Digital System Design ELEC373/473

Algorithmic State Machines (ASM)

BCD to Excess 3

Verilog Sequential Template

```
module model_name (list of outputs and inputs);
external signal declarations
internal signal declarations
begin
        -- the state process defines the storage elements
        always @ (posedge (or negedge) clock, optional reset)
        begin
                verilog statements for storage elements (normally nonblocking)
        end

        -- the comb process defines the combinational logic
        always @ (level sensitivity list – usually includes all inputs and state vars)
        begin
                verilog statements which specify combinational logic
                (normally use a case statement to identify states)
        end
endmodule;
```

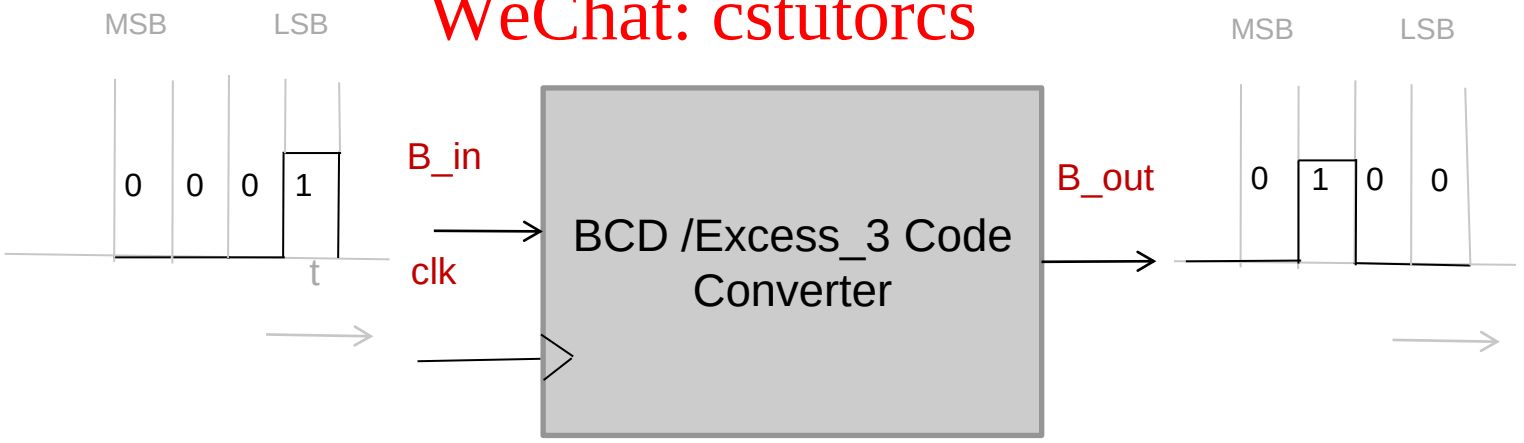
BCD to XS-3 Code Converter

- An Excess-3 code is obtained by adding 0011 to the BCD code.
- An Excess-3 code is self complementing. The 9's complement of a number can be obtained by bitwise complementing it.
- The 10's complement of an XS-3 is its 9's complement plus 1.
- Negative numbers are represented by their 10's complement.
- Example:
- 873 - 218 = 655
- -218 ⟹ 781 (9's complement)
- -218 ⟹ 782 (10's complement)
- 873 +782 = 1655 (overflow)
- 1655 – 1000 = 655 (correct result)

Decimal	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

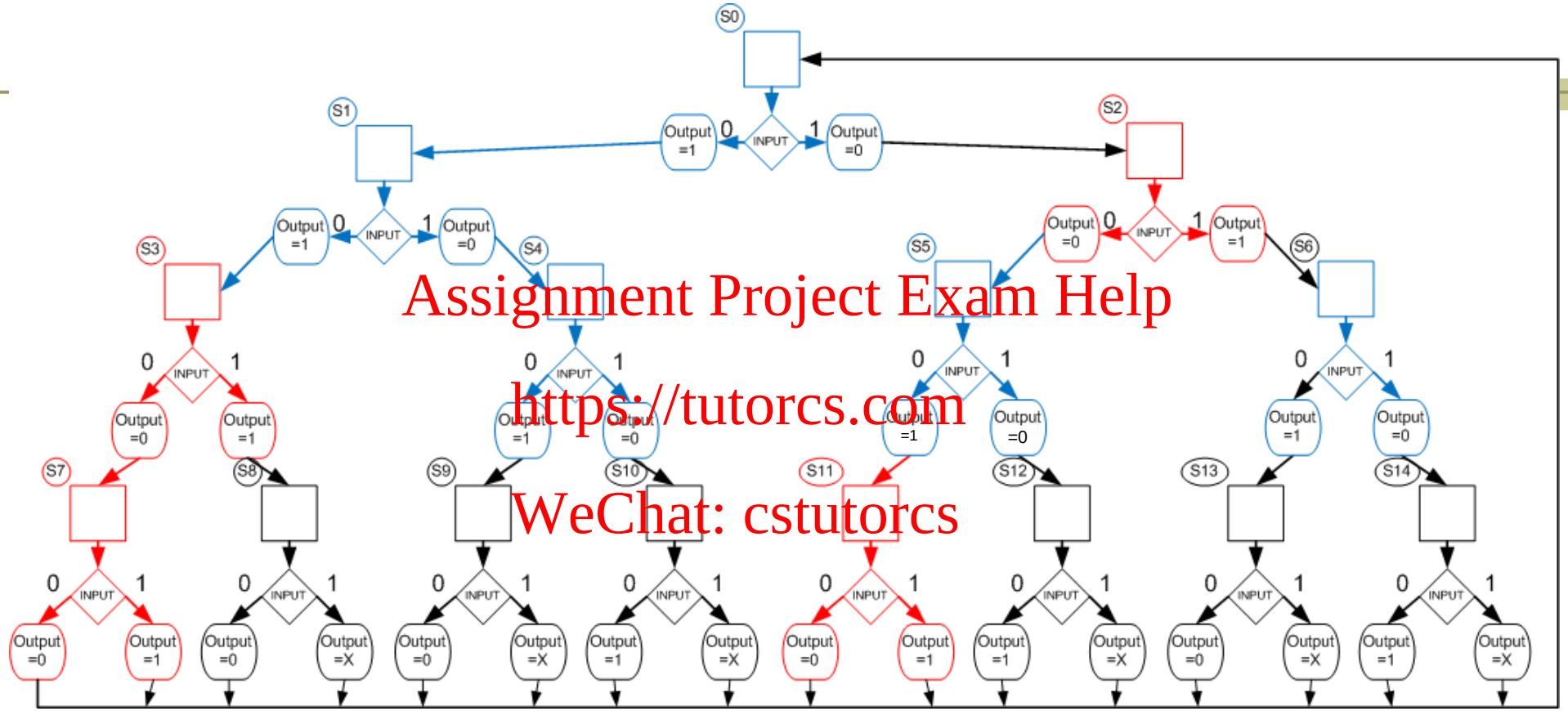
BCD to XS-3 Code Converter

- A Mealy FSM is designed to convert a serial bit stream of a BCD code to Excess-3 bit stream.



LSB is inserted first.

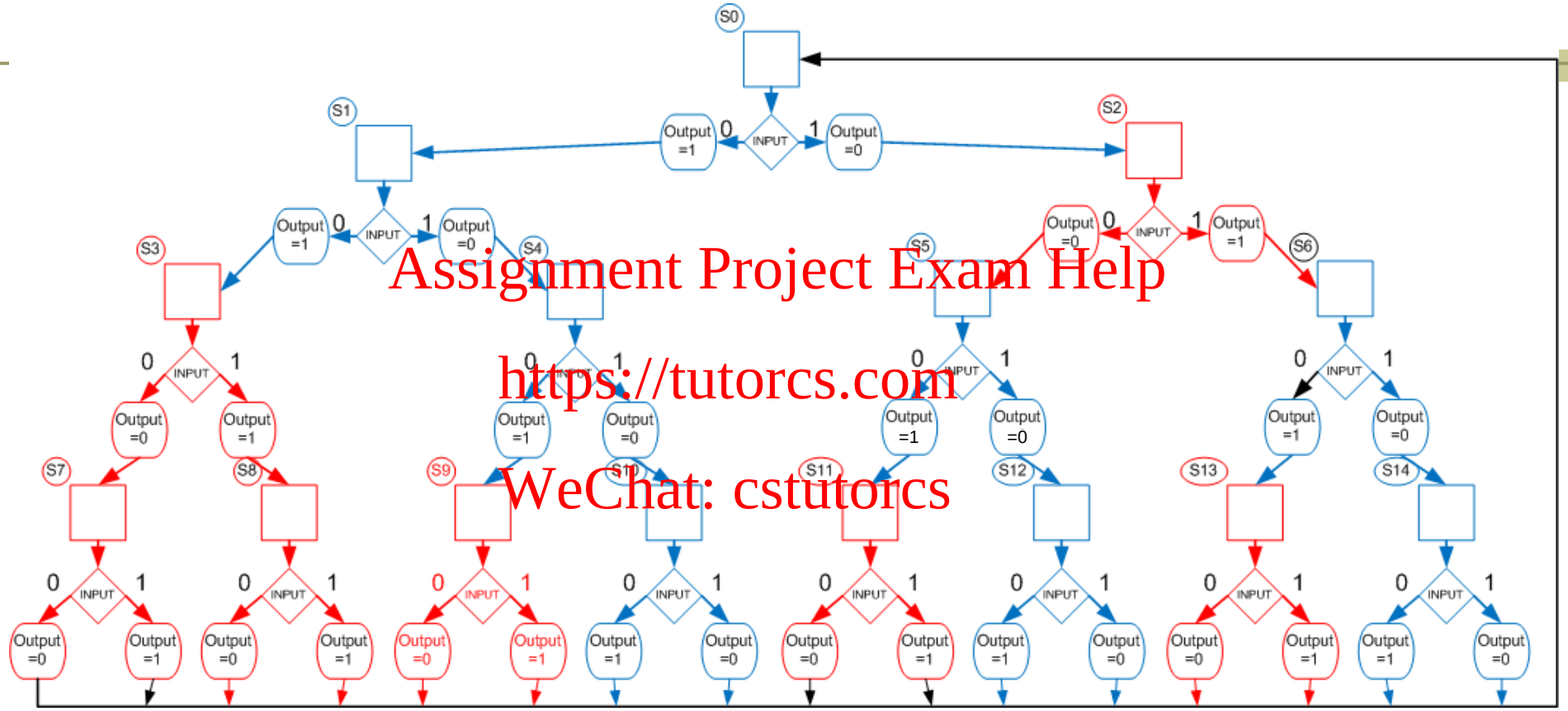
ASM Chart



There are 4 possible types of states in each time period.

1) The output is the same as the input (red states – S2, S3, S7, S11).
2) The output is the inversion of the input (blue states – S0, S1, S4, S5, S6).
3) The output is 1 for valid input values (could be S10, S12, or S14).
4) The output is 0 for valid input values (could be S8, S9, or S13).

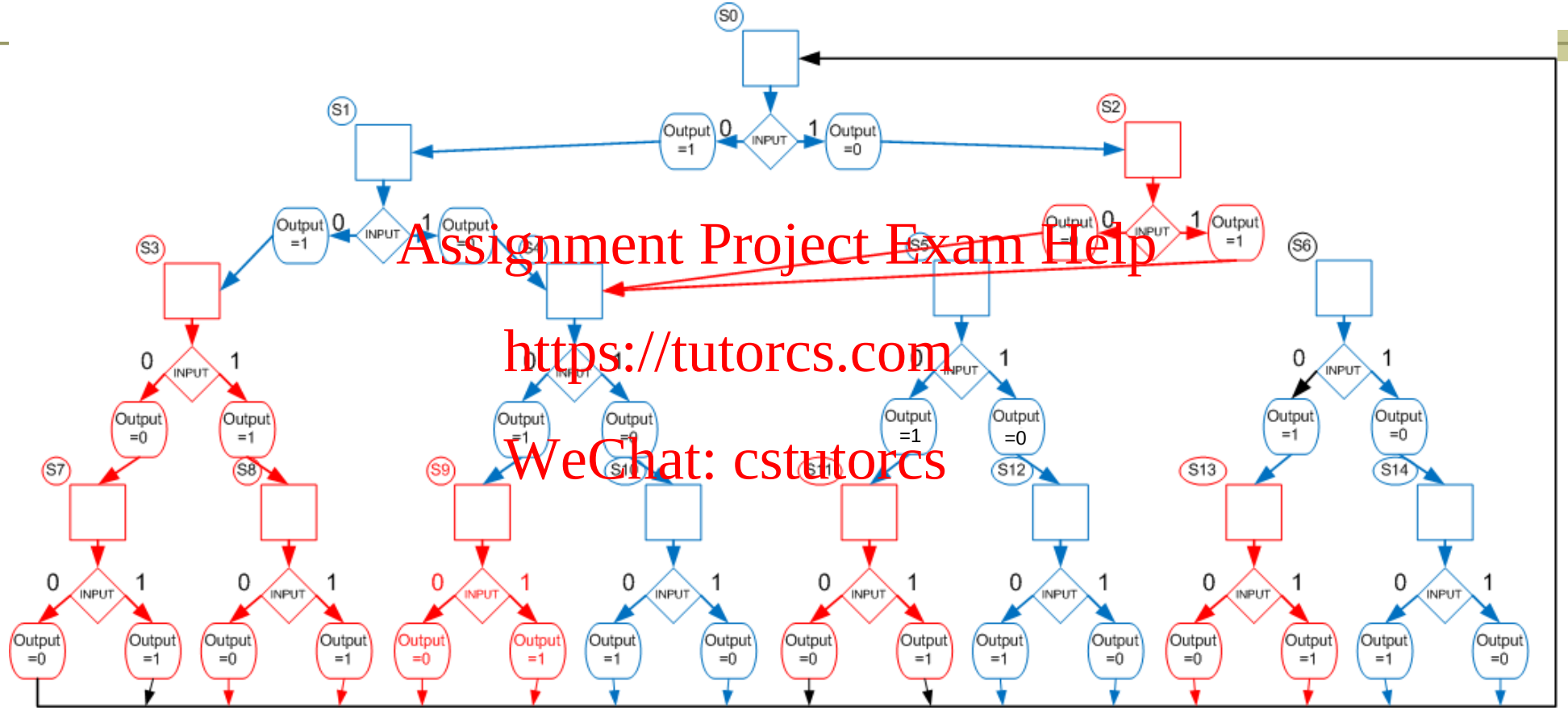
State Reduction (1)



By setting the don't cares the state types can be reduced to 2 types.

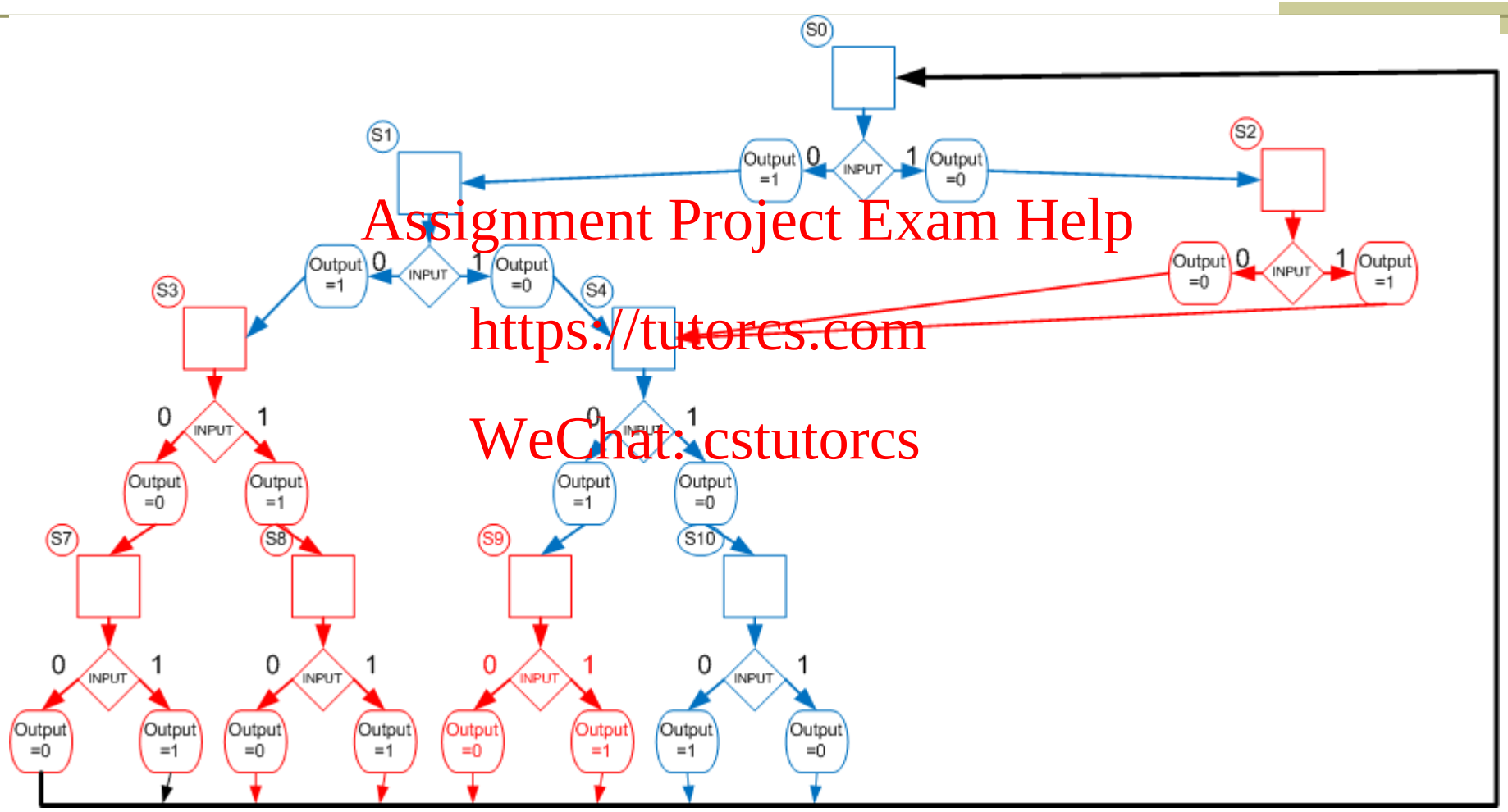
1) The output is the same as the input (red states – S2, S3, S7, S8, S9, S11, S13).
2) The output is the inversion of the input (blue states–S0,S1,S4,S5,S6,S10,S12,S14).

State Reduction (2)

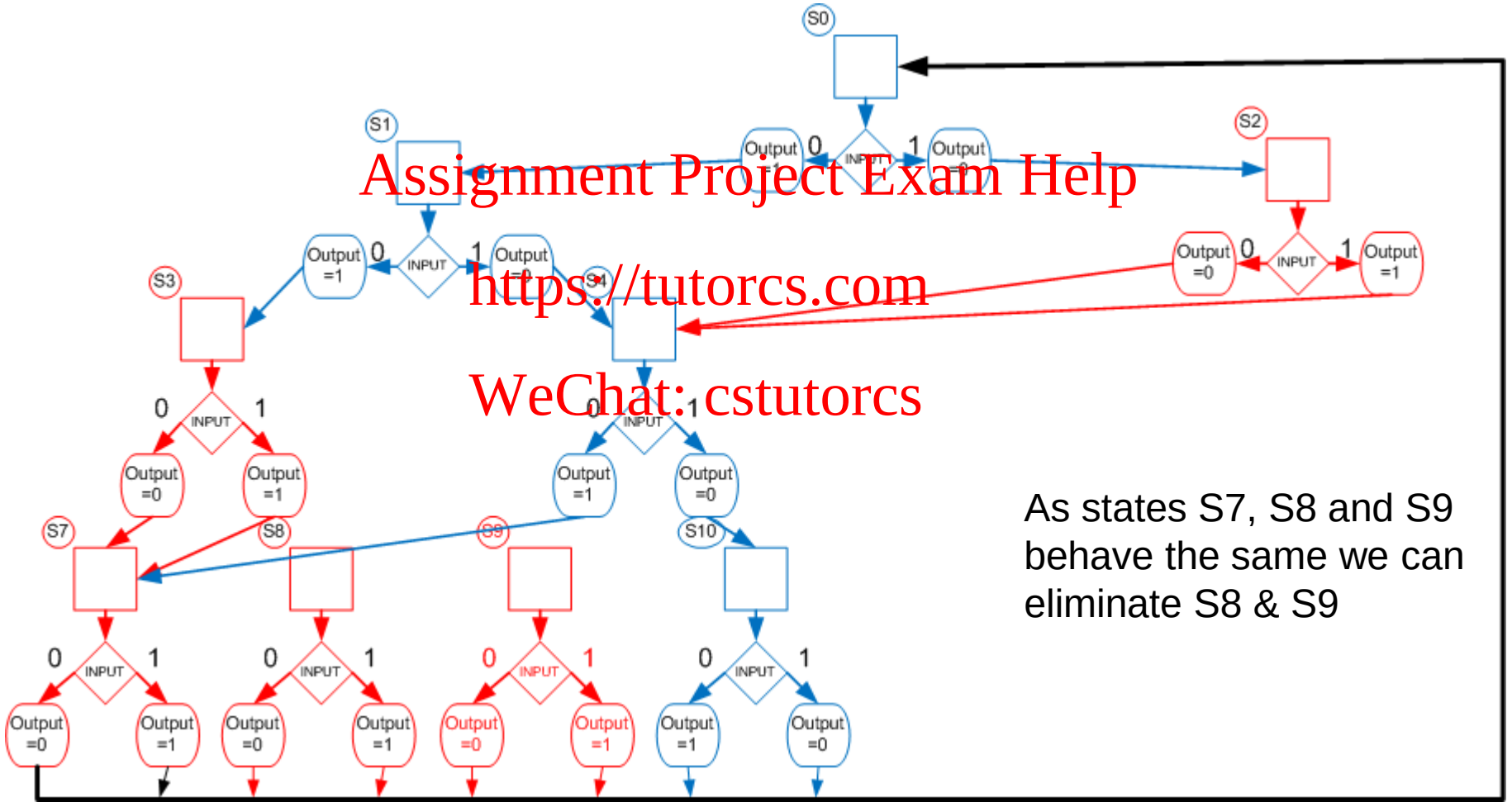


By going from S2 to S4 we get the same functionality but can eliminate six states S5, S6, S11, S12, S13 & S14

State Reduction (3)

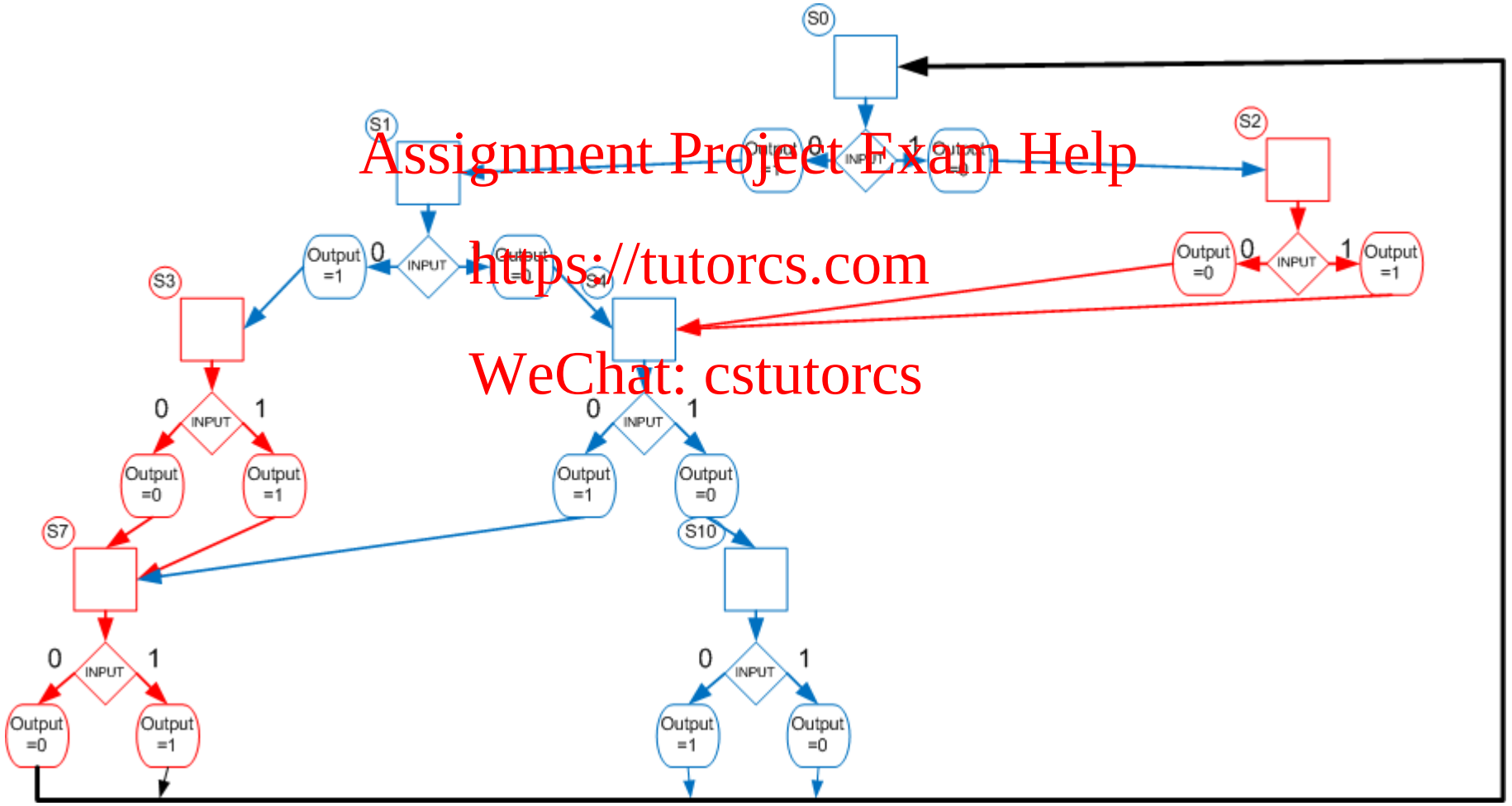


State Reduction (4)

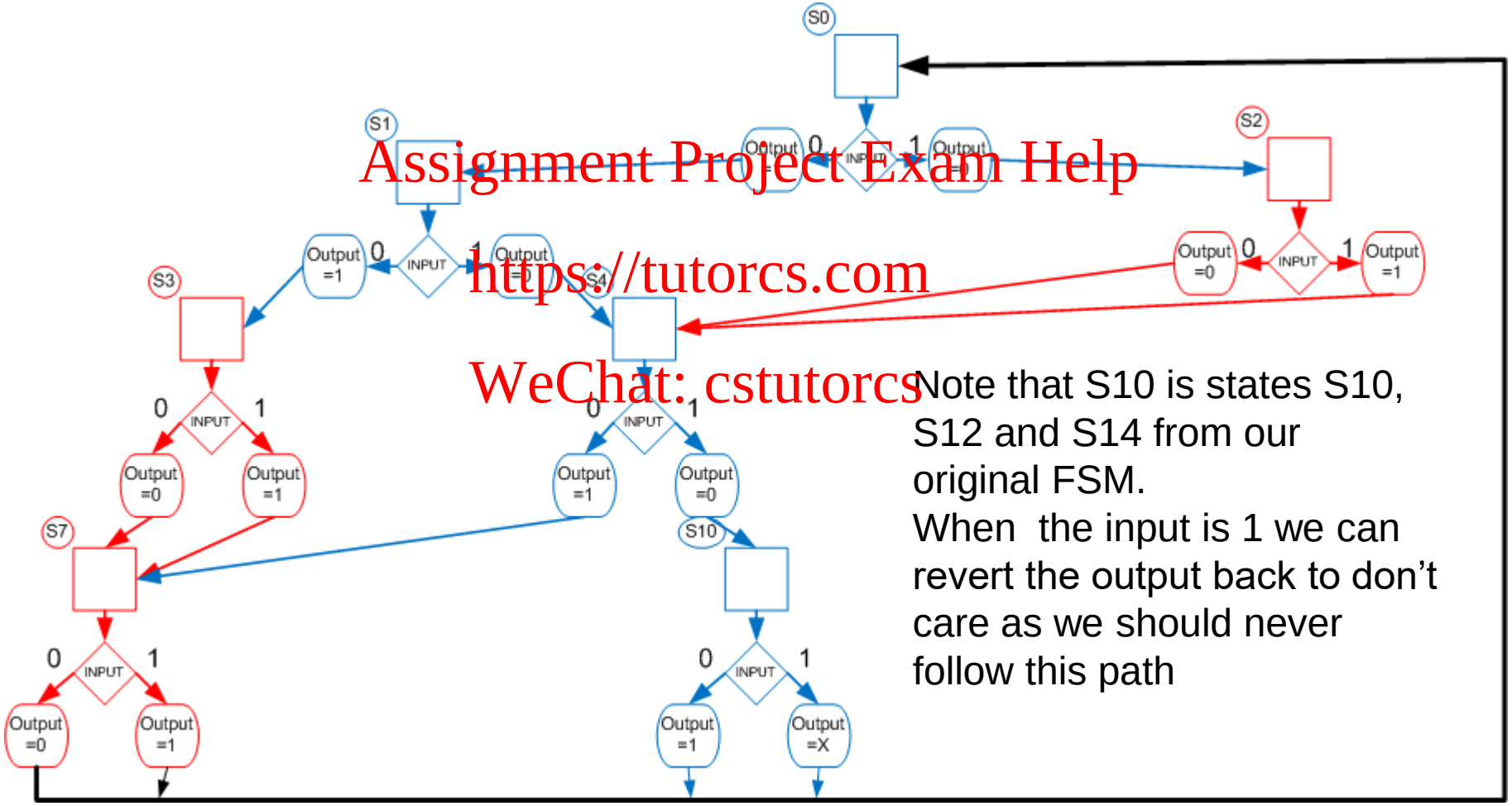


As states S7, S8 and S9 behave the same we can eliminate S8 & S9

State Reduction (5)

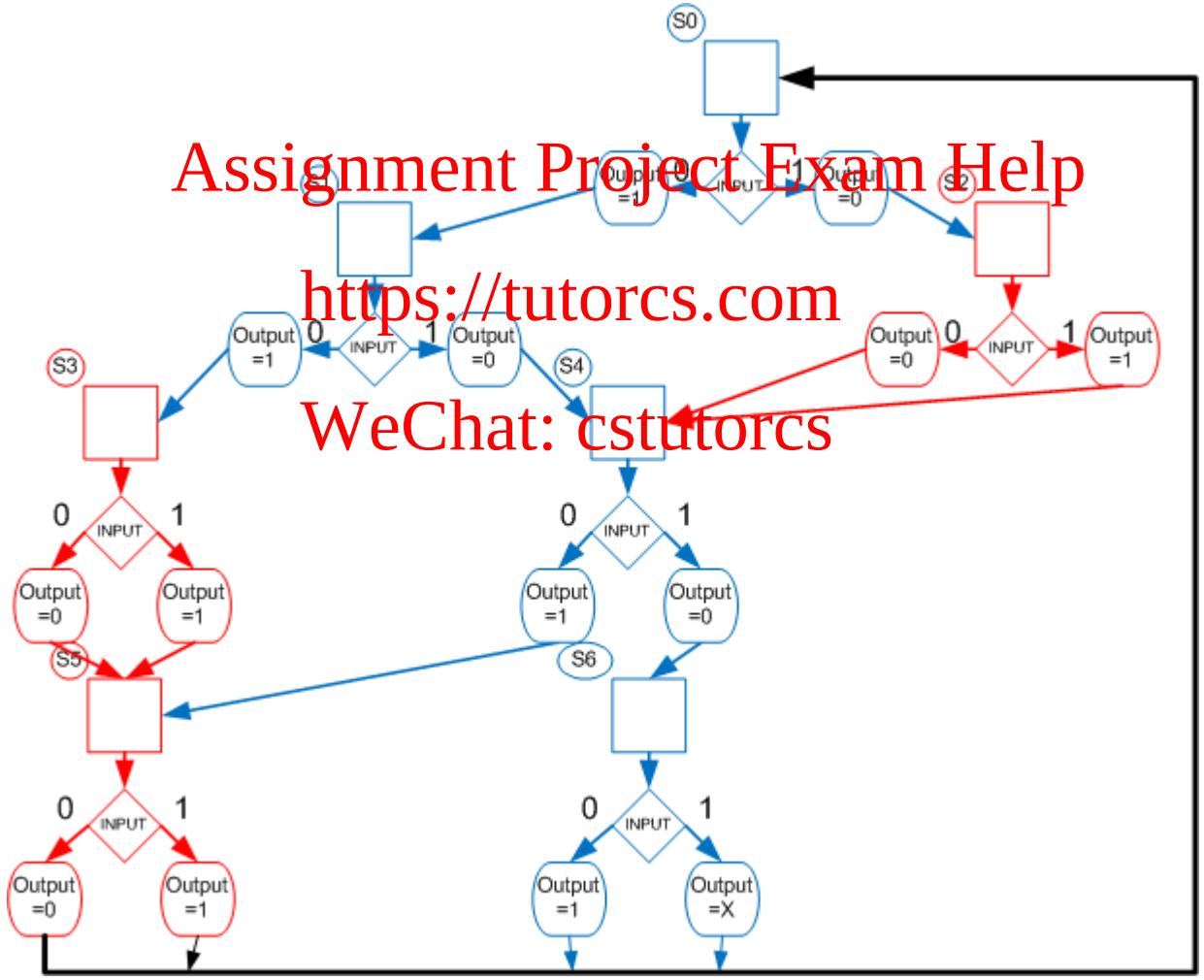


State Reduction (6)



Note that S10 is states S10, S12 and S14 from our original FSM.
When the input is 1 we can revert the output back to don't care as we should never follow this path

State Diagram with Minimum States



Verilog Model

Edge-sensitive cyclic behaviour describes the state transition.

Level-sensitive cyclic behaviour describes the next state and the output logic.

```
1  module BCD_to_Excess_3b(B_out, B_in, clk, reset_b);
2      output      B_out;
3      input       B_in, clk, reset_b;
4      parameter   S_0 = 3'b000,
5                  S_1 = 3'b001,
6                  S_2 = 3'b101,
7                  S_3 = 3'b111,
8                  S_4 = 3'b011,
9                  S_5 = 3'b110,
10                 S_6 = 3'b010;
11
12     reg [2:0]   state, next_state;
13     reg         B_out;
14
15     always@(posedge clk or negedge reset_b)
16         if (reset_b == 0) state <=S_0; else state <=next_state;
17
18     always @(state or B_in) begin
19         B_out=0;
20         case (state)
21             S_0: if(B_in == 0)begin next_state = S_1; B_out = 1; end
22                     else if (B_in == 1)begin next_state = S_2; end
```

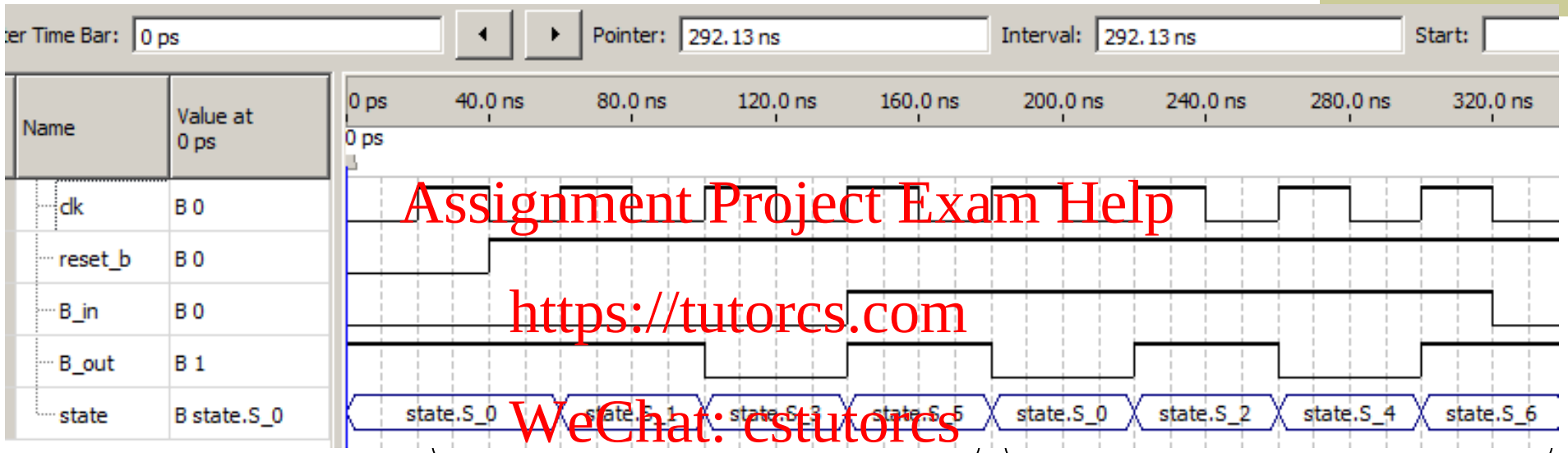

Nonblocking assignments are used in the edge-sensitive behaviour while procedural or blocked assignments are used in the level-sensitive behaviours.

Verilog Model

```
18      always @(state or B_in) begin
19          B_out=0;
20          case (state)
21              S_0: if(B_in == 0)begin next_state = S_1; B_out = 1; end
22                   else if (B_in == 1)begin next_state = S_2; end
23              S_1: if(B_in == 0)begin next_state = S_3; B_out = 1; end
24                   else if (B_in == 1)begin next_state = S_4; end
25              S_2: begin next_state = S_4; B_out = B_in; end
26              S_3: begin next_state = S_5; B_out = B_in; end
27              S_4: if(B_in == 0)begin next_state = S_5; B_out = 1; end
28                   else if (B_in == 1)begin next_state = S_6; end
29              S_5: begin next_state = S_0; B_out = B_in; end
30              S_6: begin next_state = S_0; B_out = 1; end
31          endcase
32      end
33  endmodule
```


Using don't cares in the state graph the S_6 output has been designed to be 1 regardless of its input.

Functional Simulation

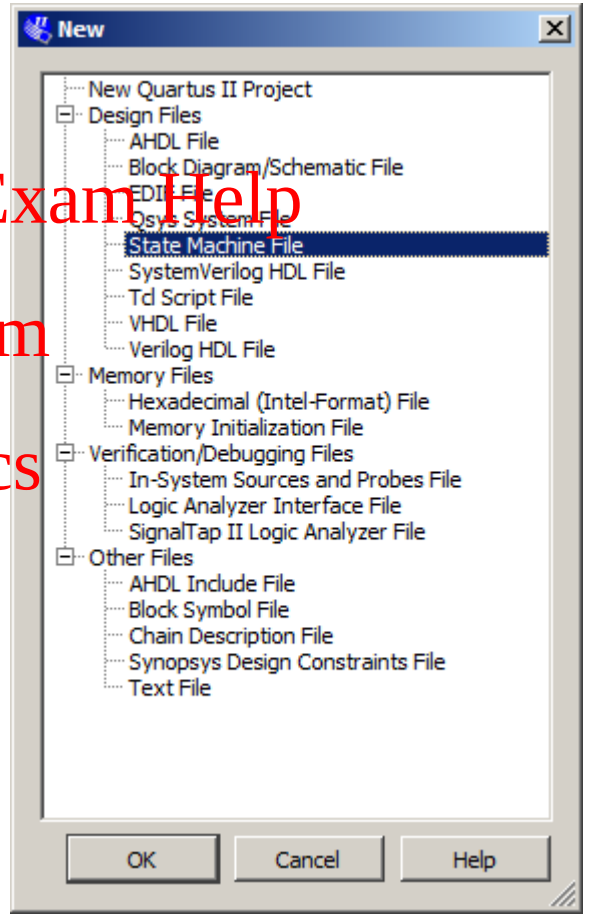


Input = 1000
Output = 1011

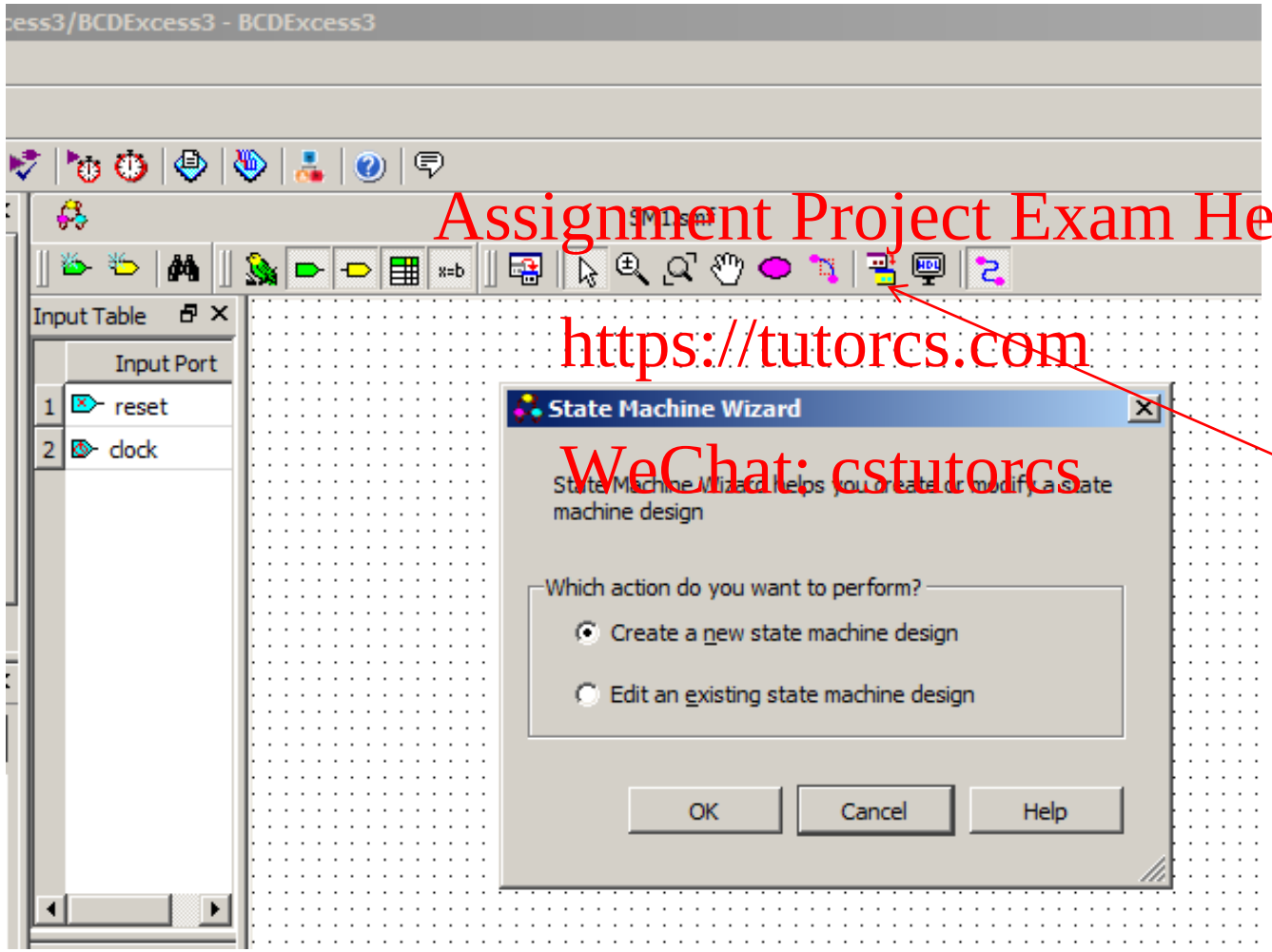
Input = 1111 (illegal)
Output = 1010

State Machine Editor

- Quartus can generate HDL code from the entered state machine.
- Create a new state machine file
- Then use the state machine wizard for entering your design

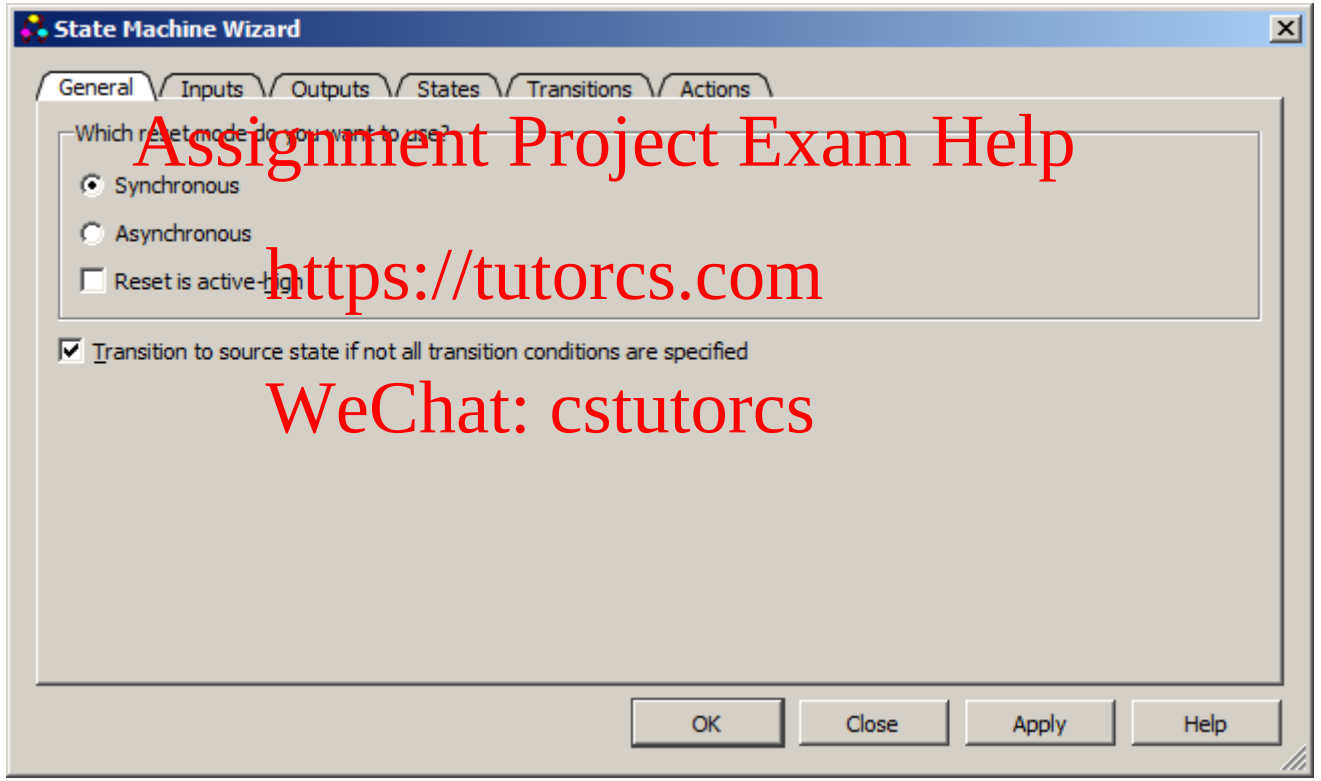


State Machine Wizard

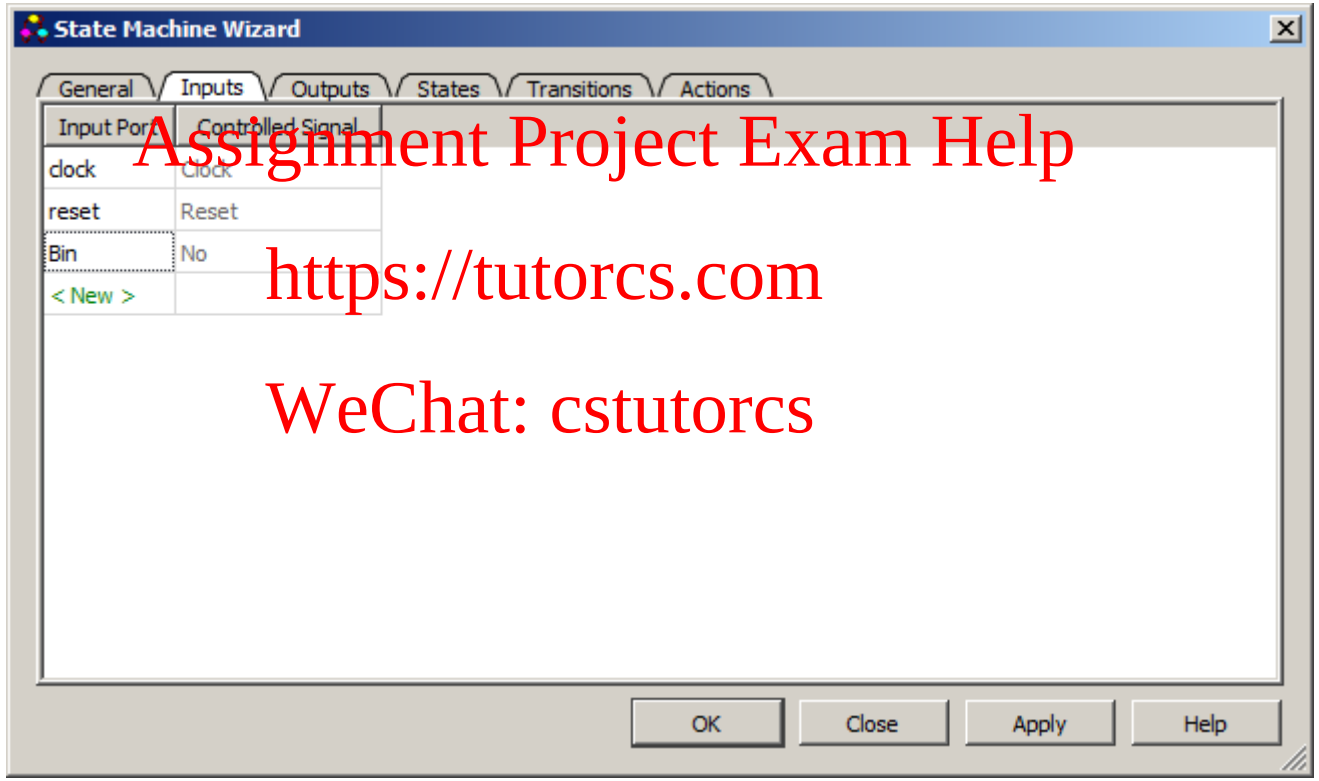


State Machine Wizard

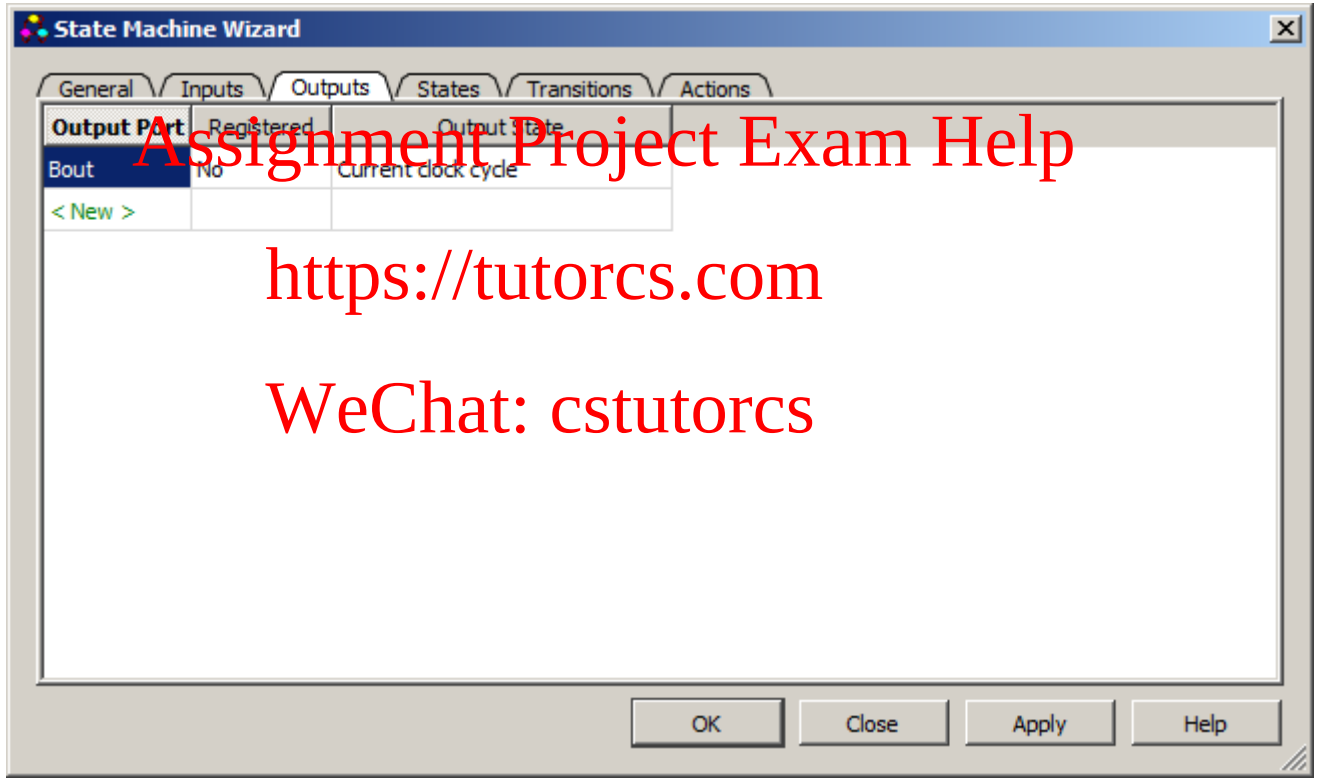
Choose the type of Reset and default states



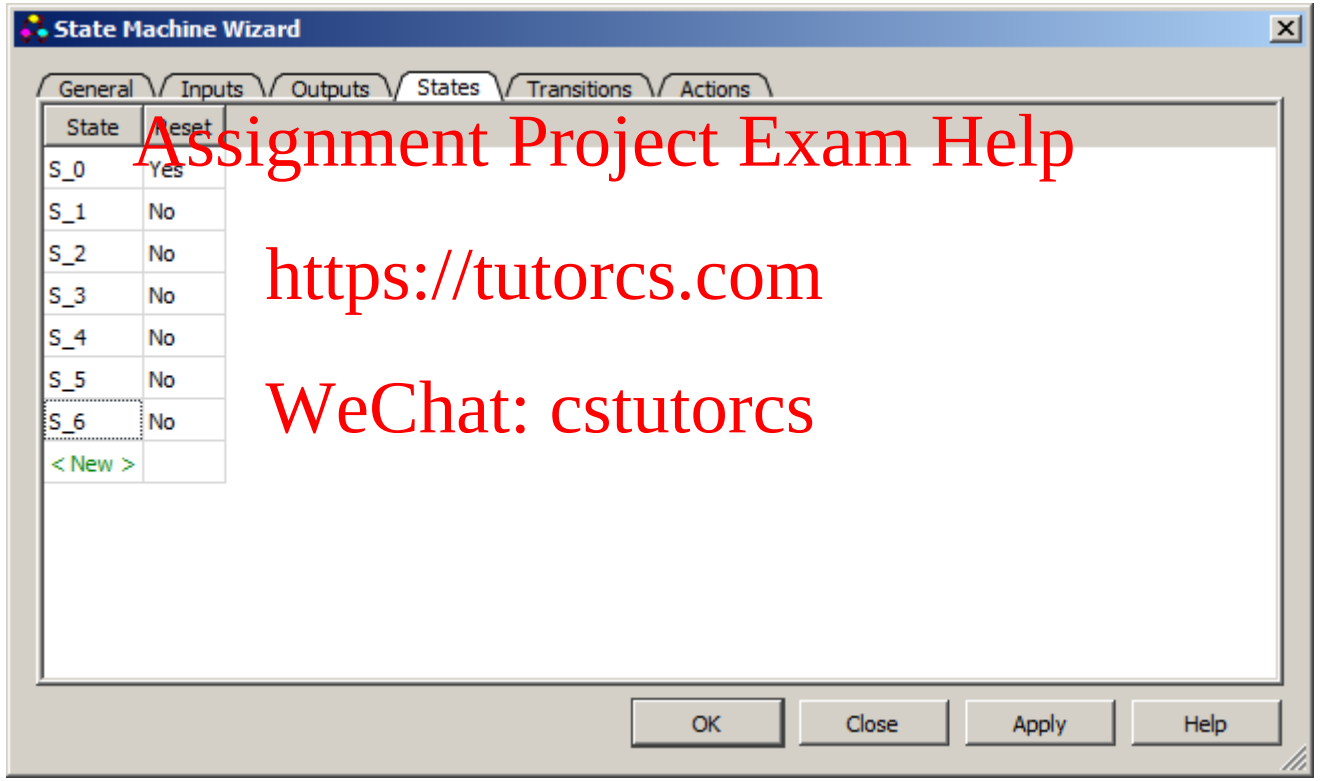
Define inputs



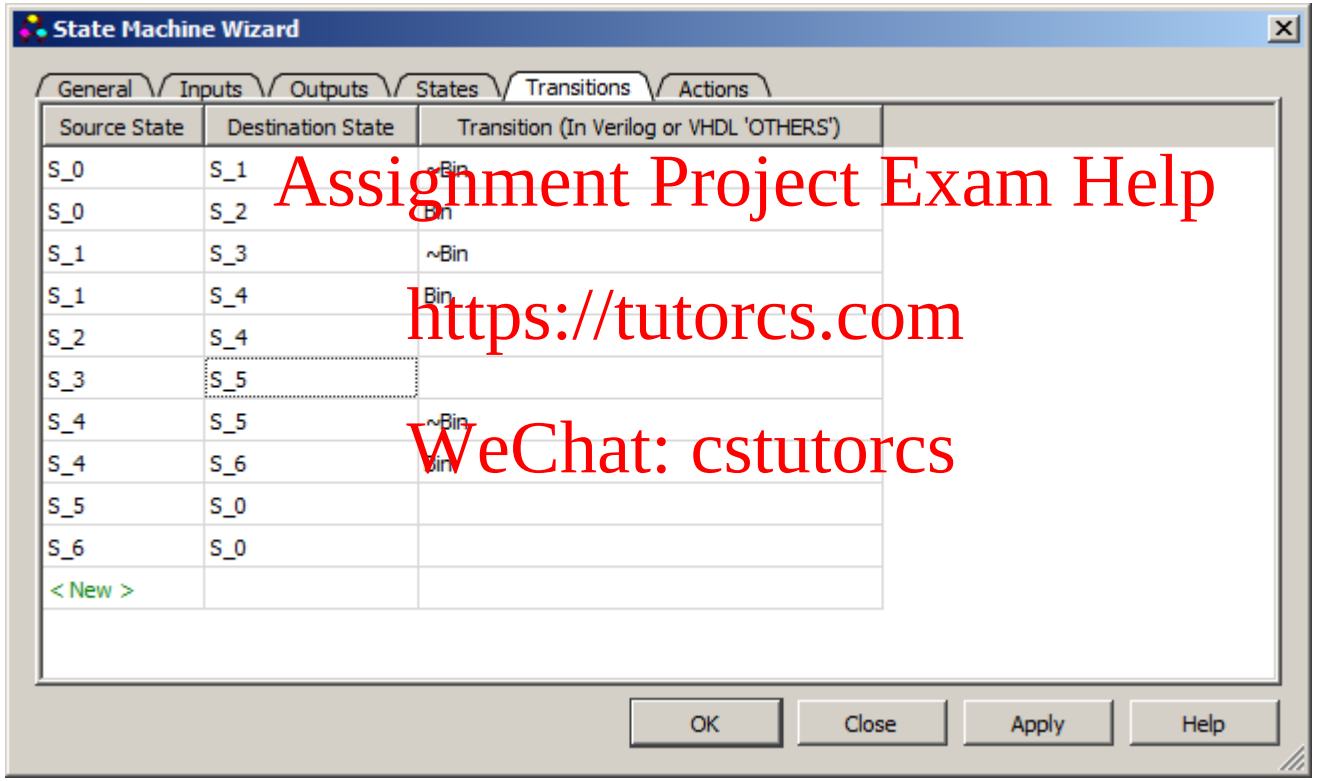
Define outputs



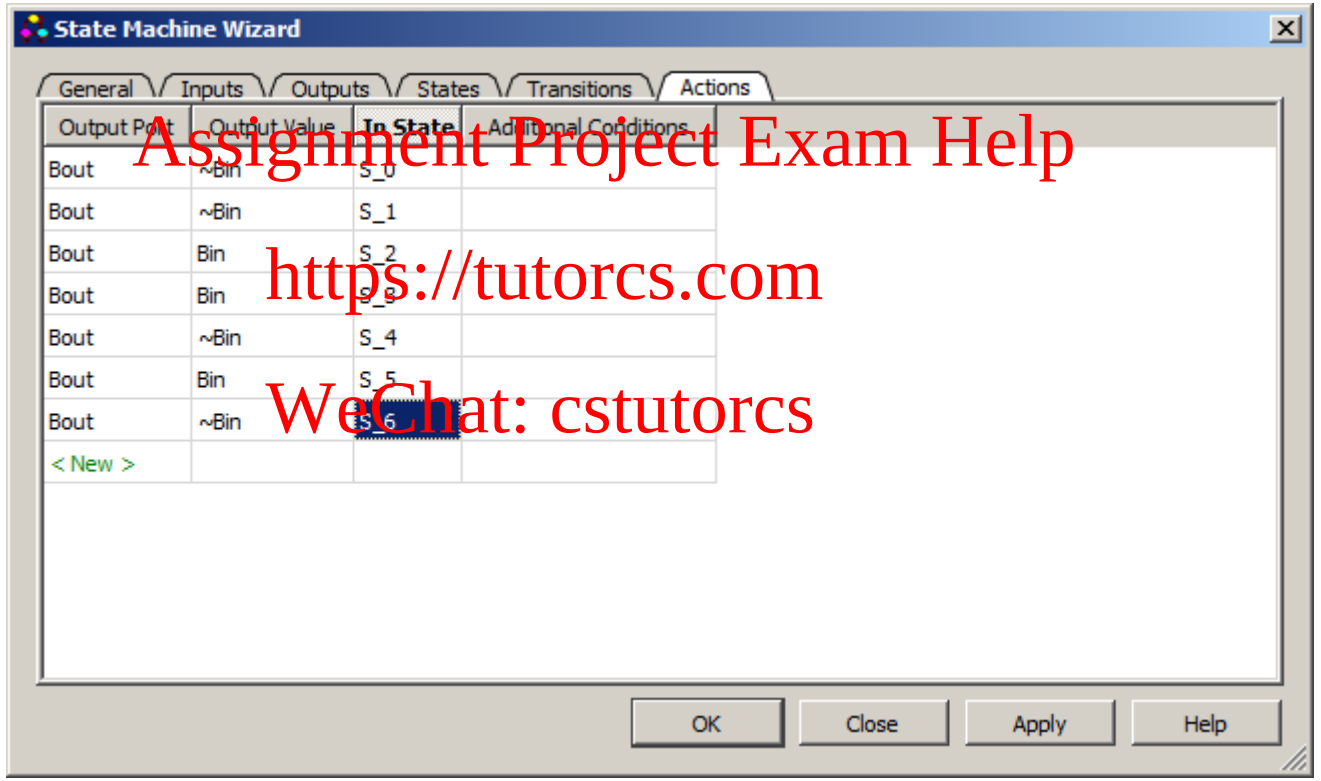
Define the states



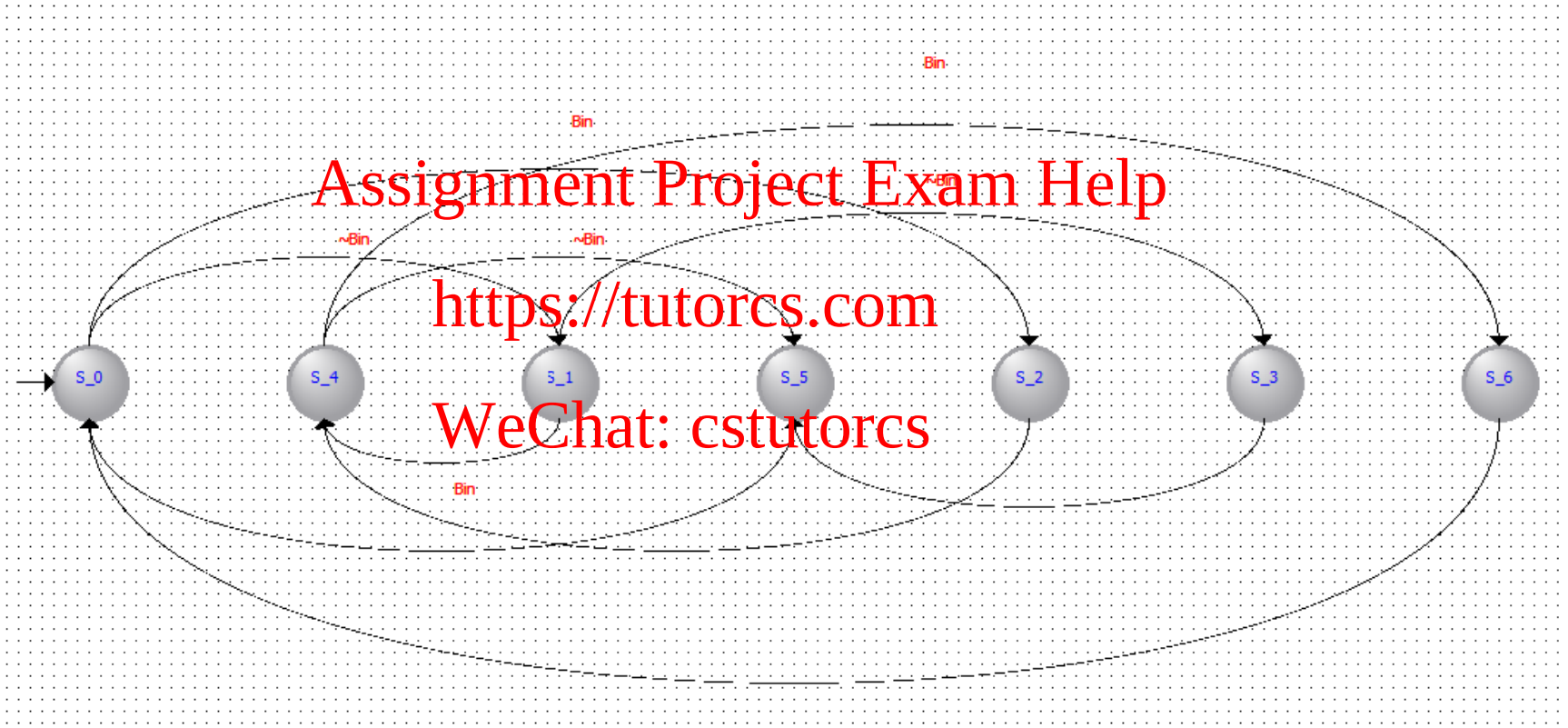
Define the transitions



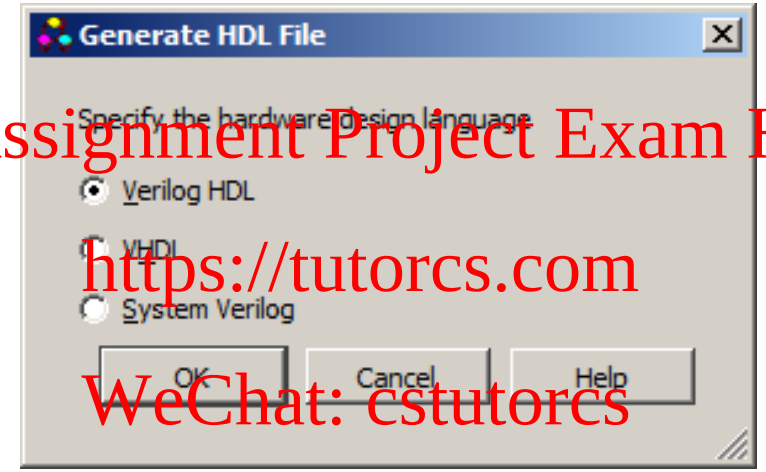
Define the Actions (outputs)



Check the generated state diagram



Save the state machine file and generate a HDL file



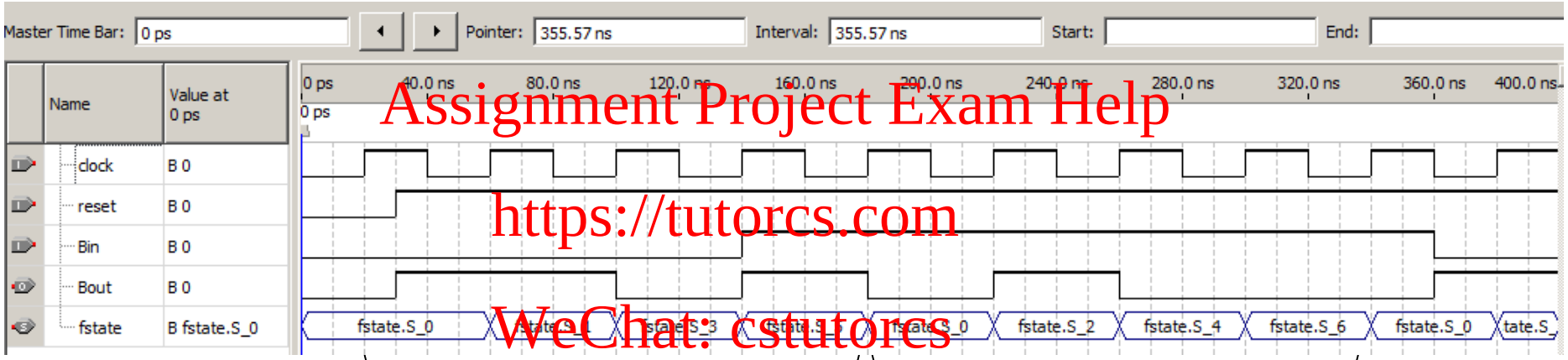
Check the generated Verilog file

```
22  module BCDExcess3 (
23      clock,reset,Bin,
24      Bout);
25
26      input clock;
27      input reset;
28      input Bin;
29      tri0 reset;
30      tri0 Bin;
31      output Bout;
32      reg Bout;
33      reg [6:0] fstate;
34      reg [6:0] reg_fstate;
35      parameter S_0=0,S_1=1,S_2=2,S_3=3,S_4=4,S_5=5,S_6=6;
36
37      always @(posedge clock)
38      begin
39          if (clock) begin
40              fstate <= reg_fstate;
41          end
42      end
43
44      always @(fstate or reset or Bin)
45      begin
46          if (~reset) begin
47              reg_fstate <= S_0;
48              Bout <= 1'b0;
49          end
50          else begin
51              Bout <= 1'b0;
52              case (fstate)
53                  S_0: begin
```

```
43
44      always @(fstate or reset or Bin)
45      begin
46          if (~reset) begin
47              reg_fstate <= S_0;
48              Bout <= 1'b0;
49          end
50          else begin
51              Bout <= 1'b0;
52              case (fstate)
53                  S_0: begin
54                      if (~(Bin))
55                          reg_fstate <= S_1;
56                      else if (Bin)
57                          reg_fstate <= S_2;
58                      // Inserting 'else' block to prevent latch inference
59                      else
60                          reg_fstate <= S_0;
61
62                      Bout <= 1'b0;
63                  end
64                  S_1: begin
65                      if (~(Bin))
66                          reg_fstate <= S_3;
67                      else if (Bin)
68                          reg_fstate <= S_4;
69                      // Inserting 'else' block to prevent latch inference
70                      else
71                          reg_fstate <= S_1;
72
73                      Bout <= ~(Bin);
74                  end
75                  S_2: begin
76                      reg_fstate <= S_4;
77
78                      Bout <= Bin;
79                  end
80                  S_3: begin
```

```
80              S_3: begin
81                  reg_fstate <= S_5;
82
83                  Bout <= Bin;
84              end
85              S_4: begin
86                  if (~(Bin))
87                      reg_fstate <= S_5;
88                  else if (Bin)
89                      reg_fstate <= S_6;
90                  // Inserting 'else' block to prevent latch inference
91                  else
92                      reg_fstate <= S_4;
93
94                  Bout <= ~(Bin);
95              end
96              S_5: begin
97                  reg_fstate <= S_0;
98
99                  Bout <= Bin;
100             end
101             S_6: begin
102                 reg_fstate <= S_0;
103
104                 Bout <= ~(Bin);
105             end
106             default: begin
107                 Bout <= 1'bx;
108                 $display ("Reach undefined state");
109             end
110         endcase
111     end
112 end
113 endmodule // BCDExcess3
114
```

Simulate the system



Input = 1000
Output = 1011

Input = 1111 (illegal)
Output = 0010

Notice the illegal output is different from the first design.