Assignments-2

[1] Design 4-bit Ripple Carry Adder with the help of 1-bit adder.

Program :-

testbench.sv

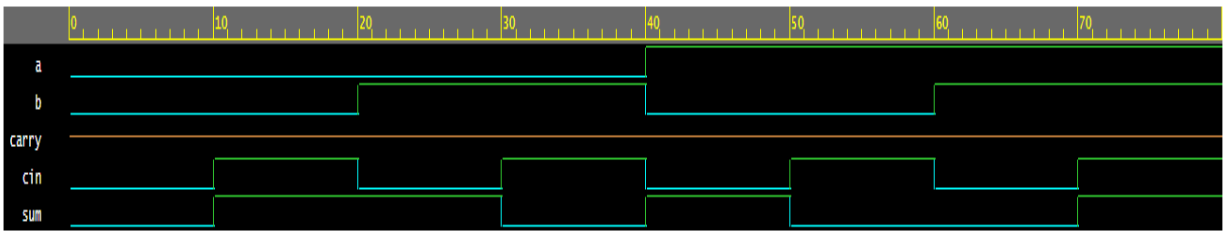
SV/Verilog Testbench

```
module rca_tb;
reg [3:0]a,b;
reg cin;
wire [3:0]sum;
wire c4;

rca uut(a,b,cin,sum,c4);

initial begin
  $dumpfile("dump.vcd");
  $dumpvars(1);
cin = 0;
a = 4'b0110;
b = 4'b1100;
#10
a = 4'b1110;
b = 4'b1000;
#10
a = 4'b0111;
b = 4'b1110;
#10
a = 4'b0010;
b = 4'b1001;
#10
$finish();
end

endmodule
```

design.sv

```
module full_adder(
    input a,b,cin,
    output sum,carry);

assign sum = a ^ b ^ cin;
assign carry = (a & b)|(b & cin)|(cin & a);

endmodule
module rca(
    input [3:0]a,b,
    input cin,
    output [3:0]sum,
    output c4);

wire c1,c2,c3;      //Carry out of each full adder

full_adder fa0(a[0],b[0],cin,sum[0],c1);
full_adder fa1(a[1],b[1],c1,sum[1],c2);
full_adder fa2(a[2],b[2],c2,sum[2],c3);
full_adder fa3(a[3],b[3],c3,sum[3],c4);

endmodule
```

Output :-



[2] Design D-flipflop and reuse it to implement 4- bit Johnson Counter.

Program :-

testbench.sv

SV/Verilog Testbench

```
// Code your testbench here
// or browse Examples
module jc_tb;
  reg clk,reset;
  wire [3:0] out;

  johnson_counter dut (.out(out), .reset(reset), .clk(clk));

  always
    #5 clk =~clk;

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

  initial
    begin
     $monitor( $time, " clk=%b, out= %b, reset=%b",
clk,out,reset);
     #105  $stop;
   end

endmodule
```

design.sv

```

module johnson_counter( out,reset,clk);
input clk,reset;
output [3:0] out;

reg [3:0] q;

always @(posedge clk)
begin

if(reset)
 q=4'd0;
 else
    begin
        q[3]<=q[2];
        q[2]<=q[1];
        q[1]<=q[0];
        q[0]<=(~q[3]);
    end
 end

assign out=q;
endmodule
```

Output :-

Log Share

```
[2023-08-24 05:32:48 UTC] iverilog '-Wall' design.sv testbench.sv  && unbuffer vvp a.out
                   0 clk=0, out= xxxx, reset=1
                   5 clk=1, out= 0000, reset=1
                  10 clk=0, out= 0000, reset=1
                  15 clk=1, out= 0000, reset=1
                  20 clk=0, out= 0000, reset=0
                  25 clk=1, out= 0001, reset=0
                  30 clk=0, out= 0001, reset=0
                  35 clk=1, out= 0011, reset=0
                  40 clk=0, out= 0011, reset=0
                  45 clk=1, out= 0111, reset=0
                  50 clk=0, out= 0111, reset=0
                  55 clk=1, out= 1111, reset=0
                  60 clk=0, out= 1111, reset=0
                  65 clk=1, out= 1110, reset=0
                  70 clk=0, out= 1110, reset=0
                  75 clk=1, out= 1100, reset=0
                  80 clk=0, out= 1100, reset=0
                  85 clk=1, out= 1000, reset=0
                  90 clk=0, out= 1000, reset=0
                  95 clk=1, out= 0000, reset=0
                 100 clk=0, out= 0000, reset=0
```

[3] Reuse 2:1 Mux code to implement 8:1 Mux.

Program :-

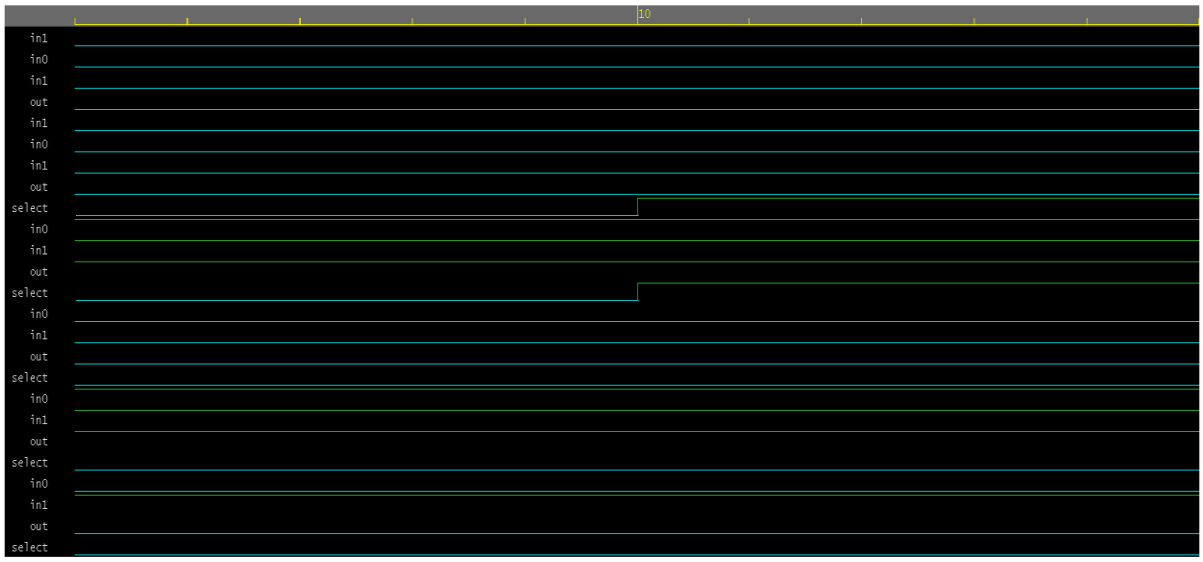
testbench.sv

```
// Code your testbench here
// or browse Examples
module tb_mux_8to1;
    reg [7:0] inputs;
    reg [2:0] select;
    wire out;
    mux_8to1 uut (
        .inputs(inputs),
        .select(select),
        .out(out)
    );
    reg clk = 0;
    always #5 clk = ~clk;
    initial begin
        $dumpfile("tb_mux_8to1.vcd");
        $dumpvars(0, tb_mux_8to1);
        inputs = 8'b11001100;
        select = 3'b000;
        #10;
        $display("Input: %b, Select: %b, Output: %b", inputs, select, out);
        select = 3'b001;
        #10;
        $display("Input: %b, Select: %b, Output: %b", inputs, select, out);
        $finish;
    end

endmodule
```

design.sv

```
module mux_2to1 (
    input wire in0,
    input wire in1,
    input wire select,
    output wire out
);
    assign out = select ? in1 : in0;
endmodule
module mux_8to1 (
    input wire [7:0] inputs,
    input wire [2:0] select,
    output wire out
);
    wire [1:0] s1;
    wire [1:0] s2;
    mux_2to1 mux0 (
        .in0(inputs[0]),
        .in1(inputs[1]),
        .select(select[0]),
        .out(s1[0])
    );
    mux_2to1 mux1 (
        .in0(inputs[2]),
        .in1(inputs[3]),
        .select(select[0]),
        .out(s1[1])
    );
    mux_2to1 mux2 (
        .in0(inputs[4]),
        .in1(inputs[5]),
        .select(select[1]),
        .out(s2[0])
    );
    mux_2to1 mux3 (
        .in0(inputs[6]),
        .in1(inputs[7]),
        .select(select[1]),
        .out(s2[1])
```

Output :-



[4] Design a Full Subtractor with Gate Level Modeling Style.(use primitive gates)

Program:-

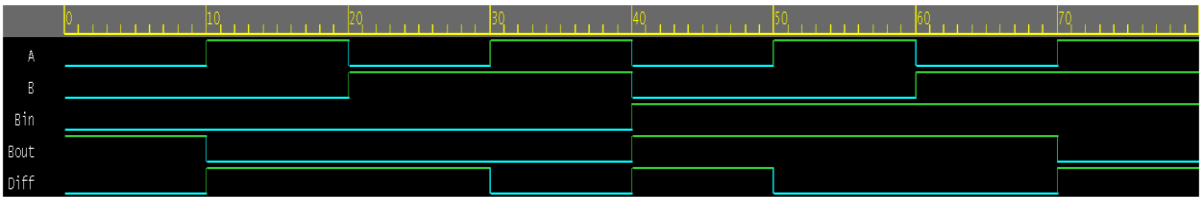
testbench.sv

```
module Testbench;
    reg A, B, Bin;
    wire Diff, Bout;
    FullSubtractor uut (
        .A(A),
        .B(B),
        .Bin(Bin),
        .Diff(Diff),
        .Bout(Bout)
    );
    initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
        $display("A\tB\tBin\tDiff\tBout");
        $monitor("%b\t%b\t%b\t%b\t%b", A, B, Bin, Diff, Bout);
        A = 0; B = 0; Bin = 0;
        #10;
        A = 1; B = 0; Bin = 0;
        #10;
        A = 0; B = 1; Bin = 0;
        #10;
        A = 1; B = 1; Bin = 0;
        #10;
        A = 0; B = 0; Bin = 1;
        #10;
        A = 1; B = 0; Bin = 1;
        #10;
        A = 0; B = 1; Bin = 1;
        #10;
        A = 1; B = 1; Bin = 1;
        #10;
        $finish;
    end
endmodule
```

design.sv

```
module FullSubtractor (
    input A,
    input B,
    input Bin,
    output Diff,
    output Bout
);

    wire X1, X2, X3, X4, X5;

    assign X1 = A ^ B;
    assign X2 = X1 ^ Bin;
    assign X3 = (A & B) | (X2 & ~Bin);
    assign Diff = X1 ^ Bin;
    assign Bout = ~X3;

endmodule
```

Output :-



[5] Design a 2X4 decoder using gate level modelling.

Program :-

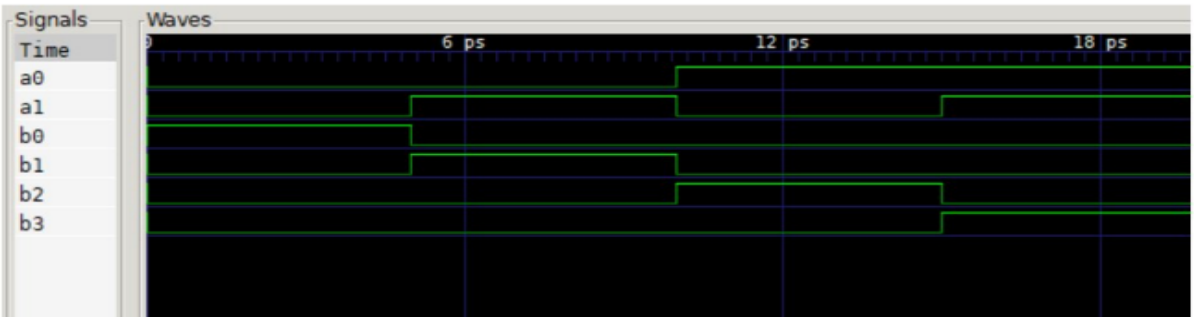
testbench.sv

```
module main;
reg a0,a1; wire b0,b1,b2,b3;
initial begin
  $dumpvars(1);
  $dumpfile("decoder.vcd");
  $monitor("time=%g | a0=%b| a1 = %b | b0=%b | b1=%b | b2 = %b
| b3=%b "Stime,a0,a1,b0,b1, b2,b3);
end
  decoder_2x4 uut(a0,a1,b0,b1,b2,b3);
initial
begin

a0=0; a1=0;
#5 a0=0; a1= 1;
#5 a0= 1; a1=0;
#5 a0= 1; a1= 1;
#5 a0=0; a1=0;
end
  endmodule
```

design.sv

```

module

  decoder_2x4(a0,a1,b0,b1,b2,b3);

output bo,b1,b2,b3;
input a0,a1;
wire w1,w2;

  not n1 (w1,a0);
  not n2 (w2,a2);
  and a1 (b0,w1,w2);
  and a2 (b1,w1,a1);
  and a3 (b2,a0,w2);
  and a4 (b3,a0,a1);

endmodule
```

Output :-



[6] Design a 4x1 mux using operators. (use data flow)

Program :-

testbench.sv

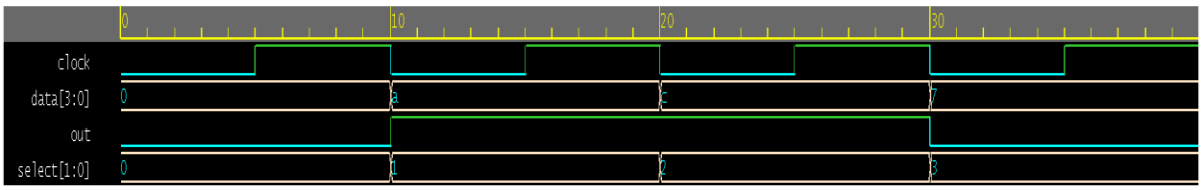
SV/Verilog Testbench

```
module tb_mux_4x1;
  reg [3:0] data;
  reg [1:0] select;
  wire out;
  mux_4x1 uut (
    .data(data),
    .select(select),
    .out(out)
  );
  reg clock = 0;
  always #5 clock = ~clock;
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    data = 4'b0000;
    select = 2'b00;
    #10;
    data = 4'b1010;
    select = 2'b01;
    #10;
    data = 4'b1100;
    select = 2'b10;
    #10;
    data = 4'b0111;
    select = 2'b11;
    #10;
    $finish;
  end
  always @(posedge clock) begin
    $display("Time = %0t: Data = %b, Select = %b, Output = %b", $time, data, select, out);
  end

endmodule
```

design.sv

```
module mux_4x1 (
    input wire [3:0] data,
    input wire [1:0] select,
    output wire out
);

assign out = (select == 2'b00) ? data[0] :
             (select == 2'b01) ? data[1] :
             (select == 2'b10) ? data[2] :
             data[3];

endmodule
```

Output :-



[7] Design a Full adder using half adder.

Program :-

testbench.sv

SV/Verilog Testbench

```
module full_adder_tb;
reg a,b,cin;
wire sum,carry;

full_adder uut(a,b,cin,sum,carry);

initial begin
  $dumpvars(1);
  $dumpfile("dump.vcd");
a = 0; b = 0; cin = 0;
#10
a = 0; b = 0; cin = 1;
#10
a = 0; b = 1; cin = 0;
#10
a = 0; b = 1; cin = 1;
#10
a = 1; b = 0; cin = 0;
#10
a = 1; b = 0; cin = 1;
#10
a = 1; b = 1; cin = 0;
#10
a = 1; b = 1; cin = 1;
#10
$finish();
end

endmodule
```

design.sv

```
module half_adder (
    input a,b,
    output sum,carry
);

assign sum = a ^ b;
assign carry = a & b;

endmodule
module full_adder(
    input a,b,cin,
    output sum,carry
);

wire c,c1,s;

half_adder ha0(a,b,s,c);
half_adder ha1(cin,s,sum,c1);
  or or1(c|c1);


endmodule
```

Output :-