

CX1005

Digital Logic

Finite State Machines in Verilog

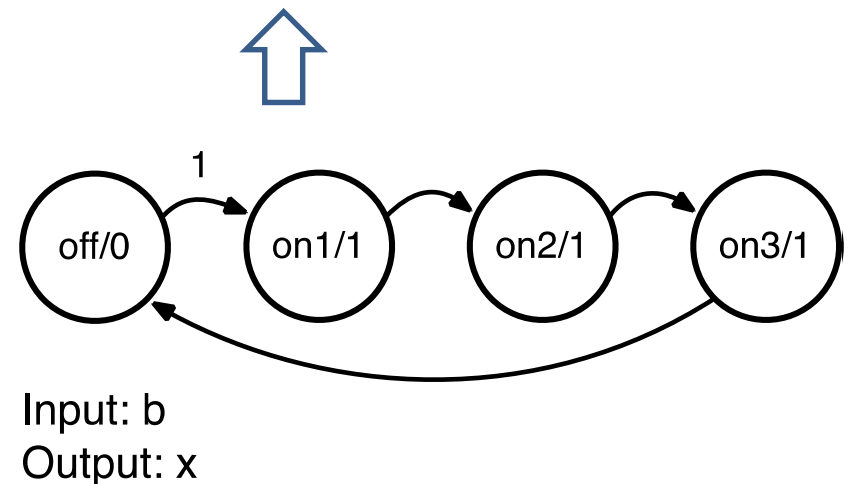
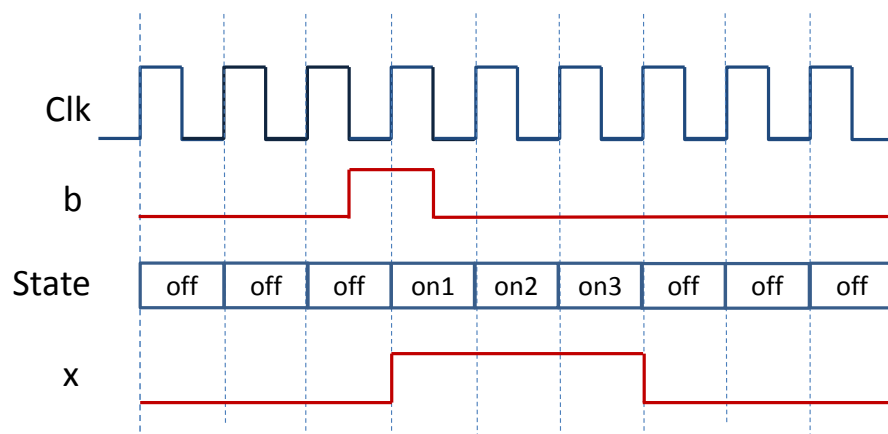
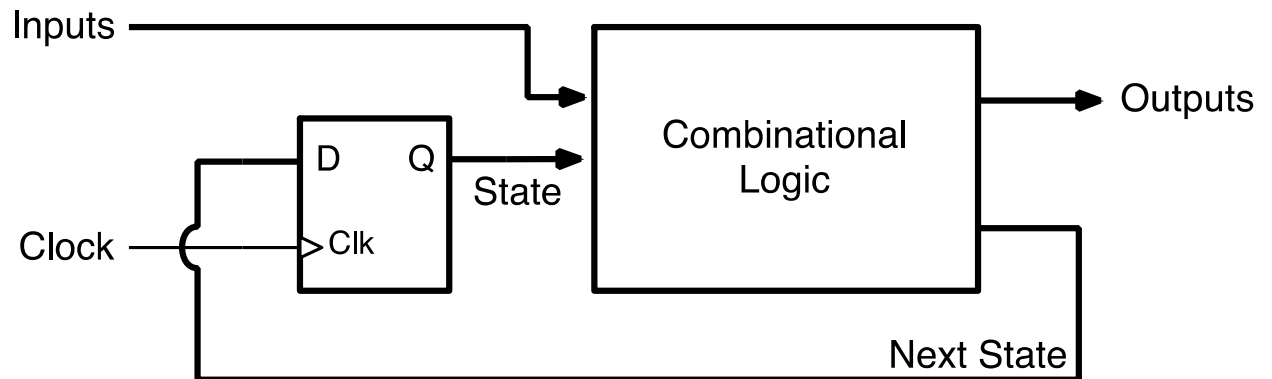
Mapping to Implementation

- This FSM is easy to understand, but how can we then turn this into a circuit?
- The first thing to consider is the structure we are trying to build:
 - A state register that holds the current state
 - Some logic to determine what the next state should be
 - Some logic to drive the output correctly
- At each clock cycle, the state will be updated to whatever was determined to be the next state

Mapping to Implementation

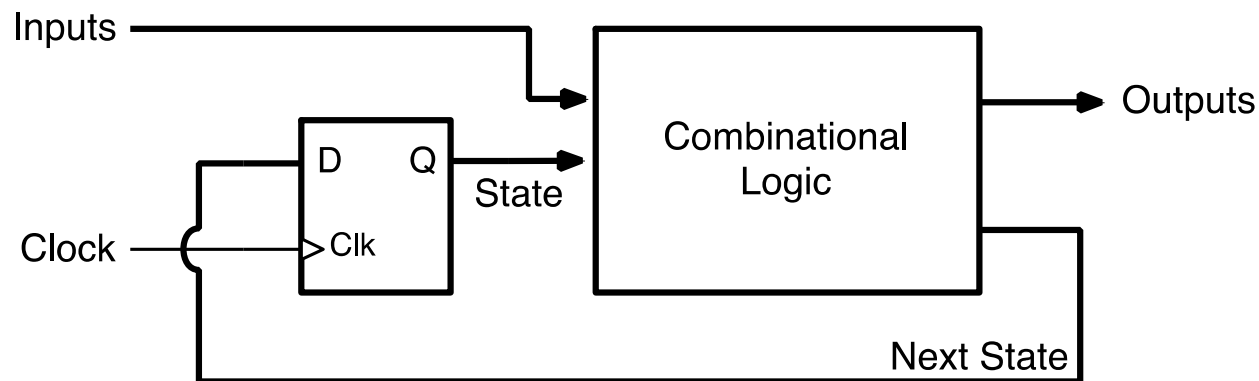
- Hence, the basic structure looks like this:

- A state register that holds the current state
- Some logic to determine what the next state should be
- Some logic to drive the output correctly



Mapping to Implementation

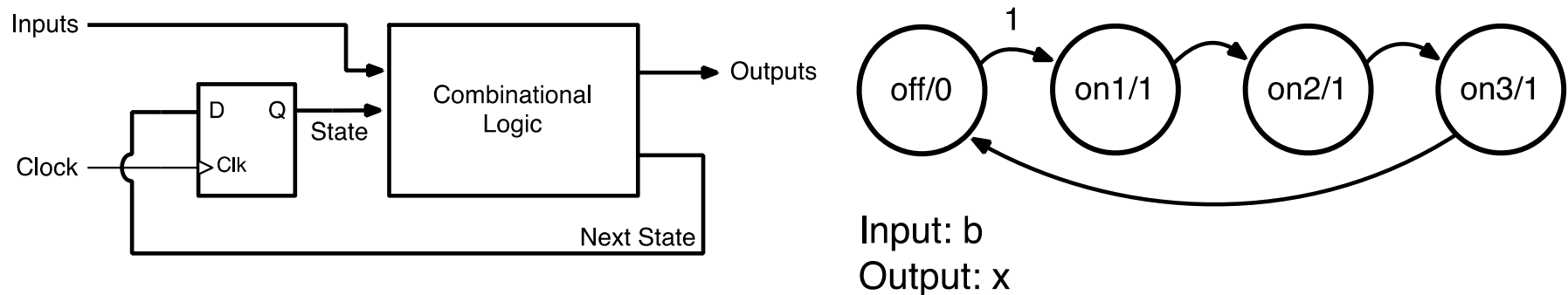
- Hence, the basic structure looks like this:



- The combinational logic the following based on the **current state** and **inputs**
 - **Next State**
 - **Outputs**
- We must turn the states into some binary representation, then we can use those values and the inputs to write the logic to produce the next state and outputs

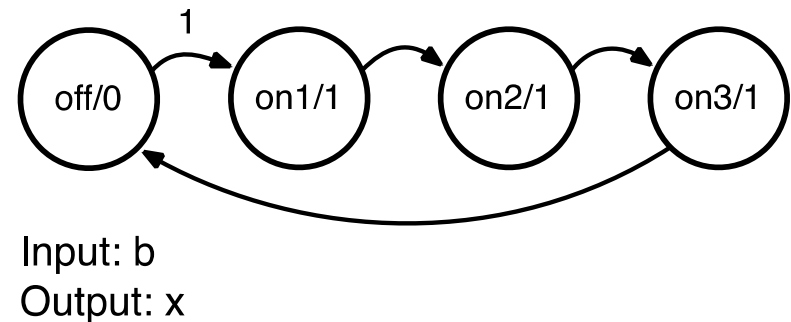
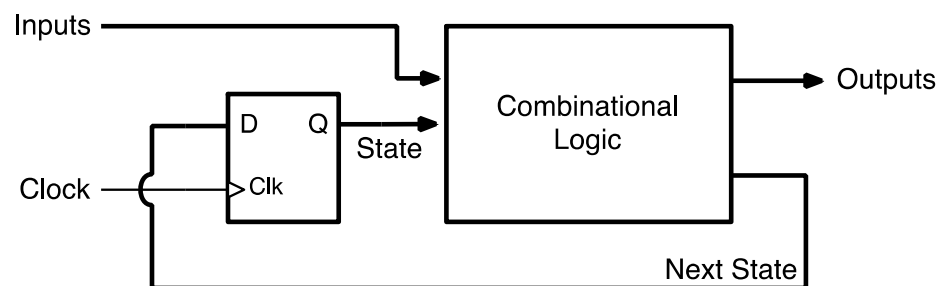
Mapping Finite State Machines

- The state register stores the current state
- To encode the state, we must assign a binary value to each state
- Hence for m states, we need $(\log_2 m)$ bits to encode the state
- For four states, we would need two bits, so we could name them $s[1:0]$
- This represents the current state



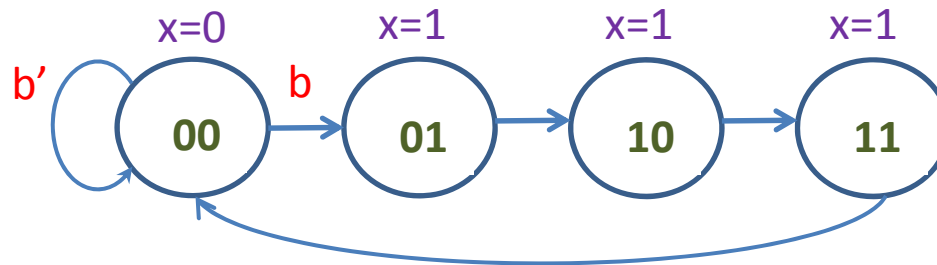
Mapping Finite State Machines

- We then need corresponding values to represent the next state, $ns[1:0]$
- We must then determine the combinational logic that gives the corresponding next state for the current state, as per our transition diagram
- Separately, we can then work out the logic for the output



Mapping Example

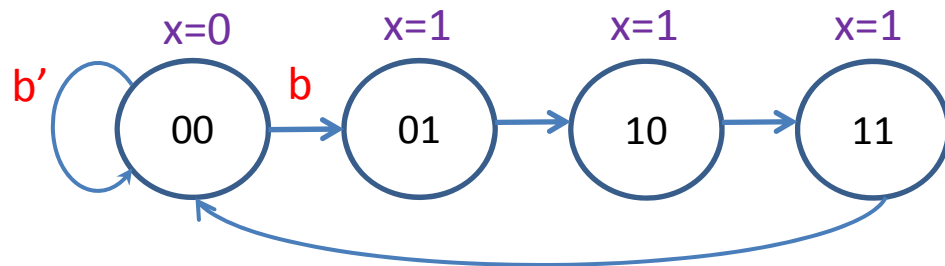
- First, we take our transition diagram



- We then give each state a unique binary value
- We call this encoding the states
- Any encoding will work as long as each state is unique

Mapping Example

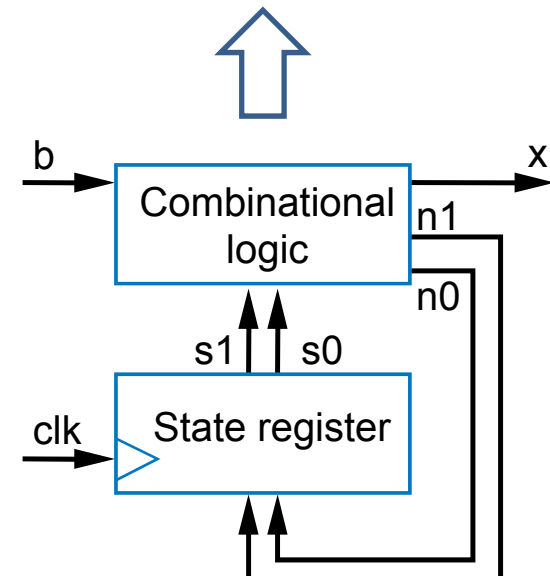
- The circuit we are designing looks like this:
- We now have a value of $s1$, $s0$ for each state
- We can use this to build a table that captures the transitions to $ns1$, $ns0$, and the output



Combinational Logic

Input: $s1$, $s0$, b

Output: $n1$, $n0$, x




Current State: $s[1:0]$

Next State: $n[1:0]$


Mapping Example

- The transition table:


	Inputs			Outputs		
	s1	s0	b	x	n1	n0
<i>off</i>	0	0	0	0	0	0
	0	0	1	0	0	1
<i>on1</i>	0	1	0	1	1	0
	0	1	1	1	1	0
<i>on2</i>	1	0	0	1	1	1
	1	0	1	1	1	1
<i>on3</i>	1	1	0	1	0	0
	1	1	1	1	0	0




Current State



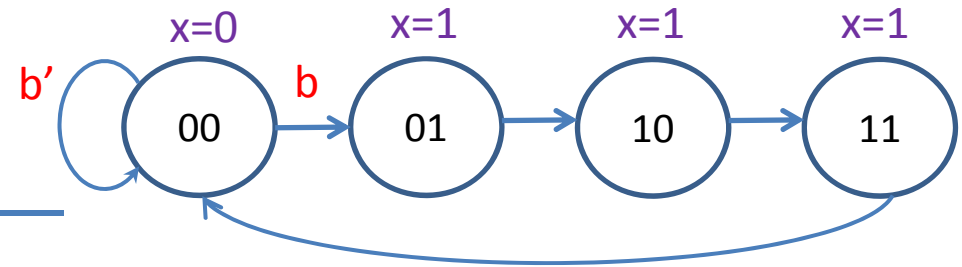
Input



Output



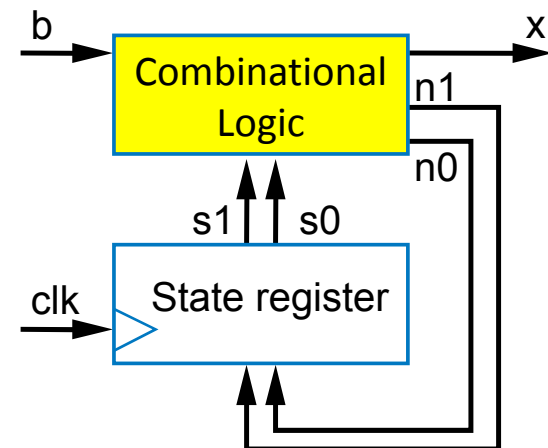
Next State



Combinational Logic

Input: s1, s0, b

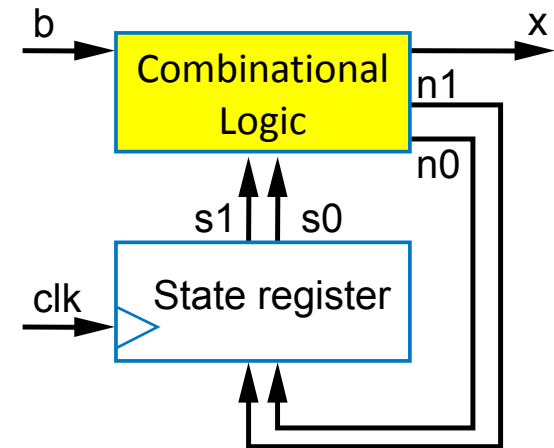
Output: n1, n0, x



Mapping Example

- Now determine the combinational logic:

Inputs				Outputs		
	s1	s0	b	x	n1	n0
<i>off</i>	0	0	0	0	0	0
	0	0	1	0	0	1
<i>on1</i>	0	1	0	1	1	0
	0	1	1	1	1	0
<i>on2</i>	1	0	0	1	1	1
	1	0	1	1	1	1
<i>on3</i>	1	1	0	1	0	0
	1	1	1	1	0	0



$$x = s1 + s0$$

(note that $x=1$ if $s1=1$ or $s0=1$)

$$n1 = s1's0b' + s1's0b + s1s0'b' + s1s0'b$$

$$n1 = s1's0 + s1s0'$$

$$n0 = s1's0'b + s1s0'b' + s1s0'b$$

$$n0 = s1's0'b + s1s0'$$

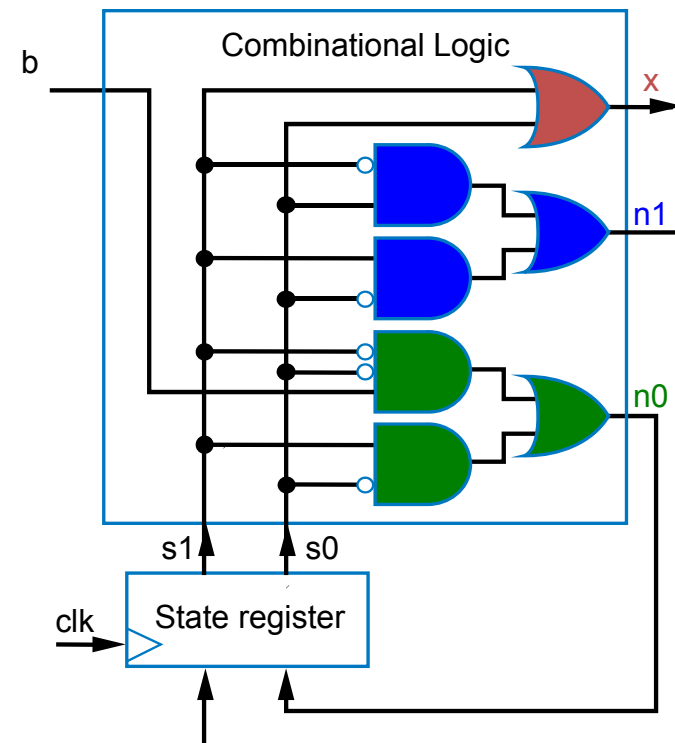
Mapping Example

- The circuit can then be implemented:

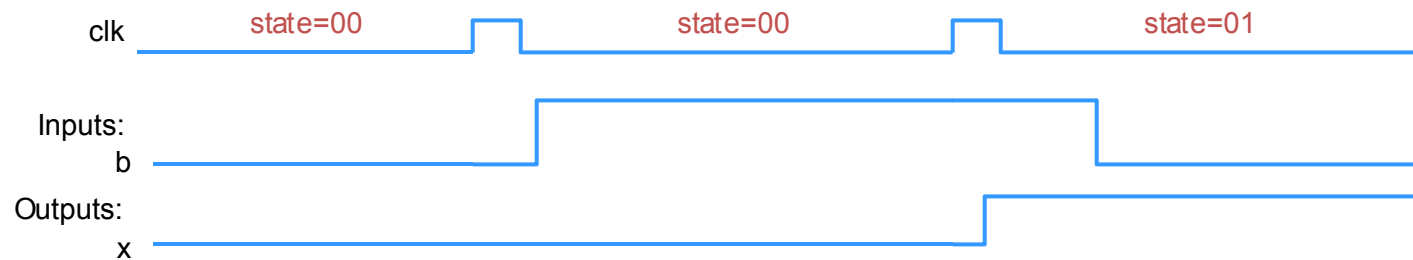
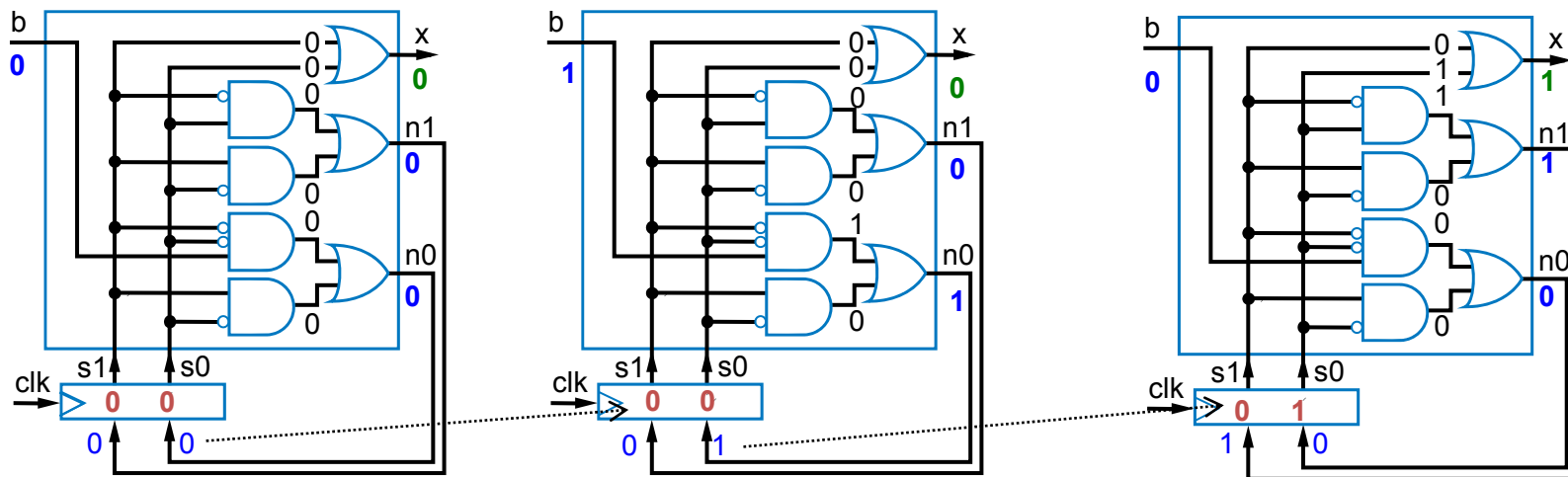
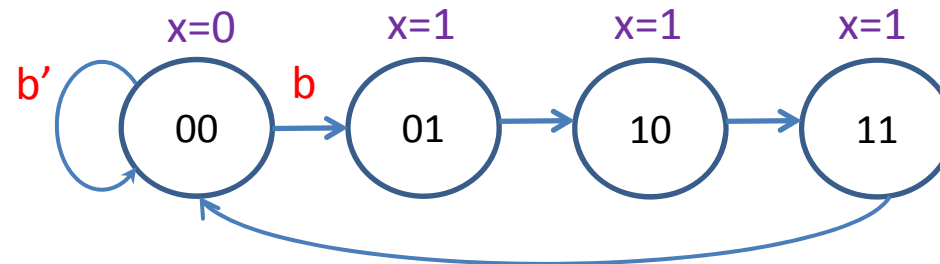
$$x = s1 + s0$$

$$n1 = s1' s0 + s1 s0'$$

$$n0 = s1' s0' b + s1 s0'$$



Mapping Example



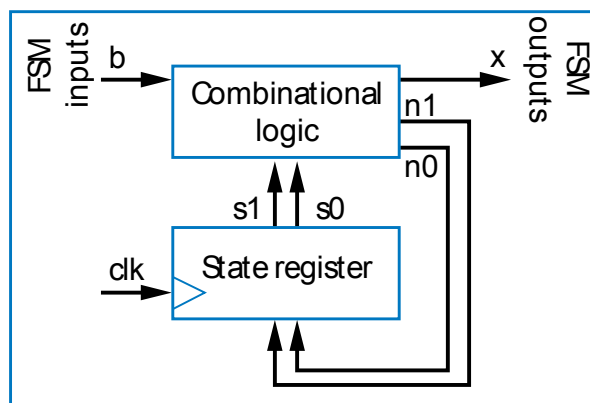
Mapping Finite State Machines

- Hence, the stages of implementing FSMs are:
 - Draw state transition diagram using names for states
 - Construct state transition table using names – *optional*
 - Encode the states
 - Write a state transition table using encoded states
 - Determine the logic equations for the next state bits and the outputs
 - Implement the circuit

- Note that different encodings will give different circuits, but the resulting functionality will be identical

Example in Verilog

- So to implement the previous example, we simply implement the logic in Verilog
- The three equations we obtained earlier plus a state register will give the full implementation
- Three assign statements implement the combinational logic
- A single register block creates the state register

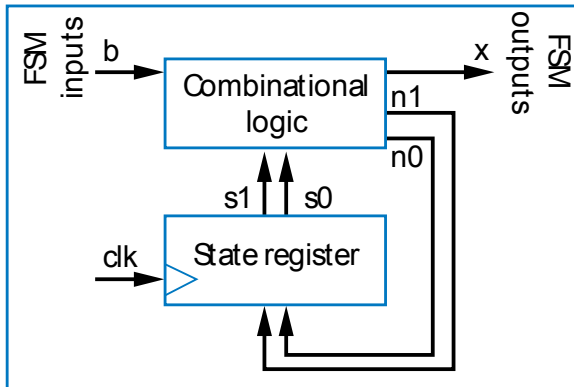


$$x = s1 + s0$$

$$n1 = s1' s0 + s1 s0' = s1 \text{ xor } s0$$

$$n0 = s1' s0' b + s1 s0'$$

Example in Verilog



$$x = s1 + s0$$

$$n1 = s1's0 + s1s0' = s1 \text{ xor } s0$$

$$n0 = s1's0'b + s1s0'$$

```
module pulse3 (input b,
               input clk, rst,
               output x);

  wire n1, n0;
  reg s1, s0;

  assign n1 = s1 ^ s0;
  assign n0 = (~s1&~s0&b) | (s1&~s0);
  assign x = s1 | s0;

  always @ (posedge clk)
  begin
    if(rst) begin
      s1 <= 1'b0;
      s0 <= 1'b0;
    end else begin
      s1 <= n1;
      s0 <= n0;
    end
  end
end
endmodule
```


Reset in State Machines

- Whenever implementing a finite state machine, **always** ensure the state register has a reset and that you have defined a valid state
- Otherwise, we can't always predict which state the FSM will start in
- If the FSM starts in an invalid state, we have no idea where it will transition, since we may not have fully specified the transition logic
- This is especially important when the number of states is not a power of two, and hence some state values may be unused

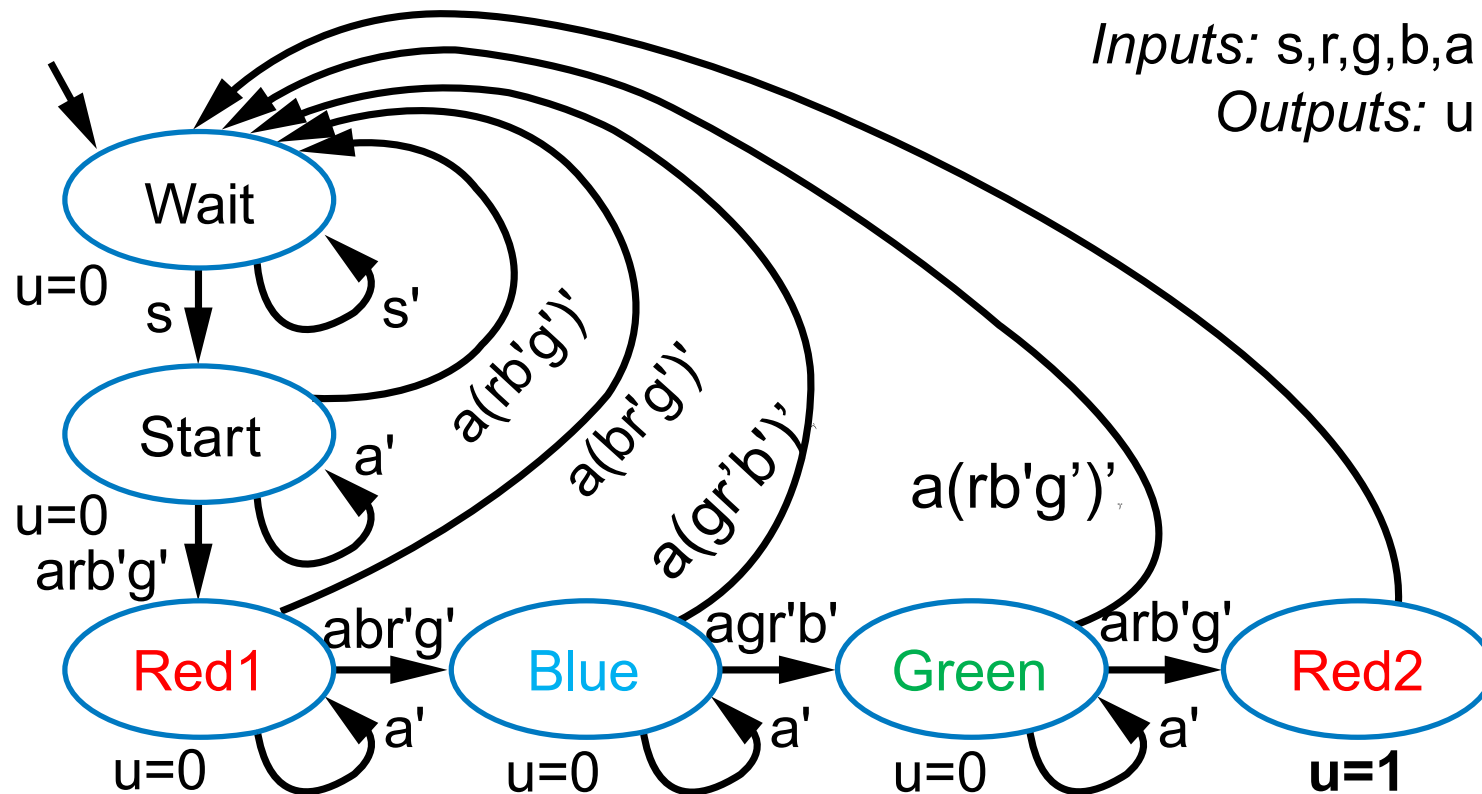
		n_1	n_0	
0 0		0	1	$n_1 = S_1 \text{ XOR } S_0$
0 1		1	0	$n_0 = S_0'$
1 0		1	1	
1 1		0	0	

More Complex FSMs

- The previous example had just four states
- In such cases, determining the transition logic for the individual states is easy, by looking at the transition table
- For more complex state machines, we can use Verilog's behavioral description to do this for us
- A combinational always block with a case statement can capture transitions in an easy to read manner
- We can implement the output logic separately to the transitions if that helps

A More Complex Example

- Consider the sequence detector from the last lecture:



Verilog Constants

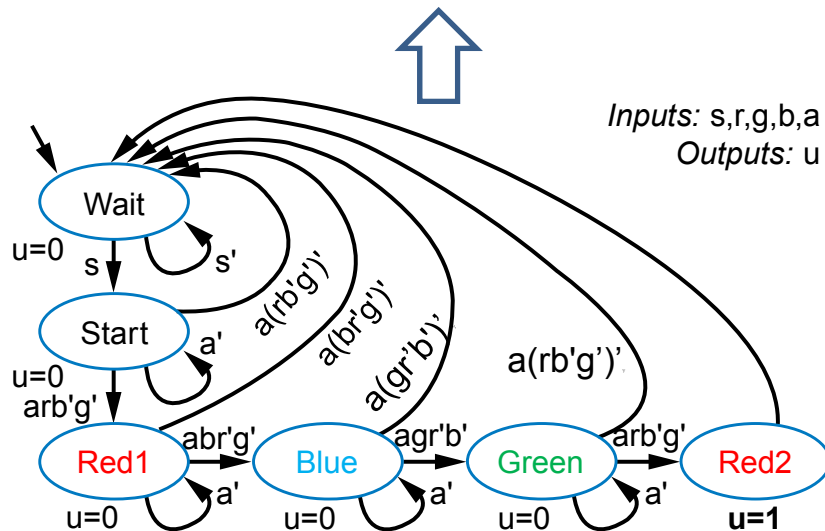
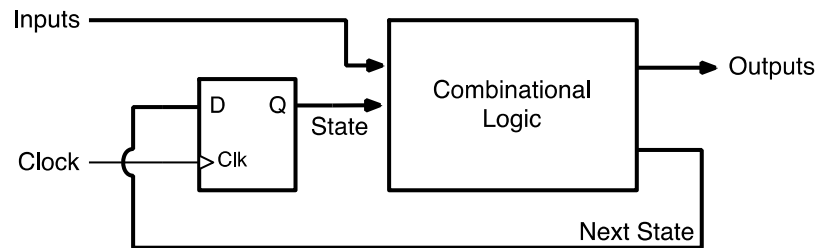
- First we need to encode the states:
 - Wait = 000, Start = 001, Red1 = 010, Blue = 011, Green = 100, Red2 = 101
- As we assign each state a binary value, we'd still rather not deal with binary numbers in our Verilog code.
- We can use the Verilog parameter keyword to declare named constants:

```
parameter st1 = 2'b00, st2 = 2'b01, st3 = 2'b10;
```

- We can then use these names in place of the values anywhere in the code.

```
nst = st2;
```

A More Complex Example



```

module seqdet (input s, r, g, b, a,
               input clk, rst,
               output u);

parameter waite=3'b000, start=3'b001,
red1=3'b010, blue=3'b011,
green=3'b100, red2=3'b101;

reg [2:0] nst, st;

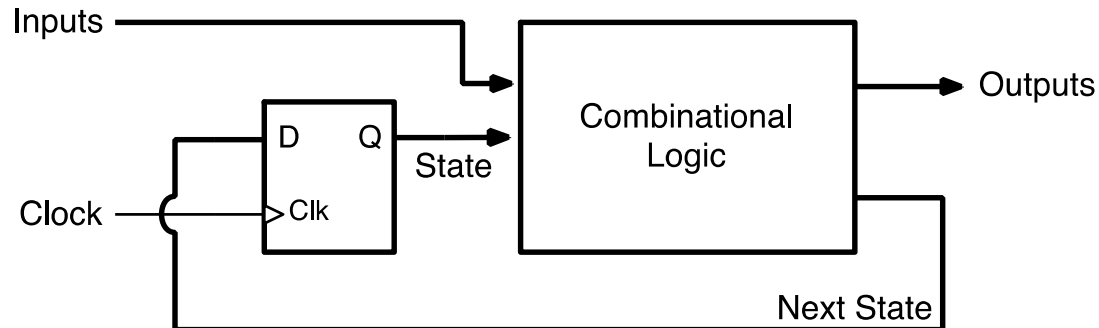
assign u = (st == red2);
//u = (st == red2) ? 1'b1 : 1'b0;

always @ (posedge clk)
begin
    if(rst)
        st <= waite;
    else
        st <= nst;
end

/* continued... */
endmodule
    
```

A More Complex Example

- On the previous slide:
 - We declare two 3-bit reg signals (since both will be assigned from always blocks)
 - The output is high only when the current state is Red2 (101), so we can assign it directly
 - The state register process is always the same



- We still need to describe the combinational logic to determine the next state (based on current state and inputs)

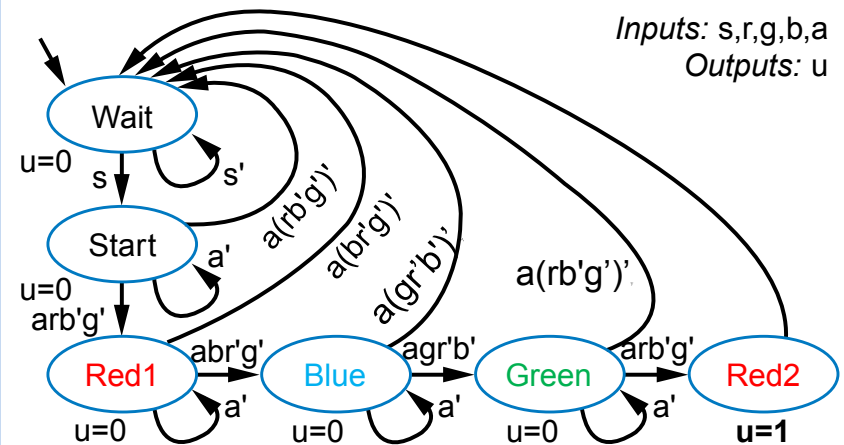
A More Complex Example

```

always @ *
begin
    nst = st;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: nst = waite;

        default: nst = waite;
    endcase
end

```

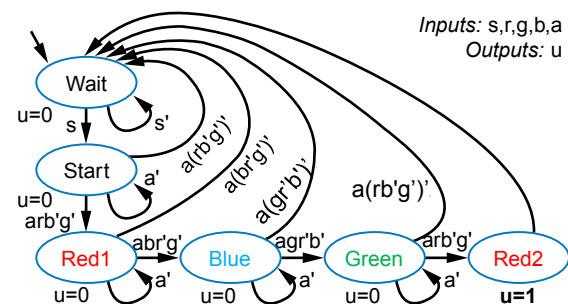


A More Complex Example

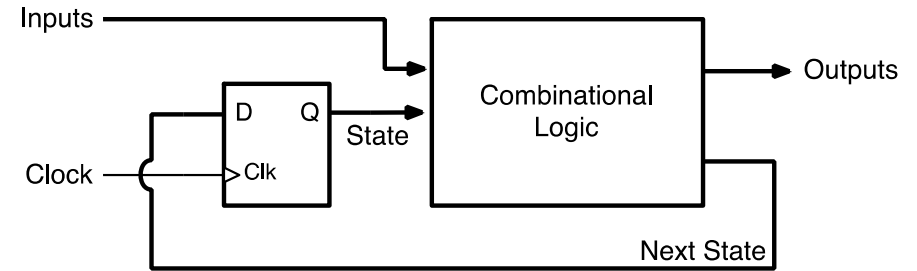
- For each state, we use the information in the diagram to determine the next state based on the inputs
- We can see that we check a (i.e. a button was pressed) then if the right single button was pressed, we move to the next state
- If any other or multiple buttons were pressed (else) we go to 000
 - The default case takes care of ensuring nst is assigned in all cases
- A default at the top says that if no assignment is made, we stay in the current state

```
always @ *
begin
    nst = st;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: nst = waite;

        default: nst = waite;
    endcase
end
```



A More Complex Example



```

module seqdet (input s, r, g, b, a,
               input clk, rst,
               output u);

parameter waite=3'b000, start=3'b001,
red1=3'b010, blue=3'b011,
green=3'b100, red2=3'b101;

reg [2:0] nst, st;

assign u = (st == red2);
//u = (st == red2) ? 1'b1 : 1'b0;

always @ (posedge clk)
begin
    if(rst)
        st <= waite;
    else
        st <= nst;
end

...

endmodule
  
```

```

always @ *
begin
    nst = st;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: nst = waite;

        default: nst = waite;
    endcase
end
  
```

A More Complex Example

- Note that nst is the next state value; it is only clocked into the state register at the next rising edge

```
module seqdet (input s, r, g, b, a,
               input clk, rst,
               output u);

parameter waite=3'b000, start=3'b001,
red1=3'b010, blue=3'b011,
green=3'b100, red2=3'b101;

reg [2:0] nst, st;

assign u = (st == red2);
//u = (st == red2) ? 1'b1 : 1'b0;

always @ (posedge clk)
begin
    if(rst)
        st <= waite;
    else
        st <= nst;
end

...

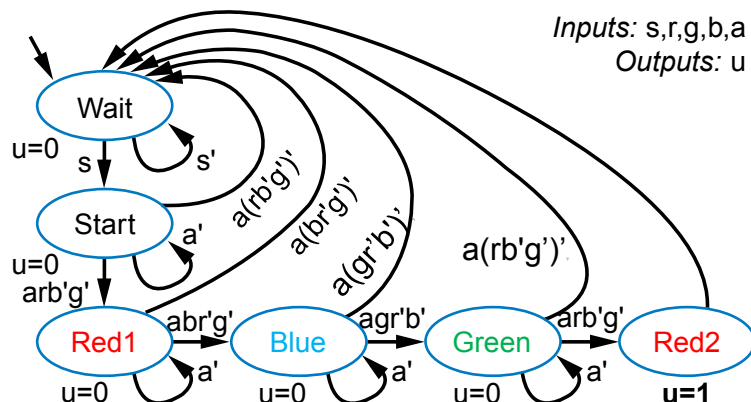
endmodule
```

```
always @ *
begin
    nst = st;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: nst = waite;

        default: nst = waite;
    endcase
end
```

A More Complex Example

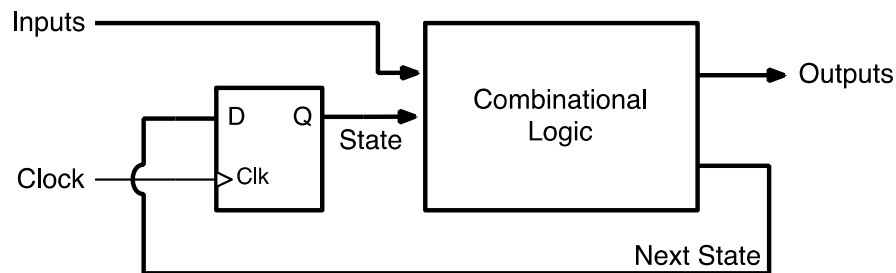
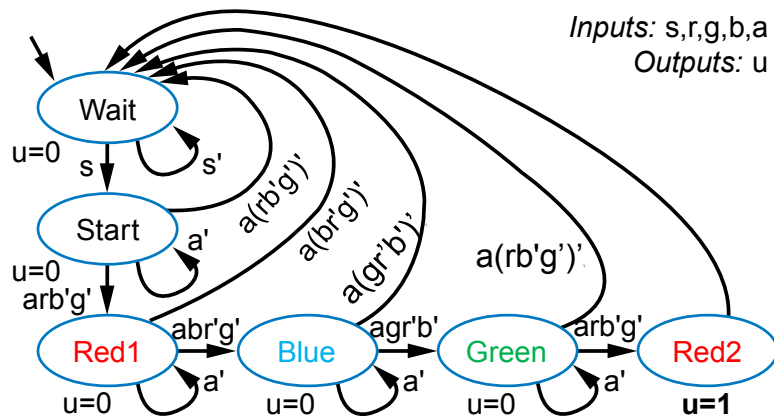
- We could also have added the output assignment to the same always block as the transitions
- Remember to change the u port to **reg**



```

always @ *
begin
    nst = st; u = 1'b0;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: begin
            nst = waite;
            u = 1'b1;
        end
        default: nst = waite;
    endcase
end
  
```

A More Complex Example



```

module seqdet (input s, r, g, b, a,
               input clk, rst,
               output u);

parameter waite=3'b000, start=3'b001, red1=3'b010,
blue=3'b011, green=3'b100, red2=3'b101;

reg [2:0] nst, st;

assign u = (st == red2); → Logic to drive output

always @ (posedge clk)
begin
    if(rst)
        st <= waite;
    else
        st <= nst;
end } State register

always @ *
begin
    nst = st;
    case(st)
        waite: if(s) nst = start;
        start: if(a)
            if(r&~b&~g) nst = red1;
            else nst = waite;
        red1: if(a)
            if(b&~r&~g) nst = blue;
            else nst = waite;
        blue: if(a)
            if(g&~r&~b) nst = green;
            else nst = waite;
        green: if(a)
            if(r&~g&~b) nst = red2;
            else nst = waite;
        red2: nst = waite;

        default: nst = waite;
    endcase
end } Logic to determine next state
end
endmodule

```

FSM Structure in Verilog

- So the generalized FSM structure looks like this:

```
module seqdet (input /* FSM inputs */
               input clk, rst,
               output /* FSM outputs */);

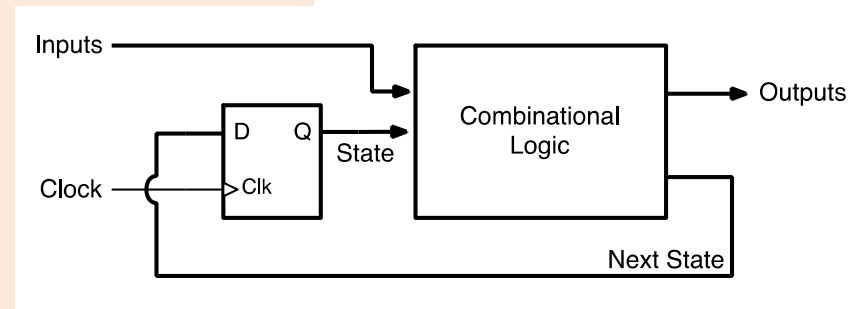
parameter st1 = ... /* state names and assignments */

reg [2:0] nst, st; /* next and current state signal
*/

/* state register always block
always the same structure
st <= nst, with a reset to initial */

/* state transition always block
generally uses a case statement
assigning to nst based on st and inputs */

/* output assignment can be separate using
assign or always block, or combined in
state transition always block */
endmodule
```



Notes on Implementing FSMs

- It is possible to write a whole FSM in a single synchronous always block
 - but that is harder to verify
- Hence, always separate the state register: a simple block that assigns next state to current state at each clock edge
- Ensure you reset the state register to a valid state
- If the output and transitions can be combined cleanly, you can do that
- Otherwise, assign the output separately

Notes on Implementing FSMs

- Use a default next state (and output) assignment at the top of the state transition block to minimize the number of statements – only other cases needed
- Always ensure next state is assigned in every case
- Add a default case that transitions other invalid states to the initial state
- Using a combinational always block, it becomes easy to test that it works, since we always know what outputs we should get, based on the state transition diagram

```
always @ *  
begin  
    nst = st;  
    case(st)  
        waite: if(s) nst = start;  
        start: if(a)  
            if(r&~b&~g) nst = red1;  
            else nst = waite;  
        red1: if(a)  
            if(b&~r&~g) nst = blue;  
            else nst = waite;  
        blue: if(a)  
            if(g&~r&~b) nst = green;  
            else nst = waite;  
        green: if(a)  
            if(r&~g&~b) nst = red2;  
            else nst = waite;  
        red2: nst = waite;  
  
        default: nst = waite;  
    endcase  
end
```