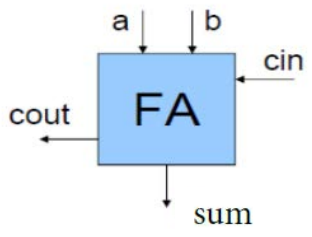
Computer Organization & Architecture

Lab 1

Objective: To analyze the efficiency of different adder topologies in terms of processing delay.

Problem 1: Implement addition of two 4-bit unsigned numbers A and B provided 4 full adders and two 4-bit registers R0 and R1.

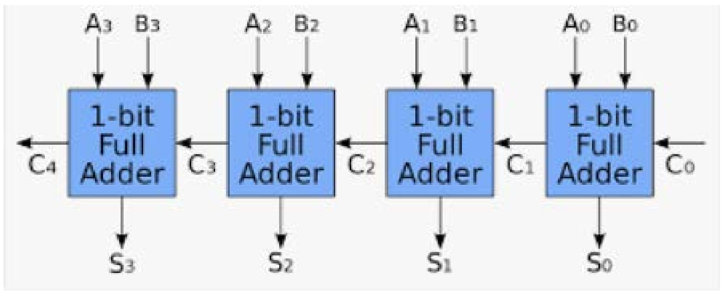
Description of 1-bit Full Adder:



where,

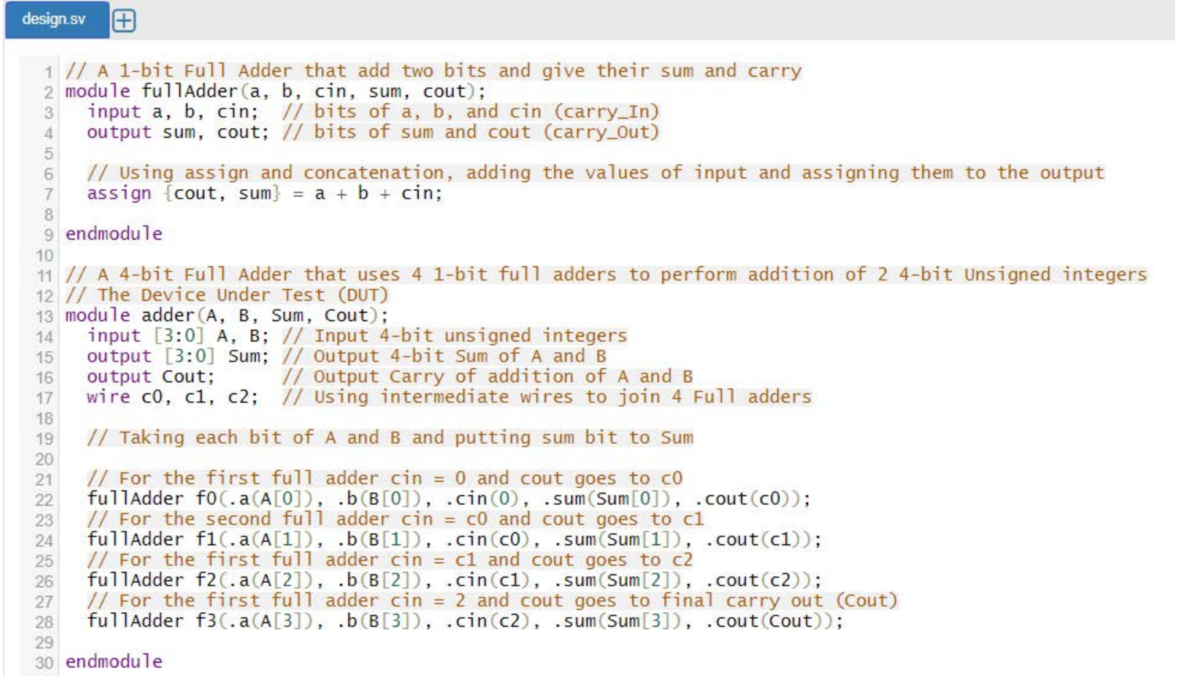
sum = a ^ b ^ cin, and
cout = (a & b) | (cin & (a ^ b))

Approach: To solve the above problem, I will join the 4 full adders parallely such that, cout of 1 full adder goes to cin of another full adder like shown below.
(**Model better known as Ripply Carry Adder**)

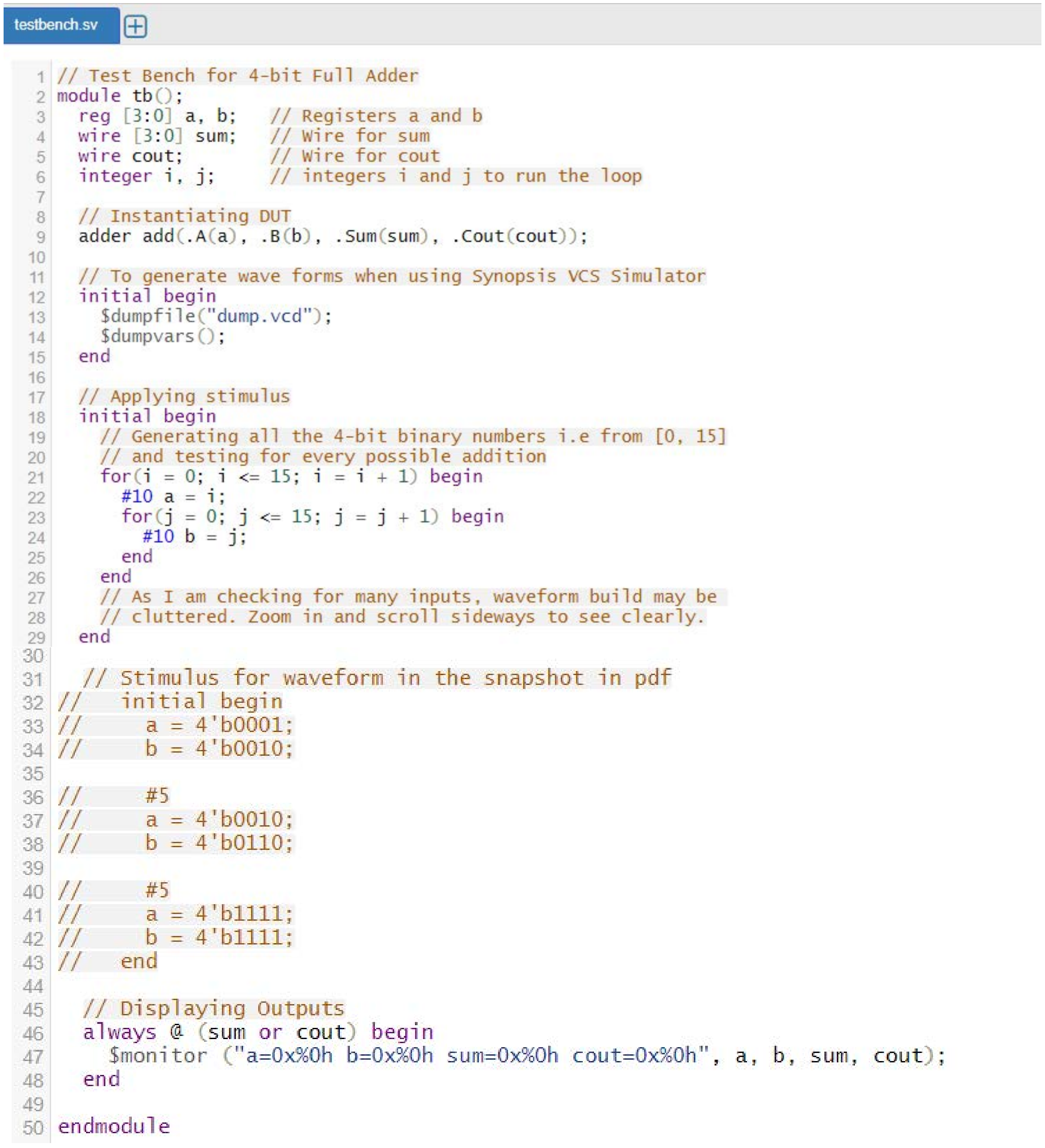


here, C0 = 0 and C4 will be final carry of the addition of A and B

Verilog Code: Below is a screenshot from the eda playground for the implementation of above problem using verilog code.
(The code is also attached as q1_design.sv and q1_testbench.sv with this submission)


design.sv

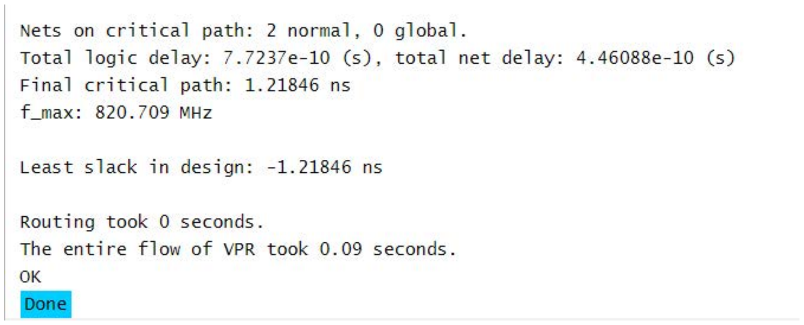
```
// A 1-bit Full Adder that add two bits and give their sum and carry
module fullAdder(a, b, cin, sum, cout);
  input a, b, cin;  // bits of a, b, and cin (carry_In)
  output sum, cout; // bits of sum and cout (carry_Out)

  // Using assign and concatenation, adding the values of input and assigning them to the output
  assign {cout, sum} = a + b + cin;

endmodule

// A 4-bit Full Adder that uses 4 1-bit full adders to perform addition of 2 4-bit Unsigned integers
// The Device Under Test (DUT)
module adder(A, B, Sum, Cout);
  input [3:0] A, B; // Input 4-bit unsigned integers
  output [3:0] Sum; // Output 4-bit Sum of A and B
  output Cout;      // Output Carry of addition of A and B
  wire c0, c1, c2;  // Using intermediate wires to join 4 Full adders

  // Taking each bit of A and B and putting sum bit to Sum

  // For the first full adder cin = 0 and cout goes to c0
  fullAdder f0(.a(A[0]), .b(B[0]), .cin(0), .sum(Sum[0]), .cout(c0));
  // For the second full adder cin = c0 and cout goes to c1
  fullAdder f1(.a(A[1]), .b(B[1]), .cin(c0), .sum(Sum[1]), .cout(c1));
  // For the first full adder cin = c1 and cout goes to c2
  fullAdder f2(.a(A[2]), .b(B[2]), .cin(c1), .sum(Sum[2]), .cout(c2));
  // For the first full adder cin = 2 and cout goes to final carry out (Cout)
  fullAdder f3(.a(A[3]), .b(B[3]), .cin(c2), .sum(Sum[3]), .cout(Cout));

endmodule
```


testbench.sv

```
// Test Bench for 4-bit Full Adder
module tb();
  reg [3:0] a, b;    // Registers a and b
  wire [3:0] sum;    // Wire for sum
  wire cout;         // Wire for cout
  integer i, j;      // integers i and j to run the loop

  // Instantiating DUT
  adder add(.A(a), .B(b), .Sum(sum), .Cout(cout));

  // To generate wave forms when using Synopsis VCS Simulator
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars();
  end

  // Applying stimulus
  initial begin
    // Generating all the 4-bit binary numbers i.e from [0, 15]
    // and testing for every possible addition
    for(i = 0; i <= 15; i = i + 1) begin
      #10 a = i;
      for(j = 0; j <= 15; j = j + 1) begin
        #10 b = j;
      end
    end
    // As I am checking for many inputs, waveform build may be
    // cluttered. Zoom in and scroll sideways to see clearly.
  end

  // Stimulus for waveform in the snapshot in pdf
//   initial begin
//     a = 4'b0001;
//     b = 4'b0010;

//     #5
//     a = 4'b0010;
//     b = 4'b0110;

//     #5
//     a = 4'b1111;
//     b = 4'b1111;
//   end

  // Displaying Outputs
  always @ (sum or cout) begin
    $monitor ("a=0x%0h b=0x%0h sum=0x%0h cout=0x%0h", a, b, sum, cout);
  end

endmodule
```

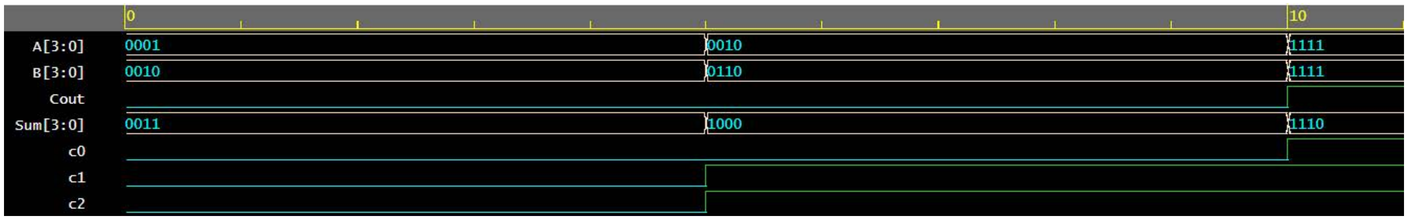
Experimental Results:

Critical Path (Evaluated using VTR 7.0 Simulator) = 1.21846 ns



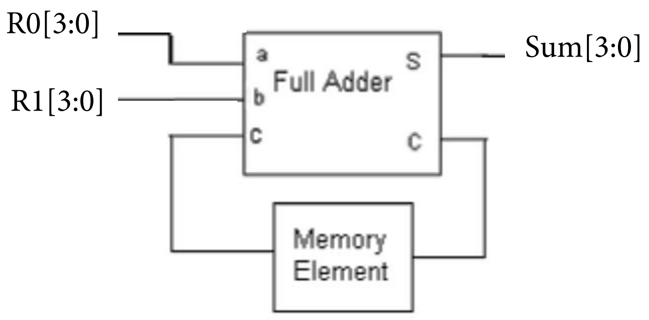
```
Nets on critical path: 2 normal, 0 global.
Total logic delay: 7.7237e-10 (s), total net delay: 4.46088e-10 (s)
Final critical path: 1.21846 ns
f_max: 820.709 MHz

Least slack in design: -1.21846 ns

Routing took 0 seconds.
The entire flow of VPR took 0.09 seconds.
OK
Done
```

Waveform: (Made using Synopsys VCS 2020.03 Simulator)



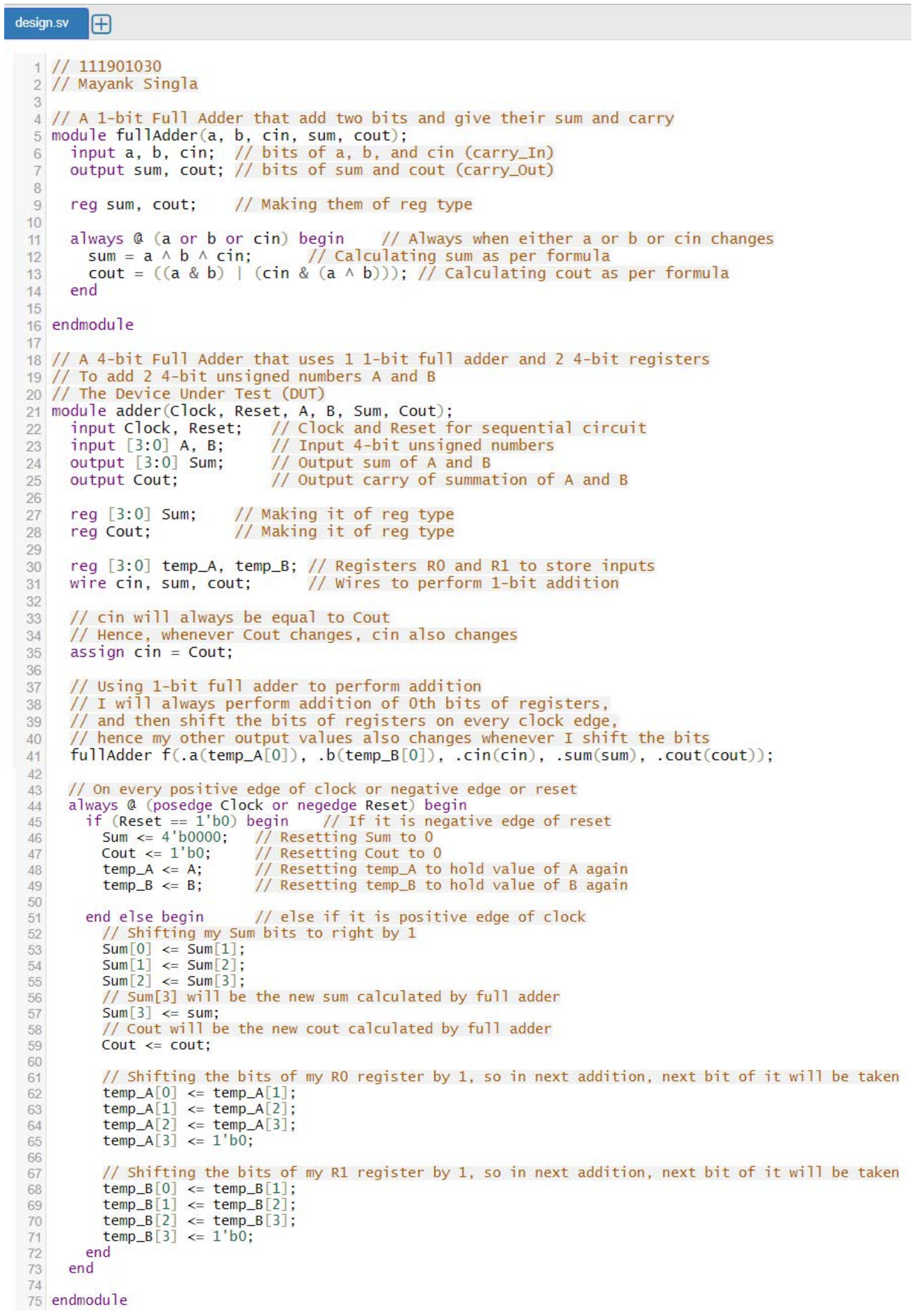
Problem 2: Implement addition of two 4-bit unsigned numbers A and B provided 1 full adder and two 4-bit registers R0 and R1.

Approach: To solve the above problem, I will use the registers as memory element and create a sequential circuit. For the reset of my circuit, I will set my register values to given numbers A and B and output sum and carry to 0. Then for every clock cycle, I will use the given 1-bit full adder to calculate the sum of 0th bits of R0 and R1 and store it in my 3rd bit of Sum. Then I will shift my all values of Sum, R0 and R1 to one bit right, so that in the next clock cycle next bits will add up and get stored in Sum. After 4 clock cycles, I will have my final sum in Sum calculated and Cout as my carry.

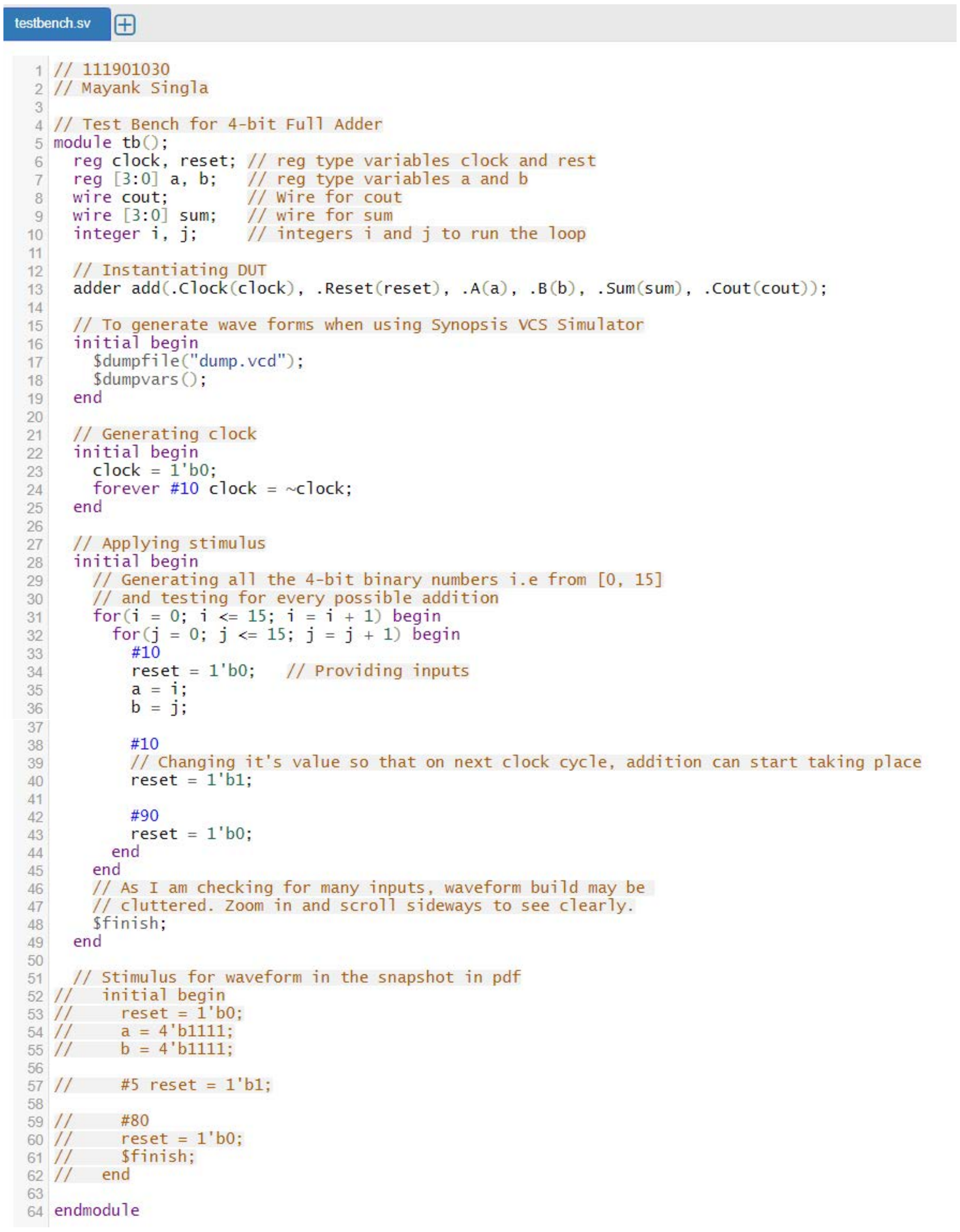


Verilog Code: Below is a screenshot from the eda playground for the implementation of above problem using verilog.

(The code is also attached as q2_design.sv and q2_testbench.sv with this submission)



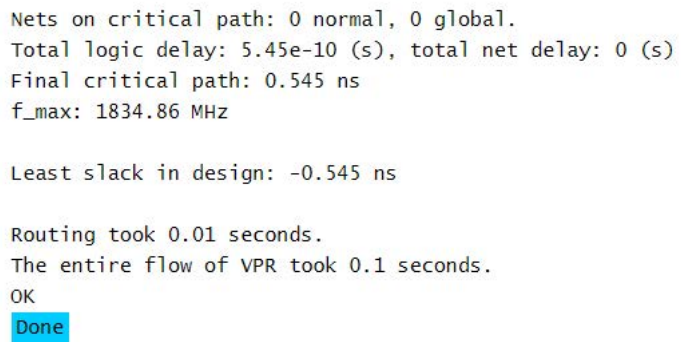
```verilog
// 111901030
// Mayank Singla

// A 1-bit Full Adder that add two bits and give their sum and carry
module fullAdder(a, b, cin, sum, cout);
  input a, b, cin;  // bits of a, b, and cin (carry_In)
  output sum, cout; // bits of sum and cout (carry_Out)

  reg sum, cout;    // Making them of reg type

  always @ (a or b or cin) begin    // Always when either a or b or cin changes
    sum = a ^ b ^ cin;        // Calculating sum as per formula
    cout = ((a & b) | (cin & (a ^ b))); // Calculating cout as per formula
  end

endmodule

// A 4-bit Full Adder that uses 1 1-bit full adder and 2 4-bit registers
// To add 2 4-bit unsigned numbers A and B
// The Device Under Test (DUT)
module adder(Clock, Reset, A, B, Sum, Cout);
  input Clock, Reset;   // Clock and Reset for sequential circuit
  input [3:0] A, B;     // Input 4-bit unsigned numbers
  output [3:0] Sum;     // Output sum of A and B
  output Cout;          // Output carry of summation of A and B

  reg [3:0] Sum;    // Making it of reg type
  reg Cout;         // Making it of reg type

  reg [3:0] temp_A, temp_B; // Registers R0 and R1 to store inputs
  wire cin, sum, cout;      // Wires to perform 1-bit addition

  // cin will always be equal to Cout
  // Hence, whenever Cout changes, cin also changes
  assign cin = Cout;

  // Using 1-bit full adder to perform addition
  // I will always perform addition of 0th bits of registers,
  // and then shift the bits of registers on every clock edge,
  // hence my other output values also changes whenever I shift the bits
  fullAdder f(.a(temp_A[0]), .b(temp_B[0]), .cin(cin), .sum(sum), .cout(cout));

  // On every positive edge of clock or negative edge or reset
  always @ (posedge Clock or negedge Reset) begin
    if (Reset == 1'b0) begin    // If it is negative edge of reset
      Sum <= 4'b0000;   // Resetting Sum to 0
      Cout <= 1'b0;     // Resetting Cout to 0
      temp_A <= A;      // Resetting temp_A to hold value of A again
      temp_B <= B;      // Resetting temp_B to hold value of B again

    end else begin      // else if it is positive edge of clock
      // Shifting my Sum bits to right by 1
      Sum[0] <= Sum[1];
      Sum[1] <= Sum[2];
      Sum[2] <= Sum[3];
      // Sum[3] will be the new sum calculated by full adder
      Sum[3] <= sum;
      // Cout will be the new cout calculated by full adder
      Cout <= cout;

      // Shifting the bits of my R0 register by 1, so in next addition, next bit of it will be taken
      temp_A[0] <= temp_A[1];
      temp_A[1] <= temp_A[2];
      temp_A[2] <= temp_A[3];
      temp_A[3] <= 1'b0;

      // Shifting the bits of my R1 register by 1, so in next addition, next bit of it will be taken
      temp_B[0] <= temp_B[1];
      temp_B[1] <= temp_B[2];
      temp_B[2] <= temp_B[3];
      temp_B[3] <= 1'b0;
    end
  end

endmodule
```



```
// 111901030
// Mayank Singla

// Test Bench for 4-bit Full Adder
module tb();
  reg clock, reset; // reg type variables clock and rest
  reg [3:0] a, b;   // reg type variables a and b
  wire cout;        // Wire for cout
  wire [3:0] sum;   // wire for sum
  integer i, j;     // integers i and j to run the loop

  // Instantiating DUT
  adder add(.Clock(clock), .Reset(reset), .A(a), .B(b), .Sum(sum), .Cout(cout));

  // To generate wave forms when using Synopsis VCS Simulator
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars();
  end

  // Generating clock
  initial begin
    clock = 1'b0;
    forever #10 clock = ~clock;
  end

  // Applying stimulus
  initial begin
    // Generating all the 4-bit binary numbers i.e from [0, 15]
    // and testing for every possible addition
    for(i = 0; i <= 15; i = i + 1) begin
      for(j = 0; j <= 15; j = j + 1) begin
        #10
        reset = 1'b0;   // Providing inputs
        a = i;
        b = j;

        #10
        // Changing it's value so that on next clock cycle, addition can start taking place
        reset = 1'b1;

        #90
        reset = 1'b0;
      end
    end
    // As I am checking for many inputs, waveform build may be
    // cluttered. Zoom in and scroll sideways to see clearly.
    $finish;
  end

  // Stimulus for waveform in the snapshot in pdf
//   initial begin
//     reset = 1'b0;
//     a = 4'b1111;
//     b = 4'b1111;

//     #5 reset = 1'b1;

//     #80
//     reset = 1'b0;
//     $finish;
//   end

endmodule
```

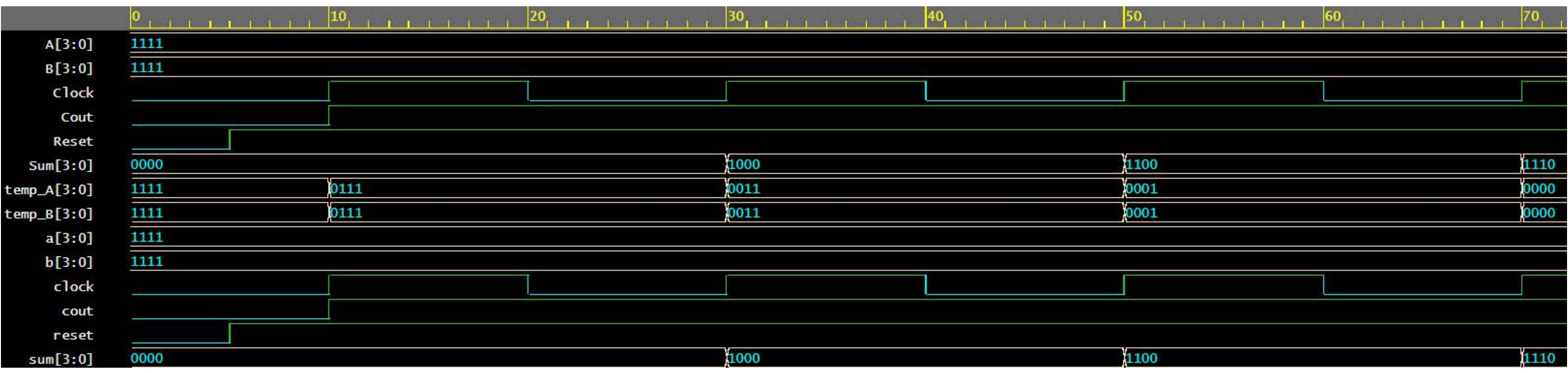
Experimental Results:

Critical Path (Evaluated using VTR 7.0 Simulator) = 0.545 ns



```
Nets on critical path: 0 normal, 0 global.
Total logic delay: 5.45e-10 (s), total net delay: 0 (s)
Final critical path: 0.545 ns
f_max: 1834.86 MHz

Least slack in design: -0.545 ns

Routing took 0.01 seconds.
The entire flow of VPR took 0.1 seconds.
OK
Done
```

Waveform: (Made using Synopsys VCS 2020.03 Simulator)



The final value of Sum of A = 15 (1111) and B = 15 (1111) is Sum = 14 (1110) and Cout = 1
(The Binary number generated was (11110) = 30, which due to overflow, Sum = 14 (1110) and Cout = 1)

Conclusion:

From the above two analysis of problems, it can be clearly seen that value of Critical Path is less in second case and more in first case.
Also, the hardware required is more in first case and less in second case.

Hence, we can say that second approach has better performance than the first approach.
So, for doing addition, we should definitely go with the second approach.

Submitted By:
Mayank Singla