ECE 327/627
Digital Hardware Systems

Lecture 5: Finite State Machines

Andrew Boutros

andrew.boutros@uwaterloo.ca

In the Previous Lecture …

- Two design case studies using what we learned about SystemVerilog so far
    - Nvidia Tensor Cores
    - Google’s TPU Systolic Matrix Multiplier
- Trade-offs between the two design styles
    - Latency
    - Throughput
    - Frequency
    - Area
    - Power
    - Input bandwidth
    - Routing wires (i.e., physical design)

What are Finite State Machines (FSMs)?

- To perform any useful computation:
  - Datapath (muscles) – to execute operations
  - Control logic (brain) – what to execute & when?

What are Finite State Machines (FSMs)?

- To perform any useful computation:
  - Datapath (muscles) – to execute operations
  - Control logic (brain) – what to execute & when?

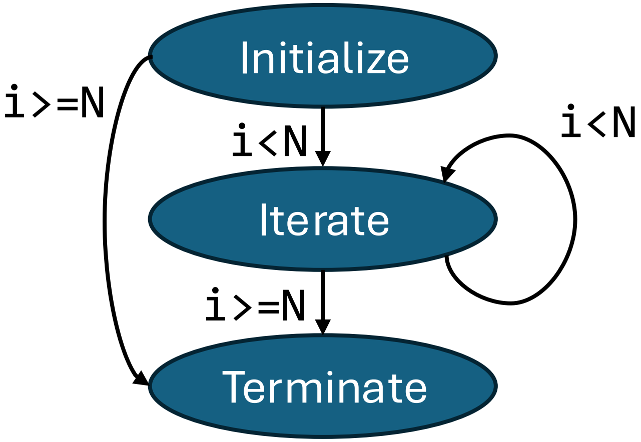
- FSMs offer a systematic way to reason about control logic as:
  - **States** that capture which step of the computation we are currently in
  - **Transitions** that capture the ordering and progression of steps
  - **Actions** that describe the computation in a specific state or transition

What are Finite State Machines (FSMs)?

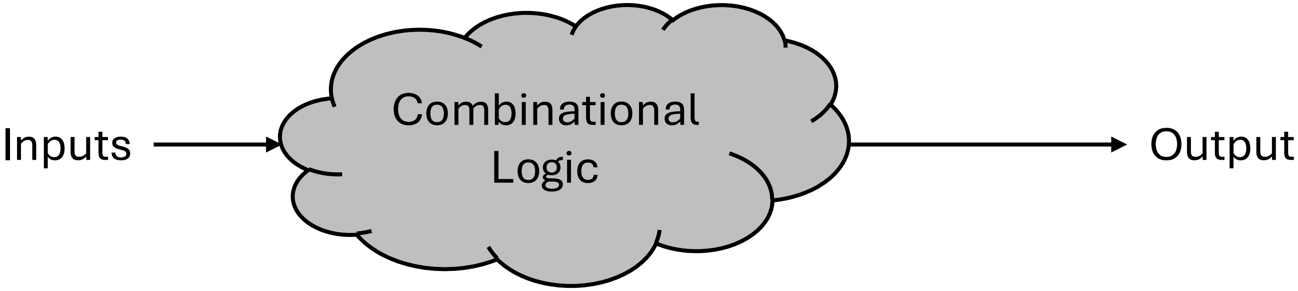
- To perform any useful computation:
  - Datapath (muscles) – to execute operations
  - Control logic (brain) – what to execute & when?

- FSMs offer a systematic way to reason about control logic as:
  - **States** that capture which step of the computation we are currently in
  - **Transitions** that capture the ordering and progression of steps
  - **Actions** that describe the computation in a specific state or transition

- Trivial example: A for loop is an FSM
  - Example: `for(int i=0; i<N; i++)`
  - Initialize, Iterate, and Terminate are **states**
  - Loop body and counter updates are **actions**
  - Jump or branch instructions are **transitions**

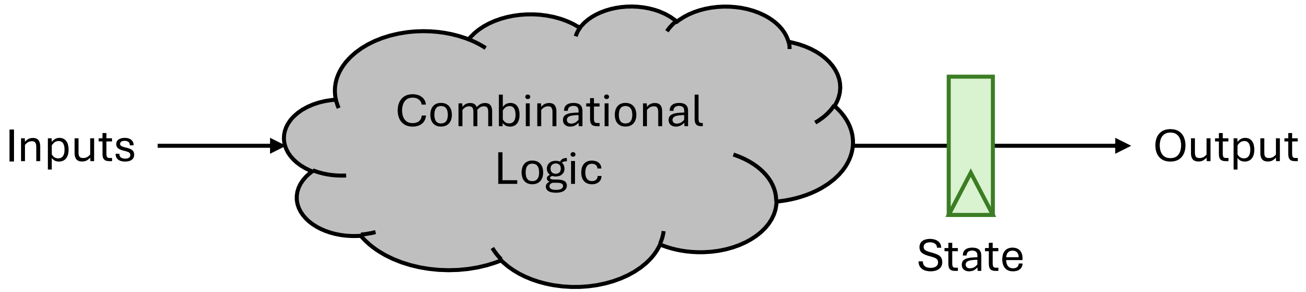


What does that mean in hardware?



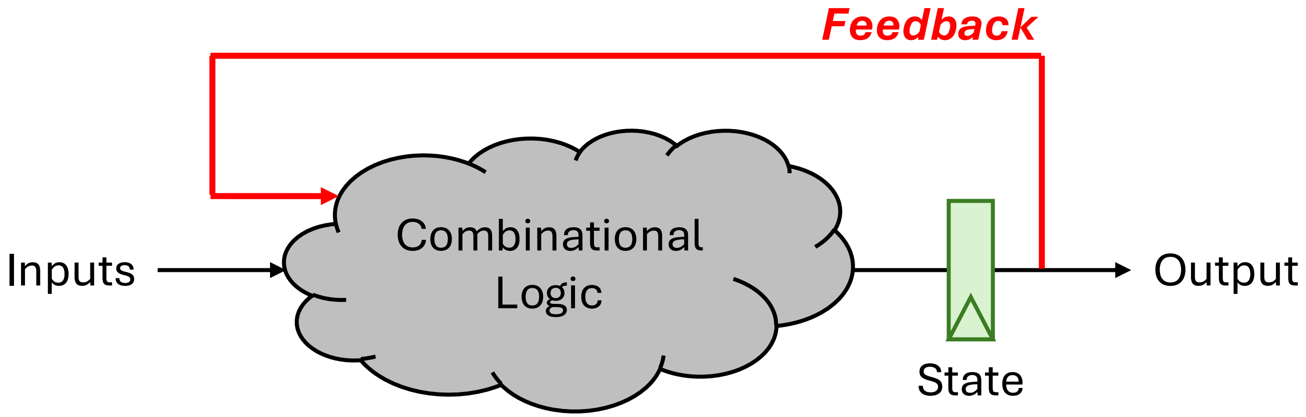
- Combinational logic has no state
  - Consists of logic gates, arithmetic units, and wires

What does that mean in hardware?



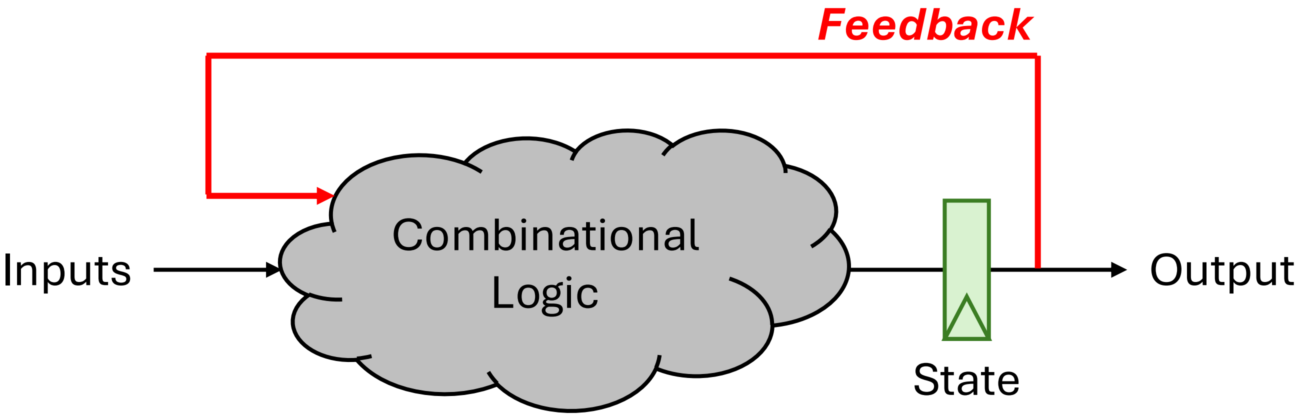
- Combinational logic has no state
  - Consists of logic gates, arithmetic units, and wires
- Sequential logic adds registers or “states”
  - So far, we designed feed-forward circuits → data flows in one direction
  - State is not used to determine the behavior of the computation

What does that mean in hardware?



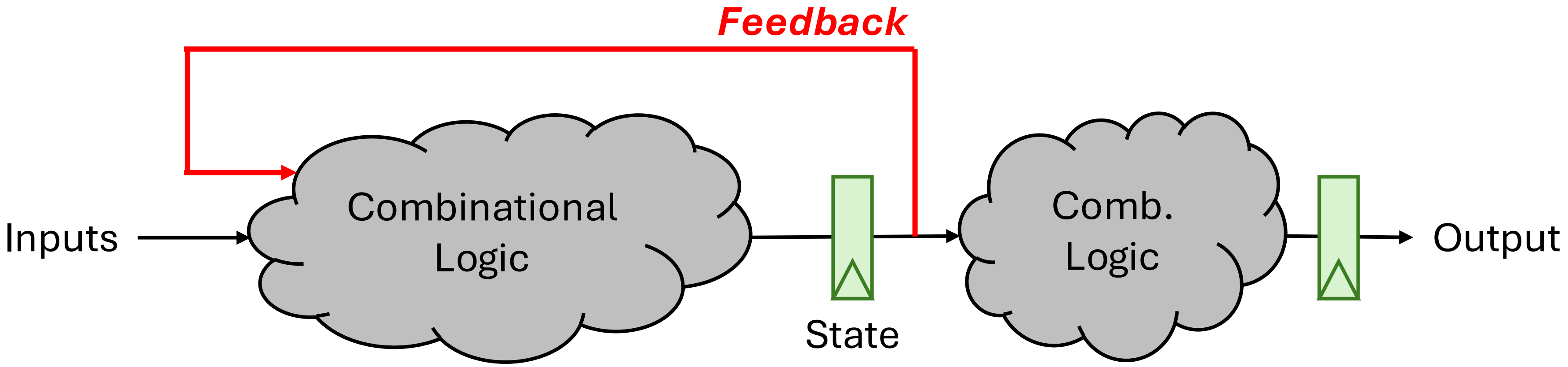
- Combinational logic has no state
  - Consists of logic gates, arithmetic units, and wires
- Sequential logic adds registers or "states"
  - So far, we designed feed-forward circuits → data flows in one direction
  - State is not used to determine the behavior of the computation
- FSMs introduce feedback
  - Current state is used to determine the behavior of next computation

What does that mean in hardware?



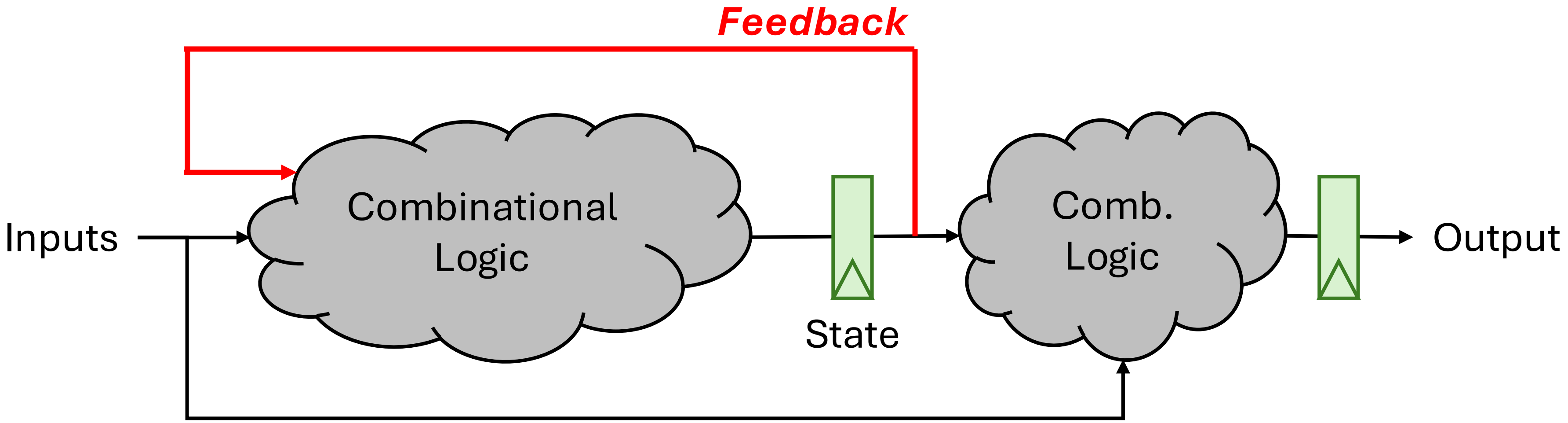
- In some cases (not often), output is the state

What does that mean in hardware?



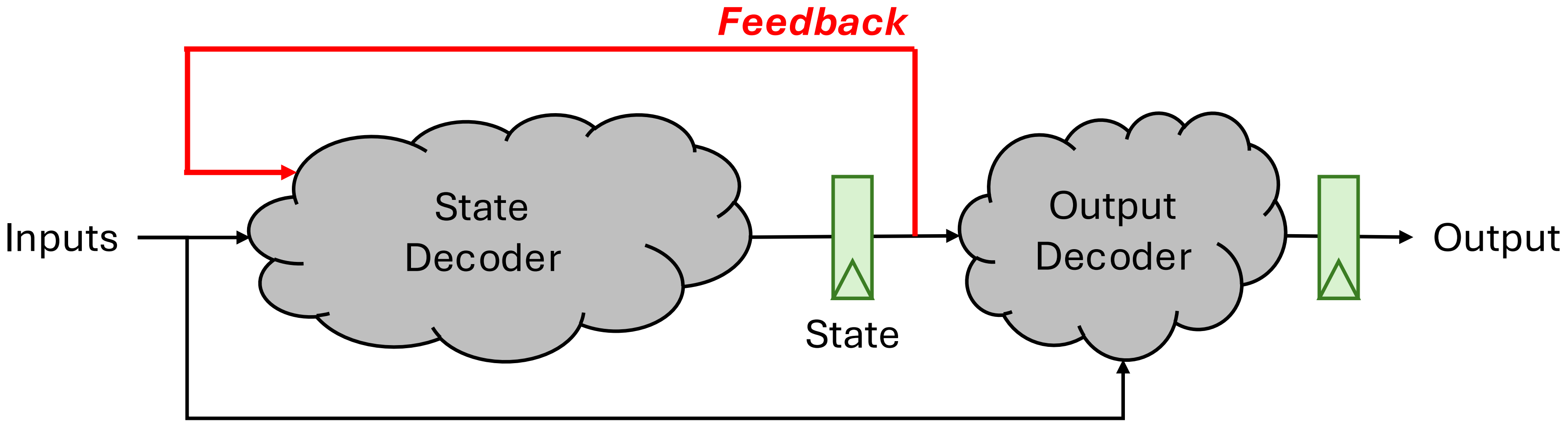
- In some cases (not often), output is the state
- In many cases, decode output from state using some more combinational logic
  - Output lagging one cycle behind state

What does that mean in hardware?



- In some cases (not often), output is the state
- In many cases, decode output from state using some more combinational logic
  - Output lagging one cycle behind state
- Can work around that by using previous state + incoming inputs to decode output
  - State & output are in sync, but possibly slower & bigger area

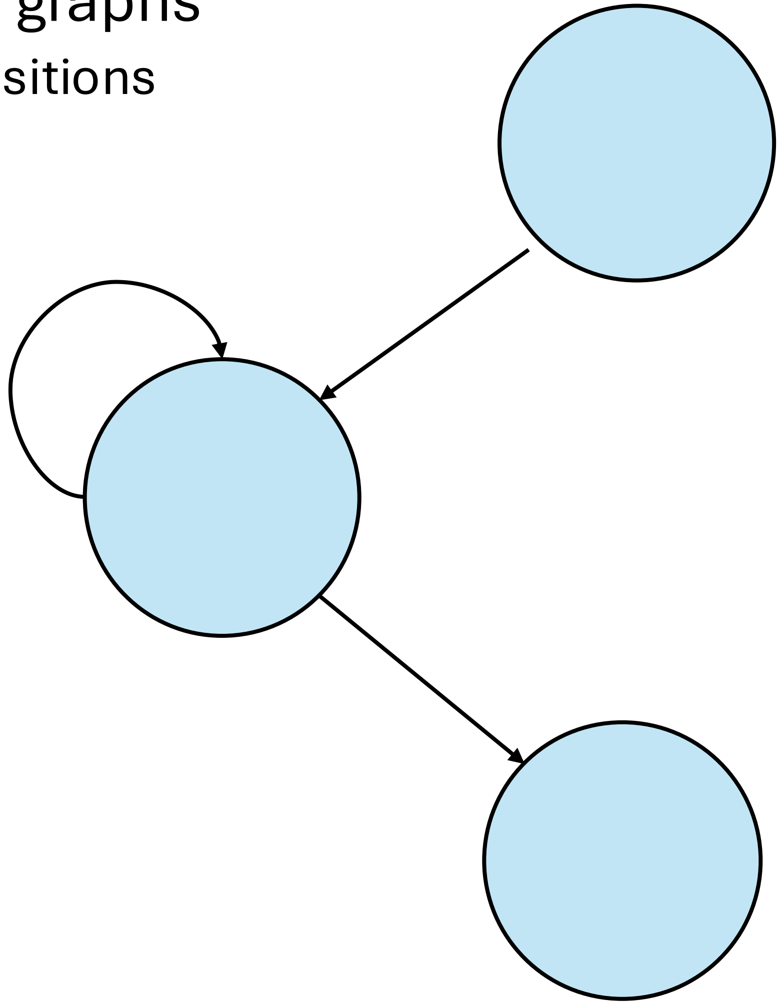
What does that mean in hardware?



- In some cases (not often), output is the state
- In many cases, decode output from state using some more combinational logic
  - Output lagging one cycle behind state
- Can work around that by using previous state + incoming inputs to decode output
  - State & output are in sync, but possibly slower & bigger area

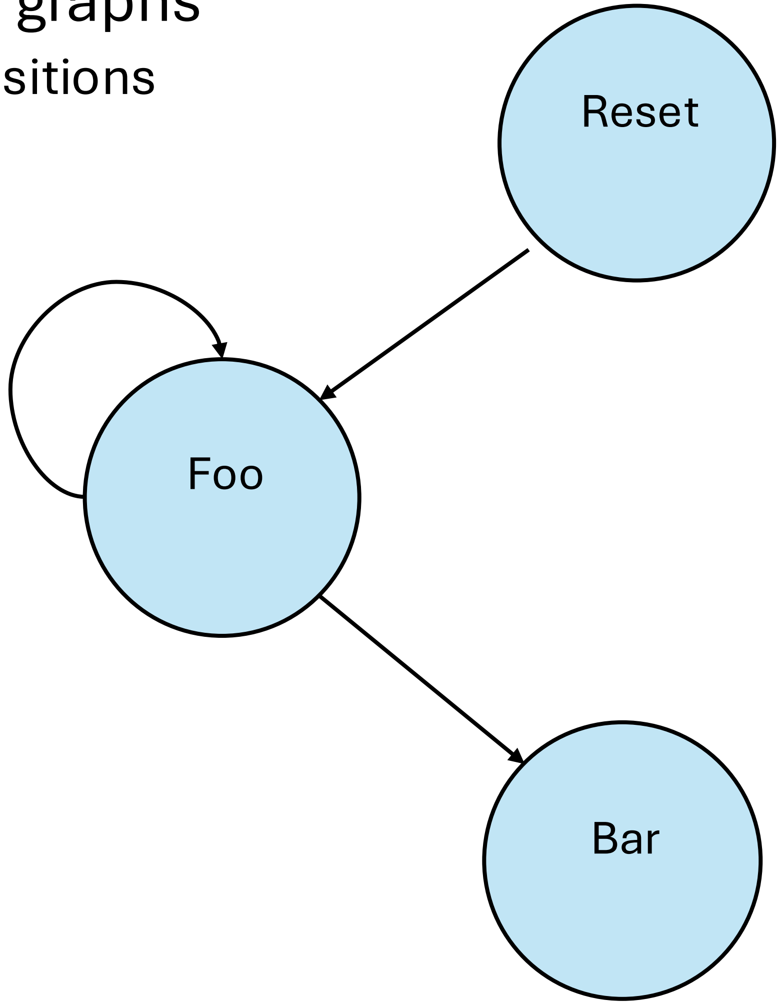
Visual Representation

- FSMs are represented as directed graphs
  - Nodes are states and edges are transitions



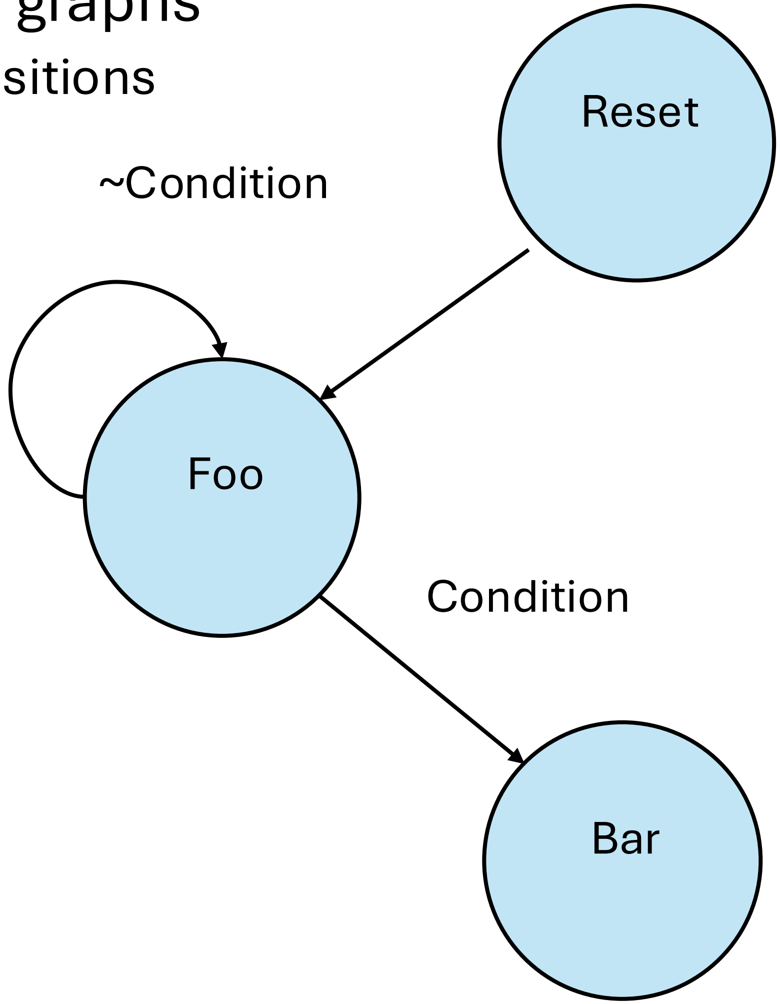
Visual Representation

- FSMs are represented as directed graphs
  - Nodes are states and edges are transitions
- Each state has a name



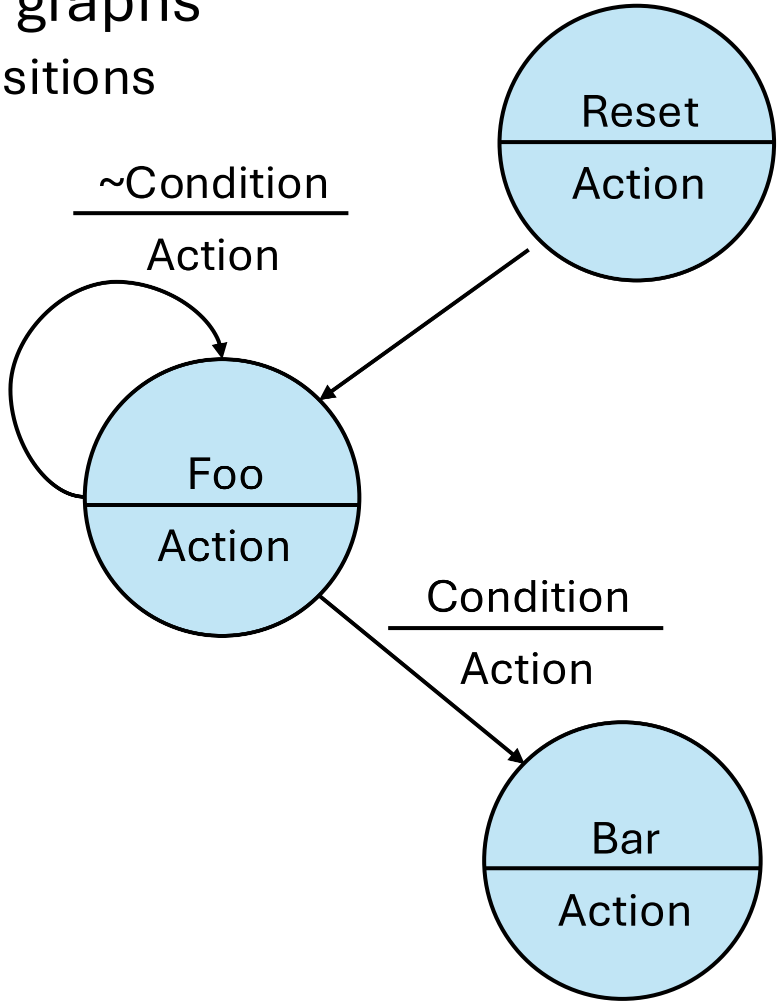
Visual Representation

- FSMs are represented as directed graphs
  - Nodes are states and edges are transitions
- Each state has a name
- Each transition has a condition
  - No condition explicitly specified → this transition always happens



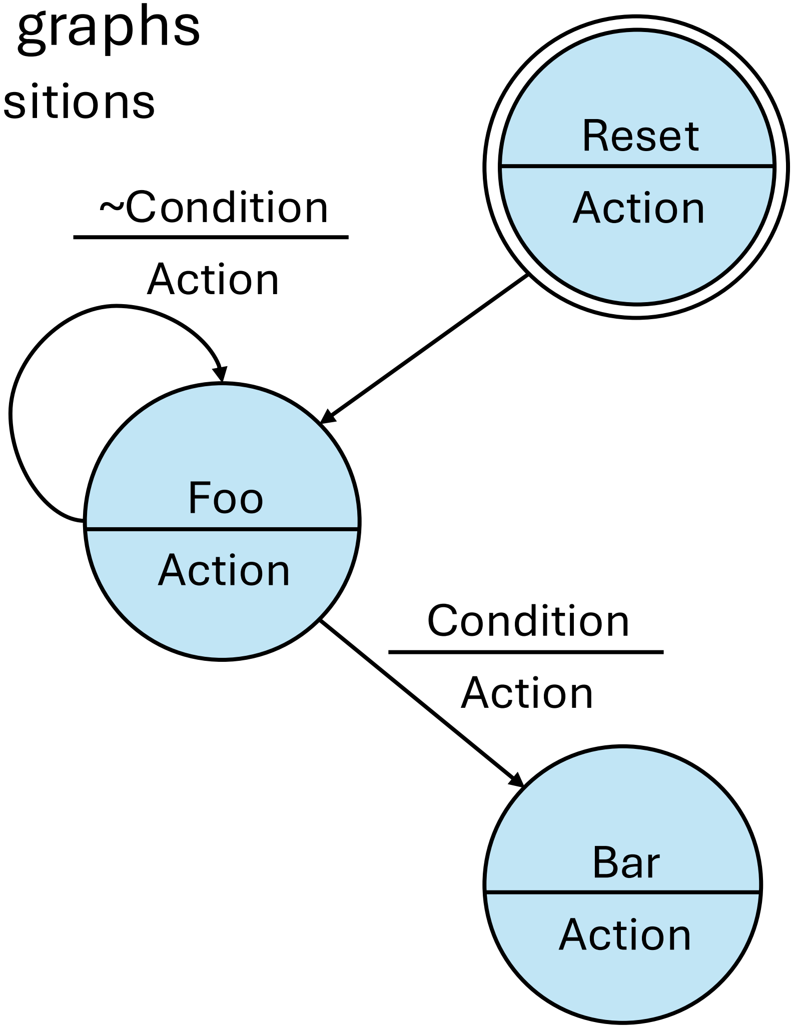
Visual Representation

- FSMs are represented as directed graphs
  - Nodes are states and edges are transitions
- Each state has a name
- Each transition has a condition
  - No condition explicitly specified → this transition always happens
- States and transitions can specify actions that happen in a given state or when transitioning from one state to another



Visual Representation

- FSMs are represented as directed graphs
  - Nodes are states and edges are transitions
- Each state has a name
- Each transition has a condition
  - No condition explicitly specified → this transition always happens
- States and transitions can specify actions that happen in a given state or when transitioning from one state to another
- Reset (wake up) state usually highlighted with double borders
  - If reset signal is asserted, return to this state



Recipe for Designing FSMs

1. Determine the number of states needed for a given problem and give them (representative) names
2. Determine the inputs to and outputs of the FSM
3. Define state transitions and their conditions
4. Define what actions happen in each state or transition
5. Verify FSM logic is correct:
    - Does it get stuck in a certain state? Is that a bug or by design?
    - Are all transition conditions properly defined?
    - Are all states reachable (i.e., no dead states)?

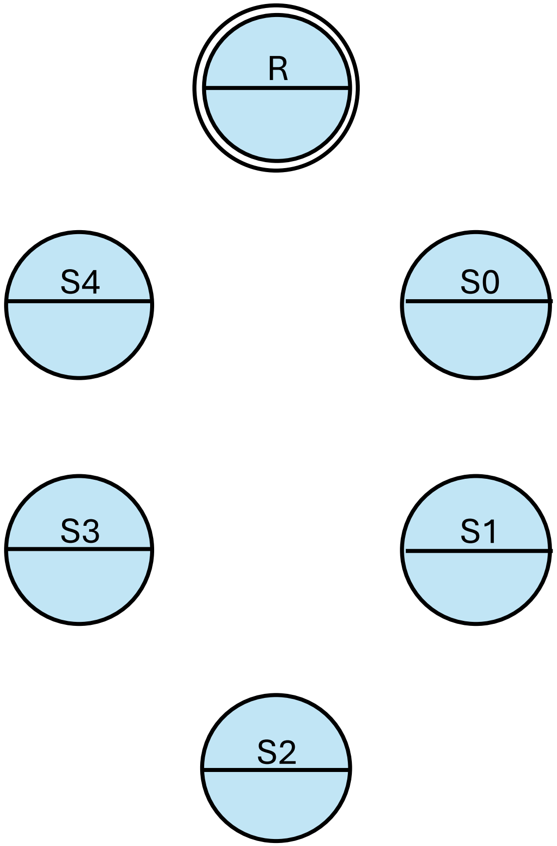
Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

1. Determine the number of states needed for a given problem and give them (representative) names

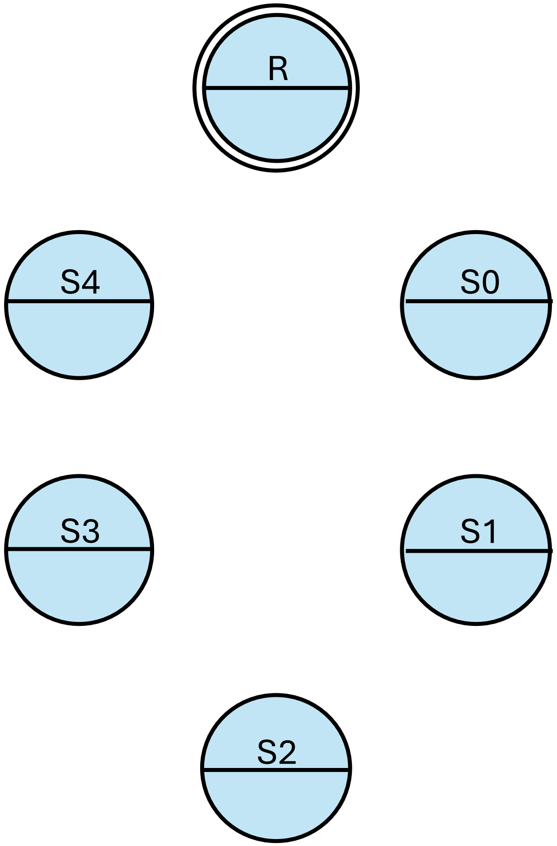


Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

2. Determine the inputs to and outputs of the FSM

No inputs & Output is 3-bit counter value (cntr)

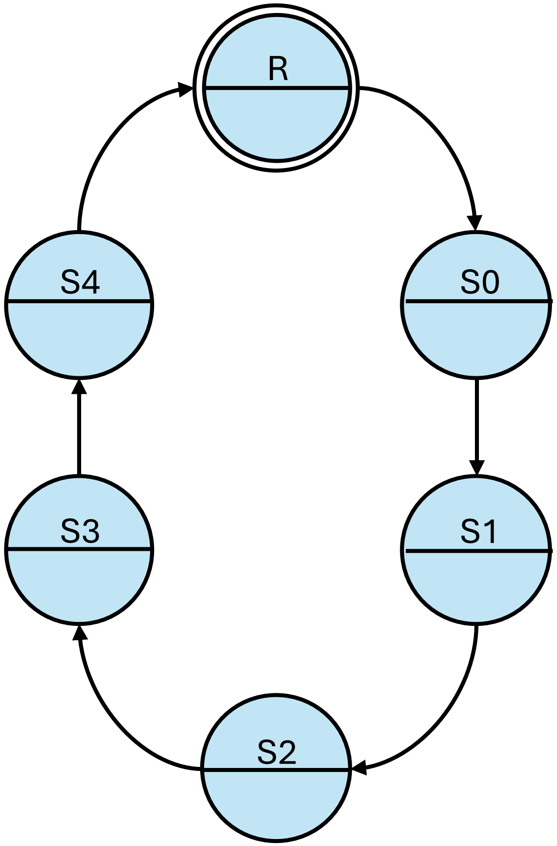


Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

3. Define state transitions and their conditions

Transitions have no conditions and → FSM will transition from one state to the next each clock cycle in an infinite loop

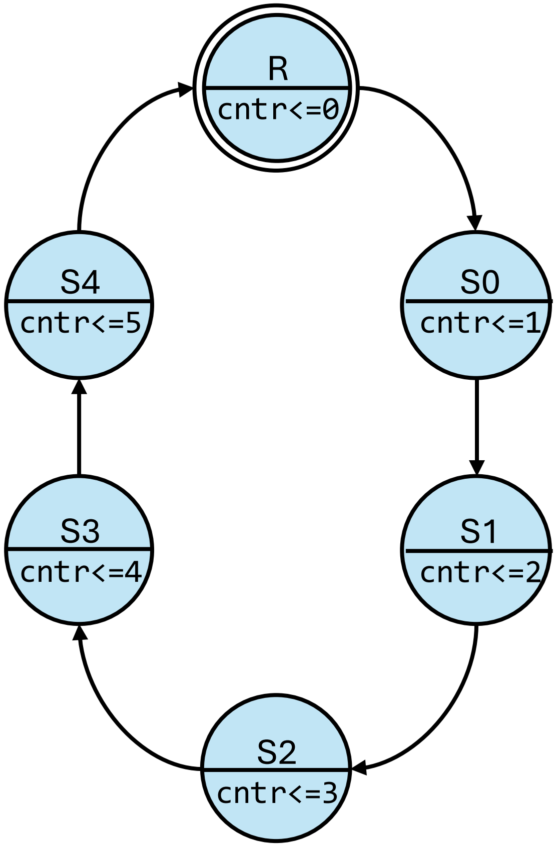


Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

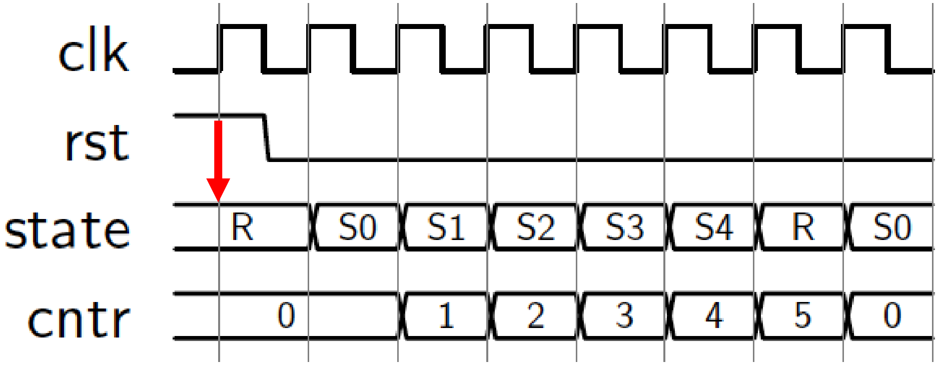
4. Define what actions happen in each state or transition

Assign a new constant value to the `cntr` *output*

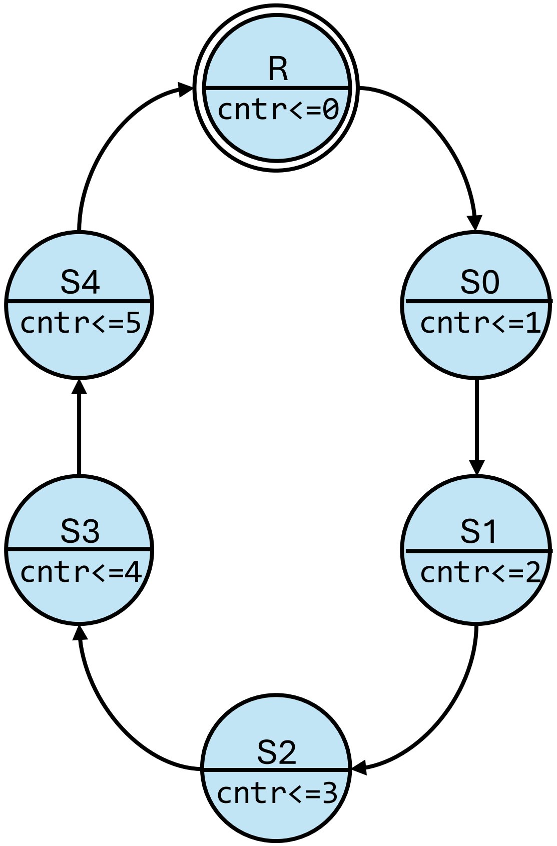


Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever

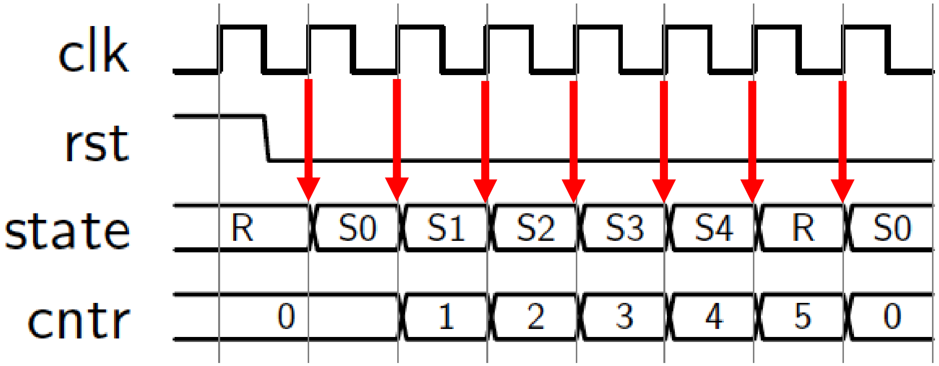


When `rst` is high, stay in reset state R



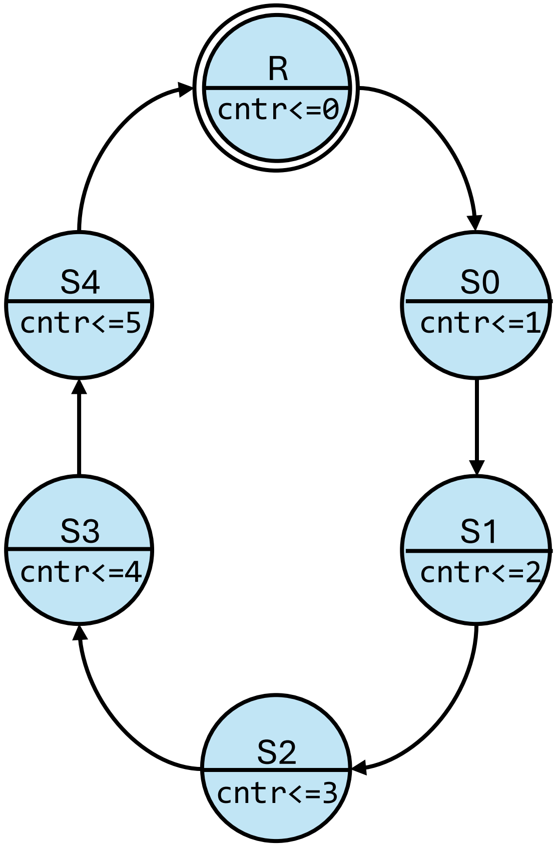
Simple Example: Free-Running Counter

Design a counter that counts from zero to five, then restarts at zero and keeps running forever



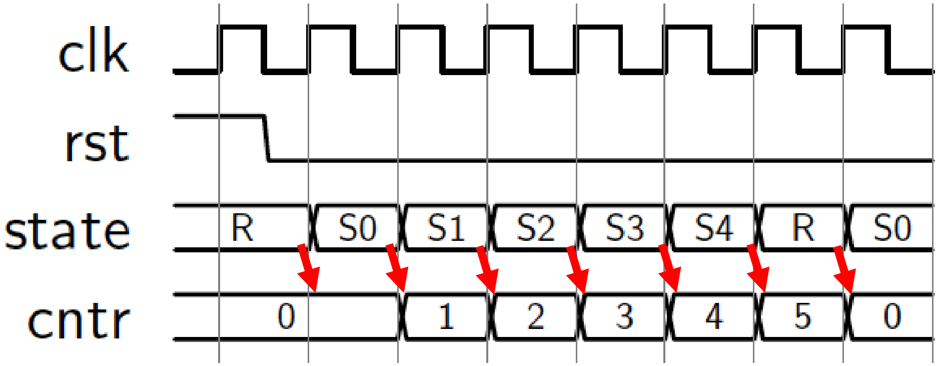
When `rst` *is high, stay in reset state R*

Every rising clock edge, a state transition happens



Simple Example: Free-Running Counter

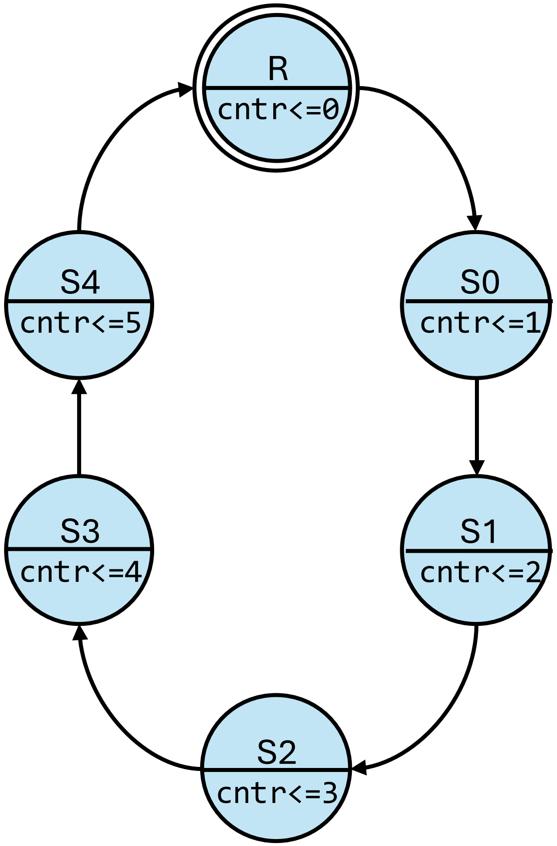
Design a counter that counts from zero to five, then restarts at zero and keeps running forever



When `rst` is high, stay in reset state R

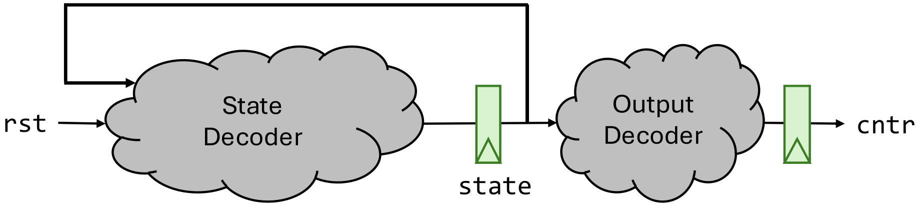
Every rising clock edge, a state transition happens

Since output is registered, `cntr` value is updated at the next rising clock edge



Simple Example: Free-Running Counter

```
module cntr5 (
     input  clk,
     input  rst,
     output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;
```

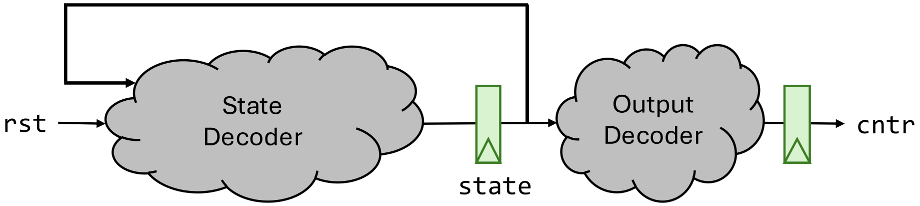


```
endmodule
```

Simple Example: Free-Running Counter

```
module cntr5 (
     input  clk,
     input  rst,
     output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;
```

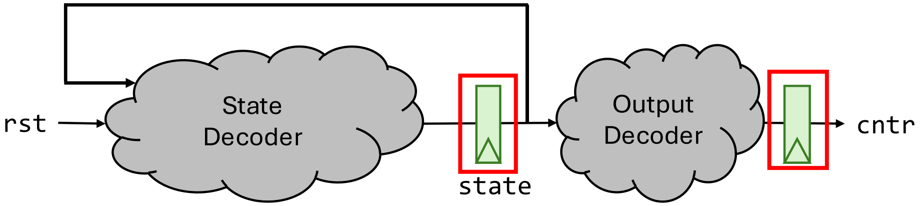
Usually optimized by synthesis engine as one-hot encoding



```
endmodule
```

Simple Example: Free-Running Counter

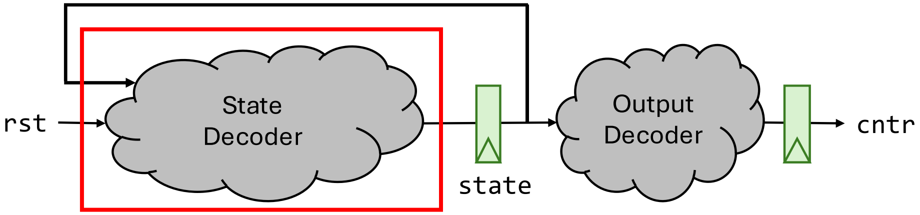
```
module cntr5 (
     input  clk,
     input  rst,
     output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
endmodule
```

Simple Example: Free-Running Counter

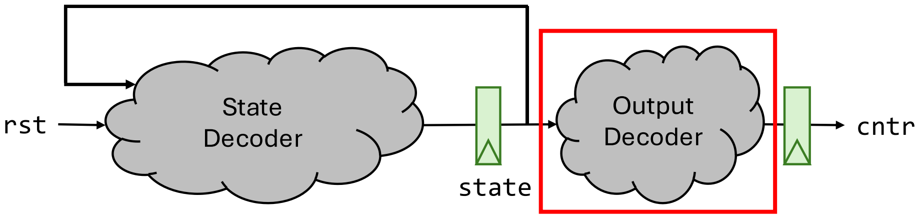
```
module cntr5 (
      input  clk,
      input  rst,
      output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = S0;
    S0: next_state = S1;
    S1: next_state = S2;
    S2: next_state = S3;
    S3: next_state = S4;
    S4: next_state = R;
    default: next_state = R;
  endcase
end






endmodule
```

Simple Example: Free-Running Counter

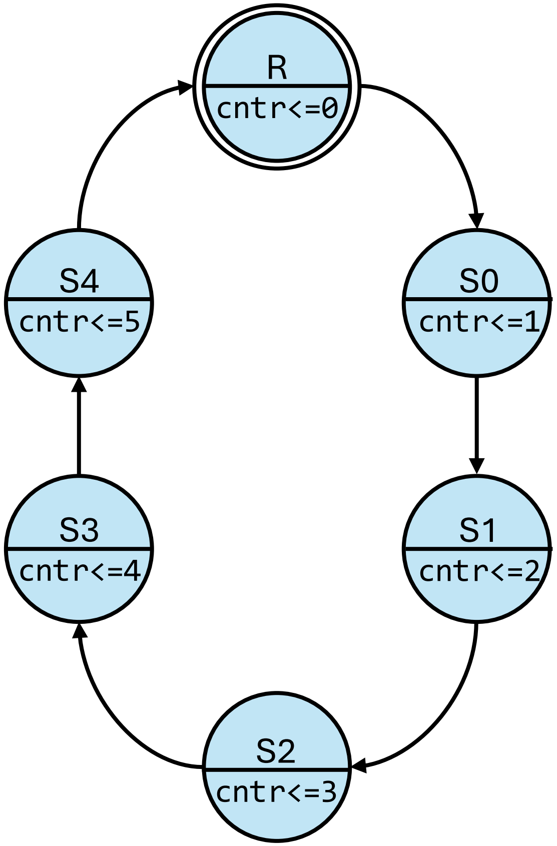
```
module cntr5 (
     input  clk,
     input  rst,
     output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = S0;
    S0: next_state = S1;
    S1: next_state = S2;
    S2: next_state = S3;
    S3: next_state = S4;
    S4: next_state = R;
    default: next_state = R;
  endcase
end

always_comb begin: output_decoder
  case (state)
    R : cntr_val = 0;
    S0: cntr_val = 1;
    S1: cntr_val = 2;
    S2: cntr_val = 3;
    S3: cntr_val = 4;
    S4: cntr_val = 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

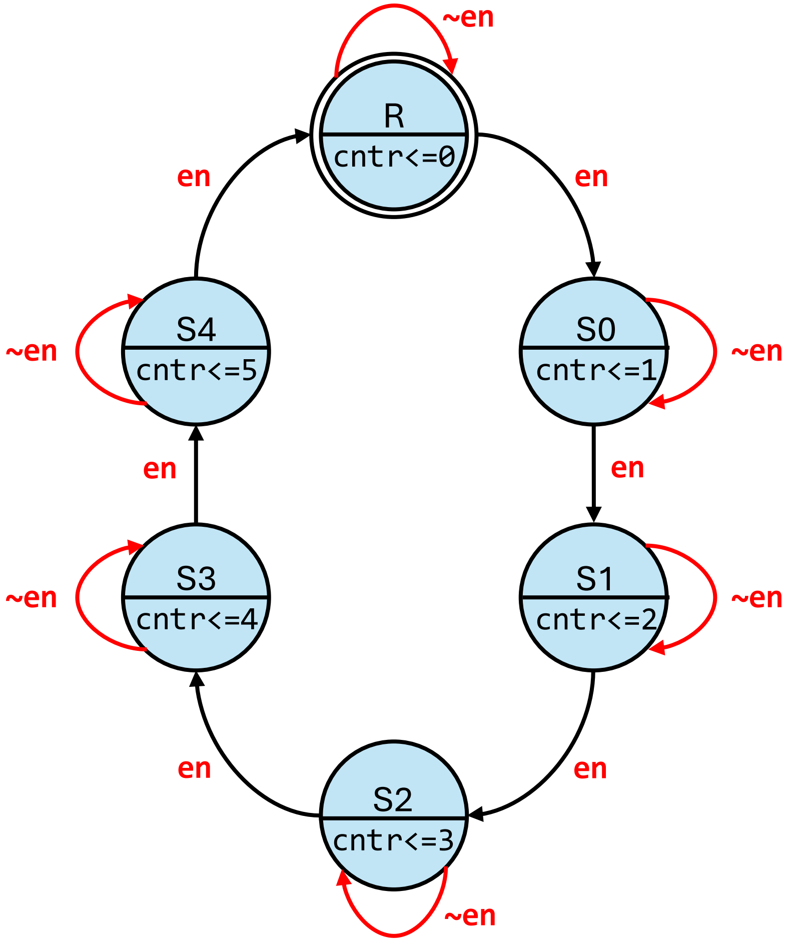
Simple Example: Externally Enabled Counter

Modify the free-running counter we designed to have an enable signal. The counter advances only when enable is asserted.



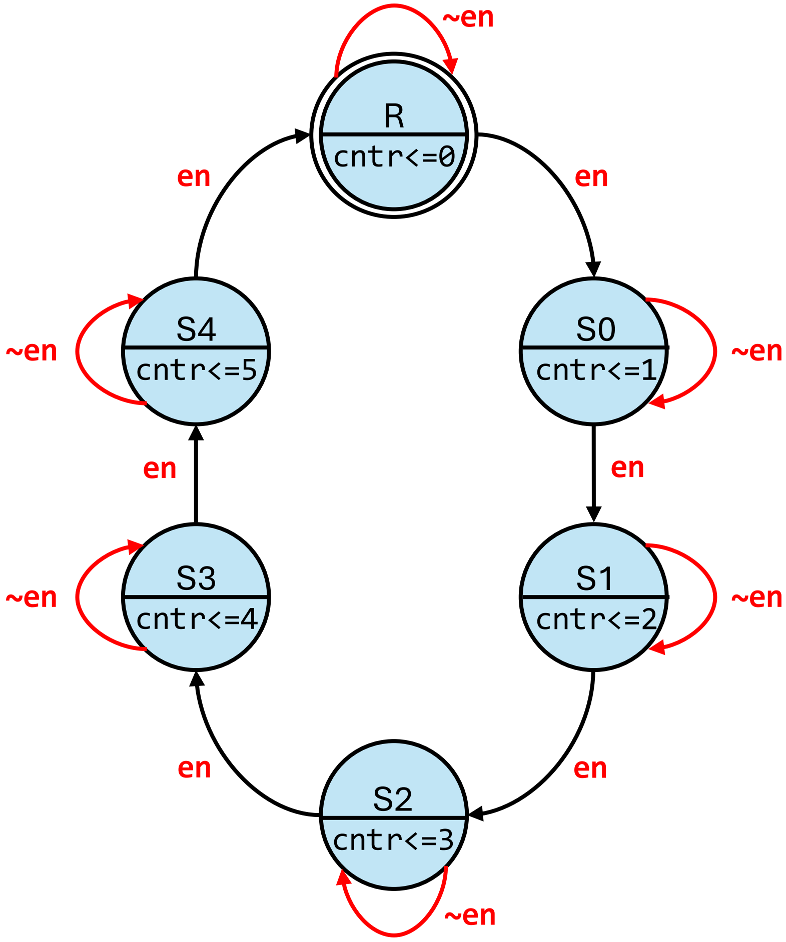
Simple Example: Externally Enabled Counter

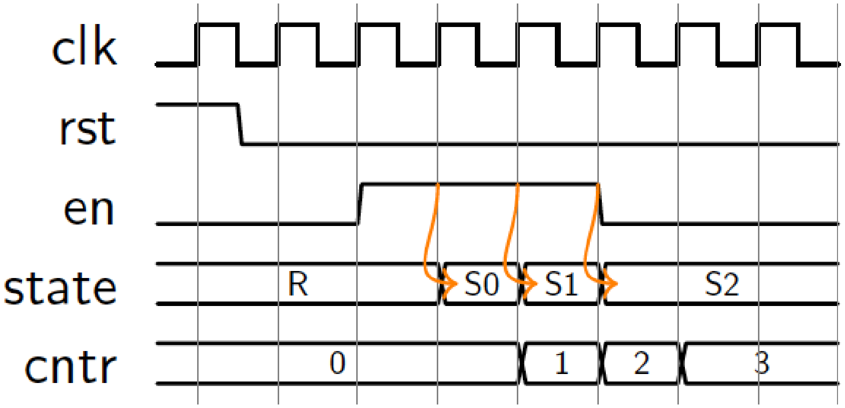
Modify the free-running counter we designed to have an enable signal. The counter advances only when enable is asserted.



Simple Example: Externally Enabled Counter

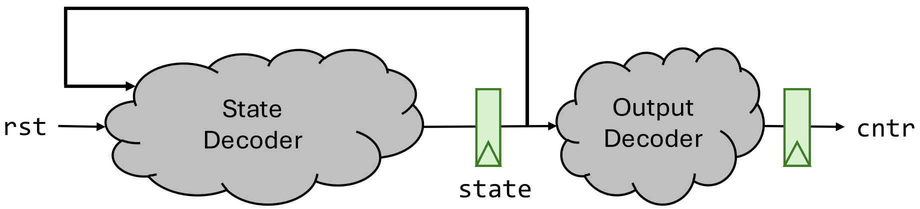
Modify the free-running counter we designed to have an enable signal. The counter advances only when enable is asserted.





Simple Example: Externally Enabled Counter

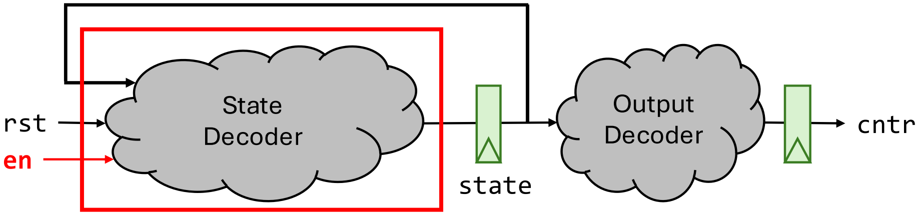
```
module cntr5 (
     input  clk,
     input  rst,
     output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = S0;
    S0: next_state = S1;
    S1: next_state = S2;
    S2: next_state = S3;
    S3: next_state = S4;
    S4: next_state = R;
    default: next_state = R;
  endcase
end

always_comb begin: out_decoder
  case (state)
    R : cntr_val = 0;
    S0: cntr_val = 1;
    S1: cntr_val = 2;
    S2: cntr_val = 3;
    S3: cntr_val = 4;
    S4: cntr_val = 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

Simple Example: Externally Enabled Counter

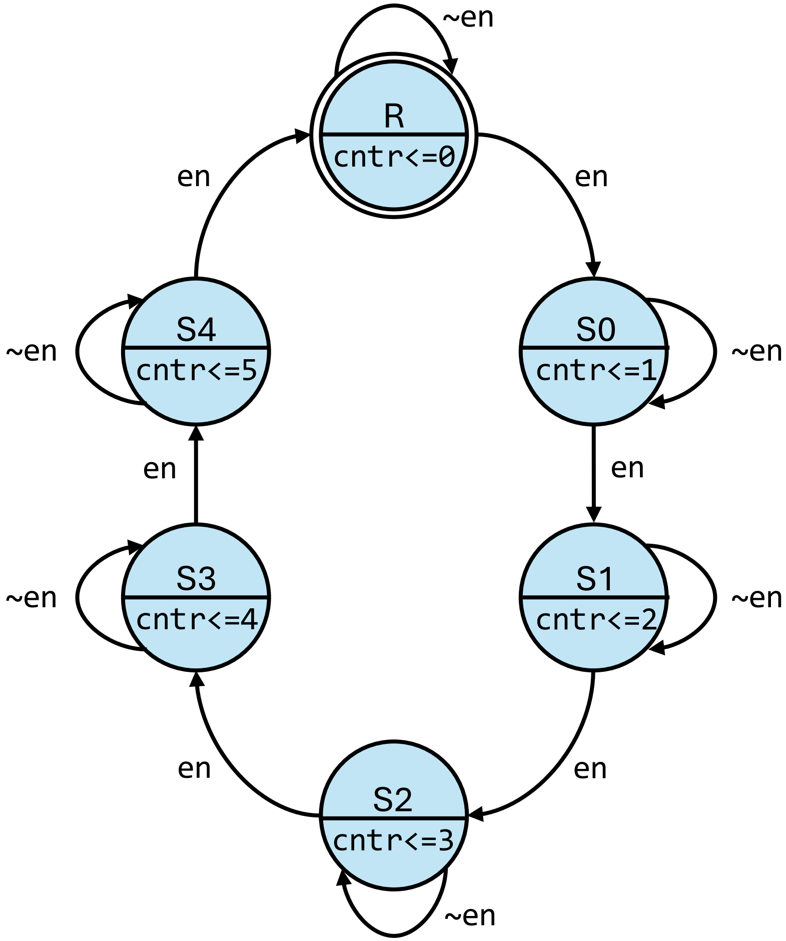
```
module cntr5_en (
      input  clk,
      input  rst,
      input  en,
      output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```

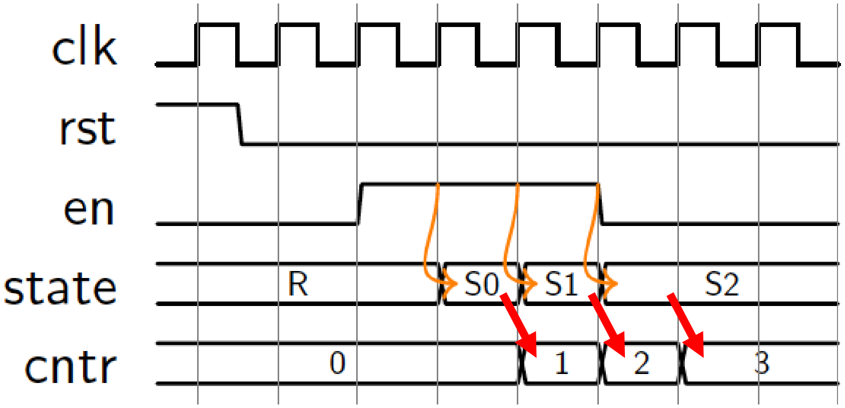


```
always_comb begin: state_decoder
  case (state)
    R : next_state = (en)? S0 : R;
    S0: next_state = (en)? S1 : S0;
    S1: next_state = (en)? S2 : S1;
    S2: next_state = (en)? S3 : S2;
    S3: next_state = (en)? S4 : S3;
    S4: next_state = (en)?  R : S4;
    default: next_state = R;
  endcase
end

always_comb begin: out_decoder
  case (state)
    R : cntr_val = 0;
    S0: cntr_val = 1;
    S1: cntr_val = 2;
    S2: cntr_val = 3;
    S3: cntr_val = 4;
    S4: cntr_val = 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

Synchronizing Outputs & State Transitions

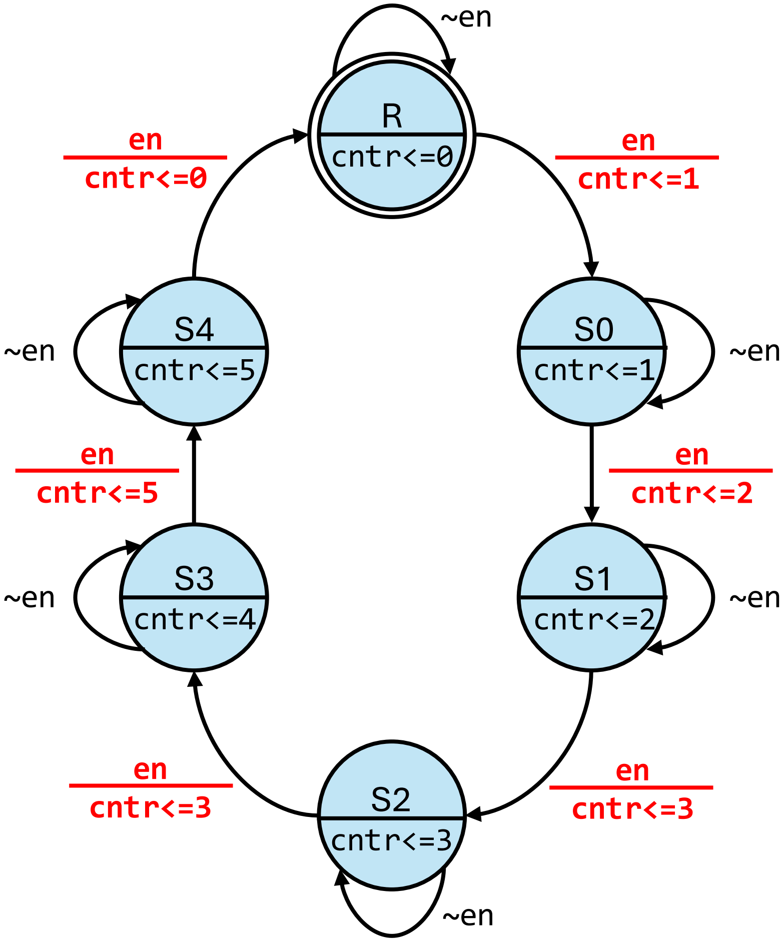
Is there a way to get rid of the one cycle lag between output updates and state transitions?



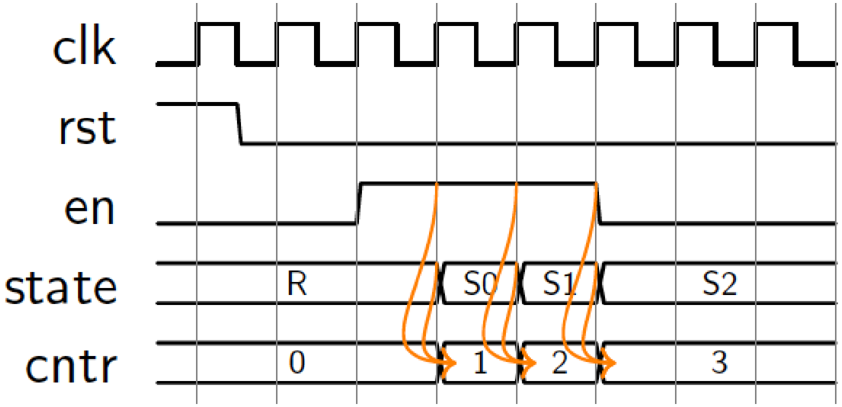


Synchronizing Outputs & State Transitions

Is there a way to get rid of the one cycle lag between output updates and state transitions?

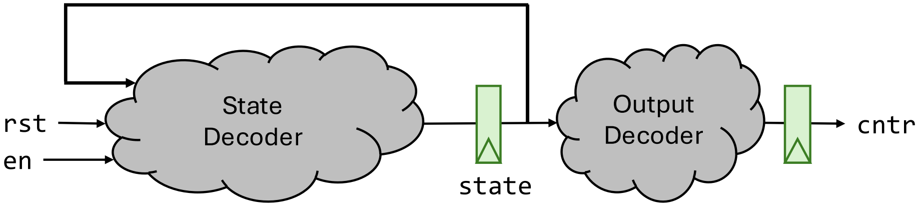


Transition actions!



Synchronizing Outputs & State Transitions

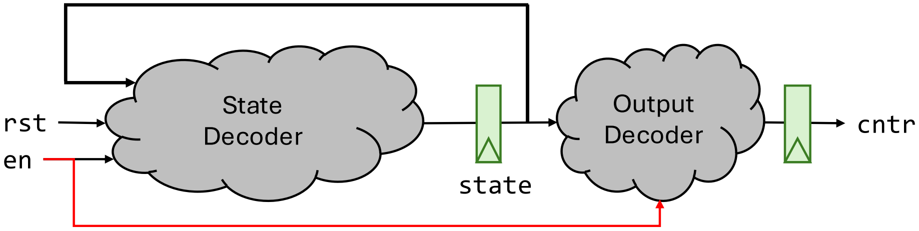
```
module cntr5_en (
      input  clk,
      input  rst,
      input  en,
      output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = (en)? S0 : R;
    S0: next_state = (en)? S1 : S0;
    S1: next_state = (en)? S2 : S1;
    S2: next_state = (en)? S3 : S2;
    S3: next_state = (en)? S4 : S3;
    S4: next_state = (en)?  R : S4;
    default: next_state = R;
  endcase
end

always_comb begin: out_decoder
  case (state)
    R : cntr_val = 0;
    S0: cntr_val = 1;
    S1: cntr_val = 2;
    S2: cntr_val = 3;
    S3: cntr_val = 4;
    S4: cntr_val = 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

Synchronizing Outputs & State Transitions

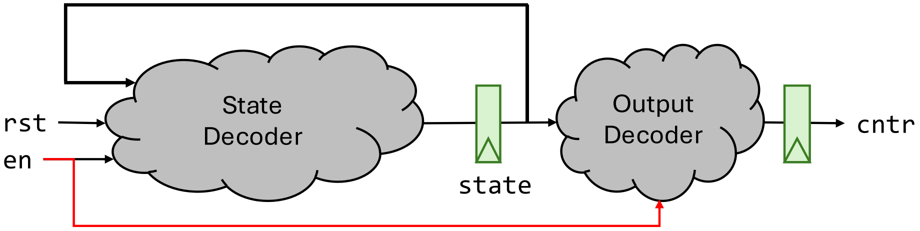
```
module cntr5_en_sync (
      input  clk,
      input  rst,
      input  en,
      output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = (en)? S0 : R;
    S0: next_state = (en)? S1 : S0;
    S1: next_state = (en)? S2 : S1;
    S2: next_state = (en)? S3 : S2;
    S3: next_state = (en)? S4 : S3;
    S4: next_state = (en)?  R : S4;
    default: next_state = R;
  endcase
end

always_comb begin: out_decoder
  case (state)
    R : cntr_val = (en)? 1 : 0;
    S0: cntr_val = (en)? 2 : 1;
    S1: cntr_val = (en)? 3 : 2;
    S2: cntr_val = (en)? 4 : 3;
    S3: cntr_val = (en)? 5 : 4;
    S4: cntr_val = (en)? 0 : 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

Synchronizing Outputs & State Transitions

```
module cntr5_en_sync (
        input  clk,
        input  rst,
        input  en,
        output logic [2:0] cntr
);

enum {R,S0,S1,S2,S3,S4} state, next_state;
logic [2:0] cntr_val;

always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 'd0;
        state <= R;
    end else begin
        cntr <= cntr_val;
        state <= next_state;
    end
end
```



```
always_comb begin: state_decoder
  case (state)
    R : next_state = (en)? S0 : R;
    S0: next_state = (en)? S1 : S0;
    S1: next_state = (en)? S2 : S1;
    S2: next_state = (en)? S3 : S2;
    S3: next_state = (en)? S4 : S3;
    S4: next_state = (en)?  R : S4;
    default: next_state = R;
  endcase
end

always_comb begin: out_decoder
  case (state)
    R : cntr_val = (en)? 1 : 0;
    S0: cntr_val = (en)? 2 : 1;
    S1: cntr_val = (en)? 3 : 2;
    S2: cntr_val = (en)? 4 : 3;
    S3: cntr_val = (en)? 5 : 4;
    S4: cntr_val = (en)? 0 : 5;
    default: cntr_val = 0;
  endcase
end

endmodule
```

Drawbacks?!

Next Lecture

More examples on FSM design …