DSD LAB FAT

1) MUX Design

Aim: To design a Verilog model and verify the truth table of the multiplexer using Gate-level modelling and Data flow modelling

Tools Required: ModelSim

Procedure:

1) Determine the required number of inputs and outputs and assign a symbol to each
2) Derive the truth table
3) Obtain the simplified Boolean function for each output as a function of the input variables
4) Draw the logic diagram and verify the design

Theory: A Multiplexer selects one of many inputs signals and forwards the selected input to a single output line. It acts as a data selector. The input line chosen is determined by a separate set of select lines.

Program:

Gate-level Modelling:

line 10: and g2(w5,a,w2,I2);

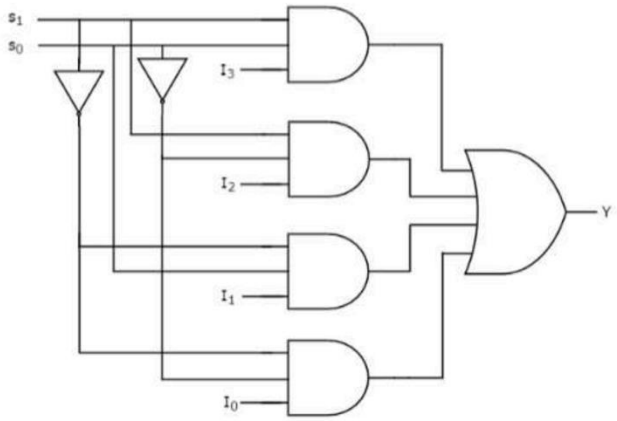
```
 2  module mux_41(a,b,I0,I1,I2,I3,F);
 3  input (a,b,I0,I1,I2,I3);
 4  output F;
 5  wire w1, w2, w3, w4, w5, w6;
 6  not (w1,a);
 7  not (w2,b);
 8  and g0(w3,w1,w2,I0);
 9  and g1(w4,w1,b,I1);
10  and g3(w6,a,b,I3);
11  or (F,w3,w4,w5,w6);
12  endmodule
```

Data-flow Modelling:

```
module mux1_4(a,b,i0,i1,i2,i3,f);
input a,b,i0,i1,i2,i3;
output f;
assign f=((~a)&(~b)&(i0)) | ((~a)&(b)&(i1)) | ((a)&(~b)&(i2)) | ((a)&(b)&(i3));
endmodule
```

Truth Table:

a	b	F
0	0	I0
0	1	I1
1	0	I2
1	1	I3



Conclusion: With the use of gate-level and data flow modelling, the Verilog model for multiplexer were created and simulated successfully. The simulation results matched the theoretical truth tables, indicating the design were correct.

2) Decoder

Theory: Decoder is a full subtractor and a combinational circuit that subtracts 2 single-bit binary numbers while also considering a borrow-in bit from a previous stage. It produces a difference bit and a borrow-out bit as its outputs.

Program:

Gate-level Modelling:

```
module decoder(a,b,d0,d1,d2,d3);
input a,b;
output d0,d1,d2,d3;
wire w1,w2;
not g1(w1,a);
not g2(w2,b);
and g3(d0,w1,w2);
and g4(d1,w1,b);
and g5(d2,a,w2);
and g6(d1,a,b);
endmodule
```

Data-flow Modelling:

```
module decoder1(a,b,d0,d1,d2,d3);
input a,b;
output d0,d1,d2,d3;
assign nota=~a;
assign notb=~b;
assign d0=nota&notb;
assign d1=nota&b;
assign d2=a&notb;
assign d3=a&b;
endmodule;
```

Truth Table:

a	b	d0	d1	d2	d3	
0	0	1	0	0	0	d0=a'b'
0	1	0	1	0	0	d1=a'b
1	0	0	0	1	0	d2=ab'
1	1	0	0	0	1	d3=ab

3) 3-Bit Comparator

Theory: A 3-Bit Comparator is a combinational circuit that compares 2 3-Bit binary numbers (A&B) and produces output indicating their relative magnitudes. The comparator has 3 outputs: A>B, A=B, A<B. The circuit checks bit by bit from the most significant bit (MSB) to the least significant bit (LSB) to determine the relationship. Such comparators are commonly used in arithmetic and logic units and control circuits.

Program:

```
module comp_3bit(A,B,E,G,L);
input [2:0] A,B;
output E,G,L;
wire x0,w1,x2;
assign x[0]=A[0]-^B[0];
assign x1=A[1]-^B[1];
assign x2=A[2]-^B[2];
assign E=x2&x1&x0;
assign G=(A[2] & (~B[2])) | (x2 & A[1] & (~B[1])) | (x2 & x1 &(A[0] & (~B[0])));
assign L=(B[2] & (~A[2])) | (x2 & (B[1] & (~A[1]))) | (x2 & x1 & (B[0] & (~A[0])));
endmodule

module tb_comp();
reg [2:0] A,B;
wire E,G,L;
comp_3bit mm0(E,G,L,A,B);
initial
begin
A=3'b111; B=3'b111;
#100 A=3'b111; B=3'b001;
#100 A=3'b001; B=3'b111;
end
endmodule
```

Truth Table:

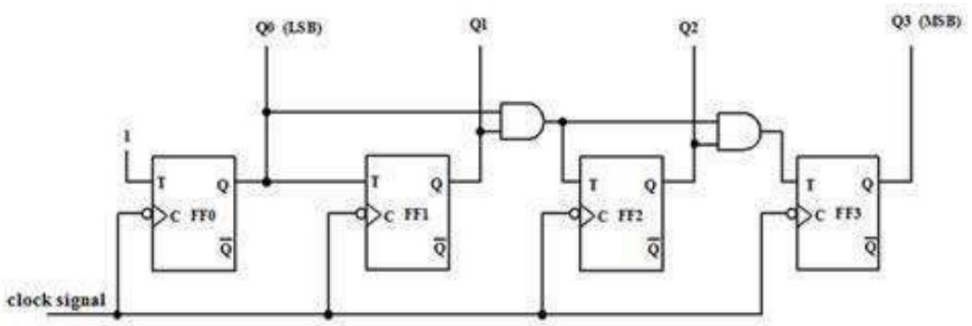
A	B	E	G	L
000	000	1	0	0
000	001	0	0	1
000	010	0	0	1
000	011	0	0	1
000	100	0	0	1
---	---	---	---	---
111	110	0	1	0
111	111	1	0	1

4) Up Counter

Theory: Up Counter is a sequential circuit where all flip-flops are connected to the same clock pulse and the output state increments with each pulse.

Program:

```
module up_counter(Q,clk,rst);
input clk,rst;
output [3:0] Q;
reg [3:0] Q;
always @ (posedge clk or posedge rst)
begin
if(rst)
    Q<=4'b0;
else
    Q<=Q+1;
end
endmodule

module up_counter_test;
wire [3:0] Q;
reg clk, rst;
up_counter c1(Q,clk,rst);
always
#5 clk=~clk;
initial
begin
clk=0; rst=1;
#10 rst=0;
#200 $stop;
end
endmodule
```



Truth Table:

clk	Q2	Q1	Q0

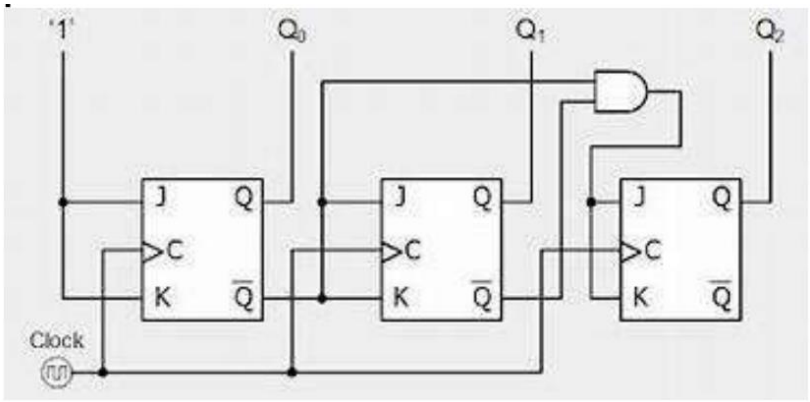
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

5) Down Counter

Theory: Down Counter is a sequential circuit where all flip-flops are connected to the same clock pulse and the output state decrements with each pulse.

Program:

```
module down_counter(Q,clk,rst);
input clk,rst;
output [3:0] Q;
reg [3:0] Q;
always @ (posedge clk or posedge rst)
begin
if(rst)
    Q<=4'b0;
else
    Q<=Q-1;
end
endmodule

module down_counter_test;
wire [3:0] Q;
reg clk, rst;
down_counter c1(Q,clk,rst);
always
#5 clk=~clk;
initial
begin
clk=0; rst=1;
#10 rst=0;
#200 $stop;
end
endmodule
```



Truth Table:

clk	Q2	Q1	Q0

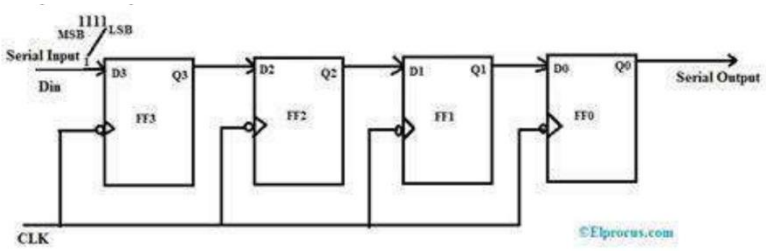
0	0	0	0
1	1	1	1
2	1	1	0
3	1	0	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1

6) SISO

Theory: A SISO shift register is a sequential circuit that takes one bit of data in (serially) per clock pulse and outputs one bit of data (serially) per clock pulse.

Program:

```
module siso(so,si,clk,rst);
input si,clk,rst;
output so;
reg [3:0] q;
always @ (posedge clk or posedge rst)
begin
if(rst)
    q<=4'b0;
else
    q<={si,q[3:1]};
end
assign so=q[0];
endmodule

module siso_test;
wire so;
reg si,clk,rst;
siso s1(so,si,clk,rst);
always
#5 clk=~clk;
initial
begin
si=0; clk=0; rst=1;
#10 rst=0;
#10 si=1;
#10 si=0;
#10 si=0;
#10 si=1;
#10 si=1;
#10 si=0;
#50 $stop;
end
endmodule
```



Truth Table:

si	clk	rst	so

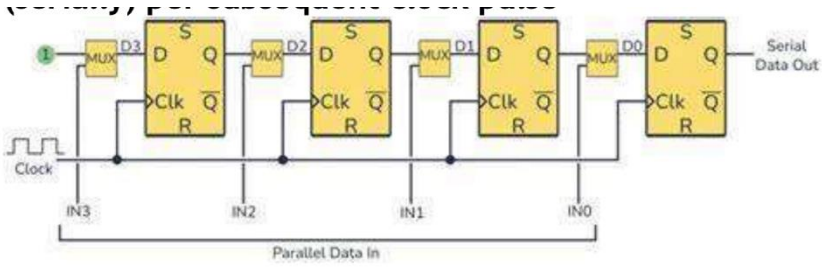
1	0	0	0
1	1	0	1
1	0	0	1
0	1	0	0

7) PISO

Theory: PISO shift register is a sequential circuit that takes all data bits simultaneously (parallely) on one clock pulse but outputs are one bit of data (serially) per subsequent clock pulse.

Program:

```
module piso(so,in,load,clk,rst);
input load,clk,rst;
output so;
input[3:0] in;
reg [3:0] q;
always @ (posedge clk or posedge rst)
begin
if(rst)
    q<=4'b0;
else if(load)
    q<=in;
else
    q<={1'b0,q[3:1]};
end
assign so=~q[0];
endmodule

module piso_test;
wire so;
reg load,clk,rst;
reg [3:0] in;
piso p1(so,in,load,clk,rst);
always
#5 clk=~clk;
initial
begin
load=1; clk=0; rst=1;
in=4'b1001;
#10 rst=0;
#10 load=0;
#10 load=1; in=4'b0011;
#10 load=0;
#50 $stop;
end
endmodule
```



Truth Table:

in	rst	clk	load

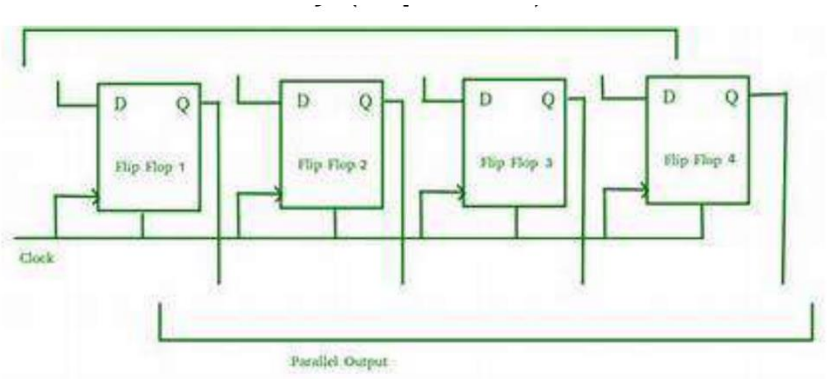
1001	1	0	1
1001	1	1	1
0011	0	0	1
0011	0	1	0

8) PIPO

Theory: PIPO shift register is a sequential circuit that takes all data bits simultaneously (parallely) on one clock pulse and outputs are all stored data bits simultaneously (parallely).

Program:

```
module pipo(q,in,load,clk,rst);
input load,clk,rst;
output [3:0] q;
input[3:0] in;
reg [3:0] q;
always @ (posedge clk or posedge rst)
begin
if(rst)
    q<=4'b0;
else if(load)
    q<=in;
else
    q<=q;
end
endmodule

module pipo_test;
wire [3:0] q;
reg load,clk,rst;
reg [3:0] in;
pipo p1(q,in,load,clk,rst);
always
#5 clk=~clk;
initial
begin
load=1; clk=0; rst=1;
in=4'b1001;
#10 rst=0;
#10 load=0;
#10 load=1; in=4'b0011;
#10 load=0;
#50 $stop;
end
endmodule
```



Truth Table:

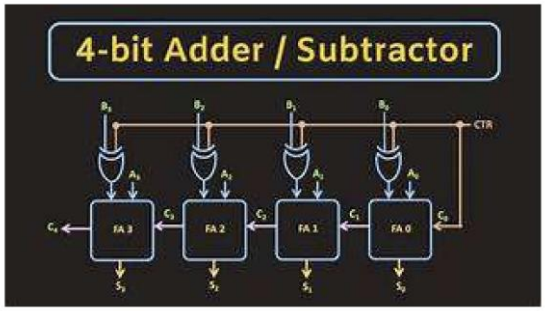
in	rst	clk	load
1001	1	0	1
1001	1	1	1
1001	0	0	0
0011	0	1	1

9) Ripple Carry Adder-Subtractor

Theory: A ripple carry adder-subtractor is a combinational circuit that can perform both addition and subtraction on binary numbers. It uses a ripple carry adder along with XOR gates and a control input. When the control input is 0, the circuit performs addition. When the control input is 1, the circuit performs subtraction by taking the 2's complement of one operand (inverting the bits and adding 1). The operation is selected based on the control signal.

Program:

```
module rca_add_sub (A, B, CTRL, S, Cout);
input  [3:0] A, B;
input  CTRL;
output [3:0] S;
output Cout;
wire [3:0] B_mod;
wire c1, c2, c3;
assign B_mod = B ^ {4{CTRL}};
full_adder fa0(A[0], B_mod[0], CTRL, S[0], c1);
full_adder fa1(A[1], B_mod[1], c1  , S[1], c2);
full_adder fa2(A[2], B_mod[2], c2  , S[2], c3);
full_adder fa3(A[3], B_mod[3], c3  , S[3], Cout);
endmodule

module full_adder(x, y, cin, sum, cout);
input x, y, cin;
output sum, cout;
assign sum  = x ^ y ^ cin;
assign cout = (x & y) | (cin & (x ^ y));
endmodule

module tb_rca_add_sub;
reg  [3:0] A, B;
reg  CTRL;
wire [3:0] S;
wire Cout;
rca_add_sub uut(A, B, CTRL, S, Cout);
initial
begin
$monitor("CTRL=%b  A=%b  B=%b  ->  S=%b Cout=%b", CTRL, A, B, S, Cout);
CTRL=0; A=4'b0001; B=4'b0000; #10;
CTRL=0; A=4'b0010; B=4'b0100; #10;
CTRL=0; A=4'b1011; B=4'b0110; #10;
CTRL=1; A=4'b0001; B=4'b0000; #10;
CTRL=1; A=4'b0010; B=4'b0100; #10;
CTRL=1; A=4'b1011; B=4'b0110; #10;
CTRL=1; A=4'b0101; B=4'b0011; #10;
$stop;
end
endmodule
```



Truth Table:

CTRL	A	B	S	Cout

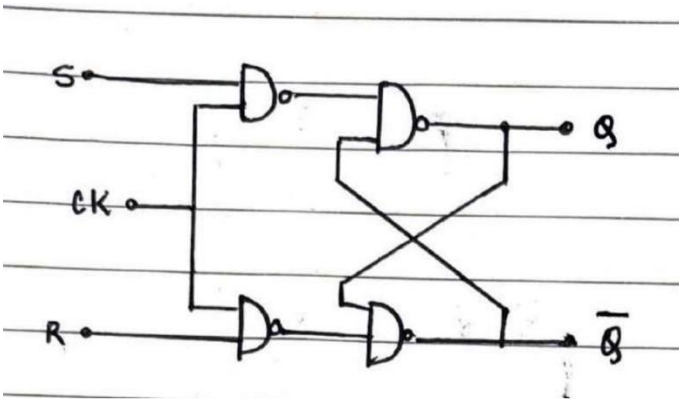
0	0001	0000	0001	0
0	0010	0100	0110	0
0	1011	0110	0001	1
1	0001	0000	0001	1
1	0010	0100	1110	0
1	1011	0110	0101	1
1	0101	0011	0010	1

10) SR FF

Theory: An SR flip-flop is a bistable device used to store a single bit, with two inputs: Set (S) and Reset (R). Setting S=1, R=0 stores 1; S=0, R=1 stores 0; both inputs low hold the previous state. S=1, R=1 leads to an invalid state

Program:

```
module srff (r, s, clk, rst, q, qbar);
input r, s, clk, rst;
output q, qbar;
reg q;
assign qbar = ~q;

always @ (posedge clk or posedge rst)
begin
    if (rst)
        q <= 1'b0;
    else
    begin
        case ({r, s})
            2'b00: q <= q;
            2'b01: q <= 1'b1;
            2'b10: q <= 1'b0;
            2'b11: q <= 1'bz;
        endcase
    end
end
endmodule

module srff_test;
reg r, s, clk, rst;
wire q, qbar;

srff sr1(r, s, clk, rst, q, qbar);

always #5 clk = ~clk;

initial
begin
    r = 0; s = 0; clk = 0; rst = 1;
    #10 rst = 0;
    #10 r = 0; s = 1;
    #10 r = 1; s = 0;
    #10 r = 1; s = 1;
    #50 $stop;
end
endmodule
```



Truth Table:

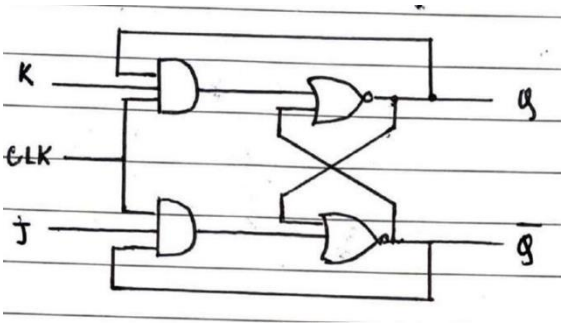
S	R	Q
0	0	No change
0	1	0
1	0	1
1	1	X

11) JK FF

Theory: A JK flip-flop is a versatile memory circuit using inputs J and K. When J=1 and K=0, it sets; J=0 and K=1, it resets; both low hold the state; both high make the output toggle. It overcomes the invalid state and enables reliable sequential logic.

Program:

```
module jkff (j, k, clk, rst, q, qbar);
input j, k, clk, rst;
output q, qbar;
reg q;
assign qbar = ~q;

always @ (posedge clk or posedge rst)
begin
    if (rst)
        q <= 1'b0;
    else
    begin
        case ({j, k})
            2'b00: q <= q;
            2'b01: q <= 1'b0;
            2'b10: q <= 1'b1;
            2'b11: q <= ~q;
        endcase
    end
end
endmodule


module jkff_test;
reg j, k, clk, rst;
wire q, qbar;
jkff jk(j, k, clk, rst, q, qbar);

always #5 clk = ~clk;

initial
begin
    j = 0; k = 0; clk = 0; rst = 1;
    #10 rst = 0;
    #10 j = 0; k = 1;
    #10 j = 1; k = 0;
    #10 j = 1; k = 1;
    #50 $stop;
end
endmodule
```



Truth Table:

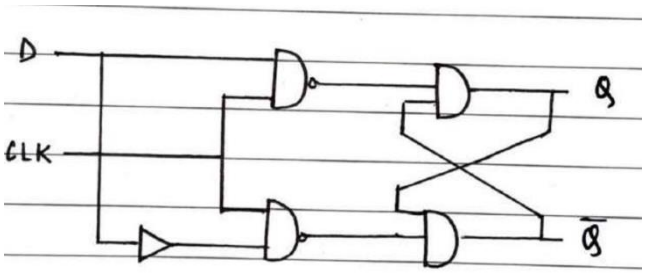
J	K	Q
0	0	No change
0	1	0
1	0	1
1	1	Toggle

12) D FF

Theory: A D flip-flop is a memory device with a data input (D) and a clock input. On each clock pulse, it stores the value from D at the output Q. The Q output follows the D input only when triggered by the clock, ensuring stable, synchronized data storage

Program:

```
module dff (d, clk, rst, q, qbar);
input d, clk, rst;
output q, qbar;
reg q;
assign qbar = ~q;

always @ (posedge clk or posedge rst)
begin
    if (rst)
        q <= 1'b0;
    else
        q <= d;
end
endmodule


module dff_test;
reg d, clk, rst;
wire q, qbar;
dff d1(d, clk, rst, q, qbar);

always #5 clk = ~clk;

initial
begin
    d = 1; clk = 0; rst = 1;
    #10 rst = 0;
    #10 d = 0;
    #10 d = 1;
    #50 $stop;
end
endmodule
```



Truth Table:

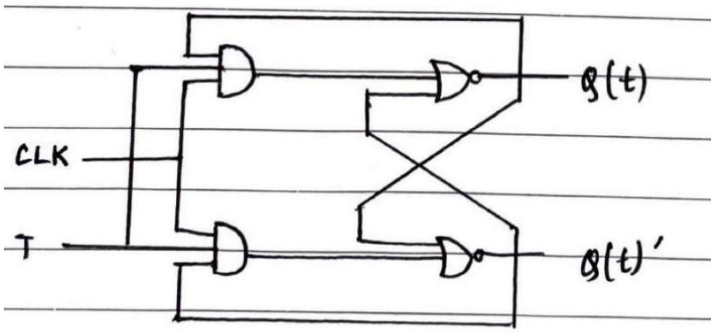
D	Q
0	0
1	1

13) T FF

Theory: A T flip-flop is a bistable circuit used for toggling output. With each clock pulse, if the input T is high, the output Q switches its state from 0 to 1 or from 1 to 0. If T is low, the output holds its state, enabling frequency division and counters.

Program:

```
module tff(t, clk, rst, q, qbar);
input t, clk, rst;
output q, qbar;
reg q;
assign qbar = ~q;

always @ (posedge clk or posedge rst)
begin
    if (rst)
        q <= 1'b0;
    else if (t)
        q <= ~q;
    else
        q <= q;
end
endmodule


module tff_test;
reg t, clk, rst;
wire q, qbar;
tff t1(t, clk, rst, q, qbar);

always #5 clk = ~clk;

initial
begin
    t = 1; clk = 0; rst = 1;
    #10 rst = 0;
    #10 t = 0;
    #10 t = 1;
    #50 $stop;
end
endmodule
```



Truth Table:

T	Q
0	No change
1	Toggle