Chapter 23
SystemVerilog State Machines

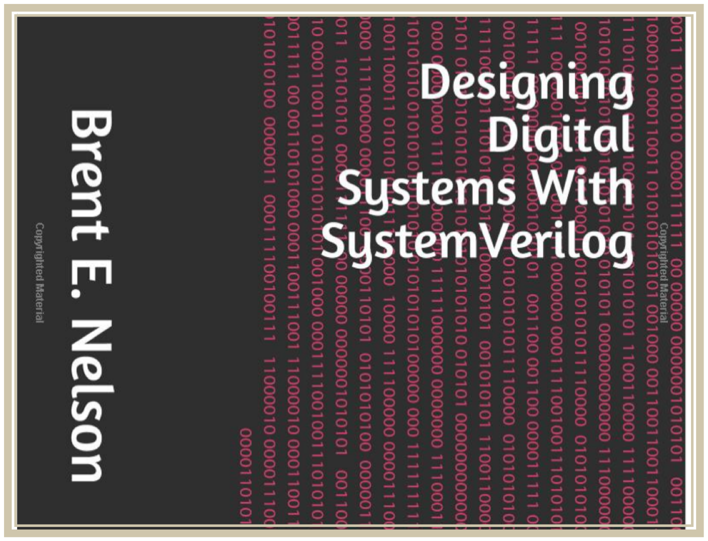
ECEn 220

Fundamentals of Digital Systems

BYU

BRIGHAM YOUNG

UNIVERSITY

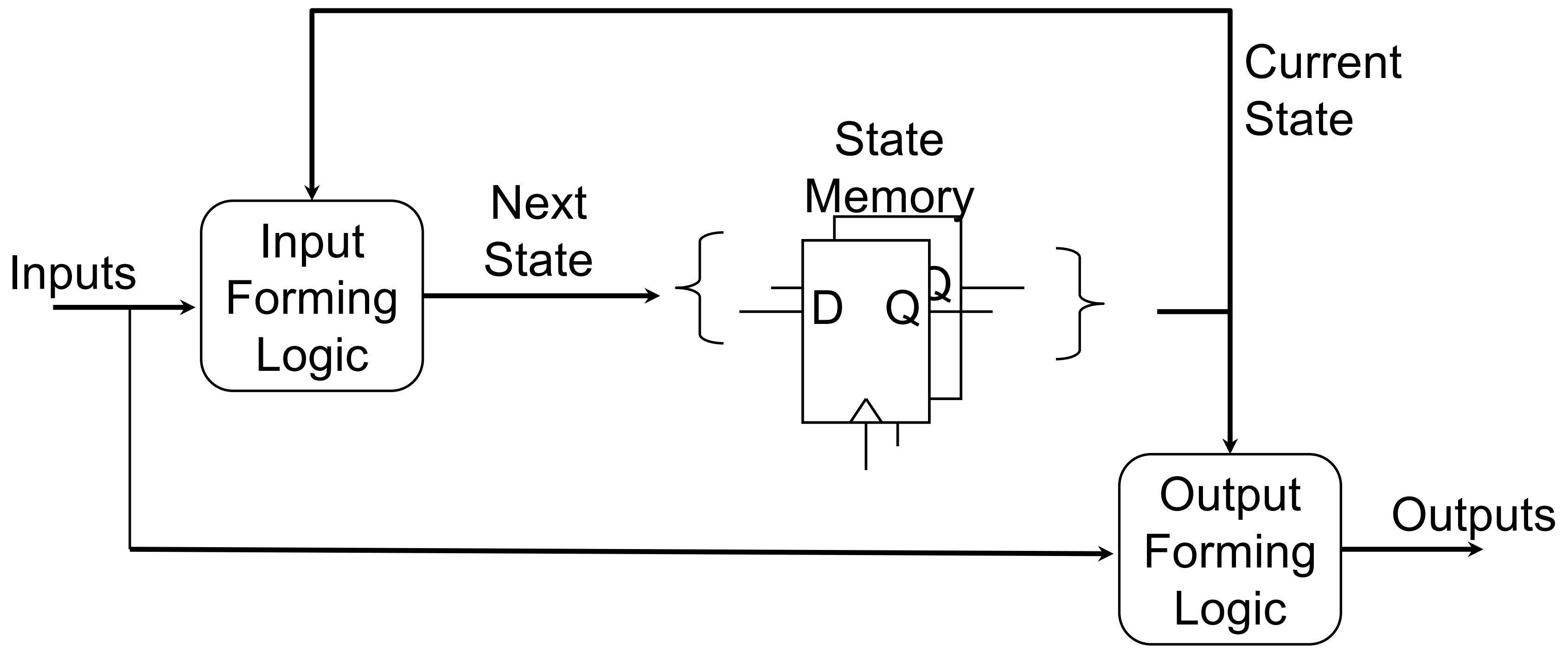

Designing Digital Systems With SystemVerilog

Brent E. Nelson

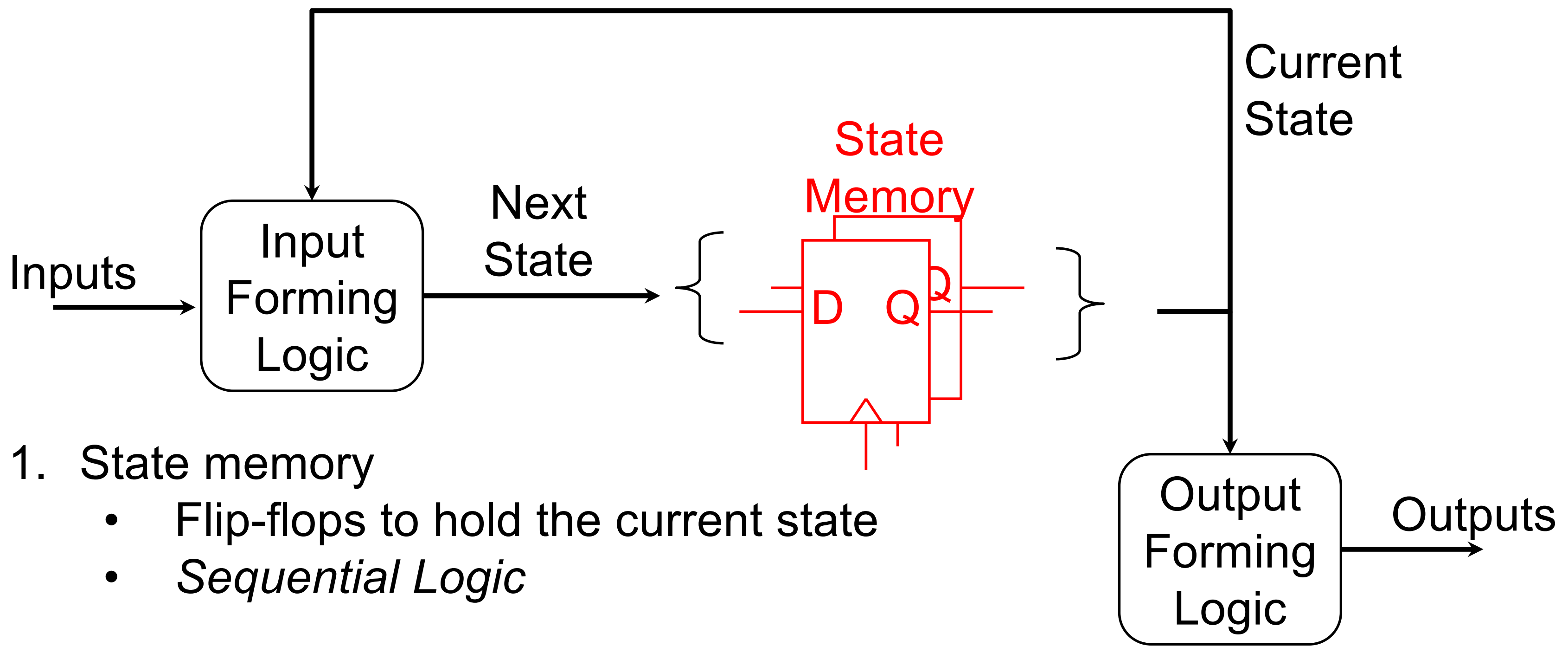
Copyrighted Material

BYU
Computer Engineering
Electrical Engineering

Components of an FSM

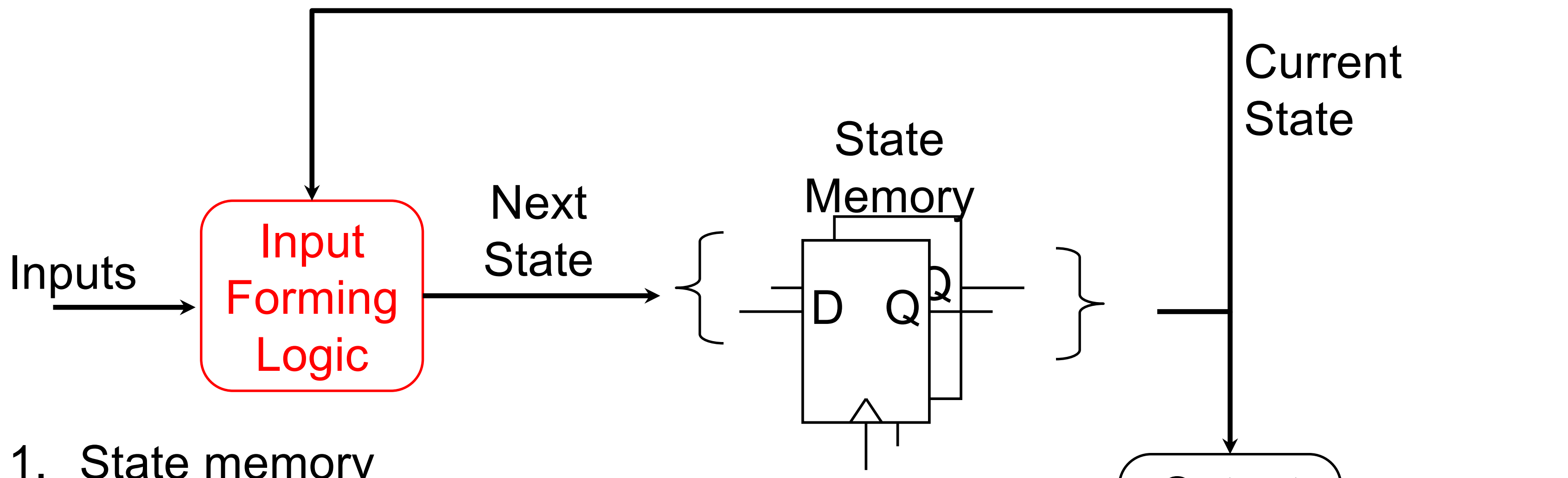


Components of a FSM



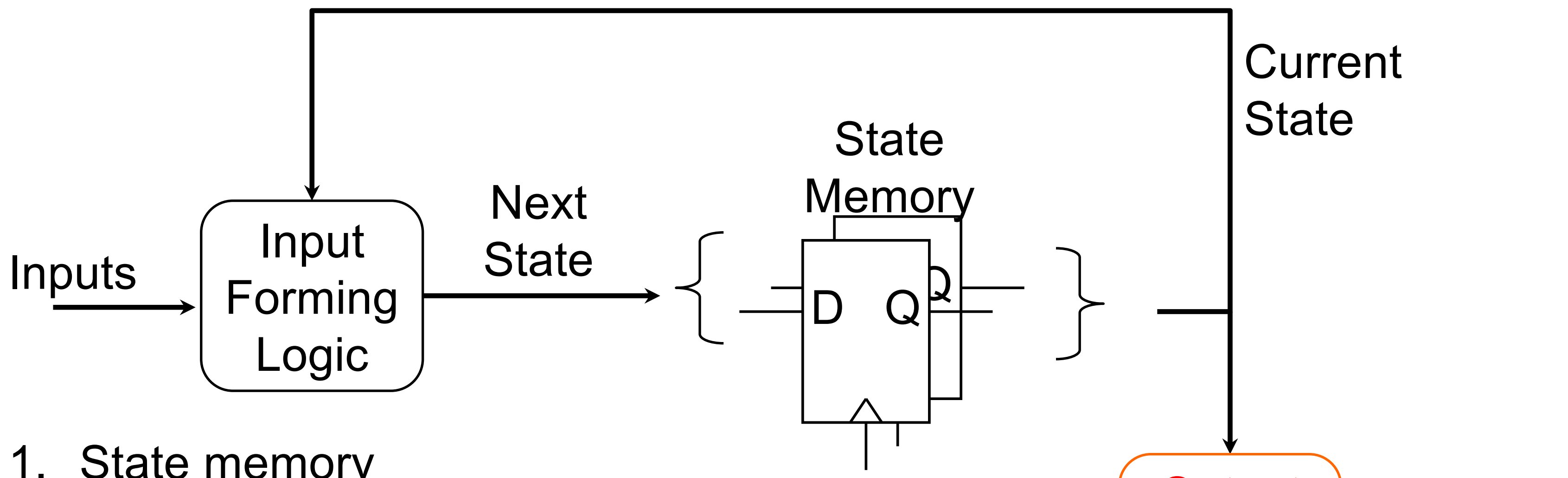
1. State memory
   - Flip-flops to hold the current state
   - *Sequential Logic*

Components of a FSM



1. State memory
   - Flip-flops to hold the current state
   - *Sequential Logic*
2. Input Forming Logic (IFL)
   - Logic to determine the *next state* of the FSM
   - *Combinational Logic*

Components of a FSM



1. State memory
   - Flip-flops to hold the current state
   - *Sequential Logic*
2. Input Forming Logic (IFL)
   - Logic to determine the *next state* of the FSM
   - *Combinational Logic*
3. Output Forming Logic (OFL)
   - Logic to determine FSM *outputs*
   - *Combinational logic*

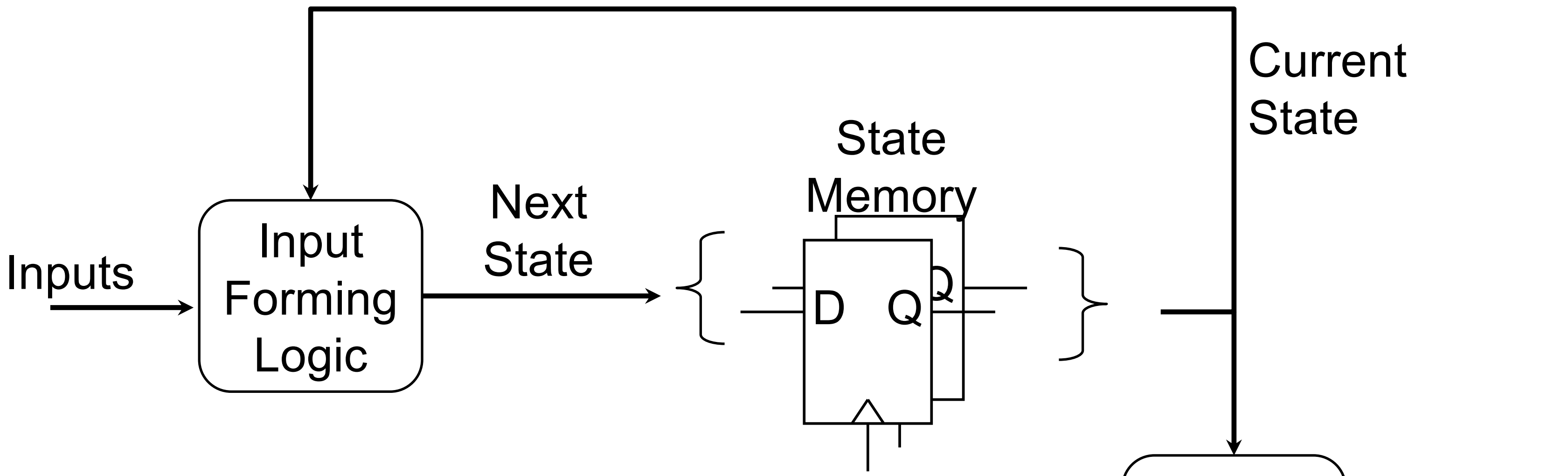
SystemVerilog FSMs

- SystemVerilog code is needed for <u>each</u> of the three components of an FSM
  - State FF code
  - IFL code
  - OFL code

State Machine Coding Styles

- One always_comb block for IFL and OFL and one always_ff block for state register
- One always ff block for the state register and IFL and the OFL is done using either dataflow assign statements or an always_comb block.
- Three separate blocks for the state register, IFL, and ODL

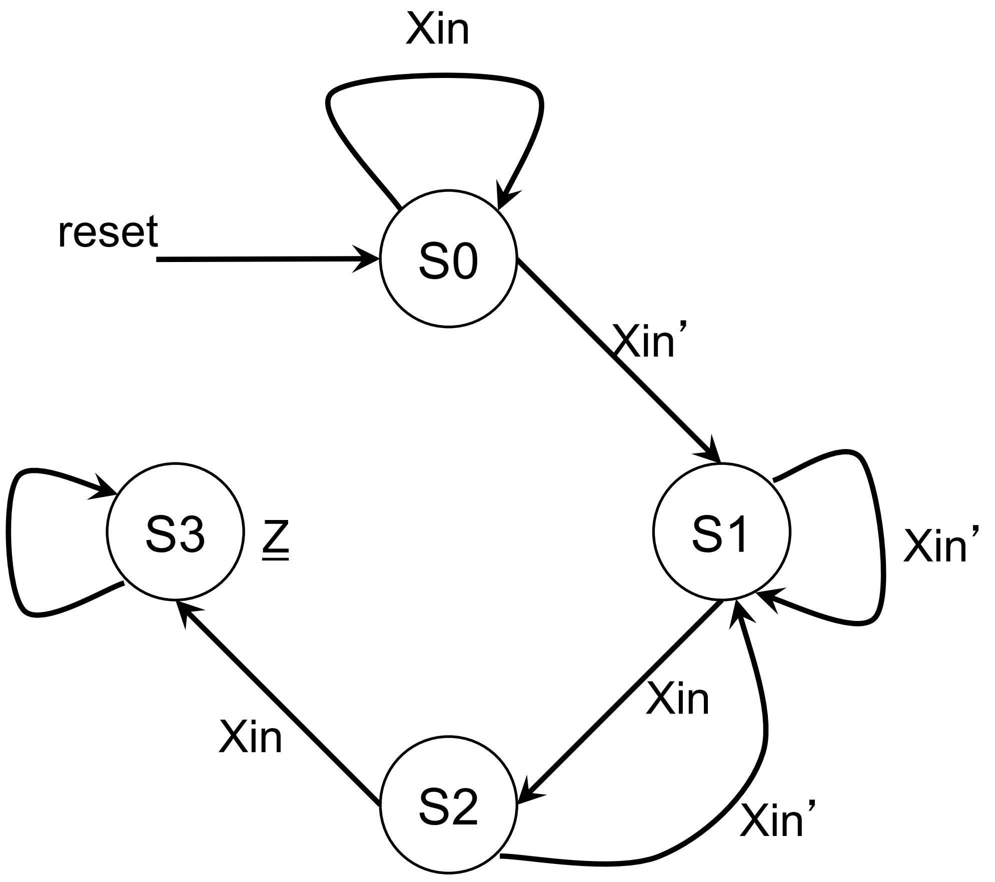
Moore Output



Enumerate the states →

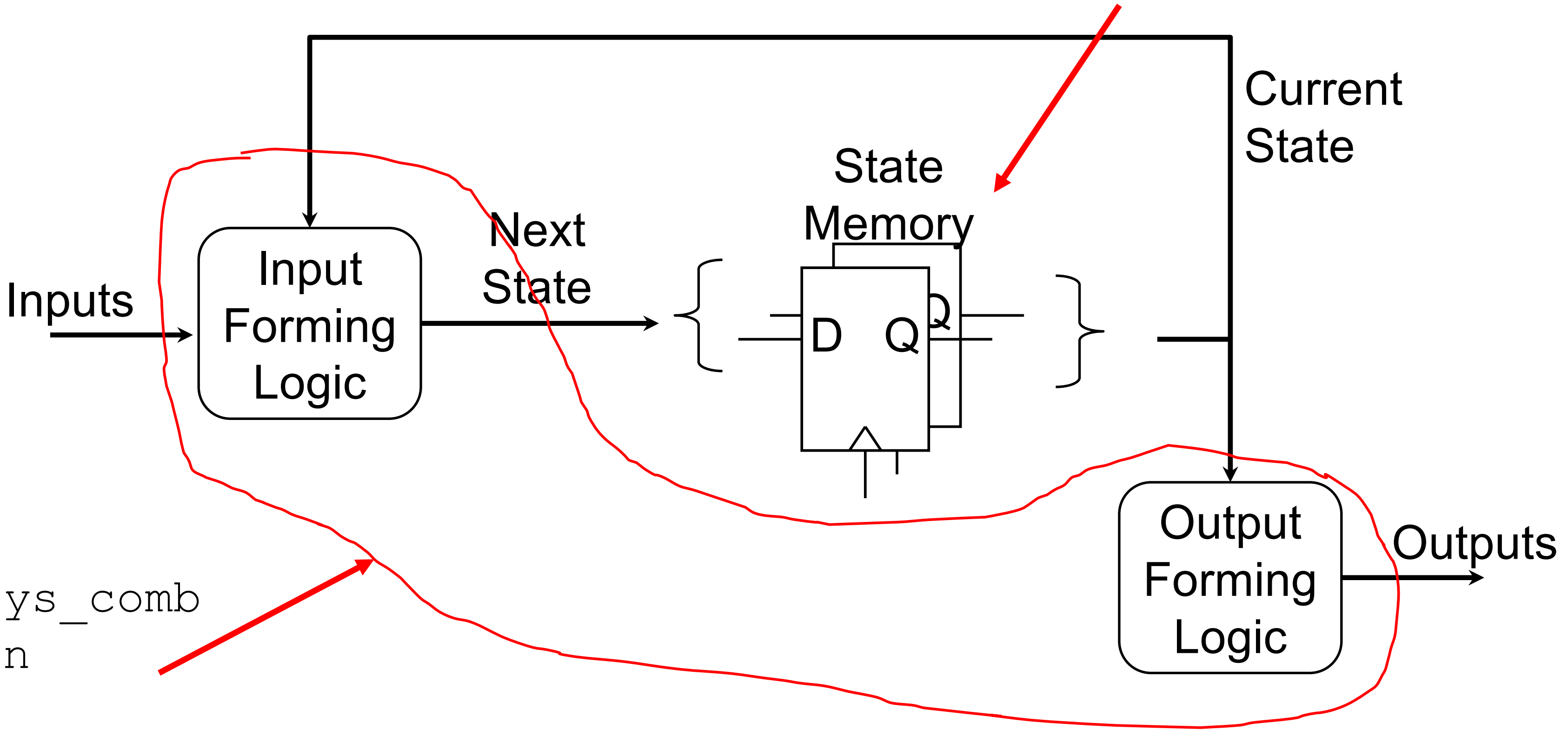
```
typedef enum {s0, s1, s2, s3} StateType;
StateType ns, cs;

always_comb
     begin
        ns = cs;        // default
        Z = 0;          // default

        if (reset)
             ns = s0;
        else
             case (cs)
                s0: if (!Xin)
                       ns = s1;
                s1: if (Xin)
                       ns = s2;
                s2: if (Xin)
                       ns = s3;
                    else
                       ns = s1;
                s3: Z = 1'b1;     // Moore
             endcase
        end

always_ff @(posedge clk)
     cs <= ns;
```

A Sequence Detector



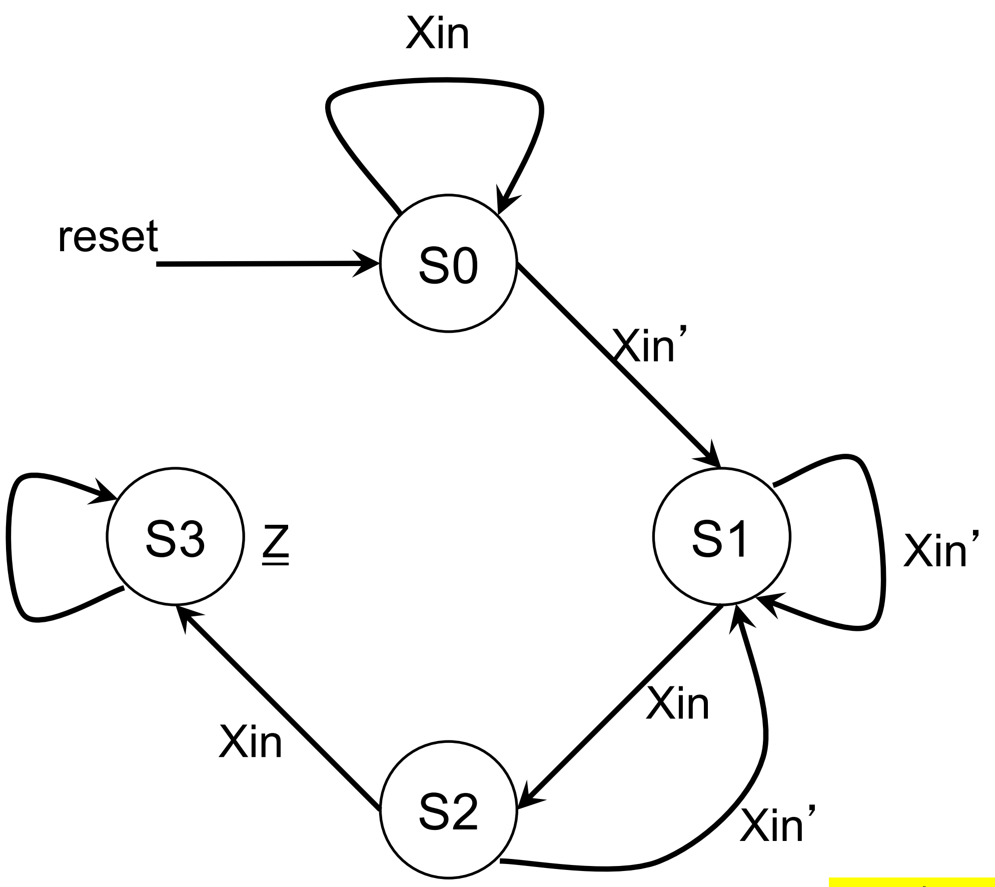
Moore Output

```
always_ff @(posedge clk)
    cs <= ns;
```



```
always_comb
begin

end
```

A Sequence Detector

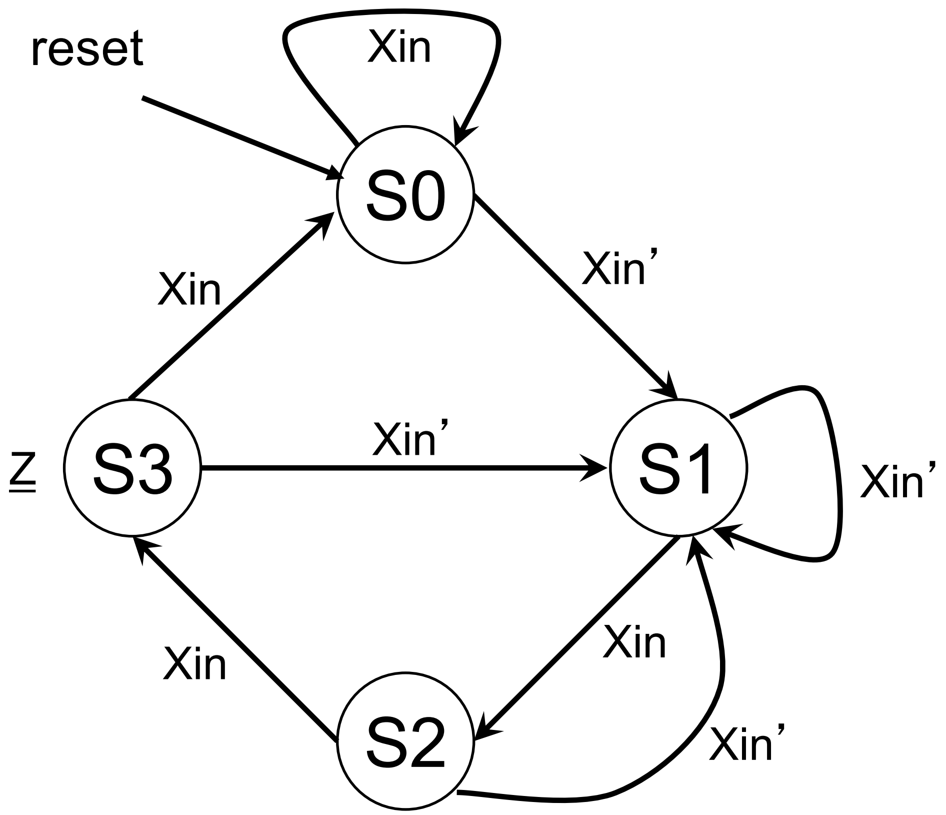


```
always_comb
     begin
        ns = cs;
        ~~Z = 0;~~

        if (reset)
             ns = s0;
        else
             case (cs)
                s0: if (!Xin)
                       ns = s1;
                s1: if (Xin)
                       ns = s2;
                s2: if (Xin)
                       ns = s3;
                    else
                       ns = s1;
                ~~s3: Z = 1’b1;~~
             endcase
        end
```

Use dataflow for Moore output →

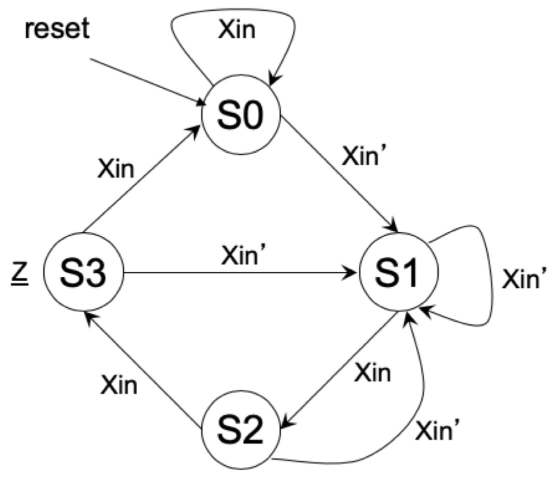
```
Assign Z=(cs == S3) ? 1’b1 : 1’b0;

always_ff @(posedge clk)
     cs <= ns;
```

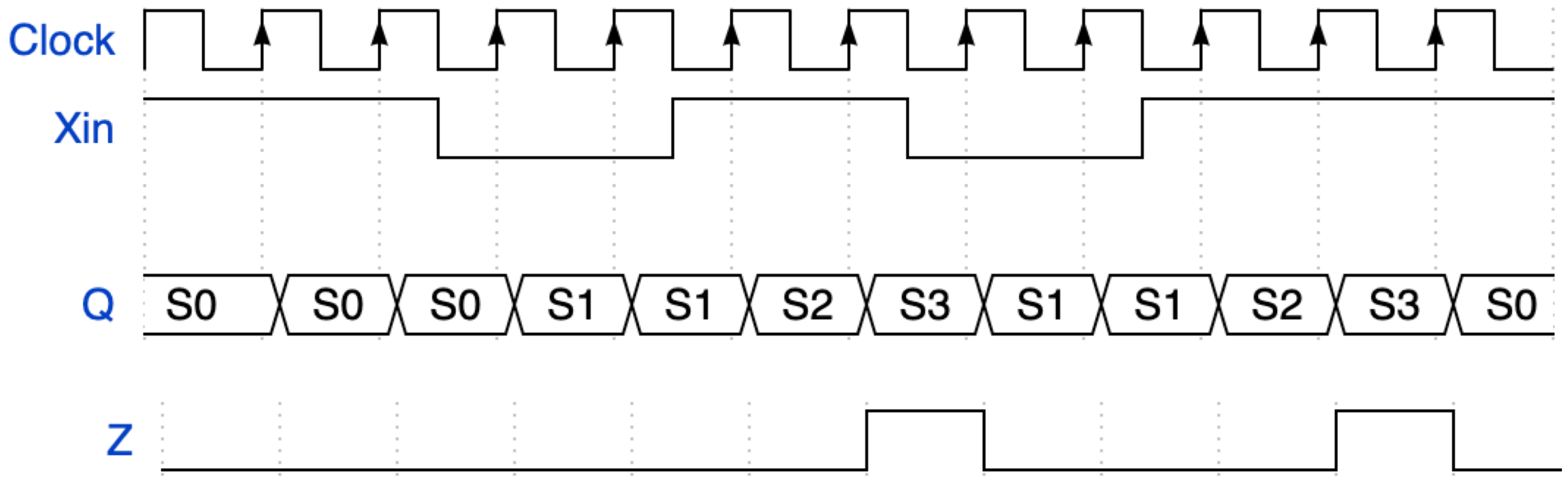
A Better Sequence Detector



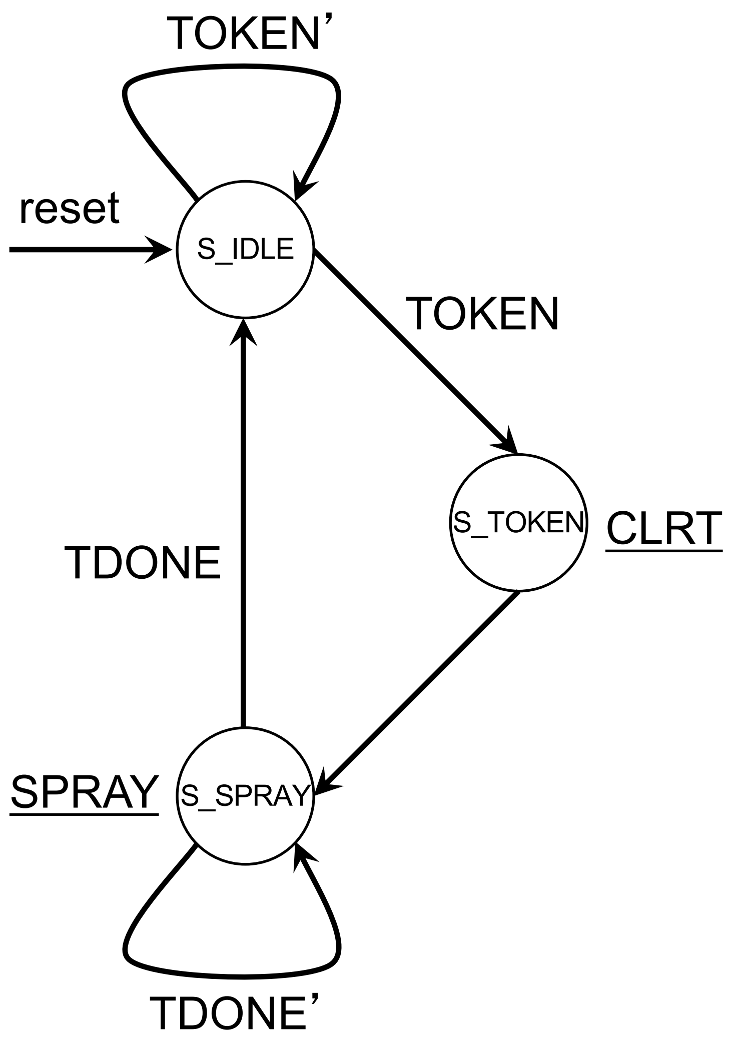
```
typedef enum {S0, S1, S2, S3} StateType;
StateType ns, cs;

always_comb
     begin
        ns = cs;
        Z = 0;

     if (reset)
        ns = S0;
     else
        case (cs)
            S0: if (!Xin) ns = S1;
            S1: if (Xin) ns = S2;
            S2: if (Xin) ns = S3;
                   else ns = S1;
            S3: begin
                   Z = 1'b1;
                   if (!Xin) ns = S1;
                       else ns = S0;
                end
        endcase
     end

always_ff @(posedge clk)
        cs <= ns;
```

State Transition and Output



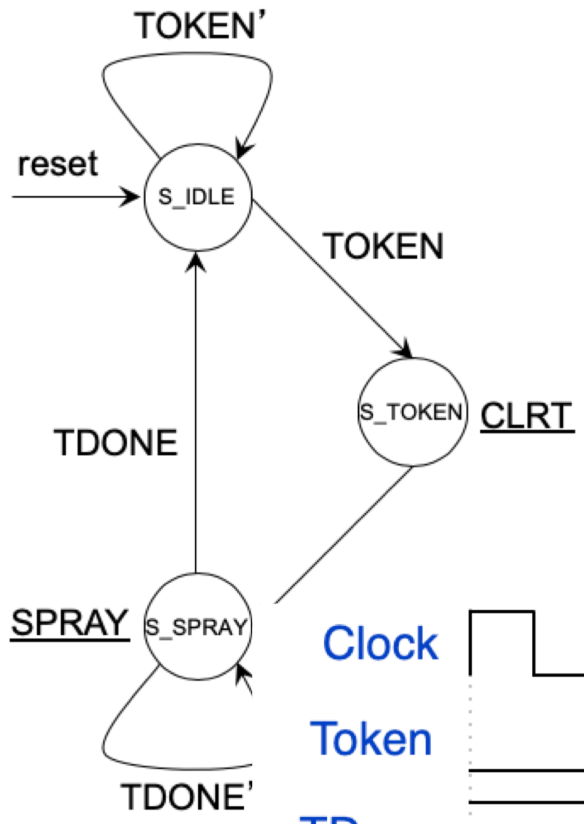


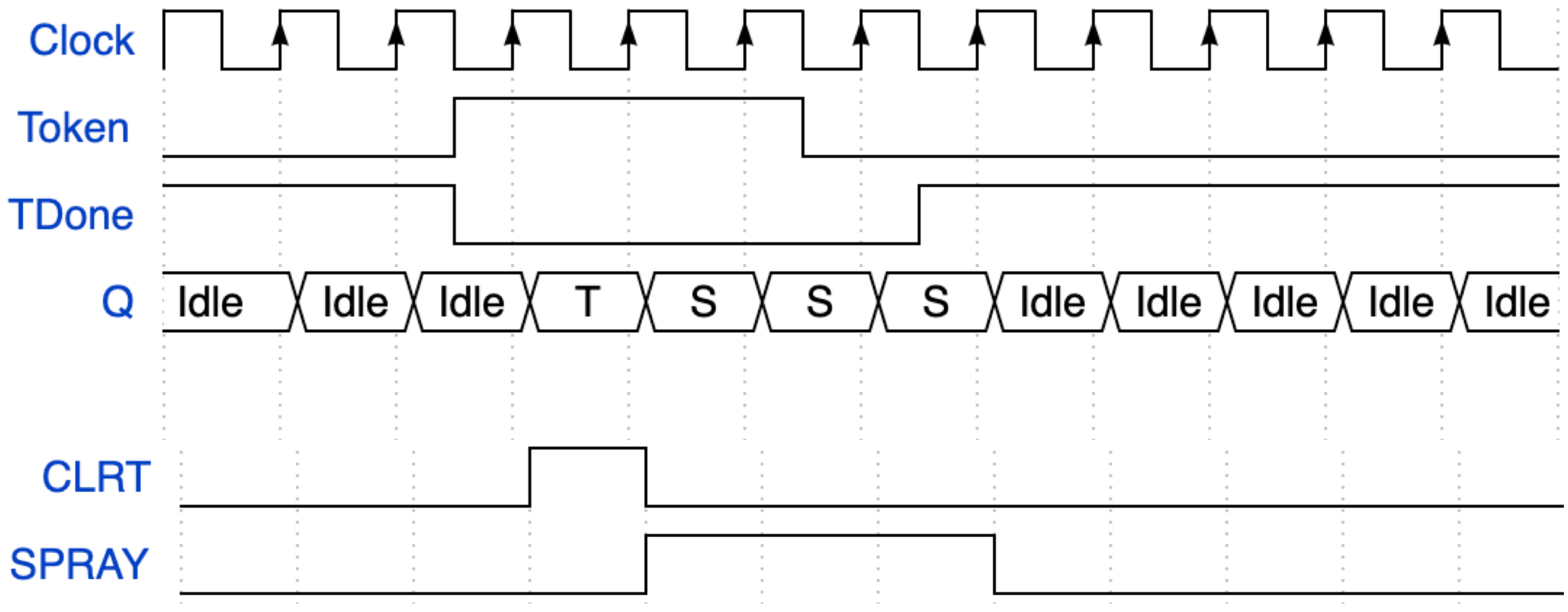
A Car Wash FSM



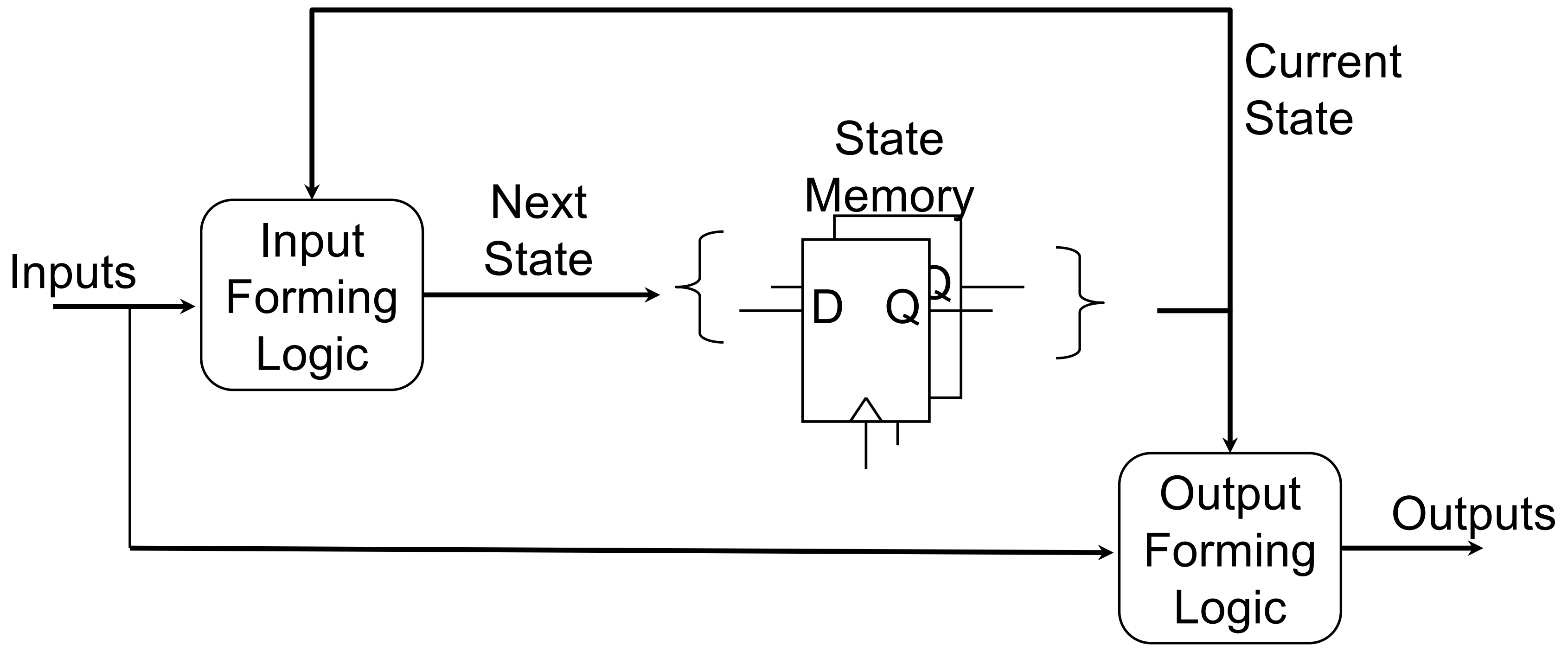
```
typedef enum {sIdle, sToken, sSpray} StateType;
StateType ns, cs;

always_comb
    begin
      ns = cs;
      clrt = 0;
      spray = 0;

    if (reset)
      ns = sIdle;
    else
      case (cs)
          sIdle: if (token) ns = sToken;
          sToken: begin
                clrt = 1;
                ns = sSpray;
            end
          sSpray: begin
                spray = 1;
                if (tdone) ns = sIdle;
            end
          default: ns = sIdle;
      endcase
    end

always_ff @(posedge clk)
      cs <= ns;
```

State Transition and Output

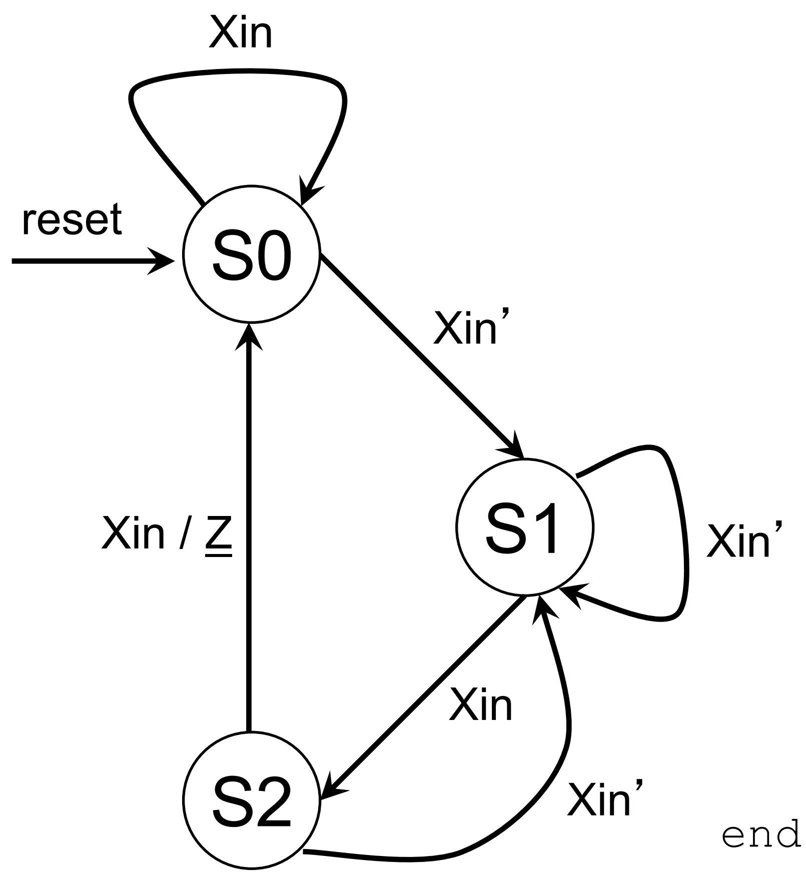




Mealy Output

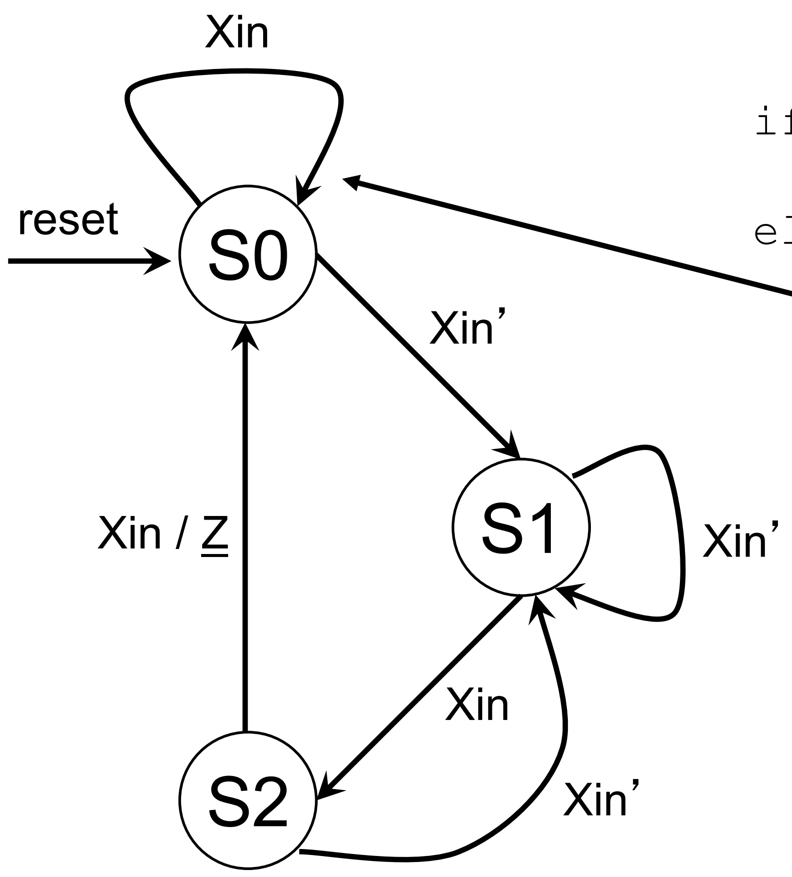


Sequence Detector with Mealy Output



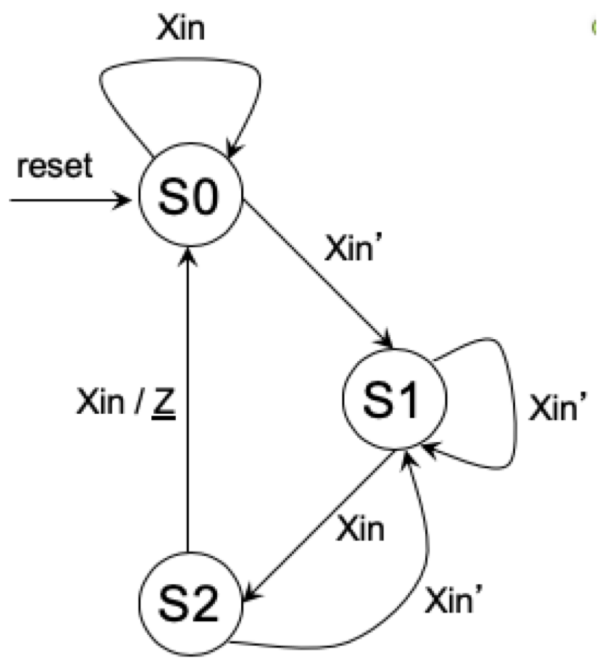
```
typedef enum {s0, s1, s2} stateType
stateType ns, cs;

always_comb
begin
      ns = cs;
      Z = 0;
    if (reset)
      ns = s0;
    else
      case (state)
              s0: if (!Xin) ns = s1;
              s1: if (Xin) ns = s2;
              s2: if (Xin)
                   begin
                     ns = s0;
                     Z = 1; // Mealy output
                   end
                  else ns = s1;
      endcase
 end

always_ff @(posedge clk)
      cs <= ns;
```

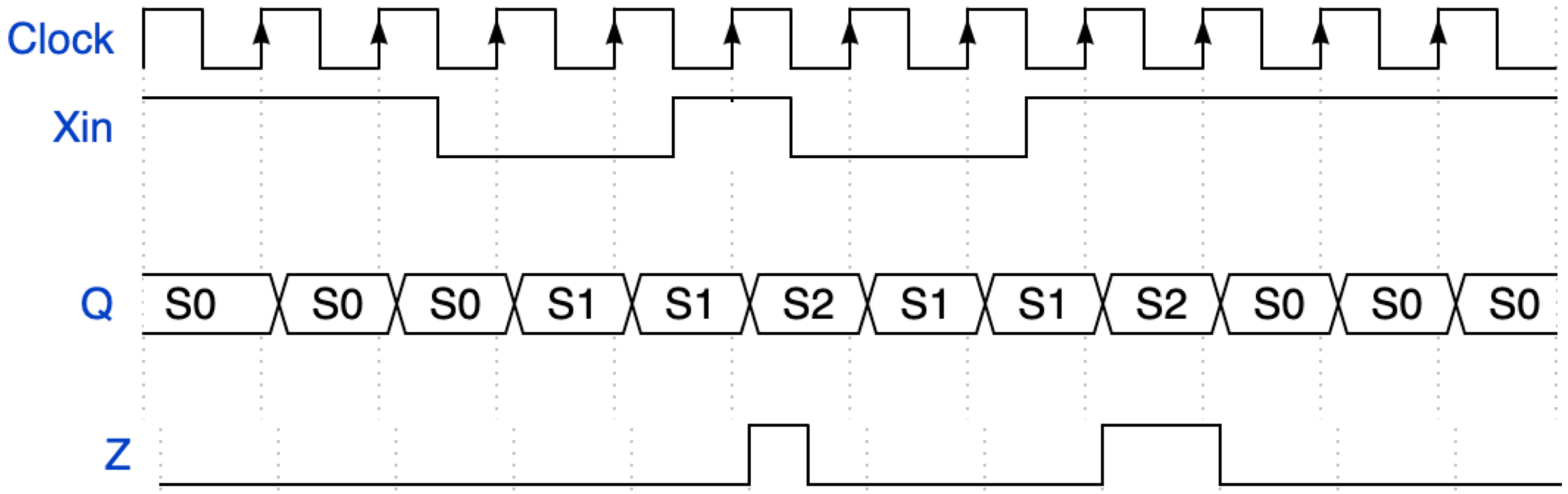
Defensive Coding Style



```
typedef enum {s0, s1, s2, ERR='X} stateType
stateType ns, cs;

always_comb
     begin
       ns = ERR;
       Z = 0;
     if (reset)
       ns = s0;
     else
       case (state)
               s0: if (!Xin) ns = s1;
                      else ns = s0;
               s1: if (Xin) ns = s2;
                      else ns = s1;
               s2: if (Xin)
                    begin
                      ns = s0;
                      Z = 1; // Mealy output
                    end
                   else ns = s1;
       endcase
     end

always_ff @(posedge clk)
       cs <= ns;
```

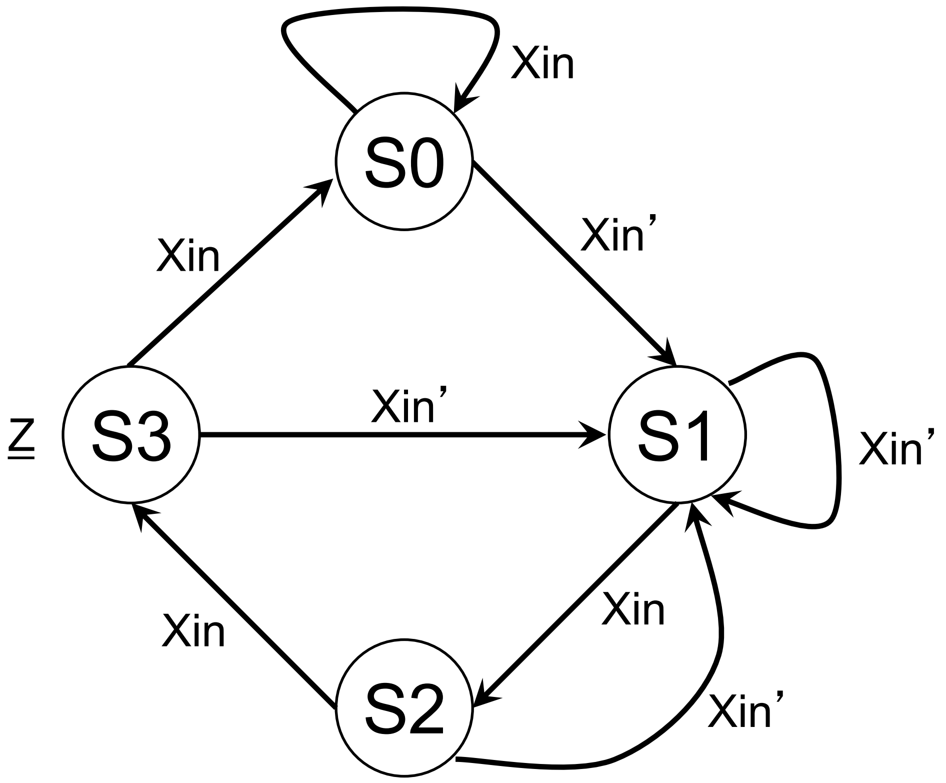
State Transition and Output





Input Forming Logic

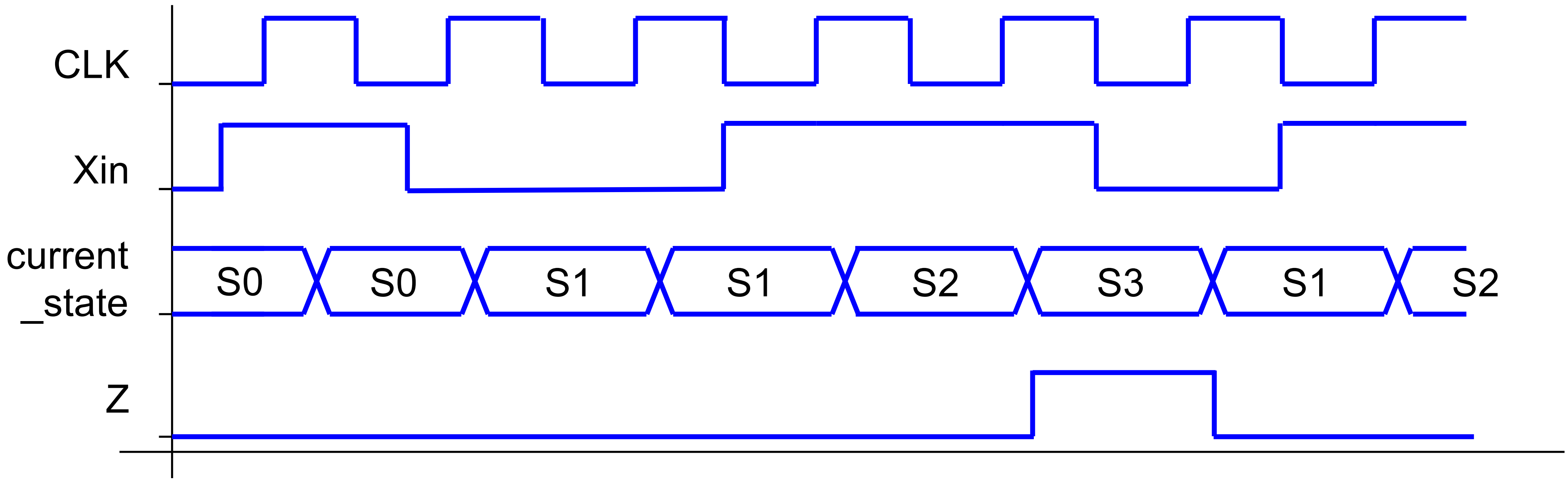
- *Both Combinational and Sequential Logic*



```
always_ff @(posedge clk)
  case (current_state)
    S0: if (Xin == 1'b0) current_state <= S1
    S1: if (Xin == 1'b1) current_state <= S2
    S2: if (Xin == 1'b1) current_state <= S3
          else current_state <= S1;
    S3: if (Xin == 1'b0) current_state <= S1
          else current_state <= S0;
  endcase
```

```
module sequence(
   input logic clk,Xin,
   output logic Z
   );

typedef enum {S0, S1, S2, S3} StateType;
StateType next_state, current_state = S0;
```

```
always_ff @(posedge clk)
   case (current_state)
   S0: if (Xin == 1'b0) current_state <= S1;
   S1: if (Xin == 1'b1) current_state <= S2;
   S2: if (Xin == 1'b1) current_state <= S3;
            else current_state <= S1;
   S3: if (Xin == 1'b0) current_state <= S1;
            else current_state <= S0;
   endcase

assign Z = (current_state == S3);

endmodule
```



Can IFL, State FFs, and OFL be combined?

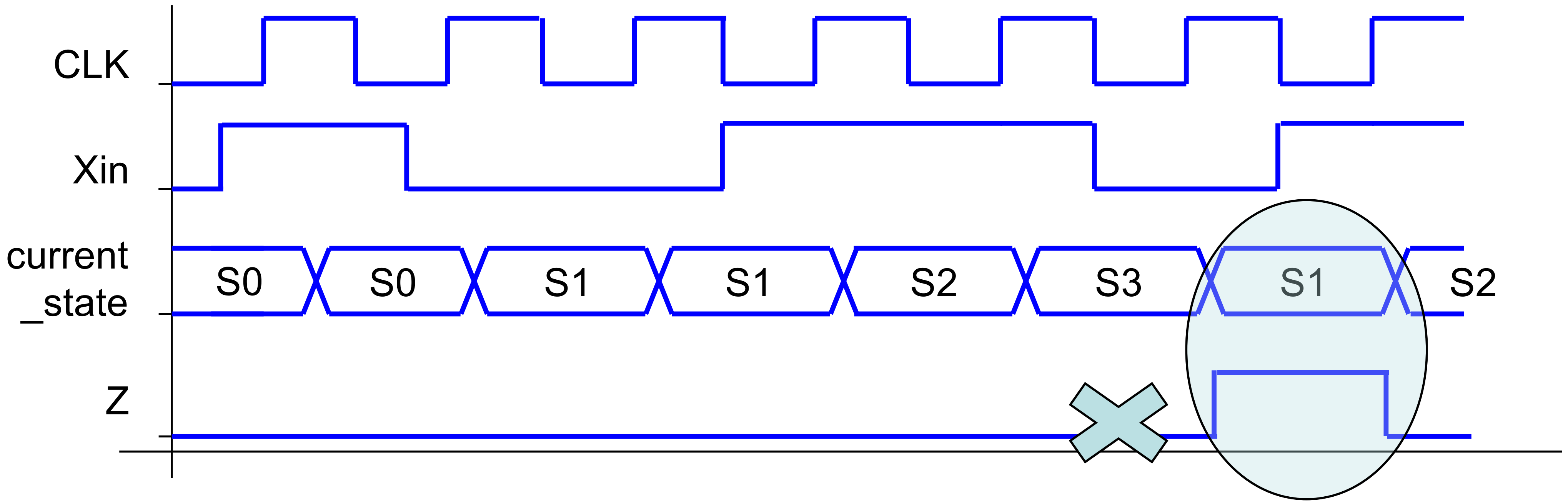
```
always_ff @(posedge clk)
begin
  Z <= 1'b0;
  case (current_state)
    S0:
      if (Xin == 1'b0) current_state <= S1;
    S1:
      if (Xin == 1'b1) current_state <= S2;
    S2:
      if (Xin == 1'b1) current_state  <= S3;
      else current_state <= S1;
    S3: begin
       Z <= 1'b1;
       if (Xin == 1'b0) current_state <= S1;
       else current_state <= S0;
    end
  endcase
end
```

Problem:
A FF will be generated for the 'Z' output (all signals assigned in a clocked always process will have an FF).

An FF on the output Z will cause the signal to be delayed by one clock cycle. The output will no longer be asserted during the S3 state, but during the following cycle.

```
module sequence(
  input logic clk, Xin,
  output logic Z
  );

  typedef enum {S0, S1, S2, S3} StateType;
  StateType next_state, current_state = S0;

  always_ff@(posedge clk) begin
    Z <= 1'b0;
    case (current_state)
```

```
S0: if (Xin == 1'b0) current_state <= S1;
   else current_state <= S0;
S1: if (Xin == 1'b1) current_state <= S2;
       else current_state <= S1;
S2:  if (Xin == 1'b1) current_state <= S3;
   else current_state <= S1;
S3: begin
    Z <= 1'b1;
    if (Xin == 1'b0) current_state <= S1;
    else current_state <= S0;
   end
  endcase
 end
endmodule
```



CLK
Xin
current
_state
S0
S0
S1
S1
S2
S3
S1
S2
Z