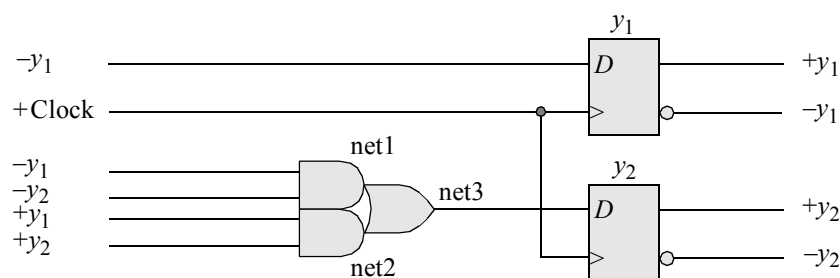


# 2

## Synthesis of Synchronous Sequential Machines 1 Using Verilog HDL

### 2.1 Problems

- 2.1 Determine the counting sequence for the counter shown below by designing the counter using structural modeling with built-in primitives and  $D$  flip-flops that were designed using behavioral modeling. The  $D$  flip-flops have an implied reset input. Obtain the structural design module, the test bench module, the outputs, and the waveforms. The counter is reset initially; that is,  $y_1y_2 = 00$ , where  $y_2$  is the low-order flip-flop.



```
//structural counter using built-in primitives

module ctr_struct1 (rst_n, clk, y);

input rst_n, clk;
output [1:2] y;

//define internal nets
wire net1, net2, net3;

//instantiate the logic for flip-flop y[1]
d_ff_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .d(~y[1]),
    .q(y[1])
);

//instantiate the logic for flip-flop y[2]
and (net1, ~y[1], ~y[2]);
and (net2, y[1], y[2]);
or (net3, net1, net2);

d_ff_bh inst2 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net3),
    .q(y[2])
);

endmodule
```

```

//test bench for ctr_struc1

module ctr_struc1_tb;

reg rst_n, clk;
wire [1:2] y;

//display outputs
initial
$monitor ("count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define length of simulation
initial
begin
    #60 $finish;
end

//instantiate the module into the test bench
ctr_struc1 inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

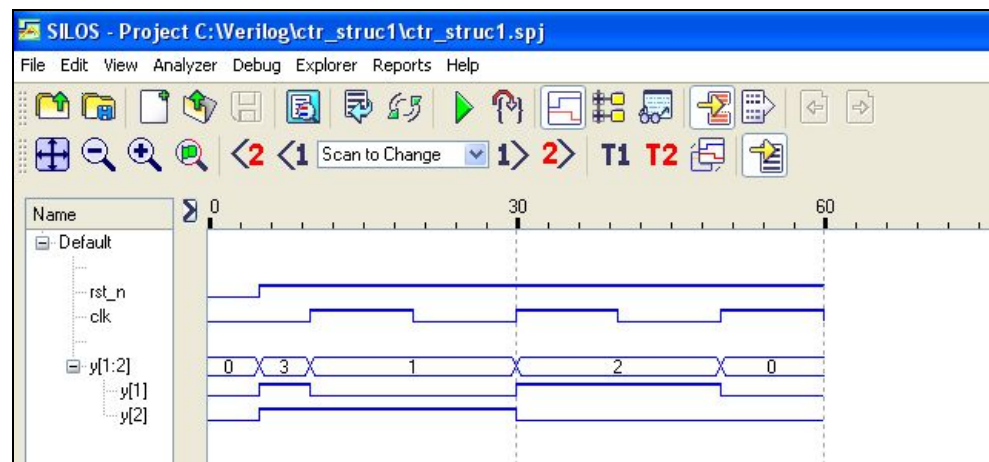
endmodule

```

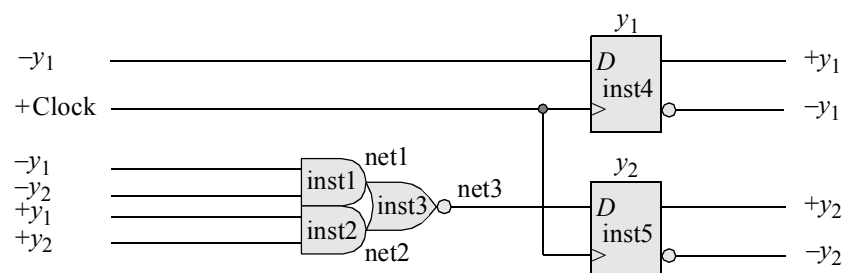
```

count = 00
count = 11
count = 01
count = 10
count = 00

```



- 2.2 Determine the counting sequence for the counter shown below by designing the counter using structural modeling with instantiated logic gates that were designed using dataflow modeling and  $D$  flip-flops that were designed using behavioral modeling. The  $D$  flip-flops have an implied reset input. Obtain the structural design module, the test bench module, the outputs, and the waveforms. The counter is reset initially; that is,  $y_1 y_2 = 00$ , where  $y_2$  is the low-order flip-flop.



```

//structural counter using built-in primitives

module ctr_struc2 (rst_n, clk, y);

input rst_n, clk;
output [1:2] y;

//define internal nets
wire net1, net2, net3;

//instantiate the logic for flip-flop y[1]
d_ff_bh inst4 (
    .rst_n(rst_n),
    .clk(clk),
    .d(~y[1]),
    .q(y[1])
);

//instantiate the logic for flip-flop y[2]
and2_df inst1 (
    .x1(~y[1]),
    .x2(~y[2]),
    .z1(net1)
);

and2_df inst2 (
    .x1(y[1]),
    .x2(y[2]),
    .z1(net2)
);

nor2_df inst3 (
    .x1(net1),
    .x2(net2),
    .z1(net3)
);

d_ff_bh inst5 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net3),
    .q(y[2])
);

endmodule

```

```

//test bench for ctr_struc2

module ctr_struc2_tb;

reg rst_n, clk;
wire [1:2] y;

//display outputs
initial
$monitor ("count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define length of simulation
initial
begin
    #60 $finish;
end

//instantiate the module into the test bench
ctr_struc2 inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

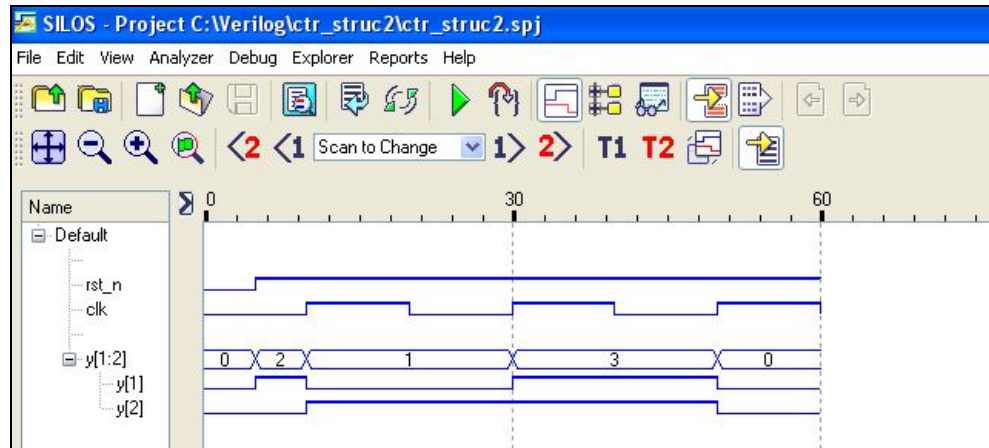
endmodule

```

```

count = 00
count = 10
count = 01
count = 11
count = 00

```



- 2.3 Using behavioral modeling, design a counter that counts in the sequence shown below. Obtain the design module, the test bench module, the outputs, and the waveforms.

$y_1y_2y_3y_4 = 0000, 1001, 0001, 1011, 0010, 1100, 0011, 1101, 1000, 1110, 1010, 1111, 0000, \dots$

```

//behavioral nonsequential counter
module ctr_nonseq_bh (rst_n, clk, count);
input rst_n, clk;
output [1:4] count;

//variables used in always are declared as reg
reg [1:4] count, next_count;

always @ (negedge rst_n or negedge clk)
begin
    if (~rst_n) //if reset = 0
        count = 4'b0000;
    else
        count = next_count;
end //continued on next page

```

```

//define the counting sequence
always @ (count)
begin
    case (count)
        4'b0000 : next_count = 4'b1001;
        4'b1001 : next_count = 4'b0001;
        4'b0001 : next_count = 4'b1011;
        4'b1011 : next_count = 4'b0010;
        4'b0010 : next_count = 4'b1100;
        4'b1100 : next_count = 4'b0011;
        4'b0011 : next_count = 4'b1101;
        4'b1101 : next_count = 4'b1000;
        4'b1000 : next_count = 4'b1110;
        4'b1110 : next_count = 4'b1010;
        4'b1010 : next_count = 4'b1111;
        4'b1111 : next_count = 4'b0000;
    endcase
end
endmodule

```

```

//test bench for modulo-10 counter
module ctr_nonseq_bh_tb;

//define inputs and outputs
reg rst_n, clk;
wire [1:4] count;

//display outputs
initial
$monitor ("count = %b", count);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

```

//continued on next page

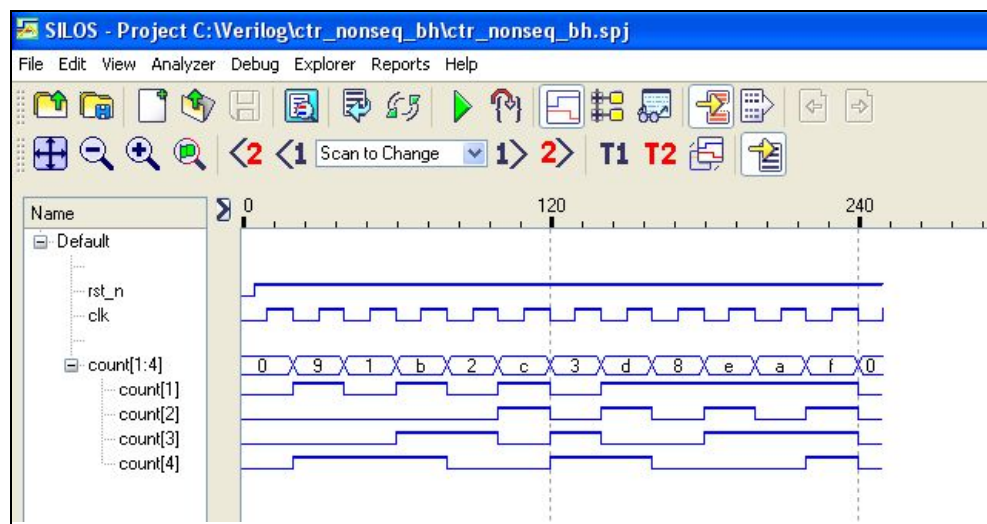


```
//define length of simulation
initial
begin
  #250 $finish;
end

//instantiate the module into the test bench
ctr_nonseq_bh inst1 (
  .rst_n(rst_n),
  .clk(clk),
  .count(count)
);

endmodule
```

count = 0000	count = 1101
count = 1001	count = 1000
count = 0001	count = 1110
count = 1011	count = 1010
count = 0010	count = 1111
count = 1100	count = 0000
count = 0011	



- 2.4 Using structural modeling, design a counter that counts in the sequence shown below. Use built-in primitives and  $D$  flip-flops that were designed using behavioral modeling. Obtain the design module, the test bench module, the outputs, and the waveforms.

$$y_1y_2y_3y_4 = 0000, 1001, 0001, 1011, 0010, 1100, 0011, 1101, 1000, 1110, 1010, 1111, 0000, \dots$$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	<sup>0</sup> 1	<sup>1</sup> 1	<sup>3</sup> 1	<sup>2</sup> 1
	01	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> 0	<sup>6</sup> 0
	11	<sup>12</sup> 0	<sup>13</sup> 1	<sup>15</sup> 0	<sup>14</sup> 1
	10	<sup>8</sup> 1	<sup>9</sup> 0	<sup>11</sup> 0	<sup>10</sup> 1

$y_1$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	<sup>0</sup> 0	<sup>1</sup> 0	<sup>3</sup> 1	<sup>2</sup> 1
	01	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> 0	<sup>6</sup> 0
	11	<sup>12</sup> 0	<sup>13</sup> 0	<sup>15</sup> 0	<sup>14</sup> 0
	10	<sup>8</sup> 1	<sup>9</sup> 0	<sup>11</sup> 0	<sup>10</sup> 1

$y_2$

$$Dy_1 = y_1'y_2' + y_2'y_4' + y_1y_3y_4' + y_1y_2y_3'y_4$$

$$Dy_2 = y_1'y_2'y_3 + y_1y_2'y_4'$$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	<sup>0</sup> 0	<sup>1</sup> 1	<sup>3</sup> 0	<sup>2</sup> 0
	01	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> 0	<sup>6</sup> 0
	11	<sup>12</sup> 1	<sup>13</sup> 0	<sup>15</sup> 0	<sup>14</sup> 1
	10	<sup>8</sup> 1	<sup>9</sup> 0	<sup>11</sup> 1	<sup>10</sup> 1

$y_3$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	<sup>0</sup> 1	<sup>1</sup> 1	<sup>3</sup> 1	<sup>2</sup> 0
	01	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> 0	<sup>6</sup> 0
	11	<sup>12</sup> 1	<sup>13</sup> 0	<sup>15</sup> 0	<sup>14</sup> 0
	10	<sup>8</sup> 0	<sup>9</sup> 1	<sup>11</sup> 0	<sup>10</sup> 1

$y_4$

$$Dy_3 = y_1'y_2'y_3'y_4 + y_1y_4' + y_1y_2'y_3$$

$$Dy_4 = y_1'y_2'y_3' + y_1'y_2'y_4 + y_1y_2y_3'y_4' + y_2'y_3'y_4 + y_1y_2'y_3y_4'$$

```

//structural nonsequential counter

module ctr_nonseq_dff (rst_n, clk, y);

input rst_n, clk;
output [1:4] y;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7, net8, net9,
      net10, net11, net12, net13, net14, net15, net16, net17,
      net18;

//-----
//instantiate the logic for flip-flop y[1]
and (net1, ~y[1], ~y[2]);
and (net2, ~y[2], ~y[4]);
and (net3, y[1], y[3], ~y[4]);
and (net4, y[1], y[2], ~y[3], y[4]);
or (net5, net1, net2, net3, net4);

d_ff_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net5),
    .q(y[1])
);

//-----
//instantiate the logic for flip-flop y[2]
and (net6, ~y[1], ~y[2], y[3]);
and (net7, y[1], ~y[2], ~y[4]);
or (net8, net6, net7);

d_ff_bh inst2 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net8),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and (net9, ~y[1], ~y[2], ~y[3], y[4]);
and (net10, y[1], ~y[4]);
and (net11, y[1], ~y[2], y[3]);
or (net12, net9, net10, net11);

//continued on next page

```

```

d_ff_bh inst3 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net12),
    .q(y[3])
);

//-----
//instantiate the logic for flip-flop y[4]
and (net13, ~y[1], ~y[2], ~y[3]);
and (net14, ~y[1], ~y[2], y[4]);
and (net15, y[1], y[2], ~y[3], ~y[4]);
and (net16, ~y[2], ~y[3], y[4]);
and (net17, y[1], ~y[2], y[3], ~y[4]);
or  (net18, net13, net14, net15, net16, net17);

d_ff_bh inst4 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net18),
    .q(y[4])
);
endmodule

```

```

//test bench for modulo-10 counter
module ctr_nonseq_dff_tb;

    reg rst_n, clk;          //define inputs and outputs
    wire [1:4] y;

    initial                  //display outputs
    $monitor ("count = %b", y);

    //define reset
    initial
    begin
        #0 rst_n = 1'b0;
        #5 rst_n = 1'b1;
    end

    initial                  //define clock
    begin
        clk = 1'b0;
        forever
            #10 clk = ~clk;
    end
end

//continued on next page

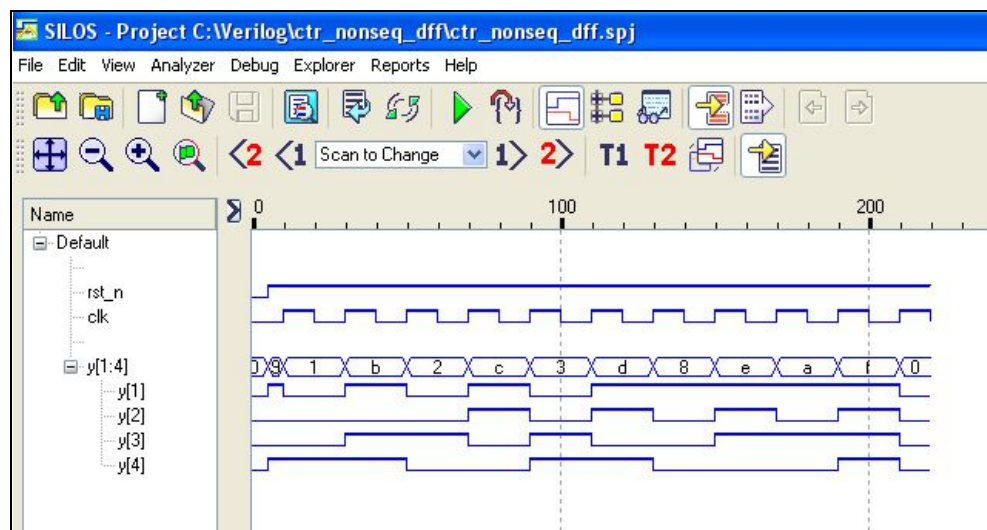
```

```
//define length of simulation
initial
begin
  #220 $finish;
end

//instantiate the module into the test bench
ctr_nonseq_dff inst1 (
  .rst_n(rst_n),
  .clk(clk),
  .y(y)
);

endmodule
```

count = 0000	count = 1101
count = 1001	count = 1000
count = 0001	count = 1110
count = 1011	count = 1010
count = 0010	count = 1111
count = 1100	count = 0000
count = 0011	



- 2.5 Design a modulo-11 counter with no self-starting state using structural modeling with built-in primitives and  $D$  flip-flops that were designed using behavioral modeling. Obtain the design module, the test bench module, the outputs, and the waveforms.

$y_1$	$y_2$	$y_3$	$y_4$
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
0	0	0	0

$y_1 y_2$		$y_3 y_4$			
		00	01	11	10
00		0 <sup>0</sup>	0 <sup>1</sup>	0 <sup>3</sup>	0 <sup>2</sup>
01		0 <sup>4</sup>	0 <sup>5</sup>	1 <sup>7</sup>	0 <sup>6</sup>
11		— <sup>12</sup>	— <sup>13</sup>	— <sup>15</sup>	— <sup>14</sup>
10		1 <sup>8</sup>	1 <sup>9</sup>	— <sup>11</sup>	0 <sup>10</sup>
		$y_1$			

$y_1 y_2$		$y_3 y_4$			
		00	01	11	10
00		0 <sup>0</sup>	0 <sup>1</sup>	1 <sup>3</sup>	0 <sup>2</sup>
01		1 <sup>4</sup>	1 <sup>5</sup>	0 <sup>7</sup>	1 <sup>6</sup>
11		— <sup>12</sup>	— <sup>13</sup>	— <sup>15</sup>	— <sup>14</sup>
10		0 <sup>8</sup>	0 <sup>9</sup>	— <sup>11</sup>	0 <sup>10</sup>
		$y_2$			

$$Dy_1 = y_1 y_3' + y_2 y_3 y_4$$

$$Dy_2 = y_2 y_3' + y_2' y_3 y_4 + y_2 y_4'$$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	0 <sup>0</sup>	1 <sup>1</sup>	0 <sup>3</sup>	1 <sup>2</sup>
	01	0 <sup>4</sup>	1 <sup>5</sup>	0 <sup>7</sup>	1 <sup>6</sup>
	11	— <sup>12</sup>	— <sup>13</sup>	— <sup>15</sup>	— <sup>14</sup>
	10	0 <sup>8</sup>	1 <sup>9</sup>	— <sup>11</sup>	0 <sup>10</sup>

$y_3$

		$y_3y_4$			
		00	01	11	10
$y_1y_2$	00	1 <sup>0</sup>	0 <sup>1</sup>	0 <sup>3</sup>	1 <sup>2</sup>
	01	1 <sup>4</sup>	0 <sup>5</sup>	0 <sup>7</sup>	1 <sup>6</sup>
	11	— <sup>12</sup>	— <sup>13</sup>	— <sup>15</sup>	— <sup>14</sup>
	10	1 <sup>8</sup>	0 <sup>9</sup>	— <sup>11</sup>	0 <sup>10</sup>

$y_4$

$$Dy_3 = y_3'y_4 + y_1'y_3y_4'$$

$$\begin{aligned} Dy_4 &= y_3'y_4' + y_1'y_4' \\ &= y_4'(y_1' + y_3') \end{aligned}$$

```
//structural modulo-11 counter

module ctr_mod11_struc_d (rst_n, clk, y);

input rst_n, clk;
output [1:4] y;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7, net8, net9,
      net10, net11, net12;

//-----
//instantiate the logic for flip-flop y[1]
and (net1, y[1], ~y[3]);
and (net2, y[2], y[3], y[4]);
or (net3, net1, net2);

d_ff_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net3),
    .q(y[1])
);

//continued on next page
```

```

//-----
//instantiate the logic for flip-flop y[2]
and (net4, y[2], ~y[3]);
and (net5, ~y[2], y[3], y[4]);
and (net6, y[2], ~y[4]);
or (net7, net4, net5, net6);

d_ff_bh inst2 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net7),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and (net8, ~y[3], y[4]);
and (net9, ~y[1], y[3], ~y[4]);
or (net10, net8, net9);

d_ff_bh inst3 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net10),
    .q(y[3])
);

//-----
//instantiate the logic for flip-flop y[4]
or (net11, ~y[1], ~y[3]);
and (net12, ~y[4], net11);

d_ff_bh inst4 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net12),
    .q(y[4])
);

endmodule

```



```

//test bench for modulo-11 counter
module ctr_mod11_struc_d_tb;

//define inputs and outputs
reg rst_n, clk;
wire [1:4] y;

//display outputs
initial
$monitor ("count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

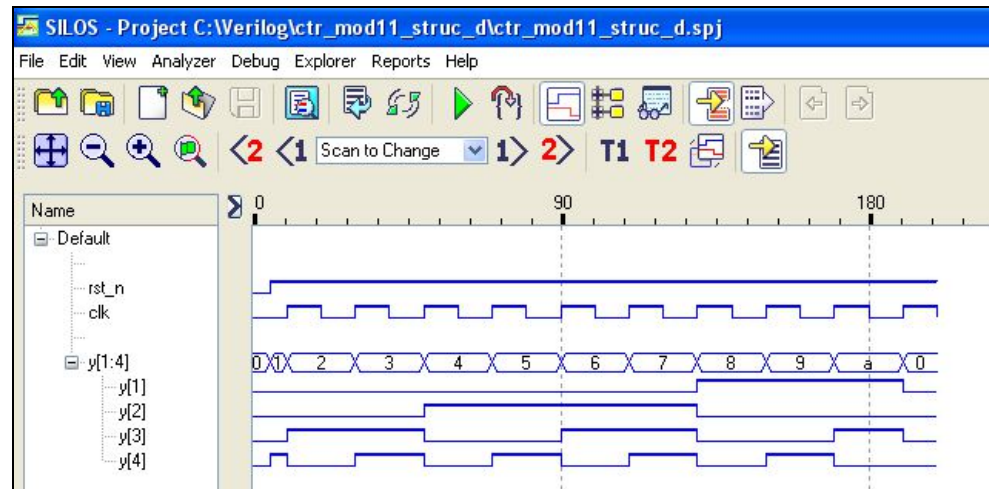
//define length of simulation
initial
begin
    #200 $finish;
end

//instantiate the module into the test bench
ctr_mod11_struc_d inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

endmodule

```

count = 0000	count = 0110
count = 0001	count = 0111
count = 0010	count = 1000
count = 0011	count = 1001
count = 0100	count = 1010
count = 0101	count = 0000



- 2.6 Design a modulo-11 counter with no self-starting state using structural modeling with logic gates that were designed using dataflow modeling and negative-edge triggered  $JK$  flip-flops that were designed using behavioral modeling. Obtain the design module, the test bench module, the outputs, and the waveforms

		$y_3y_4$			
$y_1y_2$		0 0	0 1	1 1	1 0
0 0	0 0	0 <sup>0</sup>	0 <sup>1</sup>	0 <sup>3</sup>	0 <sup>2</sup>
	0 1	4	5	7	6
1 1	1 0	12	13	15	14
	1 1	8	9	11	10

 $Jy_1$ 

$$Jy_1 = y_2y_3y_4$$

		$y_3y_4$			
$y_1y_2$		0 0	0 1	1 1	1 0
0 0	0 0	—	—	—	—
	0 1	4	5	7	6
1 1	1 0	12	13	15	14
	1 1	8	9	11	10

 $Ky_1$ 

$$Ky_1 = y_3$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> 0	<sup>1</sup> 0	<sup>3</sup> 1	<sup>2</sup> 0
01	<sup>4</sup> —	<sup>5</sup> —	<sup>7</sup> —	<sup>6</sup> —
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> 0	<sup>9</sup> 0	<sup>11</sup> —	<sup>10</sup> 0

 $Jy_2$ 

$$Jy_2 = y_3y_4$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> —	<sup>1</sup> —	<sup>3</sup> —	<sup>2</sup> —
01	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> 1	<sup>6</sup> 0
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> —	<sup>9</sup> —	<sup>11</sup> —	<sup>10</sup> —

 $Ky_2$ 

$$Ky_2 = y_3$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> 0	<sup>1</sup> 1	<sup>3</sup> —	<sup>2</sup> —
01	<sup>4</sup> 0	<sup>5</sup> 1	<sup>7</sup> —	<sup>6</sup> —
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> 0	<sup>9</sup> 1	<sup>11</sup> —	<sup>10</sup> —

 $Jy_3$ 

$$Jy_3 = y_4$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> —	<sup>1</sup> —	<sup>3</sup> 1	<sup>2</sup> 0
01	<sup>4</sup> —	<sup>5</sup> —	<sup>7</sup> 1	<sup>6</sup> 0
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> —	<sup>9</sup> —	<sup>11</sup> —	<sup>10</sup> 1

 $Ky_3$ 

$$Ky_3 = y_1y_4$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> 1	<sup>1</sup> —	<sup>3</sup> —	<sup>2</sup> 1
01	<sup>4</sup> 1	<sup>5</sup> —	<sup>7</sup> —	<sup>6</sup> 1
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> 1	<sup>9</sup> —	<sup>11</sup> —	<sup>10</sup> 0

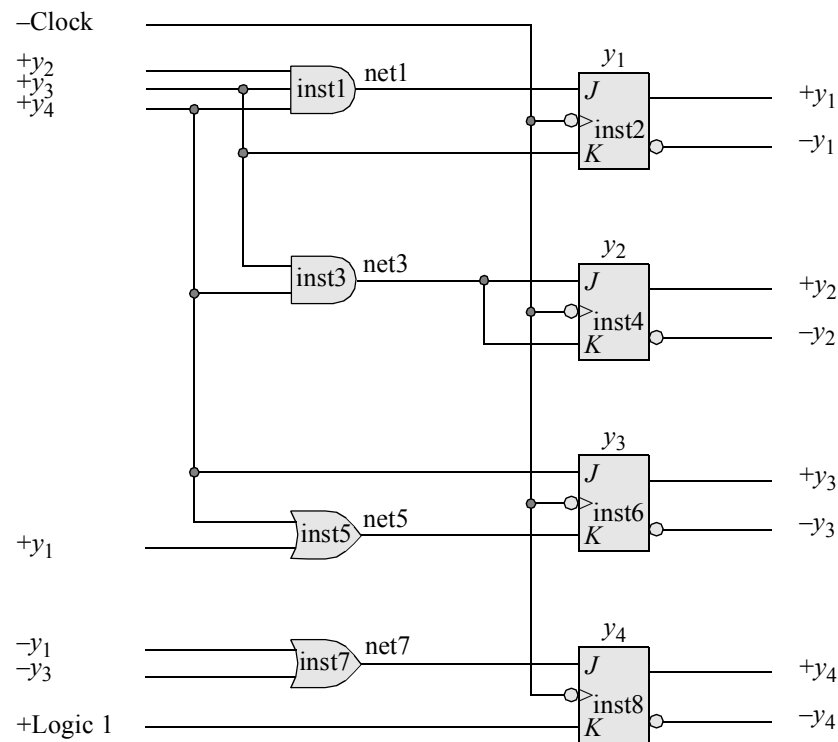
 $Jy_4$ 

$$Jy_4 = y_1' + y_3'$$

$y_1y_2 \backslash y_3y_4$	00	01	11	10
00	<sup>0</sup> —	<sup>1</sup> 1	<sup>3</sup> 1	<sup>2</sup> —
01	<sup>4</sup> —	<sup>5</sup> 1	<sup>7</sup> 1	<sup>6</sup> —
11	<sup>12</sup> —	<sup>13</sup> —	<sup>15</sup> —	<sup>14</sup> —
10	<sup>8</sup> —	<sup>9</sup> 1	<sup>11</sup> —	<sup>10</sup> —

 $Ky_4$ 

$$Ky_4 = 1$$



Assume that the  $JK$  flip-flops have negative set and reset inputs.

```
//structural modulo-11 counter

module ctr_mod11_struc_jk (rst_n, clk, y);

input rst_n, clk;
output [1:4] y;

//define internal nets
wire net1, net2, net3, net4;

//instantiate the logic for flip-flop y[1]
and3_df inst1 (
    .x1(y[2]),
    .x2(y[3]),
    .x3(y[4]),
    .z1(net1)
);

//continued on next page
```

```

jkff_neg_clk inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net1),
    .k(y[3]),
    .q(y[1])
);

//instantiate the logic for flip-flop y[2]
and2_df inst3 (
    .x1(y[3]),
    .x2(y[4]),
    .z1(net3)
);

jkff_neg_clk inst4 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net3),
    .k(net3),
    .q(y[2])
);

//instantiate the logic for flip-flop y[3]
or2_df inst5 (
    .x1(y[4]),
    .x2(y[1]),
    .z1(net5)
);

jkff_neg_clk inst6 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[4]),
    .k(net5),
    .q(y[3])
);

//continued on next page

```

```

//instantiate the logic for flip-flop y[4]
or2_df inst7 (
    .x1(~y[1]),
    .x2(~y[3]),
    .z1(net7)
);

jkff_neg_clk inst8 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net7),
    .k(1'b1),
    .q(y[4])
);

endmodule

```

```

//test bench for modulo-11 counter using JK flip-flops
module ctr_mod11_struc_jk_tb;

//define inputs and outputs
reg clk, set_n, rst_n; //inputs are reg for test bench
wire [1:4] y; //outputs are wire for test bench

//display outputs
initial
$monitor ("Count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

```

//continued on next page

```

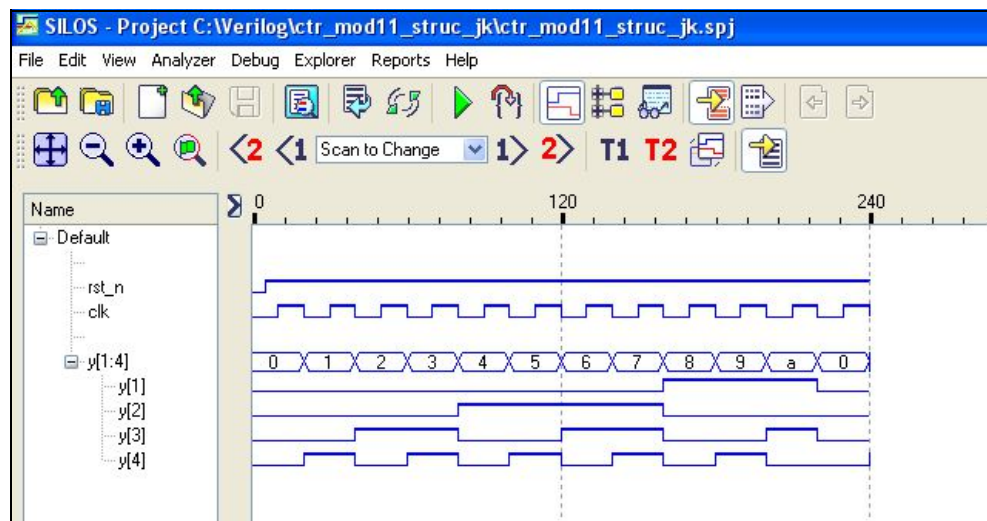
//define length of simulation
initial
begin
    #240 $finish;
end

//instantiate the module into the test bench
ctr_mod11_struct_jk inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

endmodule

```

Count = 0000	Count = 0110
Count = 0001	Count = 0111
Count = 0010	Count = 1000
Count = 0011	Count = 1001
Count = 0100	Count = 1010
Count = 0101	Count = 0000



- 2.7 Design a modulo-11 counter with no self-starting state using behavioral modeling with the **case** statement. Obtain the design module, the test bench module, the outputs, and the waveforms

```
//behavioral modulo-11 counter using the case statement

module ctr_mod11_case (clk, rst_n, y);

input clk, rst_n;
output [1:4] y;

//variables used in always are reg
reg [1:4] y, next_count;

//set next count
always @ (posedge clk or negedge rst_n)
begin
    if (~rst_n)
        y = 4'b0000;
    else
        y = next_count;
end

//determine next count
always @ (y)
begin
    case (y)
        4'b0000: next_count = 4'b0001;
        4'b0001: next_count = 4'b0010;
        4'b0010: next_count = 4'b0011;
        4'b0011: next_count = 4'b0100;
        4'b0100: next_count = 4'b0101;
        4'b0101: next_count = 4'b0110;
        4'b0110: next_count = 4'b0111;
        4'b0111: next_count = 4'b1000;
        4'b1000: next_count = 4'b1001;
        4'b1001: next_count = 4'b1010;
        4'b1010: next_count = 4'b0000;
        default: next_count = 4'b0000;
    endcase
end

endmodule
```



```

//test bench for the modulo-11 counter

module ctr_mod11_case_tb;

//inputs are reg for test bench
reg rst_n, clk;

//outputs are wire for test bench
wire [1:4] y;

//display count
initial
$monitor ("count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define length of simulation
initial
    #220 $finish;

//instantiate the module into the test bench
ctr_mod11_case inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

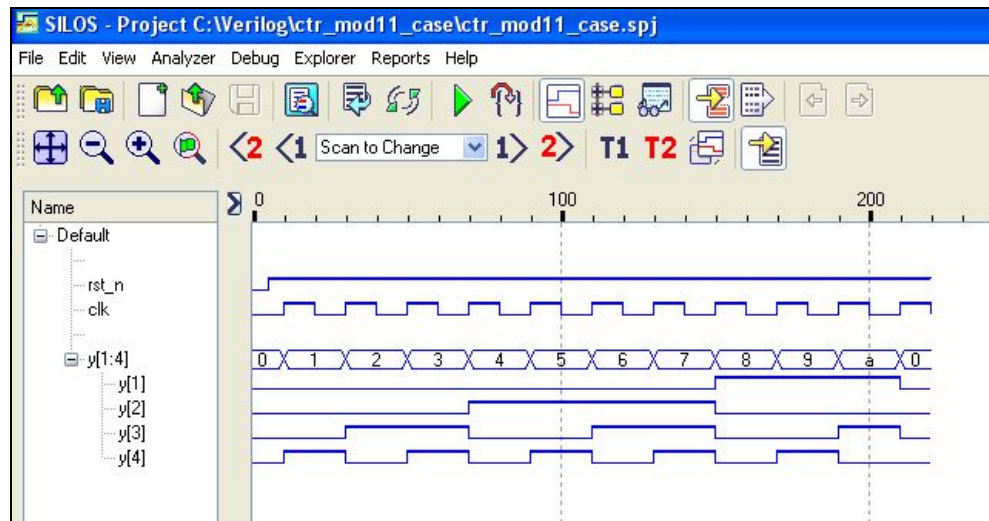
endmodule

```

```

count = 0000      count = 0110
count = 0001      count = 0111
count = 0010      count = 1000
count = 0011      count = 1001
count = 0100      count = 1010
count = 0101      count = 0000

```



- 2.8 Design a counter using structural modeling with  $JK$  flip-flops that counts in the sequence shown below. The counter is not self-starting. Obtain the design module, the test bench module, the outputs, and the waveforms.

$$y_1y_2y_3 = 000, 100, 010, 001, 100, \dots$$

Unused states: 011, 101, 110, 111

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 1	1 1	3 —	2 0
	1	4 —	5 —	7 —	6 —

$$Jy_1$$

$$Jy_1 = y_2'$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 —	1 —	3 —	2 —
	1	4 1	5 —	7 —	6 —

$$Ky_1$$

$$Ky_1 = 1$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 <sup>0</sup> 0	1 <sup>1</sup> 0	3 <sup>3</sup> -	2 <sup>2</sup> -
	1	4 <sup>4</sup> 1	5 <sup>5</sup> -	7 <sup>7</sup> -	6 <sup>6</sup> -

$Jy_2$   
 $Jy_2 = y_1$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 <sup>0</sup> -	1 <sup>1</sup> -	3 <sup>3</sup> -	2 <sup>2</sup> 1
	1	4 <sup>4</sup> -	5 <sup>5</sup> -	7 <sup>7</sup> -	6 <sup>6</sup> -

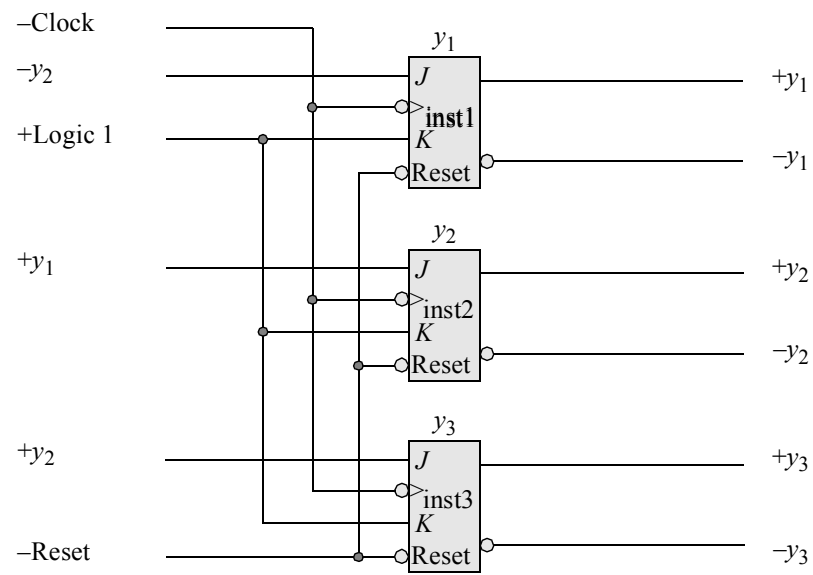
$Ky_2$   
 $Ky_2 = 1$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 <sup>0</sup> 0	1 <sup>1</sup> -	3 <sup>3</sup> -	2 <sup>2</sup> 1
	1	4 <sup>4</sup> 0	5 <sup>5</sup> -	7 <sup>7</sup> -	6 <sup>6</sup> -

$Jy_3$   
 $Jy_3 = y_2$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 <sup>0</sup> -	1 <sup>1</sup> 1	3 <sup>3</sup> -	2 <sup>2</sup> -
	1	4 <sup>4</sup> -	5 <sup>5</sup> -	7 <sup>7</sup> -	6 <sup>6</sup> -

$Ky_3$   
 $Ky_3 = 1$



```

//structural nonsequential counter us JK flip-flops

module ctr_nonseq_jk (rst_n, clk, y);

input rst_n, clk;
output [1:3] y;

//instantiate the logic for flip-flop y[1]
jkff inst1 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(~y[2]),
    .k(1'b1),
    .q(y[1])
);

//instantiate the logic for flip-flop y[2]
jkff inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[1]),
    .k(1'b1),
    .q(y[2])
);

//instantiate the logic for flip-flop y[3]
jkff inst3 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[2]),
    .k(1'b1),
    .q(y[3])
);

endmodule

```

```

//test bench for nonsequential counter using JK flip-flops

module ctr_nonseq_jk_tb;

//define inputs and outputs
reg rst_n, clk;          //inputs are reg for test bench
wire [1:3] y;             //outputs are wire for test bench

//display outputs
initial
$monitor ("Count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

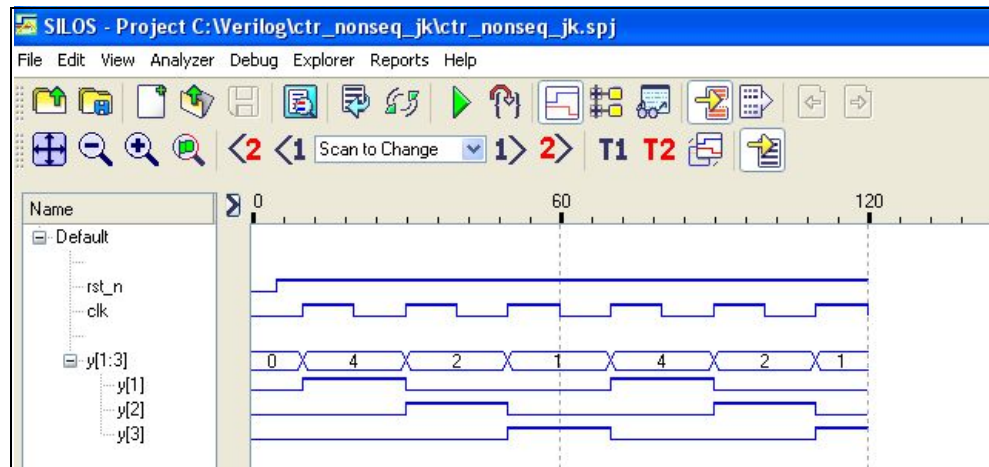
//define length of simulation
initial
begin
    #120 $finish;
end

//instantiate the module into the test bench
ctr_nonseq_jk inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

endmodule

```

```
Count = 000
Count = 100
Count = 010
Count = 001
Count = 100
Count = 010
Count = 001
```



- 2.9 Obtain the input equations for flip-flops  $y_1$  and  $y_4$  only, for a BCD counter which counts in the sequence shown below. The equations are to be in minimum form. Use  $JK$  flip-flops. There is no self-starting state.

$y_1y_2y_3y_4 = 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 0000, \dots$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0	1	3	2
	0 1	4	5	7	6
	1 1	12	13	15	14
	1 0	8	9	11	10

$Jy_1$

$$Jy_1 = y_2y_3y_4$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0	1	3	2
	0 1	4	5	7	6
	1 1	12	13	15	14
	1 0	8	9	11	10

$Ky_1$

$$Ky_1 = y_4$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0 1	1 —	3 —	2 1
	0 1	4 1	5 —	7 —	6 1
	1 1	12 —	13 —	15 —	14 —
	1 0	8 1	9 —	11 —	10 —

 $Jy_4$ 

$$Jy_4 = 1$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0 —	1 1	3 1	2 —
	0 1	4 —	5 1	7 1	6 —
	1 1	12 —	13 —	15 —	14 —
	1 0	8 —	9 1	11 —	10 —

 $Ky_4$ 

$$Ky_4 = 1$$

- 2.10 Design a counter using structural modeling with *JK* flip-flops which counts in the following decimal sequence: 0, 1, 3, 7, 6, 4, 0,  $\dots$ . Obtain the design module, the test bench module, the outputs, and the waveforms.

Unused states: 010, 101

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 0	1 0	3 1	2 —
	1	4 —	5 —	7 —	6 —

 $Jy_1$ 

$$Jy_1 = y_2$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 —	1 —	3 —	2 —
	1	4 1	5 —	7 0	6 0

 $Ky_1$ 

$$Ky_1 = y_2'$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 0	1 1	3 —	2 —
	1	4 0	5 —	7 —	6 —

 $Jy_2$ 

$$Jy_2 = y_3$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 —	1 —	3 0	2 —
	1	4 —	5 —	7 0	6 1

 $Ky_2$ 

$$Ky_2 = y_3'$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 1	1 —	3 —	2 —
	1	4 0	5 —	7 —	6 0

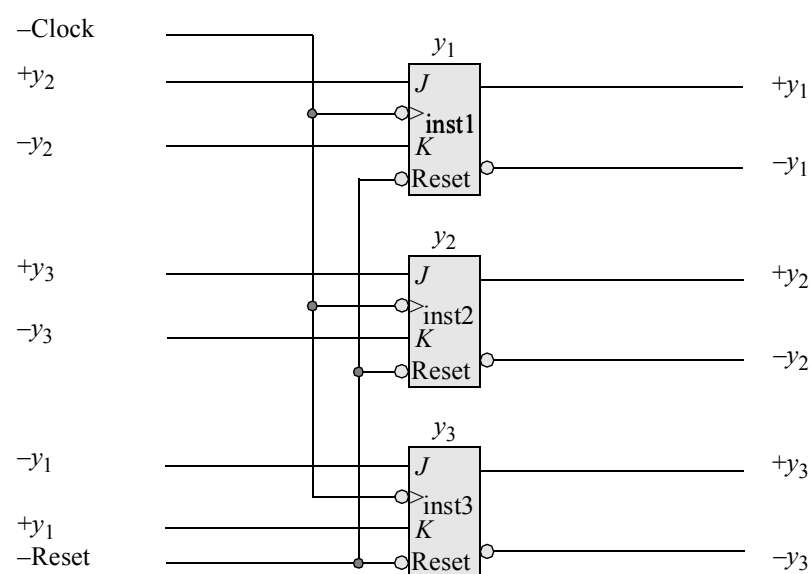
 $Jy_3$ 

$$Jy_3 = y_1'$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 —	1 0	3 0	2 —
	1	4 —	5 —	7 1	6 —

 $Ky_3$ 

$$Ky_3 = 1$$



```
//structural nonsequential counter us JK flip-flops
module ctr_nonseq2_jk (rst_n, clk, y);
input rst_n, clk;
output [1:3] y;
```

```
//instantiate the logic for flip-flop y[1]
jkff inst1 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[2]),
    .k(~y[2]),
    .q(y[1])
);
```

```
//continued on next page
```



```

//instantiate the logic for flip-flop y[2]
jkff inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[3]),
    .k(~y[3]),
    .q(y[2])
);

//instantiate the logic for flip-flop y[3]
jkff inst3 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(~y[1]),
    .k(y[1]),
    .q(y[3])
);
endmodule

```

```

//test bench for nonsequential counter using JK flip-flops
module ctr_nonseq2_jk_tb;

//define inputs and outputs
reg rst_n, clk;      //inputs are reg for test bench
wire [1:3] y;        //outputs are wire for test bench

//display outputs
initial
$monitor ("Count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

```

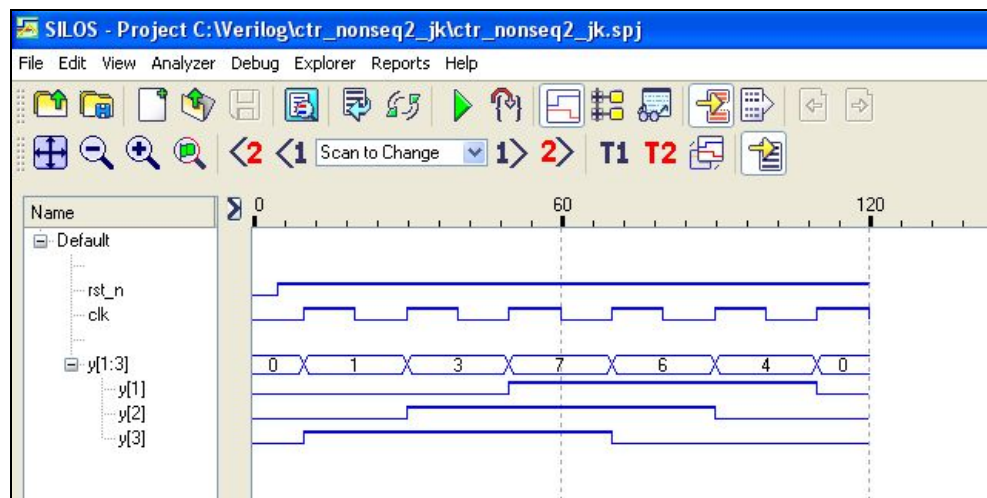
//continued on next page

```
//define length of simulation
initial
begin
    #120 $finish;
end

//instantiate the module into the test bench
ctr_nonseq2_jk inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

endmodule
```

```
Count = 000
Count = 001
Count = 011
Count = 111
Count = 110
Count = 100
Count = 000
```



- 2.11 Design a 4-bit Gray code counter using structural modeling with built-in primitives and  $JK$  flip-flops. The counter is initially reset to  $y_1y_2y_3y_4 = 0000$ . Show the equations for the  $JK$  flip-flops, first in a minimum sum-of-products form, then in an exclusive-OR/NOR form, where applicable. Obtain the design module, the test bench module, the outputs, and the waveforms.

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0 <sup>0</sup> 0	1 <sup>1</sup> 0	3 <sup>3</sup> 0	2 <sup>2</sup> 0
	0 1	4 <sup>4</sup> 1	5 <sup>5</sup> 0	7 <sup>7</sup> 0	6 <sup>6</sup> 0
	1 1	12 <sup>12</sup> —	13 <sup>13</sup> —	15 <sup>15</sup> —	14 <sup>14</sup> —
	1 0	8 <sup>8</sup> —	9 <sup>9</sup> —	11 <sup>11</sup> —	10 <sup>10</sup> —

 $Jy_1$ 

$$Jy_1 = y_2y_3'y_4'$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	— <sup>0</sup>	— <sup>1</sup>	— <sup>3</sup>	— <sup>2</sup>
	0 1	— <sup>4</sup>	— <sup>5</sup>	— <sup>7</sup>	— <sup>6</sup>
	1 1	0 <sup>12</sup>	0 <sup>13</sup>	0 <sup>15</sup>	0 <sup>14</sup>
	1 0	1 <sup>8</sup>	0 <sup>9</sup>	0 <sup>11</sup>	0 <sup>10</sup>

 $Ky_1$ 

$$Ky_1 = y_2'y_3'y_4'$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0 <sup>0</sup> 0	1 <sup>1</sup> 0	3 <sup>3</sup> 0	2 <sup>2</sup> 1
	0 1	— <sup>4</sup>	— <sup>5</sup>	— <sup>7</sup>	— <sup>6</sup>
	1 1	— <sup>12</sup>	— <sup>13</sup>	— <sup>15</sup>	— <sup>14</sup>
	1 0	0 <sup>8</sup>	0 <sup>9</sup>	0 <sup>11</sup>	0 <sup>10</sup>

 $Jy_2$ 

$$Jy_2 = y_1'y_3y_4'$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	— <sup>0</sup>	— <sup>1</sup>	— <sup>3</sup>	— <sup>2</sup>
	0 1	0 <sup>4</sup>	0 <sup>5</sup>	0 <sup>7</sup>	0 <sup>6</sup>
	1 1	0 <sup>12</sup>	0 <sup>13</sup>	0 <sup>15</sup>	1 <sup>14</sup>
	1 0	— <sup>8</sup>	— <sup>9</sup>	— <sup>11</sup>	— <sup>10</sup>

 $Ky_2$ 

$$Ky_2 = y_1y_3y_4'$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	0 <sup>0</sup> 0	1 <sup>1</sup> 1	— <sup>3</sup> —	— <sup>2</sup> —
	0 1	— <sup>4</sup> 0	— <sup>5</sup> 0	— <sup>7</sup> —	— <sup>6</sup> —
	1 1	— <sup>12</sup> 0	— <sup>13</sup> 1	— <sup>15</sup> —	— <sup>14</sup> —
	1 0	— <sup>8</sup> 0	— <sup>9</sup> 0	— <sup>11</sup> —	— <sup>10</sup> —

 $Jy_3$ 

$$Jy_3 = y_1'y_2'y_4 + y_1y_2y_4$$

$$= (y_1 \oplus y_2)y_4$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	— <sup>0</sup> —	— <sup>1</sup> —	0 <sup>3</sup> 0	0 <sup>2</sup> 0
	0 1	— <sup>4</sup> —	— <sup>5</sup> —	1 <sup>7</sup> 1	0 <sup>6</sup> 0
	1 1	— <sup>12</sup> —	— <sup>13</sup> —	0 <sup>15</sup> 0	0 <sup>14</sup> 0
	1 0	— <sup>8</sup> —	— <sup>9</sup> —	1 <sup>11</sup> 1	0 <sup>10</sup> 0

 $Ky_3$ 

$$Ky_3 = y_1'y_2y_4 + y_1y_2'y_4$$

$$= (y_1 \oplus y_2)y_4$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	1 <sup>0</sup> 1	— <sup>1</sup> —	— <sup>3</sup> —	0 <sup>2</sup> 0
	0 1	— <sup>4</sup> 0	— <sup>5</sup> —	— <sup>7</sup> —	1 <sup>6</sup> 1
	1 1	— <sup>12</sup> 1	— <sup>13</sup> —	— <sup>15</sup> —	0 <sup>14</sup> 0
	1 0	— <sup>8</sup> 0	— <sup>9</sup> —	— <sup>11</sup> —	1 <sup>10</sup> 1

 $Jy_4$ 

$$Jy_4 = y_1'y_2'y_3' + y_1y_2y_3' + y_1'y_2y_3 + y_1y_2'y_3$$

$$= (y_1 \oplus y_2)'y_3' + (y_1 \oplus y_2)y_3$$

$$= (y_1 \oplus y_2 \oplus y_3)'$$

		$y_3y_4$			
		0 0	0 1	1 1	1 0
$y_1y_2$	0 0	— <sup>0</sup> —	0 <sup>1</sup> 0	1 <sup>3</sup> 1	— <sup>2</sup> —
	0 1	— <sup>4</sup> —	1 <sup>5</sup> 1	0 <sup>7</sup> 0	— <sup>6</sup> —
	1 1	— <sup>12</sup> —	0 <sup>13</sup> 0	1 <sup>15</sup> 1	— <sup>14</sup> —
	1 0	— <sup>8</sup> —	1 <sup>9</sup> 1	0 <sup>11</sup> 0	— <sup>10</sup> —

 $Ky_4$ 

$$Ky_4 = y_1'y_2'y_3 + y_1y_2y_3 + y_1'y_2y_3' + y_1y_2'y_3'$$

$$= (y_1 \oplus y_2)'y_3 + (y_1 \oplus y_2)y_3'$$

$$= (y_1 \oplus y_2 \oplus y_3)$$

```

//structural gray code counter using built-in primitives

module gray_code_ctr_struct (set_n, rst_n, clk, y);

input set_n, rst_n, clk;
output [1:4] y;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7, net8, net9,
    net10;
wire net11, net12, net13, net14, net15, net16, net17, net18;

//-----
//instantiate the logic for flip-flop y[1]
and(net1, y[2], ~y[3], ~y[4]);
and(net2, ~y[2], ~y[3], ~y[4]);

jkff_neg_clk inst1 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net1),
    .k(net2),
    .q(y[1])
);

//-----
//instantiate the logic for flip-flop y[2]
and(net3, ~y[1], y[3], ~y[4]);
and(net4, y[1], y[3], ~y[4]);

jkff_neg_clk inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net3),
    .k(net4),
    .q(y[2])
);

//-----

//continued on next page

```

```

//-----
//instantiate the logic for flip-flop y[3]
xnor (net5, y[1], y[2]);
and (net6, net5, y[4]);
xor (net7, y[1], y[2]);
and (net8, net7, y[4]);

jkff_neg_clk inst3 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net6),
    .k(net8),
    .q(y[3])
);

//-----
//instantiate the logic for flip-flop y[4]
xnor (net9, y[1], y[2]);
and (net10, net9, ~y[3]);
xor (net11, y[1], y[2]);
and (net12, net11, y[3]);
or (net13, net10, net12);
//-----
xnor (net14, y[1], y[2]);
and (net15, net14, y[3]);
xor (net16, y[1], y[2]);
and (net17, net16, ~y[3]);
or (net18, net15, net17);

jkff_neg_clk inst4 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net13),
    .k(net18),
    .q(y[4])
);

endmodule

```

```

//test bench for the structural gray code counter
//using built-in primitives

module gray_code_ctr_struct_tb;

//define inputs and outputs
reg set_n, rst_n, clk;
wire [1:4] y;

//display outputs
initial
$monitor ("count %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10clk = ~clk;
end

//define length of simulation
initial
begin
    #340 $finish;
end

//instantiate the module into the test bench
gray_code_ctr_struct inst1 (
    .set_n(set_n),
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

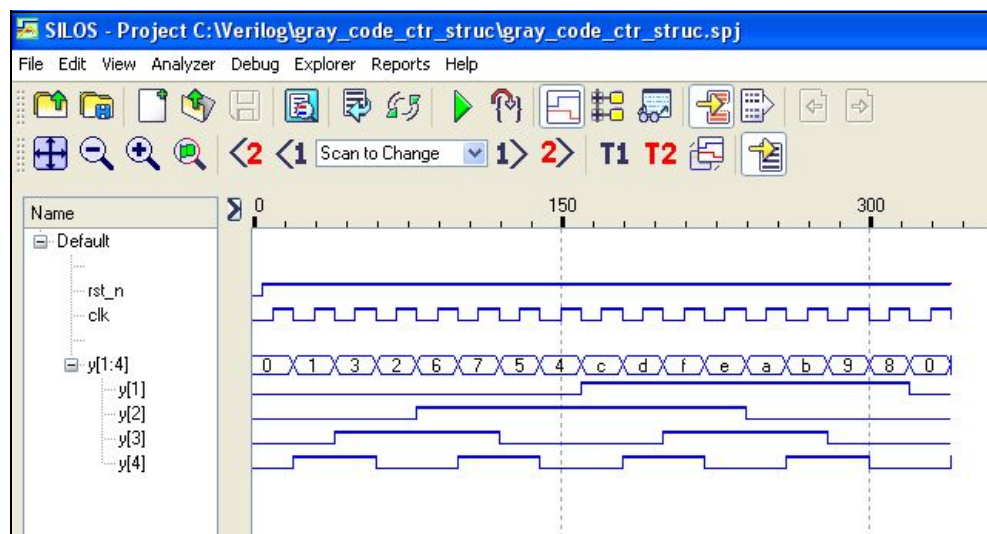
endmodule

```

```

count 0000
count 0001
count 0011
count 0010
count 0110
count 0111
count 0101
count 0100
count 1100
count 1101
count 1111
count 1110
count 1010
count 1011
count 1001
count 1000
count 0000

```



- 2.12 Repeat problem 2.11 using behavioral modeling with the **case** statement. The counting sequence is as follows: 0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8, 0,  $\dots$ . The counter is initially reset to  $y_1y_2y_3y_4 = 0000$ . Obtain the design module, the test bench module, the outputs, and the waveforms.



```

//behavioral gray code counter using the case statement

module gray_code_ctr_bh (rst_n, clk, y);

input rst_n, clk;
output [1:4] y;

//variables are reg in always
reg [1:4] y, next_count;

//set next count
always @ (negedge rst_n or posedge clk)
begin
    if (~rst_n)
        y = 4'b0000;
    else
        y = next_count;
end

//determine next count
always @ (y)
begin
    case (y)
        4'b0000 : next_count = 4'b0001;
        4'b0001 : next_count = 4'b0011;
        4'b0011 : next_count = 4'b0010;
        4'b0010 : next_count = 4'b0110;
        4'b0110 : next_count = 4'b0111;
        4'b0111 : next_count = 4'b0101;
        4'b0101 : next_count = 4'b0100;
        4'b0100 : next_count = 4'b1100;
        4'b1100 : next_count = 4'b1101;
        4'b1101 : next_count = 4'b1111;
        4'b1111 : next_count = 4'b1110;
        4'b1110 : next_count = 4'b1010;
        4'b1010 : next_count = 4'b1011;
        4'b1011 : next_count = 4'b1001;
        4'b1001 : next_count = 4'b1000;
        4'b1000 : next_count = 4'b0000;
    endcase
end

endmodule

```

```

//test bench for gray code counter using case statement

module gray_code_ctr_bh_tb;

//inputs are reg for test bench
reg rst_n, clk;

//outputs are wire for test bench
wire [1:4] y;

//display count
initial
$monitor ("count = %b", y);

//define reset
initial
begin
    #0 rst_n = 1'b0;
    #5 rst_n = 1'b1;
end

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define length of simulation
initial
    #320 $finish;

//instantiate the module into the test bench
gray_code_ctr_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .y(y)
);

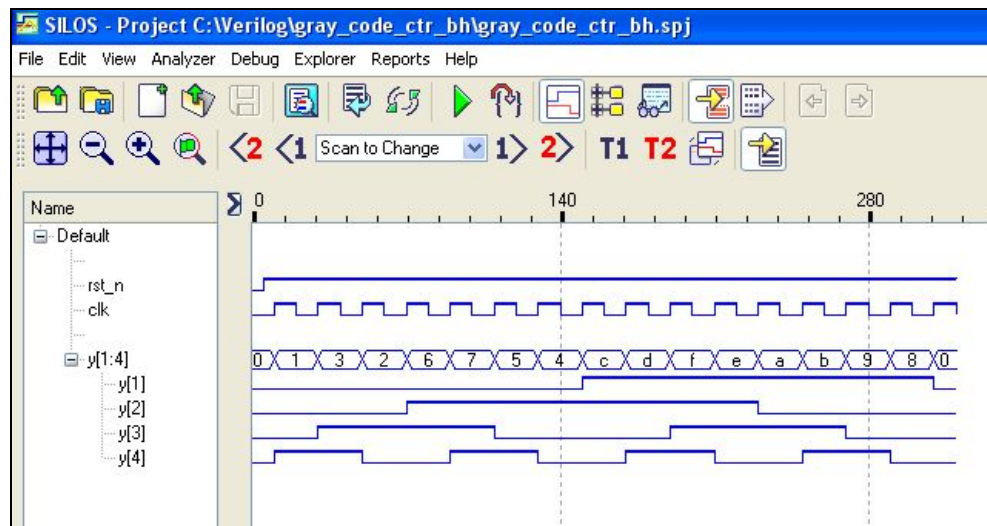
endmodule

```

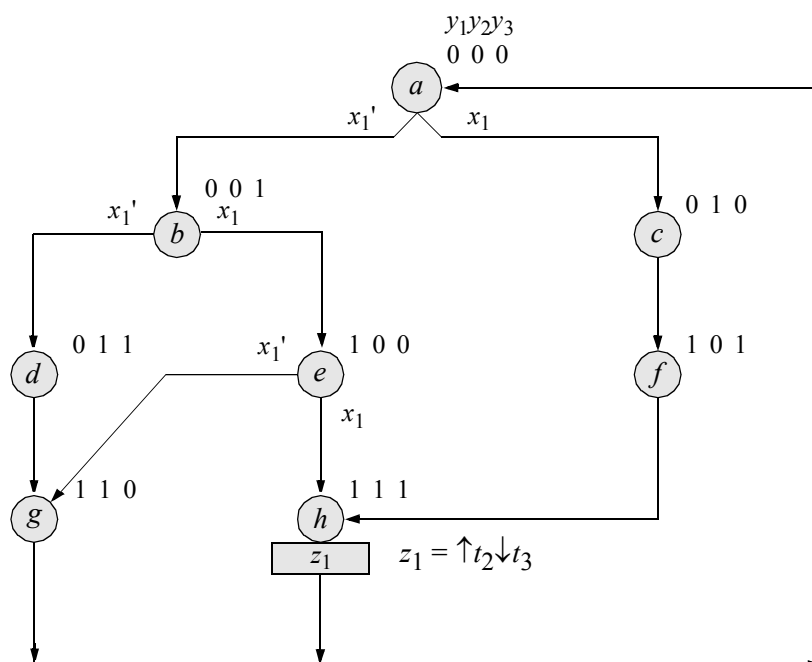
```

count = 0000
count = 0001
count = 0011
count = 0010
count = 0110
count = 0111
count = 0101
count = 0100
count = 1100
count = 1101
count = 1111
count = 1110
count = 1010
count = 1011
count = 1001
count = 1000
count = 0000

```



- 2.13 Generate a reduced state diagram for a Moore machine which generates an output  $z_1$  whenever a serial, 4-bit binary word on an input line  $x_1$  is greater than or equal to six. The first bit received in each word is the high-order bit. There is no space between words. Output  $z_1$  is asserted during the fourth bit of a word. Then implement the state diagram in behavioral modeling. Assert output  $z_1$  at time  $t_2$  and deassert  $z_1$  at time  $t_3$ . Obtain the design module, the test bench module, the outputs, and the waveforms.



```

//behavioral moore ssm to detect greater/equal to 6
module moore_ssm_ge6 (rst_n, clk, x1, y, z1);

//define inputs and outputs
input rst_n, clk, x1;
output [1:3] y;
output z1;

//variables are reg in always
reg [1:3] y, next_state;
reg z1;

//assign state codes
//parameter defines a constant
parameter state_a = 3'b000,
           state_b = 3'b001,
           state_c = 3'b010,
           state_d = 3'b011,
           state_e = 3'b100,
           state_f = 3'b101,
           state_g = 3'b110,
           state_h = 3'b111;

//continued on next page

```

```

//set next state
always @ (posedge clk)
begin
    if (~rst_n)           //if rst_n = 0 (~rst_n is true)
        y <= state_a;     //go to state_a (000)
    else
        y <= next_state;  //else go to next_state
end

//determine output
always @ (y or clk)
begin
    if (y == state_h)
    begin
        if (~clk)
            z1 = 1'b1;
        else
            z1 = 1'b0;
        end

    else
        z1 = 1'b0;
    end

end

//determine next state
always @ (x1 or y)
begin
    case (y) //case is a multiway conditional branch
    state_a: //if y = state_a, do if ... else
        if (~x1)
            next_state = state_b;
        else
            next_state = state_c;

    state_b:
        if (~x1)
            next_state = state_d;
        else
            next_state = state_e;

    state_c: next_state = state_f;

    state_d: next_state = state_g;

    state_e:
        if (~x1)
            next_state = state_g;
        else
            next_state = state_h;    //continued on next page
    endcase
end

```

```

        state_f: next_state = state_h;

        state_g: next_state = state_a;

        state_h: next_state = state_a;

        default: next_state = state_a;
    endcase
end

endmodule

```

```

//test bench for moore_ssm_ge6

module moore_ssm_ge6_tb;

    reg rst_n, clk, x1;           //inputs are reg for test bench
    wire [1:3] y;                 //outputs are wire for test bench
    wire z1;

    //display variables
    initial
    $monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

    //define clock
    initial
    begin
        clk = 1'b0;
        forever
            #10 clk = ~clk;
    end

    //define input sequence
    initial
    begin
        #0  rst_n = 1'b0;          //reset to state_a (000)
        x1 = 1'b0;

        #5  rst_n = 1'b1;          //deassert reset

        //-----
        @ (posedge clk)            //go to state_a (000)
        x1 = 1'b0; @ (posedge clk) //go to state_b (001)
        x1 = 1'b0; @ (posedge clk) //go to state_d (011)
        x1 = $random; @ (posedge clk) //go to state_g (110)
        x1 = $random; @ (posedge clk) //go to state_a (000)
        //continued on next page
    end
endmodule

```

```

//-----
x1 = 1'b0; @ (posedge clk) //go to state_b (001)
x1 = 1'b1; @ (posedge clk) //go to state_e (100)
x1 = 1'b1; @ (posedge clk) //go to state_h (111)
//assert z1
x1 = $random; @ (posedge clk) //go to state_a (000)

//-----
x1 = 1'b0; @ (posedge clk) //go to state_b (001)
x1 = 1'b1; @ (posedge clk) //go to state_e (100)
x1 = 1'b0; @ (posedge clk) //go to state_g (110)
x1 = $random; @ (posedge clk) //go to state_a (000)

//-----
x1 = 1'b1; @ (posedge clk) //go to state_c (010)
x1 = $random; @ (posedge clk) //go to state_f (101)
x1 = $random; @ (posedge clk) //go to state_h (111)
//assert z1
x1 = $random; @ (posedge clk) //go to state_a (000)

//-----
#20 $stop;

end

//-----
//instantiate the module into the test bench
moore_ssm_ge6 inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);

endmodule

```

```

x1 = 0, state = xxx, z1 = 0
x1 = 0, state = 000, z1 = 0
x1 = 0, state = 001, z1 = 0
x1 = 0, state = 011, z1 = 0
x1 = 1, state = 110, z1 = 0

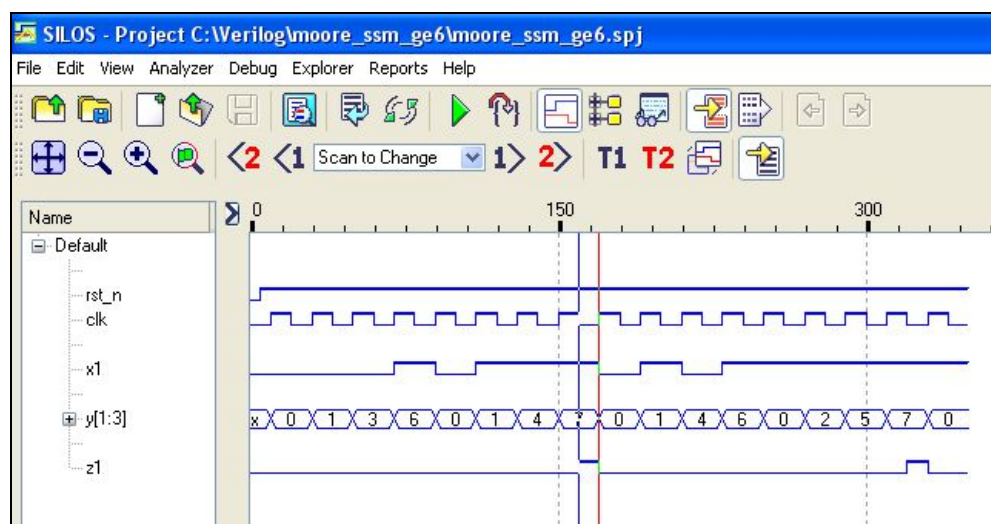
x1 = 0, state = 000, z1 = 0
x1 = 1, state = 001, z1 = 0
x1 = 1, state = 100, z1 = 0
x1 = 1, state = 111, z1 = 0
x1 = 1, state = 111, z1 = 1

x1 = 0, state = 000, z1 = 0
x1 = 1, state = 001, z1 = 0
x1 = 0, state = 100, z1 = 0
x1 = 1, state = 110, z1 = 0

x1 = 1, state = 000, z1 = 0
x1 = 1, state = 010, z1 = 0
x1 = 1, state = 101, z1 = 0
x1 = 1, state = 111, z1 = 0
x1 = 1, state = 111, z1 = 1

x1 = 1, state = 000, z1 = 0

```





- 2.14 This problem is similar to Problem 2.13 in that it detects a number that is greater than or equal to six, but uses a user-defined primitive (UDP) that is created by means of a table that defines the functionality of the primitive. The primitive generates an output  $z_1$  whenever a 4-bit binary word  $x_1, x_2, x_3, x_4$  is greater than or equal to six, where  $x_4$  is the low-order bit. Obtain the primitive module, the test bench module, the outputs, and the waveforms.

```
//circuit to detect a number greater/equal to 6
//using a user-defined primitive table

primitive ge6_table (z1, x1, x2, x3, x4);

output z1;
input x1, x2, x3, x4;

table
//inputs are in the same order as the input list
// x1 x2 x3 x4 : z1; comment is for readability
  0 0 0 0 : 0;
  0 0 0 1 : 0;
  0 0 1 0 : 0;
  0 0 1 1 : 0;
  0 1 0 0 : 0;
  0 1 0 1 : 0;
  0 1 1 0 : 1;
  0 1 1 1 : 1;
  1 0 0 0 : 1;
  1 0 0 1 : 1;
  1 0 1 0 : 1;
  1 0 1 1 : 1;
  1 1 0 0 : 1;
  1 1 0 1 : 1;
  1 1 1 0 : 1;
  1 1 1 1 : 1;
endtable

endprimitive
```

```
//test bench for detect number greater/equal to 6

module ge6_table_tb;

reg x1, x2, x3, x4;    //inputs are reg for test bench
wire z1;              //outputs are wire for test bench

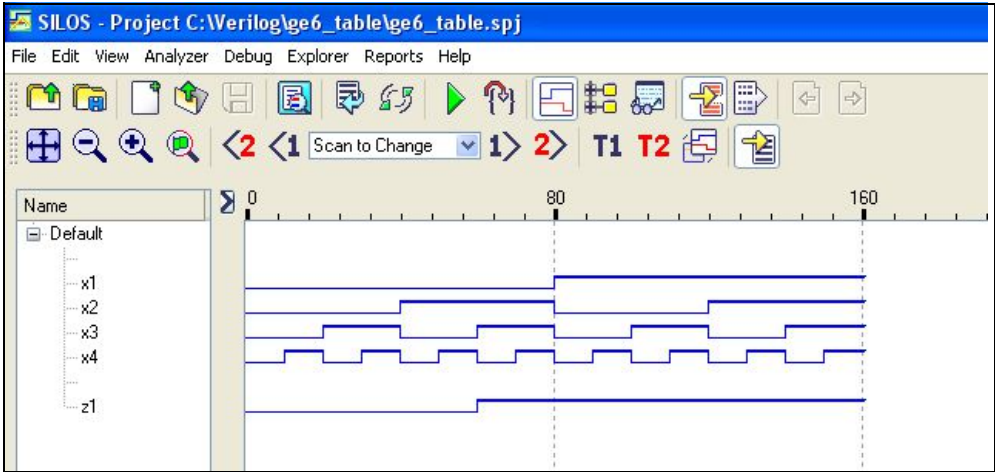
initial
begin: apply_stimulus //name is required
    reg [4:0] invect;
    for (invect = 0; invect < 16; invect = invect + 1)
        begin
            {x1, x2, x3, x4} = invect [3:0];
            #10 $display ("x1x2x3x4 = %b%b%b%b, z1 = %b",
                          x1, x2, x3, x4, z1);
        end
    end

//instantiate the module into the test bench in single line
ge6_table inst1 (z1, x1, x2, x3, x4);

endmodule
```

```
x1x2x3x4 = 0000, z1 = 0
x1x2x3x4 = 0001, z1 = 0
x1x2x3x4 = 0010, z1 = 0
x1x2x3x4 = 0011, z1 = 0
x1x2x3x4 = 0100, z1 = 0
x1x2x3x4 = 0101, z1 = 0

x1x2x3x4 = 0110, z1 = 1
x1x2x3x4 = 0111, z1 = 1
x1x2x3x4 = 1000, z1 = 1
x1x2x3x4 = 1001, z1 = 1
x1x2x3x4 = 1010, z1 = 1
x1x2x3x4 = 1011, z1 = 1
x1x2x3x4 = 1100, z1 = 1
x1x2x3x4 = 1101, z1 = 1
x1x2x3x4 = 1110, z1 = 1
x1x2x3x4 = 1111, z1 = 1
```



2.15 A simpler method to detect a number that is greater than or equal to six is to use a Karnaugh map and the continuous assignment statement. This technique models dataflow behavior and is used to design combinational logic. The continuous assignment statement uses the keyword **assign**. The design generates an output  $z_1$  whenever a 4-bit binary word  $x_1, x_2, x_3, x_4$  is greater than or equal to six, where  $x_4$  is the low-order bit. Obtain the design module, the test bench module, the outputs, and the waveforms.

		$x_3x_4$			
		0 0	0 1	1 1	1 0
$x_1x_2$	0 0	0 <sup>0</sup> 0	1 <sup>1</sup> 0	3 <sup>3</sup> 0	2 <sup>2</sup> 0
	0 1	4 <sup>4</sup> 0	5 <sup>5</sup> 0	7 <sup>7</sup> 1	6 <sup>6</sup> 1
	1 1	12 <sup>12</sup> 1	13 <sup>13</sup> 1	15 <sup>15</sup> 1	14 <sup>14</sup> 1
	1 0	8 <sup>8</sup> 1	9 <sup>9</sup> 1	11 <sup>11</sup> 1	10 <sup>10</sup> 1
		$z_1$			

$$z_1 = x_1 + x_2x_3$$

```
//dataflow to detect a number greater/equal to 6

module ge6_sop (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;

assign z1 = x1 | (x2 & x3);

endmodule
```

```
//test bench for detect number greater/equal to 6

module ge6_sop_tb;

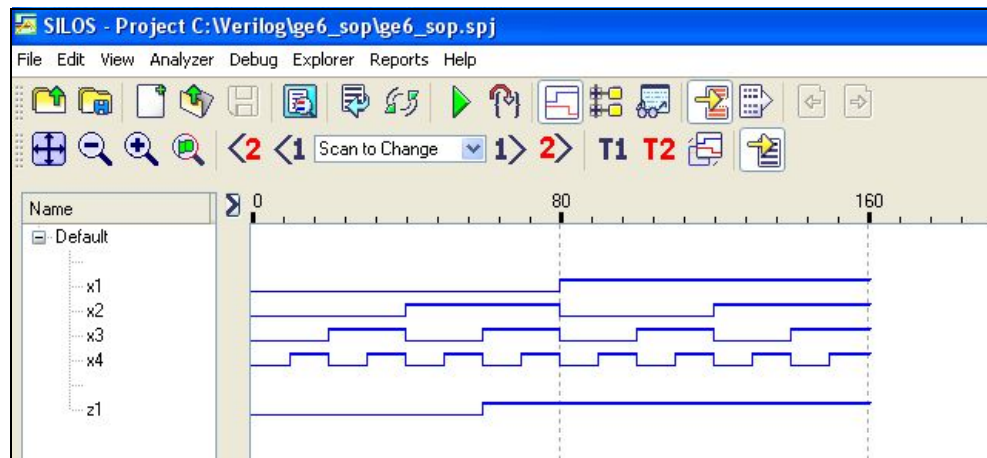
reg x1, x2, x3, x4;    //inputs are reg for test bench
wire z1;              //outputs are wire for test bench

//apply input vectors and display variables
initial
begin: apply_stimulus
    reg [4:0] invec;
    for (invec = 0; invec < 16; invec = invec + 1)
        begin
            {x1, x2, x3, x4} = invec [4:0];
            #10 $display ("x1 x2 x3x x4 = %b, z1 = %b",
                          {x1, x2, x3, x4}, z1);
        end
    end
end

//instantiate the module into the test bench
//with multiple lines
ge6_sop inst1 (
    .x1(x1),
    .x2(x2),
    .x3(x3),
    .x4(x4),
    .z1(z1)
);

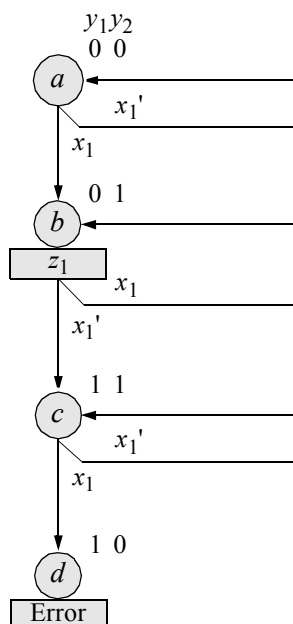
endmodule
```

x1	x2	x3x	x4	=	0000,	z1	=	0
x1	x2	x3x	x4	=	0001,	z1	=	0
x1	x2	x3x	x4	=	0010,	z1	=	0
x1	x2	x3x	x4	=	0011,	z1	=	0
x1	x2	x3x	x4	=	0100,	z1	=	0
x1	x2	x3x	x4	=	0101,	z1	=	0
x1	x2	x3x	x4	=	0110,	z1	=	1
x1	x2	x3x	x4	=	0111,	z1	=	1
x1	x2	x3x	x4	=	1000,	z1	=	1
x1	x2	x3x	x4	=	1001,	z1	=	1
x1	x2	x3x	x4	=	1010,	z1	=	1
x1	x2	x3x	x4	=	1011,	z1	=	1
x1	x2	x3x	x4	=	1100,	z1	=	1
x1	x2	x3x	x4	=	1101,	z1	=	1
x1	x2	x3x	x4	=	1110,	z1	=	1
x1	x2	x3x	x4	=	1111,	z1	=	1



2.16 Generate a reduced state diagram for a Moore machine to detect an input pattern of exactly one group of consecutive 1s on a serial input line  $x_1$ . An unconditional output  $z_1$  is asserted when the first 1 occurs and remains asserted for all additional 1s in the group. The output is deasserted for any 0s following the group. If another 1 occurs, then the machine enters and remains in a terminal state which generates an error output.

For example, a valid input sequence is 000011110000  $\dots$  0, because there is a single group of 1s. An invalid sequence is 000010011100  $\dots$  0, because there is more than one group of 1s. Show the behavioral design module, the test bench module, the outputs, and the waveforms.



```

//behavioral to detect a single group of 1s
module sngl_grp_of_1s (rst_n, clk, x1, y, z1, err);

input rst_n, clk, x1;      //define inputs and outputs
output z1, err;
output [1:2] y;

reg [1:2] y, next_state;  //variables are reg in always
reg z1, err;

//assign state codes, parameter defines a constant
//state names must have at least 2 characters
parameter    state_a = 2'b00,
               state_b = 2'b01,
               state_c = 2'b11,
               state_d = 2'b10;

//set next state
always @ (posedge clk)
begin
    if (~rst_n)           //if (~rst_n) is true,
        y <= state_a;     //y <= state_a
    else
        y <= next_state;
end

                                     //continued on next page

```

```

//determine outputs
always @ (y or x1)
begin
    if (y == state_b)           //== specifies logical
        z1 = 1'b1;              //equality or compare
    else
        z1 = 1'b0;

    if (y == state_d)
        err = 1'b1;
    else
        err = 1'b0;
end

//determine next state
always @ (y or x1)
begin
    case (y)
        state_a:
            if (~x1)
                next_state = state_a;
    else
        next_state = state_b;

        state_b:
            if (x1)
                next_state = state_b;
            else
                next_state = state_c;

        state_c:
            if (~x1)
                next_state = state_c;
            else
                next_state = state_d;

        state_d:
            next_state = state_d;

        default: next_state = state_a;
    endcase
end

endmodule

```

```

//test bench for detect a single group of 1s
module sngl_grp_of_1s_tb;

reg rst_n, clk, x1;           //inputs are reg for test bench
wire [1:2] y;                //outputs are wire for test bench
wire z1, err;

initial                      //display variables
$monitor ("x1 = %b, state = %b, z1 = %b, err = %b",
           x1, y, z1, err);

initial                      //define clock
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0    rst_n = 1'b0;        //reset to state_a
        x1 = 1'b0;

    #5    rst_n = 1'b1;        //deassert reset

    x1 = 1'b0;@ (posedge clk) //go to state_a (00)
    x1 = 1'b0;@ (posedge clk) //go to state_a (00)
    x1 = 1'b1;@ (posedge clk) //go to state_b (01)
                                //assert z1
    x1 = 1'b1;@ (posedge clk) //go to state_b (01)
                                //assert z1
    x1 = 1'b1;@ (posedge clk) //go to state_b (01)
                                //assert z1

    x1 = 1'b1;@ (posedge clk) //go to state_b (01)
                                //assert z1
    x1 = 1'b1;@ (posedge clk) //go to state_b (01)
                                //assert z1

    x1 = 1'b0;@ (posedge clk) //go to state_c (11)
    x1 = 1'b0;@ (posedge clk) //go to state_c (11)
    x1 = 1'b0;@ (posedge clk) //go to state_c (11)
    x1 = 1'b0;@ (posedge clk) //go to state_c (11)

    x1 = 1'b1;@ (posedge clk) //go to state_d (11)
                                //assert err

    #20    $stop;
end                                //continued on next page

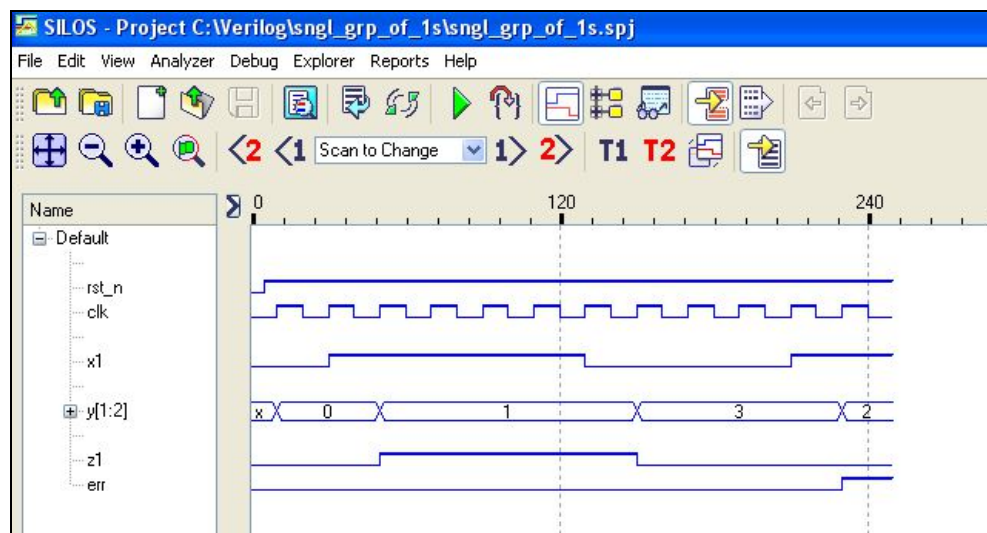
```



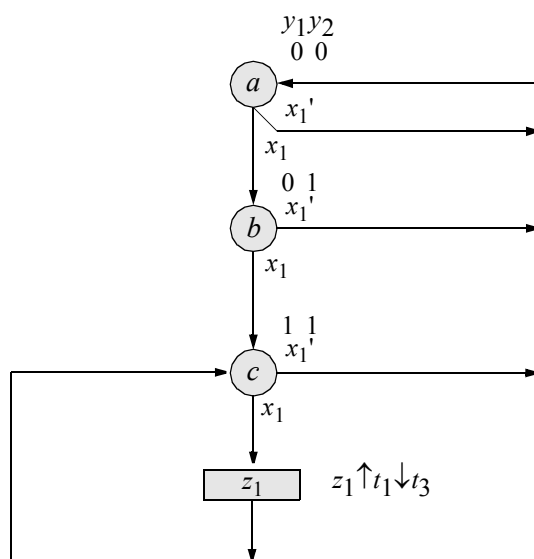
```
//instantiate the module into the test bench
sngl_grp_of_1s inst1 (
  .rst_n(rst_n),
  .clk(clk),
  .x1(x1),
  .y(y),
  .z1(z1),
  .err(err)
);

endmodule
```

```
x1 = 0, state = xx, z1 = 0, err = 0
x1 = 0, state = 00, z1 = 0, err = 0
x1 = 1, state = 00, z1 = 0, err = 0
x1 = 1, state = 01, z1 = 1, err = 0
x1 = 0, state = 01, z1 = 1, err = 0
x1 = 0, state = 11, z1 = 0, err = 0
x1 = 1, state = 11, z1 = 0, err = 0
x1 = 1, state = 10, z1 = 0, err = 1
```



- 2.17 Generate a reduced state diagram for a Mealy machine which produces a conditional output  $z_1$  whenever a serial data line  $x_1$  contains a sequence of three or more consecutive 1s. Use behavioral modeling with the **case** statement. Show the behavioral design module, the test bench module, the outputs, and the waveforms.



```
//behavioral 3 or more consecutive 1s

module consec_1s_bh (rst_n, clk, x1, y, z1);

//define inputs and output
input rst_n, clk, x1;
output [1:2] y;
output z1;

//variables are reg in always
reg [1:2] y, next_state;
wire z1;

//assign state codes, parameter defines a constant
parameter state_a = 2'b00,
           state_b = 2'b01,
           state_c = 2'b11;

always @(posedge clk) begin
    if (rst_n == 0) next_state <- state_a;
    else begin
        case (next_state)
            state_a: if (x1 == 0) next_state <- state_a;
                     else next_state <- state_b;
            state_b: if (x1 == 0) next_state <- state_a;
                     else next_state <- state_c;
            state_c: if (x1 == 0) next_state <- state_a;
                     else next_state <- state_c;
        endcase
        y <- next_state;
        z1 <- (next_state == state_c & x1);
    end
end
```

//continued on next page

```

//set next state
always @ (posedge clk)
begin
    if (~rst_n)           //reset = 1'b0
        y <= state_a;     //y <= state_a (00)
    else
        y = next_state;
end

//determine output
assign z1 = ((y[1]) && (y[2]));

//determine next state
always @ (y or x1)
begin
    case (y)
        state_a:
            if (~x1)
                next_state = state_a;
            else
                next_state = state_b;

        state_b:
            if (~x1)
                next_state = state_a;
            else
                next_state = state_c;

        state_c:
            if (~x1)
                next_state = state_a;
            else
                next_state = state_c;

        default: next_state = state_a;
    endcase
end

endmodule

```

```

//test bench for 3 or more consecutive 1s
module consec_1s_bh_tb;

reg rst_n, clk, x1;          //inputs are reg for test bench
wire [1:2] y;                //outputs are wire for test bench
wire z1;

initial                      //display variables
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

initial                      //define clock
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

initial                      //define input sequence
begin
    #0 rst_n = 1'b0;          //reset to state_a (00)
    x1 = 1'b0;
    #5 rst_n = 1'b1;          //deassert reset
    //-----
    x1 = 1'b0; @ (posedge clk) //go to state_a (00)
    x1 = 1'b1; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b0; @ (posedge clk) //go to state_a (00)
    //-----
    x1 = 1'b1; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b0; @ (posedge clk) //go to state_a (00)
    #20 $stop;
end

//instantiate the module into the test bench
consec_1s_bh inst1 (rst_n, clk, x1, y, z1);

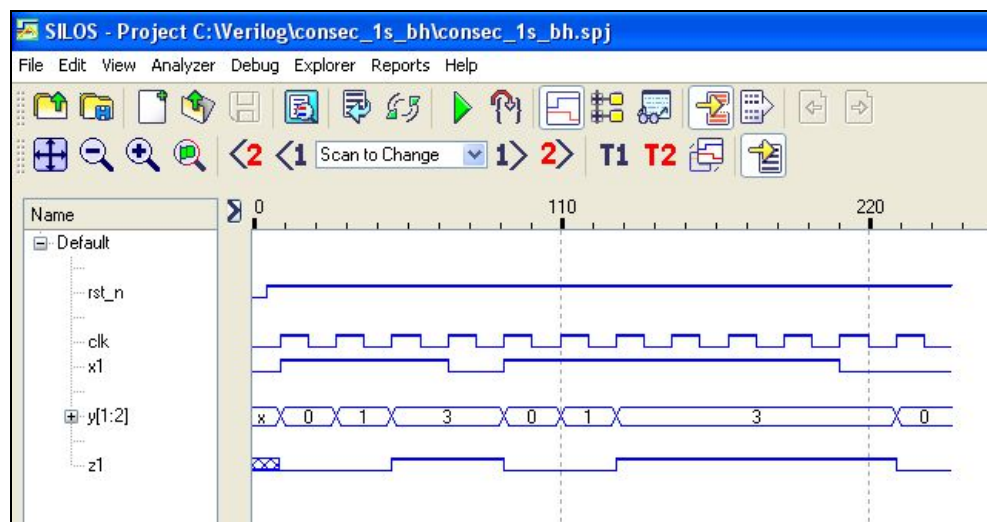
endmodule

```

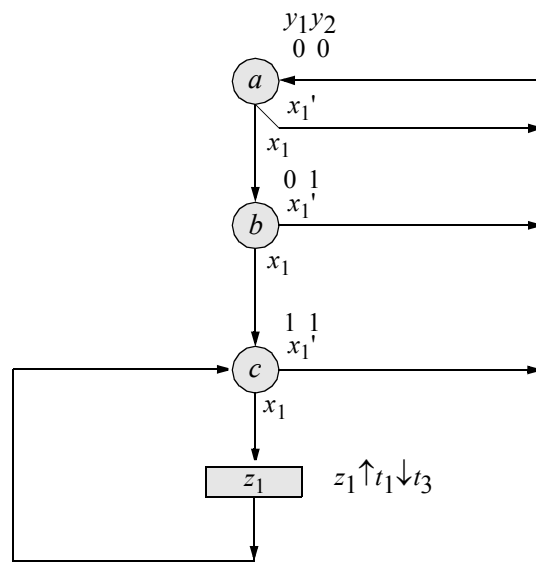
```

x1 = 0, state = xx, z1 = x
x1 = 1, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 1
x1 = 0, state = 11, z1 = 1
x1 = 1, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 1
x1 = 0, state = 11, z1 = 1
x1 = 0, state = 00, z1 = 0

```



- 2.18 Repeat Problem 2.17 for a Mealy machine which produces a conditional output  $z_1$  whenever a serial data line  $x_1$  contains a sequence of three or more consecutive 1s. Use structural modeling with built-in primitives and  $JK$  flip-flops. Show the Karnaugh maps and equations, the structural design module, the test bench module, the outputs, and the waveforms.



Present state		Input	Next state		Flip-flop inputs				Present output
$y_1$	$y_2$	$x_1$	$y_1$	$y_2$	$Jy_1$	$Ky_1$	$Jy_2$	$Ky_2$	$z_1$
0	0	0	0	0	0	–	0	–	0
0	0	1	0	1	0	–	1	–	0
0	1	0	0	0	0	–	–	1	0
0	1	1	1	1	1	–	–	0	0
1	0	0	–	–	–	–	–	–	–
1	0	1	–	–	–	–	–	–	–
1	1	0	0	0	–	1	–	1	0
1	1	1	1	1	–	0	–	0	1

		$y_2$	
		0	1
$y_1$	0	<sup>0</sup> 0	<sup>1</sup> $x_1$
	1	<sup>2</sup> —	<sup>3</sup> —

 $Jy_1$ 

$$Jy_1 = y_2x_1$$

		$y_2$	
		0	1
$y_1$	0	<sup>0</sup> —	<sup>1</sup> —
	1	<sup>2</sup> —	<sup>3</sup> $x_1'$

 $Ky_1$ 

$$Ky_1 = x_1'$$

		$y_2$	
		0	1
$y_1$	0	<sup>0</sup> $x_1$	<sup>1</sup> —
	1	<sup>2</sup> —	<sup>3</sup> —

 $Jy_2$ 

$$Jy_2 = x_1$$

		$y_2$	
		0	1
$y_1$	0	<sup>0</sup> —	<sup>1</sup> $x_1'$
	1	<sup>2</sup> —	<sup>3</sup> $x_1'$

 $Ky_2$ 

$$Ky_2 = x_1'$$

```
//structural 3 or more consecutive 1s

module consec_1s_bip_jk (rst_n, clk, x1, y, z1);

//define inputs and output
input rst_n, clk, x1;
output [1:2] y;
output z1;

//define internal nets
wire net1;

//instantiate the logic for flip-flop y[1]
and (net1, y[2], x1);
jkff_neg_clk inst1 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net1),
    .k(~x1),
    .q(y[1])
);
```

//continued on next page

```
//instantiate the logic for flip-flop y[2]
jkff_neg_clk inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(x1),
    .k(~x1),
    .q(y[2])
);

and (z1, y[1], y[2]);

endmodule
```

```
//test bench for 3 or more consecutive 1s
module consec_1s_bip_jk_tb;

reg rst_n, clk, x1;          //inputs are reg for test bench
wire [1:2] y;                //outputs are wire for test bench
wire z1;

initial                      //display variables
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

initial                      //define clock
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0 rst_n = 1'b0;          //reset to state_a (00)
    x1 = 1'b0;

    #5 rst_n = 1'b1;          //deassert reset

    //-----
    x1 = 1'b0; @ (posedge clk) //go to state_a (00)
    x1 = 1'b1; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                                //assert z1 at t2
    x1 = 1'b0; @ (posedge clk) //go to state_a (00)
                                //continued on next page
```



```
//-----
x1 = 1'b1; @ (posedge clk) //go to state_b (01)
x1 = 1'b1; @ (posedge clk) //go to state_c (11)
x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                        //assert z1 at t2
x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                        //assert z1 at t2
x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                        //assert z1 at t2
x1 = 1'b1; @ (posedge clk) //go to state_c (11)
                        //assert z1 at t2
x1 = 1'b0; @ (posedge clk) //go to state_a (00)

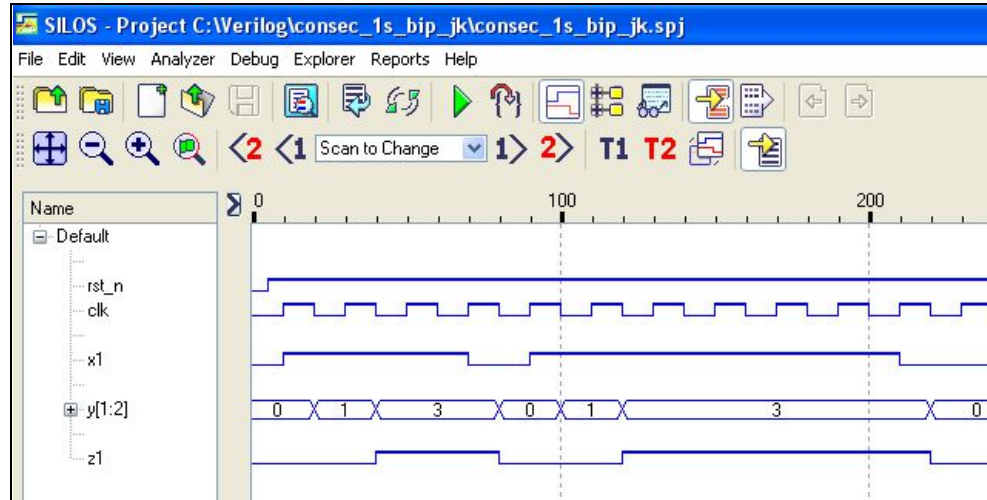
//-----

#20    $stop;
end

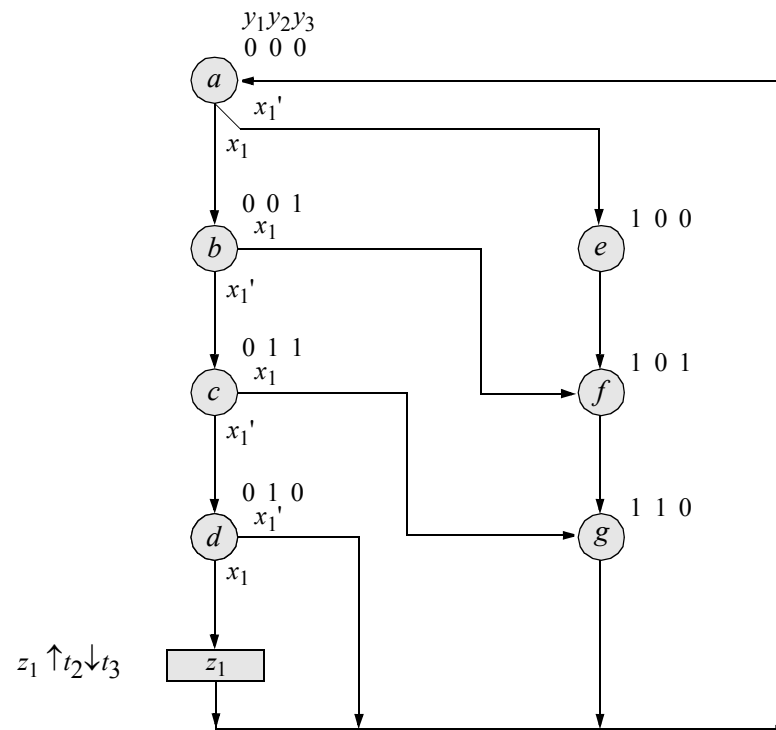
//instantiate the module into the test bench
consec_ls_bip_jk inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);

endmodule
```

```
x1 = 0, state = 00, z1 = 0
x1 = 1, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 1
x1 = 0, state = 11, z1 = 1
x1 = 0, state = 00, z1 = 0
x1 = 1, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 1
x1 = 0, state = 11, z1 = 1
x1 = 0, state = 00, z1 = 0
```



- 2.19 Generate a reduced state diagram for a Mealy machine which detects a 4-bit word of 1001 on a serial input line  $x_1$ . If a correct sequence is detected, then a conditional output  $z_1$  is generated. There is no spacing between words. There is also no overlapping of words. Assert  $z_1$  from time  $t_2$  to time  $t_3$ . Obtain the behavioral design module, the test bench module, the outputs, and the waveforms.



```

//behavioral detect 1001
module detect_1001_bh (rst_n, clk, x1, y, z1);

//define inputs and output
input rst_n, clk, x1;
output [1:3] y;
output z1;

reg [1:3] y, next_state;    //variables are reg in always
wire z1;

//assign state codes, parameter defines a constant
parameter state_a = 3'b000,
           state_b = 3'b001,
           state_c = 3'b011,
           state_d = 3'b010,
           state_e = 3'b100,
           state_f = 3'b101,
           state_g = 3'b110;

//set next state
always @ (posedge clk)
begin
    if (~rst_n)        //reset = 1'b0
        y <= state_a;  //y <= state_a (000)
    else
        y = next_state;
end

//determine output
assign z1 = ((~y[1]) && (y[2]) && (~y[3]) && x1 && ~clk);

//determine next state
always @ (y or x1)
begin
    case (y)
        state_a:
            if (x1)
                next_state = state_b;
            else
                next_state = state_e;

        state_b:
            if (~x1)
                next_state = state_c;
            else
                next_state = state_f;
    endcase
end

```

//continued on next page

```

state_c:
    if (~x1)
        next_state = state_d;
    else
        next_state = state_g;

state_d:
    if (x1)
        next_state = state_a;
    else
        next_state = state_a;

state_e: next_state = state_f;

state_f: next_state = state_g;

state_g: next_state = state_a;

default: next_state = state_a;

endcase
end
endmodule

```

```

//test bench to detect 1001

module detect_1001_bh_tb;

reg rst_n, clk, x1;    //inputs are reg for test bench
wire [1:3] y;          //outputs are wire for test bench
wire z1;

//display variables
initial
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//continued on next page

```

```

//define input sequence
initial
begin
    #0 rst_n = 1'b0;                //reset to state_a (000)
    x1 = 1'b0;
    #5 rst_n = 1'b1;                //deassert reset
//-----
        @ (posedge clk)            //go to state_a (000)
    x1 = 1'b1; @ (posedge clk)      //go to state_b (001)
    x1 = 1'b0; @ (posedge clk)      //go to state_c (011)
    x1 = 1'b0; @ (posedge clk)      //go to state_d (010)
    x1 = 1'b1; @ (posedge clk)      //go to state_a (000)
//assert z1 at t2
//-----
    x1 = 1'b0; @ (posedge clk)      //go to state_e (100)
    x1 = $random; @ (posedge clk)   //go to state_f (101)
    x1 = $random; @ (posedge clk)   //go to state_g (110)
    x1 = $random; @ (posedge clk)   //go to state_a (000)

    #20 $stop;
end

//instantiate the module into the test bench
detect_1001_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);

endmodule

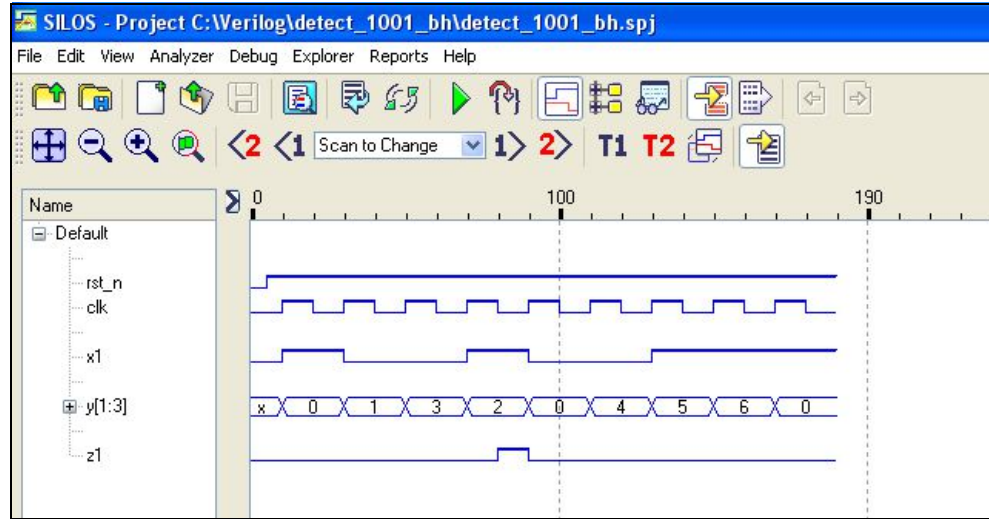
```

```

x1 = 0, state = xxx, z1 = 0
x1 = 1, state = 000, z1 = 0
x1 = 0, state = 001, z1 = 0
x1 = 0, state = 011, z1 = 0
x1 = 1, state = 010, z1 = 0
x1 = 1, state = 010, z1 = 1

x1 = 0, state = 000, z1 = 0
x1 = 0, state = 100, z1 = 0
x1 = 1, state = 101, z1 = 0
x1 = 1, state = 110, z1 = 0
x1 = 1, state = 000, z1 = 0

```



- 2.20 Repeat Problem 2.19 for a Mealy machine which detects a 4-bit word of 1001 on a serial input line  $x_1$ . If a correct sequence is detected, then a conditional output  $z_1$  is generated. There is no spacing between words. There is also no overlapping of words. Assert  $z_1$  from time  $t_2$  to time  $t_3$ . Obtain the structural design module using instantiated logic gates and  $D$  flip-flops, the test bench module, the outputs, and the waveforms.

		$y_2y_3$			
		00	01	11	10
$y_1$	0	$x_1'$	$x_1$	$x_1$	0
	1	1	1	—	0

$$Dy_1 = y_3x_1 + y_2'y_3'x_1' + y_1y_2'$$

		$y_2y_3$			
		00	01	11	10
$y_1$	0	0	$x_1'$	1	0
	1	0	1	—	0

$$Dy_2 = y_3x_1' + y_1y_3 + y_2y_3$$

		$y_2y_3$			
		00	01	11	10
$y_1$	0	$x_1$	1	0	0
	1	1	0	—	0

$$Dy_3 = y_1'y_2'x_1 + y_1'y_2'y_3 + y_1y_2'y_3'$$

$Dy_1 = y_3x_1 + y_2'y_3'x_1' + y_1y_2'$	$Dy_2 = y_3x_1' + y_1y_3 + y_2y_3$
$y_3x_1$ inst1, net1	$y_3x_1'$ inst6, net6
$y_2'y_3'x_1'$ inst2, net2	$y_1y_3$ inst7, net7
$y_1y_2'$ inst3, net3	$y_2y_3$ inst8, net8
OR          inst4, net4	OR          inst9, net9
$Dff$ inst5	$Dff$ inst10

$Dy_3 = y_1'y_2'x_1 + y_1'y_2'y_3 + y_1y_2'y_3'$   
 $y_1'y_2'x_1$    inst11, net11  
 $y_1'y_2'y_3$    inst12, net12  
 $y_1y_2'y_3'$    inst13, net13  
 OR          inst14, net14  
 $Dff$           inst15

```
//structural to detect 1001 using D flip-flops

module detect_1001_dff (rst_n, clk, x1, y, z1);

//define inputs and output
input rst_n, clk, x1;
output [1:3] y;
output z1;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7, net8, net9,
      net10, net11, net12, net13, net14;

//-----
//instantiate the logic for flip-flop y[1]
and2_df inst1 (
    .x1(y[3]),
    .x2(x1),
    .z1(net1)
);

and3_df inst2 (
    .x1(~y[2]),
    .x2(~y[3]),
    .x3(~x1),
    .z1(net2)
);

//continued on next page
```

```

and2_df inst3 (
    .x1(y[1]),
    .x2(~y[2]),
    .z1(net3)
);

or3_df inst4 (
    .x1(net1),
    .x2(net2),
    .x3(net3),
    .z1(net4)
);

d_ff_bh inst5 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net4),
    .q(y[1])
);

//-----
//instantiate the logic for flip-flop y[2]
and2_df inst6 (
    .x1(y[3]),
    .x2(~x1),
    .z1(net6)
);

and2_df inst7 (
    .x1(y[1]),
    .x2(y[3]),
    .z1(net7)
);

and2_df inst8 (
    .x1(y[2]),
    .x2(y[3]),
    .z1(net8)
);

or3_df inst9 (
    .x1(net6),
    .x2(net7),
    .x3(net8),
    .z1(net9)
);

//continued on next page

```



```

d_ff_bh inst10 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net9),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and3_df inst11 (
    .x1(~y[1]),
    .x2(~y[2]),
    .x3(x1),
    .z1(net11)
);

and3_df inst12 (
    .x1(~y[1]),
    .x2(~y[2]),
    .x3(y[3]),
    .z1(net12)
);

and3_df inst13 (
    .x1(y[1]),
    .x2(~y[2]),
    .x3(~y[3]),
    .z1(net13)
);

or3_df inst14 (
    .x1(net11),
    .x2(net12),
    .x3(net13),
    .z1(net14)
);

d_ff_bh inst15 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net14),
    .q(y[3])
);

//-----
//define output
assign z1 = ((~y[1]) && (y[2]) && (~y[3]) && x1 && ~clk);

endmodule

```

```

//test bench to detect 1001
module detect_1001_dff_tb;

reg rst_n, clk, x1;          //inputs are reg for test bench
wire [1:3] y;                //outputs are wire for test bench
wire z1;

//display variables
initial
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0 rst_n = 1'b0;          //reset to state_a (000)
    x1 = 1'b0;
    #5 rst_n = 1'b1;          //deassert reset
    //-----
    x1 = 1'b1; @ (posedge clk) //go to state_b (001)
    x1 = 1'b0; @ (posedge clk) //go to state_c (011)
    x1 = 1'b0; @ (posedge clk) //go to state_d (010)
    x1 = 1'b1; @ (posedge clk) //go to state_a (000)
                                //assert z1 at t2
    //-----
    x1 = 1'b0; @ (posedge clk) //go to state_e (100)
    x1 = $random; @ (posedge clk) //go to state_f (101)
    x1 = $random; @ (posedge clk) //go to state_g (110)
    x1 = $random; @ (posedge clk) //go to state_a (000)
    #20 $stop;
end

//instantiate the module into the test bench
detect_1001_dff inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);

endmodule

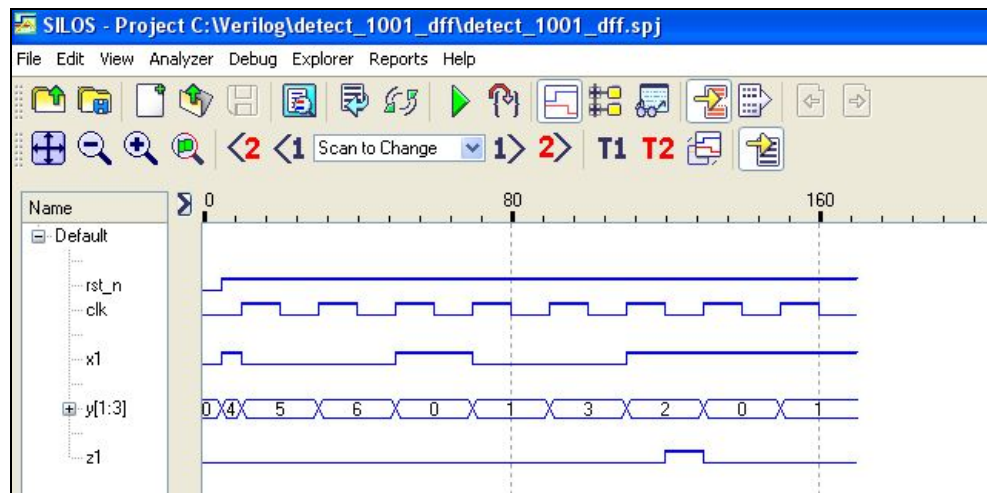
```

```

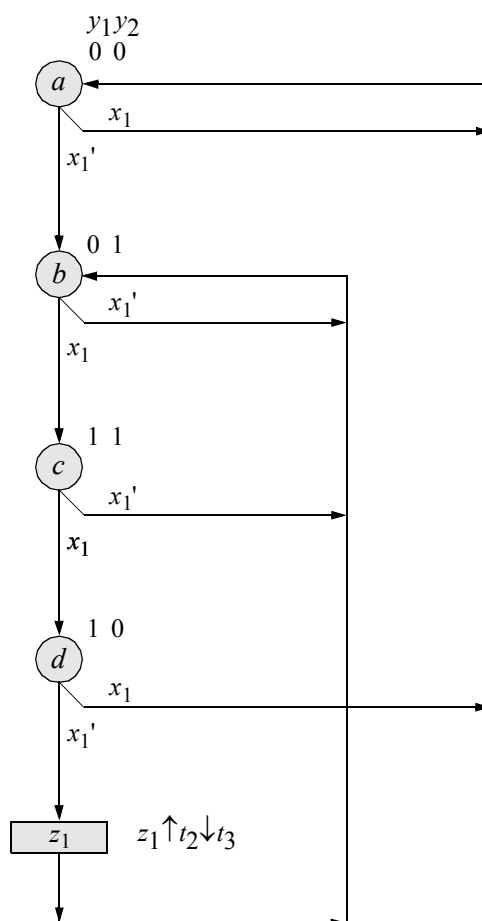
x1 = 0, state = 000, z1 = 0
x1 = 1, state = 100, z1 = 0
x1 = 0, state = 101, z1 = 0
x1 = 0, state = 110, z1 = 0
x1 = 1, state = 000, z1 = 0
x1 = 0, state = 001, z1 = 0
x1 = 0, state = 011, z1 = 0
x1 = 1, state = 010, z1 = 0
x1 = 1, state = 010, z1 = 1

x1 = 1, state = 000, z1 = 0
x1 = 1, state = 001, z1 = 0

```



- 2.21 Obtain the state diagram for a Mealy synchronous sequential machine to detect a sequence of 0110 on a serial input line  $x_1$ . Overlapping sequences are valid. Output  $z_1$  will be asserted during the last half of the clock cycle in the state in which the final 0 is detected. Then obtain the behavioral module, the test bench module, the outputs, and the waveforms. In the test bench, check different paths in the state diagram for correct functional operation and include valid overlapping sequences.



```
//behavioral detect 0110
module detect_0110_bh (rst_n, clk, x1, y, z1);

//define inputs and output
input rst_n, clk, x1;
output [1:2] y;
output z1;

reg [1:2] y, next_state; //variables are reg in always
wire z1;

//assign state codes; parameter defines a constant
parameter state_a = 2'b00,
            state_b = 2'b01,
            state_c = 2'b11,
            state_d = 2'b10;

//continued on next page
```

```

//set next state
always @ (posedge clk)
begin
    if (~rst_n)          //reset = 1'b0
        y <= state_a;    //y <= state_a (00)
    else
        y = next_state;
end

//define output
assign z1 = ((y[1]) && (~y[2]) && ~x1 && ~clk);

//determine next state
always @ (y or x1)
begin
    case (y)
        state_a:
            if (~x1)
                next_state = state_b;
            else
                next_state = state_a;

        state_b:
            if (x1)
                next_state = state_c;
            else
                next_state = state_b;

        state_c:
            if (x1)
                next_state = state_d;
            else
                next_state = state_b;

        state_d:
            if (~x1)
                next_state = state_b;
            else
                next_state = state_a;

        default: next_state = state_a;

    endcase
end

endmodule

```

```

//test bench to detect 0110

module detect_0110_bh_tb;

reg rst_n, clk, x1;          //inputs are reg for test bench
wire [1:2] y;                //outputs are wire for test bench
wire z1;

initial                      //display variables
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

initial                      //define clock
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

initial                      //define input sequence
begin
    #0   rst_n = 1'b0;
        x1 = 1'b0;
    #5   rst_n = 1'b1;
//-----
        @ (posedge clk) //go to state_a (00)
    x1 = 1'b0; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b1; @ (posedge clk) //go to state_d (10)
    x1 = 1'b0; @ (posedge clk) //assert z1; go to state_b (010)
//-----
    x1 = 1'b0; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b0; @ (posedge clk) //go to state_b (01)
    x1 = 1'b1; @ (posedge clk) //go to state_c (11)
    x1 = 1'b1; @ (posedge clk) //go to state_d (10)
    x1 = 1'b0; @ (posedge clk) //assert z1; go to state_b (010)
//-----
    #20 $stop;
end

//instantiate the module into the test bench
detect_0110_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);
endmodule

```

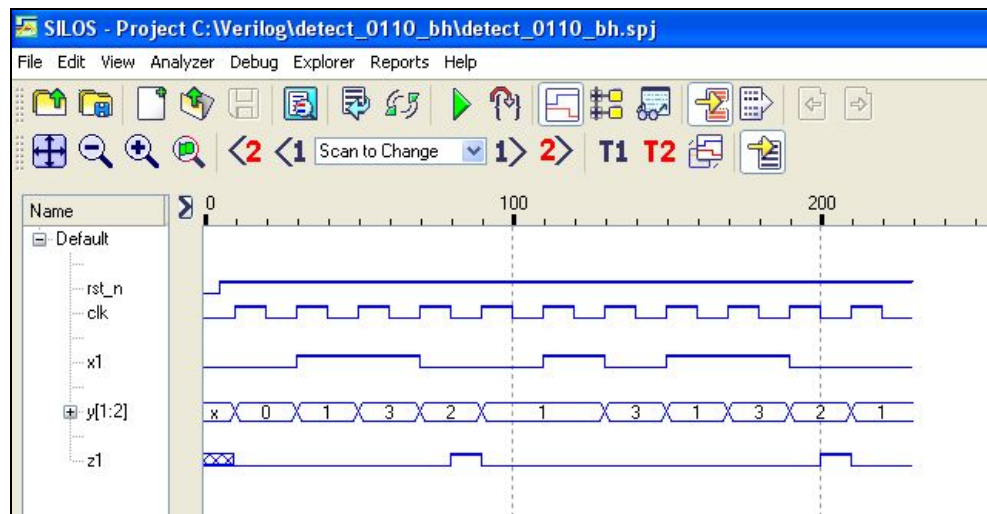
```

x1 = 0, state = xx, z1 = x
x1 = 0, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 0
x1 = 0, state = 10, z1 = 0
x1 = 0, state = 10, z1 = 1

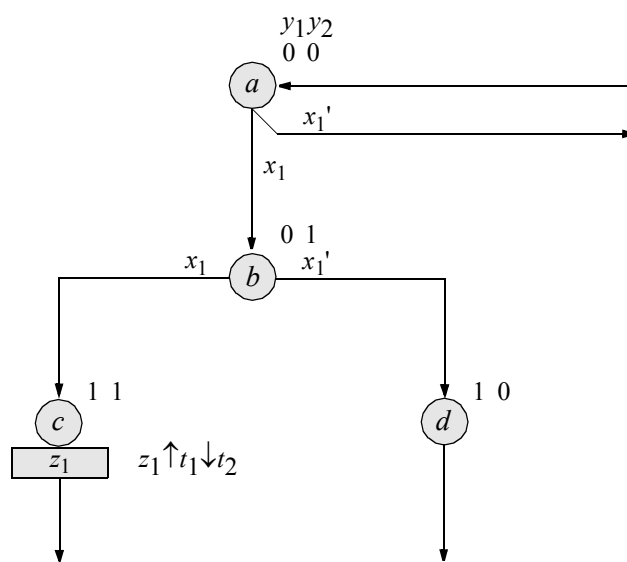
x1 = 0, state = 01, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 0, state = 11, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 1, state = 11, z1 = 0
x1 = 0, state = 10, z1 = 0
x1 = 0, state = 10, z1 = 1

x1 = 0, state = 01, z1 = 0

```



- 2.22 In the Moore machine shown on the next page, a glitch is possible on output  $z_1$  for a transition from state  $b$  to state  $d$  if flip-flop  $y_1$  sets before flip-flop  $y_2$  resets. The glitch can be avoided if the output is delayed by a time increment so that the output is asserted only after the machine has stabilized. The delay can be sufficiently small so that the assertion and deassertion of output  $z_1$  can still be considered to be  $\uparrow t_1 \downarrow t_2$ . Obtain the behavioral design module with an output delay of two time units, the test bench module, the outputs, and the waveforms.



```

//behavioral delay clock to avoid glitches
module moore_no_glitch (rst_n, clk, x1, y, z1);

input rst_n, clk, x1;          //define inputs and outputs
output [1:2] y;
output z1;

reg [1:2] y, next_state;      //variables are reg in always
wire z1;

//assign state codes; parameter defines a constant
parameter state_a = 2'b00,
            state_b = 2'b01,
            state_c = 2'b11,
            state_d = 2'b10;

always @ (posedge clk)        //set next state
begin
    if (~rst_n)                //rst_n = 1'b0
        y = state_a;           //y <= state_a (00)
    else
        y <= next_state;
    end

//define output
assign #2 z1 = y[1] && y[2] && clk;
//continued on next page

```



```

//determine next state
always @ (y or x1)
begin
    case (y)
        state_a:
            if (x1)
                next_state = state_b;
            else
                next_state = state_a;

        state_b:
            if (x1)
                next_state = state_c;
            else
                next_state = state_d;

        state_c: next_state = state_a;

        state_d: next_state = state_a;

        default: next_state = state_a;

    endcase
end

endmodule

```

```

//test bench for moore delay clock to avoid glitches
module moore_no_glitch_tb;

reg rst_n, clk, x1;          //inputs are reg for test bench
wire [1:2] y;                //outputs are wire for test bench
wire z1;

//display variables
initial
$monitor ("x1 = %b, state = %b, z1 = %b", x1, y, z1);

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//continued on next page

```

```

//define input sequence
initial
begin
    #0 rst_n = 1'b0;           //reset to state_a (00)
    x1 = 1'b0;

    #5 rst_n = 1'b1;           //deassert reset
//-----
    x1 = 1'b0;@ (posedge clk)   //go to state_a (00)
    x1 = 1'b1;@ (posedge clk)   //go to state_b (01)
    x1 = 1'b1;@ (posedge clk)   //go to state_c (11)
                                //assert z1 (t1 -- t2)
    x1 = $random;@ (posedge clk) //go to state_a (00)
//-----
    x1 = 1'b1;@ (posedge clk)   //go to state_b (01)
    x1 = 1'b0;@ (posedge clk)   //go to state_d (10)
    x1 = $random;@ (posedge clk) //go to state_a (00)
//-----
    #10 $stop;
end

//instantiate the module into the test bench
moore_no_glitch inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1)
);

endmodule

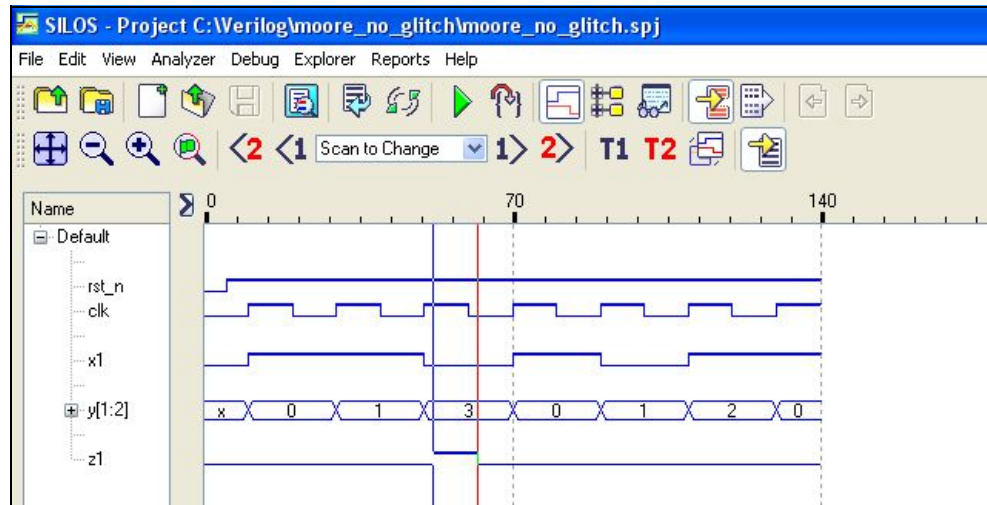
```

```

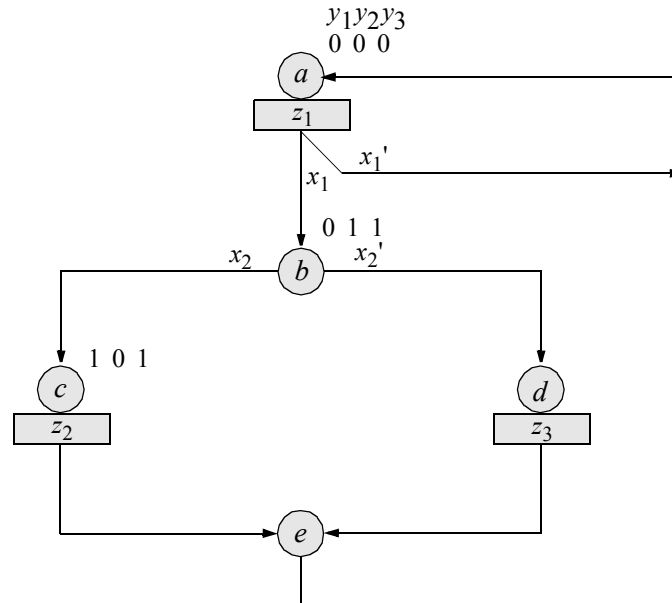
x1 = 0, state = xx, z1 = 0
x1 = 1, state = 00, z1 = 0
x1 = 1, state = 01, z1 = 0
x1 = 0, state = 11, z1 = 0
x1 = 0, state = 11, z1 = 1

x1 = 0, state = 11, z1 = 0
x1 = 1, state = 00, z1 = 0
x1 = 0, state = 01, z1 = 0
x1 = 1, state = 10, z1 = 0
x1 = 1, state = 00, z1 = 0

```



- 2.23 Select state codes for states  $d$  and  $e$  for the Moore machine shown below so that there will be no output glitches. Consider all state transitions. Then design a structural module for the Moore machine using built-in primitives and  $D$  flip-flops. Obtain the test bench module, the outputs, and the waveforms.



State  $d$ :  $y_1y_2y_3 = 110$ . State  $e$ :  $y_1y_2y_3 = 100$ .  
 Unused states:  $y_1y_2y_3 = 001, 010, 111$ .

		$y_2y_3$			
$y_1$		0 0	0 1	1 1	1 0
	0	<sup>0</sup> 0	<sup>1</sup> —	<sup>3</sup> 1	<sup>2</sup> —
	1	<sup>4</sup> 0	<sup>5</sup> 1	<sup>7</sup> —	<sup>6</sup> 1

 $Dy_1$ 

$$Dy_1 = y_3 + y_2$$

		$y_2y_3$			
$y_1$		0 0	0 1	1 1	1 0
	0	<sup>0</sup> $x_1$	<sup>1</sup> —	<sup>3</sup> $x_2'$	<sup>2</sup> —
	1	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> —	<sup>6</sup> 0

 $Dy_2$ 

$$Dy_2 = y_1'y_2'x_1 + y_2y_3x_2'$$

		$y_2y_3$			
$y_1$		0 0	0 1	1 1	1 0
	0	<sup>0</sup> $x_1$	<sup>1</sup> —	<sup>3</sup> $x_2$	<sup>2</sup> —
	1	<sup>4</sup> 0	<sup>5</sup> 0	<sup>7</sup> —	<sup>6</sup> 0

 $Dy_3$ 

$$Dy_3 = y_1'y_2'x_1 + y_2y_3x_2$$

```
//structural moore with no glitches
module moore_no_glitch_d (rst_n, clk, x1, x2, y, z1, z2, z3);

//define inputs and outputs
input rst_n, clk, x1, x2;
output [1:3] y;
output z1, z2, z3;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7;

//-----
//instantiate the logic for flip-flop y[1]
or (net1, y[3], y[2]);

d_ff_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net1),
    .q(y[1])
);

//continued on next page
```

```

//-----
//instantiate the logic for flip-flop y[2]
and (net2, ~y[1], ~y[2], x1);
and (net3, y[2], y[3], ~x2);
or (net4, net2, net3);

d_ff_bh inst2 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net4),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and (net5, ~y[1], ~y[2], x1);
and (net6, y[2], y[3], x2);
or (net7, net5, net6);

d_ff_bh inst3 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net7),
    .q(y[3])
);

//define the outputs
assign z1 = ~y[1] && ~y[2] && ~y[3];
assign z2 = y[1] && ~y[2] && y[3];
assign z3 = y[1] && y[2] && ~y[3];

endmodule

```

```

//test bench for moore no glitch machine

module moore_no_glitch_d_tb;

reg rst_n, clk, x1, x2; //inputs are reg for test bench
wire [1:3] y; //outputs are wire for test bench
wire z1, z2, z3;

//display variables
initial
$monitor ("x1 x1 = %b, state = %b, z1 = %b, z2 = %b, z3 = %b",
           {x1, x2}, y, z1, z2, z3);
//continued on next page

```

```

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0 rst_n = 1'b0;          //reset to state_a (00)
    x1 = 1'b0;
    x2 = 1'b0;

    #5 rst_n = 1'b1;          //deassert reset
    //-----
    x1 = 1'b0; x2 = $random; @ (posedge clk) //assert z1
    x1 = 1'b1; x2 = $random; @ (posedge clk) //go to state_b
    x1 = $random; x2 = 1'b1; @ (posedge clk) //go to state_c
                                         //assert z2
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_e
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_a
                                         //assert z1
    //-----
    x1 = 1'b1; x2 = $random; @ (posedge clk) //go to state_b
    x1 = $random; x2 = 1'b0; @ (posedge clk) //go to state_d
                                         //assert z3
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_a
    //-----

    #10 $stop;
end

//instantiate the module into the test bench
moore_no_glitch_d inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .x2(x2),
    .y(y),
    .z1(z1),
    .z2(z2),
    .z3(z3)
);

endmodule

```

```

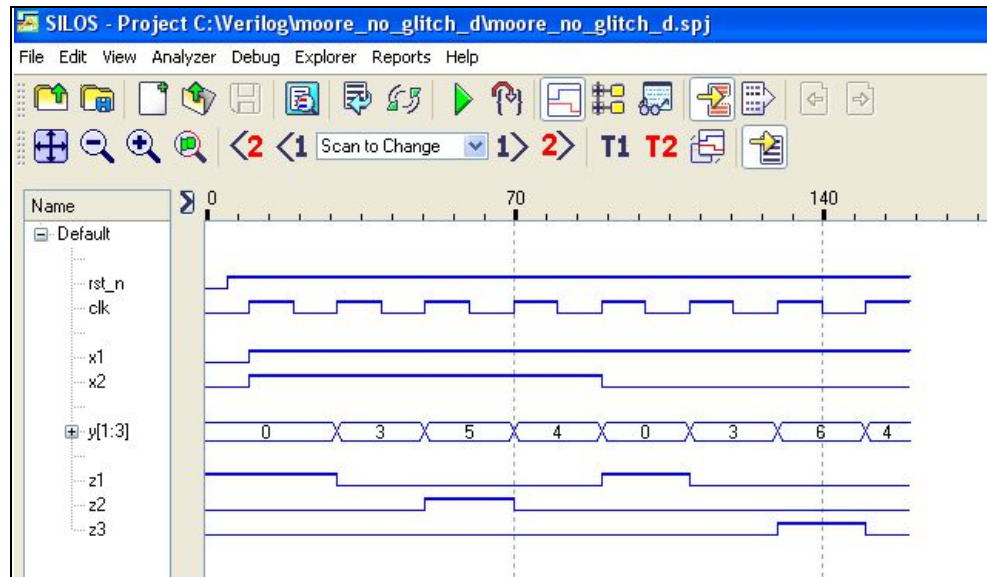
x1 x2 = 00, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 11, state = 000, z1 = 1, z2 = 0, z3 = 0

x1 x2 = 11, state = 011, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 11, state = 101, z1 = 0, z2 = 1, z3 = 0

x1 x2 = 11, state = 100, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 10, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 10, state = 011, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 10, state = 110, z1 = 0, z2 = 0, z3 = 1

x1 x2 = 10, state = 100, z1 = 0, z2 = 0, z3 = 0

```



- 2.24 This repeats Problem 2.23 using the same selected state codes for states *d* and *e* for the Moore machine shown in Problem 2.23 so that there will be no output glitches. Design a structural module for the Moore machine using built-in primitives and *JK* flip-flops. Obtain the test bench module, the outputs, and the waveforms

Present state			Input		Next state			Flip-flop inputs				Present output		
$y_1$	$y_2$	$y_3$	$x_1$	$x_2$	$y_1$	$y_2$	$y_3$	$Jy_1$	$Ky_1$	$Jy_2$	$Ky_2$	$Jy_3$	$Ky_3$	$z_1$ $z_2$ $z_3$
0	0	0	0	–	0	0	0	0	–	0	–	0	–	1 0 0
0	0	0	1	–	0	1	1	0	–	1	–	1	–	1 0 0
0	0	1	–	–	–	–	–	–	–	–	–	–	–	– – –
0	0	1	–	–	–	–	–	–	–	–	–	–	–	– – –
0	1	0	–	–	–	–	–	–	–	–	–	–	–	– – –
0	1	0	–	–	–	–	–	–	–	–	–	–	–	– – –
0	1	1	–	0	1	1	0	1	–	–	0	–	1	0 0 0
0	1	1	–	1	1	0	1	1	–	–	1	–	0	0 0 0
1	0	0	–	–	0	0	0	–	1	0	–	0	–	0 0 0
1	0	0	–	–	0	0	0	–	1	0	–	0	–	0 0 0
1	0	1	–	–	1	0	0	–	0	0	–	–	1	0 1 0
1	0	1	–	–	1	0	0	–	0	0	–	–	1	0 1 0
1	1	0	–	–	1	0	0	–	0	–	1	0	–	0 0 1
1	1	0	–	–	1	0	0	–	0	–	1	0	–	0 0 1
1	1	1	–	–	–	–	–	–	–	–	–	–	–	– – –
1	1	1	–	–	–	–	–	–	–	–	–	–	–	– – –

		$y_2y_3$			
		00	01	11	10
$y_1$	0	0 <sup>0</sup> 0	– <sup>1</sup>	1 <sup>3</sup> 1	– <sup>2</sup>
	1	– <sup>4</sup>	– <sup>5</sup>	– <sup>7</sup>	– <sup>6</sup>

 $Jy_1$ 

$$Jy_1 = y_2$$

		$y_2y_3$			
		00	01	11	10
$y_1$	0	– <sup>0</sup>	– <sup>1</sup>	– <sup>3</sup>	– <sup>2</sup>
	1	1 <sup>4</sup> 1	0 <sup>5</sup> 0	– <sup>7</sup>	0 <sup>6</sup> 0

 $Ky_1$ 

$$Ky_1 = y_2'y_3'$$



		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 $x_1$	1 —	3 —	2 —
	1	4 0	5 0	7 —	6 —

 $Jy_2$ 

$$Jy_2 = y_1'x_1$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 0	1 —	3 $x_2$	2 —
	1	4 —	5 —	7 —	6 1

 $Ky_2$ 

$$Ky_2 = y_3x_2 + y_1$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 $x_1$	1 —	3 —	2 —
	1	4 0	5 —	7 —	6 0

 $Jy_3$ 

$$Jy_3 = y_1'x_1$$

		$y_2y_3$			
		0 0	0 1	1 1	1 0
$y_1$	0	0 —	1 —	3 $x_2'$	2 —
	1	4 —	5 1	7 —	6 —

 $Ky_3$ 

$$Ky_3 = x_2' + y_2'$$

```
//structural moore with no glitches using JK flip-flops
module moore_no_glitch_jk (rst_n, clk, x1, x2, y, z1, z2, z3);

//define inputs and outputs
input rst_n, clk, x1, x2;
output [1:3] y;
output z1, z2, z3;

//define internal nets
wire net1, net2, net3, net4, net5, net6;

//-----
//instantiate the logic for flip-flop y[1]
and(net1, ~y[2], ~y[3]);

jkff_neg_clk inst1 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(y[2]),
    .k(net1),
    .q(y[1])
);

//continued on next page
```

```

//-----
//instantiate the logic for flip-flop y[2]
and (net2, ~y[1], x1);
and (net3, y[3], x2);
or (net4, net3, y[1]);

jkff_neg_clk inst2 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net2),
    .k(net4),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and (net5, ~y[1], x1);
or (net6, ~x2, ~y[2]);

jkff_neg_clk inst3 (
    .set_n(1'b1),
    .rst_n(rst_n),
    .clk(clk),
    .j(net5),
    .k(net6),
    .q(y[3])
);

//-----
//define the outputs
assign z1 = ~y[1] && ~y[2] && ~y[3];
assign z2 = y[1] && ~y[2] && y[3];
assign z3 = y[1] && y[2] && ~y[3];

endmodule

```

```

//test bench for moore with no glitches

module moore_no_glitch_jk_tb;

reg rst_n, clk, x1, x2;    //inputs are reg for test bench
wire [1:3] y;              //outputs are wire for test bench
wire z1, z2, z3;

//display variables
initial
$monitor ("x1 x2 = %b, state = %b, z1 = %b, z2 = %b, z3 = %b",
         {x1, x2}, y, z1, z2, z3);

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0 rst_n = 1'b0;          //reset to state_a (00)
    x1 = 1'b0;
    x2 = 1'b0;

    #5 rst_n = 1'b1;          //deassert reset
//-----
    x1 = 1'b0; x2 = $random; @ (posedge clk) //assert z1
    x1 = 1'b1; x2 = $random; @ (posedge clk) //go to state_b
    x1 = $random; x2 = 1'b1; @ (posedge clk) //go to state_c
                                         //assert z2
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_e
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_a
                                         //assert z1
//-----
    x1 = 1'b1; x2 = $random; @ (posedge clk) //go to state_b
    x1 = $random; x2 = 1'b0; @ (posedge clk) //go to state_d
                                         //assert z3
    x1 = $random; x2 = $random; @ (posedge clk) //go to state_a
//-----
    #10 $stop;
end

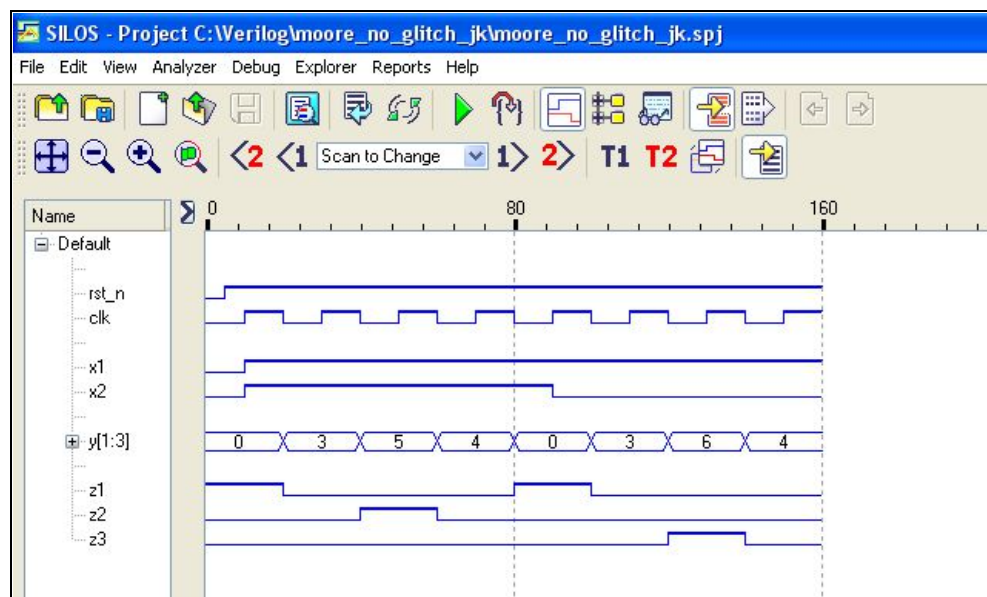
                                     //continued on next page

```

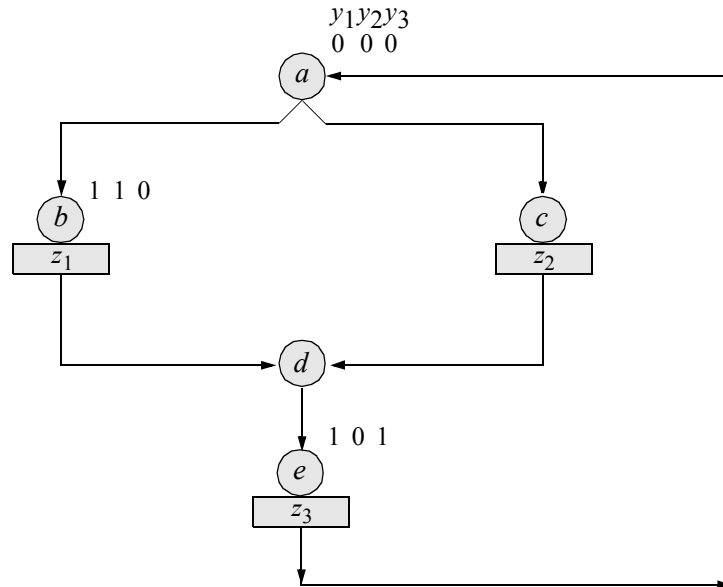
```
//instantiate the module into the test bench
moore_no_glitch_jk inst1 (
  .rst_n(rst_n),
  .clk(clk),
  .x1(x1),
  .x2(x2),
  .y(y),
  .z1(z1),
  .z2(z2),
  .z3(z3)
);

endmodule
```

```
x1 x2 = 00, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 11, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 11, state = 011, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 11, state = 101, z1 = 0, z2 = 1, z3 = 0
x1 x2 = 11, state = 100, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 11, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 10, state = 000, z1 = 1, z2 = 0, z3 = 0
x1 x2 = 10, state = 011, z1 = 0, z2 = 0, z3 = 0
x1 x2 = 10, state = 110, z1 = 0, z2 = 0, z3 = 1
x1 x2 = 10, state = 100, z1 = 0, z2 = 0, z3 = 0
```



- 2.25 Select state codes for states  $c$  and  $d$  for the Moore machine shown below so that there will be no output glitches. Obtain the structural design module using built-in primitives and  $D$  flip-flops, the test bench module, the outputs, and the waveforms. The outputs are asserted at time  $t_1$  through  $t_3$ .



State  $c$ :  $y_1y_2y_3 = 011$ . State  $d$ :  $y_1y_2y_3 = 010$ .

Unused states:  $y_1y_2y_3 = 001, 100, 111$ .

		$y_2y_3$			
$y_1$		00	01	11	10
	0	<sup>0</sup> $x_1$	<sup>1</sup> —	<sup>3</sup> 0	<sup>2</sup> 1
	1	<sup>4</sup> —	<sup>5</sup> 0	<sup>7</sup> —	<sup>6</sup> 0

$Dy_1$

$$Dy_1 = y_2'y_3'x_1 + y_1'y_2y_3'$$

		$y_2y_3$			
$y_1$		00	01	11	10
	0	<sup>0</sup> 1	<sup>1</sup> —	<sup>3</sup> 1	<sup>2</sup> 0
	1	<sup>4</sup> —	<sup>5</sup> 0	<sup>7</sup> —	<sup>6</sup> 1

$Dy_2$

$$Dy_2 = y_2'y_3' + y_2y_3 + y_1y_2$$

		$y_2y_3$			
$y_1$		00	01	11	10
	0	<sup>0</sup> $x_1'$	<sup>1</sup> —	<sup>3</sup> 0	<sup>2</sup> 1
	1	<sup>4</sup> —	<sup>5</sup> 0	<sup>7</sup> —	<sup>6</sup> 0

$Dy_3$

$$Dy_3 = y_2'y_3'x_1' + y_1'y_2y_3'$$

```

//structural moore no glitch with built-in primitives
//and D flip-flops

module moore1_bip_dff (rst_n, clk, x1, y, z1, z2, z3);

//define inputs and outputs
input rst_n, clk, x1;
output [1:3] y;
output z1, z2, z3;

//define internal nets
wire net1, net2, net3, net4, net5, net6, net7,
      net8, net9, net10;

//-----
//instantiate the logic for flip-flop y[1]
and (net1, ~y[2], ~y[3], x1);
and (net2, ~y[1], y[2], ~y[3]);
or (net3, net1, net2);

d_ff_bh inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net3),
    .q(y[1])
);

//-----
//instantiate the logic for flip-flop y[2]
and (net4, ~y[2], ~y[3]);
and (net5, y[2], y[3]);
and (net6, y[1], y[2]);
or (net7, net4, net5, net6);

d_ff_bh inst2 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net7),
    .q(y[2])
);

//-----
//instantiate the logic for flip-flop y[3]
and (net8, ~y[2], ~y[3], ~x1);
and (net9, ~y[1], y[2], ~y[3]);
or (net10, net8, net9);

//continued on next page

```

```

d_ff_bh inst3 (
    .rst_n(rst_n),
    .clk(clk),
    .d(net10),
    .q(y[3])
);

//-----
//define the outputs
assign z1 = y[1] && y[2] && ~y[3];
assign z2 = ~y[1] && y[2] && y[3];
assign z3 = y[1] && ~y[2] && y[3];

endmodule

```

```

//test bench for moore no glitch machine

module moore1_bip_dff_tb;

reg rst_n, clk, x1;           //inputs are reg for test bench
wire [1:3] y;                 //outputs are wire for test bench
wire z1, z2, z3;

//display variables
initial
$monitor ("x1=%b, state=%b, z1=%b, z2=%b, z3=%b",
           x1, y, z1, z2, z3);

//define clock
initial
begin
    clk = 1'b0;
    forever
        #10 clk = ~clk;
end

//define input sequence
initial
begin
    #0   rst_n = 1'b0;           //reset to state_a (000)
        x1 = 1'b1;

    #10  rst_n = 1'b1;           //deassert reset

                                     //continued on next page

```

```

//-----
x1 = 1'b1;@ (posedge clk)          //go to state_b (110)
                                   //assert z1
x1 = $random;@ (posedge clk)       //go to state_d (010)
x1 = $random;@ (posedge clk)       //go to state_e (101)
                                   //assert z3
x1 = 1'b0;@ (posedge clk)          //go to state_a (000)
//-----
x1 = 1'b0;@ (posedge clk)          //go to state_c (011)
                                   //assert z2
x1 = $random;@ (posedge clk)       //go to state_d (010)
x1 = $random;@ (posedge clk)       //go to state_e (101)
                                   //assert z3
x1 = $random;@ (posedge clk)       //go to state_a (000)
//-----
x1 = 1'b1;@ (posedge clk)          //go to state_b (110)
                                   //assert z1
//-----
#20 $stop;
end

//-----
//instantiate the module into the test bench
moore1_bip_dff inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .y(y),
    .z1(z1),
    .z2(z2),
    .z3(z3)
);

endmodule

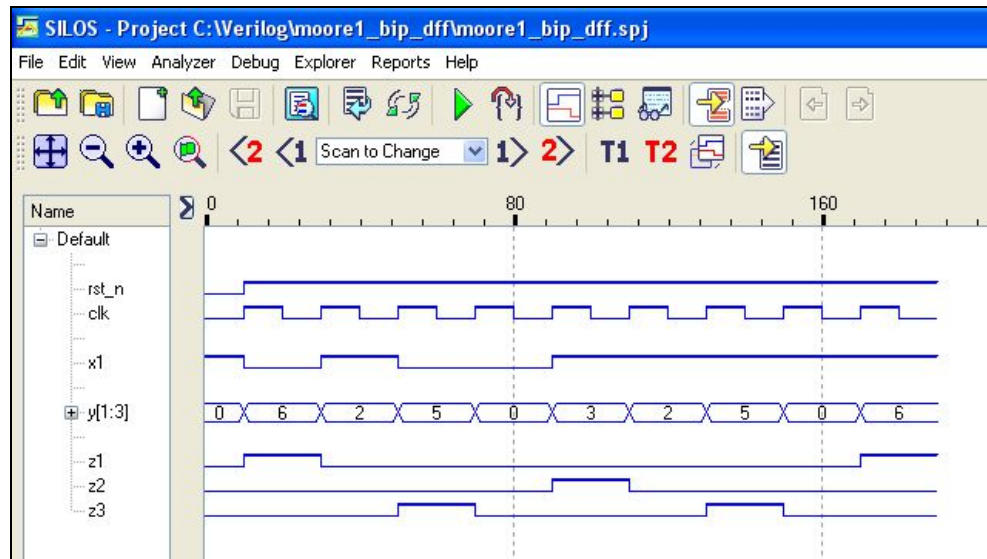
```

```

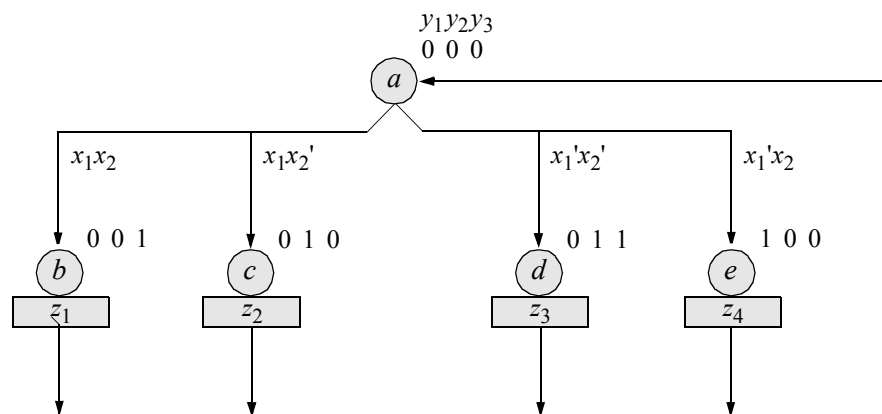
x1=1, state=000, z1=0, z2=0, z3=0
x1=0, state=110, z1=1, z2=0, z3=0
x1=1, state=010, z1=0, z2=0, z3=0
x1=0, state=101, z1=0, z2=0, z3=1
x1=0, state=000, z1=0, z2=0, z3=0
x1=1, state=011, z1=0, z2=1, z3=0
x1=1, state=010, z1=0, z2=0, z3=0
x1=1, state=101, z1=0, z2=0, z3=1
x1=1, state=000, z1=0, z2=0, z3=0
x1=1, state=110, z1=1, z2=0, z3=0

```





- 2.26 Obtain the behavioral design module for the Moore machine represented by the state diagram shown below. To avoid any possible output glitches, delay the output assertion by two time units. Obtain the test bench module, the outputs, and the waveforms.



```

//behavioral moore with 4 outputs
module moore_4_outputs (rst_n, clk, x1, x2, y, z1, z2, z3, z4);

//specify inputs and outputs
input rst_n, clk, x1, x2;
output [1:3] y;           //y is an array of 2 bits
output z1, z2, z3, z4;

reg [1:3] y, next_state;   //must be reg for always
wire z1, z2, z3, z4;

//assign state codes
parameter state_a = 3'b000, //parameter defines a constant
            state_b = 3'b001, //state names must have
            state_c = 3'b010, //at least 3 characters
            state_d = 3'b011,
            state_e = 3'b100;

//set next state
always @ (posedge clk)
begin
    if (~rst_n) //if (~rst_n) is true
        y <= state_a;
    else
        y <= next_state;
end

//determine outputs
assign #2 z1 = (~y[1] & ~y[2] & y[3] & ~clk);
assign #2 z2 = (~y[1] & y[2] & ~y[3] & ~clk);
assign #2 z3 = (~y[1] & y[2] & y[3] & ~clk);
assign #2 z4 = (y[1] & ~y[2] & ~y[3] & ~clk);

//determine next state
//case is a multiple-way conditional branch
//if y = state_a, then do else-if . . else-if
always @ (y or x1 or x2)
begin
    case (y)
        state_a:
            if (x1==1 & x2==1)
                next_state = state_b;
            else if (x1==1 & x2==0)
                next_state = state_c;
            else if (x1==0 & x2==0)
                next_state = state_d;
            else if (x1==0 & x2==1)
                next_state = state_e;
    endcase
end

```

//continued on next page

```

        state_b: next_state = state_a;
        state_c: next_state = state_a;
        state_d: next_state = state_a;
        state_e: next_state = state_a;
        default: next_state = state_a;
    endcase
end
endmodule

```

```

//test bench for moore with 4 outputs
module moore_4_outputs_tb;

    reg rst_n, clk, x1, x2, x3; //inputs are reg for test bench
    wire [1:3] y;               //outputs are wire for test bench
    wire z1, z2, z3, z4;

    //display variables
    initial
    $monitor ("x1 x2 = %b, state = %b, z1 z2 z3 z4 = %b",
              {x1, x2}, y, {z1, z2, z3, z4});

    //define clock
    initial
    begin
        clk = 1'b0;
        forever
            #10 clk = ~clk;
    end

    //define input sequence
    initial
    begin
        #0 rst_n = 1'b0;           //reset to state_a (000)
        x1 = 1'b0; x2 = 1'b0;

        #5 rst_n = 1'b1;
    //-----
        x1 = 1'b1; x2 = 1'b1;
        @ (posedge clk)             //go to state_b (001), assert z1
        @ (posedge clk)             //go to state_a (000)

        x1 = 1'b1; x2 = 1'b0;
        @ (posedge clk)             //go to state_c (010), assert z2
        @ (posedge clk)             //go to state_a (000)

        //continued on next page
    end
endmodule

```

```

x1 = 1'b0;x2 = 1'b0;
@ (posedge clk)      //go to state_d (011), assert z3
@ (posedge clk)      //go to state_a (000)

x1 = 1'b0;x2 = 1'b1;
@ (posedge clk)      //go to state_e (100), assert z4
@ (posedge clk)      //go to state_a (000)
//-----
#30 $stop;
end

//-----
//instantiate the module into the test bench
moore_4_outputs inst1 (
    .rst_n(rst_n),
    .clk(clk),
    .x1(x1),
    .x2(x2),
    .y(y),
    .z1(z1),
    .z2(z2),
    .z3(z3),
    .z4(z4)
);

endmodule

```

```

x1 x2 = 00, state = xxx, z1 z2 z3 z4 = 0000
x1 x2 = 11, state = xxx, z1 z2 z3 z4 = 0000
x1 x2 = 11, state = 000, z1 z2 z3 z4 = 0000
x1 x2 = 10, state = 001, z1 z2 z3 z4 = 0000
x1 x2 = 10, state = 001, z1 z2 z3 z4 = 1000
x1 x2 = 10, state = 001, z1 z2 z3 z4 = 0000
x1 x2 = 10, state = 000, z1 z2 z3 z4 = 0000
x1 x2 = 00, state = 010, z1 z2 z3 z4 = 0000
x1 x2 = 00, state = 010, z1 z2 z3 z4 = 0100
x1 x2 = 00, state = 010, z1 z2 z3 z4 = 0000
x1 x2 = 00, state = 000, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 011, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 011, z1 z2 z3 z4 = 0010
x1 x2 = 01, state = 011, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 000, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 100, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 100, z1 z2 z3 z4 = 0001
x1 x2 = 01, state = 100, z1 z2 z3 z4 = 0000
x1 x2 = 01, state = 000, z1 z2 z3 z4 = 0000

```

