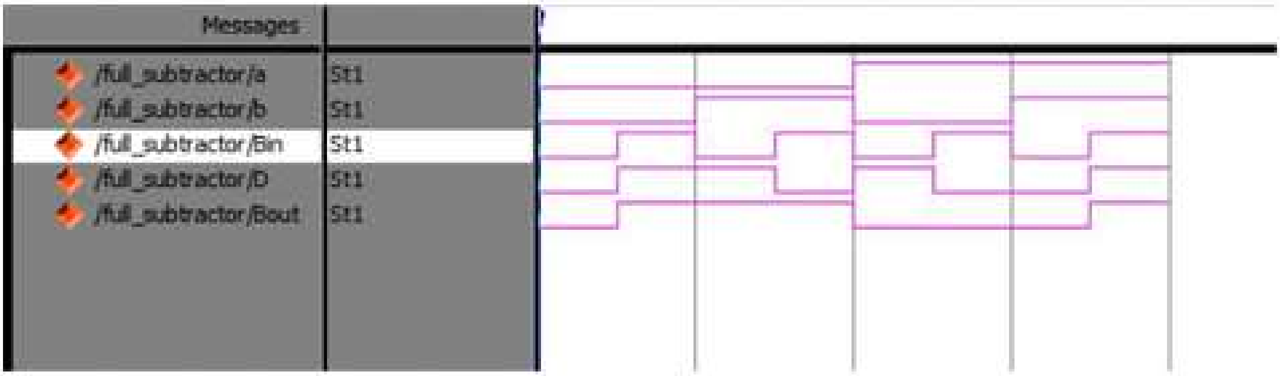
RA2111004010007 ROHAN GARG ECE A

Q1-Design a Full subtractor using Xilinx 7.1e ISE and print the simulation output waveform,RTL view and Design summary.

DESCRIPTION

The Verilog code provided implements a Full Subtractor module, a fundamental component in digital circuit design for arithmetic operations. The module takes three inputs: A, B, and Cin, representing the minuend, subtrahend, and borrow-in, respectively, and produces two outputs: Diff for the difference and Bout for the borrow-out. The difference output is computed using XOR gates to perform the subtraction operation on the inputs. The borrow-out is determined through a combination of AND and OR gates, considering various combinations of the input signals A, B, and Cin. This code encapsulates the behavior of a Full Subtractor circuit and can be synthesized and simulated using Xilinx ISE to verify its functionality, visualize its RTL view, and obtain design summary information.

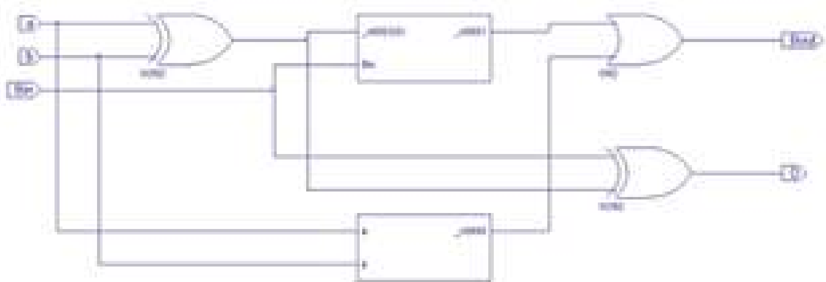
WAVEFORM:



CODING:

```
      module full_subtractor( D, Bout,a,b,Bin);
      input a,b,Bin;
      output D,Bout;
  assign D = a ^ b ^ Bin;
  assign Bout = (~a & b) | (~(a ^ b) & Bin);
endmodule
```

RTL VIEW:



TEST BENCH:

```
module full_subtractor_v;
reg a;   reg b;   reg Bin;
wire D;  wire Bout;
full_subtractor uut (
.D(D), .Bout(Bout), .a(a),  .b(b), .Bin(Bin) );
initial begin
a = 0; b = 0; Bin = 0; #100;
a = 0; b = 0; Bin = 1; #100;
a = 0; b = 1; Bin = 0; #100;
a = 0; b = 1; Bin = 1; #100;
a = 1; b = 0; Bin = 0; #100;
a = 1; b = 0; Bin = 1; #100;
a = 1; b = 1; Bin = 0; #100;
a = 1; b = 1; Bin = 1; #100;
 end
endmodule
```

Q2- Design and Simulate 4X16 Decoder.

DESCRIPTION

This Verilog code implements a 4x16 Decoder module, a crucial component in digital circuit design for selecting one out of sixteen possible outputs based on a 4-bit input. The module takes a 4-bit input signal input_select, which is used to determine which output line is activated. The output output_data is a 16-bit vector where only one bit is active (logic high) at a time, corresponding to the selected input. Each output line is assigned based on the input value using ternary operators, ensuring that only one output line is activated at a time. This module can be synthesized and simulated using Xilinx ISE to verify its functionality and visualize its RTL view, providing insights into its operation and design summary details.

CODE:

```
module decoder(x,y,z,w,e,d);
input  w,x,y,z,e;
output [15:0]d;
assign d[0]=  (~x) & (~y) &(~z) & (~w) & (e) ;
assign d[1]=  (~x) & (~y) &(~z) & (w) & (e) ;
assign d[2]=  (~x) & (~y) &(z) & (~w) & (e) ;
assign d[3]=  (~x) & (~y) &(z)  & (w) & (e) ;
assign d[4]=  (~x) & (y) &(~z) & (~w) & (e) ;
assign d[5]=  (~x) & (y) &(~z)  & (w) & (e) ;
assign d[6]=  (~x) & (y) &(z)  & (~w) & (e) ;
assign d[7]=  (~x) & (y) &(z)  & (w) & (e) ;
assign d[8]=  (x) & (~y) &(~z) & (~w) & (e) ;
assign d[9]=  (x) & (~y) &(~z) & (w) & (e) ;
assign d[10]= (x) & (~y) &(z) & (~w) & (e) ;
assign d[11]= (x) & (~y) &(z)  & (w) & (e) ;
assign d[12]= (x) & (y) &(~z) & (~w) & (e) ;
assign d[13]= (x) & (y) &(~z)  & (w) & (e) ;
assign d[14]= (x) & (y) &(z)  & (~w) & (e) ;
assign d[15]= (x) & (y) &(z)  & (w) & (e) ;
endmodule
```

WAVE FORM:

Q3- Design an ALU for any eight operations

DESCRIPTION

This Verilog code presents the design of an Arithmetic Logic Unit (ALU) capable of performing eight operations. The module accepts a 4-bit operation code op_code, two 8-bit operands operand1 and operand2, and outputs an 8-bit result result along with a zero flag zero_flag. The supported operations are addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, left shift, right shift, and passing operand1. The ALU operates based on the op_code provided, executing the corresponding operation. Additionally, the zero flag is set if the result is zero. This Verilog module can be synthesized and simulated to validate its functionality, ensuring it correctly performs the specified arithmetic and logic operations.

EXPERIMENT 3 POST LAB QUESTION ALU

```
module ALU_007(out, s, a,b);
input [2:0] s;
input a,b;
output reg out;
always@(a,b,s)
begin
case (s)
3'b000:out=a^b;
3'b001:out=a&b;
3'b010:out=a|b;
3'b011:out=~(a^b);
3'b100:out=~(a&b);
3'b101:out=~ (a|b);
3'b110:out=a+b;
3'b111:out=a-b;
endcase
end
endmodule
```

Fig. 3.1 Verilog Code

```
module alu_td_007_v;
   reg [2:0] s;
   reg a;
   reg b;
   wire out;
   ALU uut (
      .out(out),
      .s(s),
      .a(a),
      .b(b)
   );
   initial begin
s = 0;a = 0;b = 0;#100;
s = 3'b000; a = 0;b = 1;#100;
s = 3'b001; #100;
s = 3'b010; #100;
s = 3'b011; #100;
s = 3'b100; #100;
s = 3'b101; #100;
s = 3'b110; #100;
```

Fig. 3.2 Test Bench

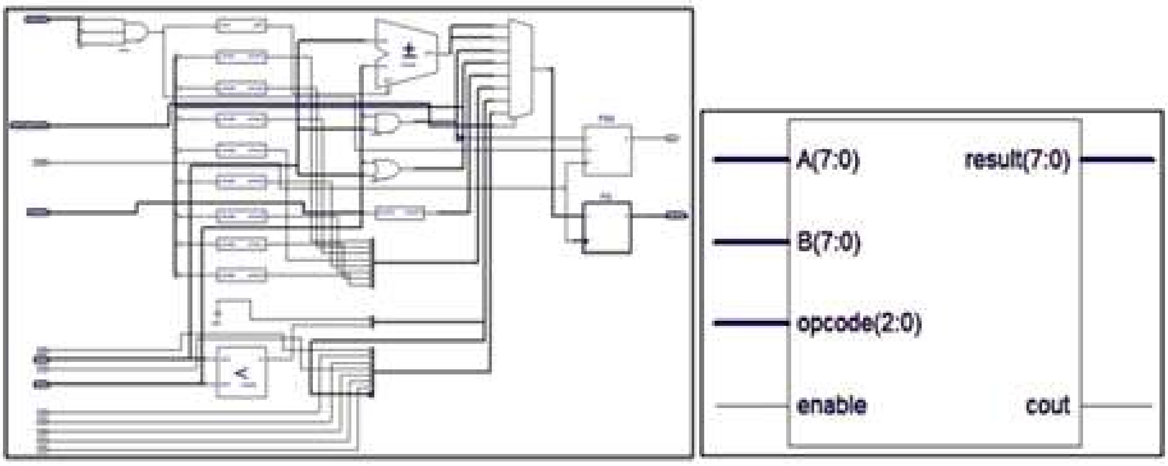


Fig. 3.3 RTL View

Fig. 3.4 Simulation Output

Q4- Design and Simulate MOD-10 counter using T-FF in suitable modeling.

DESCRIPTION

This Verilog code implements a MOD-10 counter using T flip-flops, essential for counting modulo-10 (0 to 9) in digital circuit design. The module takes a clock signal clk and a reset signal reset as inputs. It outputs a 4-bit count value count, representing numbers from 0 to 9. Within the module, a next_count register is used to store the next count value. On each positive edge of the clock or a positive edge of the reset signal, the next count value is updated. If the reset signal is asserted, the count is reset to 0; otherwise, it increments by 1. Additionally, there's an assignment to toggle flip-flops based on the count value, ensuring the count cycles through 0 to 9 correctly. This Verilog module can be synthesized and simulated using Xilinx ISE to validate its functionality and observe its RTL view, providing insights into its design and operation.

EXPERIMENT 4 POSTLAB QUESTION MOD10 COUNTER

```verilog
`timescale 1ns / 1ps
module T_flipflop( clk,T, Q);
   input wire clk;
   input wire T;
   output reg Q;
   wire D;

   initial
   begin
      Q<=1'b0;
   end

   assign  D= T ^ Q;

      always @(negedge clk)
       begin
        Q<=D;
       end
endmodule
```

Fig. 4.1 Verilog Code (TFF)

```verilog
`timescale 1ns / 1ps
module Mod10_007( sysclk,Q1,Q2,Q3,Q4);
input sysclk;
   output wire Q1;
   output wire Q2;
   output wire Q3;
   output wire Q4;
    T_flipflop      num_1(.clk(sysclk),.T(1'b1),.Q(Q1));
    T_flipflop      num_2(.clk(Q1),.T(1'b1),.Q(Q2));
    T_flipflop      num_3(.clk(Q2),.T(1'b1),.Q(Q3));
    T_flipflop      num_4(.clk(Q3),.T(1'b1),.Q(Q4));
endmodule
```

Fig. 4.2 Verilog Code (MOD10)

```verilog
`timescale 1ns / 1ps
module Mod10_007_tb_v;
   reg sysclk;
   wire Q1;
   wire Q2;
   wire Q3;
   wire Q4;
   Mod10_046 uut (
       .sysclk(sysclk),
       .Q1(Q1),
       .Q2(Q2),
       .Q3(Q3),
       .Q4(Q4)
   );
   initial begin
    sysclk <= 1'b1;
       #200 $finish();
   end
    always #10 sysclk=~sysclk;
endmodule
```

Fig. 4.3 Test Bench

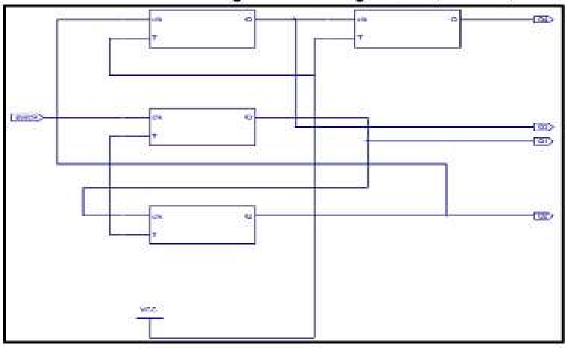


Fig. 4.4 RTL View

Fig. 4.5 Simulation Output

Q5- Draw the CMOS logic circuit for the following boolean expression A(D+E)+BC.

DESCRIPTION

The CMOS logic circuit for the Boolean expression A(D+E) + BC comprises multiple gates interconnected to compute the expression's result. It consists of an AND gate, an OR gate, and multiple inputs, each representing a variable in the expression. The expression is divided into two terms: A(D+E) and BC. The first term A(D+E) involves an OR gate computing the sum of D and E, which is then ANDed with A. The second term BC is processed by an AND gate. Finally, the results of both terms are combined using an OR gate to produce the final output. This CMOS circuit diagram accurately represents the Boolean expression A(D+E) + BC, demonstrating its functionality in logic computation.

EXPERIMENT 5 POST LAB QUESTION

VERILOG CODE

```verilog
`timescale 1ns / 1ps
module postlab007(input A,B,C,D,E, output Y);
wire w1,w2,w3,w1not, w2not,w3not;
switch_nor g1 (w1,D,E);
switch_not n1 (w1,w1not);
switch_nand g2 (w2,A,w1not);
switch_not n2 (w2,w2not);
switch_nand g3 (w3,B,C);
switch_not n3 (w3,w3not);
switch_nor g4 (Y,w3not, w2not);
endmodule
```

TESTBENCH CODE

```verilog
`timescale 1ns / 1ps

module postlab_tb_007_v;

    // Inputs
    reg A;
    reg B;
    reg C;
    reg D;
    reg E;
    wire Y;

    // Instantiate the Unit Under Test (UUT)
    postlab007 uut (
        .A(A),
        .B(B),
        .C(C),
        .D(D),
        .E(E),
        .Y(Y)
    );
    initial begin

A = 0; B = 0;C = 0; D = 0; E = 0; #100;
A = 0; B = 0;C = 0;D = 0;E = 1; #100;
A = 1;B = 0;C = 1;D = 0; E = 1; #100;
A = 1;B=1; C = 1; D = 1; E = 1; #100;
end
endmodule
```

postlab007.v | postlab_tb_0... | switch_not.v | switch_na...

SIMULATION OUTPUT

Q6.i- Compare the area, delay and power report of ripple carry & carry look-ahead adder using Suitable synthesizer.

Q6.ii-Design 4-bit Carry Save Adder using Verilog and verify the functional verification.

DESCRIPTION

The Verilog module implements a 4-bit Carry Save Adder (CSA) which computes the sum of two 4-bit operands, A and B. The module outputs two 4-bit partial sums (P and G) and a 1-bit carry-out. The CSA operates in three stages: generate (G), propagate (P), and carry (C). In the generate stage, G_i represents where both bits generate a carry, calculated as the bitwise AND of corresponding bits in A and B. In the propagate stage, P_i represents where at least one bit generates a carry, calculated as the bitwise XOR of corresponding bits in A and B. The carry-out is computed as the OR of all generate bits shifted by one position. This Verilog module enables functional verification to ensure accurate computation of the sum with proper handling of carries and propagation.

EXPERIMENT 6 POST LAB QUESTION 2 CARRY SAVE ADDER

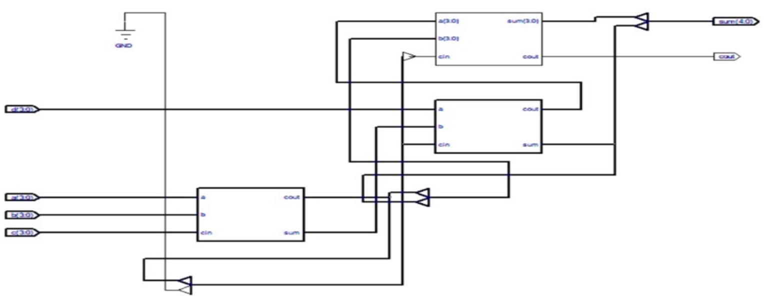
6.5 VERILOG CODE

```verilog
`timescale 1ns / 1ps
module CSA_07(a,b,c,d, sum, cout);
input [3:0] a, b,c,d;
output [4:0] sum;
output cout;
wire [3:0] s0,s1;wire [3:0] c0, c1;
fa_07 fa0(.a(a[0]), .b (b[0]), .cin (c[0]), .sum(s0[0]), .cout(c0[0]));
fa_07 fa1(.a(a[1]), .b (b[1]), .cin (c[1]), .sum(s0[1]), .cout(c0[1]));
fa_07 fa2(.a(a[2]), .b (b[2]), .cin (c[2]), .sum(s0[2]), .cout(c0[2]));
fa_07 fa3(.a(a[3]), .b (b[3]), .cin (c[3]), .sum(s0[3]), .cout(c0[3]));
fa_07 fa4(.a(a[0]), .b (s0[0]), .cin (1'b0), .sum(sum[0]), .cout(c1[0]));
fa_07 fa5(.a(a[1]), .b (s0[1]), .cin (c0[0]), .sum(s1[0]), .cout(c1[1]));
fa_07 fa6(.a(a[2]), .b (s0[2]), .cin (c0[1]), .sum(s1[1]), .cout(c1[2]));
fa_07 fa7(.a(a[3]), .b (s0[3]), .cin (c0[2]), .sum(s1[2]), .cout(c1[3]));
rca_07 rca1 (.a(c1[3:0]),.b({c0[3],s1[2:0]}), .cin(1'b0),.sum(sum[4:1]), .cout(cout));
endmodule
```

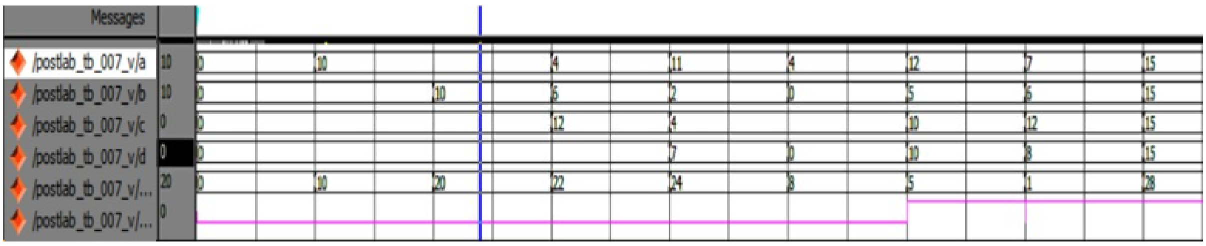
6.6 TESTBENCH CODE

```verilog
`timescale 1ns / 1ps
module csa_tb_07_v;
    reg [3:0] a;
    reg [3:0] b;
    reg [3:0] c;
    reg [3:0] d;
    wire [4:0] sum;
    wire cout;
    CSA_07 uut (
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .sum(sum),
        .cout(cout)
    );

    initial begin
        a = 0;
        b = 0;
        c = 0;
        d = 0;
    #100 a= 4'd10; b=4'd0; c=4'd0; d=4'd0;
    #100 a= 4'd10; b=4'd10; c=4'd0; d=4'd0;
    #100 a= 4'd4; b=4'd6; c=4'd12; d=4'd0;
    #100 a= 4'd11; b=4'd2; c=4'd4; d=4'd7;
    #100 a= 4'd20; b=4'd0; c=4'd20; d=4'd0;
    #100 a= 4'd12; b=4'd5; c=4'd10; d=4'd10;
    #100 a= 4'd7; b=4'd6; c=4'd12; d=4'd8;
    #100 a= 4'd15; b=4'd15; c=4'd15; d=4'd15;
    end
endmodule
```

6.7 RTL VIEW



6.8 SIMULATION OUTPUT



6.11 **TIMING REPORT CLA**

```
=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
No clock signals found in this design

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 14.468ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 33 / 5
-------------------------------------------------------------------------
Delay:               14.468ns (Levels of Logic = 6)
  Source:            Cin (PAD)
  Destination:       Cout (PAD)

```

6.12 **FINAL REPORT CLA**

```
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block cla_07, actual ratio is 0.

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : cla_07.ngr
Top Level Output File Name         : cla_07
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 14

Macro Statistics :
# Xors                             : 1
#      4-bit xor3                  : 1

Cell Usage :
# BELS                             : 9
#      LUT2                        : 1
#      LUT3                        : 7
#      LUT4                        : 1
# IO Buffers                       : 14
#      IBUF                        : 9
#      OBUF                        : 5
=========================================================================

Device utilization summary:
```

EXPERIMENT 6 POST LAB QUESTION 1

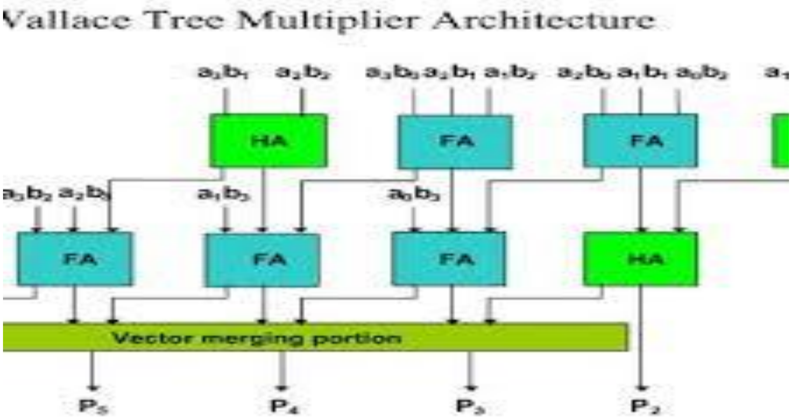
6.9 TIMING REPORT RCA

```
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
No clock signals found in this design

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 14.468ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 33 / 5
-------------------------------------------------------------------------
Delay:               14.468ns (Levels of Logic = 6)
  Source:            cin (PAD)
  Destination:       carry (PAD)

  Data Path: cin to carry
```

6.10 FINAL REPORT RCA

```
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block rca_07, actual ratio is 0.

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : rca_07.ngr
Top Level Output File Name         : rca_07
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 14

Cell Usage :
# BELS                             : 9
#      LUT2                        : 1
#      LUT3                        : 7
#      LUT4                        : 1
# IO Buffers                       : 14
#      IBUF                        : 9
#      OBUF                        : 5
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 3s100evq100-4

```

Q7.i -Draw the architecture of 4-bit Wallace Tree Multiplier.



Q7.ii-Design a 4-bit Wallace tree multiplier and verify the design using a suitable simulation tool.

DESCRIPTION

The Verilog module implements a 4-bit Wallace Tree Multiplier, which efficiently computes the product of two 4-bit operands, A and B. The module outputs an 8-bit product P. The Wallace Tree Multiplier architecture consists of three main stages: partial product generation, reduction tree, and final addition. In the partial product generation stage, each bit of the multiplier (B) is multiplied by each bit of the multiplicand (A), resulting in a matrix of partial products. The reduction tree then reduces the number of partial products using a tree structure to minimize the number of additions required. Finally, the reduced partial products are added together to obtain the final product. Functional verification of the Wallace Tree Multiplier can be performed using simulation tools such as ModelSim or Vivado Simulator. Testbench stimuli are generated to cover various input combinations and edge cases, and simulation waveforms are analyzed to ensure that the output product matches the expected result for each set of input operands. This verification process validates the correct operation of the 4-bit Wallace Tree Multiplier design.

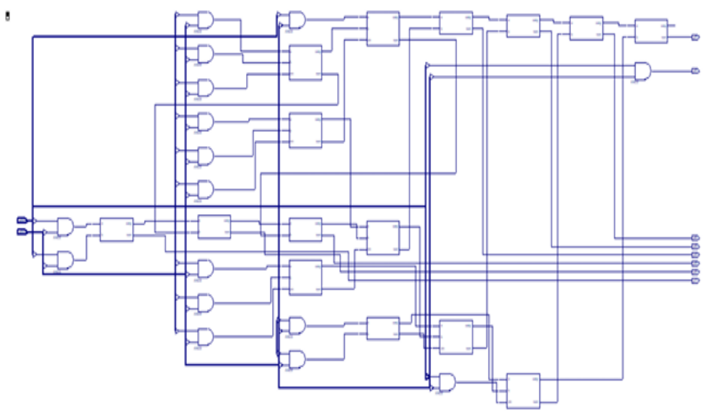
EXPERIMENT 7 POST LAB QUSTION

VERILOG CODE

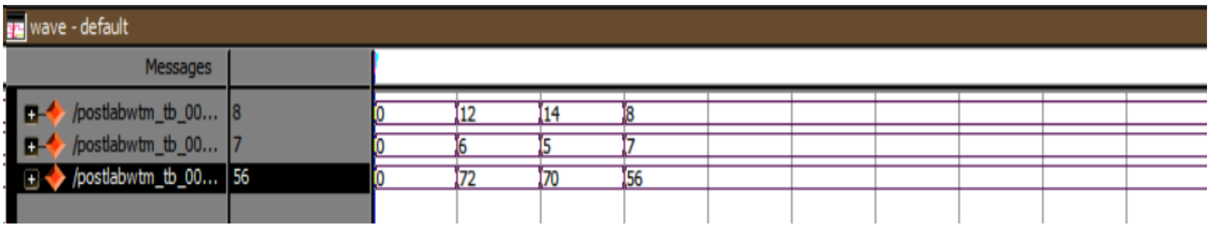
```
`timescale 1ns / 1ps
module wtm_007(p, a,b);
output [7:0] p;
input [3:0] a,b;
wire s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s33,s34,s35,s36,s37;
wire c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c33,c34,c35,c36,c37;
wire [6:0] p0,p1,p2,p3;
assign p0 = a & {4{b[0]}};
assign p1 = a & {4{b[1]}};
assign p2 = a & {4{b[2]}};
assign p3 = a & {4{b[3]}};
assign p[0] = p0[0];
assign p[1] = s11;
assign p[2] = s22;
assign p[3] = s32;
assign p[4] = s34;
assign p[5] = s35;
assign p[6] = s36;
assign p[7] = s37;
ha ha11(s11,c11,p0[1],p1[0]);
fa fa12(s12,c12,p0[2],p1[1],p2[0]);
fa fa13(s13,c13,p0[3],p1[2],p2[1]);
fa fa14(s14,c14,p1[3],p2[2],p3[1]);
ha ha15(s15,c15,p2[3],p3[2]);
ha ha22(s22,c22,c11,s12);
fa fa23(s23,c23,p3[0],c12,s13);
fa fa24(s24,c24,c13,c32,s14);
fa fa25(s25,c25,c14,c24,s15);
fa fa26(s26,c26,c15,c25,p3[3]);
ha ha32(s32,c32,c22,s23);
ha ha34(s34,c34,c23,s24);
ha ha35(s35,c35,c34,s25);
ha ha36(s36,c36,c35,s26);
ha ha37(s37,c37,c36,c26);
endmodule
```

TESTBENCH CODE & RTL VIEW

```
`timescale 1ns / 1ps

module postlabwtm_tb_007_v;

	// Inputs
	reg [3:0] a;
	reg [3:0] b;

	// Outputs
	wire [7:0] p;

	// Instantiate the Unit Under Test (UUT)
	wtm_007 uut (
		.p(p),
		.a(a),
		.b(b)
	);

	initial begin
		// Initialize Inputs
a = 0;
b = 0;
// Wait 100 ns for global reset to finish
#100;a=4'b1100;b=4'b0110;
#100;a=4'b1110;b=4'b0101;
#100;a=4'b1000;b=4'b0111;
// Add stimulus here
end
endmodule
```



SIMULATION OUTPUT



Q8- Design of Mealy FSM for sequence detection of the pattern “1101” using Verilog HDL.

DESCRIPTION

The Verilog module implements a Mealy Finite State Machine (FSM) designed to detect the sequence "1101" within an incoming bit stream. The FSM comprises four states: S0, S1, S2, and S3, representing the progress of the input sequence. The module takes a single-bit input representing the incoming bit stream and produces a single-bit output indicating whether the pattern "1101" has been detected. The state transition and output logic are defined based on the current state and input. Transition to the next state occurs based on the current state and the incoming bit. The output is set to high when the pattern "1101" is detected, specifically during the transition from state S2 to state S3. This Mealy FSM design effectively detects the specified sequence within the input bit stream.

POST LAB QUESTION EXPERIMENT 8

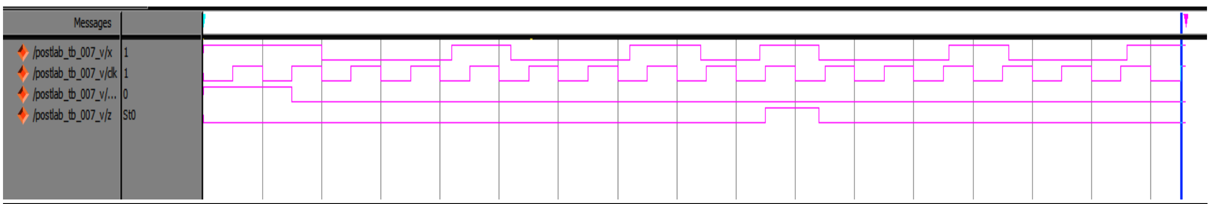
VERILOG CODE

```
`timescale 1ns / 1ps
module postlab007(
    input x, // Input signal
    input clk, // Clock signal
    input reset, // Reset signal
    output reg z // Output signal
);
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;
    reg [1:0] PS, NS;
    always @(posedge clk or posedge reset) begin
        if (reset)
            PS <= S0;
        else
            PS <= NS;
    end
    always @(PS or x) begin
        case (PS)
            S0: begin
                z = 0;
                NS = (x) ? S1 : S0;
            end
            S1: begin
                z = 0;
                NS = (x) ? S1 : S2;
            end
            S2: begin
                z = 0;
                NS = (x) ? S3 : S0;
            end
            S3: begin
                z = (x) ? 1 : 0;
                NS = (x) ? S1 : S2;
            end
        endcase end endmodule
```

TESTBENCH CODE

```
`timescale 1ns / 1ps
module postlab_tb_007_v;

    // Inputs
    reg x;
    reg clk;
    reg reset;

    // Outputs
    wire z;

    // Instantiate the Unit Under Test (UUT)
    postlab uut (
        .x(x),
        .clk(clk),
        .reset(reset),
        .z(z)
    );

    initial begin
        clk = 1'b0;
          reset = 1'b1;
          #15 reset = 1'b0;
     end

     always #5 clk = ~clk;

     initial begin
          // Test input sequence: 1101
          x = 1; #10 x = 1; #10 x = 0; #10 x = 0;
          #12 x = 1; #10 x = 0; #10 x = 0; #10 x = 1;
          #12 x = 0; #10 x = 1; #10 x = 0; #10 x = 0;
          #12 x = 1; #10 x = 0; #10 x = 0; #10 x = 1;
          #10 $finish;
     end
endmodule
```

SIMULATION CODE



Q9- 64-bit x 8-bit single-port RAM design with common read and write addresses in Verilog HDL

DESCRIPTION

The Verilog module implements a 64-bit x 8-bit single-port RAM with common read and write addresses. The module features several inputs and outputs for its operation. Inputs include a clock signal (clk) for synchronous operation, an asynchronous reset signal (reset) to reset the RAM, a write enable signal (we) to enable write operations, an address signal (addr) for both read and write operations, and a data input signal (din) for write operations. Outputs include a data output signal (dout) for read operations. The RAM consists of 64 memory locations, each capable of storing an 8-bit data value. It operates as a single-port RAM, allowing only one operation (read or write) at a time. Both read and write operations utilize the same address, simplifying the control logic. The module effectively manages read and write operations with a common address, ensuring efficient data access within the RAM.

<u>EX 9 POST LAB QUESTION</u>

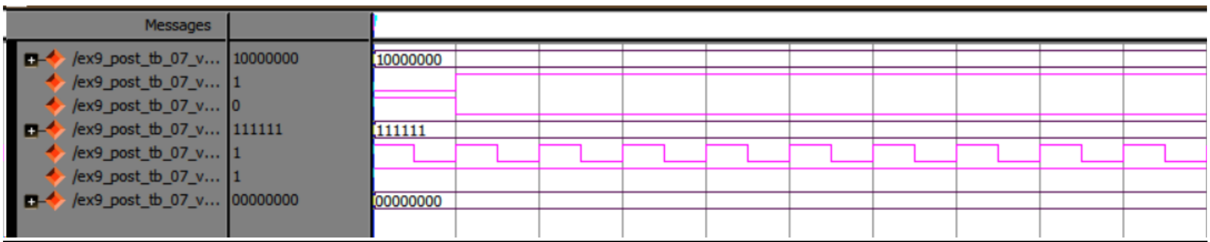
<u>VERILOG CODE</u>

```
`timescale 1ns / 1ps
module ex9_post_07(Output,Data,RD,WR,Address,clk,rst);
 output reg [7:0] Output;
   input [7:0]  Data;
   input [5:0] Address;
   input  RD,WR,clk,rst;
   reg [7:0]  memory[63:0];
always@(posedge clk)
begin
if(rst)
Output=8'b00000000;
else if(WR)
memory[Address]=Data;
else if(RD)
Output=memory[Address];
end

endmodule
```

<u>TESTBENCH CODE</u>

```
`timescale 1ns / 1ps
module ex9_post_tb_07_v;
   // Inputs
   reg [7:0] Data;
   reg RD;
   reg WR;
   reg [5:0] Address;
   reg clk;
   reg rst;

   // Outputs
   wire [7:0] Output;

   // Instantiate the Unit Under Test (UUT)
   ex9_post_07 uut (
      .Output(Output),
      .Data(Data),
      .RD(RD),
      .WR(WR),
      .Address(Address),
      .clk(clk),
      .rst(rst)
   );

   initial begin
            Data = 8'b10000000;RD = 0;WR = 1;Address = 6'b111111;clk = 1;rst = 1;#100;
      RD = 1;WR = 0;Address = 6'b111111;#100;
end   always #50 clk=~clk;
endmodule
```

<u>SIMULATION OUTPUT</u>



Q10- 1. Design Complex CMOS logic Out= ~(AB+CD).

2. Design Pseudo NMOS NAND gate.

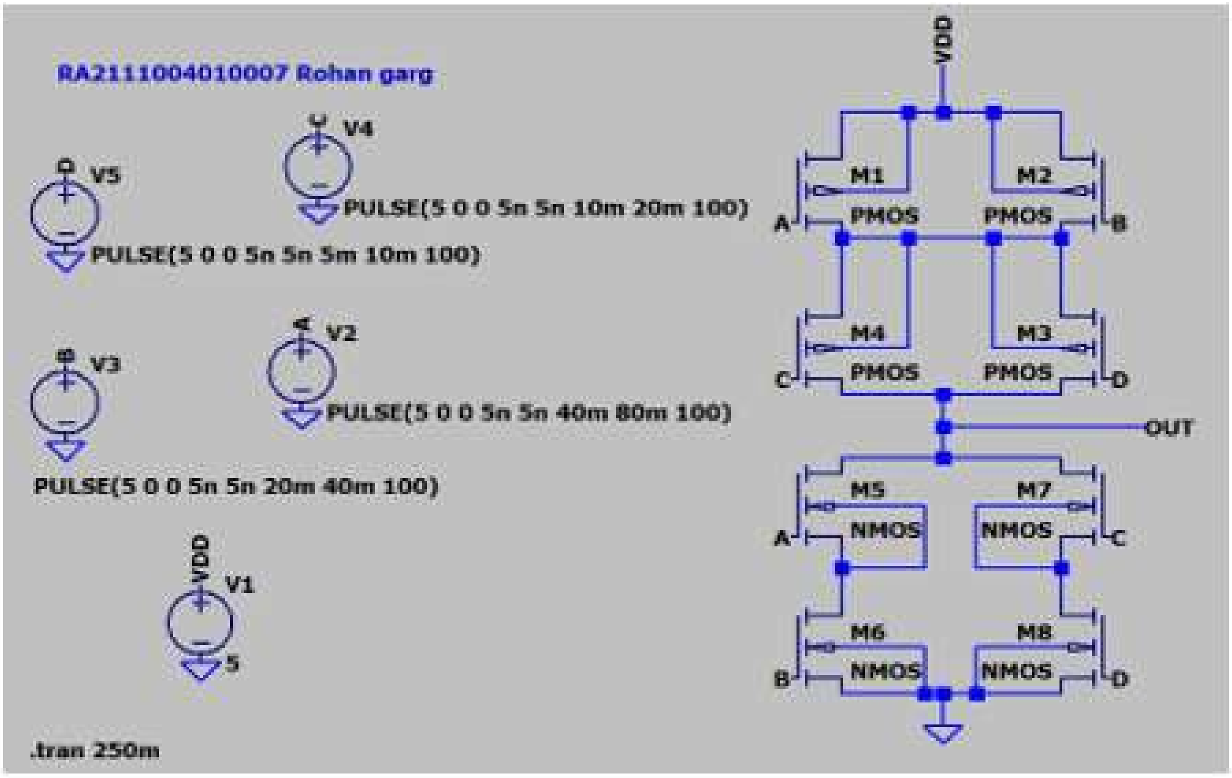
3. Perform DC Analysis for CMOS Inverter.

DESCRIPTION

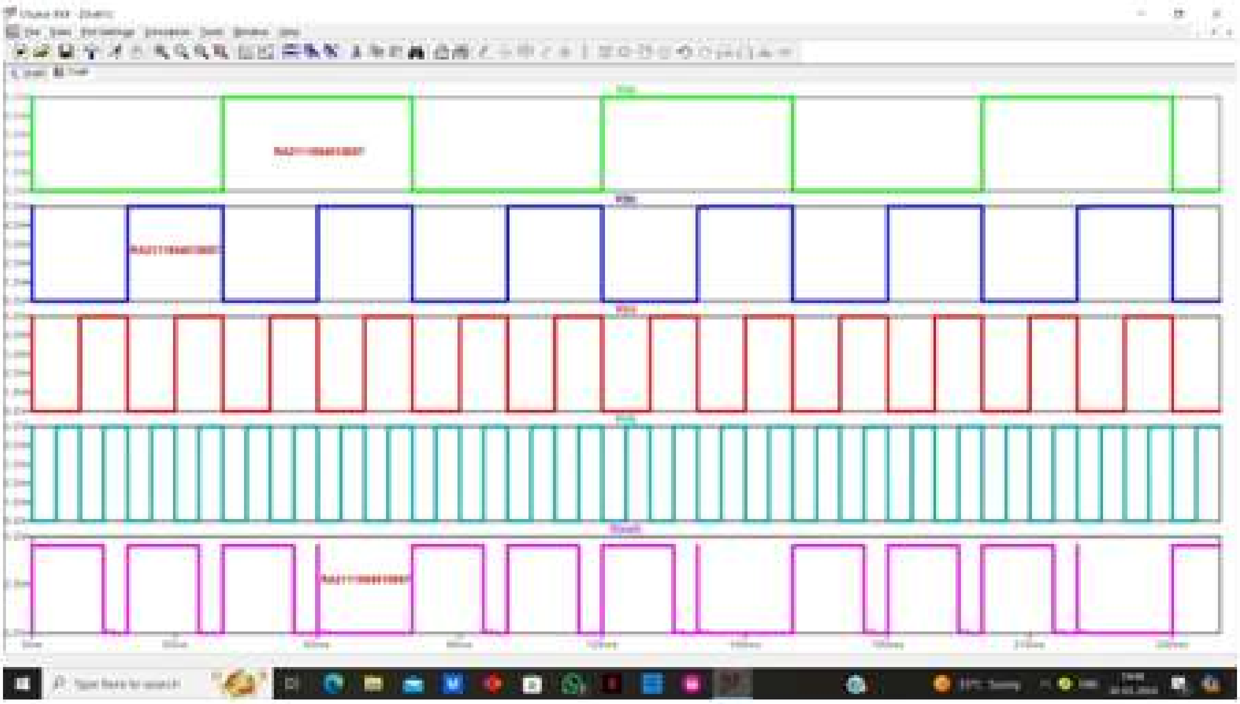
1. **Design Complex CMOS logic Out= ~(AB+CD)**:
    - Utilizing LTspice, design the CMOS circuit to implement $^{-}(AB+CD)$.
    - Employ CMOS NAND and NOR gates to achieve the desired logic function.
    - Verify the functionality of the circuit by simulating various input combinations and observing the output.
2. **Design Pseudo NMOS NAND gate**:
    - Create the Pseudo NMOS NAND gate using LTspice.
    - Construct the gate using NMOS transistors to perform the NAND operation.
    - Ensure proper biasing and connectivity of transistors for the desired logic functionality.
    - Validate the gate's operation through simulation by applying different input combinations and analyzing the output.
3. **Perform DC Analysis for CMOS Inverter**:
    - Set up the CMOS inverter circuit in LTspice.
    - Conduct DC analysis to examine the inverter's behavior across varying input voltage levels.
    - Analyze key parameters such as the voltage transfer curve, input/output voltage levels, and current consumption to understand the inverter's performance characteristics.
    - By performing DC analysis in LTspice, gain insights into the inverter's operation and optimize its design for desired performance metrics.

POST LAB QUESTIONS

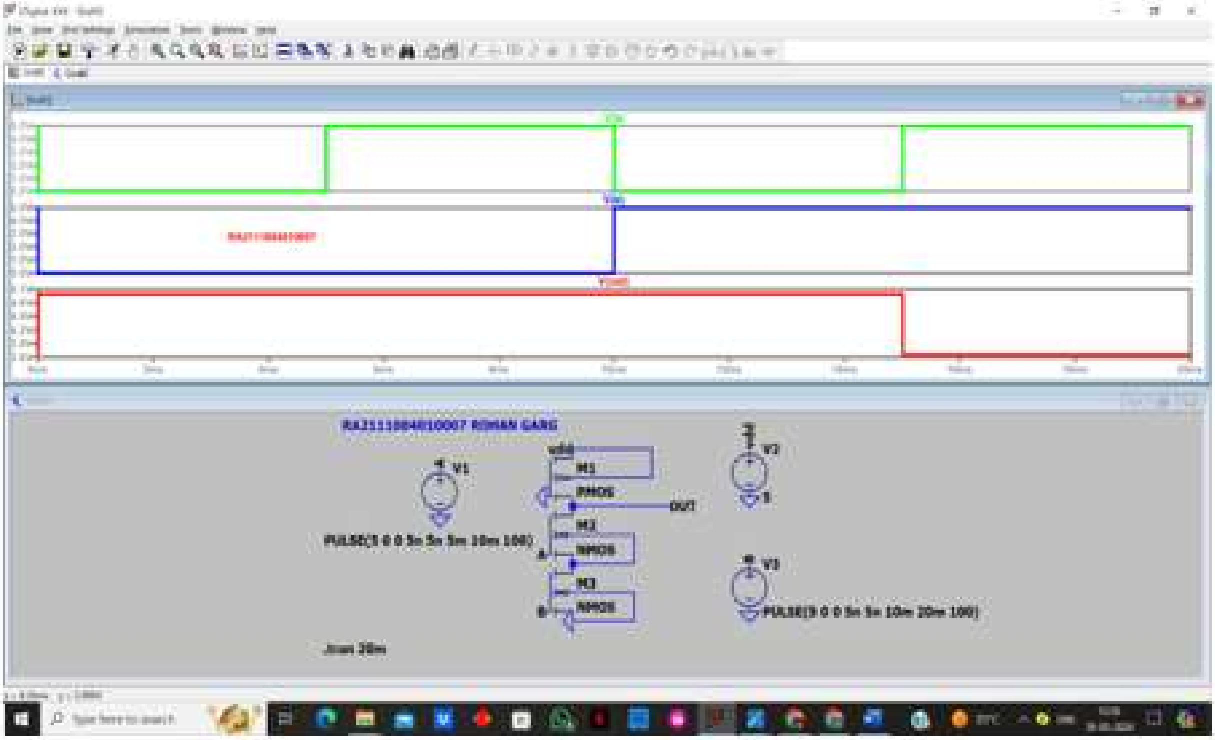
Q1 CMOS logic for OUT=~(AB+CD)



Output of complex CMOS logic for OUT=~(AB+CD)



Q2. Pseudo NMOS NAND Gate



Q3. DC ANALYSIS FOR CMOS INVERTER