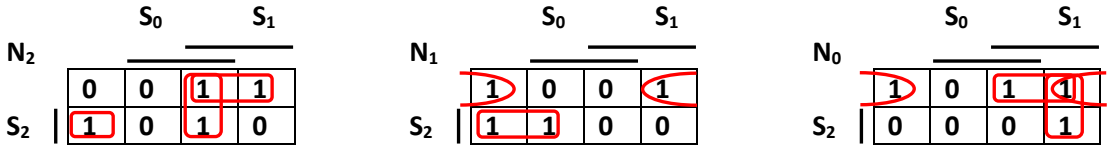
In **Task 1**, we have to design an FSM that generates the output sequence determined by the personal code. If the personal code is **3527461**, then the FSM should generate the **0, 3, 5, 2, 7, 4, 6, 1, …** repeating output sequence and the output values must be taken directly from the state register (output encoding). The following state table describes the behavior of the FSM. Because output encoding is used, we can get the encoded state table by converting the decimal values to binary.

Current state	Next state
0	3
1	0
2	7
3	5
4	6
5	2
6	1
7	4

Current state $S_2S_1S_0$	Next state $N_2N_1N_0$
000	011
001	000
010	111
011	101
100	110
101	010
110	001
111	100

The encoded state table is the truth table of the next state logic. The minimized two-level logic equations can be derived from the Karnaugh maps of the next state bits N_2, N_1 and N_0.



$N_2 = S_0'S_1'S_2 + S_0S_1 + S_1S_2'$ = $S_1'(S_0'S2) + S1(S_0+S_2') = S_1'(S_0'S2) + S1(S_0'S_2)' = S_1$ xor $(S_0'S_2)$
$N_1 = S_0'S_2' + S_1'S_2$
$N_0 = S_0'S_2' + S_1S_2' + S_0'S_1$

The logic equation of the N_2 next state bit can be simplified further using the De Morgan's law if XOR/XNOR gates are allowed. The circuit diagram can be drawn up according to the equations. The Verilog HDL implementation of the FSM can be seen below.

```
 3  module task1(
 4     input  wire       clk,
 5     input  wire       rst,
 6     output wire [2:0] y
 7  );
 8
 9  //Defining the states.
10  localparam S0 = 3'd0;
11  localparam S1 = 3'd1;
12  localparam S2 = 3'd2;
13  localparam S3 = 3'd3;
14  localparam S4 = 3'd4;
15  localparam S5 = 3'd5;
16  localparam S6 = 3'd6;
17  localparam S7 = 3'd7;
18
19  //State register.
20  reg [2:0] state;
21
22  //Driving the output.
23  assign y = state;
```

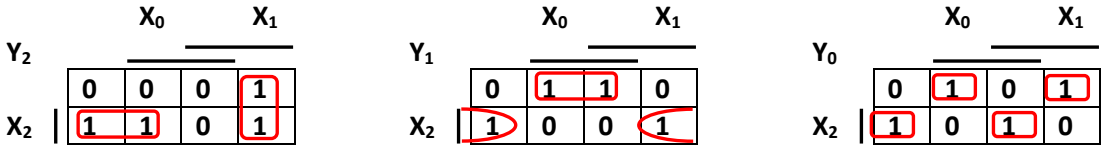
```
25  //Implementing the state register
26  //and the next state logic.
27  always @(posedge clk)
28  begin
29     if (rst)
30        state <= S0;
31     else
32        case (state)
33           S0: state <= S3;
34           S1: state <= S0;
35           S2: state <= S7;
36           S3: state <= S5;
37           S4: state <= S6;
38           S5: state <= S2;
39           S6: state <= S1;
40           S7: state <= S4;
41        endcase
42  end
43
44  endmodule
```

In **Task 2**, we have to design a 3-bit binary up-counter and a combinational output logic that maps the output of the counter in natural order to the numbers in the order determined by the personal code. If the personal code is **3527461**, then the **01234567** → **03527461** mapping should be done (see the table below). We can get the truth table of the mapping logic by converting the decimal numbers to binary. The counter design is not described here because it is very similar to the **Task 1**.

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

Input $X_2X_1X_0$	Output $Y_2Y_1Y_0$
000	000
001	011
010	101
011	010
100	111
101	100
110	110
111	001

The minimized two-level logic equations can be derived from the Karnaugh maps of the output bits Y_2, Y_1 and Y_0.



$Y_2 = X_0'X_1 + X_1'X_2$
$Y_1 = X_0X_2' + X_0'X_2$ = X_0 xor X_2
$Y_0 = X_0X_1'X_2' + X_0'X_1X_2' + X_0'X_1'X_2 + X_0X_1X_2$ = X_0 xor X_1 xor X_2

The logic equations of the Y_1 and Y_2 outputs can be simplified further if XOR/XNOR gates are allowed. The circuit diagram can be drawn up according to the equations. The Verilog HDL implementation of the counter and the mapping logic can be seen below.

```
 3  module task2(
 4     input  wire       clk,
 5     input  wire       rst,
 6     output reg  [2:0] y
 7  );
 8
 9  //3-bit binary up-counter.
10  reg [2:0] cnt;
11
12  always @(posedge clk)
13  begin
14     if (rst)
15        cnt <= 3'd0;
16     else
17        cnt <= cnt + 3'd1;
18  end
```

```
20  //Combinational mapping logic.
21  always @(*)
22  begin
23     case (cnt)
24        3'd0: y <= 3'd0;
25        3'd1: y <= 3'd3;
26        3'd2: y <= 3'd5;
27        3'd3: y <= 3'd2;
28        3'd4: y <= 3'd7;
29        3'd5: y <= 3'd4;
30        3'd6: y <= 3'd6;
31        3'd7: y <= 3'd1;
32     endcase
33  end
34
35  endmodule
```

The following Verilog test fixture can be used for testing the two designs. The timing diagram shows the expected results.

```
25  module hw1_TF;
26
27      //Inputs
28      reg clk;
29      reg rst;
30
31      //Outputs
32      wire [2:0] task1_y, task2_y;
33
34      //Instantiate the Unit Under Test (UUT)
35      task1 uut1(.clk(clk), .rst(rst), .y(task1_y));
36      task2 uut2(.clk(clk), .rst(rst), .y(task2_y));
37
38      initial
39      begin
40          // Initialize Inputs
41          clk = 0;
42          rst = 0;
43
44          // Wait 100 ns for global reset to finish
45          #100;
46
47          // Add stimulus here
48          rst = 1;
49          #20 rst = 0;
50      end
51
52      //Clock generation.
53      always
54          #10 clk <= ~clk;
55
56  endmodule
```