Verilog Program Examples

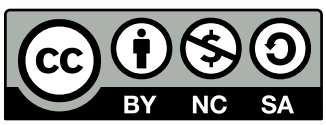
using

iverilog and *GTKwave*

GURUPRASAD

ASSISTANT PROFESSOR
DEPT. OF ELECTRONICS & COMMUNICATION
MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL, INDIA

Published in September, 2021.



Contents

List of Figures

Chapter 1

Data flow style of modeling

1. Write a behavioral verilog code for a full adder using *assign* statements.

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

assign sum = a ^ b ^ c;
assign carry = (a&b) | (a&c) | (c&b);
endmodule
```

Testbench

```
module full_adder_test ;
reg a,b,c;
wire sum,carry;

// Instanstiation
full_adder FA(a,b,c,sum,carry);

initial
begin
        $dumpfile("full_adder.vcd");
        $dumpvars(0,full_adder_test);
        $monitor($time,"a =%b, b=%b, c=%b, sum=%b, carry=%b",a,b,c,sum,
   carry);
        a=0; b=0; c=0;
        #5 c=1;
        #5 b=1;
        #5 a=1;
        #5 $finish;
end
endmodule
```

Result

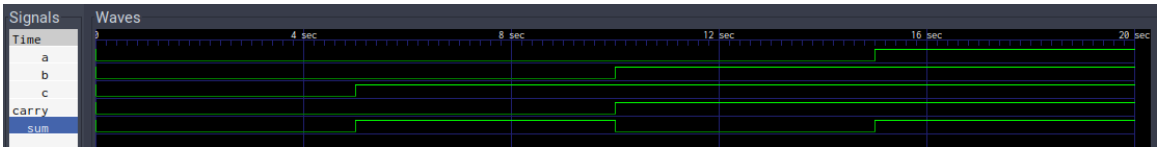


Figure 1.1: Simulation result of Program-1 in Chapter-1 (Full adder)

2. Write a behavioral verilog code for a gated D latch using conditional assignment statement.

```
module Dff_conditional (D, En, Q);
input D,En;
output Q;

assign Q = En ? D : Q ;
// D latch using dataflow and blocking assignment.
// Initial value for Q can not be assigned during the simulation
endmodule
```

Testbench

```
module Dff_conditional_test ;
reg D,En;
wire Q;

// Instanstiation
Dff_conditional G0 (D, En, Q);

initial
begin
        $dumpfile("Dff_conditional.vcd");
        $dumpvars(0,Dff_conditional_test);
        $monitor($time," D=%b, En=%b, Q=%b",D,En,Q);
        D=1;En=0;
        #5 En=1;
        #5 D=0;
        #5 $finish;
end
endmodule
```

Result

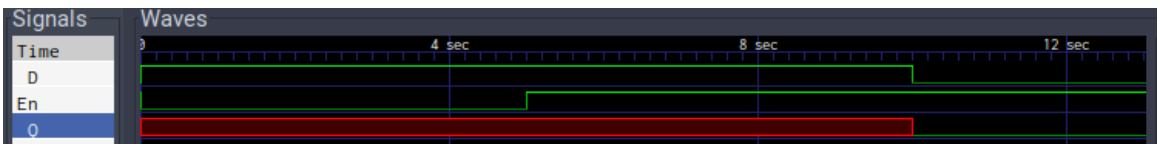


Figure 1.2: Simulation result of Program-2 in Chapter-1 (D latch)

3. Write a verilog code for 2 to 1 Multiplexer in data flow style.

```
module mux_2to1 (in,sel,out);
input [1:0] in;
input sel;
output out;

assign out=in[sel];
// note index number of ‘in’ variable. The value of sel is converted to
   // decimal automatically.
endmodule
```

Testbench

```
module mux_2to1_test ;
reg [1:0] in;
reg sel;
wire out;

// Instanstiation
mux_2to1 MUX (in,sel,out);

initial
begin
        $dumpfile("mux_2to1.vcd");
        $dumpvars(0,mux_2to1_test);
        $monitor($time," in=%2b, sel=%b, out=%b",in,sel,out);
        in=2'b01;sel=0;
        #5 sel=1;
        #5 in=2'b11;
        #5 sel=0;
        #5 $finish;
end
endmodule
```

Result

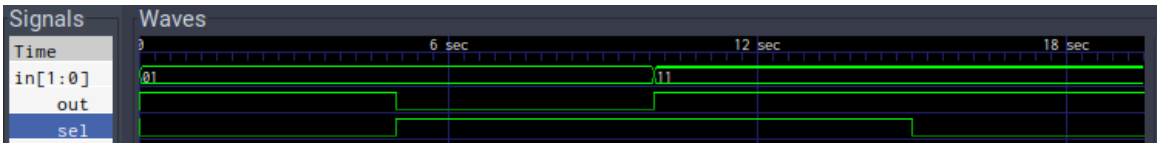


Figure 1.3: Simulation result of Program-3 in Chapter-1 (2:1 Mux)

Chapter 2

Verilog coding using procedural statements

1. Write a behavioral verilog code for 2 to 4 decoder with active low output and active high input.

```
module decoder_2to4 (din, enable, dout);
input [1:0] din;
input enable;
output reg [3:0] dout;

always @ (*)
begin
        if (enable)
        case (din)
                0 : dout = 4'b1110;
                1 : dout = 4'b1101;
                2 : dout = 4'b1011;
                3 : dout = 4'b0111;
                default: dout =4'b1111;
        endcase
        else
                dout = 4'b1111;
end
endmodule
```

Testbench

```
module decoder_2to4_test ;
reg [1:0] din;
reg enable;
wire [3:0] dout;

// Instanstiation
decoder_2to4 DECODER (din, enable, dout);

initial
begin
```

```
        $dumpfile("decoder_2to4.vcd");
        $dumpvars(0,decoder_2to4_test);
        $monitor($time," din=%2b, enable=%b, dout=%4b",din,enable,dout);
        #10 enable=1;
        #20 din=2'b11;
        #20 din=2'b01;
        #20 din=2'b10;
end

initial
begin
        din=2'b00;
        enable=0;
        #100 $finish;
end
endmodule
```

Result

Figure 2.1: Simulation result of Program-1 in Chapter-2 (2:4 Decoder)

2. Write a behavioral verilog code for 8 to 3 priority encoder with highest priority assigned for lower bits.

```
module priority_encoder (din, enable, dout);
input [7:0] din;
input enable;
output reg [2:0] dout;

always @ (*)
begin
        if (enable)
        casez (din) // Casez is used for handling dont cares.
                8'b???????1 : dout=3'b000;
                8'b??????1? : dout=3'b001;
                8'b?????1?? : dout=3'b010;
                8'b????1??? : dout=3'b011;
                8'b???1???? : dout=3'b100;
                8'b??1????? : dout=3'b101;
                8'b?1?????? : dout=3'b110;
                8'b1??????? : dout=3'b111;
                default: dout =3'bzzz;
        endcase
```

```
        else
                dout = 3'bzzz;
end
endmodule
```

Testbench

```
module priority_encoder_test ;
reg [7:0] din;
reg enable;
wire [2:0] dout;

// Instanstiation
priority_encoder ENCODER (din, enable, dout);

initial
begin
        $dumpfile("priority_encoder.vcd");
        $dumpvars(0,priority_encoder_test);
        $monitor($time," din=%8b, enable=%b, dout=%3b",din,enable,dout);
        #10 enable=1;
        #20 din=8'b11100011;
        #20 din=8'b10000000;
        #20 din=8'b11110000;
        #20 din=8'b00000000;
end

initial
begin
        din=8'b00000001;
        enable=0;
        #100 $finish;
end
endmodule
```

Result

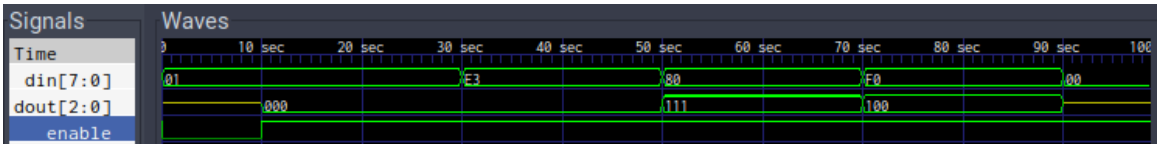


Figure 2.2: Simulation result of Program-2 in Chapter-2 (Priority encoder)

3. Write a verilog code for edge triggered D flip-flop with asynchronous set and reset.

```
module Dff_edge (D, reset, set, clk, Q, Qbar);
input D, set, reset, clk;
output reg Q;
output Qbar;
```

```
assign Qbar= ~Q;

always @ (posedge clk or negedge set or negedge reset)
begin
        if (reset == 0)
                Q<=0;
        else if (set == 0)
                Q<=1;
        else
                Q<=D;
end // ‘assign’ and ‘always’ statements are concurrent.
endmodule
```

Testbench

```
module Dff_edge_test ;
reg D,set,reset,clk;
wire Q,Qbar;

// Instanstiation
Dff_edge G0 (D, reset, set, clk, Q, Qbar );

//Clock generation
always #5 clk=~clk ;

// Initialisation
initial
begin
        D=1;    reset=0;
        set=1;
        clk=0;
        #100 $finish;
end

// Data plot and data change
initial
begin
        $dumpfile("Dff_edge.vcd");
        $dumpvars(0,Dff_edge_test);
        $monitor($time," D=%b, reset=%b, set=%b, Q=%b, Qbar=%b", D, reset
   , set, Q, Qbar);
        #12 reset=1;
        #20 D=0;
        #10 set=0;
        #20 set=1;
        #10 D=1;
end
endmodule
```

Result

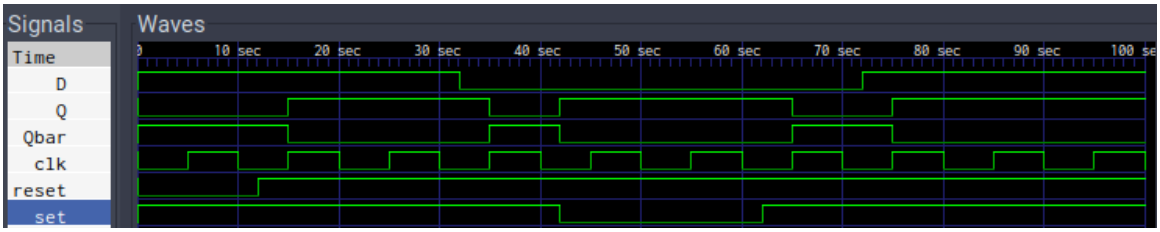


Figure 2.3: Simulation result of Program-3 in Chapter-2 (Edge triggered D FF)

4. Write a verilog code using sequential statements for a ring counter.

```
module ring_counter (clk, init, count);
input init, clk;
output reg [7:0] count;

// Use of non blocking assignment is preferred here.
always @ (posedge clk)
begin
        if (init == 1) // initialize the counter to 10000000
                count <= 8'b10000000;
        else
        begin
                count[0] <= count [7];
                count[1] <= count [0];
                count[2] <= count [1];
                count[3] <= count [2];
                count[4] <= count [3];
                count[5] <= count [4];
                count[6] <= count [5];
                count[7] <= count [6];
        end
end
endmodule
```

Testbench

```
module ring_counter_test ;
reg init, clk;
wire [7:0] count;

// Instanstiation
ring_counter G0 (clk, init, count);

// Clock generation
always #5 clk=~clk ;

// Initialization
```

```
initial
begin
        init=1;
        clk=0;
        #100 $finish;
end

// Data plot and data change
initial
begin
        $dumpfile("ring_counter.vcd");
        $dumpvars(0,ring_counter_test);
        $monitor($time," init=%b, count=%8b", init, count);
        #12 init=0;
end
endmodule
```

Result

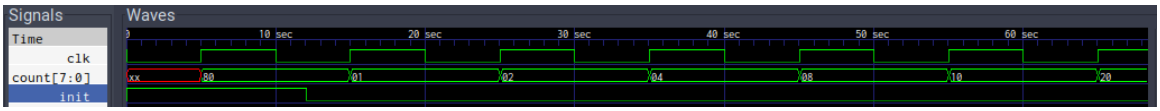


Figure 2.4: Simulation result of Program-4 in Chapter-2 (Ring counter)

5. Write a verilog code for a 4 bit ripple up counter with asynchronous clear.

```
module ripple_counter (clk, reset, Q0, Q1, Q2, Q3);
input clk, reset;
output reg Q0, Q1, Q2, Q3;

always @ (negedge clk or posedge reset)
begin
        if (reset)
                Q0 <= 0;
        else
                Q0 <= ~Q0;
end

always @ (negedge Q0 or posedge reset)
begin
        if (reset)
                Q1 <= 0;
        else
                Q1 <= ~Q1;
end

always @ (negedge Q1 or posedge reset)
```

```
begin
        if (reset)
                Q2 <= 0;
else
                Q2 <= ~Q2;
end

always @ (negedge Q2 or posedge reset)
begin
        if (reset)
                Q3 <= 0;
        else
                Q3 <= ~Q3;
end
endmodule
```

Testbench

```
module ripple_counter_test ;
reg clk, reset;
wire Q0, Q1, Q2, Q3;

// Instanstiation
ripple_counter G0 (clk, reset, Q0, Q1, Q2, Q3);

// Clock generation
always #5 clk=~clk;

// Initialisation
initial
begin
        reset=1;
        clk=0;
        #200 $finish;
end

// Data plot and data change
initial
begin
        $dumpfile("ripple_counter.vcd");
        $dumpvars(0,ripple_counter_test);
        $monitor($time," reset=%b, Q0=%b, Q1=%b, Q2=%b, Q3=%b",reset, Q0,
    Q1, Q2, Q3 );
        #12 reset=0;
        #170 reset=1;
        #5 reset=0;
end

endmodule
```

Result

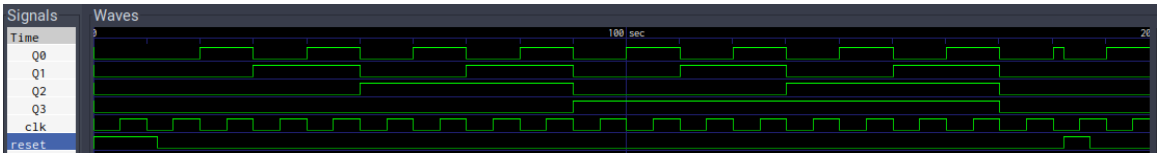


Figure 2.5: Simulation result of Program-5 in Chapter-2 (Ripple counter)

6. Write a verilog code in behavioral style for a synchronous up-down counter.

```
module counter_sync (mode, load, clk, reset, din, count);
input mode, reset, load, clk;
input  [0:7] din;
output reg [0:7] count;

// Index [0:7] makes data storage as 0th bit at left most and 7th
// bit as right most. If din<=00001011 then display will be 0B.
// However 0th bit is 0 and 7th bit is 1. Addition is din+1=0C.
always @ (posedge clk)
        if (reset == 1) // reset has the highest priority
                count <= 0;
        else if (load == 1) // if load is kept high, it will not count
                count <= din;
        else if ( mode )
                count <= count+1;
        else
                count <= count-1;
endmodule
```

Testbench

```
module counter_sync_test ;
reg mode, load, reset, clk;
reg [0:7] din;
wire [0:7] count;

// Instanstiation
counter_sync G0 (mode, load, clk, reset, din, count);

// Clock generation
always #5 clk=~clk ;

// Initialisation
initial
begin
        reset=1;
        load=0;
        mode=1;
```

```
        clk=0;
        din=8'b00000001;
        #150 $finish;
end

// Data plot and data change
initial
begin
        $dumpfile("counter_sync.vcd");
        $dumpvars(0,counter_sync_test);
        $monitor($time," mode=%b, load=%b, reset=%b, din=%8b, count=%8b",
    mode, load, reset, din, count);
        #12 reset=0;
        #50 mode=0;
        #50 load=1;
        #5 load=0;
end
endmodule
```

Result

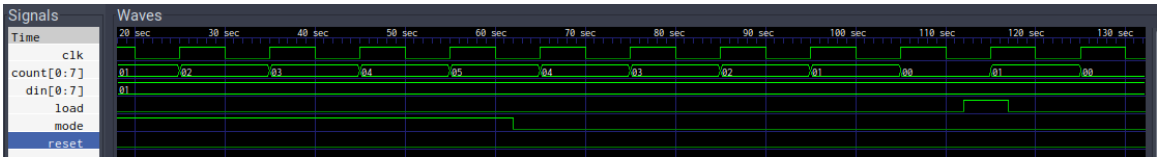


Figure 2.6: Simulation result of Program-6 in Chapter-2 (Up-Down counter)

7. Write a verilog code using sequential statements for a bidirectional shift register.

```
module bidir_shiftreg (rin, lin, mode,  clk, Q0,Q1,Q2,Q3);
input rin, lin, mode, clk;
output reg Q0,Q1,Q2,Q3;

// Use of non blocking assignment is preferred here.
always @ (posedge clk)
begin
        if (mode) // Shift right
        begin
                Q0 <= rin;
                Q1 <= Q0;
                Q2 <= Q1;
                Q3 <= Q2;
        end
        else // Shift left
        begin
                Q0 <= Q1;
                Q1 <= Q2;
```

```
                Q2 <= Q3;
                Q3 <= lin;
        end
end
endmodule
```

Testbench

```
module bidir_shiftreg_test ;
reg rin, lin, mode, clk ;
wire Q0,Q1,Q2,Q3;

bidir_shiftreg G0 (rin, lin, mode, clk, Q0,Q1,Q2,Q3);

// Clock generation
always #5 clk=~clk ;

initial
begin
        mode=1;
        clk=1;
        rin=0;
        lin=1;
        #100 $finish;
end

initial
begin
        $dumpfile("bidir_shiftreg.vcd");
        $dumpvars(0,bidir_shiftreg_test);
        $monitor($time," rin=%b, lin=%b, mode=%b, Q0=%b, Q1=%b, Q2=%b, Q3
   =%b", rin, lin, mode, Q0,Q1,Q2,Q3);
        #45 mode=0;
end
endmodule
```

Result

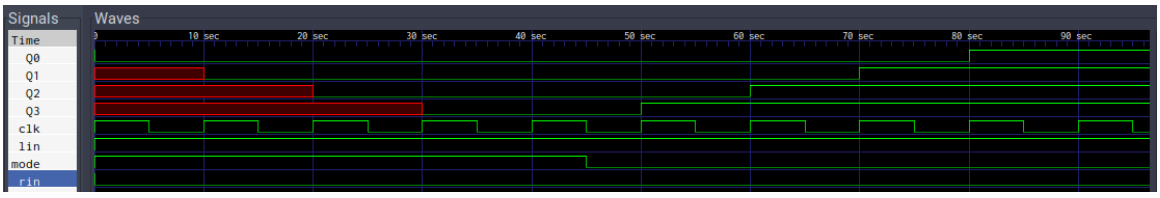


Figure 2.7: Simulation result of Program-7 in Chapter-2 (Bidirectional Shift register)

Chapter 3

Structural Modeling

1. Write a verilog code in structural style for a 4 bit binary ripple adder using full adders.

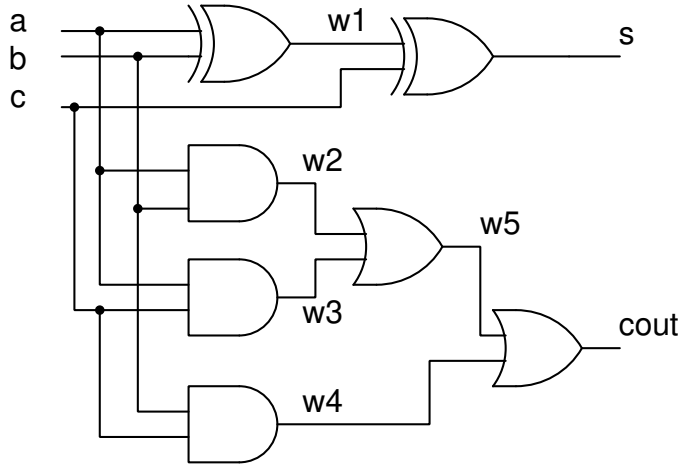


Figure 3.1: Structure of the full adder

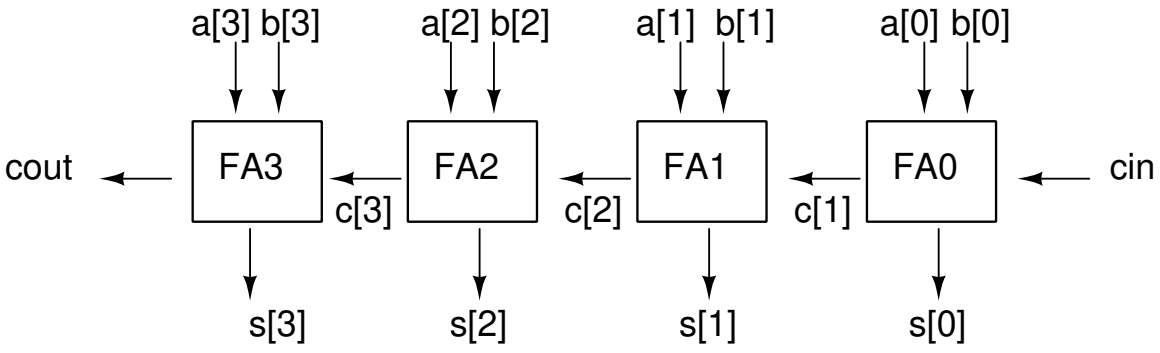


Figure 3.2: Structure of the 4 bit ripple adder

```
// Module for full adder using primitive gates
module full_adder (a,b,c,s,cout);
input a, b, c;
output s,cout;
```

```
wire w1, w2, w3, w4, w5;

// s=w1 xor c    w1=a xor b
xor (w1, a, b);
xor (s, c, w1);
// cout= w5 or w4   w5= w2 or w3
and (w2, a, b);
and (w3, a, c);
and (w4, b, c);
or (w5, w2, w3);
or (cout, w5, w4);
endmodule

// Module for 4 bit ripple carry adder
module ripple_adder ( a, b, cin, s, cout);
input [3:0] a, b;
input cin;
output [3:0] s;
output cout;
wire [3:1] c;

full_adder FA0 (a[0], b[0], cin,  s[0], c[1]);
full_adder FA1 (a[1], b[1], c[1], s[1], c[2]);
full_adder FA2 (a[2], b[2], c[2], s[2], c[3]);
full_adder FA3 (a[3], b[3], c[3], s[3], cout);
endmodule
```

Testbench

```
module ripple_adder_test ;
reg [3:0] a, b;
reg cin;
wire [3:0] s;
wire cout;

//Instanstiation
ripple_adder G0 (a, b, cin, s, cout);

initial
begin
        a=4'b0000;
        b=4'b0000;
        cin=0;
        #50 $finish;
end

initial
begin
        $dumpfile("ripple_adder.vcd");
```

```
        $dumpvars(0,ripple_adder_test);
        $monitor($time," a=%4b, b=%4b, cin=%b, s=%4b, cout=%b",a, b, cin,
    s, cout );
        #5 a=4'b1111;
        #5 cin=1;
        #5 b=4'b1111;
        #5 cin=0;
        #5 a=4'b1010;
end
endmodule
```

Result

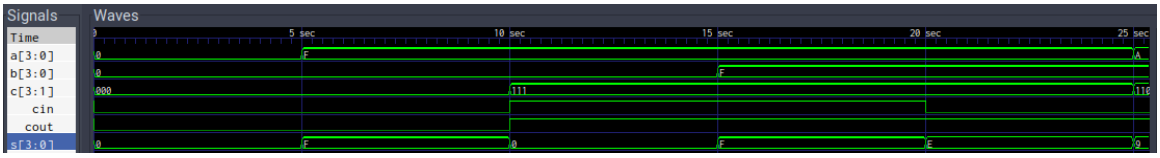


Figure 3.3: Simulation result of Program-1 in Chapter-3 (Ripple adder)

2. Write a verilog code in structural style for a BCD adder using 4 bit ripple adders.

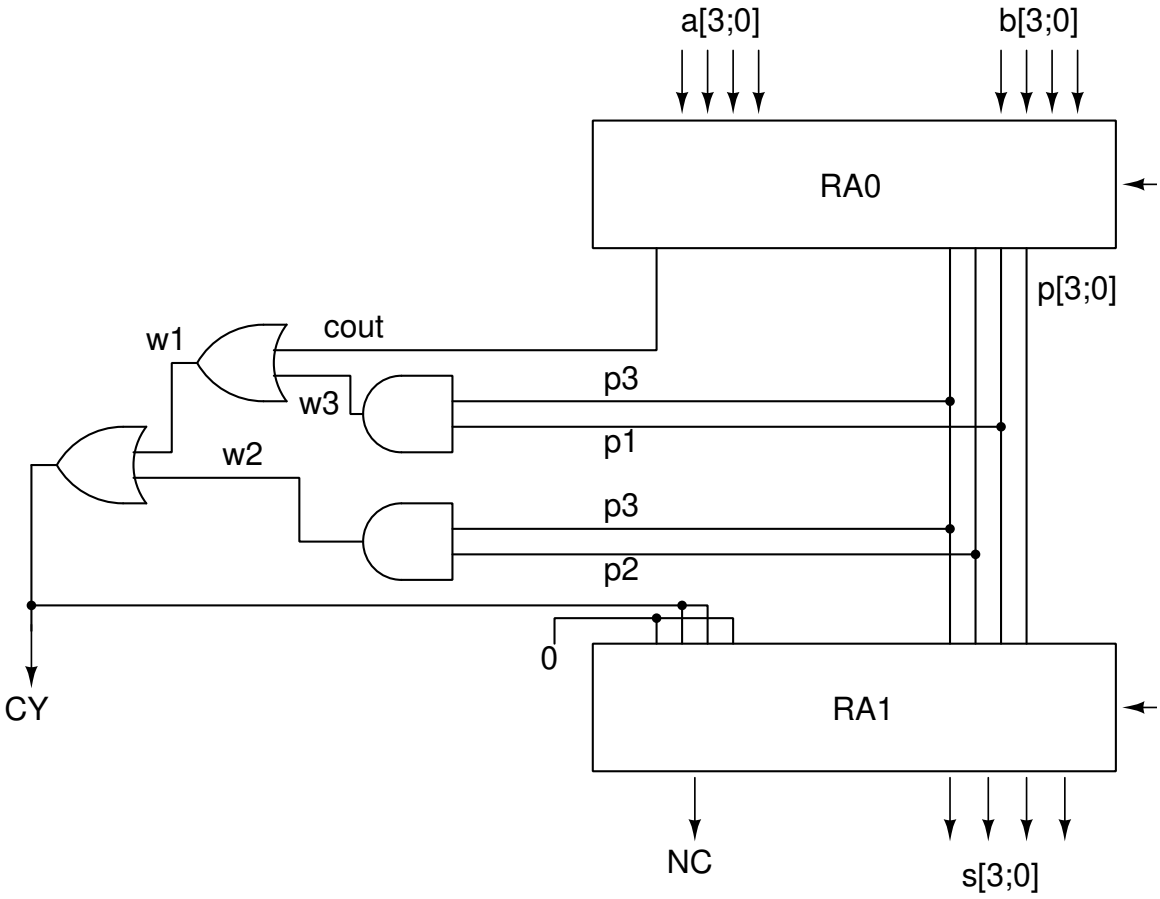


Figure 3.4: Structure of the BCD adder

```
// Module for full adder using primitive gates
module full_adder (a,b,c,s,cout);
input a, b, c;
output s,cout;
wire w1, w2, w3, w4, w5;

// s=w1 xor c    w1=a xor b
xor (w1, a, b);
xor (s, c, w1);
// cout= w5 or w4   w5= w2 or w3
and (w2, a, b);
and (w3, a, c);
and (w4, b, c);
or (w5, w2, w3);
or (cout, w5, w4);
endmodule

// Module for 4 bit ripple carry adder
module ripple_adder ( a, b, cin, s, cout);
input [3:0] a, b;
input cin;
output [3:0] s;
output cout;
wire [3:1] c;

full_adder FA0 (a[0], b[0], cin,  s[0], c[1]);
full_adder FA1 (a[1], b[1], c[1], s[1], c[2]);
full_adder FA2 (a[2], b[2], c[2], s[2], c[3]);
full_adder FA3 (a[3], b[3], c[3], s[3], cout);
endmodule

// Module for bcd adder
module bcd_adder (a, b, s, CY);
input [3:0] a, b;
output CY;
output [3:0] s;
wire w1, w2, w3, cout;
wire [3:0] p;

ripple_adder RA0 (a, b, 1'b0 , p, cout ); //one of the inputs is constant

and (w3, p[1], p[3]);
and (w2, p[2], p[3]);
or (w1, cout, w3);
or (CY, w1, w2);

ripple_adder RA1 ({1'b0,CY,CY,1'b0}, p, 1'b0 , s, );//One of the outputs
   is left open
```

```
// Note CY is input and as well as output
endmodule
```

Testbench

```
module bcd_adder_test ;
reg [3:0] a, b;
wire [3:0] s;
wire CY;

// Instanstiation
bcd_adder G0 ( a, b, s, CY);

initial
begin
        a=4'b0000;
        b=4'b0000;
        #50 $finish;
end

initial
begin
        $dumpfile("bcd_adder.vcd");
        $dumpvars(0,bcd_adder_test);
        $monitor($time,"a=%4b, b=%4b, s=%4b, CY=%b",a, b, s, CY );
        #5 a=4'b0110;
        #5 b=4'b0011;
        #5 b=4'b0111;
        #5 a=4'b1001;
        #5 b=4'b1001;
end
endmodule
```

Result

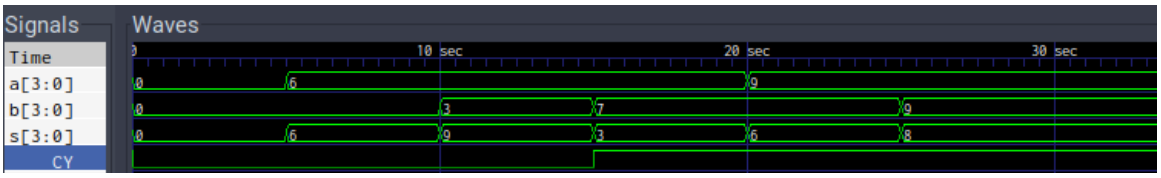


Figure 3.5: Simulation result of Program-2 in Chapter-3 (BCD adder)

3. Write a verilog code in structural style for a master slave JK flip flop using SR latch.

```
// Module for SR latch
module sr_latch (S, R, enable, Q, Qbar);
input S, R, enable;
output reg Q;
output Qbar;
```

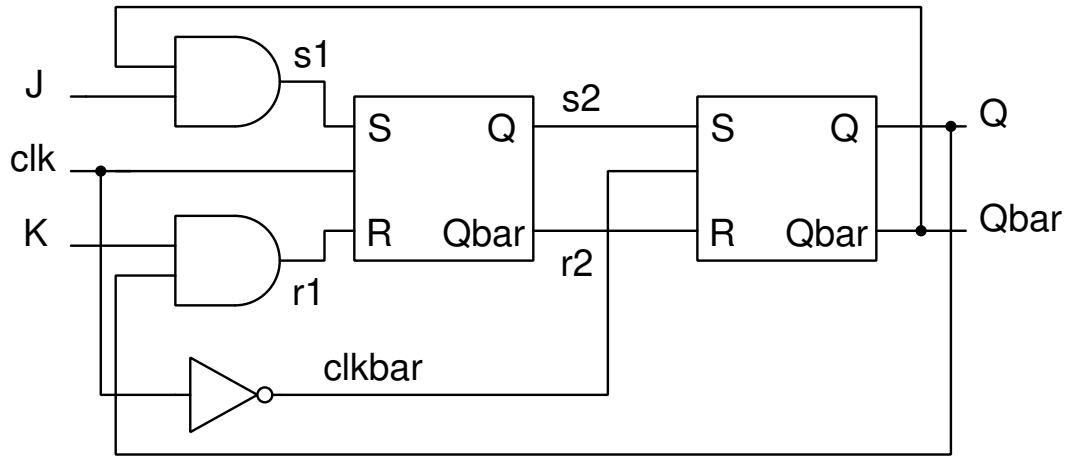


Figure 3.6: Structure of the Master slave JK FF

```
assign Qbar = ~Q;
initial Q = 0; // Initialize to avoid getting x in simulation.

always @ (S,R,enable)
        casez ({S,R,enable})
                3'b000:  Q = Q;
                3'b001:  Q = Q;
                3'b010:  Q = Q;
                3'b011:  Q = 0;
                3'b100:  Q = Q;
                3'b101:  Q = 1;
                3'b110:  Q = Q;
                3'b111:  Q = 0;
                default:  Q = Q;
        endcase
endmodule

// Module for Master slave flip flop
module master_slave_JK (J, K, clk, Q, Qbar);
input J,K,clk;
output Q, Qbar;
wire s1, r1, s2, r2, clkbar;

and G0 (s1, J, Qbar);
and G1 (r1, K, Q);
not G2 (clkbar,clk);
sr_latch G3 (s1, r1, clk, s2, r2);
sr_latch G4 (s2, r2, clkbar, Q, Qbar);
endmodule
```

Testbench

```
module master_slave_JK_test ;
```

```
reg J, K, clk;
wire Q, Qbar;

// Instanstiation
master_slave_JK G0 (J, K, clk, Q, Qbar);

always #5 clk=~clk;

initial
begin
        clk=1;
        J=1;
        K=0;
        #60 $finish;
end

initial
begin
        $dumpfile("master_slave_JK.vcd");
        $dumpvars(0,master_slave_JK_test);
        $monitor($time," J=%b, K=%b, Q=%b, Qbar=%b", J, K, Q, Qbar );
        #12 J=0;
        #10 K=1;
        #10 K=0;
        #10 J=1;
end
endmodule
```

Result

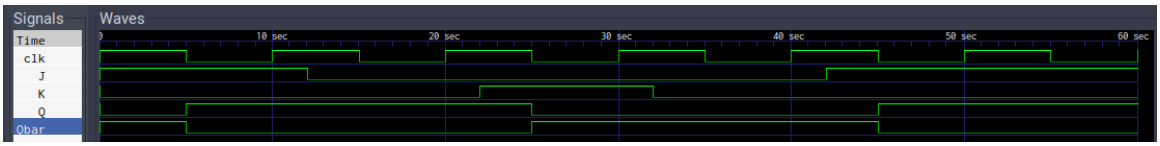


Figure 3.7: Simulation result of Program-3 in Chapter-3 (Master-Slave FF)

4. Write a verilog code in structural style for a negative edge triggered ripple decade counter using JK flip-flops.

```
// Module for negative edge triggered JK flip-flop
module JKff_edge (J, K, clk,clr, Q, Qbar);
input J, K, clk,clr;
output Qbar;
output reg Q;

assign Qbar = ~Q;

always @ (negedge clk or negedge clr)
```

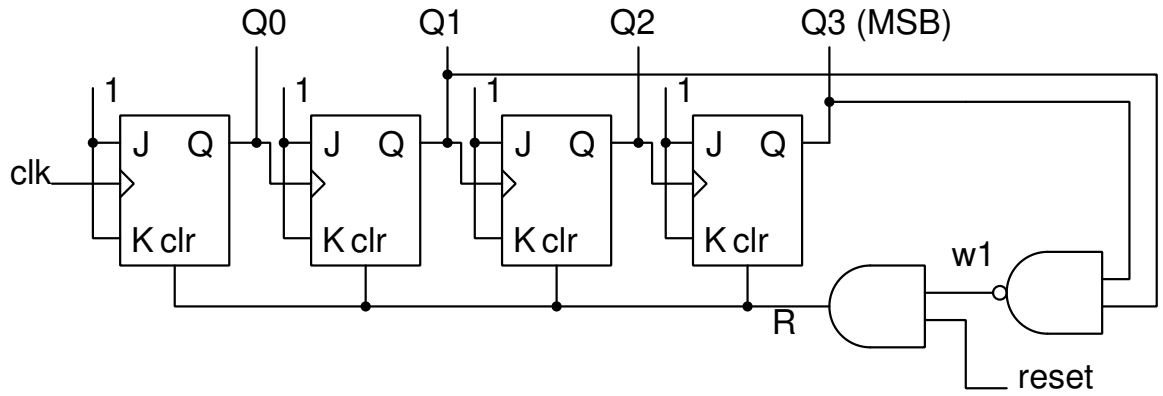


Figure 3.8: Structure of the ripple decade counter

```
begin
        if (clr == 0)
                Q <= 0;
        else
                case({J,K})
                        2'b00 : Q<=Q;
                        2'b01 : Q<=0;
                        2'b10 : Q<=1;
                        2'b11 : Q<=~Q;
                        default: Q<=Q;
                endcase
        end
endmodule

// Module for decade ripple counter
module ripple_decade_counter (clk, reset, Q0, Q1, Q2, Q3);
input clk,reset;
output Q0, Q1, Q2, Q3;
wire w1, R;
// Reset input is required or else initial value of Qs will be x
// and counter will not work.
nand G0 (w1, Q3, Q1);
and G1 (R,w1,reset); // An AND gate is used to provide reset capability.
JKff_edge FF0 (1'b1, 1'b1, clk, R, Q0, );
JKff_edge FF1 (1'b1, 1'b1, Q0,  R, Q1, );
JKff_edge FF2 (1'b1, 1'b1, Q1,  R, Q2, );
JKff_edge FF3 (1'b1, 1'b1, Q2,  R, Q3, );
endmodule
```

Testbench

```
module ripple_decade_counter_test ;
reg clk,reset;
wire Q0, Q1, Q2, Q3;

// Instanstiation
```

```
ripple_decade_counter G0 (clk, reset, Q0, Q1, Q2, Q3);

always #5 clk=~clk;

initial
begin
        clk=0;
        reset=0;
        #200 $finish;
end

initial
begin
        $dumpfile("ripple_decade_counter.vcd");
        $dumpvars(0,ripple_decade_counter_test);
        $monitor($time," reset=%b, Q0=%b, Q1=%b, Q2=%b, Q3=%b",reset, Q0,
    Q1, Q2, Q3 );
        #34 reset=1;
end
endmodule
```

Result

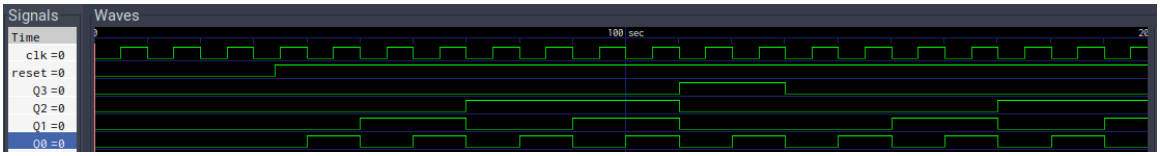


Figure 3.9: Simulation result of Program-4 in Chapter-3 (Ripple decade counter)

5. Write a verilog code in structural style for a 3 bit synchronous counter using T flip-flops.

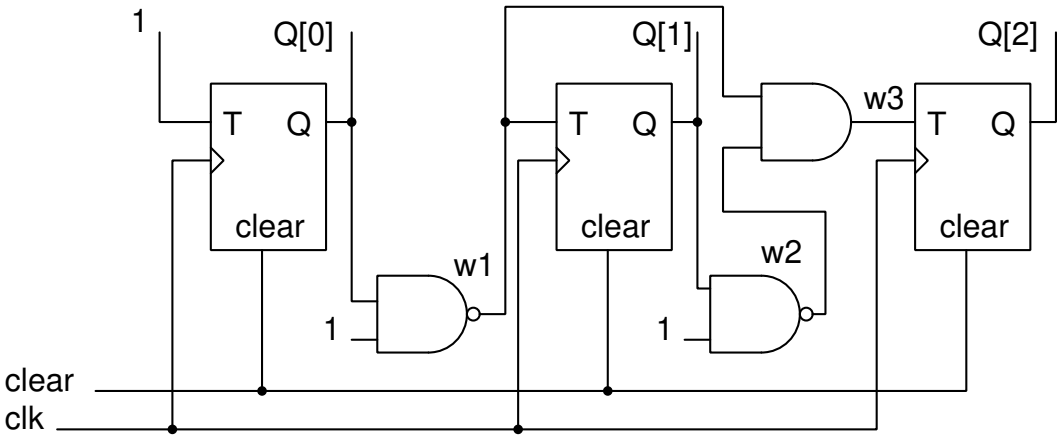


Figure 3.10: Structure of the Synchronous down counter

```
// Module for positive edge triggered T FF
module tff (Q, T, clear, clk);
input T,clear, clk;
output reg Q;

always @ (posedge clk or negedge clear)
begin
        if ( clear==0)
                Q<=0;
        else if ( T==1)
                Q<=~Q;
        else Q<=Q;
end
endmodule

// Module for 3 bit synchronous counter
module synchronous_counter (clear, clk, Q);
input clear, clk;
output [2:0] Q;
wire w1, w2, w3;

tff FF0 (Q[0], 1'b1, clear, clk);
tff FF1 (Q[1], w1 , clear, clk);
tff FF2 (Q[2], w3,  clear, clk);
nand G0 (w1,1'b1, Q[0]);
nand G1 (w2,1'b1, Q[1]);
and G2 (w3, w1, w2);
endmodule
```

Testbench

```
module synchronous_counter_test ;
reg clear, clk;
wire [2:0] Q;

// Instanstiation
synchronous_counter G0 (clear, clk, Q);

always #5 clk=~clk;

initial
begin
        clear=0;
        clk=1;
        #150 $finish;
end

initial
begin
```

```
        $dumpfile("synchronous_counter.vcd");
        $dumpvars(0,synchronous_counter_test);
        $monitor($time, " clear=%b, Q=%3b", clear, Q );
        #12 clear =1;
end
endmodule
```

Result

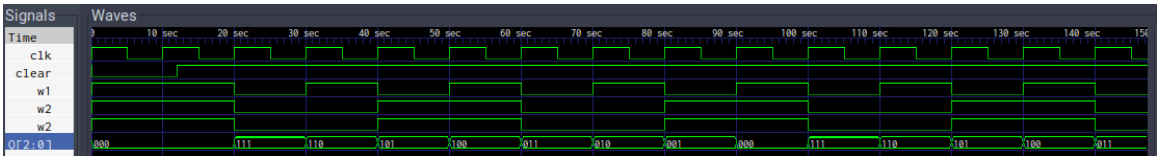


Figure 3.11: Simulation result of Program-5 in Chapter-3 (Synchronous down counter)

Chapter 4

Other features of verilog language

1. Write a verilog code for implementing an 4 bit ALU block using 'functions' which performs the following operations - addition, subtraction, multiplication, exclusive-OR.

```
module alu_function (a,b,sel,z);
input [3:0] a,b;
input [1:0] sel;
output reg [7:0] z;

function [7:0] add;
        input [3:0] a,b;
        begin
                add = a+b;
        end
endfunction

function [7:0] sub;
        input [3:0] a,b;
        begin
                sub = a-b;
        end
endfunction

function [7:0] mul;
        input [3:0] a,b;
        begin
                mul = a*b;
        end
endfunction

function [7:0] exclusive;
        input [3:0] a,b;
        integer i;
        begin
                for (i=0; i<4; i=i+1)
                        exclusive[i]=a[i] ^ b[i];
```

```
                        exclusive[7:4]=4'b0000;
        end
endfunction

always@(a,b,sel)
        case(sel)
                2'b00: z=add(a,b);
                2'b01: z=sub(a,b);
                2'b10: z=mul(a,b);
                2'b11: z=exclusive(a,b);
                default: z=8'b00000000;
        endcase
endmodule
```

Testbench

```
module alu_function_test ;
reg [3:0] a,b;
reg [1:0] sel;
wire [7:0] z;

alu_function G0 (a,b,sel,z);

initial
begin
        a=4'b0000;
        b=4'b0000;
        sel=2'b00;
        #50 $finish;
end

initial
begin
        $dumpfile("alu_function.vcd");
        $dumpvars(0,alu_function_test);
        $monitor($time," a=%4b, b=%4b, sel=%b, z=%8b", a,b,sel,z );
        #5 a=4'b1111;b=4'b1110;
        #10 sel=2'b01;
        #10 sel=2'b10;
        #10 sel=2'b11;
end
endmodule
```

Result

Figure 4.1: Simulation result of Program-1 in Chapter-4 (ALU)

2. Write a verilog code for a implementing $f = \sum m(1,3,4,6)$ using 2:4 decoders with active low outputs. Use tasks for 2:4 decoders and NAND gates.

```
module decoder_ckt (x,y,z,f);
input x,y,z;
output reg f; // task output must be register
reg [3:0] r1,r2; // decoder outputs

// Task definition for decoder
task decoder;
        input p,q,enable;
        output d0,d1,d2,d3;
        begin
                if(enable==0)
                        {d3,d2,d1,d0}=4'b1111;
                else
                begin
                        case({p,q})
                                2'b00: {d3,d2,d1,d0}=4'b1110;
                                2'b01: {d3,d2,d1,d0}=4'b1101;
                                2'b10: {d3,d2,d1,d0}=4'b1011;
                                2'b11: {d3,d2,d1,d0}=4'b0111;
                                default: {d3,d2,d1,d0}=4'b1111;
                        endcase
                end
        end
endtask

// Task definition for 4 input NAND gate
task nand4;
        input w,x,y,z;
        output f;
        begin
                if(!w || !x || !y || !z)
                        f=1;
                else
                        f=0;
        end
endtask

always@(x,y,z)
begin
        // Order of statement is important; executes sequentially
        decoder (y,z,~x,r1[0],r1[1],r1[2],r1[3]);
        decoder (y,z,x, r2[0],r2[1],r2[2],r2[3]);
        nand4 (r1[1],r1[3],r2[0],r2[2],f);
end
endmodule
```

Testbench

```
module decoder_ckt_test ;
reg x,y,z;
wire f;

// Instanstiation
decoder_ckt G0 (x,y,z,f);

initial
begin
        x=0;
        y=0;
        z=0;
        #50 $finish;
end

initial
begin
        $dumpfile("decoder_ckt.vcd");
        $dumpvars(0,decoder_ckt_test);
        $monitor($time," x=%b, y=%b, z=%b, f=%b", x, y, z, f );
        #5 x=1;
        #5 y=1;
        #5 z=1;
        #5 x=0;
        #5 z=0;

end
endmodule
```

Result

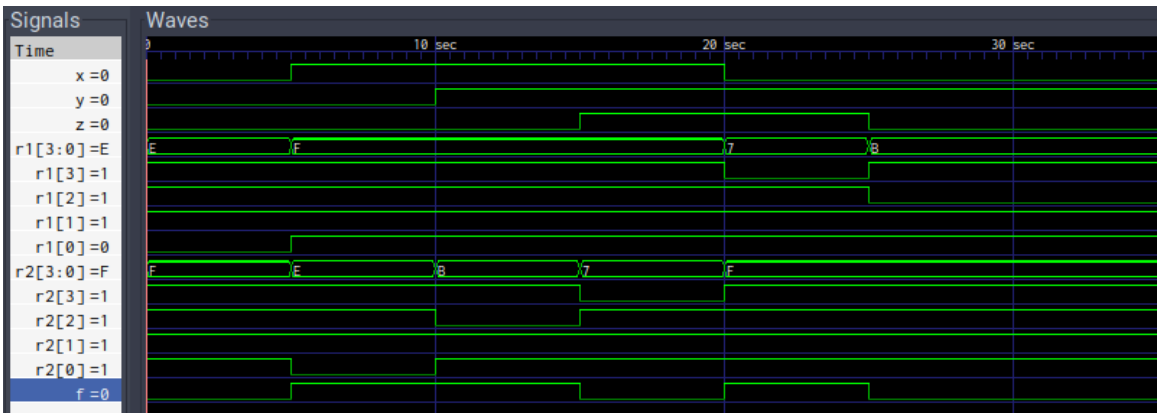


Figure 4.2: Simulation result of Program-2 in Chapter-4 (Decoder based circuit)

3. Write a verilog code for a 8:1 multiplexer implemented using 4:1 and 2:1 multiplexers as UDPs.

```
// UDP for 4:1 Mux
primitive udp_mux4 (f, s1, s0, i0, i1, i2, i3);
        input s1, s0, i0,i1, i2, i3;
        output f;
        table
            // s1  s0  i0  i1  i2  i3   :  f
                0   0   0   ?   ?   ?   :  0;
                0   0   1   ?   ?   ?   :  1;
                0   1   ?   0   ?   ?   :  0;
                0   1   ?   1   ?   ?   :  1;
                1   0   ?   ?   0   ?   :  0;
                1   0   ?   ?   1   ?   :  1;
                1   1   ?   ?   ?   0   :  0;
                1   1   ?   ?   ?   1   :  1;
        endtable
endprimitive

// UDP for 2:1 Mux
primitive udp_mux2 (f, s, i0, i1);
        input s, i0, i1;
        output f;
        table
                // s  i0  i1  :  f
                   0   0   ?  :  0;
                   0   1   ?  :  1;
                   1   ?   0  :  0;
                   1   ?   1  :  1;
           endtable
endprimitive

// Module for 8:1 Mux
module mux_ckt_udp (i, s, f);
input [7:0] i;
input [2:0] s;
output f;
wire w1, w2;

udp_mux4 MUX0 ( w1, s[1], s[0], i[0], i[1], i[2], i[3]);
udp_mux4 MUX1 ( w2, s[1], s[0], i[4], i[5], i[6], i[7]);
udp_mux2 MUX2 ( f,  s[2], w1, w2);
endmodule
```

Testbench

```
module mux_ckt_udp_test ;
reg [7:0] i;
```

```
reg [2:0] s;
integer p;
wire f;

// Instanstiation
mux_ckt_udp G0 (i, s, f);

initial
begin
        i=8'b11010011;
        for (p=0; p<8; p=p+1)
        begin
                s=p; #5;
                $display ("T=%2d, s[2]=%b, s[1]=%b, s[0]=%b, f=%b",
                $time, s[2], s[1], s[0], f);
        end
end

initial
begin
        $dumpfile("mux_ckt_udp.vcd");
        $dumpvars(0,mux_ckt_udp_test);
        #50 $finish;
end
endmodule
```

Result

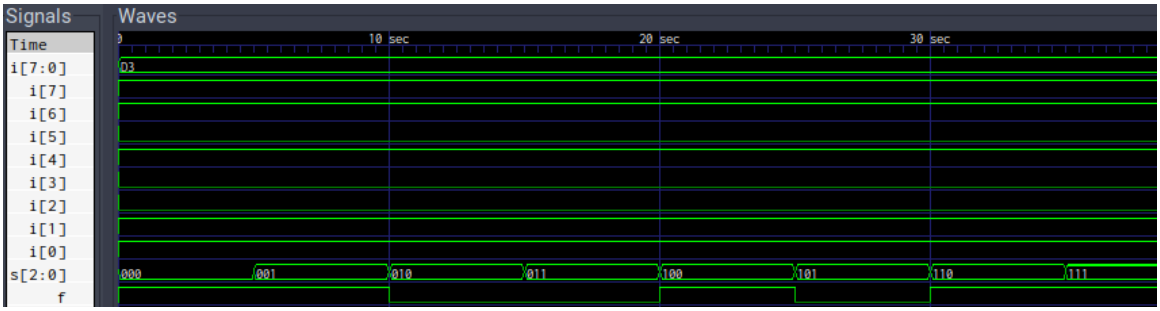


Figure 4.3: Simulation result of Program-3 in Chapter-4 (8:1 Mux using UDPs)

4. Write a verilog code for a 3 bit ripple down counter using positive edge triggered T flip-flops. Use UDP for T flip-flop.

```
// Primitive for +ive edge triggered T flip flop
primitive udp_tff (Q, T, clear, clk);
input T, clear, clk;
output Q;
reg Q;
```

```
initial
        Q=0;

table
        // T    clear    clk    :   Q  :  Qnew
           ?     0       ?      :   ?  :   0; // Clear
           ?    (01)     ?      :   ?  :   -; // Ignore transition of
   clear
           0     1      (01)    :   ?  :   -; // No toggle
           1     1      (01)    :   0  :   1; // Toggle
           1     1      (01)    :   1  :   0;
           ?     1      (?0)    :   ?  :   -; // Ignore -ve edge
           ?     1      (0x)    :   ?  :   -;
           ?     1      (1x)    :   ?  :   -;
          (??)   ?       ?      :   ?  :   -; // Ignore change of T on
   steady clk
   endtable
endprimitive

// Module for 3 bit ripple counter
module counter_udp (clear, clk, Q);
input clear, clk;
output [2:0] Q;

//wire w1, w2, w3;

udp_tff FF0 (Q[0], 1'b1, clear, clk);
udp_tff FF1 (Q[1], 1'b1, clear, Q[0]);
udp_tff FF2 (Q[2], 1'b1, clear, Q[1]);
endmodule
```

Testbench

```
module counter_udp_test ;
reg clear, clk;
wire [2:0] Q;

// Instanstiation
counter_udp G0 (clear, clk, Q);

always #5 clk=~clk;

initial
begin
        clear=0;
        clk=1;
        #150 $finish;
end
```

```
initial
begin
        $dumpfile("counter_udp.vcd");
        $dumpvars(0,counter_udp_test);
        $monitor($time, " clear=%b, Q=%3b", clear, Q );
        #12 clear =1;
end
endmodule
```

Result

Figure 4.4: Simulation result of Program-4 in Chapter-4 (Ripple counter using UDPs)

Chapter 5

Switch level modeling

1. Write a switch level verilog code for a CMOS inverter.

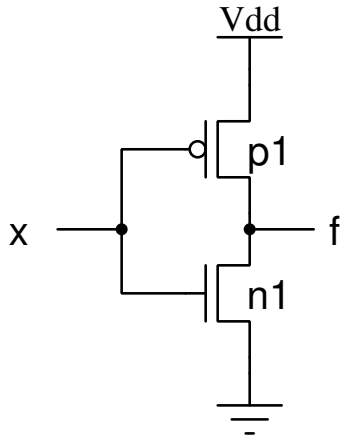


Figure 5.1: Transistor level circuit diagram of inverter

```
module inverter_switch (x,f);
input x;
output f;
supply1 vdd;
supply0 gnd;

pmos p1 (f, vdd, x);
nmos n1 (f, gnd, x);

endmodule
```

Testbench

```
module inverter_switch_test ;
reg x;
wire f;

// Instanstiation
inverter_switch G0 (x,f);
```

```
initial
begin
        x=0;
        #30 $finish;
end

initial
begin
        $dumpfile("inverter_switch.vcd");
        $dumpvars(0,inverter_switch_test);
        $monitor($time,"x = %b, f = %b",x,f );
        #5 x=1;
        #10 x=0;
        #5 x=1;
end
endmodule
```

Result

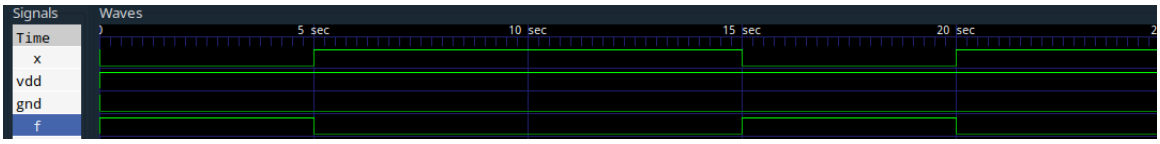


Figure 5.2: Simulation result of Program-1 in Chapter-5 (Switch level NOT)

2. Write a switch level verilog code for a two input NAND gate.

```
module nand_switch (x,y,f);
input x,y;
output f;
supply1 vdd;
supply0 gnd;
wire a;

pmos p1 (f, vdd, x);
pmos p2 (f, vdd, y);
nmos n1 (f, a, x);
nmos n2 (a, gnd, y);
endmodule
```

Testbench

```
module nand_switch_test ;
reg x,y;
wire f;
integer i;

nand_switch G0 (x,y,f);
```

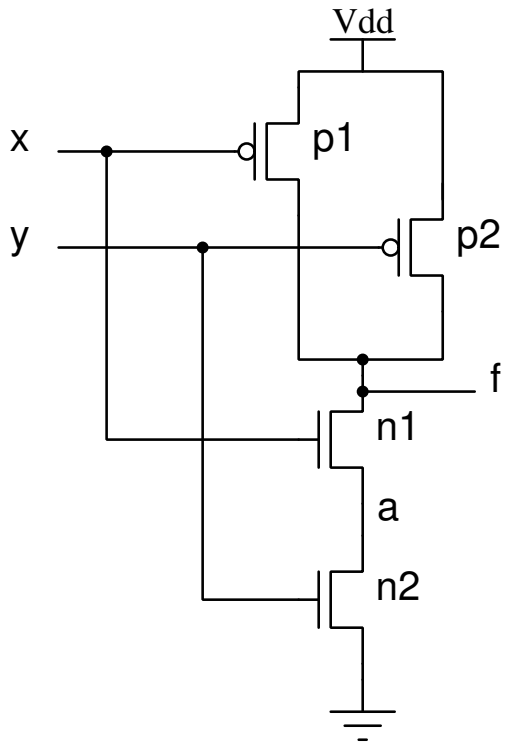


Figure 5.3: Transistor level circuit diagram of NAND gate

```
initial
begin
        x=0; y=0;
        #20 $finish;
end

initial
begin
        $dumpfile("nand_switch.vcd");
        $dumpvars(0,nand_switch_test);
        $monitor($time, " x = %b, y = %b, f = %b",x,y,f );
        for (i=0; i<4; i=i+1)
        begin
                        {x,y}=i;
                        #5;
        end
end
endmodule
```

Result

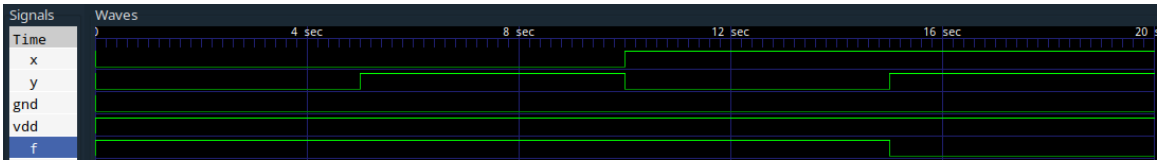


Figure 5.4: Simulation result of Program-2 in Chapter-5 (Switch level NAND)

2. Write a switch level verilog code for a two input NOR gate.

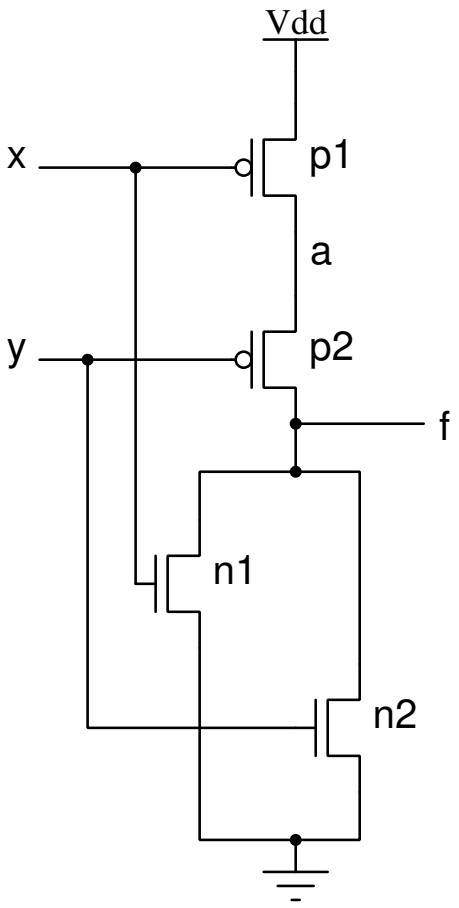


Figure 5.5: Transistor level circuit diagram of NOR gate

```
module nor_switch (x,y,f);
input x,y;
output f;
supply1 vdd;
supply0 gnd;
wire a;

pmos p1 (a, vdd, x);
pmos p2 (f, a, y);
nmos n1 (f, gnd, x);
nmos n2 (f, gnd, y);

endmodule
```

Testbench

```
module nor_switch_test ;
reg x,y;
wire f;
integer i;
```

```
// Instanstiation
nor_switch G0 (x,y,f);

initial
begin
        x=0; y=0;
        #20 $finish;
end

initial
begin
        $dumpfile("nor_switch.vcd");
        $dumpvars(0,nor_switch_test);
        $monitor($time, " x = %b, y = %b, f = %b",x,y,f );
        for (i=0; i<4; i=i+1)
        begin
                {x,y}=i;
                #5;
        end
end
endmodule
```

Result

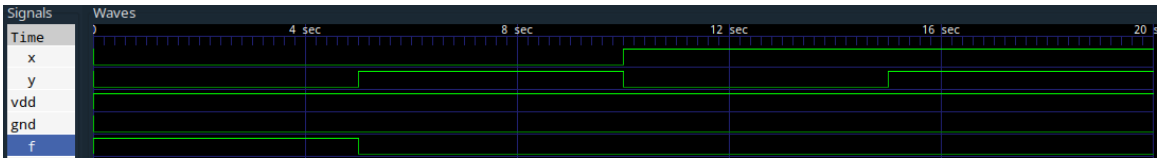


Figure 5.6: Simulation result of Program-3 in Chapter-5 (Switch level NOR)

Chapter 6

Finite State Machine modeling

1. Write a verilog code for a FSM which is defined as following - The has three lamps, RED, GREEN and YELLOW, that should glow cyclically with a fixed interval (say, 1 second).

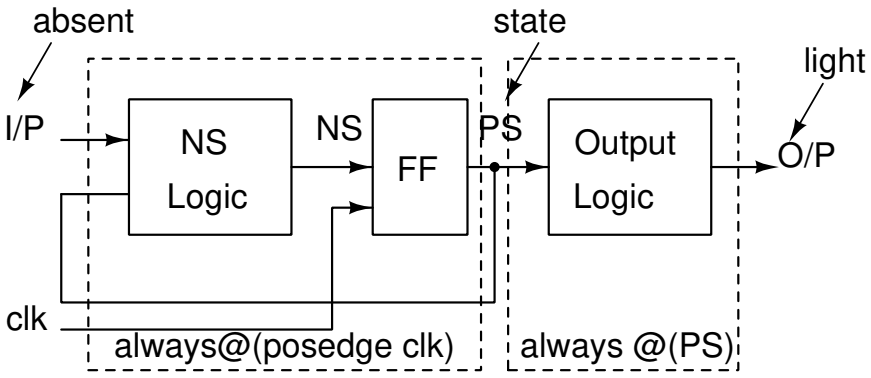


Figure 6.1: Structure of model used for FSM-1

```
module FSM_1 (clk,light);
input clk ;
output reg [2:0] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
reg [1:0] state;

// It is a Moore model but input component is
// absent. Refer Moore Block diagram.
always @(posedge clk) // Always block for NS logic and FF
        case(state)
                S0: state <= S1;
                S1: state <= S2;
                S2: state <= S0;
                default: state <= S0;
        endcase

always @ (state) // Always block for output logic
```

```
        case(state)
                S0: light = RED;
                S1: light = GREEN;
                S2: light = YELLOW;
                default: light = RED;
        endcase
endmodule
```

Testbench

```
module FSM_1_test ;
reg clk;
wire [2:0] light;

// Instanstiation
FSM_1 G0 (clk, light);

always #5 clk=~clk;

initial
begin
        clk=0;
        #50 $finish;
end

initial
begin
        $dumpfile("FSM_1.vcd");
        $dumpvars(0,FSM_1_test);
        $monitor($time," LIGHT =%3b",light );
end
endmodule
```

Result

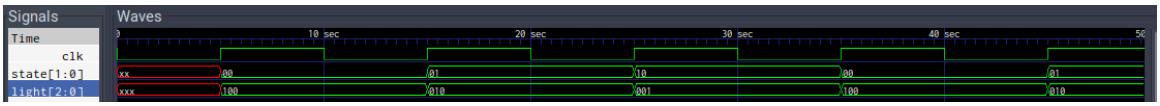


Figure 6.2: Simulation result of Program-1 in Chapter-6 (LED glowing)

2. Write a verilog code for a synchronous binary parity checker which produces output high whenever it receives even number of 1s in binary input stream.

```
module FSM_2 (x, clk, z);
input x, clk ;
output reg z;
parameter EVEN=0, ODD=1;
reg state;
```

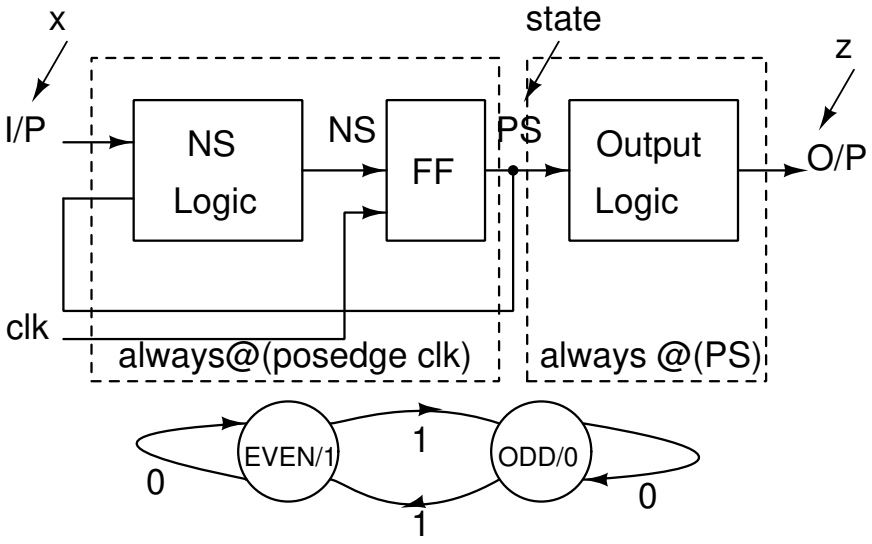


Figure 6.3: Structure of model used for FSM-2

```
// It is a Moore model.
always @(posedge clk) // Always block for NS logic and FF
        case(state)
                EVEN: state <= x ? ODD : EVEN;
                ODD:  state <= x ? EVEN: ODD;
                default: state <= EVEN;
        endcase

always @ (state) // Always block for output logic
        case(state)
                ODD: z = 0;
                EVEN: z = 1;
                default: z = 1;
        endcase
endmodule
```

Testbench

```
module FSM_2_test ;
reg x, clk;
wire z;

//Instanstiation
FSM_2 G0 (x, clk, z);

always #5 clk=~clk;

initial
begin
        clk=0;
        x=0;
        #70 $finish;
```

```
end

initial
begin
        $dumpfile("FSM_2.vcd");
        $dumpvars(0,FSM_2_test);
        $monitor($time," x = %b, z = %b ",x,z );
        #12 x=1;
        #10 x=0;
        #10 x=0;
        #10 x=1;
end
endmodule
```

Result

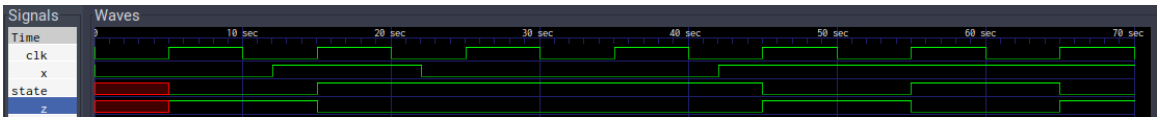


Figure 6.4: Simulation result of Program-2 in Chapter-6 (Serial parity checker)

3. Write a verilog code for a synchronous overlapping sequence detector which detects '0110' in a binary data.

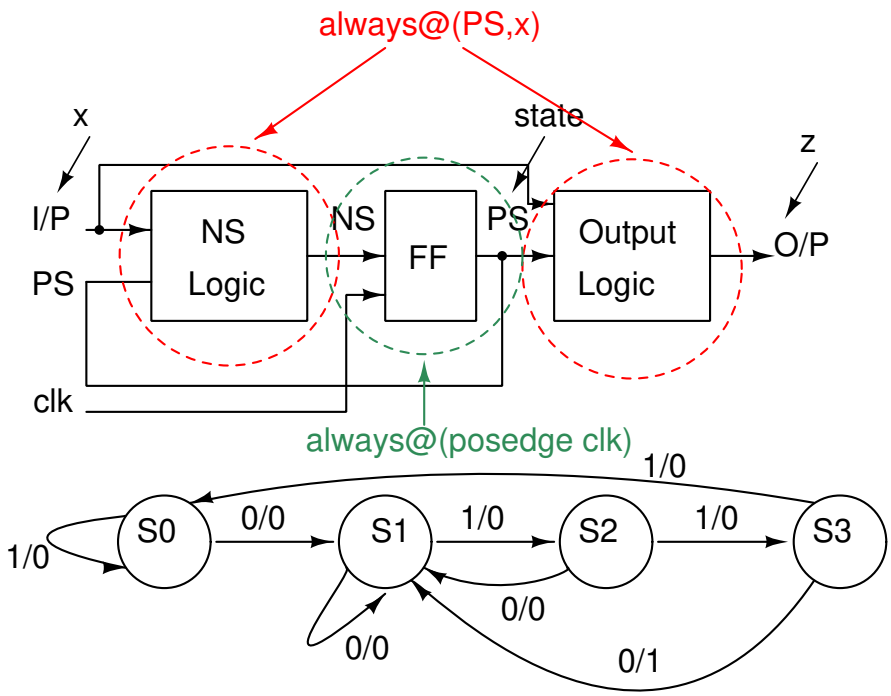


Figure 6.5: Structure of model used for FSM-3

```
module FSM_3 (x, clk, z);
input x, clk ;
```

```
output reg z;
parameter S0=0, S1=1, S2=2, S3=3;
reg [1:0] PS, NS;

// It is a Mealy model.
always @(posedge clk) // Always block for FF
        PS <= NS;

always @(PS, x) // Always block for combinational block
        case(PS)
                S0:
                begin
                        NS= x ? S0 : S1;
                        z = x ? 0 : 0;
                end
                S1:
                begin
                    NS = x ? S2 : S1;
                        z = x ? 0 : 0;
                end
                S2:
                begin
                        NS = x ? S3 : S1;
                        z = x ? 0 : 0;
                end
                S3:
                begin
                        NS = x ? S0 : S1;
                        z = x ? 0: 1;
                end
                default:
                begin
                        NS = S0;
                        z = 0;
                end
        endcase
endmodule
```

Testbench

```
module FSM_3_test ;
reg x, clk;
wire z;

// Instanstiation
FSM_3 G0 (x, clk, z);

always #5 clk=~clk;
```

```
initial
begin
        clk=0;
        x=0;
        #150 $finish;
end

initial
begin
        $dumpfile("FSM_3.vcd");
        $dumpvars(0,FSM_3_test);
        $monitor($time," x = %b, z = %b ",x,z );
        #12 x=0; #10 x=0; #10 x=1; #10 x=1;
        #10 x=0; #10 x=1; #10 x=1; #10 x=0;
        #10 x=1; #10 x=0;
end
endmodule
```

Result

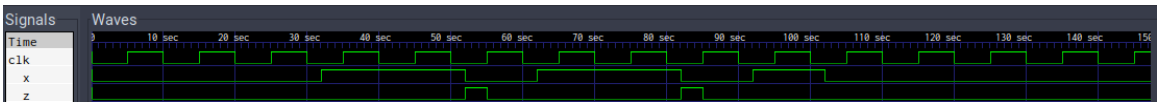


Figure 6.6: Simulation result of Program-3 in Chapter-6 (Sequence detector)

References

[1] NPTEL video lecture series by Prof. Indranil Sengupta, IIT Kharagpur, `https://nptel.ac.in/courses/106/105/106105165/#`

[2] Samir Palnitkar, "Verilog HDL: A Guide to Digital Design and Synthesis", Prentice Hall PTR, 2003.

[3] *Icarus Verilog*, `http://iverilog.icarus.com/home`

[4] *GTKwave*, `http://gtkwave.sourceforge.net/`