 , P1P0Cin) ;

    //Carry C3 = G2 | (P2 & G1) | (P2 & P1 & G0) | (P2 & P1 & P0 & Cin):
    and #11 (P2G1 , P[2] , G[1]) ;
    and #11 (P2P1 , P[2] , P[1]) ;
    and #11 (P2P1G0 , P2P1 , G[0]) ;
    and #11 (P2P1P0 , P2P1 , P[0]) ;
    and #11 (P2P1P0Cin , P2P1P0 , Cin) ;
    or  #11 (C[3] , G[2] , P2G1 , P2P1G0 , P2P1P0Cin) ;

    //Carry-out = G3 | (P3 & G2) | (P3 & P2 & G1) | (P3 & P2 & P1 & G0) | (P3 & P2 & P1 & P0 & Cin):
    and #11 (P3G2 , P[3] , G[2]) ;
    and #11 (P3P2 , P[3] , P[2]) ;
    and #11 (P3P2G1 , P3P2 , G[1]) ;
    and #11 (P3P2P1 , P3P2 , P[1]) ;
    and #11 (P3P2P1G0 , P3P2P1 , G[0]) ;
    and #11 (P3P2P1P0 , P3P2P1 , P[0]) ;
    and #11 (P3P2P1P0Cin , P3P2P1P0 , Cin) ;
    or  #11 (Cout , G[3] , P3G2 , P3P2G1 , P3P2P1G0 , P3P2P1P0Cin) ;

    //Sum = P ^ C:
    xor #15 (Sum[0] , P[0] , Cin) ;
    xor #15 (Sum[1] , P[1] , C[1]) ;
    xor #15 (Sum[2] , P[2] , C[2]) ;
    xor #15 (Sum[3] , P[3] , C[3]) ;

endmodule
```



```verilog
    //Carry C2 = G1 | (P1 & G0) | (P1 & P0 & Cin):
    or #11 (P1G0 , P[1] , G[0]) ;   //or instead of and
    or #11 (P1P0 , P[1] , P[0]) ;  //or instead of and
    or #11 (P1P0Cin , P1P0 , Cin) ;  //or instead of and
    and  #11 (C[2] , G[1] , P1G0 , P1P0Cin) ;  //and instead of or

    //Carry C3 = G2 | (P2 & G1) | (P2 & P1 & G0) | (P2 & P1 & P0 & Cin):
    and #11 (P2G1 , P[2] , G[1]) ;
    and #11 (P2P1 , P[2] , P[1]) ;
    and #11 (P2P1G0 , P2P1 , G[0]) ;
    and #11 (P2P1P0 , P2P1 , P[0]) ;
    and #11 (P2P1P0Cin , P2P1P0 , Cin) ;
    and  #11 (C[3] , G[2] , P2G1 , P2P1G0 , P2P1P0Cin) ;  //and instead of or

    //Carry-out = G3 | (P3 & G2) | (P3 & P2 & G1) | (P3 & P2 & P1 & G0) | (P3 & P2 & P1 & P0 & Cin):
    and #11 (P3G2 , P[3] , G[2]) ;
    and #11 (P3P2 , P[3] , P[2]) ;
    and #11 (P3P2G1 , P3P2 , G[1]) ;
    and #11 (P3P2P1 , P3P2 , P[1]) ;
    and #11 (P3P2P1G0 , P3P2P1 , G[0]) ;
    and #11 (P3P2P1P0 , P3P2P1 , P[0]) ;
    and #11 (P3P2P1P0Cin , P3P2P1P0 , Cin) ;
    and  #11 (Cout , G[3] , P3G2 , P3P2G1 , P3P2P1G0 , P3P2P1P0Cin) ;  //and instead of or

    //Sum = P ^ C:
    xor #15 (Sum[0] , P[0] , Cin) ;
    xor #15 (Sum[1] , P[1] , C[1]) ;
    xor #15 (Sum[2] , P[2] , C[2]) ;
    xor #15 (Sum[3] , P[3] , C[3]) ;

endmodule
```

Initially, all test cases passed with no detected errors to a file. After injecting the faults, most test cases failed, and errors were accurately identified and logged to error_report.txt. This confirmed the robustness of the testbench and its ability to validate both correct functionality and error-handling mechanisms.

- ✓ Note: the testbenches, screen shots of the outputs and test cases table will be in 📎 Appendices section

5. Result Summary and Conclusion

This project focused on designing, simulating, and verifying a registered 3-digit BCD adder system using Verilog, with careful attention to both functionality and error detection in two stages: Stage 1 uses a Ripple Carry Adder (RCA), while Stage 2 uses a faster Carry Look-Ahead Adder (CLA).

By building each module step by step and testing them through well-structured testbenches, we ensured the system behaved correctly under normal and faulty conditions. A summary of performance metrics is shown in the table below:

Design Stage	Latency (ns)	Max Frequency (MHz)	Error Detection	Structural Complexity
1: RCA	340	2.94	High	Moderate
2: CLAA	121	8.26	High	High

In Stage 1, the ripple carry design used four 1-bit full adders in series, where each carry-out signal ripples to the next stage. While this structure is straightforward and easy to implement, it causes longer delays because each carry must wait for the one before it. In contrast, Stage 2 used a carry look-ahead logic system that calculates all carries in parallel using generate and propagate signals. This significantly reduced the delay and increased the overall performance. Despite using more gates, the CLA's parallelism led to a lower latency and a much higher maximum operating frequency.

Throughout the project, we used several important concepts and techniques. Structural modeling was key for building the modules in a clean, organized way. Gate-level modeling helped us simulate real timing behavior by including delays for logic gates. The use of always, initial, and assign statements allowed us to implement both combinational and sequential logic. Another major technique was the use of testbenches to apply inputs, monitor outputs, and report mismatches, which was crucial in verifying the functionality of the system.

One of the most important achievements was proving the testbench's reliability. All module codes are free of syntax errors, and the outputs are correct when the design is implemented as intended. When faults were intentionally inserted into the logic, the testbench successfully detected the errors and logged them clearly to an external text file (error_report.txt), including useful information such as the type of failure and place. This highlights the importance of robust verification in digital design.

From this project, I gained a solid understanding of how structural design works in Verilog, how delays affect timing, and how to simulate real hardware behavior. I also learned the value of creating reusable, readable code and using systematic testing to catch even small internal faults. If this project were to be improved in the future, one area to consider would be pipelining to increase speed. It could also be optimized for better power efficiency and made scalable for larger digit operations or more advanced BCD arithmetic systems. Overall, this project was a great learning experience in combining theory with hands-on design and verification practice.

📎 Appendices

This section has all modules, screen shots of the output after simulation and run, block diagram for each stage and test cases table.

```
/* Structural Verilog model for a 1-bit Full Adder */
        /*inputs: 1-bit A
                1-bit B
                Carry-in
        */
        /*outputs: 1-bit Sum
                 Carry-out
        */

`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.

module FA_1Bit(input A , input B , input Cin , output Sum , output Cout) ;
        wire AxorB ;
        wire AandB ;
        wire AandCin ;
        wire BandCin ;
        wire AandBorAandCin ;
        xor #15 (AxorB , A , B) ;  //(A^B)
        xor #15 (Sum , AxorB , Cin) ;  //Sum = (A^B)^C
        and #11 (AandB , A , B) ;       //(A&B)
        and #11 (AandCin , A , Cin) ;  //(A&Cin)
        and #11 (BandCin , B , Cin) ;  //(B&Cin)
        or #11 (AandBorAandCin, AandB , AandCin) ;  //(A&B) | (A&Cin)
        or #11 (Cout , AandBorAandCin , BandCin) ;  //Cout = (A&B) | (A&Cin) | (B&Cin)
endmodule
```

```
/* Structural Verilog model for a 4-bit Full Adder(Ripple Carry Adder) */
```

```
        /*inputs: 4-bit A
                4-bit B
                Carry-in
        */
        /*outputs: 4-bit Sum
                 Carry-out
        */
        /* The design uses four 1-bit full adders connected in series */

`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.

module FA_4Bit(input [3:0] A , input [3:0] B , input Cin , output [3:0] Sum , output Cout) ;
        wire c1 ;
        wire c2 ;
        wire c3 ;
        FA_1Bit U1 (.A(A[0]) , .B(B[0]) , .Cin(Cin) , .Sum(Sum[0]) , .Cout(c1)) ;
        FA_1Bit U2 (.A(A[1]) , .B(B[1]) , .Cin(c1) , .Sum(Sum[1]) , .Cout(c2)) ;
        FA_1Bit U3 (.A(A[2]) , .B(B[2]) , .Cin(c2) , .Sum(Sum[2]) , .Cout(c3)) ;
        FA_1Bit U4 (.A(A[3]) , .B(B[3]) , .Cin(c3) , .Sum(Sum[3]) , .Cout(Cout)) ;
endmodule
```

```
/*D-ff*/
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.

module Register_1Bit(input clk , input rst , input en , input D , output reg Q) ;
   always @(posedge clk or posedge rst)
        begin
        if (rst)
            Q <= 1'b0 ;  //Reset to 0 when reset is asserted
        else if (en)
            Q <= D ;  //Update q to d on clock edge
    end
endmodule
```

```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module Register_4Bit(input clk , input rst , input en , input [3:0] D , output [3:0] Q) ;
            Register_1Bit R0 (.clk(clk) , .rst(rst) , .en(en) , .D(D[0]) , .Q(Q[0])) ;
            Register_1Bit R1 (.clk(clk) , .rst(rst) , .en(en) , .D(D[1]) , .Q(Q[1])) ;
            Register_1Bit R2 (.clk(clk) , .rst(rst) , .en(en) , .D(D[2]) , .Q(Q[2])) ;
            Register_1Bit R3 (.clk(clk) , .rst(rst) , .en(en) , .D(D[3]) , .Q(Q[3])) ;
endmodule
```

```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module Register_12Bit(input clk , input rst , input en , input [11:0] D , output [11:0] Q) ;
            Register_4Bit R0 (.clk(clk) , .rst(rst) , .en(en) , .D(D[3:0]) , .Q(Q[3:0])) ;
            Register_4Bit R1 (.clk(clk) , .rst(rst) , .en(en) , .D(D[7:4]) , .Q(Q[7:4])) ;
            Register_4Bit R2 (.clk(clk) , .rst(rst) , .en(en) , .D(D[11:8]) , .Q(Q[11:8])) ;
endmodule
```

```
/* Structural Verilog model for a 1-Digit BCD Adder */
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.
```

```
module BCD_1Digit(input [3:0] A , input [3:0] B , input Cin , output [3:0] S , output Cout);
   wire [3:0] sum1 ;
   wire [3:0] sum2 ;
   wire carry1 ;
   wire carry2 ;
   wire invalid_bcd ;
   wire [3:0] correction = 4'b0110 ;  //6
   wire s1 ;
   wire s2 ;
   //First 4-bit adder(A+B+Cin):
   FA_4Bit Adder1 (.A(A) , .B(B) , .Cin(Cin) , .Sum(sum1) , .Cout(carry1)) ;
   //Check if sum > 9 or carry1 = 1:
   /* (sum1 > 1001) = sum1[3] & (sum1[2] | sum1[1]) */
   or #11 (s1 , sum1[2] , sum1[1]) ;
   and #11 (s2 , sum1[3] , s1) ;
   or #11 (invalid_bcd , carry1 , s2) ;
   //Second 4-bit adder for BCD correction(if invalid, add 6):
   FA_4Bit Adder2 (.A(sum1) , .B(correction) , .Cin(0) , .Sum(sum2) , .Cout(carry2)) ;
   //output:
   assign S = invalid_bcd ? sum2 : sum1 ;
   assign Cout = invalid_bcd ? carry2 : carry1 ;
endmodule
```

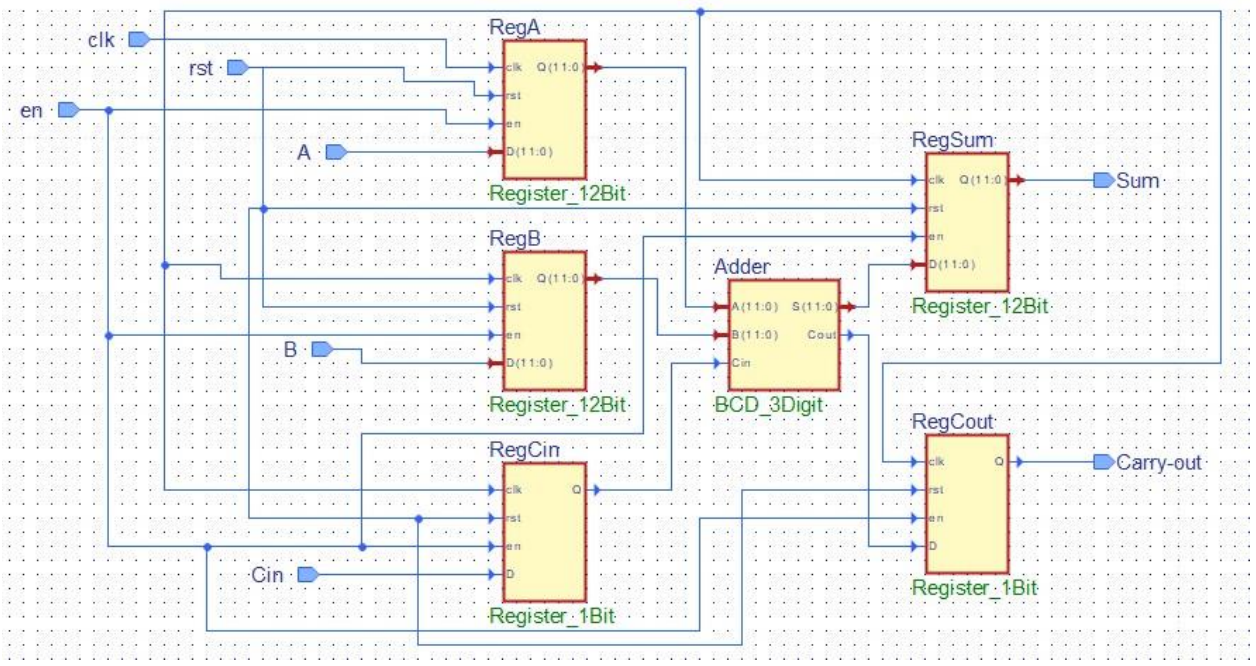
```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for delays and simulation time.


module BCD_3Digit(input [11:0] A , input [11:0] B , input Cin , output [11:0] S , output Cout) ;
```

```
    wire c1 ;

    wire c2 ;

    BCD_1Digit D0 (.A(A[3:0]) ,  .B(B[3:0]) ,  .Cin(Cin) , .S(S[3:0]) ,  .Cout(c1)) ;  //Digit
0(Least Significant)

    BCD_1Digit D1 (.A(A[7:4]) ,  .B(B[7:4]) ,  .Cin(c1) ,  .S(S[7:4]) ,  .Cout(c2)) ;  //Digit 1

    BCD_1Digit D2 (.A(A[11:8]) , .B(B[11:8]) , .Cin(c2) ,  .S(S[11:8]) , .Cout(Cout)) ;  //Digit
2(Most Significant)

endmodule
```

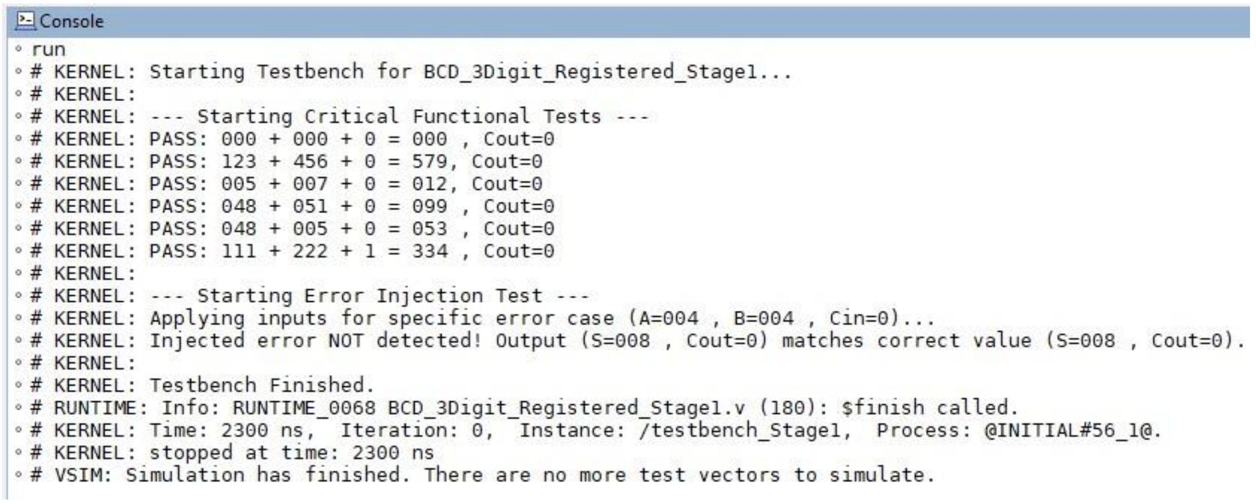
```
//3-digit BCD input A , 3-digit BCD input B and Carry in

//3-digit sum output and Carry out


`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module BCD_3Digit_Registered_Stage1 (input clk , input rst , input en , input [11:0] A , input
[11:0] B , input Cin , output [11:0] S , output Cout) ;

    wire [11:0] A_reg ;

    wire [11:0] B_reg ;

    wire Cin_reg ;

    wire [11:0] S_reg ;

    wire Cout_reg ;

    Register_12Bit RegA (.clk(clk) , .rst(rst) , .en(en) , .D(A) , .Q(A_reg)) ;

    Register_12Bit RegB (.clk(clk) , .rst(rst) , .en(en) , .D(B) , .Q(B_reg)) ;

    Register_1Bit RegCin (.clk(clk) , .rst(rst) , .en(en) , .D(Cin) , .Q(Cin_reg)) ;

    BCD_3Digit BCD_Adder (.A(A_reg) , .B(B_reg) , .Cin(Cin_reg) , .S(S) , .Cout(Cout)) ;  //3-
digit BCD addition

Register_12Bit RegS (.clk(clk), .rst(rst) , .en(en) , .D(S) ,   .Q(S_reg)) ;

Register_1Bit RegCout (.clk(clk) , .rst(rst) , .en(en) , .D(Cout) , .Q(Cout_reg)) ;

endmodule
```

```
//testbench:

module testbench_Stage1 ;

    parameter CLK_PERIOD = 100 ;  //Defines the clock period in nanoseconds.

    reg clk ;                          // Clock signal
    reg rst ;                         // Asynchronous reset signal (active high)
    reg en ;                           // Enable signal for registers
    reg [11:0] A_in ;                // 12-bit BCD input A (3 digits)
    reg [11:0] B_in ;                // 12-bit BCD input B (3 digits)
    reg Cin ;                          // 1-bit Carry-in
    wire [11:0] S_out ;              // 12-bit BCD sum output (3 digits)
    wire Cout ;                        // 1-bit Carry-out

    //an instance of registered 3-digit BCD adder module and connect each reg to input and each
wire to output:
    BCD_3Digit_Registered_Stage1 M (.clk(clk) , .rst(rst) , .en(en) , .A(A_in) , .B(B_in) ,
.Cin(Cin) , .S(S_out) , .Cout(Cout)) ;

    //--- Clock Generation ---
    //This 'initial' block runs once at the start of the simulation.
    initial
            begin
        clk = 1'b0 ;  //Initialize clock to 0.
        //'forever' loop creates a continuous clock signal.
        //# (CLK_PERIOD / 2) waits for half the clock period.
        //clk = ~clk; toggles the clock value (0 to 1, 1 to 0).
        forever #(CLK_PERIOD / 2) clk = ~clk ;
    end

    //--- Test Sequence and Verification Logic ---
    //This 'initial' block defines the sequence of actions for the test.
    initial
            begin
                        integer file_handle ;
                        //Display a starting message in the simulation console.
        $display("Starting Testbench for BCD_3Digit_Registered_Stage1...") ;

        //1. Initialize Inputs and Apply Reset
        rst = 1'b1 ;                //Assert the reset signal (make it high).
        en = 1'b0 ;                 //Keep the register enable low during reset.
        A_in = 12'h000 ;            //Set inputs to 'X' (unknown) during reset - good practice.
        B_in = 12'h000 ;
        Cin = 1'b0 ;
        //Wait for a couple of clock cycles while reset is active.
        # (CLK_PERIOD * 2) ;

        //2. Deassert Reset and Enable
        rst = 1'b0 ;                //Deassert the reset signal (make it low).
        en = 1'b1 ;                 //Assert the register enable (make it high).
        //Wait one full clock cycle for the reset signal change to take effect
        # (CLK_PERIOD * 2) ;

        //--- Functional Verification Test Cases ---
        $display("\n--- Starting Critical Functional Tests ---") ;
```

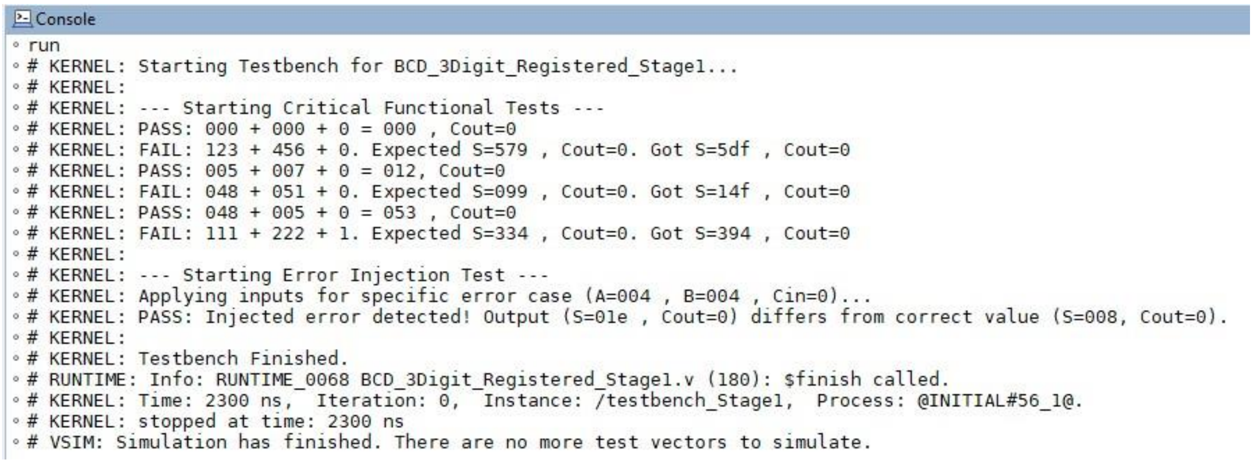
```
        //Test Case 1: Zero Test (000 + 000, Cin=0 -> S=000, Cout=0)

        //Apply inputs.

        A_in = 12'h000 ;

        B_in = 12'h000 ;

        Cin = 1'b0 ;

        //Wait one full clock cycle for the inputs to be registered and the result to propagate through
the combinational logic and be captured by the output registers.

        # (CLK_PERIOD * 2) ;

        //Check the registered outputs against the expected values.

        //Use '===' (case equality) for comparison, which also checks for X and Z values.

        if (S_out === 12'h000 && Cout === 1'b0)

                        $display("PASS: 000 + 000 + 0 = %h , Cout=%b" , S_out , Cout) ;

        else

                        $display("FAIL: 000 + 000 + 0. Expected S=000 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 2: Simple Addition (123 + 456, Cin=0 -> S=579, Cout=0)

        A_in = 12'h123 ;

        B_in = 12'h456 ;

        Cin = 1'b0 ;

        # (CLK_PERIOD * 2) ; //Wait for result

        if (S_out === 12'h579 && Cout === 1'b0)

                        $display("PASS: 123 + 456 + 0 = %h, Cout=%b" , S_out , Cout) ;

        else

                        $display("FAIL: 123 + 456 + 0. Expected S=579 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 3: Single Digit Correction (005 + 007, Cin=0 -> S=012, Cout=0)
        //Tests if the BCD correction logic (add 6 when sum > 9) works in the least significant digit.

        A_in = 12'h005 ;

        B_in = 12'h007 ;

        Cin = 1'b0 ;

        # (CLK_PERIOD * 2) ; //Wait for result

        if (S_out === 12'h012 && Cout === 1'b0)

                        $display("PASS: 005 + 007 + 0 = %h, Cout=%b" , S_out , Cout) ;

        else

                        $display("FAIL: 005 + 007 + 0. Expected S=012 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 4: Carry Propagation (No Correction) (048 + 051, Cin=0 -> S=099, Cout=0)

        //Tests carry propagation between digits without needing BCD correction.

        A_in = 12'h048 ;

        B_in = 12'h051 ;

        Cin = 1'b0 ;

        # (CLK_PERIOD * 2) ; //Wait for result

        if (S_out === 12'h099 && Cout === 1'b0)

                        $display("PASS: 048 + 051 + 0 = %h , Cout=%b" , S_out , Cout) ;

        else

                        $display("FAIL: 048 + 051 + 0. Expected S=099 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 5: Carry Propagation with Correction (Digit 0) (048 + 005, Cin=0 -> S=053,
Cout=0)

        //Tests interaction: Digit 0 needs correction (8+5=13 -> BCD 3 carry 1), carry propagates to
Digit 1.

        A_in = 12'h048 ;

        B_in = 12'h005;

        Cin = 1'b0;
        # (CLK_PERIOD * 2); //Wait for result
        if (S_out === 12'h053 && Cout === 1'b0)
                                $display("PASS: 048 + 005 + 0 = %h , Cout=%b" , S_out , Cout) ;
        else
                                $display("FAIL: 048 + 005 + 0. Expected S=053 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 6: Test with Cin (111 + 222, Cin=1 -> S=334, Cout=0)
        //Simple case testing the initial carry-in.
        A_in = 12'h111 ;
        B_in = 12'h222 ;
       Cin = 1'b1 ;
        # (CLK_PERIOD * 2) ; //Wait for result
        if (S_out === 12'h334 && Cout === 1'b0)
                                $display("PASS: 111 + 222 + 1 = %h , Cout=%b" , S_out , Cout) ;
        else
                                $display("FAIL: 111 + 222 + 1. Expected S=334 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;



        //--- Error Injection Test Case ---
        //This test case targets the specific injected error:
        //In BCD_1Digit.v, reversed OR and AND gates.
        //Test Input: A=004, B=004, Cin=0
        //Expected CORRECT Output: S=008, Cout=0
        //Expected FAULTY Output (due to incorrect correction): S=014 (invalid BCD), Cout=1
(incorrect)
        //We check if the output differs from the CORRECT output.
        $display("\n--- Starting Error Injection Test ---") ;
        $display("Applying inputs for specific error case (A=004 , B=004 , Cin=0)...") ;
        A_in = 12'h004 ;
        B_in = 12'h004 ;
        Cin = 1'b0 ;
        # (CLK_PERIOD * 2) ; //Wait for result

        //Check if the output DIFFERS from the known correct output (S=008, Cout=0)
        if (S_out !== 12'h008 || Cout !== 1'b0)
                begin
          $display("PASS: Injected error detected! Output (S=%h , Cout=%b) differs from correct
value (S=008, Cout=0)." , S_out , Cout) ;
          file_handle = $fopen("error_report.txt" , "w") ; //Open file for writing
          $fdisplay(file_handle , "Error detected: Incorrect gate: \nOR instead of AND: or #11 (s2 ,
sum1[3] , s1) ; \nand AND instead of OR: and #11 (s1 , sum1[2] , sum1[1]) ; \nin BCD_1Digit.v
caught by testbench.") ; //Write message
          $fclose(file_handle) ; //Close file
        end
                else
                begin
          $display("Injected error NOT detected! Output (S=%h , Cout=%b) matches correct value
(S=008 , Cout=0)." , S_out , Cout) ;
        end

        //--- End Simulation ---
        $display("\nTestbench Finished.") ;
        //Wait a few more cycles to allow final messages to print before ending.
        # (CLK_PERIOD * 5) ;
        $finish ; //Verilog system task to end the simulation.
    end
endmodule
```



Stage 1 Block diagram

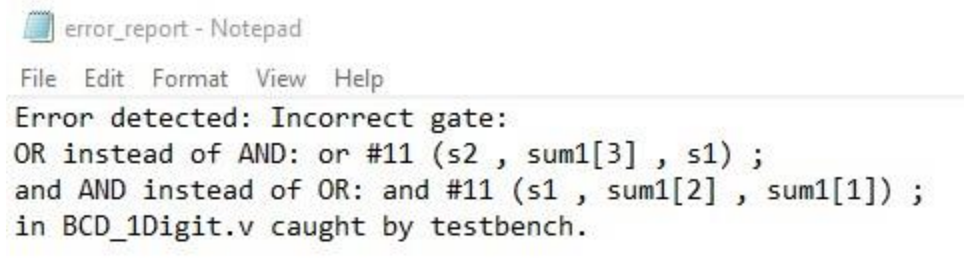

```
Console
◦ run
◦ # KERNEL: Starting Testbench for BCD_3Digit_Registered_Stage1...
◦ # KERNEL:
◦ # KERNEL: --- Starting Critical Functional Tests ---
◦ # KERNEL: PASS: 000 + 000 + 0 = 000 , Cout=0
◦ # KERNEL: PASS: 123 + 456 + 0 = 579, Cout=0
◦ # KERNEL: PASS: 005 + 007 + 0 = 012, Cout=0
◦ # KERNEL: PASS: 048 + 051 + 0 = 099 , Cout=0
◦ # KERNEL: PASS: 048 + 005 + 0 = 053 , Cout=0
◦ # KERNEL: PASS: 111 + 222 + 1 = 334 , Cout=0
◦ # KERNEL:
◦ # KERNEL: --- Starting Error Injection Test ---
◦ # KERNEL: Applying inputs for specific error case (A=004 , B=004 , Cin=0)...
◦ # KERNEL: Injected error NOT detected! Output (S=008 , Cout=0) matches correct value (S=008 , Cout=0).
◦ # KERNEL:
◦ # KERNEL: Testbench Finished.
◦ # RUNTIME: Info: RUNTIME_0068 BCD_3Digit_Registered_Stage1.v (180): $finish called.
◦ # KERNEL: Time: 2300 ns,  Iteration: 0,  Instance: /testbench_Stage1,  Process: @INITIAL#56_1@.
◦ # KERNEL: stopped at time: 2300 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Stage 1 output



```
◦ run
◦ # KERNEL: Starting Testbench for BCD_3Digit_Registered_Stage1...
◦ # KERNEL:
◦ # KERNEL: --- Starting Critical Functional Tests ---
◦ # KERNEL: PASS: 000 + 000 + 0 = 000 , Cout=0
◦ # KERNEL: FAIL: 123 + 456 + 0. Expected S=579 , Cout=0. Got S=5df , Cout=0
◦ # KERNEL: PASS: 005 + 007 + 0 = 012, Cout=0
◦ # KERNEL: FAIL: 048 + 051 + 0. Expected S=099 , Cout=0. Got S=14f , Cout=0
◦ # KERNEL: PASS: 048 + 005 + 0 = 053 , Cout=0
◦ # KERNEL: FAIL: 111 + 222 + 1. Expected S=334 , Cout=0. Got S=394 , Cout=0
◦ # KERNEL:
◦ # KERNEL: --- Starting Error Injection Test ---
◦ # KERNEL: Applying inputs for specific error case (A=004 , B=004 , Cin=0)...
◦ # KERNEL: PASS: Injected error detected! Output (S=01e , Cout=0) differs from correct value (S=008, Cout=0).
◦ # KERNEL:
◦ # KERNEL: Testbench Finished.
◦ # RUNTIME: Info: RUNTIME_0068 BCD_3Digit_Registered_Stage1.v (180): $finish called.
◦ # KERNEL: Time: 2300 ns,  Iteration: 0,  Instance: /testbench_Stage1,  Process: @INITIAL#56_1@.
◦ # KERNEL: stopped at time: 2300 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Stage 1 output after Injecting error



```
Error detected: Incorrect gate:
OR instead of AND: or #11 (s2 , sum1[3] , s1) ;
and AND instead of OR: and #11 (s1 , sum1[2] , sum1[1]) ;
in BCD_1Digit.v caught by testbench.
```

error_report.txt contents in stage 1

```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for delays and simulation time.


module CLA_4Bit (input [3:0] A , input [3:0] B , input Cin , output [3:0] Sum , output Cout) ;
   wire [3:0] G ;    //Generate
   wire [3:0] P ;    //Propagate
   wire [3:1] C ;    //Internal carries
   wire P0Cin ;
   wire P1G0 ;
   wire P1P0 ;
   wire P1P0Cin ;
   wire P2G1 ;
   wire P2P1 ;
wire P2P1G0 ;
wire P2P1P0 ;
wire P2P1P0Cin ;
wire P3G2 ;
wire P3P2 ;
wire P3P2P1 ;
wire P3P2P1G0 ;
wire P3P2P1P0 ;
wire P3P2P1P0Cin ;
wire P3P2G1 ;
//Propagate (P = A ^ B):
xor #15 (P[0] , A[0] , B[0]) ;
xor #15 (P[1] , A[1] , B[1]) ;
xor #15 (P[2] , A[2] , B[2]) ;
xor #15 (P[3] , A[3] , B[3]) ;
//Generate (G = A & B):
and #11 (G[0] , A[0] , B[0]) ;
and #11 (G[1] , A[1] , B[1]) ;
and #11 (G[2] , A[2] , B[2]) ;
and #11 (G[3] , A[3] , B[3]) ;
//Carry C1 = G0 | (P0 & Cin):
and #11 (P0Cin , P[0] , Cin) ;
or  #11 (C[1] , G[0] , P0Cin) ;
//Carry C2 = G1 | (P1 & G0) | (P1 & P0 & Cin):
and #11 (P1G0 , P[1] , G[0]) ;
and #11 (P1P0 , P[1] , P[0]) ;
and #11 (P1P0Cin , P1P0 , Cin) ;
or  #11 (C[2] , G[1] , P1G0 , P1P0Cin) ;
```

```
    //Carry C3 = G2 | (P2 & G1) | (P2 & P1 & G0) | (P2 & P1 & P0 & Cin):
    and #11 (P2G1 , P[2] , G[1]) ;
    and #11 (P2P1 , P[2] , P[1]) ;
    and #11 (P2P1G0 , P2P1 , G[0]) ;
    and #11 (P2P1P0 , P2P1 , P[0]) ;
    and #11 (P2P1P0Cin , P2P1P0 , Cin) ;
    or  #11 (C[3] , G[2] , P2G1 , P2P1G0 , P2P1P0Cin) ;
    //Carry-out = G3 | (P3 & G2) | (P3 & P2 & G1) | (P3 & P2 & P1 & G0) | (P3 & P2 & P1 & P0
& Cin):
    and #11 (P3G2 , P[3] , G[2]) ;
    and #11 (P3P2 , P[3] , P[2]) ;
    and #11 (P3P2G1 , P3P2 , G[1]) ;
    and #11 (P3P2P1 , P3P2 , P[1]) ;
    and #11 (P3P2P1G0 , P3P2P1 , G[0]) ;
    and #11 (P3P2P1P0 , P3P2P1 , P[0]) ;
    and #11 (P3P2P1P0Cin , P3P2P1P0 , Cin) ;
    or  #11 (Cout , G[3] , P3G2 , P3P2G1 , P3P2P1G0 , P3P2P1P0Cin) ;
    //Sum = P ^ C:
    xor #15 (Sum[0] , P[0] , Cin) ;
    xor #15 (Sum[1] , P[1] , C[1]) ;
    xor #15 (Sum[2] , P[2] , C[2]) ;
    xor #15 (Sum[3] , P[3] , C[3]) ;
endmodule
```

```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module BCD_1Digit_CLA (input  [3:0] A , input  [3:0] B , input Cin , output [3:0] S , output
Cout) ;
```

```
   wire [3:0] sum1 ;          //Initial sum
   wire [3:0] sum_corrected ;  //Sum after BCD correction
   wire carry1 ;              //Carry from first CLA adder
   wire carry2 ;              //Carry from correction adder
   wire invalid_bcd ;         //Flag: sum > 9 or carry1
   wire [3:0] correction = 4'b0110 ;  // +6 in binary
   wire check_high ;  //Intermediate wires for invalid BCD check
   wire check_bcd ;
   //First CLA adder(A + B + Cin):
   CLA_4Bit Adder1 (.A(A) , .B(B) , .Cin(Cin) , .Sum(sum1) , .Cout(carry1)) ;
   //Check if sum1 > 9:
   or  #11 (check_high , sum1[2] , sum1[1]) ;
   and #11 (check_bcd , sum1[3] , check_high) ;
   or  #11 (invalid_bcd , carry1 , check_bcd) ;
   //Second CLA adder(sum1 + 6):
   CLA_4Bit Adder2 (.A(sum1) , .B(correction) , .Cin(1'b0), .Sum(sum_corrected) ,
.Cout(carry2)) ;
   //output:
   assign S = invalid_bcd ? sum_corrected : sum1 ;
   assign Cout  = invalid_bcd ? carry2 : carry1 ;
endmodule
```

```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module BCD_3Digit_CLA(input [11:0] A , input [11:0] B , input Cin , output [11:0] S , output
Cout) ;
   wire c1 ;
   wire c2 ;
```

```
    BCD_1Digit_CLA D0 (.A(A[3:0]) ,  .B(B[3:0]) ,  .Cin(Cin) , .S(S[3:0]) ,  .Cout(c1)) ;  //Digit
0(Least Significant)
    BCD_1Digit_CLA D1 (.A(A[7:4]) ,  .B(B[7:4]) ,  .Cin(c1) ,  .S(S[7:4]) ,  .Cout(c2)) ;  //Digit 1
    BCD_1Digit_CLA D2 (.A(A[11:8]) , .B(B[11:8]) , .Cin(c2) ,  .S(S[11:8]) , .Cout(Cout)) ;
//Digit 2(Most Significant)
endmodule
```

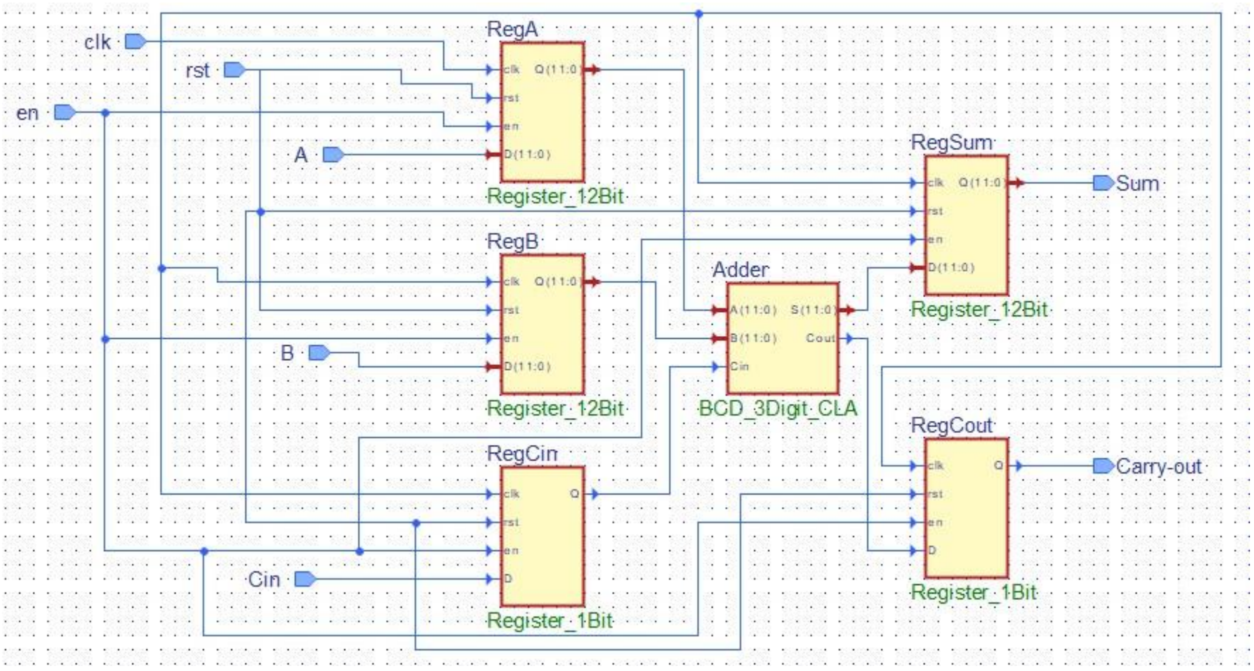
```
`timescale 1ns / 1ps  //timescale directive: Specifies the time unit (1ns) and precision (1ps) for
delays and simulation time.


module BCD_3Digit_Registered_Stage2 (input clk , input rst , input en , input [11:0] A , input
[11:0] B , input Cin , output [11:0] S , output Cout) ;
    wire [11:0] A_reg ;
    wire [11:0] B_reg ;
    wire Cin_reg ;
    wire [11:0] S_reg ;
    wire Cout_reg ;
    Register_12Bit RegA (.clk(clk), .rst(rst) , .en(en) , .D(A) , .Q(A_reg)) ;
    Register_12Bit RegB (.clk(clk) , .rst(rst) , .en(en), .D(B), .Q(B_reg)) ;
    Register_1Bit  RegCin (.clk(clk) , .rst(rst) , .en(en) , .D(Cin) , .Q(Cin_reg)) ;
    BCD_3Digit_CLA Adder (.A(A_reg) , .B(B_reg) , .Cin(Cin_reg) , .S(S) , .Cout(Cout)) ;
    Register_12Bit RegS (.clk(clk), .rst(rst) , .en(en) , .D(S) , .Q(S_reg)) ;
    Register_1Bit RegCout (.clk(clk) , .rst(rst) , .en(en) , .D(Cout) , .Q(Cout_reg)) ;
endmodule


//testbench:


module testbench_Stage2 ;
```

```
   parameter CLK_PERIOD = 100 ;  //Defines the clock period in nanoseconds.


   reg clk ;                       // Clock signal

   reg rst ;                       // Asynchronous reset signal (active high)

   reg en ;                        // Enable signal for registers

   reg [11:0] A_in ;               // 12-bit BCD input A (3 digits)

   reg [11:0] B_in ;               // 12-bit BCD input B (3 digits)

   reg Cin ;                       // 1-bit Carry-in

   wire [11:0] S_out ;             // 12-bit BCD sum output (3 digits)

   wire Cout ;                     // 1-bit Carry-out


   //an instance of registered 3-digit BCD adder module and connect each reg to input and each
wire to output:

   BCD_3Digit_Registered_Stage2 M (.clk(clk) , .rst(rst) , .en(en) , .A(A_in) , .B(B_in) ,
.Cin(Cin) , .S(S_out) , .Cout(Cout)) ;


   //--- Clock Generation ---

   //This 'initial' block runs once at the start of the simulation.

   initial

        begin

      clk = 1'b0 ;  //Initialize clock to 0.

      //'forever' loop creates a continuous clock signal.

      //# (CLK_PERIOD / 2) waits for half the clock period.

      //clk = ~clk; toggles the clock value (0 to 1, 1 to 0).

      forever #(CLK_PERIOD / 2) clk = ~clk ;

   end


   //--- Test Sequence and Verification Logic ---

   //This 'initial' block defines the sequence of actions for the test.
```

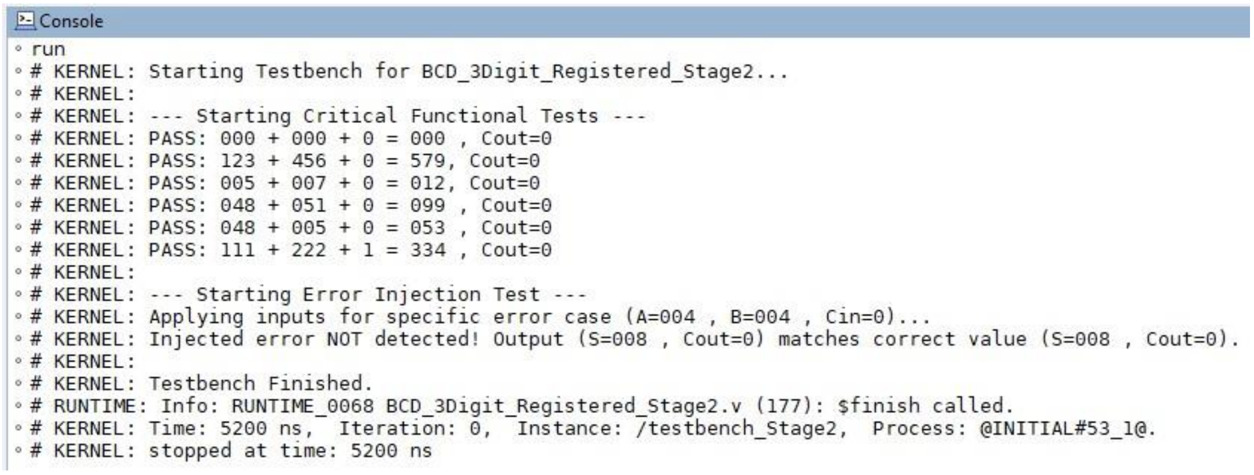
```
initial
      begin
               integer file_handle ;
               //Display a starting message in the simulation console.
   $display("Starting Testbench for BCD_3Digit_Registered_Stage2...") ;

   //1. Initialize Inputs and Apply Reset
   rst = 1'b1 ;             //Assert the reset signal (make it high).
   en = 1'b0 ;              //Keep the register enable low during reset.
   A_in = 12'h000 ;           //Set inputs to 'X' (unknown) during reset - good practice.
   B_in = 12'h000 ;
   Cin = 1'b0 ;
   //Wait for a couple of clock cycles while reset is active.
   # (CLK_PERIOD * 13) ;

   //2. Deassert Reset and Enable
   rst = 1'b0 ;             //Deassert the reset signal (make it low).
   en = 1'b1 ;              //Assert the register enable (make it high).
   //Wait one full clock cycle for the reset signal change to take effect
   # (CLK_PERIOD * 13) ;

   //--- Functional Verification Test Cases ---
   $display("\n--- Starting Critical Functional Tests ---") ;

   //Test Case 1: Zero Test (000 + 000, Cin=0 -> S=000, Cout=0)
   //Apply inputs.
   A_in = 12'h000 ;
   B_in = 12'h000 ;
```

```
        Cin = 1'b0 ;

        //Wait one full clock cycle for the inputs to be registered and the result to propagate through
the combinational logic and be captured by the output registers.

        # (CLK_PERIOD * 3) ;

        //Check the registered outputs against the expected values.

        //Use '===' (case equality) for comparison, which also checks for X and Z values.

        if (S_out === 12'h000 && Cout === 1'b0)

                                $display("PASS: 000 + 000 + 0 = %h , Cout=%b" , S_out , Cout) ;

        else

                                $display("FAIL: 000 + 000 + 0. Expected S=000 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 2: Simple Addition (123 + 456, Cin=0 -> S=579, Cout=0)

        A_in = 12'h123 ;

        B_in = 12'h456 ;

        Cin = 1'b0 ;

        # (CLK_PERIOD * 3) ; //Wait for result

        if (S_out === 12'h579 && Cout === 1'b0)

                                $display("PASS: 123 + 456 + 0 = %h, Cout=%b" , S_out , Cout) ;

        else

                                $display("FAIL: 123 + 456 + 0. Expected S=579 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 3: Single Digit Correction (005 + 007, Cin=0 -> S=012, Cout=0)

        //Tests if the BCD correction logic (add 6 when sum > 9) works in the least significant digit.

        A_in = 12'h005 ;

        B_in = 12'h007 ;

        Cin = 1'b0 ;

        # (CLK_PERIOD * 3) ; //Wait for result
        if (S_out === 12'h012 && Cout === 1'b0)
                                $display("PASS: 005 + 007 + 0 = %h, Cout=%b" , S_out , Cout) ;
        else
                                $display("FAIL: 005 + 007 + 0. Expected S=012 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 4: Carry Propagation (No Correction) (048 + 051, Cin=0 -> S=099, Cout=0)
        //Tests carry propagation between digits without needing BCD correction.
        A_in = 12'h048 ;
        B_in = 12'h051 ;
        Cin = 1'b0 ;
        # (CLK_PERIOD * 3) ; //Wait for result
        if (S_out === 12'h099 && Cout === 1'b0)
                                $display("PASS: 048 + 051 + 0 = %h , Cout=%b" , S_out , Cout) ;
        else
                                $display("FAIL: 048 + 051 + 0. Expected S=099 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 5: Carry Propagation with Correction (Digit 0) (048 + 005, Cin=0 -> S=053,
Cout=0)
        //Tests interaction: Digit 0 needs correction (8+5=13 -> BCD 3 carry 1), carry propagates to
Digit 1.
        A_in = 12'h048 ;
        B_in = 12'h005;
        Cin = 1'b0;
        # (CLK_PERIOD * 3) ; //Wait for result
        if (S_out === 12'h053 && Cout === 1'b0)
                                $display("PASS: 048 + 005 + 0 = %h , Cout=%b" , S_out , Cout) ;
        else
```

```
                                $display("FAIL: 048 + 005 + 0. Expected S=053 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;


        //Test Case 6: Test with Cin (111 + 222, Cin=1 -> S=334, Cout=0)
        //Simple case testing the initial carry-in.
        A_in = 12'h111 ;
        B_in = 12'h222 ;
        Cin = 1'b1 ;
        # (CLK_PERIOD * 3) ; //Wait for result
        if (S_out === 12'h334 && Cout === 1'b0)
                                $display("PASS: 111 + 222 + 1 = %h , Cout=%b" , S_out , Cout) ;
        else
                                $display("FAIL: 111 + 222 + 1. Expected S=334 , Cout=0. Got S=%h ,
Cout=%b" , S_out , Cout) ;



        //--- Error Injection Test Case ---
        //This test case targets the specific injected error:
        //In CLA_4Bit.v, reversed AND and OR gates.
        //Test Input: A=004, B=004, Cin=0
        //Expected CORRECT Output: S=008, Cout=0
        //Expected FAULTY Output (due to incorrect correction): S=014 (invalid BCD), Cout=1
(incorrect)
        //We check if the output differs from the CORRECT output.
        $display("\n--- Starting Error Injection Test ---") ;
        $display("Applying inputs for specific error case (A=004 , B=004 , Cin=0)...") ;
        A_in = 12'h004 ;
        B_in = 12'h004 ;
        Cin = 1'b0 ;
        # (CLK_PERIOD * 3) ; //Wait for result

        //Check if the output DIFFERS from the known correct output (S=008, Cout=0)

        if (S_out !== 12'h008 || Cout !== 1'b0)

                    begin

            $display("PASS: Injected error detected! Output (S=%h , Cout=%b) differs from correct
value (S=008, Cout=0)." , S_out , Cout) ;

            file_handle = $fopen("error_report.txt" , "w") ; //Open file for writing

            $fdisplay(file_handle , "Error detected: Incorrect gate: OR instead of AND: \nor #11
(P1G0 , P[1] , G[0]) ;\nor #11 (P1P0 , P[1] , P[0]) ;\nor #11 (P1P0Cin , P1P0 , Cin) ;\nand AND
instead of OR: \nand  #11 (C[2] , G[1] , P1G0 , P1P0Cin) ; \nand  #11 (C[3] , G[2] , P2G1 ,
P2P1G0 , P2P1P0Cin) ; \nand  #11 (Cout , G[3] , P3G2 , P3P2G1 , P3P2P1G0 , P3P2P1P0Cin) ;
\nin CLA_4Bit.v caught by testbench.") ; //Write message

            $fclose(file_handle) ; //Close file

        end

                    else

                    begin

            $display("Injected error NOT detected! Output (S=%h , Cout=%b) matches correct value
(S=008 , Cout=0)." , S_out , Cout) ;

        end




        //--- End Simulation ---

        $display("\nTestbench Finished.") ;

        //Wait a few more cycles to allow final messages to print before ending.

        # (CLK_PERIOD * 5) ;

        $finish ; //Verilog system task to end the simulation.

    end

endmodule
```

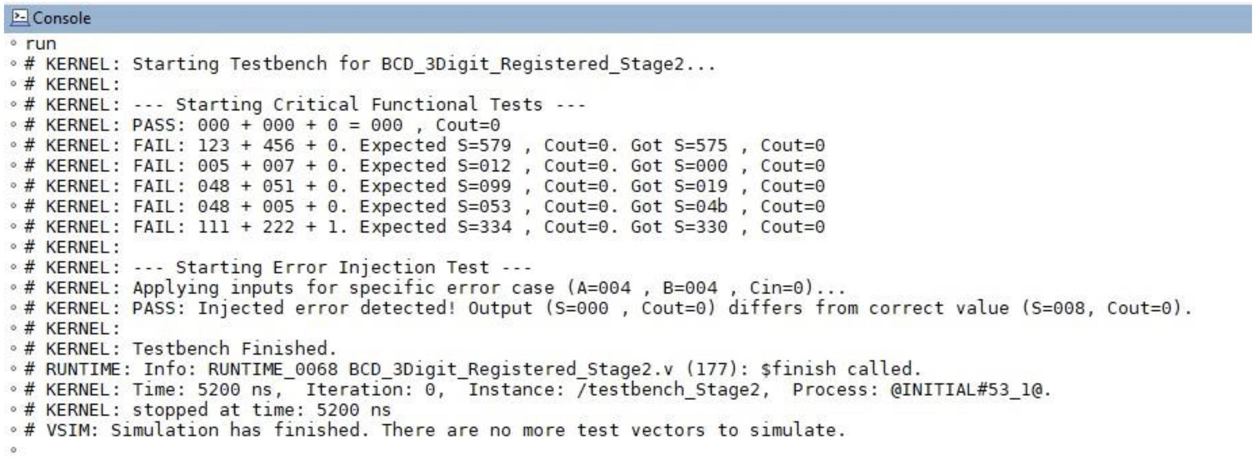


Stage 2 Block diagram


Console

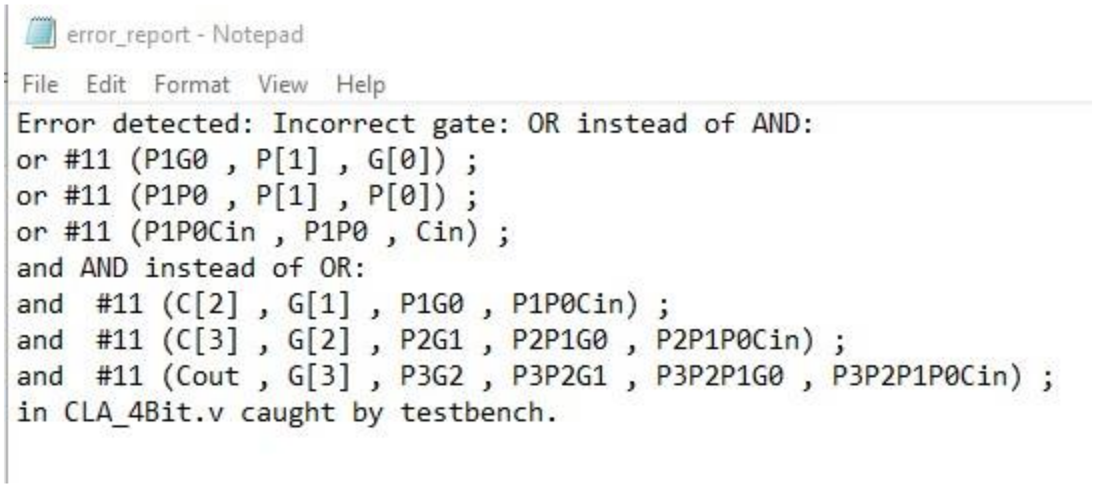
```
◦ run
◦ # KERNEL: Starting Testbench for BCD_3Digit_Registered_Stage2...
◦ # KERNEL:
◦ # KERNEL: --- Starting Critical Functional Tests ---
◦ # KERNEL: PASS: 000 + 000 + 0 = 000 , Cout=0
◦ # KERNEL: PASS: 123 + 456 + 0 = 579, Cout=0
◦ # KERNEL: PASS: 005 + 007 + 0 = 012, Cout=0
◦ # KERNEL: PASS: 048 + 051 + 0 = 099 , Cout=0
◦ # KERNEL: PASS: 048 + 005 + 0 = 053 , Cout=0
◦ # KERNEL: PASS: 111 + 222 + 1 = 334 , Cout=0
◦ # KERNEL:
◦ # KERNEL: --- Starting Error Injection Test ---
◦ # KERNEL: Applying inputs for specific error case (A=004 , B=004 , Cin=0)...
◦ # KERNEL: Injected error NOT detected! Output (S=008 , Cout=0) matches correct value (S=008 , Cout=0).
◦ # KERNEL:
◦ # KERNEL: Testbench Finished.
◦ # RUNTIME: Info: RUNTIME_0068 BCD_3Digit_Registered_Stage2.v (177): $finish called.
◦ # KERNEL: Time: 5200 ns,  Iteration: 0,  Instance: /testbench_Stage2,  Process: @INITIAL#53_1@.
◦ # KERNEL: stopped at time: 5200 ns
```

Stage 2 output



```
Console
run
# KERNEL: Starting Testbench for BCD_3Digit_Registered_Stage2...
# KERNEL:
# KERNEL: --- Starting Critical Functional Tests ---
# KERNEL: PASS: 000 + 000 + 0 = 000 , Cout=0
# KERNEL: FAIL: 123 + 456 + 0. Expected S=579 , Cout=0. Got S=575 , Cout=0
# KERNEL: FAIL: 005 + 007 + 0. Expected S=012 , Cout=0. Got S=000 , Cout=0
# KERNEL: FAIL: 048 + 051 + 0. Expected S=099 , Cout=0. Got S=019 , Cout=0
# KERNEL: FAIL: 048 + 005 + 0. Expected S=053 , Cout=0. Got S=04b , Cout=0
# KERNEL: FAIL: 111 + 222 + 1. Expected S=334 , Cout=0. Got S=330 , Cout=0
# KERNEL:
# KERNEL: --- Starting Error Injection Test ---
# KERNEL: Applying inputs for specific error case (A=004 , B=004 , Cin=0)...
# KERNEL: PASS: Injected error detected! Output (S=000 , Cout=0) differs from correct value (S=008, Cout=0).
# KERNEL:
# KERNEL: Testbench Finished.
# RUNTIME: Info: RUNTIME_0068 BCD_3Digit_Registered_Stage2.v (177): $finish called.
# KERNEL: Time: 5200 ns,  Iteration: 0,  Instance: /testbench_Stage2,  Process: @INITIAL#53_1@.
# KERNEL: stopped at time: 5200 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Stage 2 output after injecting error



```
error_report - Notepad
File  Edit  Format  View  Help
Error detected: Incorrect gate: OR instead of AND:
or #11 (P1G0 , P[1] , G[0]) ;
or #11 (P1P0 , P[1] , P[0]) ;
or #11 (P1P0Cin , P1P0 , Cin) ;
and AND instead of OR:
and  #11 (C[2] , G[1] , P1G0 , P1P0Cin) ;
and  #11 (C[3] , G[2] , P2G1 , P2P1G0 , P2P1P0Cin) ;
and  #11 (Cout , G[3] , P3G2 , P3P2G1 , P3P2P1G0 , P3P2P1P0Cin) ;
in CLA_4Bit.v caught by testbench.
```

error_report.txt contents in stage 2

Test case	A	B	Cin	purpose
1	000	000	0	Zero Case: Verifies basic no-operation behavior.
2	123	456	0	Normal Addition: Ensures mid-range values are handled correctly.
3	005	007	0	LSB Correction: Tests BCD correction in the least significant digit.
4	048	051	0	Carry Without Correction: Carry propagation without needing fix.
5	048	005	0	Carry With Correction: Tests both correction and propagation.
6	111	222	1	Carry-in Test: Verifies handling of an initial carry-in.
7	004	004	0	Injected Error: Detects fault due to intentional gate logic error.

Test cases table