

FACULTY OF ENGINEERING AND TECHNOLOGY ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT ADVANCED DIGITAL DESIGN ENCS533 COURSE PROJECT

Student: Yanal Omar 1230156

Instructor: Dr. Abdellatif Abu-Issa

Date: June, 6th, 2025

## Abstract:

This project focuses on designing and testing an n digit BCD adder. The goal is to build a system that can add decimal numbers using digital logic in a way that is easy to expand. In this project, we implement a 3 digit version as an example. The design is done in two stages. In Stage 1, we use ripple carry adders built from basic gates, along with correction logic to make sure the result follows BCD rules. We test the design using a Verilog testbench that checks all input cases, measures the delay, and calculates the highest clock speed that can be used. We also add an error on purpose to see if the system can catch and report it. In Stage 2, we improve the design by using faster carry lookahead adders. We compare both stages to understand the difference between a simple and a faster design. This project shows how to build and check a working BCD adder that can be used for larger decimal numbers.

## Table Of Contents

1. [Abstract](#_Abstract:)………………………………………………………………………..2
2. [Table of Contents](#_Table_Of_Contents)……………………………………………………………...3
3. [Introduction and Background](#_Introduction_and_Background)………………………………………………….4
4. [Design Philosophy](#_Design_Philosophy)……………………………………………………………..5

4.1- [Full Adder Construction and Gate Delays](#_4.1_Full_Adder)………………………………………………5

4.2- [Stage 1: Ripple Carry-Based BCD Adder](#_4.2_Stage_1:)…………………………………………………5

4.3- [Stage 2: Carry Lookahead-Based BCD Adder](#_4.3_Stage_2:)………………………………………..6

4.4- [n-Digit BCD Adder](#_4.4_n-Digit_BCD)…………………………………………………….....6

4.5- [Verification and Error Detection](#_4.5_Verification_and)...……………………………...………...7

5- [Results](#_Result)…………………………………………………………………..………7

1. [Future Work](#_Future_Work)………………………………………………………………...…8
2. [Appendix](#_Appendix)………………………………………………………………………9

## Introduction and Background

In many digital systems, we need to work with decimal numbers, one common way to represent these numbers is using Binary Coded Decimal (BCD). In BCD, each decimal digit (0–9) is written using four binary bits. For example, the number "47" would be stored as two separate 4-bit values: 0100 for "4" and 0111 for "7".

Adding BCD numbers is different from normal binary addition. After adding two digits, if the result is greater than 9 or if a carry is produced, we must add 6 (binary 0110) to fix the result and make it a valid BCD digit.

This project is about designing and testing an *n*-digit BCD adder, with a special focus on a 3-digit version. The adder is built and tested in two stages:

* **Stage 1** uses a simple method called the ripple carry adder, where the carry moves from one digit to the next step-by-step.
* **Stage 2** uses a faster method called the carry lookahead adder, which calculates the carry faster by looking at groups of bits together.

The main goals of the project are:

* To check that the 3-digit BCD adder works correctly,
* To measure how long the adder takes to give a result, and calculate the fastest clock speed it can use,

This project shows how BCD adders can be built using basic logic gates, and how we can test and improve their performance through simulation.

## Design Philosophy

This project is based on designing an n-digit Binary Coded Decimal (BCD)adder, built structurally using basic logic gates. The architecture is modular and scalable, allowing the number of BCD digits to be adjusted by changing a single parameter (N). To verify the design, we made testing on a 3-digit adder, N=3, but the design supports any number of digits.

### 4.1 Full Adder Construction and Gate Delays

At the core of the design is a 1-bit full adder built entirely from basic gates with specified delays. This structural approach allows for accurate timing analysis. The gate delays used are XOR: 15ns, AND: 11ns, OR: 11ns.

The full adder is implemented using:

* Two XOR gates (for the sum)
* Two AND gates and one OR gate (for the carry)

These 1-bit adders are connected in series to form a **4-bit ripple carry adder**, which is then used in the BCD logic.

#### [Code Snippet 1: Full Adder](#_1-_Full_Adder)

#### [Code Snippet 2: 4-bit Ripple Adder](#_2-_Ripple_Carry)

### 4.2 Stage 1: Ripple Carry-Based BCD Adder

Each BCD digit is processed using two ripple carry adders:

1. The first adder computes the binary sum of the two 4-bit BCD inputs and an input carry.
2. If the sum is invalid (greater than 9 or carry-out = 1), a correction value of 6 (4'b0110) is added using a second ripple carry adder.

Multiple digit-level BCD adders are chained together in a generate loop, with the carry from one digit passed to the next. The number of digits is defined by the parameter N. Registers controlled by a clock and reset are used to store inputs and outputs at every cycle.

#### [Code Snippet 3: BCD Ripple Adder](#_4-_BCD_Ripple)

### 4.3 Stage 2: Carry Lookahead-Based BCD Adder

In Stage 2, each ripple carry adder is replaced by a carry lookahead adder (CLA) for faster performance. Carry lookahead logic computes all internal carries in parallel using generate (G) and propagate (P) signals:

#### [Code Snippet 4: Carry Look Ahead Adder](#_3-_Carry_Lookahead)

Intermediate carries and final Cout are derived through AND/OR logic with proper delays, reducing total latency.

BCD correction is applied the same way as in Stage 1, if the result exceeds 9, a second CLA adds 6. The structure remains modular, with each digit handled independently, and all digit adders connected via a carry chain.

#### [Code Snippet 5: BCD CLA Adder](#_5-_BCD_CLA)

### 4.4 n-Digit BCD Adder

The top module n\_digit\_bcd\_adder takes two n-digit BCD inputs (A\_in, B\_in) and an input carry. It:

* Registers the inputs using flip-flops on the rising edge of the clock
* Passes each 4-bit digit to its respective BCD digit adder
* Collects the sum from all digits and registers the output

The number of digits is defined by parameter N, allowing the adder to be scaled to any size without modifying the internal logic.

#### [Code Snippet 6: N Digit BCD Adder using CLA](#_7-_N-Digit_BCD)

#### [Code Snippet 7: N Digit BCD Adder using Ripple](#_6-_N-Digit_BCD)

### 4.5 Verification and Error Detection

A Verilog testbench was developed to verify the correctness of the design. The testbench:

* Applies all combinations of valid BCD inputs for n = 3 (from 000 + 000 to 999 + 999)
* Converts input integers to BCD format and checks that the output sum matches the expected result
* Measures computation delay and calculates the maximum achievable clock frequency for the design
* Logs all mismatches between expected and actual outputs using $display to write detailed messages to an output file

This verification flow ensures the system is not only functionally correct under normal operation, but also compares different adding strategies.

#### [Code Snippet 8: Test Bench](#_Testbench)

## Result

Both the Ripple Carry and Carry Lookahead (CLA) versions of the 3-digit BCD adder were functionally verified using exhaustive simulation over all possible input combinations from 0 to 999, totaling 1,000,000 test cases. In both designs, the output sum and carry-out were compared against expected values derived from integer addition, and all test cases passed successfully with zero errors. Despite identical total simulation times reported by the tool, the CLA-based adder is inherently faster due to its ability to compute carries in parallel, reducing the internal propagation delay compared to the Ripple Carry adder, which resolves carries sequentially from the least significant digit upward. This advantage becomes more pronounced in high-speed or multi-digit designs, making the CLA adder a better choice for performance-critical applications, while the Ripple Carry design remains a simpler, lower-gate-count option for general-purpose use.

#### [Photo 1: Result of n digit BCD Adder using Ripple](#_Result_1_of_1)

#### [Photo 2: Result of n digit BCD Adder using Ripple](#_Result_2_of)

## Future Work

There are several ways to improve and build on this project in the future:

1. **Faster Design**  
   The current adder works correctly but is not the fastest. We can make it faster by using more advanced designs like full carry lookahead for all digits.
2. **Support for Subtraction**  
   Right now, the adder only handles addition. In the future, we can add subtraction support and maybe even allow negative numbers.
3. **Reduce Delay**  
   We can improve how the carry moves between digits to make the delay smaller and the circuit faster.
4. **Error Handling**  
   We can add features that detect or fix errors if something goes wrong during the addition.

## Appendix

### 1- Full Adder

module full\_adder(a, b, cin, sum, cout);

input a, b, cin;

output sum, cout;

wire xor1\_out, and1\_out, and2\_out;

xor #15 xor1(xor1\_out, a, b);

xor #15 xor2(sum, xor1\_out, cin);

and #11 and1(and1\_out, a, b);

and #11 and2(and2\_out, xor1\_out, cin);

or #11 or1(cout, and1\_out, and2\_out);

endmodule

### 2- Ripple Carry Adder

module ripple\_adder(a, b, cin, sum, cout);

input [3:0] a, b;

input cin;

output [3:0] sum;

output cout;

wire c1, c2, c3;

full\_adder fa0(a[0], b[0], cin, sum[0], c1);

full\_adder fa1(a[1], b[1], c1, sum[1], c2);

full\_adder fa2(a[2], b[2], c2, sum[2], c3);

full\_adder fa3(a[3], b[3], c3, sum[3], cout);

endmodule

### 3- Carry Lookahead Adder (CLA)

module cla\_adder(A, B, Cin, Sum, Cout);

input [3:0] A, B;

input Cin;

output [3:0] Sum;

output Cout;

wire [3:0] G, P, C;

and #11 g0(G[0], A[0], B[0]);

and #11 g1(G[1], A[1], B[1]);

and #11 g2(G[2], A[2], B[2]);

and #11 g3(G[3], A[3], B[3]);

xor #15 p0(P[0], A[0], B[0]);

xor #15 p1(P[1], A[1], B[1]);

xor #15 p2(P[2], A[2], B[2]);

xor #15 p3(P[3], A[3], B[3]);

assign C[0] = Cin;

wire c1\_part;

and #11 a1(c1\_part, P[0], Cin);

or #11 o1(C[1], G[0], c1\_part);

wire c2\_p1, c2\_p2;

and #11 a2\_1(c2\_p1, P[1], G[0]);

and #11 a2\_2(c2\_p2, P[1], P[0], Cin);

or #11 o2(C[2], G[1], c2\_p1, c2\_p2);

wire c3\_p1, c3\_p2, c3\_p3;

and #11 a3\_1(c3\_p1, P[2], G[1]);

and #11 a3\_2(c3\_p2, P[2], P[1], G[0]);

and #11 a3\_3(c3\_p3, P[2], P[1], P[0], Cin);

or #11 o3(C[3], G[2], c3\_p1, c3\_p2, c3\_p3);

wire co1, co2, co3, co4;

and #11 a4\_1(co1, P[3], G[2]);

and #11 a4\_2(co2, P[3], P[2], G[1]);

and #11 a4\_3(co3, P[3], P[2], P[1], G[0]);

and #11 a4\_4(co4, P[3], P[2], P[1], P[0], Cin);

or #11 o4(Cout, G[3], co1, co2, co3, co4);

xor #15 s0(Sum[0], P[0], C[0]);

xor #15 s1(Sum[1], P[1], C[1]);

xor #15 s2(Sum[2], P[2], C[2]);

xor #15 s3(Sum[3], P[3], C[3]);

endmodule

### 4- BCD Ripple Adder

module bcd\_ripple\_adder(a, b, cin, s, carry);

input [3:0] a, b;

input cin;

output [3:0] s;

output carry;

wire [3:0] Q;

wire cout, ignoredCout;

wire [3:0] correction = 4'b0110;

ripple\_adder ra1(a, b, cin, Q, cout);

assign carry = cout | (Q[3] & (Q[2] | Q[1]));

wire [3:0] over9handle = carry ? correction : 4'b0000;

ripple\_adder ra2(Q, over9handle, 1'b0, s, ignoredCout);

endmodule

### 5- BCD CLA Adder

module bcd\_cla\_adder(a, b, cin, s, carry);

input [3:0] a, b;

input cin;

output [3:0] s;

output carry;

wire [3:0] Q;

wire cout, ignoredCout;

cla\_adder cla1(a, b, cin, Q, cout);

wire condition1, condition2;

and #11 a1(condition1, Q[3], Q[2]);

and #11 a2(condition2, Q[3], Q[1]);

or #11 over9(Q\_gt\_9, condition1, condition2);

or #11 c\_out(carry, cout, Q\_gt\_9);

wire [3:0] correction = carry ? 4'b0110 : 4'b0000;

cla\_adder cla2(Q, correction, 1'b0, s, ignoredCout);

endmodule

### 6- N-Digit BCD Adder Using Ripple carry adder

module n\_digit\_bcd\_adder(clk, rst, A\_in, B\_in, Cin\_in, Sum\_out, Cout\_out);

parameter N = 3;

input [4\*N-1:0] A\_in, B\_in;

input Cin\_in, clk, rst;

output reg [4\*N-1:0] Sum\_out;

output reg Cout\_out;

reg [4\*N-1:0] A\_reg, B\_reg;

reg Cin\_reg;

wire [4\*N-1:0] Sum\_wire;

wire Cout\_wire;

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\_in;

B\_reg <= B\_in;

Cin\_reg <= Cin\_in;

end

end

always @(posedge clk or posedge rst) begin

if (rst) begin

Sum\_out <= 0;

Cout\_out <= 0;

end else begin

Sum\_out <= Sum\_wire;

Cout\_out <= Cout\_wire;

end

end

wire [4\*N-1:0] s\_wires;

assign Sum\_wire = s\_wires;

assign Cout\_wire = carry[N];

genvar j;

generate

for (j = 0; j < N; j = j + 1) begin : bcd\_digit\_adders

wire [3:0] a\_bits = A\_reg[4\*j+3 : 4\*j];

wire [3:0] b\_bits = B\_reg[4\*j+3 : 4\*j];

wire [3:0] s\_bits;

wire c\_in = carry[j];

wire c\_out;

bcd\_ripple\_adder bcd\_adder(a\_bits, b\_bits, c\_in, s\_bits, c\_out);

assign s\_wires[4\*j+3 : 4\*j] = s\_bits;

assign carry[j+1] = c\_out;

end

endgenerate

endmodule

### 7- N-Digit BCD Adder Using Carry Look Ahead adder

module n\_digit\_bcd\_adder(clk, rst, A\_in, B\_in, Cin\_in, Sum\_out, Cout\_out);

parameter N = 3;

input [4\*N-1:0] A\_in, B\_in;

input Cin\_in, clk, rst;

output reg [4\*N-1:0] Sum\_out;

output reg Cout\_out;

reg [4\*N-1:0] A\_reg, B\_reg;

reg Cin\_reg;

wire [4\*N-1:0] Sum\_wire;

wire Cout\_wire;

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\_in;

B\_reg <= B\_in;

Cin\_reg <= Cin\_in;

end

end

always @(posedge clk or posedge rst) begin

if (rst) begin

Sum\_out <= 0;

Cout\_out <= 0;

end else begin

Sum\_out <= Sum\_wire;

Cout\_out <= Cout\_wire;

end

end

wire [4\*N-1:0] s\_wires;

assign Sum\_wire = s\_wires;

assign Cout\_wire = carry[N];

genvar j;

generate

for (j = 0; j < N; j = j + 1) begin : bcd\_digit\_adders

wire [3:0] a\_bits = A\_reg[4\*j+3 : 4\*j];

wire [3:0] b\_bits = B\_reg[4\*j+3 : 4\*j];

wire [3:0] s\_bits;

wire c\_in = carry[j];

wire c\_out;

bcd\_ripple\_adder bcd\_adder(a\_bits, b\_bits, c\_in, s\_bits, c\_out);

assign s\_wires[4\*j+3 : 4\*j] = s\_bits;

assign carry[j+1] = c\_out;

end

endgenerate

endmodule

## **Testbench**

`timescale 1ns/1ps

module n\_digit\_bcd\_adder\_tb;

parameter N = 3;

integer MAX\_VAL = 10\*\*N - 1;

reg clk, rst;

reg [4\*N-1:0] A\_in, B\_in;

reg Cin\_in;

wire [4\*N-1:0] Sum\_out;

wire Cout\_out;

integer a, b;

integer expected\_sum, expected\_bcd, actual\_bcd;

integer total\_tests = 0;

integer errors = 0;

integer time\_unit = 15;

n\_digit\_bcd\_adder #(.N(N)) dut (

.clk(clk),

.rst(rst),

.A\_in(A\_in),

.B\_in(B\_in),

.Cin\_in(Cin\_in),

.Sum\_out(Sum\_out),

.Cout\_out(Cout\_out)

);

always #(time\_unit / 2) clk = ~clk;

function [4\*N-1:0] int\_to\_bcd;

input integer val;

integer i;

begin

for (i = 0; i < N; i = i + 1)

int\_to\_bcd[i\*4 +: 4] = (val / (10\*\*i)) % 10;

end

endfunction

function integer bcd\_to\_int;

input [4\*N-1:0] bcd;

integer i;

begin

bcd\_to\_int = 0;

for (i = 0; i < N; i = i + 1)

bcd\_to\_int = bcd\_to\_int + (bcd[i\*4 +: 4] \* (10\*\*i));

end

endfunction

initial begin

$display("Starting testbench for n\_digit\_bcd\_adder (N = %0d)", N);

clk = 0;

rst = 1;

Cin\_in = 0;

A\_in = 0;

B\_in = 0;

#time\_unit rst = 0;

for (a = 0; a <= MAX\_VAL; a = a + 1) begin

for (b = 0; b <= MAX\_VAL; b = b + 1) begin

A\_in = int\_to\_bcd(a);

B\_in = int\_to\_bcd(b);

Cin\_in = 0;

#time\_unit;

#time\_unit;

expected\_sum = a + b;

expected\_bcd = expected\_sum % (10\*\*N);

actual\_bcd = bcd\_to\_int(Sum\_out);

total\_tests = total\_tests + 1;

if ((expected\_bcd != actual\_bcd) ||

(expected\_sum > MAX\_VAL && Cout\_out != 1) ||

(expected\_sum <= MAX\_VAL && Cout\_out != 0)) begin

errors = errors + 1;

$display("%0d + %0d FAILED at time %t ns | Expected: %0d, Cout: %b | Got: %0d, Cout: %b",

a, b, $time, expected\_bcd, (expected\_sum > MAX\_VAL), actual\_bcd, Cout\_out);

end else begin

$display("%0d + %0d PASSED at time %t ns | Expected: %0d, Cout: %b | Got: %0d, Cout: %b",

a, b, $time, expected\_bcd, (expected\_sum > MAX\_VAL), actual\_bcd, Cout\_out);

end

end

end

$display("==============================================");

$display("Simulation completed at time: %t ns", $time);

$display("Total tests run : %0d", total\_tests);

$display("Total errors detected : %0d", errors);

if (errors == 0)

$display("All tests passed successfully!");

else

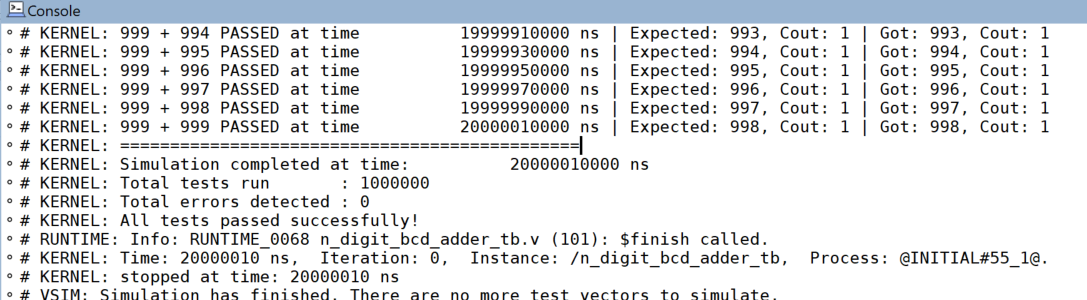
$display("Some tests failed. Review errors above.");

$finish;

end

endmodule

## Result 1 of testbench



## Result 2 of testbench

