VERILOG CODING EXAMPLES

Jairaj Mirashi
Design Verification Engineer

Examples with code

2.3.1 Following is the Verilog code for flip-flop with a positive-edge clock.

```
module flop (clk, d, q);
input clk, d;
output q;
reg q;

always @(posedge clk)
begin
q <= d;
end
endmodule
```

2.3.2 Following is Verilog code for a flip-flop with a negative-edge clock and

asynchronous
clear.

```
module flop (clk, d, clr, q);
input clk, d, clr;
output q;
reg q;
always @(negedge clk or posedge clr)
begin
if (clr)
q <= 1'b0;
else
q <= d;
end
endmodule
```

2.3.3 Following is Verilog code for the flip-flop with a positive-edge clock and synchronous set.

```
module flop (clk, d, s, q);
input clk, d, s;
output q;
reg q;
always @(posedge clk)
begin
if (s)
q <= 1'b1;
else
q <= d;
end
endmodule
```

2.3.4 Following is Verilog code for the flip-flop with a positive-edge clock and clock enable.

```
module flop (clk, d, ce, q);
input clk, d, ce;
output q;
reg q;
always @(posedge clk)
begin
if (ce)
q <= d;
end
endmodule
```

2.3.5 Following is Verilog code for a 4-bit register with a positive-edge clock,

asynchronous set
and clock enable.

```
module flop (clk, d, ce, pre, q);
input clk, ce, pre;
input [3:0] d;
output [3:0] q;
reg [3:0] q;
always @(posedge clk or posedge pre)
begin
if (pre)
q <= 4'b1111;
else if (ce)
q <= d;
end
endmodule
```

2.3.6 Following is the Verilog code for a latch with a positive gate.

```
module latch (g, d, q);
input g, d;
output q;
reg q;
always @(g or d)
begin
if (g)
q <= d;
end
endmodule
```

2.3.7 Following is the Verilog code for a latch with a positive gate and an asynchronous clear.

```
module latch (g, d, clr, q);
input g, d, clr;
output q;
reg q;
always @(g or d or clr)
begin
if (clr)
q <= 1'b0;
else if (g)
q <= d;
end
endmodule
```

2.3.8 Following is Verilog code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
module latch (g, d, pre, q);
input g, pre;
input [3:0] d;
output [3:0] q;
reg [3:0] q;
always @(g or d or pre)
begin
if (pre)
q <= 4'b1111;
else if (~g)
q <= d;
end
endmodule
```

2.3.9 Following is Verilog code for a tristate element using a combinatorial process and always block.

```
module three_st (t, i, o);
input t, i;
output o;
reg o;
always @(t or i)
begin
if (~t)
o = i;
else
o = 1'bZ;
end
endmodule
```

2.3.10 Following is the Verilog code for a tristate element using a concurrent assignment.

```
module three_st (t, i, o);
input t, i;
output o;
assign o = (~t) ? i: 1'bZ;
endmodule
```

2.3.11 Following is the Verilog code for a 4-bit unsigned up counter with asynchronous clear.

```
module counter (clk, clr, q);
input clk, clr;
output [3:0] q;
reg [3:0] tmp;
```

```
 always @(posedge clk or posedge clr)
begin
if (clr)
tmp <= 4'b0000;
else
tmp <= tmp + 1'b1;
end
assign q = tmp;
endmodule
```

2.3.12 Following is the Verilog code for a 4-bit unsigned down counter with synchronous set.

```
module counter (clk, s, q);
input clk, s;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk)
begin
if (s)
tmp <= 4'b1111;
else
tmp <= tmp - 1'b1;
end
assign q = tmp;
endmodule
```

2.3.13 Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

```
module counter (clk, load, d, q);
input clk, load;
```

```
 input [3:0] d;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk or posedge load)
begin
if (load)
tmp <= d;
else
tmp <= tmp + 1'b1;
end
assign q = tmp;
endmodule
```

2.3.14 Following is the Verilog code for a 4-bit unsigned up counter with a synchronous load with a constant.

```
module counter (clk, sload, q);
input clk, sload;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk)
begin
if (sload)
tmp <= 4'b1010;
else
tmp <= tmp + 1'b1;
end
assign q = tmp;
endmodule
```

2.3.15 Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

```
module counter (clk, clr, ce, q);
input clk, clr, ce;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk or posedge clr)
begin
if (clr)
tmp <= 4'b0000;
else if (ce)
tmp <= tmp + 1'b1;
end
assign q = tmp;
endmodule
```

2.3.16 Following is the Verilog code for a 4-bit unsigned up/down counter with an asynchronous clear.

```
module counter (clk, clr, up_down, q);
input clk, clr, up_down;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk or posedge clr)
begin
if (clr)
tmp <= 4'b0000;
else if (up_down)
tmp <= tmp + 1'b1;
else
tmp <= tmp - 1'b1;
end
assign q = tmp;
endmodule
```

2.3.17 Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset.

```
module counter (clk, clr, q);
input clk, clr;
output signed [3:0] q;
reg signed [3:0] tmp;
always @ (posedge clk or posedge clr)
begin
if (clr)
tmp <= 4'b0000;
else
tmp <= tmp + 1'b1;
end
assign q = tmp;
endmodule
```

2.3.18 Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

```
module counter (clk, clr, q);
parameter MAX_SQRT = 4, MAX = (MAX_SQRT*MAX_SQRT);
input clk, clr;
output [MAX_SQRT-1:0] q;
reg [MAX_SQRT-1:0] cnt;
always @ (posedge clk or posedge clr)
begin
if (clr)
cnt <= 0;
else
```

```
 cnt <= (cnt + 1) %MAX;
end
assign q = cnt;
endmodule
```

2.3.19 Following is the Verilog code for a 4-bit unsigned up accumulator with an asynchronous clear.

```
module accum (clk, clr, d, q);
input clk, clr;
input [3:0] d;
output [3:0] q;
reg [3:0] tmp;
always @(posedge clk or posedge clr)
begin
if (clr)
tmp <= 4'b0000;
else
tmp <= tmp + d;
end
assign q = tmp;
endmodule
```

2.3.20 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, serial in and serial out.

```
module shift (clk, si, so);
input clk,si;
output so;
reg [7:0] tmp;
always @(posedge clk)
begin
```

```
 tmp <= tmp << 1;
tmp[0] <= si;
end
assign so = tmp[7];
endmodule
```

2.3.21 Following is the Verilog code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
module shift (clk, ce, si, so);
input clk, si, ce;
output so;
reg [7:0] tmp;
always @(negedge clk)
begin
if (ce) begin
tmp <= tmp << 1;
tmp[0] <= si;
end
end
assign so = tmp[7];
endmodule
```

2.3.22 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, asynchronous clear, serial in and serial out.

```
module shift (clk, clr, si, so);
input clk, si, clr;
output so;
reg [7:0] tmp;
always @(posedge clk or posedge clr)
begin
```

```
 if (clr)
tmp <= 8’b00000000;
else
tmp <= {tmp[6:0], si};
end
assign so = tmp[7];
endmodule
```

2.3.23 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

```
module shift (clk, s, si, so);
input clk, si, s;
output so;
reg [7:0] tmp;
always @(posedge clk)
begin
if (s)
tmp <= 8’b11111111;
else
tmp <= {tmp[6:0], si};
end
assign so = tmp[7];
endmodule
```

2.3.24 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
module shift (clk, si, po);
input clk, si;
output [7:0] po;
reg [7:0] tmp;
```

```
 always @(posedge clk)
begin
tmp <= {tmp[6:0], si};
end
assign po = tmp;
endmodule
```

2.3.25 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
module shift (clk, load, si, d, so);
input clk, si, load;
input [7:0] d;
output so;
reg [7:0] tmp;
always @(posedge clk or posedge load)
begin
if (load)
tmp <= d;
else
tmp <= {tmp[6:0], si};
end
assign so = tmp[7];
endmodule
```

2.3.26 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

```
module shift (clk, sload, si, d, so);
input clk, si, sload;
input [7:0] d;
output so;
```

```
 reg [7:0] tmp;
always @(posedge clk)
begin
if (sload)
tmp <= d;
else
tmp <= {tmp[6:0], si};
end
assign so = tmp[7];
endmodule
```

2.3.27 Following is the Verilog code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
module shift (clk, si, left_right, po);
input clk, si, left_right;
output po;
reg [7:0] tmp;
always @(posedge clk)
begin
if (left_right == 1'b0)
tmp <= {tmp[6:0], si};
else
tmp <= {si, tmp[7:1]};
end
assign po = tmp;
endmodule
```

2.3.28 Following is the Verilog code for a 4-to-1 1-bit MUX using an If statement.

```
module mux (a, b, c, d, s, o);
```

```
 input a,b,c,d;
input [1:0] s;
output o;
reg o;
always @(a or b or c or d or s)
begin
if (s == 2'b00)
o = a;
else if (s == 2'b01)
o = b;
else if (s == 2'b10)
o = c;
else
o = d;
end
endmodule
```

2.3.29 Following is the Verilog Code for a 4-to-1 1-bit MUX using a Case statement.

```
module mux (a, b, c, d, s, o);
input a, b, c, d;
input [1:0] s;
output o;
reg o;
always @(a or b or c or d or s)
begin
case (s)
2'b00 : o = a;
2'b01 : o = b;
2'b10 : o = c;
default : o = d;
endcase
```

```
 end
endmodule
```

2.3.30 Following is the Verilog code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
module mux (a, b, c, d, s, o);
input a, b, c, d;
input [1:0] s;
output o;
reg o;
always @(a or b or c or d or s)
begin
if (s == 2’b00)
o = a;
else if (s == 2’b01)
o = b;
else if (s == 2’b10)
o = c;
end
endmodule
```

2.3.31 Following is the Verilog code for a 1-of-8 decoder.

```
module mux (sel, res);
input [2:0] sel;
output [7:0] res;
reg [7:0] res;
always @(sel or res)
begin
case (sel)
3’b000 : res = 8’b00000001;
```

```
 3’b001 : res = 8’b00000010;
3’b010 : res = 8’b00000100;
3’b011 : res = 8’b00001000;
3’b100 : res = 8’b00010000;
3’b101 : res = 8’b00100000;
3’b110 : res = 8’b01000000;
default : res = 8’b10000000;
endcase
end
endmodule
```

2.3.32 Following Verilog code leads to the inference of a 1-of-8 decoder.

```
module mux (sel, res);
input [2:0] sel;
output [7:0] res;
reg [7:0] res;
always @(sel or res) begin
case (sel)
3’b000 : res = 8’b00000001;
3’b001 : res = 8’b00000010;
3’b010 : res = 8’b00000100;
3’b011 : res = 8’b00001000;
3’b100 : res = 8’b00010000;
3’b101 : res = 8’b00100000;
// 110 and 111 selector values are unused
default : res = 8’bxxxxxxxx;
endcase
end
endmodule
```

2.3.33 Following is the Verilog code for a 3-bit 1-of-9 Priority Encoder.

```
module priority (sel, code);
input [7:0] sel;
output [2:0] code;
reg [2:0] code;
always @(sel)
begin
if (sel[0])
code = 3’b000;
else if (sel[1])
code = 3’b001;
else if (sel[2])
code = 3’b010;
else if (sel[3])
code = 3’b011;
else if (sel[4])
code = 3’b100;
else if (sel[5])
code = 3’b101;
else if (sel[6])
code = 3’b110;
else if (sel[7])
code = 3’b111;
else
code = 3’bxxx;
end
endmodule
```

2.3.34 Following is the Verilog code for a logical shifter.

```
module lshift (di, sel, so);
```

```
 input [7:0] di;
input [1:0] sel;
output [7:0] so;
reg [7:0] so;
always @(di or sel)
begin
case (sel)
2'b00 : so = di;
2'b01 : so = di << 1;
2'b10 : so = di << 2;
default : so = di << 3;
endcase
end
endmodule
```

2.3.35 Following is the Verilog code for an unsigned 8-bit adder with carry in.

```
module adder(a, b, ci, sum);
input [7:0] a;
input [7:0] b;
input ci;
output [7:0] sum;

assign sum = a + b + ci;

endmodule
```

2.3.36 Following is the Verilog code for an unsigned 8-bit adder with carry out.

```
module adder(a, b, sum, co);
input [7:0] a;
input [7:0] b;
output [7:0] sum;
```

```
 output co;
wire [8:0] tmp;

assign tmp = a + b;
assign sum = tmp [7:0];
assign co = tmp [8];

endmodule
```

2.3.37 Following is the Verilog code for an unsigned 8-bit adder with carry in and carry out.

```
module adder(a, b, ci, sum, co);
input ci;
input [7:0] a;
input [7:0] b;
output [7:0] sum;
output co;
wire [8:0] tmp;

assign tmp = a + b + ci;
assign sum = tmp [7:0];
assign co = tmp [8];

endmodule
```

2.3.38 Following is the Verilog code for an unsigned 8-bit adder/subtractor.

```
module addsub(a, b, oper, res);
input oper;
input [7:0] a;
input [7:0] b;
```

```
 output [7:0] res;
reg [7:0] res;
always @(a or b or oper)
begin
if (oper == 1'b0)
res = a + b;
else
res = a - b;
end
endmodule
```

2.3.39 Following is the Verilog code for an unsigned 8-bit greater or equal comparator.

```
module compar(a, b, cmp);
input [7:0] a;
input [7:0] b;
output cmp;

assign cmp = (a >= b) ? 1'b1 : 1'b0;

endmodule
```

2.3.40 Following is the Verilog code for an unsigned 8x4-bit multiplier.

```
module compar(a, b, res);
input [7:0] a;
input [3:0] b;
output [11:0] res;

assign res = a * b;
```

```
endmodule
```

2.3.41 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input clk;
input [17:0] a;
input [17:0] b;
output [35:0] mult;
reg [35:0] mult;
reg [17:0] a_in, b_in;
wire [35:0] mult_res;
reg [35:0] pipe_1, pipe_2, pipe_3;

assign mult_res = a_in * b_in;

always @(posedge clk)
begin
a_in <= a;
b_in <= b;
pipe_1 <= mult_res;
pipe_2 <= pipe_1;
pipe_3 <= pipe_2;
mult <= pipe_3;
end
endmodule
```

2.3.42 Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
```

```
 input clk;
input [17:0] a;
input [17:0] b;
output [35:0] mult;
reg [35:0] mult;
        reg [17:0] a_in, b_in;
         reg [35:0] mult_res;
       reg [35:0] pipe_2, pipe_3;
      always @(posedge clk)
begin
a_in <= a;
b_in <= b;
mult_res <= a_in * b_in;
pipe_2 <= mult_res;
          pipe_3 <= pipe_2;
            mult <= pipe_3;
end
endmodule
```

2.3.43 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input clk;
input [17:0] a;
input [17:0] b;
output [35:0] mult;
reg [35:0] mult;
reg [17:0] a_in, b_in;
wire [35:0] mult_res;
reg [35:0] pipe_1, pipe_2, pipe_3;

assign mult_res = a_in * b_in;
```

```
always @(posedge clk)
begin
a_in <= a;
b_in <= b;
pipe_1 <= mult_res;
pipe_2 <= pipe_1;
pipe_3 <= pipe_2;
mult <= pipe_3;
end
endmodule
```

2.3.44 Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
input clk;
input [17:0] a;
input [17:0] b;
output [35:0] mult;
reg [35:0] mult;
reg [17:0] a_in, b_in;
reg [35:0] mult_res;
reg [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
a_in <= a;
b_in <= b;
mult_res <= a_in * b_in;
pipe_2 <= mult_res;
pipe_3 <= pipe_2;
mult <= pipe_3;
end
```

```
endmodule
```

2.3.45 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as shift registers.

```
module mult3(clk, a, b, mult);
input clk;
input [17:0] a;
input [17:0] b;
output [35:0] mult;
reg [35:0] mult;
reg [17:0] a_in, b_in;
wire [35:0] mult_res;
reg [35:0] pipe_regs [3:0];

assign mult_res = a_in * b_in;

always @(posedge clk)
begin
a_in <= a;
b_in <= b;
{pipe_regs[3],pipe_regs[2],pipe_regs[1],pipe_regs[0]} <=
{mult, pipe_regs[3],pipe_regs[2],pipe_regs[1]};
end
endmodule
```

2.3.46 Following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in Verilog.

```
module mvl_multaddsub1(clk, a, b, c, res);
input clk;
input [07:0] a;
```

```
 input [07:0] b;
input [07:0] c;
output [15:0] res;
reg [07:0] a_reg1, a_reg2, b_reg1, b_reg2;
wire [15:0] multaddsub;
always @(posedge clk)
begin
a_reg1 <= a;
a_reg2 <= a_reg1;
b_reg1 <= b;
b_reg2 <= b_reg1;
end
        assign multaddsub = a_reg2 * b_reg2 + c;
assign res = multaddsub;
endmodule
```

2.3.47 Following is the Verilog code for resource sharing.

```
module addsub(a, b, c, oper, res);
input oper;
input [7:0] a;
input [7:0] b;
input [7:0] c;
output [7:0] res;
reg [7:0] res;
always @(a or b or c or oper)
begin
if (oper == 1'b0)
res = a + b;
else
res = a - c;
end
```

```
endmodule
```

2.3.48 Following templates show a single-port RAM in read-first mode.

```
module raminfr (clk, en, we, addr, di, do);
input clk;
input we;
input en;
input [4:0] addr;
input [3:0] di;
output [3:0] do;
reg [3:0] RAM [31:0];
reg [3:0] do;
always @(posedge clk)
begin
if (en)
begin
if (we)
RAM[addr] <= di;

do <= RAM[addr];
end
end
endmodule
```

2.3.49 Following templates show a single-port RAM in write-first mode.

```
module raminfr (clk, we, en, addr, di, do);
input clk;
input we;
input en;
```

```
 input [4:0] addr;
input [3:0] di;
output [3:0] do;
reg [3:0] RAM [31:0];
reg [4:0] read_addr;
always @(posedge clk)
begin
if (en) begin
if (we)
RAM[addr] <= di;
read_addr <= addr;
end
end
assign do = RAM[read_addr];
endmodule
```

2.3.50 Following templates show a single-port RAM in no-change mode.

```
module raminfr (clk, we, en, addr, di, do);
input clk;
input we;
input en;
input [4:0] addr;
input [3:0] di;
output [3:0] do;
reg [3:0] RAM [31:0];
reg [3:0] do;
always @(posedge clk)
begin
if (en) begin
if (we)
RAM[addr] <= di;
```

```
 else
do <= RAM[addr];
end
end
endmodule
```

2.3.51 Following is the Verilog code for a single-port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;
reg [3:0] ram [31:0];
always @(posedge clk)
begin
if (we)
ram[a] <= di;
end
assign do = ram[a];
endmodule
```

2.3.52 Following is the Verilog code for a single-port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;
```

```
 reg [3:0] ram [31:0];
reg [3:0] do;
always @(posedge clk)
begin
if (we)
ram[a] <= di;
do <= ram[a];
end
endmodule
```

2.3.53 Following is the Verilog code for a single-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, di, do);
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;
reg [3:0] ram [31:0];
reg [4:0] read_a;
always @(posedge clk)
begin
if (we)
ram[a] <= di;
read_a <= a;
end
assign do = ram[read_a];
endmodule
```

2.3.54 Following is the Verilog code for a single-port block RAM with enable.

```
module raminfr (clk, en, we, a, di, do);
input clk;
input en;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;
reg [3:0] ram [31:0];
reg [4:0] read_a;
always @(posedge clk)
begin
if (en) begin
if (we)
ram[a] <= di;
read_a <= a;
end
end
assign do = ram[read_a];
endmodule
```

2.3.55 Following is the Verilog code for a dual-port RAM with asynchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input clk;
input we;
input [4:0] a;
input [4:0] dpra;
input [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg [3:0] ram [31:0];
```

```
 always @(posedge clk)
begin
if (we)
ram[a] <= di;
end
assign spo = ram[a];
assign dpo = ram[dpra];
endmodule
```

2.3.56 Following is the Verilog code for a dual-port RAM with false synchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input clk;
input we;
input [4:0] a;
input [4:0] dpra;
input [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg [3:0] ram [31:0];
reg [3:0] spo;
reg [3:0] dpo;
always @(posedge clk)
begin
if (we)
ram[a] <= di;

spo = ram[a];
dpo = ram[dpra];
end
endmodule
```

2.3.57 Following is the Verilog code for a dual-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input clk;
input we;
input [4:0] a;
input [4:0] dpra;
input [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg [3:0] ram [31:0];
reg [4:0] read_a;
reg [4:0] read_dpra;
always @(posedge clk)
begin
if (we)
ram[a] <= di;
read_a <= a;
read_dpra <= dpra;
end
assign spo = ram[read_a];
assign dpo = ram[read_dpra];
endmodule
```

2.3.58 Following is the Verilog code for a dual-port RAM with enable on each port.

```
module raminfr (clk, ena, enb, wea, addra, addrb, dia, doa, dob);
input clk, ena, enb, wea;
input [4:0] addra, addrb;
input [3:0] dia;
output [3:0] doa, dob;
```

```
 reg [3:0] ram [31:0];
reg [4:0] read_addra, read_addrb;
always @(posedge clk)
begin
if (ena) begin
if (wea) begin
ram[addra] <= dia;
end
end
end

always @(posedge clk)
begin
if (enb) begin
read_addrb <= addrb;
end
end
assign doa = ram[read_addra];
assign dob = ram[read_addrb];
endmodule
```

2.3.59 Following is Verilog code for a ROM with registered output.

```
module rominfr (clk, en, addr, data);
input clk;
input en;
input [4:0] addr;
output reg [3:0] data;
always @(posedge clk)
begin
if (en)
case(addr)
```

```
 4’b0000: data <= 4’b0010;
4’b0001: data <= 4’b0010;
4’b0010: data <= 4’b1110;
4’b0011: data <= 4’b0010;
4’b0100: data <= 4’b0100;
4’b0101: data <= 4’b1010;
4’b0110: data <= 4’b1100;
4’b0111: data <= 4’b0000;
4’b1000: data <= 4’b1010;
4’b1001: data <= 4’b0010;
4’b1010: data <= 4’b1110;
4’b1011: data <= 4’b0010;
4’b1100: data <= 4’b0100;
4’b1101: data <= 4’b1010;
4’b1110: data <= 4’b1100;
4’b1111: data <= 4’b0000;
default: data <= 4’bXXXX;
endcase
end
endmodule
```

2.3.60 Following is Verilog code for a ROM with registered address.

```
module rominfr (clk, en, addr, data);
input clk;
input en;
input [4:0] addr;
output reg [3:0] data;
reg [4:0] raddr;
always @(posedge clk)
begin
if (en)
```

```
 raddr <= addr;
end

always @(raddr)
begin
if (en)
case(raddr)
4'b0000: data = 4'b0010;
4'b0001: data = 4'b0010;
4'b0010: data = 4'b1110;
4'b0011: data = 4'b0010;
4'b0100: data = 4'b0100;
4'b0101: data = 4'b1010;
4'b0110: data = 4'b1100;
4'b0111: data = 4'b0000;
4'b1000: data = 4'b1010;
4'b1001: data = 4'b0010;
4'b1010: data = 4'b1110;
4'b1011: data = 4'b0010;
4'b1100: data = 4'b0100;
4'b1101: data = 4'b1010;
4'b1110: data = 4'b1100;
4'b1111: data = 4'b0000;
default: data = 4'bXXXX;
endcase
end
endmodule
```

2.3.61 Following is the Verilog code for an FSM with a single process.

```
module fsm (clk, reset, x1, outp);
input clk, reset, x1;
```

```
 output outp;
reg outp;
reg [1:0] state;
     parameter s1 = 2’b00; parameter s2 = 2’b01;
    parameter s3 = 2’b10; parameter s4 = 2’b11;
always @(posedge clk or posedge reset)
begin
if (reset) begin
state <= s1; outp <= 1’b1;
end
else begin
case (state)
s1: begin
if (x1 == 1’b1) begin
state <= s2;
outp <= 1’b1;
end
else begin
state <= s3;
outp <= 1’b1;
end
end
s2: begin
state <= s4;
outp <= 1’b0;
end
s3: begin
state <= s4;
outp <= 1’b0;
end
s4: begin
state <= s1;
outp <= 1’b1;
end
```

```
 endcase
end
end
endmodule
```

2.3.62 Following is the Verilog code for an FSM with two processes.

```
module fsm (clk, reset, x1, outp);
input clk, reset, x1;
output outp;
reg outp;
reg [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
if (reset)
state <= s1;
else begin
case (state)
s1: if (x1 == 1'b1)
state <= s2;
else
state <= s3;
s2: state <= s4;
s3: state <= s4;
s4: state <= s1;
endcase
end
end
always @(state) begin
case (state)
```

```
 s1: outp = 1'b1;
s2: outp = 1'b1;
s3: outp = 1'b0;
s4: outp = 1'b0;
endcase
end
endmodule
```

2.3.63 Following is the Verilog code for an FSM with three processes.

```
module fsm (clk, reset, x1, outp);
input clk, reset, x1;
output outp;
reg outp;
reg [1:0] state;
reg [1:0] next_state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
if (reset)
state <= s1;
else
state <= next_state;
end

always @(state or x1)
begin
case (state)
s1: if (x1 == 1'b1)
next_state = s2;
else
```

```
 next_state = s3;
s2: next_state = s4;
s3: next_state = s4;
s4: next_state = s1;
endcase
end
```