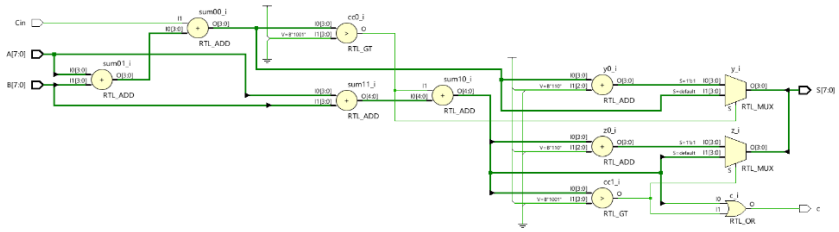
CADD JOB PROBLEMS

1. 8-bit BCD Adder

Design Code & RTL Schematic:

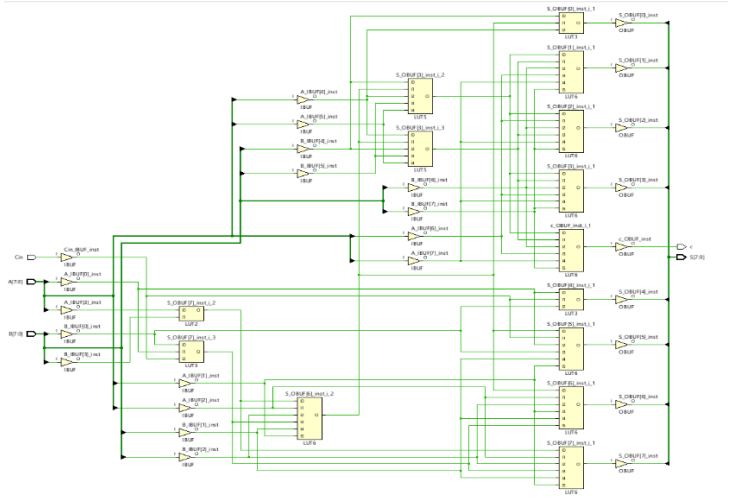
```
module BCD_Adder (
    input logic  [7:0] A,B,
    input Cin,
    output [7:0]S,
    output c
);
    wire [3:0]sum0,sum1;
    wire x0,x1;
    wire [3:0]y,z;
    wire cc0,cc1;
    assign {x0,sum0}=A[3:0]+B[3:0]+Cin;
    assign cc0=(sum0>4'd9)?1'b1:1'b0;
    assign y=cc0?sum0+4'd6:sum0;
    assign {x1,sum1}=A[7:4]+B[7:4]+cc0;
    assign cc1=(sum1 > 4'd9)?1'b1:1'b0;
    assign z=cc1?sum1+4'd6:sum1;
    assign S={y,z};
    assign c=x1|cc1;
endmodule
```

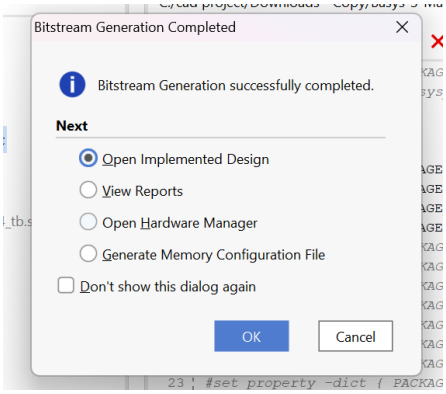


Test Bench Code & Waveform:

```
module BCD_Adder_tb;
    reg [7:0] A, B;
    reg Cin;
    wire [7:0] S;
    wire c;
    BCD_Adder uut (A,B,Cin,S,c);
    initial
     begin
        A = 8'b00000010;B = 8'b00000011;Cin = 0;#10;
        A = 8'b00001001;B = 8'b00000010;Cin = 0;#10;
        A = 8'b01011000;B = 8'b01001001;Cin = 0;#10;
        A = 8'b00000101;B = 8'b00000101;Cin = 1;#10;
        A = 8'b10011001;B = 8'b10011001; Cin = 0;#10;
        A = 8'b00000000;B = 8'b00000000;Cin = 0;#10;
        A = 8'b00100110;B = 8'b00010101;Cin = 1;#10;
    end
endmodule
```

Technology schematic & Bitstream Generation:





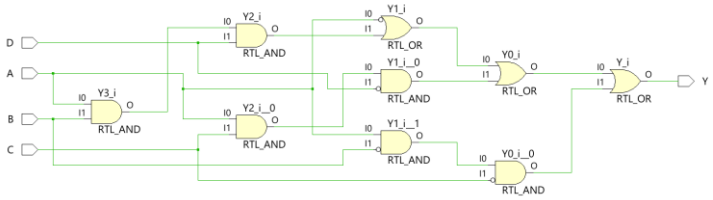
Q2. FPGA Flow – Combinational & Sequential circuits

(i) Combinational circuit:

$$Y = \overline{A}\overline{B}\overline{C}D + A\overline{B}\overline{C} + A\overline{B}CD + ABD + \overline{A}\overline{B}CD + B\overline{C}D + \overline{A}$$

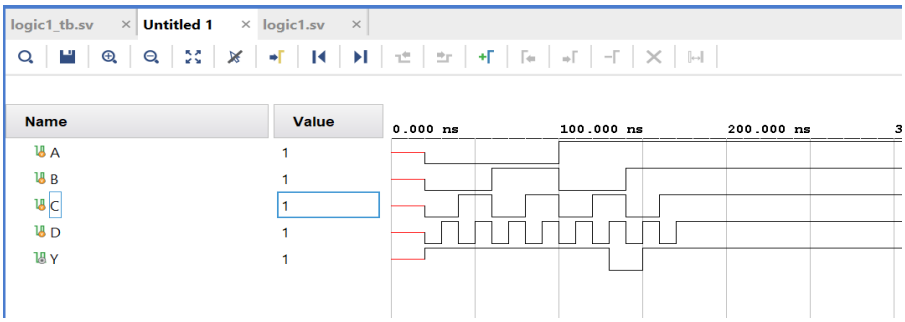
Design Code & RTL Schematic:

```
21 module logic1 (
22     input logic A, B, C, D,
23     output logic Y);
24     assign Y = ~A | (A&B&D) | (A&C&(~D)) | (A&(~B)&(~C));
25 endmodule
26
```

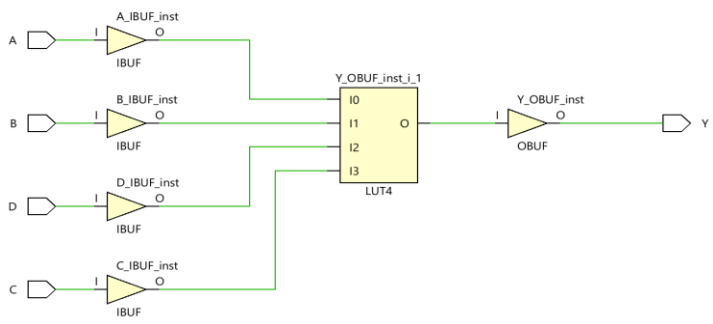


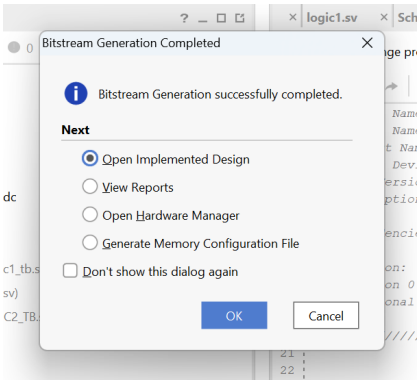
Test bench and Waveform:

```
21 module logic1_tb;
22 reg A,B,C,D;
23 wire Y;
24 logic1 dut(A,B,C,D,Y);
25 initial
26 begin
27 #20;A=0;B=0;C=0;D=0;
28 #10;A=0;B=0;C=0;D=1;
29 #10;A=0;B=0;C=1;D=0;
30 #10;A=0;B=0;C=1;D=1;
31 #10;A=0;B=1;C=0;D=0;
32 #10;A=0;B=1;C=0;D=1;
33 #10;A=0;B=1;C=1;D=0;
34 #10;A=0;B=1;C=1;D=1;
35 #10;A=1;B=0;C=0;D=0;
36 #10;A=1;B=0;C=0;D=1;
37 #10;A=1;B=0;C=1;D=0;
38 #10;A=1;B=0;C=1;D=1;
39 #10;A=1;B=1;C=0;D=0;
40 #10;A=1;B=1;C=0;D=1;
41 #10;A=1;B=1;C=1;D=0;
42 #10;A=1;B=1;C=1;D=1;
43 end
44 endmodule
```



Technology Schematic & Bitstream generation:



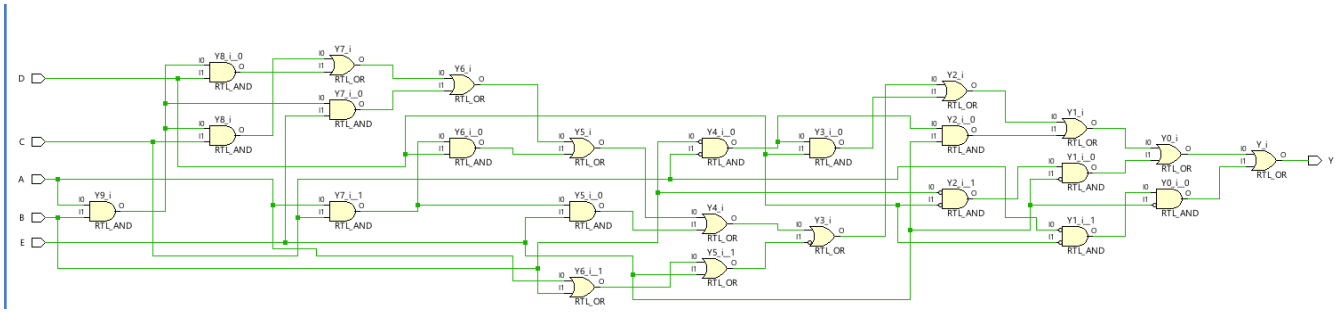


(ii) **Combinational circuit:**

$$Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D + \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$$

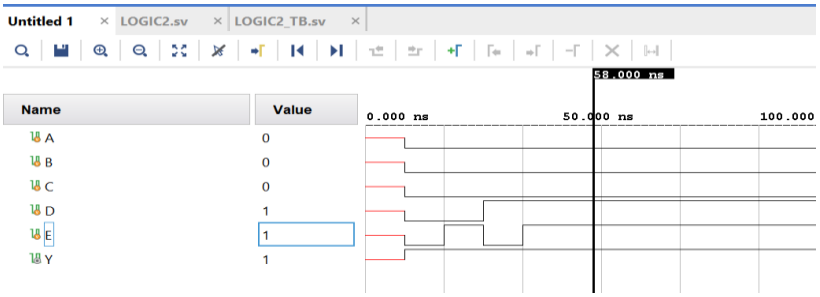
Design code & RTL Schematic:

```
21 module LOGIC2(A,B,C,D,E,Y);
22  input logic A,B,C,D,E;
23  output logic Y;
24  assign Y=( A&B&C | A&B&D | A&B&E | A&C&D | A&C&E | ~(A|B|E) | ~B&~C&D | ~B&~C&E | ~B&~D&~E | ~C&~D&~E );
25 endmodule
26
```

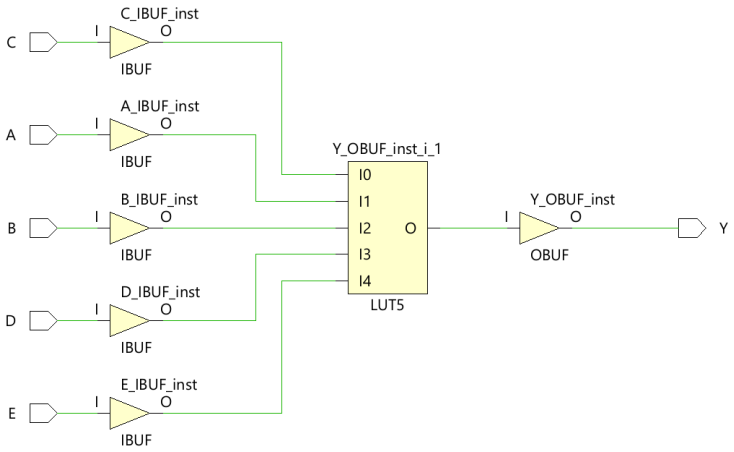


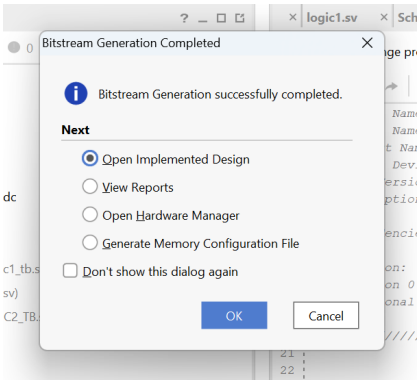
Test bench code & Waveform:

```
21 module LOGIC2_tb;
22  reg A,B,C,D,E;
23  wire Y;
24  LOGIC2 dut(A,B,C,D,E,Y);
25 initial
26 begin
27  #10;A=0;B=0;C=0;D=0;E=0;
28  #10;A=0;B=0;C=0;D=0;E=1;
29  #10;A=0;B=0;C=0;D=1;E=0;
30  #10;A=0;B=0;C=0;D=1;E=1;
31 end
32 endmodule
```

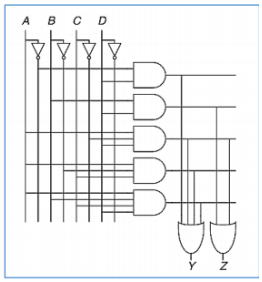


Technology schematic & bitstream generation:



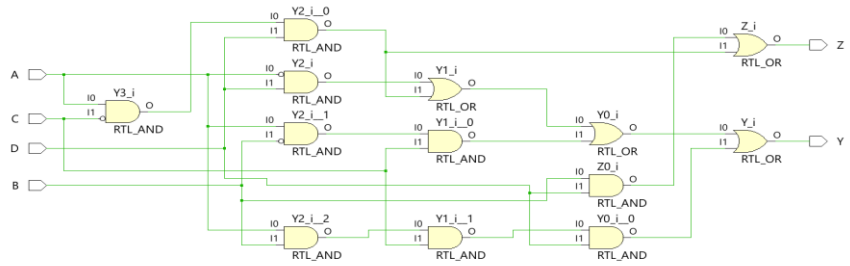


(iii) **Two output function:**



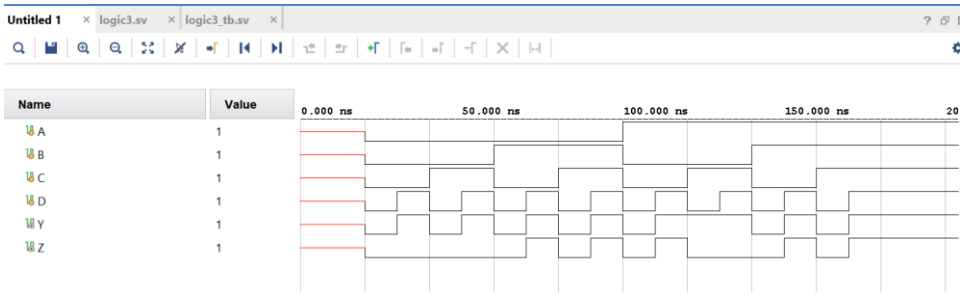
Design code & RTL Schematic:

```
21 module logic3(A,B,C,D,Y,Z);
22   input logic A,B,C,D;
23   output logic Y,Z;
24   assign Y= ( ~A&D | A&~C&D | A&~B&C | A&B&C&D );
25   assign Z= ( B&D | A&~C&D );
26 endmodule
27
```

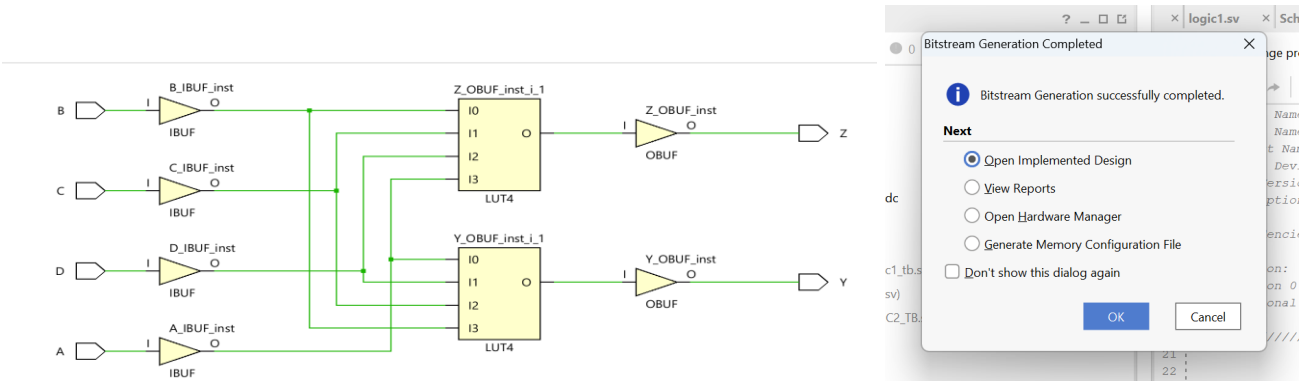


Test bench code & Waveform:

```
21 module logic3_tb;
22 reg A,B,C,D;
23 wire Y,Z;
24 logic3 dut(A,B,C,D,Y,Z);
25 initial
26 begin
27 #20;A=0;B=0;C=0;D=0;
28 #10;A=0;B=0;C=0;D=1;
29 #10;A=0;B=0;C=1;D=0;
30 #10;A=0;B=0;C=1;D=1;
31 #10;A=0;B=1;C=0;D=0;
32 #10;A=0;B=1;C=0;D=1;
33 #10;A=0;B=1;C=1;D=0;
34 #10;A=0;B=1;C=1;D=1;
35 #10;A=1;B=0;C=0;D=0;
36 #10;A=1;B=0;C=0;D=1;
37 #10;A=1;B=0;C=1;D=0;
38 #10;A=1;B=0;C=1;D=1;
39 #10;A=1;B=1;C=0;D=0;
40 #10;A=1;B=1;C=0;D=1;
41 #10;A=1;B=1;C=1;D=0;
42 #10;A=1;B=1;C=1;D=1;
43 end
44 endmodule
```



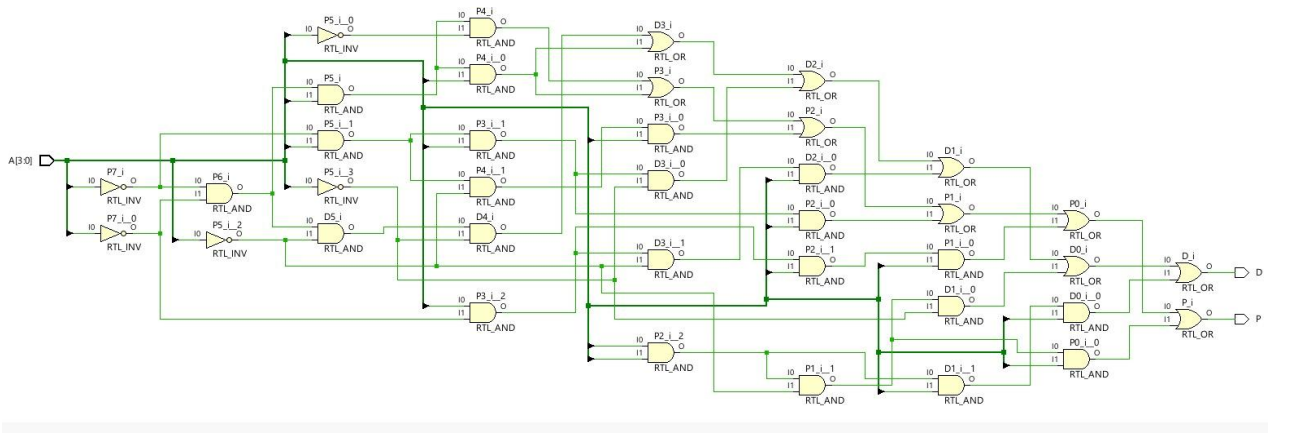
Technology schematic & Bitstream generation:



(iv) **A circuit has four inputs and two outputs. The inputs, A3:0, represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.**

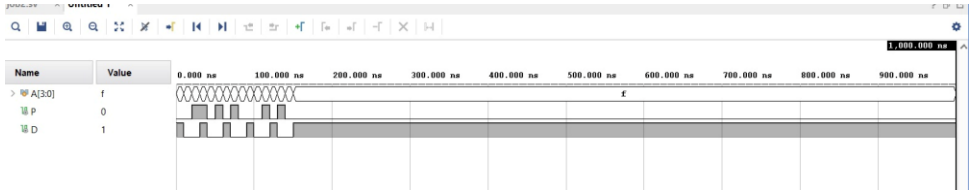
Design Code & RTL Schematic:

```
module PIME(
    input  logic [3:0] A,
    output logic P,D
);
    assign P = ((!A[3])&(!A[2])&A[1]&(!A[0]))|((!A[3])&(!A[2])&A[1]&A[0])|((!A[3])&A[2]&(!A[1])&A[0])|((!A[3])&A[2]&A[1]&A[0])|(A[3]&(!A[2])&A[1]&A[0])|(A[3]&A[2]&(!A[1])&A[0]);
    assign D=((!A[3])&(!A[2])&(!A[1])&(!A[0]))|((!A[3])&(!A[2])&A[1]&A[0])|((!A[3])&A[2]&A[1]&(!A[0]))|(A[3]&(!A[2])&(!A[1])&A[0])|(A[3]&A[2]&(!A[1])&(!A[0]))|(A[3]&A[2]&A[1]&A[0]);

endmodule
```

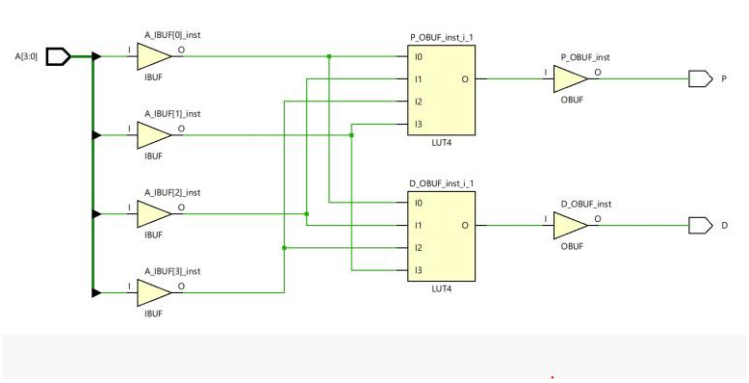


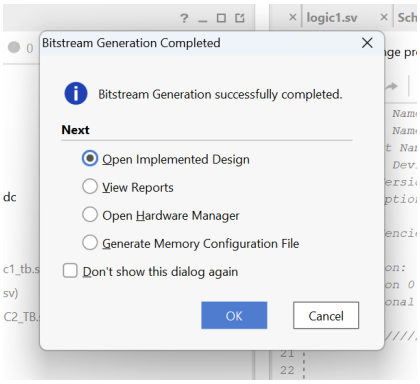
Testbench code & Waveform:

```
module PIME_tb;
    reg [3:0] A;
    wire P, D;
    PIME dut (A,P,D);
    initial
    begin
        A = 4'd0; #10;
        A = 4'd1; #10;
        A = 4'd2; #10;
        A = 4'd3; #10;
        A = 4'd4; #10;
        A = 4'd5; #10;
        A = 4'd6; #10;
        A = 4'd7; #10;
        A = 4'd8; #10;
        A = 4'd9; #10;
        A = 4'd10; #10;
        A = 4'd11; #10;
        A = 4'd12; #10;
        A = 4'd13; #10;
        A = 4'd14; #10;
        A = 4'd15; #10;
    end

endmodule
```



Technology schematic & Bitstream generation:

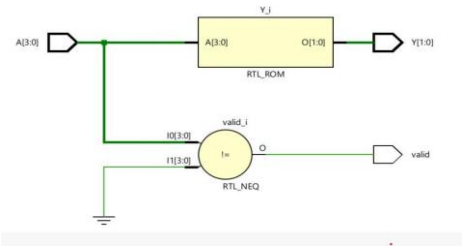




(v) **A four-input priority encoder:**

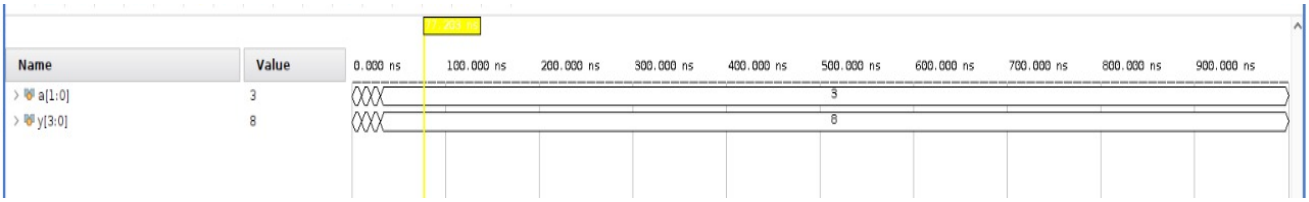
Design code and RTL Schematic:

```
21 module prio_enco(input logic [3:0] A, output logic [1:0] Y, output logic valid);
22   always_comb
23     casez(A)
24       4'b0001, 4'b001x, 4'b01xx, 4'b1xxx : Y = 2'b00;
25       4'b0010, 4'b0011, 4'b01xx, 4'b1xxx : Y = 2'b01;
26       4'b0100, 4'b0101, 4'b011x, 4'b1xxx : Y = 2'b10;
27       4'b1000, 4'b1001, 4'b101x, 4'b11xx : Y = 2'b11;
28       default : Y = 2'b00;
29     endcase
30   assign valid = (A != 4'b0000);
31 endmodule
```

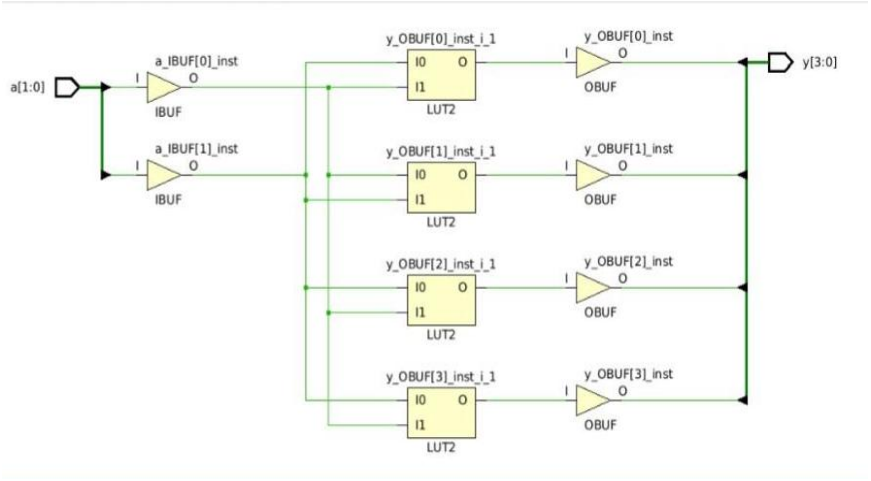


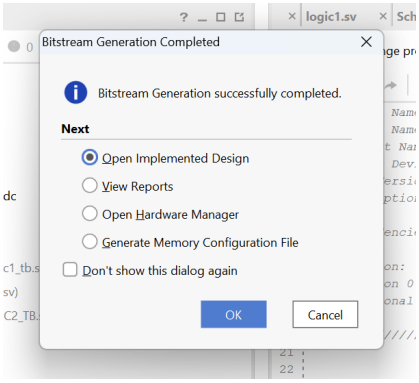
Test bench and Waveform:

```
21 module testbench;
22   reg [3:0] A;
23   wire [1:0] Y;
24   wire valid;
25
26   prio_enco DUT (A, Y, valid);
27
28   initial begin
29     A = 4'b0000; #10;
30     A = 4'b0001; #10;
31     A = 4'b0010; #10;
32     A = 4'b0100; #10;
33     A = 4'b1000; #10;
34     A = 4'b1100; #10;
35     A = 4'b1110; #10;
36     A = 4'b1111; #10;
37   end
38 endmodule
```



Technology schematic & Bitstream generation:

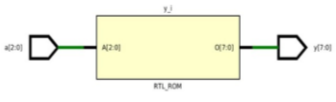




(vi) **An eight-input priority encoder:**

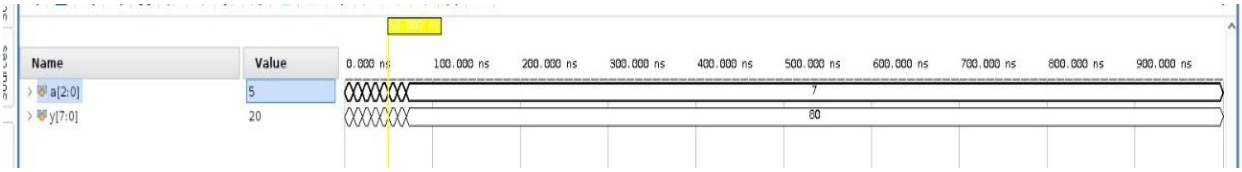
Design code and RTL Schematic:

```
module prio_enco(
  input logic [7:0] A,
  output logic [2:0] Y
);
  always_comb begin
    casez (A)
      8'b00000001, 8'b0000001x, 8'b000001xx, 8'b00001xxx, 8'b0001xxxx, 8'b001xxxxx, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b000;
      8'b00000010, 8'b0000011x, 8'b00001xxx, 8'b0001xxxx, 8'b001xxxxx, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b001;
      8'b00000100, 8'b000011xx, 8'b0001xxxx, 8'b001xxxxx, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b010;
      8'b00001000, 8'b00011xxx, 8'b001xxxxx, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b011;
      8'b00010000, 8'b001xxxxx, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b100;
      8'b00100000, 8'b01xxxxxx, 8'b1xxxxxxx: Y = 3'b101;
      8'b01000000, 8'b1xxxxxxx: Y = 3'b110;
      8'b10000000: Y = 3'b111;

      default: Y = 3'b000; // Default case for no valid input
    endcase
  end
endmodule
```

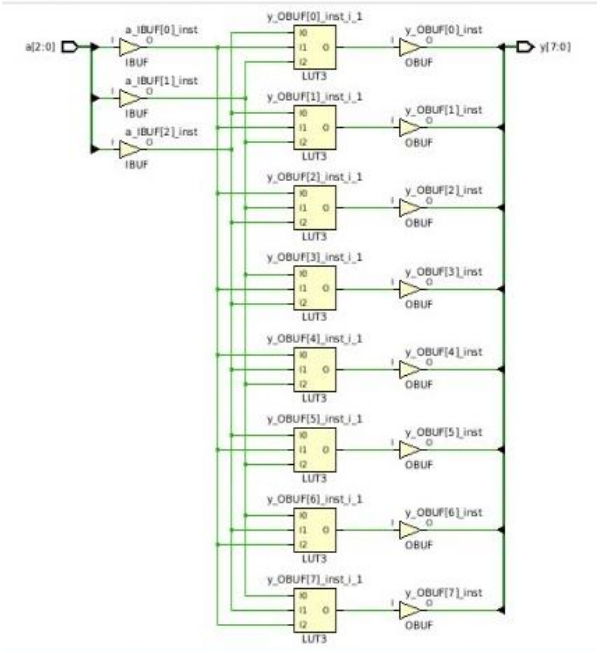


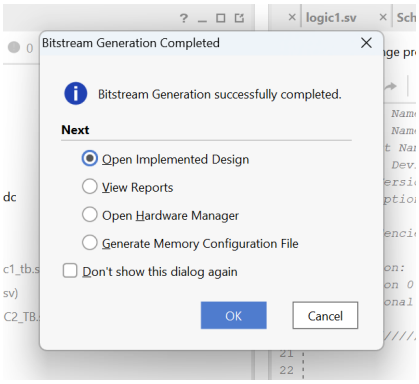
Test bench and waveform:

```
module testbench;
  reg [7:0] A;
  wire [2:0] Y;
  wire valid;
    prio_enco DUT (A, Y, valid);
  initial begin
    // Apply test cases
    A = 8'b00000000; #10;
    A = 8'b00000001; #10;
    A = 8'b00000010; #10;
    A = 8'b00000100; #10;
    A = 8'b00001000; #10;
    A = 8'b00010000; #10;
    A = 8'b00100000; #10;
    A = 8'b01000000; #10;
    A = 8'b10000000; #10;
    A = 8'b11111111; #10;
    $stop;
  end
endmodule
```



Technology schematic & Bitstream generation:

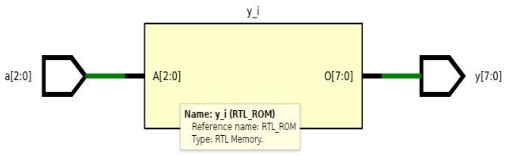




(vii) **A 3:8 decoder:**

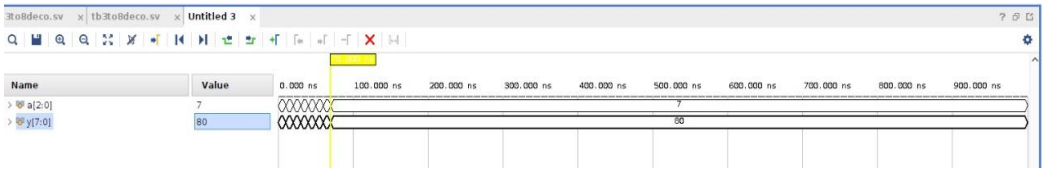
Design code & RTL schematic:

```
module deco38(input logic [2:0] a, output logic [7:0] y);
  always_comb
    case(a)
      3'b000 : y = 8'b00000001;
      3'b001 : y = 8'b00000010;
      3'b010 : y = 8'b00000100;
      3'b011 : y = 8'b00001000;
      3'b100 : y = 8'b00010000;
      3'b101 : y = 8'b00100000;
      3'b110 : y = 8'b01000000;
      3'b111 : y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

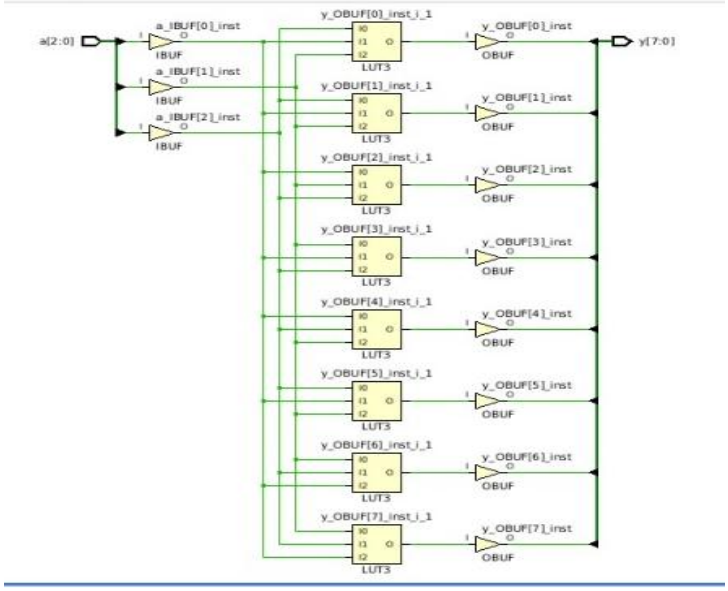


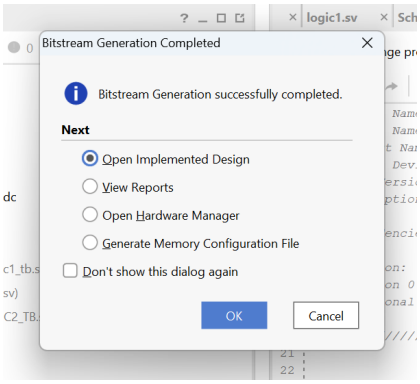
Testbench code & Waveform:

```
  module tb_deco38;

    logic [2:0] a;
    logic [7:0] y;

    deco38 dut (a, y);

    initial begin
      a = 3'b000; #10;
      a = 3'b001; #10;
      a = 3'b010; #10;
      a = 3'b011; #10;
      a = 3'b100; #10;
      a = 3'b101; #10;
      a = 3'b110; #10;
      a = 3'b111; #10;
    end
endmodule
```



Technology schematic & Bitstream generation:

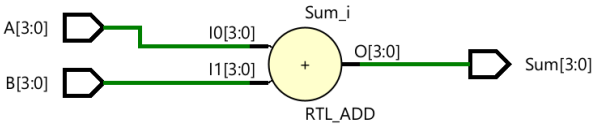




(viii) **A 4-bit carry propagate adder (with no carry in or out)**

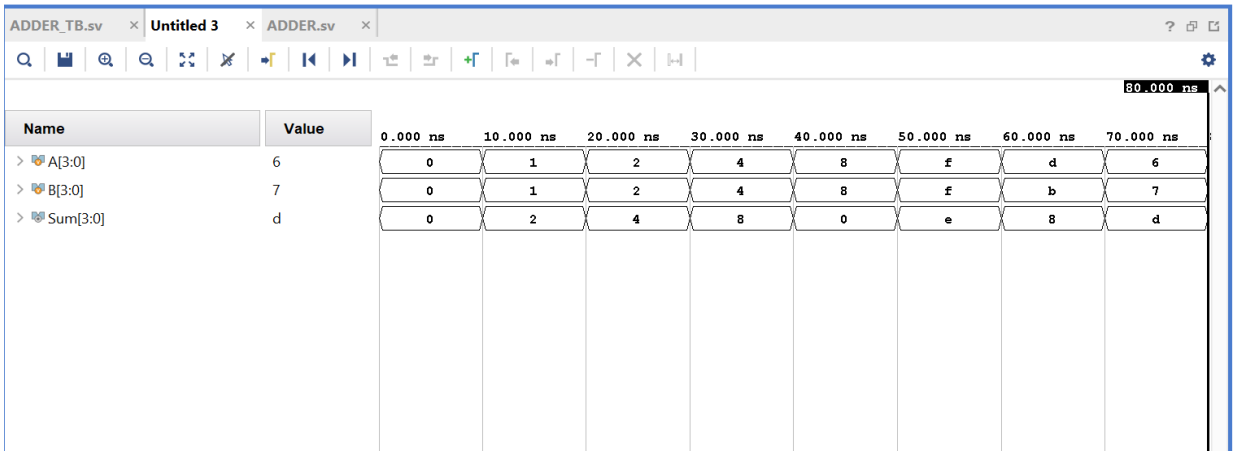
Design code & RTL Schematic:

```
21  module carry_propagate_adder(
22    input [3:0] A,
23    input [3:0] B,
24    output [3:0] Sum
25  );
26    assign Sum = A + B;
27  endmodule
```

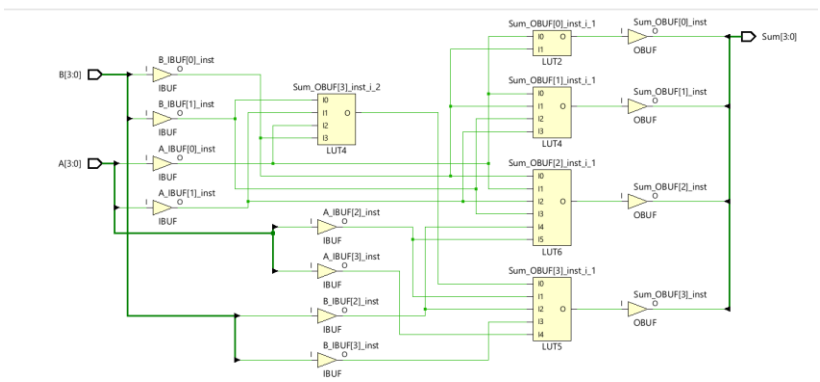


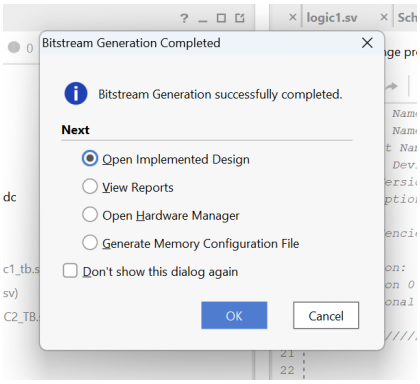
Testbench code & Waveform:

```
21  module carry_propagate_adder_tb;
22    reg [3:0] A, B;
23    wire [3:0] Sum;
24
25    carry_propagate_adder uut (
26      .A(A),
27      .B(B),
28      .Sum(Sum)
29    );
30
31    initial begin
32      A = 4'b0000; B = 4'b0000; #10;
33      A = 4'b0001; B = 4'b0001; #10;
34      A = 4'b0010; B = 4'b0010; #10;
35      A = 4'b0100; B = 4'b0100; #10;
36      A = 4'b1000; B = 4'b1000; #10;
37      A = 4'b1111; B = 4'b1111; #10;
38      A = 4'b1101; B = 4'b1011; #10;
39      A = 4'b0110; B = 4'b0111; #10;
40      $finish;
41    end
42  endmodule
```



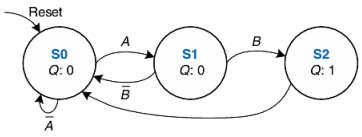
Technology schematic & bitstream generation:





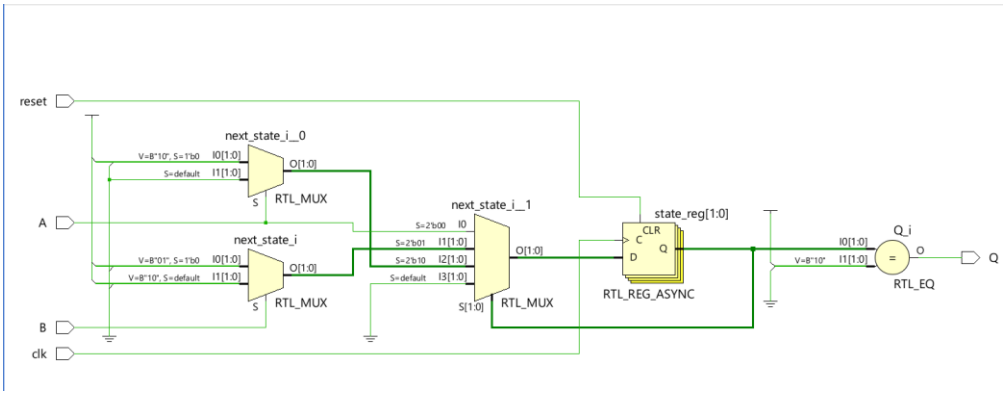
(ix) **The FSM:**

(a)



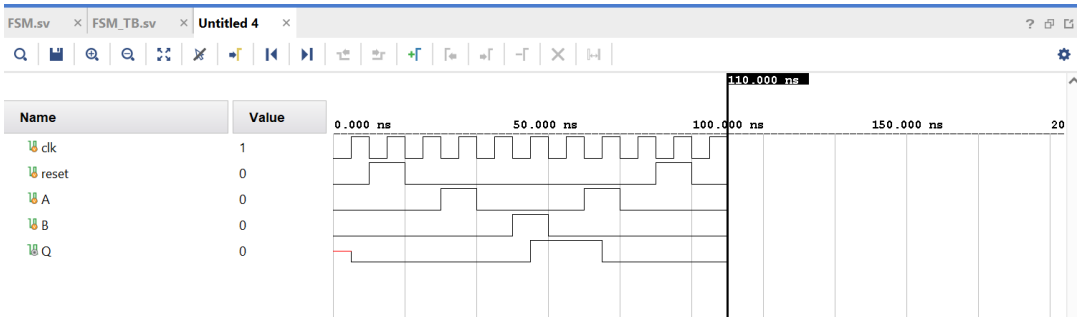
Design code & RTL Schematic:

```
module fsm(
  input logic clk,
  input logic reset,
  input logic A,
  input logic B,
  output logic Q
);
  typedef enum logic [1:0] {S0 = 2'b00, S1 = 2'b01, S2 = 2'b10} state_t;
  state_t state, next_state;

  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else
      state <= next_state;
  end
  always_ff @(state or A or B) begin
    case (state)
      S0:
        if (~A) next_state = S0;
        else next_state = S1;
      S1:
        if (~B) next_state = S1;
        else next_state = S2;
      S2:
        if (~A) next_state = S2;
        else next_state = S0;
      default: next_state = S0;
    endcase
  end
  assign Q = (state == S2);
endmodule
```

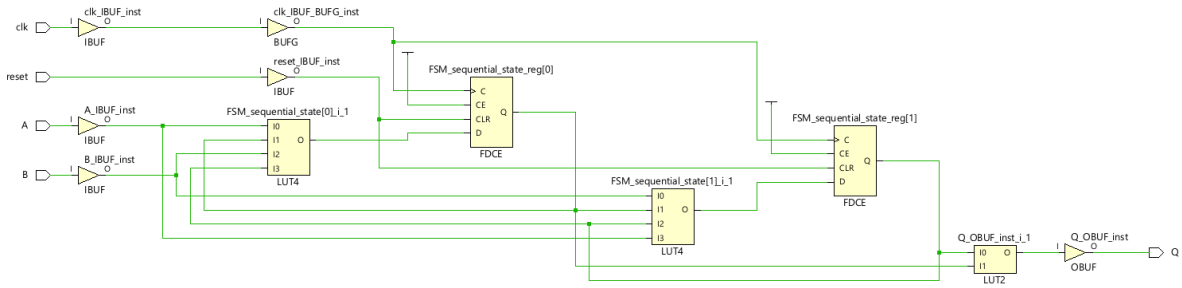


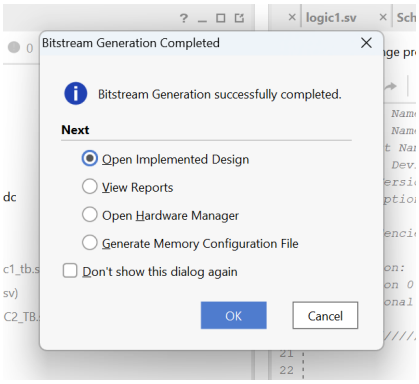
Test bench code & Waveform:

```
21  module fsm_tb;
22  reg clk,reset,A,B;
23  wire Q;
24  fsm uut(.clk(clk),.reset(reset),.A(A),.B(B),.Q(Q));
25  always #5 clk=~clk;
26  initial begin
27  clk=0;reset=0;A=0;B=0;
28  #10 reset=1;#10 reset=0;
29  #10 A=1;B=0;#10 A=0;B=0;
30  #10 A=0;B=1;#10 A=0;B=0;
31  #10 A=1;B=0;#10 A=0;B=0;
32  #10 reset=1;#10 reset=0;
33  #10 $finish;
34  end
35  endmodule
```



Technology schematic & Bitstream generation:



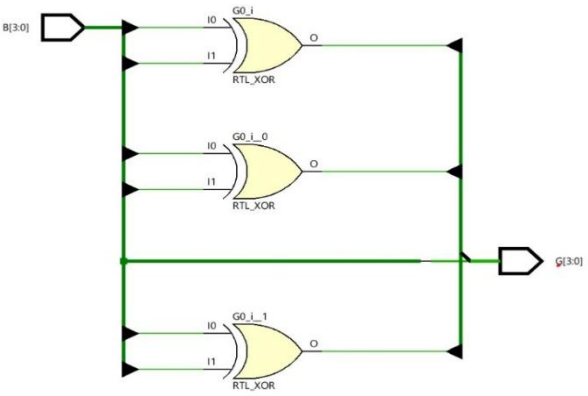


(b) Gray code counter:

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

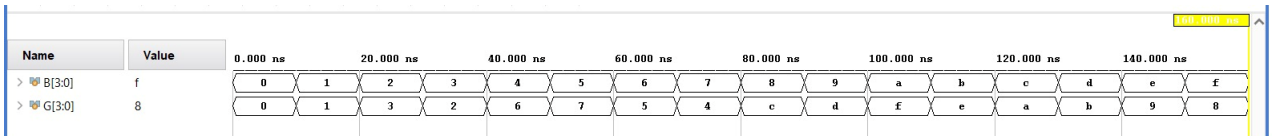
Design code & RTL Schematic:

```
21  module binary_to_gray (
22      input logic [3:0] B,
23      output logic [3:0] G
24  );
25
26      assign G[3] = B[3];
27      assign G[2] = B[3] ^ B[2];
28      assign G[1] = B[2] ^ B[1];
29      assign G[0] = B[1] ^ B[0];
30
31  endmodule
```

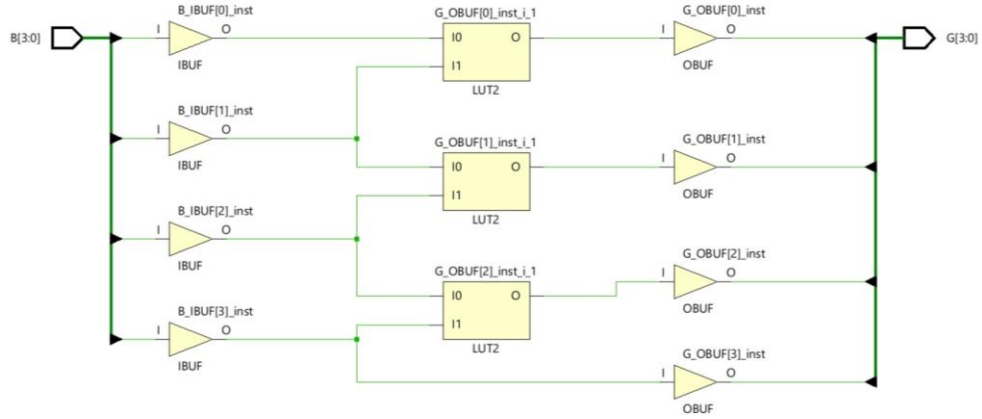


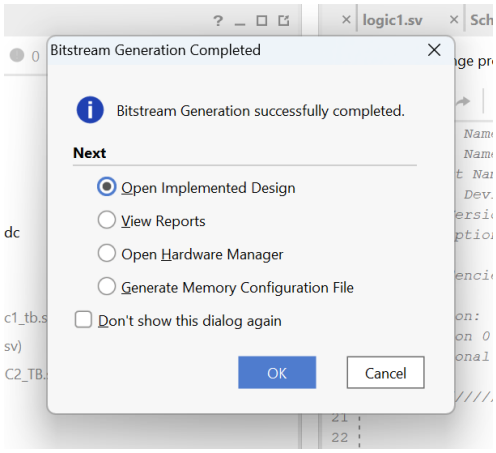
Test bench code & Waveform:

```
21  module tb_binary_to_gray();
22      logic [3:0] B;
23      logic [3:0] G;
24      binary_to_gray uut (
25          .B(B),
26          .G(G)
27      );
28      initial begin
29          B = 4'b0000; #10;
30          B = 4'b0001; #10;
31          B = 4'b0010; #10;
32          B = 4'b0011; #10;
33          B = 4'b0100; #10;
34          B = 4'b0101; #10;
35          B = 4'b0110; #10;
36          B = 4'b0111; #10;
37          B = 4'b1000; #10;
38          B = 4'b1001; #10;
39          $finish;
40      end
41  endmodule
```



Technology schematic & Bitstream generation:





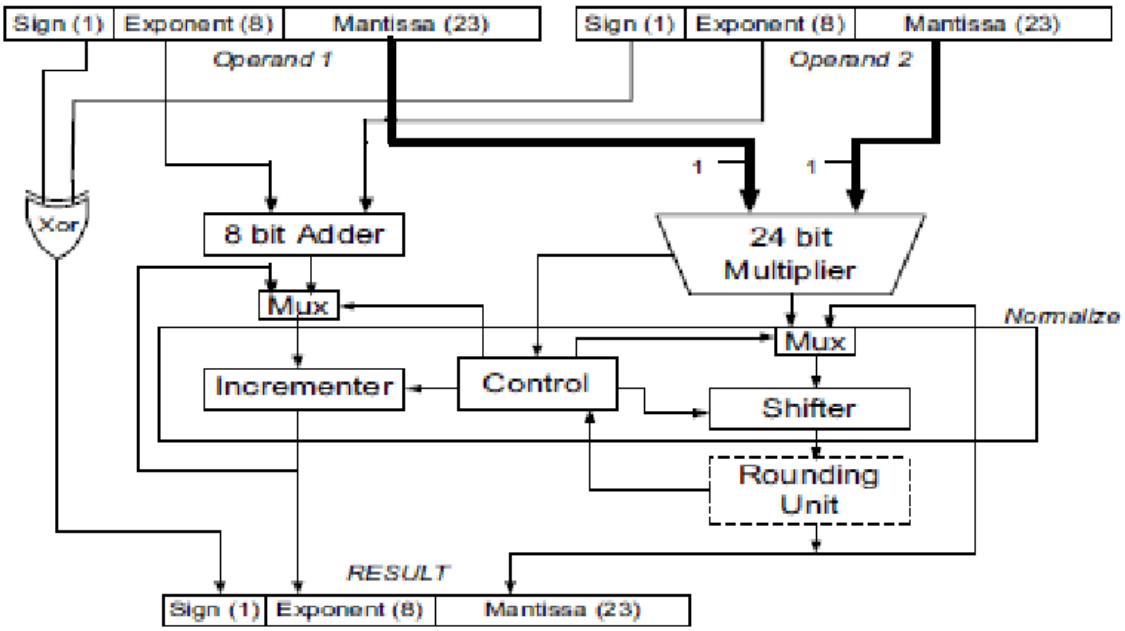
Q3. FPGA Flow – Floating point multiplier

(i)

(a) Write the steps necessary to perform 32-bit floating-point multiplication

1. Extract Components:
   - Parse the two 32-bit floating-point inputs into their components: sign (SSS), exponent (EEE), and mantissa (MMM).
   - Representation: Floating-point=(−1)S·1.M·2E\text{Floating-point} = (-1)^S \cdot 1.M \cdot 2^EFloating-point=(−1)S·1.M·2E
2. Handle Special Cases:
   - If any input is zero, the output is zero.
   - If an input is infinity or NaN, follow IEEE rules for special values.
3. Calculate Sign:
   - XOR the sign bits of the two inputs to compute the sign of the result.
4. Exponent Addition:
   - Add the exponents of the two inputs and subtract the bias (127127127).
5. Mantissa Multiplication:
   - Multiply the mantissas (M1×M2M1 \times M2M1×M2), accounting for the implicit leading 1.
   - The result is a 48-bit product. Normalize the result to ensure the leading bit is 1.
6. Truncate or Round:
   - Round or truncate the mantissa to fit into 23 bits.
7. Reassemble the Result:
   - Combine the calculated sign, adjusted exponent, and normalized/truncated mantissa into a 32-bit floating-point format.

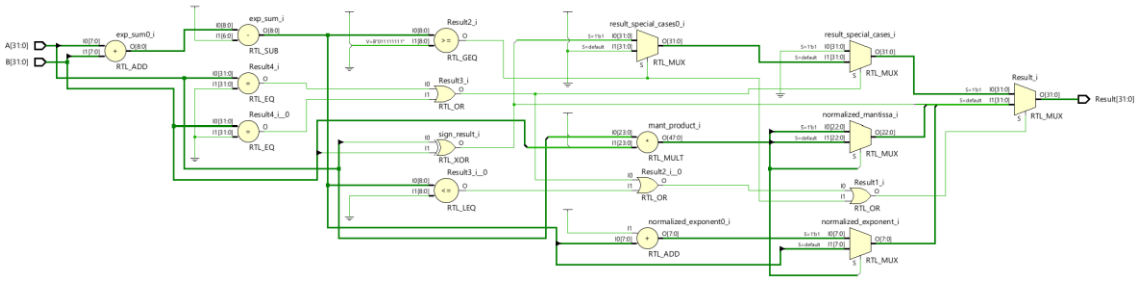
(b) Schematic of 32-bit floating point number



(c) Code:

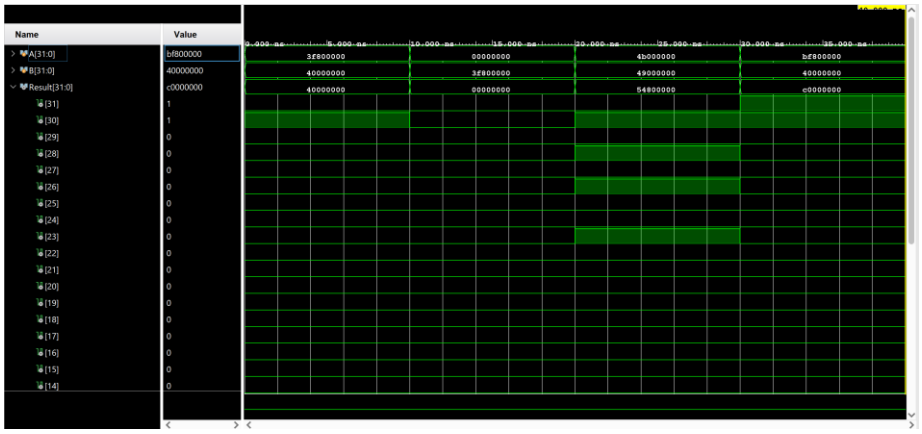
Design code & RTL Schematic:

```
module fp_multiplier (
    input [31:0] A, // 32-bit floating-point input A
    input [31:0] B, // 32-bit floating-point input B
    output reg [31:0] Result // 32-bit floating-point output
);
    // Extract components
    wire sign_a = A[31];
    wire sign_b = B[31];
    wire [7:0] exp_a = A[30:23];
    wire [7:0] exp_b = B[30:23];
    wire [23:0] mant_a = {1'b1, A[22:0]}; // Add implicit leading 1
    wire [23:0] mant_b = {1'b1, B[22:0]}; // Add implicit leading 1
// Compute sign
    wire sign_result = sign_a ^ sign_b;
// Add exponents and subtract bias (127)
    wire [8:0] exp_sum = exp_a + exp_b - 8'd127;
// Multiply mantissas
    wire [47:0] mant_product = mant_a * mant_b;
// Normalization logic
    reg [23:0] normalized_mantissa;
    reg [7:0] normalized_exponent;
    always @(*) begin
        if (mant_product[47]) begin
            normalized_mantissa = mant_product[46:23];
            normalized_exponent = exp_sum[7:0] + 1;
        end else begin
            normalized_mantissa = mant_product[45:22];
            normalized_exponent = exp_sum[7:0];
        end
    end
// Handle special cases
    wire [31:0] result_special_cases = (A == 0 || B == 0) ? 32'b0 :
                                       (exp_sum >= 8'd255) ? {sign_result, 8'd255, 23'b0} : // Infinity
                                       (exp_sum <= 8'd0) ? 32'b0 : 32'b0; // Underflow
// Assemble the result
    always @(*) begin
        if (A == 0 || B == 0 || exp_sum <= 8'd0 || exp_sum >= 8'd255)
            Result = result_special_cases;
        else
            Result = {sign_result, normalized_exponent, normalized_mantissa[22:0]};
    end
endmodule
```

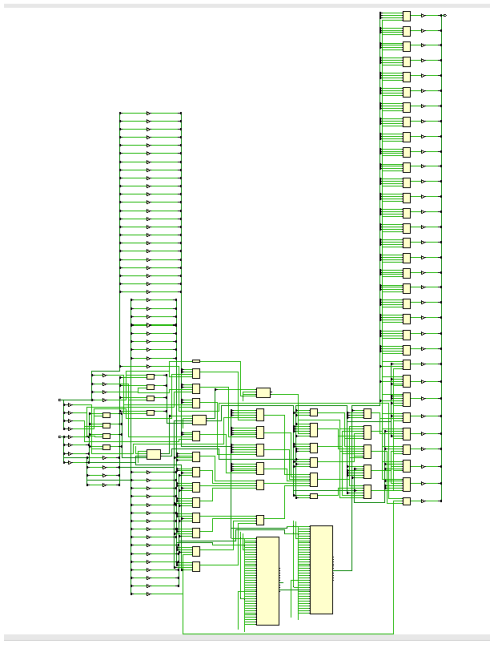


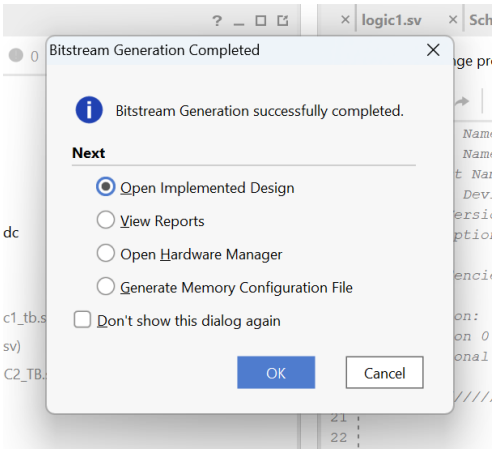
Test bench and waveform:

```
    module tb_fp_multiplier;
        reg [31:0] A, B;
        wire [31:0] Result;

        fp_multiplier uut (
            .A(A),
            .B(B),
            .Result(Result)
        );

        initial begin
            // Test 1: Multiply two small positive numbers
            A = 32'h3F800000; // 1.0
            B = 32'h40000000; // 2.0
            #10;
            $display("Test 1: 1.0 * 2.0 = %h", Result);

            // Test 2: Multiply by zero
            A = 32'h00000000; // 0.0
            B = 32'h3F800000; // 1.0
            #10;
            $display("Test 2: 0.0 * 1.0 = %h", Result);

            // Test 3: Multiply two large positive numbers
            A = 32'h4B000000; // Large number
            B = 32'h49000000; // Large number
            #10;
            $display("Test 3: Large * Large = %h", Result);

            // Test 4: Multiply with negative numbers
            A = 32'hBF800000; // -1.0
            B = 32'h40000000; // 2.0
            #10;
            $display("Test 4: -1.0 * 2.0 = %h", Result);

            // Finish simulation
            $finish;
        end
    endmodule
```



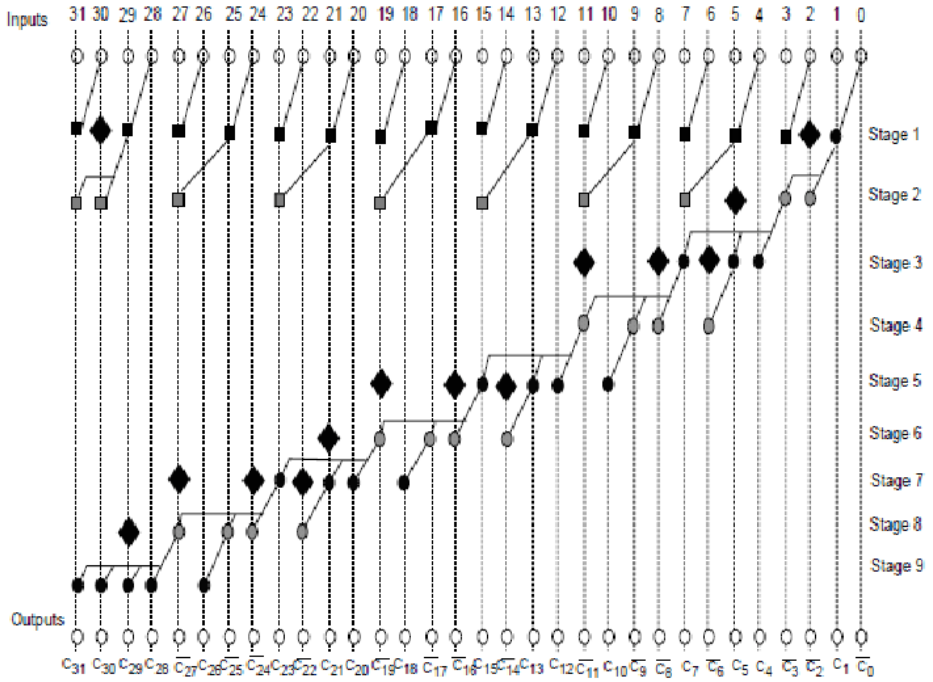
Technology schematic & bitstream generation:





(ii)

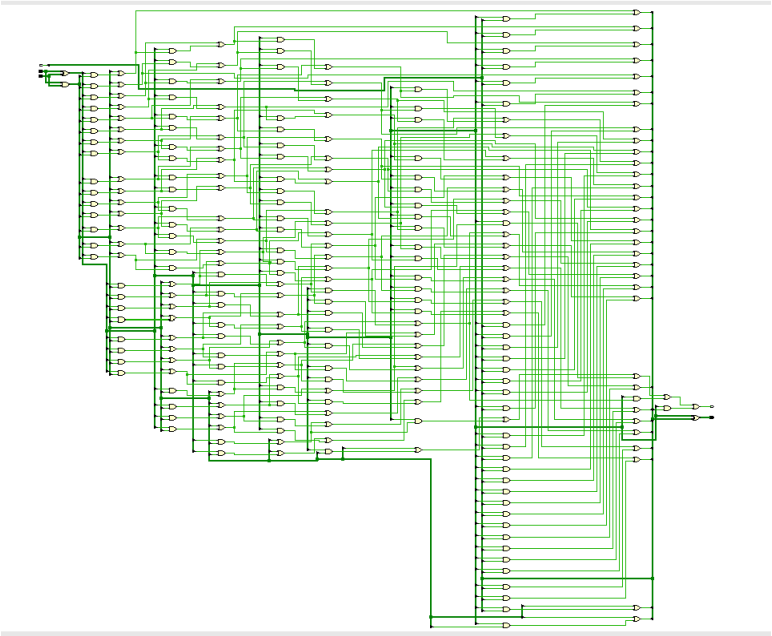
(a) Schematic of 32-bit prefix adder:



(c) Code:

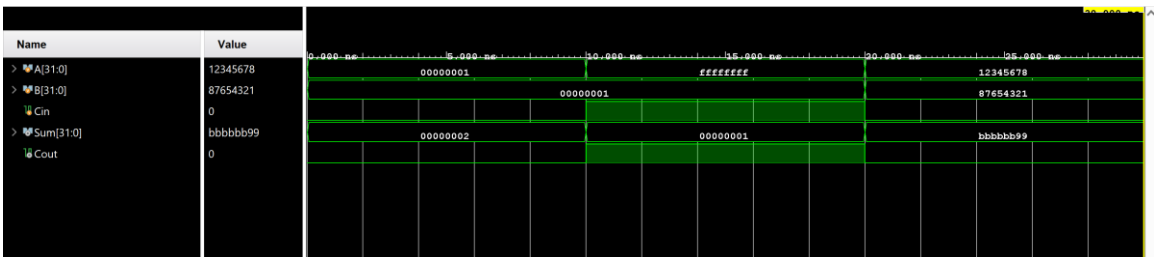
Design code & RTL Schematic:

```
module prefix_adder_32 (
    input [31:0] A, // 32-bit input A
    input [31:0] B, // 32-bit input B
    input Cin,      // Carry-in
    output [31:0] Sum, // 32-bit Sum output
    output Cout);      // Carry-out
    wire [31:0] P, G; // Propagate and Generate
    wire [31:0] C;    // Carry signals
    // Step 1: Generate Propagate and Generate signals
    assign P = A ^ B; // Propagate
    assign G = A & B; // Generate
    // Step 2: Build prefix tree for carry generation
    wire [31:0] C1, C2, C3, C4, C5;
    assign C1[0] = G[0] | (P[0] & Cin);
    genvar i;
    generate
        for (i = 1; i < 32; i = i + 1) begin...
    endgenerate
    assign C2[1] = C1[1];
    assign C2[0] = C1[0];
    generate
        for (i = 2; i < 32; i = i + 1) begin...
    endgenerate
    assign C3[3:0] = C2[3:0];
    generate
        for (i = 4; i < 32; i = i + 1) begin...
    endgenerate
    assign C4[7:0] = C3[7:0];
    generate
        for (i = 8; i < 32; i = i + 1) begin...
    endgenerate
    assign C5[15:0] = C4[15:0];
    generate
        for (i = 16; i < 32; i = i + 1) begin...
    endgenerate

    // Step 3: Generate carries and final sum
    assign C[0] = Cin;
    assign C[31:1] = C5[30:0];
    assign Cout = C5[31];
    assign Sum = P ^ C;

endmodule
```

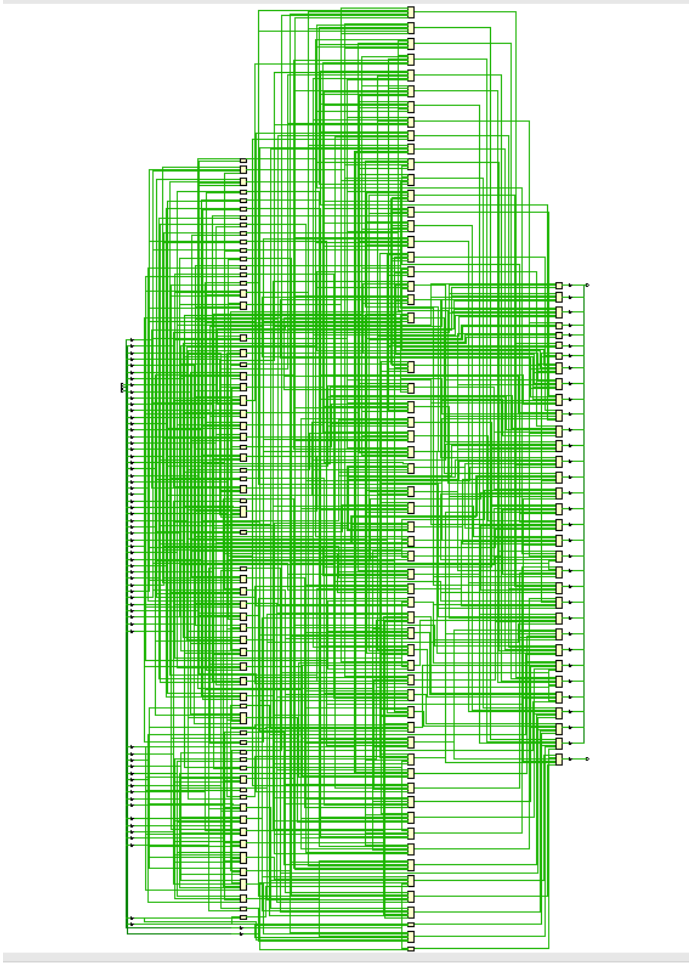


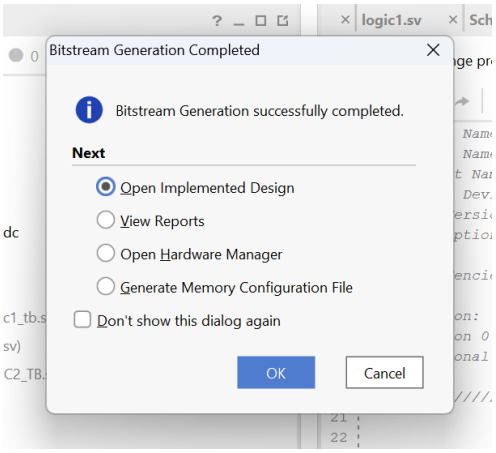
Test bench and waveform:

```
module tb_prefix_adder_32;
    reg [31:0] A, B;
    reg Cin;
    wire [31:0] Sum;
    wire Cout;
    // Instantiate the Prefix Adder
    prefix_adder_32 uut (.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout));

    initial begin
        // Test Case 1: Simple addition
        A = 32'h00000001;
        B = 32'h00000001;
        Cin = 1'b0;
        #10;
        $display("Test 1: A = %h, B = %h, Sum = %h, Cout = %b", A, B, Sum, Cout);

        // Test Case 2: Addition with carry-in
        A = 32'hFFFFFFFF;
        B = 32'h00000001;
        Cin = 1'b1;
        #10;
        $display("Test 2: A = %h, B = %h, Sum = %h, Cout = %b", A, B, Sum, Cout);

        // Test Case 3: Large numbers
        A = 32'h12345678;
        B = 32'h87654321;
        Cin = 1'b0;
        #10;
        $display("Test 3: A = %h, B = %h, Sum = %h, Cout = %b", A, B, Sum, Cout);

        // Finish simulation
        $finish;
    end
endmodule
```



Technology schematic and bitstream generation:





(iii) Delay Analysis

- Delay of Prefix Tree:
  - The Kogge-Stone adder requires log⁡2(32)=5\log_2(32) = 5log2(32)=5 levels of logic for carry propagation.
  - Each level involves a two-input gate with a delay of 100 ps.
  - Total delay = 5×100=5005 \times 100 = 5005×100=500 ps.

(iv) Pipelined 32-Bit Prefix Adder

Fastest Frequency:

- Critical path delay per stage = 100 ps.
- Maximum frequency = 10GHz

(v) Schematic for pipelined 32 bit prefix adder:

```
module pipelined_prefix_adder_32 (
    input [31:0] A, B,
    input Cin,
    output reg [31:0] Sum,
    output reg Cout,
    input clk, reset
);
    // Internal pipeline registers
    reg [31:0] P1, G1, P2, G2, P3, G3, P4, G4, P5, G5;
    reg Cin1, Cin2, Cin3, Cin4, Cin5;

    // Pipeline Stage 1: Generate Propagate and Generate signals
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            P1 <= 0; G1 <= 0; Cin1 <= 0;
        end else begin
            P1 <= A ^ B;
            G1 <= A & B;
            Cin1 <= Cin;
        end
    end

    // Pipeline Stage 2: Compute Level 1 carry
    // Add further stages as shown in part (a) for pipelined generation
    // ...
endmodule
```