VERILOG VHDL

SOFTWARE EXPERIMENTS

SOFTWARE USED: MODELSIM INTEL-FGPA



Akash Siva Teja.

22BCE1778

Submitted To: Dr.Sourabh Paul

EXPERIMENT-1 :Simple combinational circuit in all three

# models.

Aim: To write a Verilog code on ModelSim and verify the output logic for simple combinational circuit Y = AB+BC+AC.

# Software required: ModelSim software Verilog code:

1. Gate level modelling:

module simple\_bool(Y,A,B,C); input A,B,C;

output Y;

wire W1,W2,W3; and G1(W1,A,B);

and G2(W2,B,C);

and G3(W3,A,C);

or G4(Y,W1,W2,W3);

endmodule

# Data flow modelling:

module simple\_bool(Y,A,B,C); input A,B,C;

output Y;

assign Y=A&B|B&C|A&C;

endmodule

# Behavioral modelling:

module simple\_bool(Y,A,B,C); input A,B,C;

output Y;

reg AB, BC, AC;

always @(\*) begin AB = A & B;

BC = B & C;

AC = A & C;

Y = AB | BC | AC;

end endmodule

# OUTPUT:



Result: Thus simple Boolean expression is realized in Verilog HDL and its truth table is verified by simulation.

# EXPERIMENT 2:All adders and subtractors in all three models.

Aim: Write a Verilog code on Modelsim and verify the output logic by stimulation for Half Adder ,Full Adder, Half Subtractor, Full Subtractor.

# Software required: Modelsim software Verilog code:

* 1. Half Adder:

# Gate level modelling:

module H\_adder(S,C,A,B); output S,C;

input A,B;

xor a\_1(S,A,B);

and a\_2(C,A,B); endmodule

# Dataflow modelling:

module H\_adder(S,C,A,B); output S,C;

input A,B; assign S=A^B; assign C=A&B; endmodule

# Behavioral modelling:

module H\_adder(S,C,A,B); output S,C;

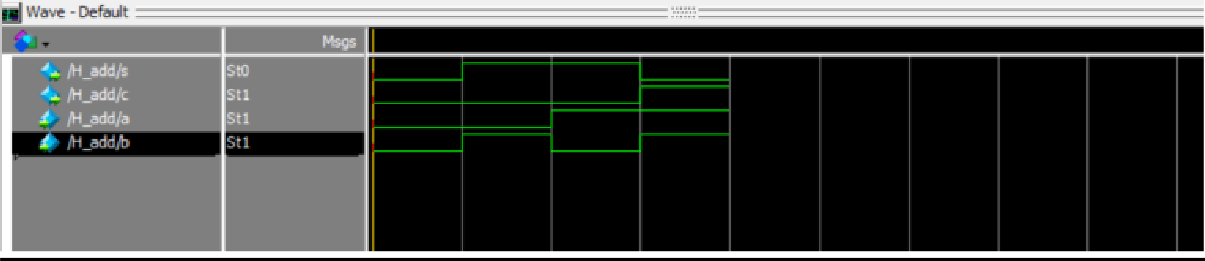
input A,B;

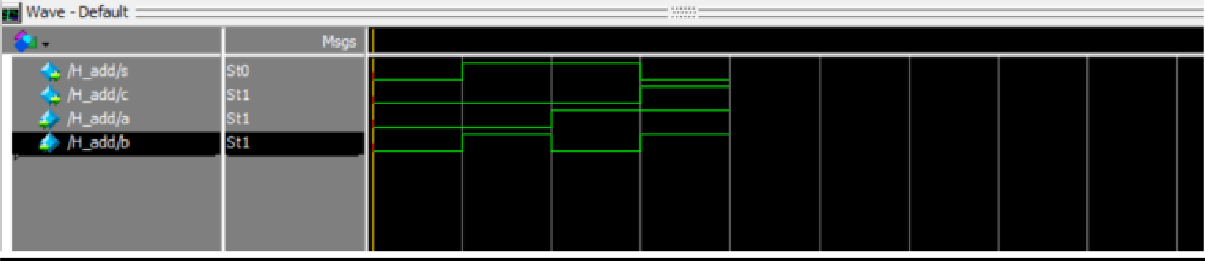
always@(A or B) begin case({A,B})

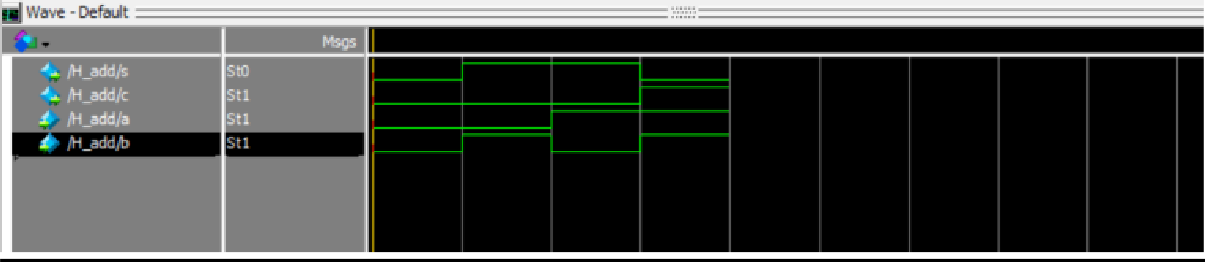
2’b00:begin S=0;C=0;end 2’b01:begin S=1;C=0;end 2’b10:begin S=1;C=0;end 2;b11:begin S=1;C=1;end endcase

end endmodule

# OUTPUT:







* 1. Full Adder:

# Gate level modelling:

module full\_adder(S, Cout, A, B, Cin); output S;

output Cout; input A; input B; input Cin; wire w1; wire w2; wire w3; wire w4; xor(w1, A, B);

xor(S, Cin, w1);

and(w2, A, B);

and(w3, A, Cin);

and(w4, B, Cin);

or(Cout, w2, w3, w4); endmodule

# Dataflow modelling:

module full\_adder(S, Cout, A, B, Cin); output S;

output Cout; assign S=A^B^Cin;

assign C=(Cin&(A^B))|(A&B); endmodule

# Behavioral modelling:

module full\_adder(S, Cout, A, B, Cin); output S;

output Cout; reg S,C;

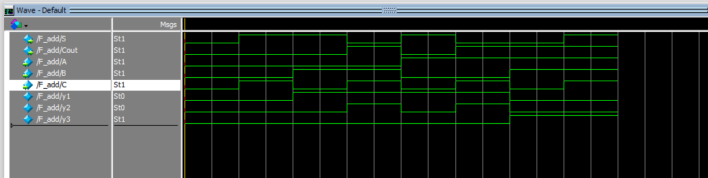
always@(A or B or Cin) begin

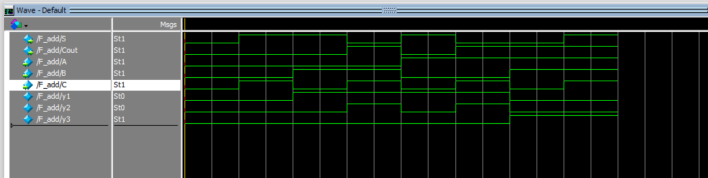
case({A,B,Cin})

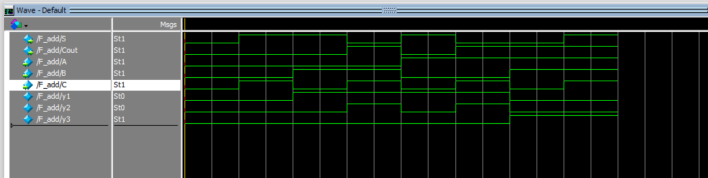
3’b000:bagin S=0;C=0;end 3’b001:bagin S=1;C=0;end 3’b010:bagin S=1;C=0;end 3’b011:bagin S=0;C=1;end 3’b100:bagin S=1;C=0;end 3’b101:bagin S=0;C=1;end 3’b110:bagin S=0;C=1;end 3’b111:bagin S=1;C=1;end endcase

end endmodule

# OUTPUT:







* 1. Half subtractor:

# Gate level modelling:

module sub\_sm(D,Bout,A,B); input A,B;

output D,Bout; wire y;

xor a\_1(D,A,B);

not a\_2(y,A);

and a\_3(Bout,y,B); endmodule

# Data flow modelling:

module sub\_sm(D,Bout,A,B); input A,B;

output D,Bout;

assign D=A^B;

assign Bout((~A)&B); endmodule

# Behavioral modelling:

module sub\_sm(D,Bout,A,B); input A,B;

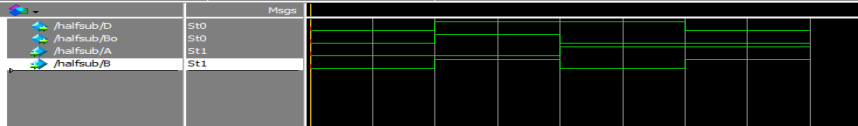
output D,Bout; reg D,Bout; always@(A or B) begin

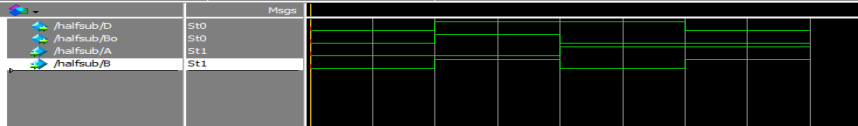
case ({A,B})

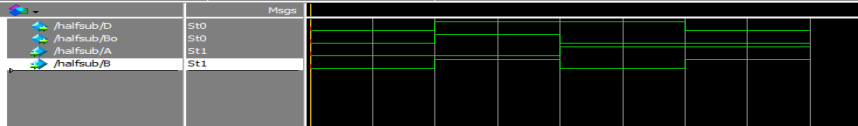
2’b00:begin D=0;Bout=0;end 2’b01:begin D=1;Bout=1;end 2’b10:begin D=1;Bout=0;end 2’b11:begin D=0;Bout=0;end endcase

end endmodule

# OUTPUT:







* 1. Full subtractor:

# Gate level modelling:

module sub\_sm(D,Bout,A,B,C); input A,B,C;

output D,Bout; wire y1,y2,y3,y4,y5; xor a\_1(D,A,B,C); not a\_2(y1,A);

and a\_3(y2,y1,B);

xor a\_4(y3,A,B); not a\_5(y4,y3); and a\_6(y5,y4,C); or a\_7(Bout,y2,y5); endmodule

# Data flow modelling:

module sub\_sm(D,Bout,A,B,C); input A,B,C;

output D,Bout; assign D=A^B^C;

assign Bout=(~A&B)|(C&(~(A^B)));

endmodule

# Behavioral modelling:

module sub\_sm(D,Bout,A,B,C); input A,B,C;

output D,Bout; reg D,Bout;

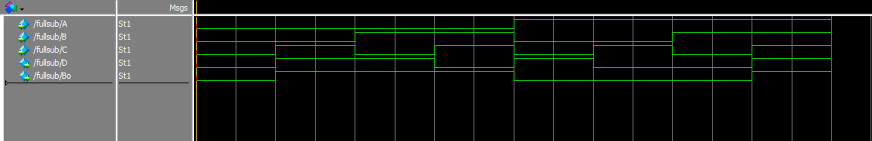
always@(A or B or C) begin

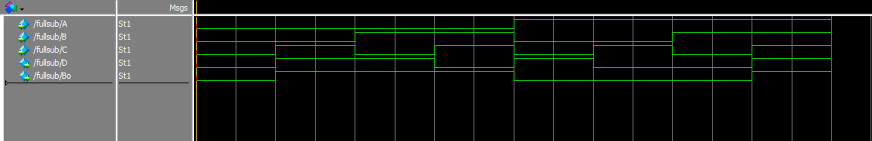
case({A,B,C})

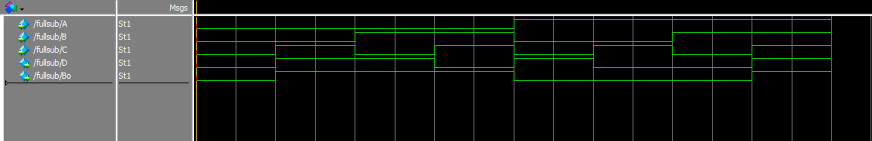
3’b000:begin D=0;Bout=0;end 3’b001:begin D=1;Bout=1;end 3’b010:begin D=1;Bout=1;end 3’b011:begin D=0;Bout=1;end 3’b100:begin D=1;Bout=0;end 3’b101:begin D=0;Bout=0;end 3’b110:begin D=0;Bout=0;end 3’b111:begin D=1;Bout=1;end endcase

end endmodule

# OUTPUT:







RESULT: Thus the Half Adder,Full Adder,Half Subtractor,Full Subtractor code is stimulated and the truth table values for sum and carry are verified.

# EXPERIMENT 3:Multiplexer in different models

Aim: To construct the multiplexer and demultiplexer in model sim.

# Software required: Modelsim software Verilog code:

* + - 1. Multiplexer

# Gate level modelling:

module mux\_ SM (a,b,c,d,select0,select1,Y) ; input a,b,c,d,select0,select1;

output y:

wire y 1,y2,y3,y4,y5,y6; not a \_1(y1, select0) ; not a\_ 2(y2, select1) ; and a\_ 3( y3,a,y1 ,y2) ;

and a \_4(y4,b,y1, select1) ; and a \_5(y5,c, select0,y2) ;

and a \_ 6(y6 ,d, select0, select1) ; or a \_7( y ,y3,y4,y5,y6), endmodule

# Data flow modelling:

module mux (a,b,c,d,select0,select1,y) ; output y;

input a,b,c,d,select0,select1;

assign y =(a& (~select0) & (~select1) | (D1& (~select0) & select1) | (D2& select0& (~select1)) | (D3& select0& select1));

endmodule

# Behavioral modelling:

module mux\_sm (a,b,c,d, select0,select1,Y); input a,b,c,d, select1,select1;

output Y; reg Y;

always@ (a or b or c or d or select0 or select1) begin

case ( { select0,select1 } )

2 ‘ b00 : Y <=a ;

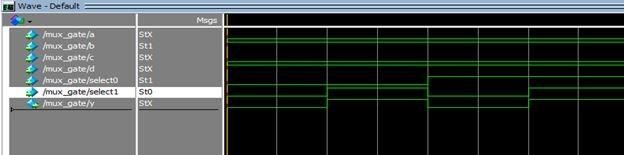
2 ‘ b01: Y <=b ;

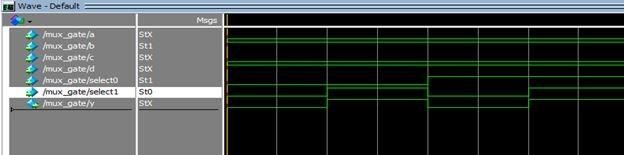
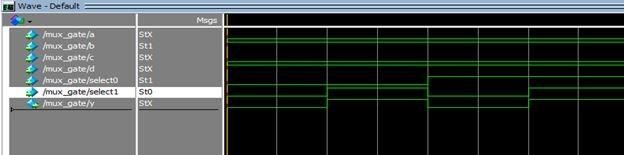
2 ‘ b10 : Y <=c ;

2 ‘ b11 : Y <=d ;

endcase end endmodule

# OUTPUT:





* + - 1. DEMULTIPLEXER:

# Gate level modelling:

module demux\_sm(D,S1,S0,Y0,Y1,Y2,Y3); input D,S1,S0;

output Y0,Y1,Y2,Y3; wire S0bar,S1bar; not a\_1(S0bar,S0); not a\_2(S1bar,S1);

and a\_3(Y0,S0bar,S1bar); and a\_4(Y1,S1bar,S0); and a\_5(Y2,S1,S0bar); and a\_6(Y3,S1,S0);

endmodule

# Data flow modelling:

module demux\_sm(D,S1,S0,Y0,Y1,Y2,Y3); input D,S1,S0;

output Y0,Y1,Y2,Y3;

assign Y0=(D&(~S1)&(~S0)); assign Y1=(D&(~S1)&(S0)); assign Y2=(D&(S1)&(~S0)); assign Y3=(D&(S1)&(S0)); endmodule

# Behavioral modelling:

module demux\_sm(D,S1,S0,Y0,Y1,Y2,Y3); input D,S1,S0;

output Y0,Y1,Y2,Y3; reg Y0,Y1,Y2,Y3;

always@(D or S1 or S0); begin

case({S1,S0}) 2'b00:Y0<=D;

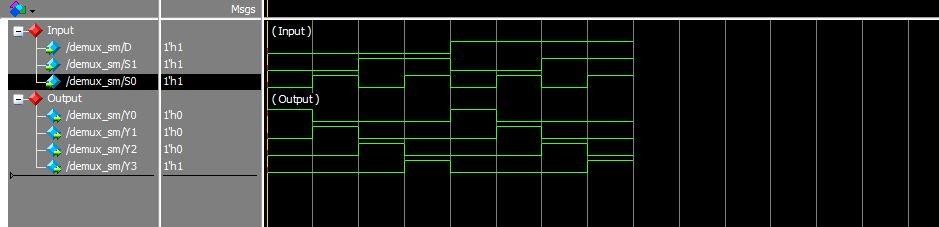
2'b01:Y1<=D;

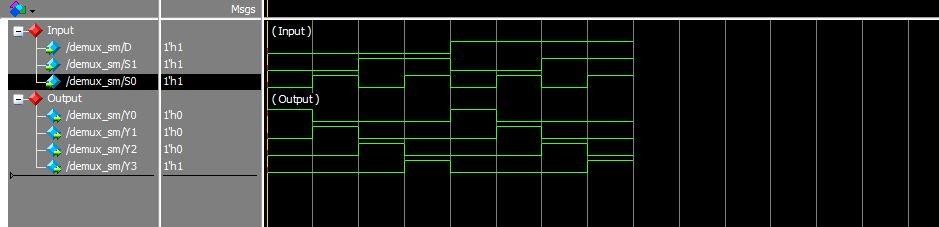
2'b10:Y2<=D;

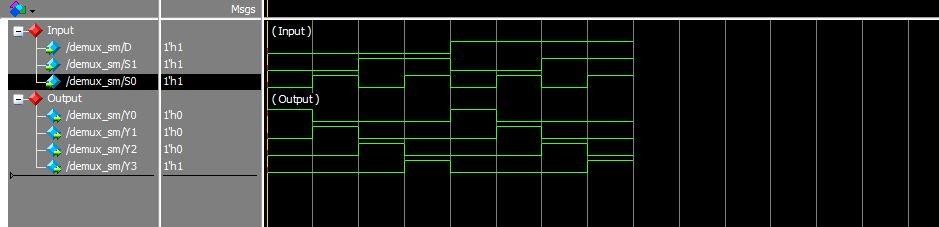
2'b11:Y3<=D;

endcase end endmodule

# OUTPUT:







RESULT: Thus the Multiplexer and Demultiplexer codes are stimulated and the truth table values are verified.

EXPERIMENT 4. 4Bit adder and adder/subtractor

AIM: Write a Verilog code on Model sim and verify the output logic by stimulation for 4bit adder and adder/subtractor.

# Software required: Model sim software Verilog code:

4Bit adder 1)Dataflow modelling:

module full\_adder\_bip (a, b, cin, sum, cout); input a, b, cin;

output sum, cout; xor inst1 (net1, a, b); and inst2 (net2, a, b);

xor inst3 (sum, net1, cin); and inst4 (net3, net1, cin); or inst5 (cout, net3, net2);

endmodule

module adder4\_ripple\_carry (a, b, cin, sum, cout); input [3:0] a, b; //define inputs and outputs

input cin;

output [3:0] sum; output cout; wire [3:0] c;

assign cout = c[3]; //continued on next page full\_adder\_bip inst0 (a[0], b[0], cin, sum[0], c[0]);

full\_adder\_bip inst1 (a[1], b[1], c[0], sum[1], c[1]);

full\_adder\_bip inst2 (a[2], b[2], c[1], sum[2], c[2]);

full\_adder\_bip inst3 (a[3], b[3], c[2], sum[3], c[3]); endmodule

# 2)Behavioral modelling:

module adder4 (a, b, cin, sum, cout); input [3:0] a, b;

input cin;

output [3:0] sum; output cout;

reg [3:0] sum; reg cout;

always @ (a or b or cin) begin

sum = a + b + cin; cout = (a[3] & b[3]) |

((a[3] | b[3]) & (a[2] & b[2])) |

((a[3] | b[3]) & (a[2] | b[2]) & (a[1] & b[1])) |

((a[3] | b[3]) & (a[2] | b[2]) & (a[1] | b[1]) & (a[0] & b[0])) |

((a[3] | b[3]) & (a[2] | b[2]) & (a[1] | b[1]) & (a[0] | b[0]) & cin);

end endmodule

# OUTPUT:





Adder\Subtactor:

module adder\_bar\_subtractor(S, C, V, A, B, m); output [3:0] S;

output C; output V; input [3:0] A;

input [3:0] B; input m;

wire C0;

|  |  |
| --- | --- |
| wire | C1; |
| wire | C2; |
| wire | C3; |
| wire | B0; |
| wire | B1; |
| wire | B2; |
| wire B3; |  |

xor(B0, B[0], m);

xor(B1, B[1], m);

xor(B2, B[2], m);

xor(B3, B[3], m);

xor(C, C3, m);

xor(V, C3, C2);

full\_adder fa0(S[0], C0, A[0], B0, m);

full\_adder fa1(S[1], C1, A[1], B1, C0);

full\_adder fa2(S[2], C2, A[2], B2, C1);

full\_adder fa3(S[3], C3, A[3], B3, C2); endmodule

module full\_adder(S, Cout, A, B, Cin); output S;

output Cout; input A; input B;

OUTPUT:



# RESULT: Thus the 4bit adder and adder/subtractor codes are stimulated and the truth table values are verified.

SR flipflop

Main program

module srff\_behave(s,r,clk, q, qbar);

input s,r,clk;

output reg q, qbar;

always@(posedge clk)

begin

if(s == 1 && r==0)

begin

q = 1;

qbar = 0;

end

else if(s == 0 && r == 1)

begin

q = 0;

qbar =1;

end

else if(s == 0 & r == 0)

begin

q <= q;

qbar <= qbar;

end

end

endmodule

JK Flipflop

module jkff\_behave(j,k,clk, q, qbar);

input j,k,clk;

output reg q, qbar;

always@(posedge clk)

begin

if(j == 1 && k==0)

begin

q = 1;

qbar = 0;

end

else if(j == 0 && k == 1)

begin

q = 0;

qbar =1;

end

else if(j == 0 & k == 0)

begin

q <= q;

qbar <= qbar;

end

else if(j == 1 && k == 1)

begin

q = ~q;

qbar =~ qbar;

end

end

endmodule

D flipflop

module srff\_behave(d, clk, q, qbar);

input d,clk;

output reg q, qbar;

always@(posedge clk)

begin

if(d == 1)

begin

q = 1;

qbar = 0;

end

else if(d == 0)

begin

q = 0;

qbar =1;

end

end

endmodule

T flipflop

module srff\_behave(T, clk, q, qbar);

input T,clk;

output reg q, qbar;

always@(posedge clk)

begin

if(T == 1)

begin

q = ~q;

qbar = ~qbar;

end

else if(T == 0)

begin

q = q;

qbar = qbar;

end

end

endmodule