Dept. of Electronics and Electrical Communication Engineering
Indian Institute of Technology Kharagpur

VLSI LABORATORY
(EC39004)

Experiment No: 1

Title: Designing MUX and Full Adder

Date of Submission: January 22nd, 2024

Name 1: Irsh Vijay
Roll Number 1: 21EC30025
Name 2: Pranjal Singh
Roll Number 2: 21EC10051

Objective 1:

- Design 8:1 MUX using 2:1 MUXs in behavioral and structural styles.
- Demonstrate through testbench waveforms and verify the results.

Introduction:

A multiplexer, a vital component in digital circuits, is a combinational circuit featuring numerous data inputs and a single output, determined by control or select inputs. With n control lines, it accommodates 2^n input lines, facilitating diverse applications in digital design. In telecommunications, multiplexers, employing techniques like Time-Division Multiplexing (TDM) and Frequency-Division Multiplexing (FDM), efficiently transmit multiple data streams over a single channel, optimizing bandwidth. In processor design, multiplexers play a pivotal role in data routing, addressing decoding, and overall circuit efficiency.

Verilog Modeling:

In Verilog, the description of digital circuits occurs through Behavioral and Structural Modeling. Behavioral modeling focuses on high-level abstraction, describing the circuit's behavior akin to coding in high-level languages like Python, utilizing statements such as 'if-else,' 'case,' and 'always.' This level abstracts away from circuit or switch-level details and often uses truth tables or boolean expressions.

Structural Modeling, in contrast, delves into lower-level details, defining circuits by specifying components, interconnections, and relationships using statements like 'wire' and 'module.' Typically, structural modeling follows behavioral modeling and precedes circuit manufacturing, enabling comprehensive testing at a lower level.

Implementation of 8:1 Multiplexer:

An 8:1 multiplexer can be ingeniously implemented using seven 2:1 multiplexers, strategically interconnected. The first level comprises four 2:1 multiplexers with a total of eight inputs, driven by a single control line (LSB). The second level involves two units, and the final level has a single unit, each with a dedicated control line. The progression of control lines ascends from the Least Significant Bit (LSB) to the Most Significant Bit (MSB) across the levels. This hierarchical arrangement optimizes the utilization of 2:1 multiplexers to create a versatile 8:1 multiplexer.

Part 1 Code:

2:1 MUX:

```
//behavioural
module mux2to1b(input [1:0] in, input sel, output out);
  assign out = sel ? in[1] : in[0];
endmodule

//structural
module mux2to1s(input [1:0] in, input sel, output out);

  wire nsel;
  wire [1:0] iw;
  not n0(nsel, sel);

  and ao(iw[0], in[0], nsel);
  and a1(iw[1], in[1], sel);

  or o0(out, iw[0], iw[1]);

endmodule
```

8:1 MUX:

```
//behavioural

module mux8to1b(input [7:0] in, input [2:0] sel, output out);

  wire [5:0] iw;

  mux2to1 m1(in[7:6], sel[0], iw[3]);
  mux2to1 m2(in[5:4], sel[0], iw[2]);
  mux2to1 m3(in[3:2], sel[0], iw[1]);
  mux2to1 m4(in[1:0], sel[0], iw[0]);

  mux2to1 m5(iw[3:2], sel[1], iw[5]);
  mux2to1 m6(iw[1:0], sel[1], iw[4]);

  mux2to1 m7 (iw[5:4], sel[2], out);

endmodule



//structural
module mux8to1s(input [7:0] in, input [2:0] sel, output out);

  wire [5:0] iw;
  mux2to1 m1(in[7:6], sel[0], iw[3]);
  mux2to1 m2(in[5:4], sel[0], iw[2]);
```

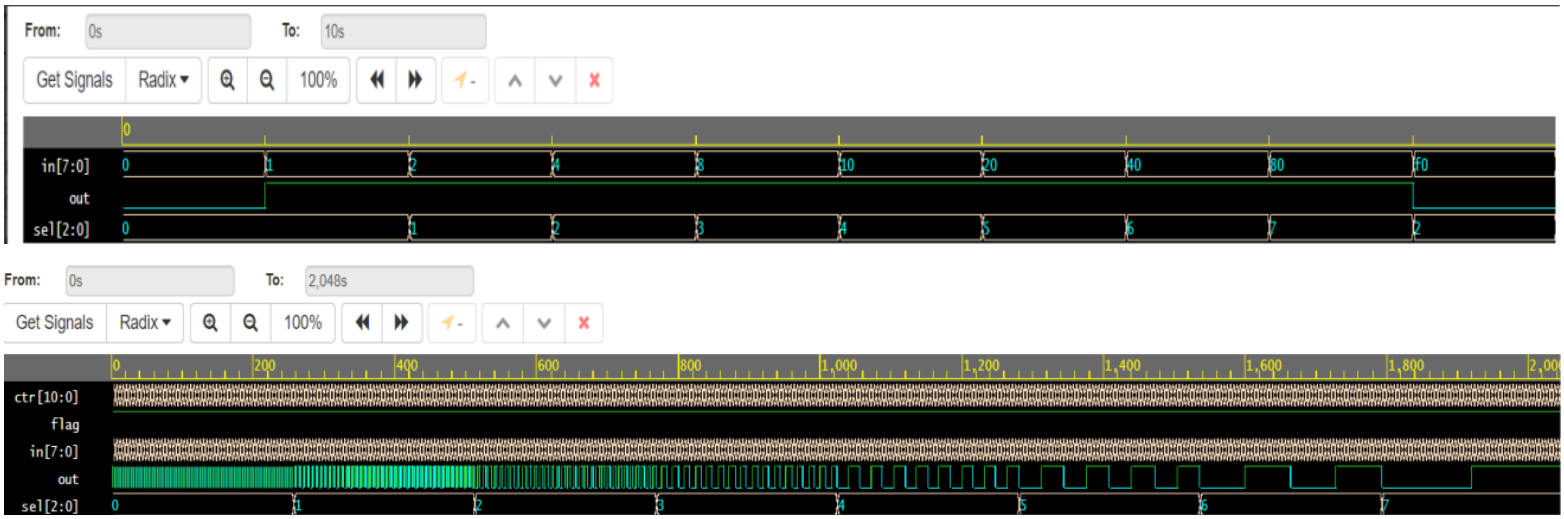
```
  mux2to1 m3(in[3:2], sel[0], iw[1]);
  mux2to1 m4(in[1:0], sel[0], iw[0]);

  mux2to1 m5(in[3:2], sel[1], iw[5]);
  mux2to1 m6(in[1:0], sel[1], iw[4]);

  mux2to1 m7 (iw[5:4], sel[2], out);

endmodule
```

Testbench Code:

```
//behavioural
module tb_mux8to10b;
reg [7:0] in; reg [2:01 sel;
wire out;
mux8tol mO(in, sel, out);
initial begin
§dumpfile("mux8to1.
§dumpvars (0);
_behavioral.vcd");
Smonitor ("input = %b, sel = %b, output = %b", in, sel, out);
in = 8'b00000000; sel = 3'b000;
#1;
in = 8'b00000001; sel = 3'b000;
#1;
in = 8'b00000010; sel = 3'b001;
#1;
in = 8'b00000100; sel = 3'b010;
#1;
in = 8'600001000; sel = 3'b011;
#1;
in = 8'b00010000; sel = 3'b100;
#1;
in = 8'b00100000; sel = 3'b101;
#1;
in = 8'b01000000; sel = 3'b110;
#1;
in = 8'b10000000; sel = 3'b111;
#1;
in = 8'b11110000; sel = 3'b010;
#1;
in = 8'b10101010; sel = 3'b100;
end
endmodule

//structural
module tb_mux8to10s;
  reg [7:01 in; reg [2:0] sel;
```

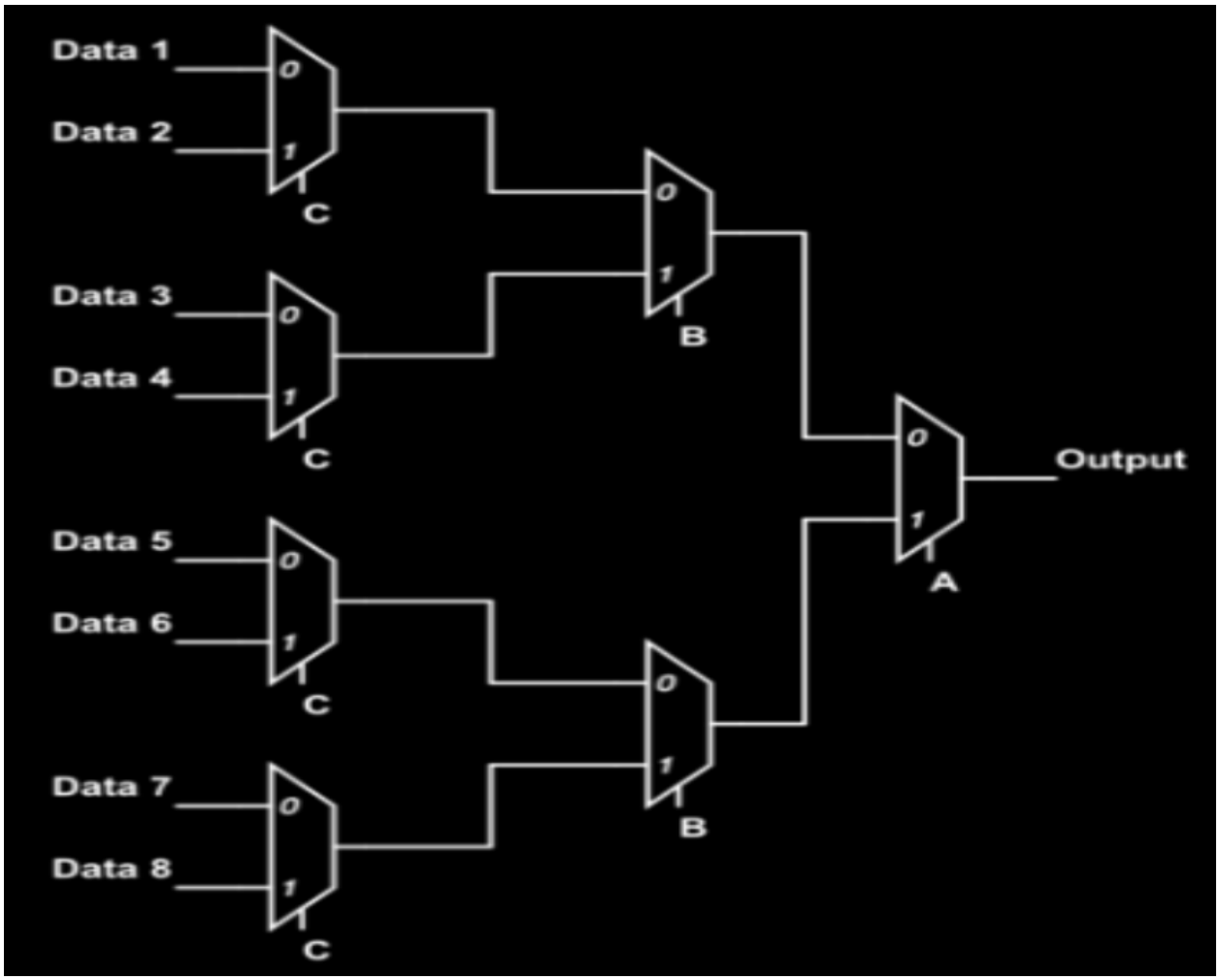
```
  wire out;
  mux8to1 mOin, sel, out);
  initial begin
  $dumpfile("mux8to1_behavioral.vcd");
  $dumpvars (0);
  Smonitor ("input = %b, sel = %b, output = %b", in, sel, out);
  in = 8'b00000000; sel = 3'b000;
  #1;
  in = 8'b00000001; sel = 3'6000;
  #1;
  in = 8'b00000010; sel = 3'b001;
  #1;
  in = 8'b00000100; sel = 3'b010;
  #1;
  in = 8'b00001000; sel = 3'b011;
  #1;
  in = 8'b00010000; sel = 3'b100;
  #1;
  in = 8'b00100000; sel = 3'b101;
  #1;
  in = 8'b01000000; sel = 3'b110;
  #1;
  in = 8'b10000000; sel = 3'b111;
  #1;
  in = 8'b11110000; sel = 3'b010;
  #1;
  in = 8'10101010; sel = 3'b100;
  end
endmodule
```

Results:



```
VCD info: dumpfile mux8to1_behavioral.vcd opened for output.
input = 00000000, sel = 000, output = 0
input = 00000001, sel = 000, output = 1
input = 00000010, sel = 001, output = 1
input = 00000100, sel = 010, output = 1
input = 00001000, sel = 011, output = 1
input = 00010000, sel = 100, output = 1
input = 00100000, sel = 101, output = 1
input = 01000000, sel = 110, output = 1
input = 10000000, sel = 111, output = 1
input = 11110000, sel = 010, output = 0
input = 10101010, sel = 100, output = 0
```

```
VCD info: dumpfile mux8to1.vcd opened for output.
input = 00000000, sel = 000, out = 0
input = 00000001, sel = 000, out = 1
input = 00000010, sel = 000, out = 0
input = 00000011, sel = 000, out = 1
input = 00000100, sel = 000, out = 0
input = 00000101, sel = 000, out = 1
input = 00000110, sel = 000, out = 0
input = 00000111, sel = 000, out = 1
input = 00001000, sel = 000, out = 0
input = 00001001, sel = 000, out = 1
input = 00001010, sel = 000, out = 0
input = 00001011, sel = 000, out = 1
input = 00001100, sel = 000, out = 0
input = 00001101, sel = 000, out = 1
input = 00001110, sel = 000, out = 0
input = 00001111, sel = 000, out = 1
input = 00010000, sel = 000, out = 0
input = 00010001, sel = 000, out = 1
input = 00010010, sel = 000, out = 0
input = 00010011, sel = 000, out = 1
input = 00010100, sel = 000, out = 0
input = 00010101, sel = 000, out = 1
input = 00010110, sel = 000, out = 0
input = 00010111, sel = 000, out = 1
```


Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Data 7
Data 8
0
1
C
B
A
Output

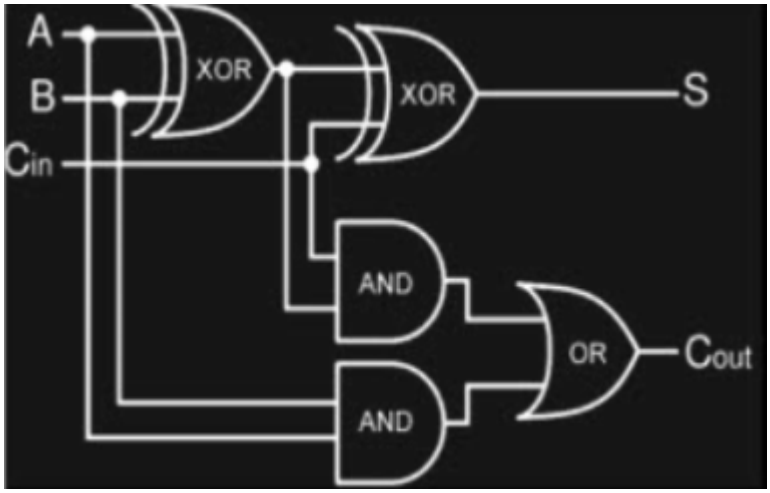
Objective 2:
Implementation and Testing of a 4-Bit Ripple Carry Adder Using 1-Bit Full Adders

Introduction:
A full adder is a fundamental digital circuit designed to add three binary bits: two inputs (A and B) and a carry input (Cin). It produces two outputs—sum (S) and carry-out (Cout). Employing XOR, AND, and OR gates, full adders play a pivotal role in arithmetic circuits, finding widespread use in microprocessor design, arithmetic logic units (ALUs), and digital signal processing. They contribute to the foundation of binary arithmetic in digital systems, crucial for precision in various computational tasks.

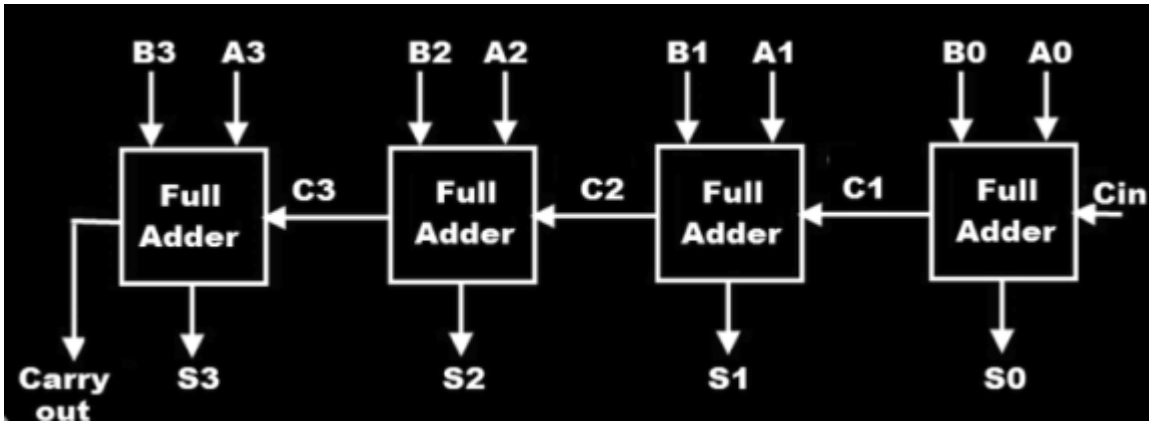
Ripple Carry Adder:
The ripple carry adder, a key component in digital circuit design, facilitates binary addition by cascading full adders. Each full adder stage generates a sum bit and a carry-out bit, with the sum bit becoming part of the final output. The carry-out bit is propagated to the subsequent stage, creating a ripple effect. Despite its simplicity, the ripple carry adder suffers from propagation delays, impacting performance in high-speed applications. In modern processors, advanced adder architectures like carry-lookahead and carry-select adders address these limitations in critical path operations.

Applications:
Ripple carry adders are extensively used in microprocessors and digital signal processing applications. In memory addressing circuits, they perform the crucial task of adding memory addresses, enabling computer architectures to access specific locations during read and write operations. While ripple carry adders may not excel in high-speed operations, their simplicity, ease of implementation, and relevance in less time-sensitive tasks make them valuable components in digital systems.

Structure of a full adder using logic gates



Ripple Carry Adder using Full adders:



Part 2 Code:

Adder Code:

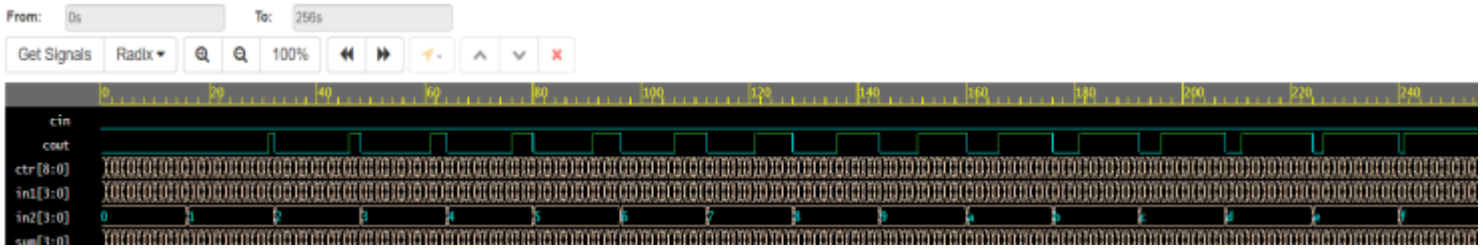
```
module fulladder (input [1:0] in, input cin, output sum, output cout);

  assign sum = in[0] ^ in[1] ^ cin;
  assign cout = (in[0]&&in[1]) || (cin&&(in[0]^in[1]));
endmodule

module ripplecarryadder_4(input [3:0] in1, input [3:0] in2, input cin, output
[3:0] sum, output cout) ;

  wire [2:0] iw;

  fulladder fa0(in1[0], in2[0], cin, sum[0], iw[0]);
  fulladder fal(in1[1], in2[1], iw[O], sum[1], iw[1]);
  fulladder fa2(in1[2], in2[2], iw[1], sum[2], iw[2]);
  fulladder fa3(in1[3], in2[3], iw[2], sum[3], cout);
endmodule
```

Testbench Code:

```
module tbripplecarryadder_40;

  reg [3:0] in1;
  reg [3:0] in2;
  reg cin;

  reg [8:0] ctr;

  wire [3:0] sum;

  wire cout;

  ripplecarryadder_4 rac0(in1, in2, cin, sum, cout) ;
  initial begin
    $dumpfile("ripplecarryadder_4.vcd");
    $dumpvars (0);
    $monitor("input 1 = %b, input 2 = %b, cin = %b, sum = %b, cout = %b", in1, in2, cin, sum, cout);
  for(ctr = 9'b0-000_0000; ctr<=9'b11111_1111; ctr=ctr+1)
    begin
      assign in1 = ctr[3:0];
      assign in2 = ctr[7:4];
      assign cin = ctr[8];
      #1;
    end
  end
endmodule
```

Results:-

```
VCD info: dumpfile ripplecarryadder_4.vcd opened for output.
input 1 = 0000, input 2 = 0000, cin = 0, sum = 0000, cout = 0
input 1 = 0001, input 2 = 0000, cin = 0, sum = 0001, cout = 0
input 1 = 0010, input 2 = 0000, cin = 0, sum = 0010, cout = 0
input 1 = 0011, input 2 = 0000, cin = 0, sum = 0011, cout = 0
input 1 = 0100, input 2 = 0000, cin = 0, sum = 0100, cout = 0
input 1 = 0101, input 2 = 0000, cin = 0, sum = 0101, cout = 0
input 1 = 0110, input 2 = 0000, cin = 0, sum = 0110, cout = 0
input 1 = 0111, input 2 = 0000, cin = 0, sum = 0111, cout = 0
input 1 = 1000, input 2 = 0000, cin = 0, sum = 1000, cout = 0
input 1 = 1001, input 2 = 0000, cin = 0, sum = 1001, cout = 0
input 1 = 1010, input 2 = 0000, cin = 0, sum = 1010, cout = 0
input 1 = 1011, input 2 = 0000, cin = 0, sum = 1011, cout = 0
input 1 = 1100, input 2 = 0000, cin = 0, sum = 1100, cout = 0
input 1 = 1101, input 2 = 0000, cin = 0, sum = 1101, cout = 0
input 1 = 1110, input 2 = 0000, cin = 0, sum = 1110, cout = 0
input 1 = 1111, input 2 = 0000, cin = 0, sum = 1111, cout = 0
input 1 = 0000, input 2 = 0001, cin = 0, sum = 0001, cout = 0
input 1 = 0001, input 2 = 0001, cin = 0, sum = 0010, cout = 0
input 1 = 0010, input 2 = 0001, cin = 0, sum = 0011, cout = 0
input 1 = 0011, input 2 = 0001, cin = 0, sum = 0100, cout = 0
input 1 = 0100, input 2 = 0001, cin = 0, sum = 0101, cout = 0
input 1 = 0101, input 2 = 0001, cin = 0, sum = 0110, cout = 0
input 1 = 0110, input 2 = 0001, cin = 0, sum = 0111, cout = 0
input 1 = 0111, input 2 = 0001, cin = 0, sum = 1000, cout = 0
input 1 = 1000, input 2 = 0001, cin = 0, sum = 1001, cout = 0
input 1 = 1001, input 2 = 0001, cin = 0, sum = 1010, cout = 0
input 1 = 1010, input 2 = 0001, cin = 0, sum = 1011, cout = 0
input 1 = 1011, input 2 = 0001, cin = 0, sum = 1100, cout = 0
input 1 = 1100, input 2 = 0001, cin = 0, sum = 1101, cout = 0
```



Discussion:-

Observation 1:

Verilog, a hardware description language, provides two primary modeling approaches: behavioral and structural. Behavioral modeling focuses on a high-level description of the required hardware, utilizing truth tables or boolean equations to implement the desired behavior. This approach is ideal for design and verification. On the other hand, structural modeling offers a lower-level description of hardware, involving various components and their interconnections. It is more detailed and is commonly employed for implementation and synthesis. In test benches, inputs are declared as reg, while outputs are declared as wire. This reflects the continuous monitoring of outputs for changes, while inputs only change at specific instances.

Observation 2:

Full adders, essential for bit addition, outperform half adders by incorporating an additional input for the input carry. This feature is crucial in ripple carry adder design, where the output carry from one stage becomes the input carry for the subsequent stages. A 4-bit ripple carry adder is a combinational digital circuit capable of adding two 4-bit numbers along with any initial carry. Full adders are extensively used in modern computers, particularly in Arithmetic Logic Units (ALUs) of Central Processing Units (CPUs). Their applications extend to advanced functionalities such as error detection and correction codes and digital signal processing.

Exhaustive testing of the 4-bit ripple carry adder was conducted using a for loop in the test bench, encompassing all possible input combinations. Due to the extensive number of combinations, not all are explicitly shown, but the provided code methodology adequately reflects the thorough testing approach employed. This rigorous testing ensures the reliability and functionality of the 4-bit ripple carry adder across diverse scenarios, validating its performance in real-world applications.