Digital Systems Assignment: Encoder and Adder Design

Sai Keerthana , Amrutha Peddinte

Roll no's: 23110229 , 23110243

January 27, 2025

Problem 1: 8-Bit Priority Encoder Design

Implement an 8-bit priority encoder design. The encoder should prioritize the highest active input when multiple inputs are high. The encoder outputs a 3-bit binary code corresponding to the highest active input.

Note: We have designed a 3x8 decoder with Enable. The decoder IC takes in 3 inputs and generates 8 outputs. Encoders are complementary ICs that take in 8-bit inputs and generate 3-bit output. The problem with the encoder is that if two inputs are "1" (active), it's unclear which one should be prioritized. Hence, encoders have to be designed as priority encoders. You may say that if input "1" and "4" are active, "4" will be prioritized over "1." In other words, since you have 8-inputs, the 1st input will have the least priority, and the 8th will have the highest priority.

Verilog Code

Below is the Verilog code for the 8-bit priority encoder:

Listing 1: 8-Bit Priority Encoder Verilog Code

```
module home_assign_1(
    input [7:0] I,
    output reg [2:0] O
);

    always @(*)
    begin
        if (I[7])
            O = 3'b111;
        else if (I[6])
            O = 3'b110;
        else if (I[5])
            O = 3'b101;
        else if (I[4])
            O = 3'b100;
        else if (I[3])
            O = 3'b011;
        else if (I[2])
            O = 3'b010;
        else if (I[1])
            O = 3'b001;
        else if (I[0])
            O = 3'b000;
        else
            O = 3'b000;
```

```
    end
endmodule
```

Testbench Code

Below is the testbench code used to simulate the priority encoder:

Listing 2: Testbench Code for 8-Bit Priority Encoder

```
module home_assign_1_tb();
    reg [7:0] I;
    wire [2:0] O;

    home_assign_1 uut (
        .I(I),
        .O(O)
    );

    initial begin
        I = 8'b00000000; #10;
        I = 8'b00000001; #10;
        I = 8'b00000010; #10;
        I = 8'b00000100; #10;
        I = 8'b00001000; #10;
        I = 8'b00010000; #10;
        I = 8'b00100000; #10;
        I = 8'b01000000; #10;
        I = 8'b10000000; #10;
        I = 8'b10000101; #10;
        I = 8'b01010101; #10;
        I = 8'b11000000; #10;
        I = 8'b01100010; #10;
        I = 8'b01001100; #10;
        I = 8'b00110010; #10;
        I = 8'b00010010; #10;
        I = 8'b00001110; #10;
        I = 8'b00000101; #10;
        I = 8'b00000011; #10;
        I = 8'b00000010; #10;
        I = 8'b00000001; #10;
        $stop();
    end
endmodule
```

Explanation:

The encoder provides a 3-bit binary code (O[2:0]) as the output for the highest active input after verifying eight inputs (I[7:0]). It prioritizes from I[7] (highest) to I[0] (lowest). O = 3'b000 if no inputs are active.

For instance:

1) If I = 8'b00100000 (only I[5] is active), O = 3'b101.

2) If multiple inputs are active (e.g., I = 8'b01000110), the highest (I[6]) is prioritized, so O = 3'b110.

The testbench applies different inputs to ensure correct functionality.

Simulation Results

Below is the simulation output of the priority encoder, showing how the encoder prioritizes the highest active input.

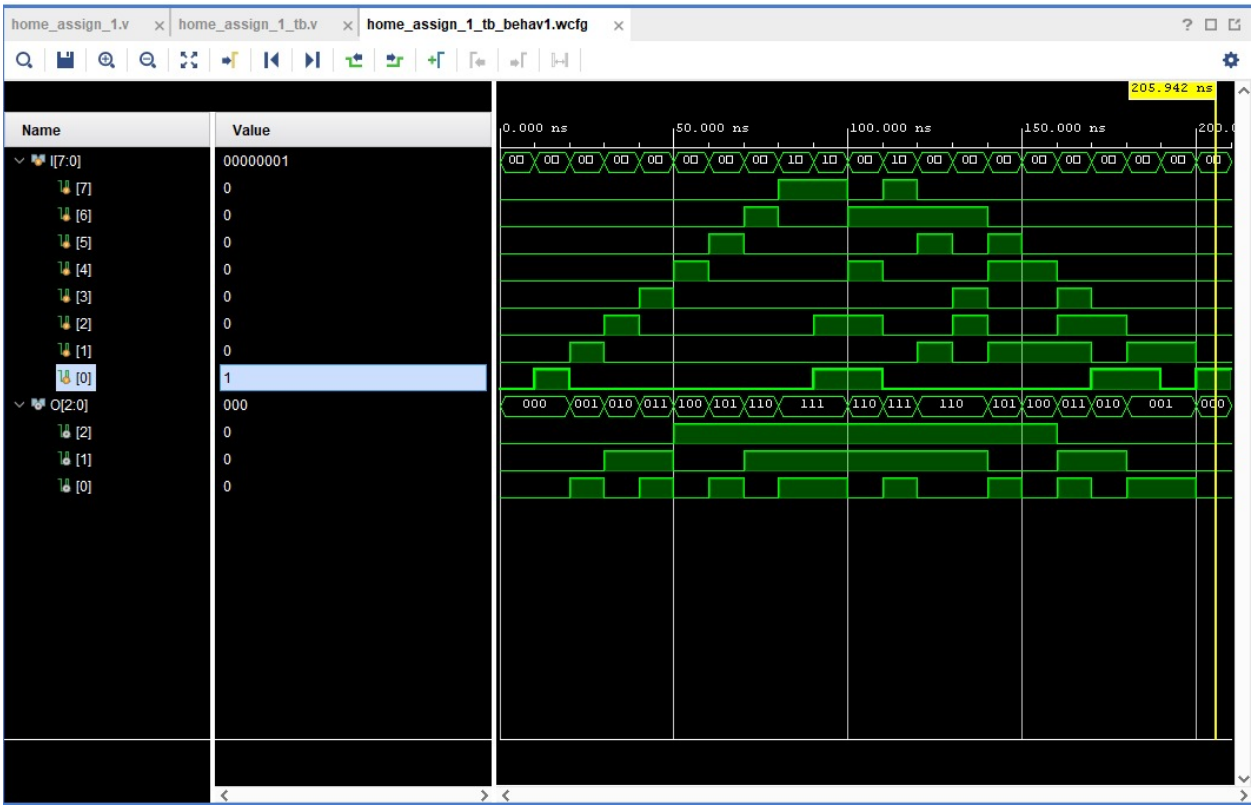

Figure 1: Simulation results for the 8-bit Priority Encoder.

Problem 2: 4-Bit Carry Lookahead Adder Design

Design a 4-bit Carry Lookahead Adder (CLA).

Verilog Code

Below is the Verilog code for the 4-bit Carry Lookahead Adder:

Listing 3: 4-Bit Carry Lookahead Adder Verilog Code

```
module home_assign_1(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] Sum,
    output Cout
);
wire [3:0] G,P;
wire [4:0] C;

    assign G= A&B;
    assign P= A^B;

    assign C[0]=Cin;
    assign C[1]=G[0]|(P[0]&C[0]);
    assign C[2]=G[1]|(P[1]&C[1]);
    assign C[3]=G[2]|(P[2]&C[2]);
    assign C[4]=G[3]|(P[3]&C[3]);

    assign Sum= P^C[3:0];

    assign Cout=C[4];
endmodule
```

Testbench Code

Below is the testbench code used to simulate the 4-bit Carry Lookahead Adder:

Listing 4: Testbench Code for 4-Bit Carry Lookahead Adder

```
module home_assign_1_tb();
    reg [3:0] A,B;
    reg Cin;
    wire [3:0] Sum;
    wire Cout;
    home_assign_1 uut(
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );
    initial begin
        A=4'b0000;B=4'b0000;Cin=0;#10;
        A=4'b0001;B=4'b0001;Cin=0;#10;
        A=4'b1111;B=4'b1111;Cin=0;#10;
```

```
        A=4'b1010;B=4'b0101;Cin=1;#10;
        A=4'b1100;B=4'b1010;Cin=1;#10;
        A=4'b1110;B=4'b0111;Cin=0;#10;
        A=4'b1110;B=4'b0111;Cin=0;#10;
        A=4'b1110;B=4'b0111;Cin=1;#10;
        A=4'b1111;B=4'b1111;Cin=1;#10;
        A=4'b1001;B=4'b1110;Cin=1;#10;
        A=4'b0011;B=4'b1001;Cin=0;#10;
        A=4'b1101;B=4'b0010;Cin=0;#10;
        A=4'b0011;B=4'b1100;Cin=1;#10;
        A=4'b0110;B=4'b1001;Cin=1;#10;
    end
endmodule
```

Explanation:

The Carry Lookahead Adder calculates the carry using 'Generate' (G) and 'Propagate' (P) logic, enabling faster addition than a ripple-carry adder. The sum is computed as $P \oplus C$ and the carry-out is derived using a series of logical AND and OR operations.

For example:

1) A = 4'b0110, B = 4'b1010, Cin = 1. Outputs: Sum = 4'b0011, Cout = 1.

2) A = 4'b1111, B = 4'b1111, Cin = 0. Outputs: Sum = 4'b1110, Cout = 1.

The testbench verifies the adder's operation with various input combinations.

Simulation Results

Below is the simulation output of the Carry Lookahead Adder, demonstrating its efficient addition process.

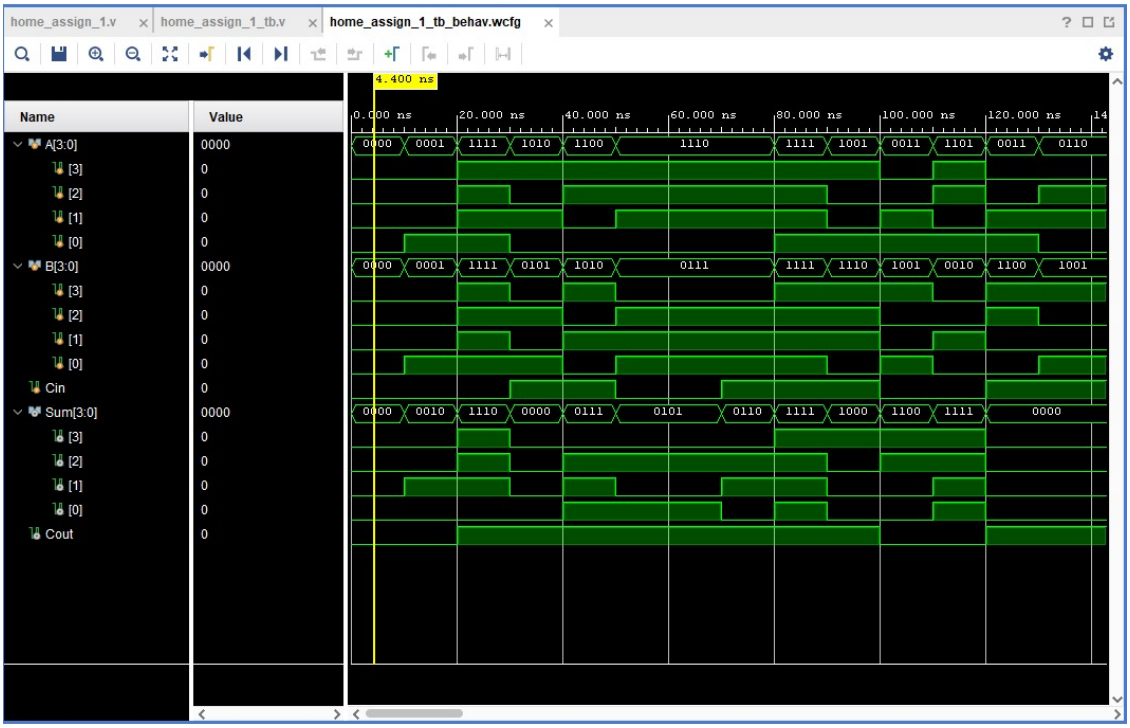

Figure 2: Simulation results for the 4-bit Carry Lookahead Adder.