Modeling of Finite State Machines

Debdeep Mukhopadhyay

Associate Professor
Dept of Computer Science and Engineering
NYU Shanghai and IIT Kharagpur

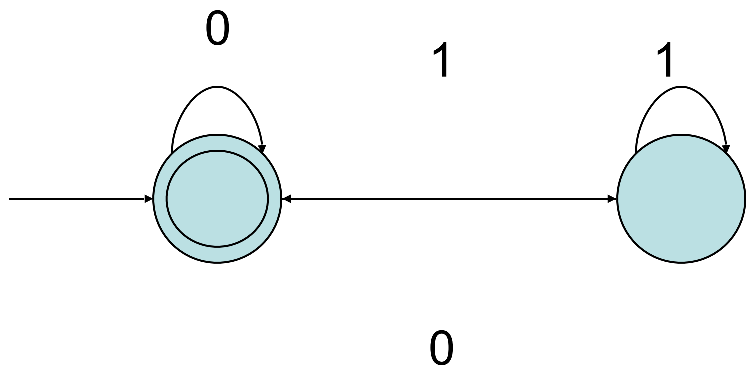
Modeling of Finite State Machines

Debdeep Mukhopadhyay
IIT *Kharagpur*

Definition

- 5 Tuple: (Q,Σ,δ,q_0,F)
- Q: Finite set of states
- Σ: Finite set of alphabets
- δ: Transition function
  - $Q \chi\ \Sigma \rightarrow\ \ Q$
- q_0 is the start state
- F is a set of accept states. They are also called final states.

Some Examples

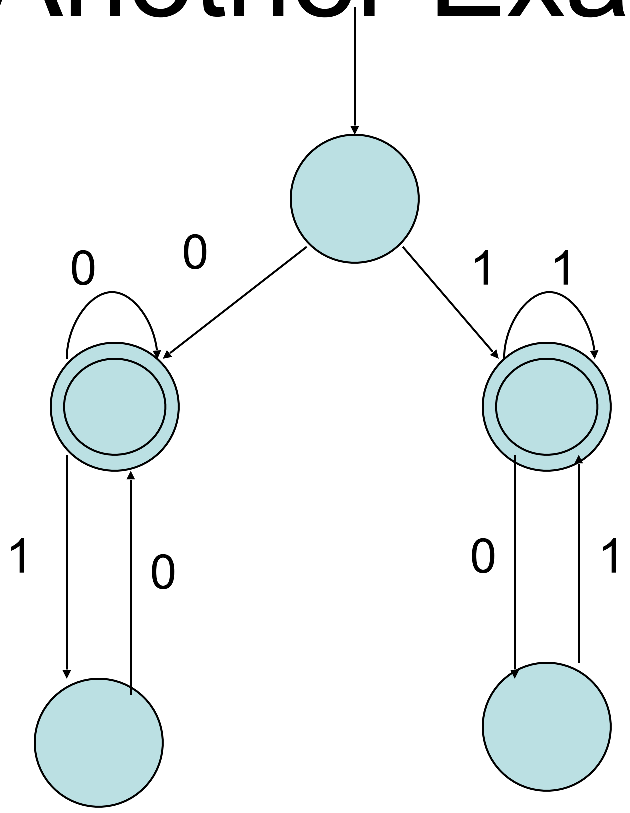


What does this FSM do?

It accepts the empty string or any string that ends with 0

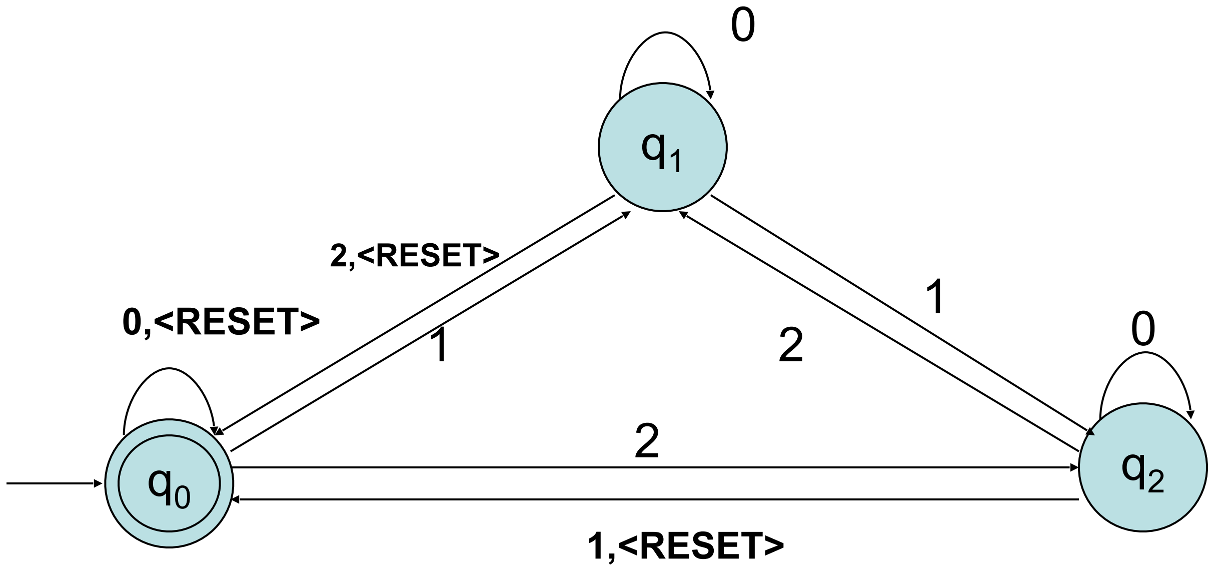
These set of strings which takes the FSM to its accepting states are often called **language** *of the automaton.*

Another Example



- *Accepts strings that starts and ends with the same bits.*

A more complicated example



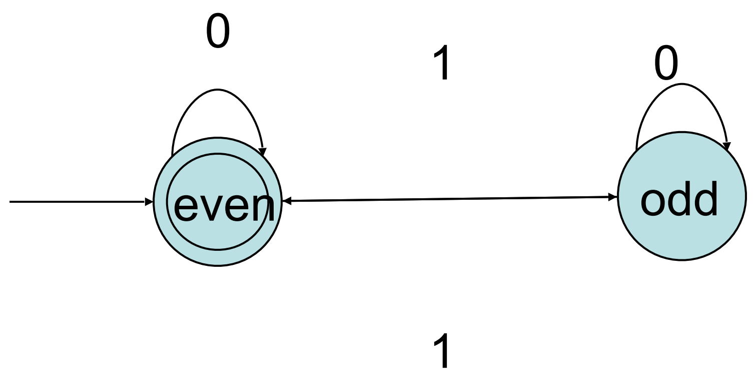
- FSM accepts if the running sum of the input strings is a multiple of 3.
- RESET symbol resets the running sum to 0.

Designing FSMs

- Its an art.
- Pretend to be an FSM and imagine the strings are coming one by one.
- Remember that there are finite states.
- So, you cannot store the entire string, but only crucial information.
- Also, you do not know when the string ends, so you should always be ready with an answer.

Example

- Design a FSM which accepts 0,1 strings which has an odd number of 1's.
- You require to remember whether there are odd 1's so far or even 1's so far.



Example

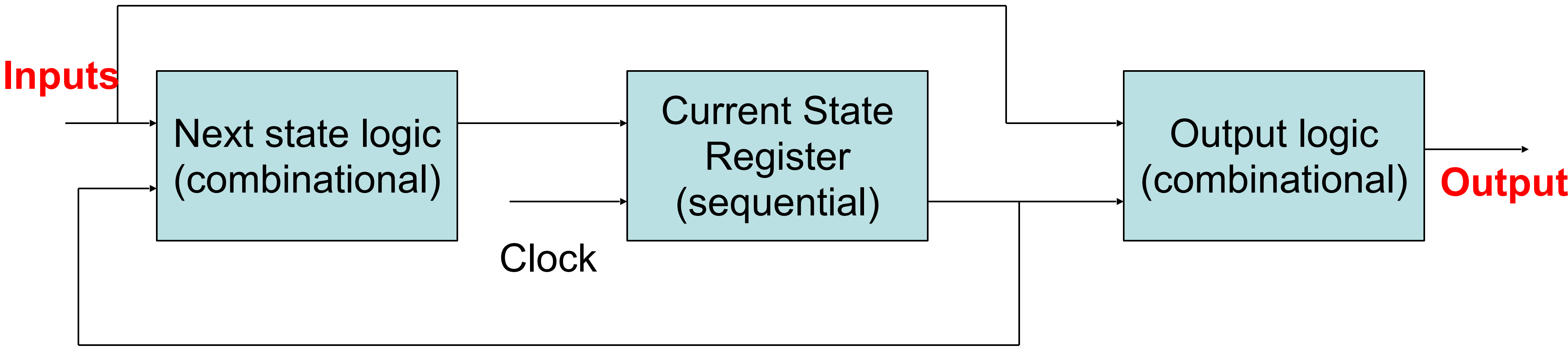
- Design a FSM that accepts strings that contain 001 as substrings.
- There are 4 possibilities
  - No string
  - seen a 0
  - seen a 00
  - seen a 001

Answer

$$\begin{tikzcd} {} \arrow[r] & q \arrow[loop above, "1"] \arrow[r, bend left, "0"] & q_0 \arrow[l, bend left, "1"] \arrow[r, "0"] & q_{00} \arrow[loop above, "0"] \arrow[r, "1"] & q_{001} \end{tikzcd}$$

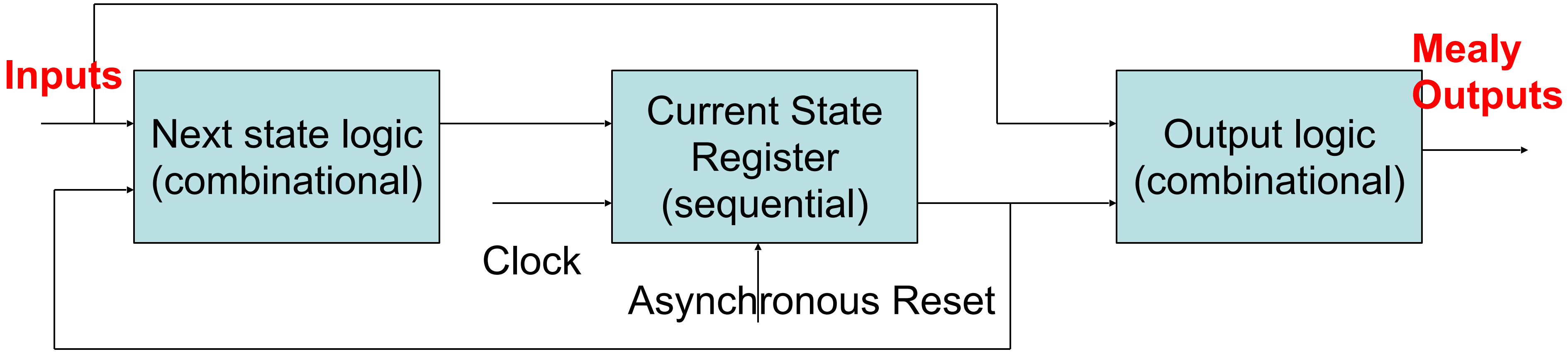
- Note that their may be cases where design of FSMS are not possible.
- Like design an FSM for strings which has the same number of 0's and 1's.

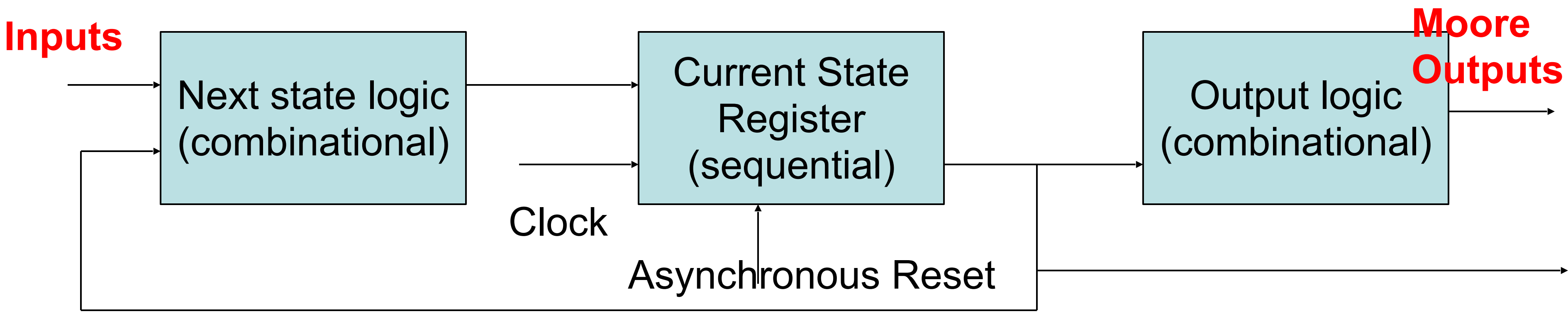
How to model such FSMs?


Inputs
Next state logic (combinational)
Current State Register (sequential)
Clock
Output logic (combinational)
Output

Simple Model of FSM

Mealy Machine/Moore Machine


Inputs
Next state logic (combinational)
Current State Register (sequential)
Clock
Asynchronous Reset
Output logic (combinational)
Mealy Outputs


Inputs
Next state logic (combinational)
Current State Register (sequential)
Clock
Asynchronous Reset
Output logic (combinational)
Moore Outputs

Modeling FSMs using Verilog

Issues

- State Encoding
  - sequential
  - gray
  - Johnson
  - one-hot

Encoding Formats

No	Sequential	Gray	Johnson	One-hot
0	000	000	0000	00000001
1	001	001	0001	00000010
2	010	011	0011	00000100
3	011	010	0111	00001000
4	100	110	1111	00010000
5	101	111	1110	00100000
6	110	101	1100	01000000
7	111	100	1000	10000000

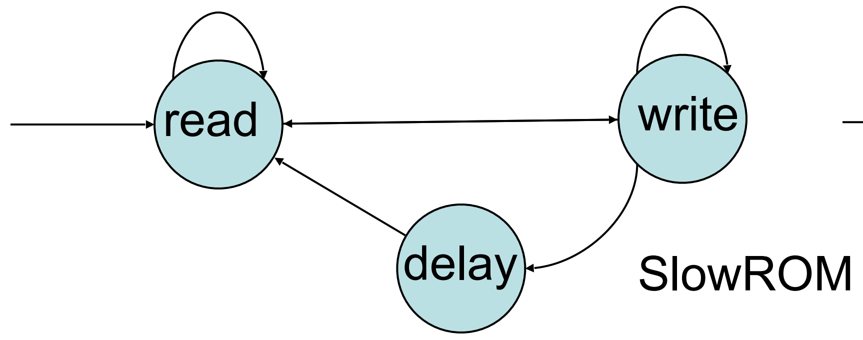
Comments on the coding styles

- **Binary:** Good for arithmetic operations. But may have more transitions, leading to more power consumptions. Also prone to error during the state transitions.
- **Gray:** Good as they reduce the transitions, and hence consume less dynamic power. Also, can be handy in detecting state transition errors.

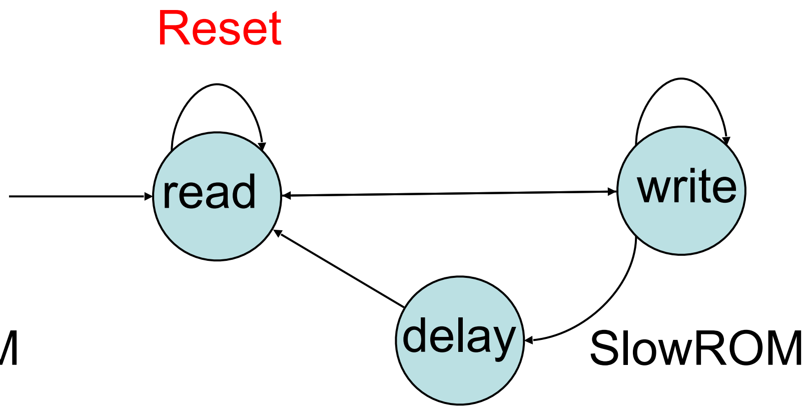
Coding Styles

- **Johnson:** Also there is one bit change, and can be useful in detecting errors during transitions. More bits are required, increases linearly with the number of states. There are unused states, so we require either explicit asynchronous reset or recovery from illegal states (even more hardware!)
- **One-hot:** yet another low power coding style, requires more no of bits. Useful for describing bus protocols.

Good and Bad FSM



FSM_BAD



FSM_GOOD

FSM State Diagram

Bad Verilog

```
always@(posedge Clock)
begin
 parameter ST_Read=0,ST_Write=1,ST_Delay=3;
   integer state;
   case(state)
        ST_Read:
          begin
              Read=1;
              Write=0;
              State=ST_Write;
          end
```

Bad Verilog

```
ST_Write:
    begin
        Read=0;
        Write=1;
        if(SlowRam)  State=ST_Delay;
        else State=ST_Read;
     end
```

Bad Verilog

```
ST_Delay:
    begin
         Read=0;
         Write=0;
         State=ST_Read;
    end
   endcase
  end
```

Why Bad?

- No reset. There are unused states in the FSM.
- Read and Write output assignments also infer an extra flip-flop.
- No default, latch is inferred.
- There is feedback logic.

Good verilog

```
always @(posedge Clock)
 begin
    if(Reset)
       CurrentState=ST_Read;
    else
       CurrentState=NextState;
  end
```

Good verilog

```
always@(CurrentState or SlowRAM)
begin
  case(CurrentState)
    ST_Read:
      begin
        Read=1; Write=0;
        NextState=ST_Write;
      end
```

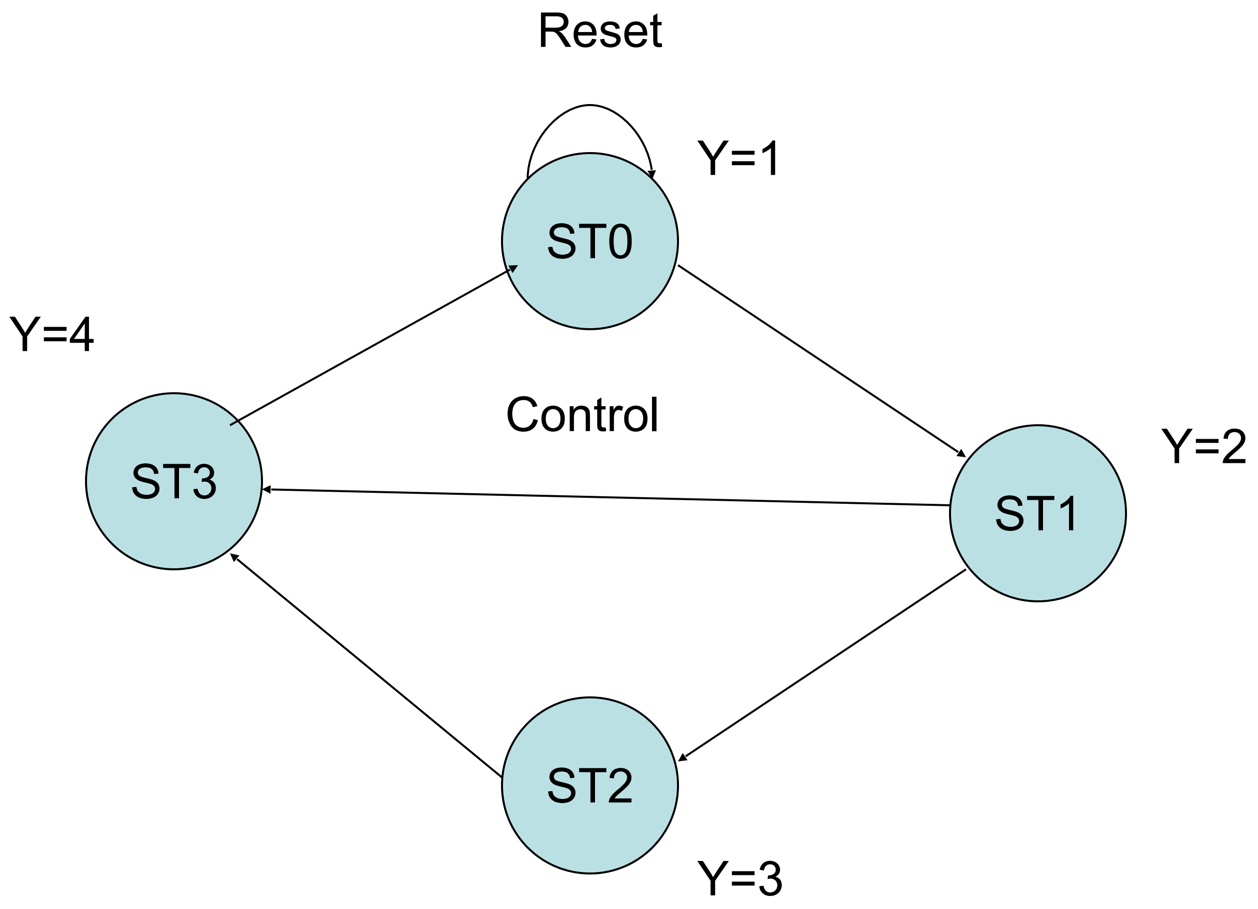
Good Verilog

```
ST_Write:
      begin
         Read=0; Write=1;
         if(SlowRAM) NextState=ST_Delay;
         else NextState=ST_Read;
      end
```

Good Verilog

```
ST_Delay:
    begin
       Read=0; Write=0; NextState=ST_Read;
    end
default:
     begin
        Read=0; Write=0; NextState=ST_Read;
     end
   endcase
end
```

One Bad and four good FSMs



Bad Verilog

```
always @(posedge Clock or posedge Reset)
begin
    if(Reset) begin
        Y=1;
        STATE=ST0;
     end
```

Bad verilog

```
else
    case(STATE)
        ST0: begin Y=1; STATE=ST1; end
        ST1: begin Y=2;
                  if(Control)  STATE=ST2;
                  else STATE=ST3;
          ST2: begin Y=3; STATE=ST3; end
          ST3: begin Y=4; STATE=ST0; end
     endcase
   end
```

Output Y is assigned under synchronous always block
so extra three latches inferred.

Good FSMs

- Separate CS, NS and OL
- Combined CS and NS. Separate OL
- Combined NS and OL. Separate CS

Next State (NS)

```
always @(control or currentstate)
begin
    NextState=ST0;
    case(currentstate)
      ST0: begin
               NextState=ST1;
             end
      ST1: begin …
      …
      ST3:
               NextState=ST0;
 endcase
end
```

Current State (CS)

```
always @(posedge Clk or posedge reset)
begin
    if(Reset)
          currentstate<=ST0;
     else
          currentstate<=Nextstate;
 end
```

Output Logic (OL)

```
always @(Currentstate)
begin
    case(Currentstate)
        ST0: Y=1;
        ST1: Y=2;
        ST2: Y=3;
        ST3: Y=4;
end
```

CS+NS

```
always@(posedge Clock or posedge reset)
begin
   if(Reset)
           State=ST0;
    else
        case(STATE)
            ST0: State<=ST1;
            ST1: if(Control) …
            ST2: …
            ST3: STATE<=ST0;
         endcase
   end
```

default not required
as it is in edge triggered
always statement

CS+NS

```
always @(STATE)
    begin
       case(STATE)
       ST0: Y=1;
       ST1: Y=2;
       ST2: Y=3;
       ST3: Y=4;
       default: Y=1;
       endcase
     end
```

default required
as it is in combinational
always statement

NS+OL

```
always @(Control or Currentstate)
begin
   case(Currentstate)
      ST0: begin
                Y=1;
                NextState=ST1;
              end
       ST1: …
       ST2: …
       ST3: …
       default: …
    endcase
  end
```

NS+OL

```
always @(posedge clock or posedge reset)
begin
  if(reset)
       Currentstate<=ST0;
  else
       Currentstate<=NextState;
end
```