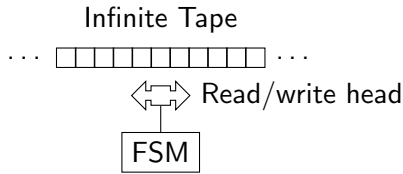
State Machines

Nachiket Kapre
nachiket@uwaterloo.ca

Outline

- Need for state machines
- Kinds of state machines in hardware
- Drawing waveforms + RTL coding styles

Why is state necessary?



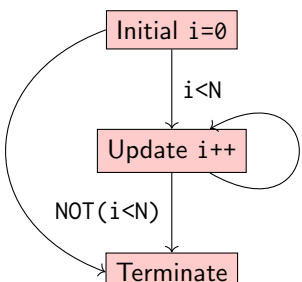
- CS theory tells us about Turing Machines!
  - Infinite tape (memory) + FSM (finite-state machine)
- (1) State (infinite tape) is a pre-requisite to compute any function that is computable! $\rightarrow$ Yes, state is necessary for computation
  - Not just that, it must be **infinite**!
- (2) FSMs here have **finite** state and manage/control the computation

State Machines in Software

- **Trivial case**:In software, `for` loops implicitly use state-machines.
  - Loop counter is the state
  - In each state, execute body of the loop
  - Update loop counter (state) and repeat
  - `JUMP` or `BRANCH` instruction is your state transition
- **Event-driven design**: Repetitive, iterative tasks need control management
  - Systematic way to reason about ordering of events
  - Define rules governing which states are allowed
  - Potentially avoid logical bugs $\rightarrow$ assert states you can never enter
  - *e.g.*: Device drivers, IO protocols are examples of state machines

Implicit State Machines in C

```
for(int i=0; i<N; i++) {
 y[i]=a[i]*x[i]*x[i]+b[i]*x[i]+c[i];
}
```



- For loops are simplest state machines in C
  - Three states: Initial, Update, Terminate
  - **Initial**: `int i=0`
  - **Update**: compute `y[i]`, `i++`
  - **Terminate**: `i==N-1`
- Conveniently pack specification of all states, transition operations in one line of code
  - `for(int i=0;i<N;i++)`
- Can visually represent them as a graph (nodes = states, edges = state transitions)

The story so far...

- Combinational logic is stateless (memoryless), composed of logic gates, arithmetic operations, and wires
- Registers add state. But, we have only covered **feed-forward** design $\rightarrow$ data flows in one direction only!
- Fundamental new idea in state machine design is that of **feed-back** $\rightarrow$ data can flow backwards in a particular manner!

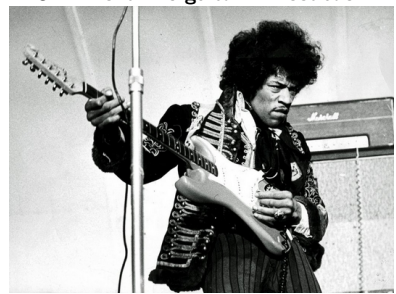
The story so far...

- Combinational logic is stateless (memoryless), composed of logic gates, arithmetic operations, and wires
- Registers add state. But, we have only covered **feed-forward** design $\rightarrow$ data flows in one direction only!
- Fundamental new idea in state machine design is that of **feed-back** $\rightarrow$ data can flow backwards in a particular manner!

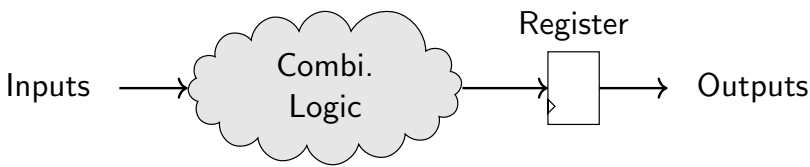
Feedforward vs. Feedback

One Direction = Feedforward

Jimi Hendrix's guitar = Feedback

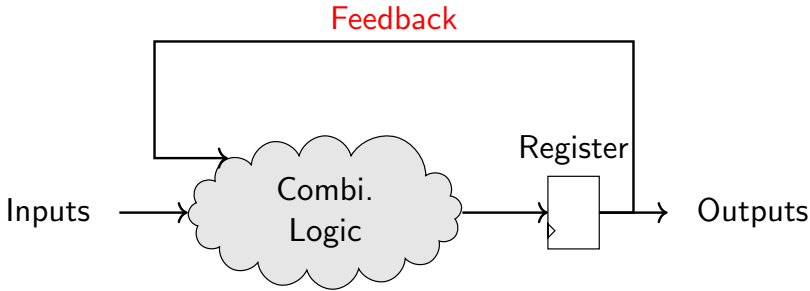


High-Level View of State Machines



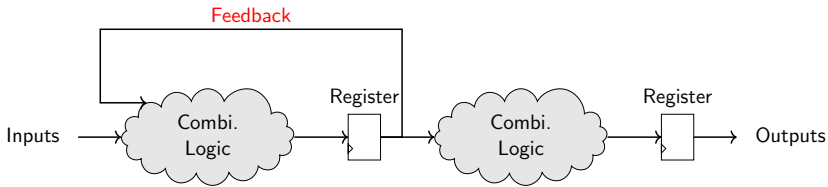
- Presence of feedback fundamental to the idea of state machines
- State machines are a powerful abstraction for reasoning about computation in presence of feedback

High-Level View of State Machines



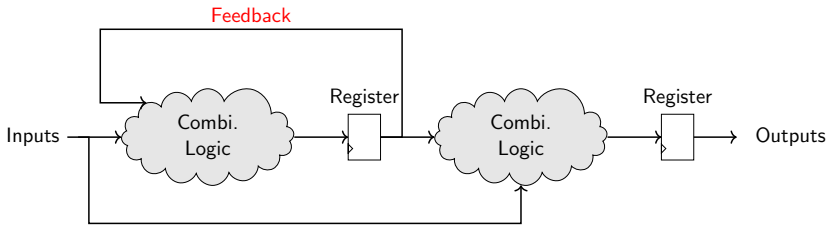
- Presence of feedback fundamental to the idea of state machines
- State machines are a powerful abstraction for reasoning about computation in presence of feedback

High-Level View of State Machines



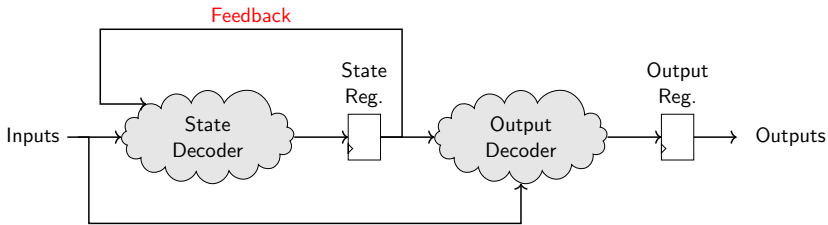
- Optionally the output computation may pass through another stage of combination logic + register
- Possible (not recommended) option to use inputs directly to compute outputs

High-Level View of State Machines



- Optionally the output computation may pass through another stage of combination logic + register
- Possible (not recommended) option to use inputs directly to compute outputs

High-Level View of State Machines

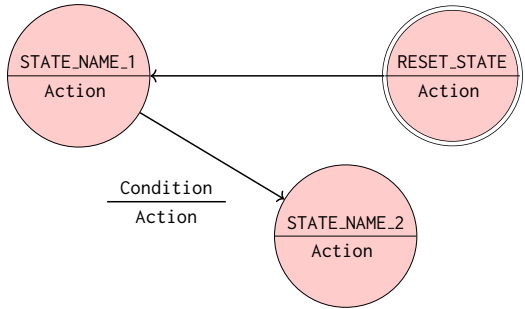


- Optionally the output computation may pass through another stage of combination logic + register
- Possible (not recommended) option to use inputs directly to compute outputs

Learning Goals

- How to draw a state machine?
  - Reason about correctness/performance here!
- How to draw a timing diagram for a state machine?
  - Expect a certain behavior of your computation
- How to implement the state machine in RTL
  - Write RTL last! Once you have a visual representation + waveform analysis

Notation

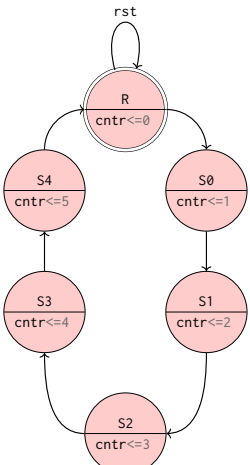


- Each state is a node in the graph – pack both state name + state action (if present) in the same bubble.
- Each transition is an edge in the graph – pack both condition (if present) and state action (if present) in the same edge.
- Show reset state with double circle (or other shape with two borders)

Recipe for State-Machine Engineering in Hardware

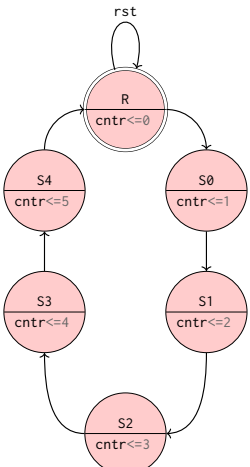
- 1. Choose number of states you need for your problem
- 2. Define state transition conditions clearly between states
- 3. Identify what actions happen in each state (or transition)
- 4. Check if your machine is correct $\rightarrow$
  - Does it get stuck in a state and never comes out (unless that is by design)
  - Is a state never reachable? Dead code. Wasting states+area
  - Check for causality $\rightarrow$ time travel is strictly forbidden
- 5. Optimize for performance $\rightarrow$
  - Use **one-hot** encoding of state bits. Needs a bit more registers, but reduces decoder cost
  - If outputs go to different module/RTL developer, they may come back saying register this signal $\rightarrow$ An extra cycle of delay requires redesign of states
  - Actions on transitions could have a slight performance penalty

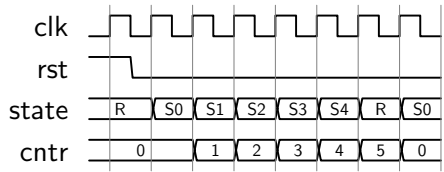
Example 1: Free-running count to 5



- **Problem**: design a counter that counts from 0 to 5, and restarts from 0, running forever
- Number of states = 6, No inputs, Output = 3-bit count
- State transitions have no conditions, no actions → state machine will transition between states on each clock
- Each state has a simple action to assign a constant value to `cntr` output

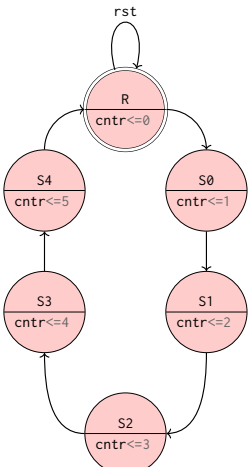
Example 1: Free-running count to 5

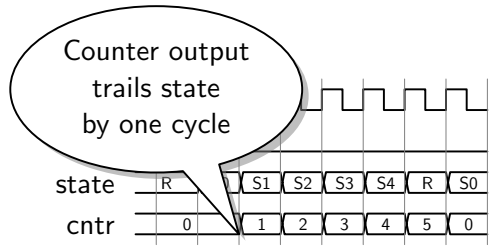




- State labels shown directly in the data bundles for `state` row
- Output of state machine trails the state by one cycle, as outputs are registered!
  - If output is combinational (not recommended), it will coincide with the state

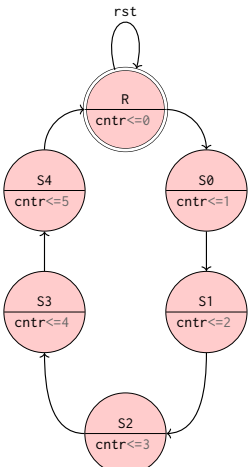
Example 1: Free-running count to 5

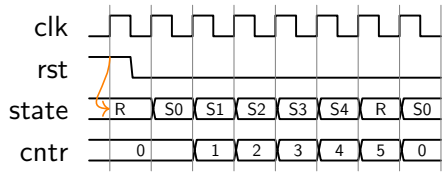




- State labels shown directly in the data bundles for `state` row
- Output of state machine trails the state by one cycle, as outputs are registered!
  - If output is combinational (not recommended), it will coincide with the state

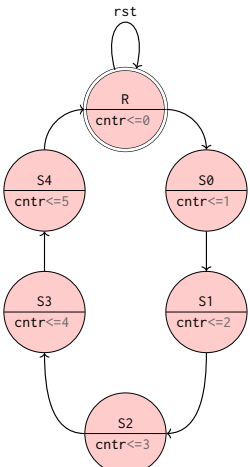
Example 1: Free-running count to 5

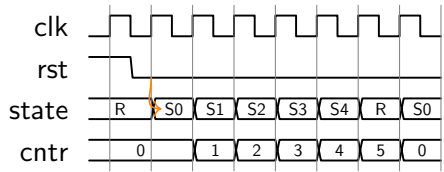




- State labels shown directly in the data bundles for `state` row
- Output of state machine trails the state by one cycle, as outputs are registered!
  - If output is combinational (not recommended), it will coincide with the state

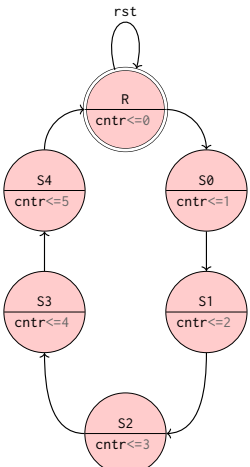
Example 1: Free-running count to 5

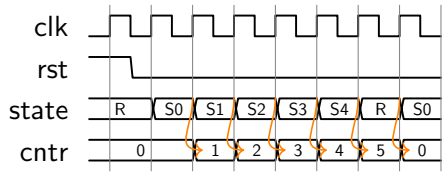




- State labels shown directly in the data bundles for `state` row
- Output of state machine trails the state by one cycle, as outputs are registered!
  - If output is combinational (not recommended), it will coincide with the state

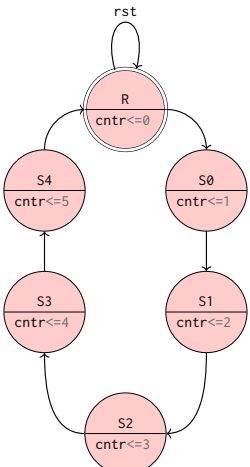
Example 1: Free-running count to 5





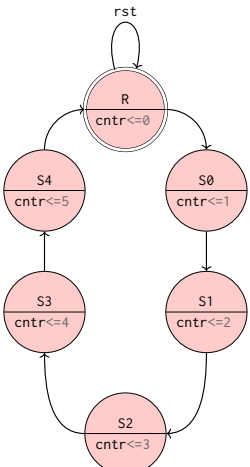
- State labels shown directly in the data bundles for `state` row
- Output of state machine trails the state by one cycle, as outputs are registered!
  - If output is combinational (not recommended), it will coincide with the state

Example 1: Free-running count to 5



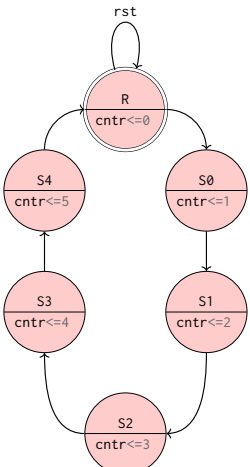

```
module cntr5 (
  input wire clk,
  input wire rst,
  output reg [2:0] cntr
 );
```

Example 1: Free-running count to 5




```
enum {R,S0,S1,S2,S3,S4} state; //
```

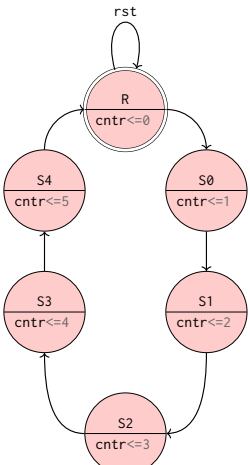
Example 1: Free-running count to 5



Enumerated SystemVerilog type

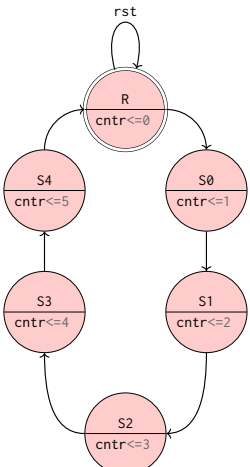
```
enum {R,S0,S1,S2,S3,S4} state; //
```

Example 1: Free-running count to 5




```
always@(posedge clk) begin: run_stmc
 if (rst) begin
  state <= R; //
 end else begin
  case (state)
   R       : state <= S0;
   S0      : state <= S1;
   S1      : state <= S2;
   S2      : state <= S3;
   S3      : state <= S4;
   S4      : state <= R;
   default : state <= R; //
  endcase
 end
end
```

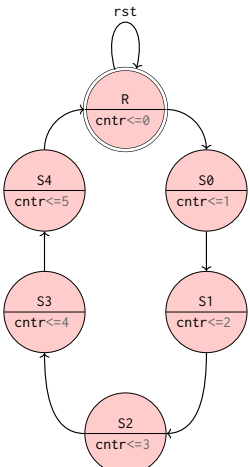
Example 1: Free-running count to 5




```
always@(posedge clk) begin: run_stmc
 if (rst) begin
  state <= R; //
 end else begin
  case (st
   R
   S0
   S1
   S2      : state <= S3;
   S3      : state <= S4;
   S4      : state <= R;
   default : state <= R; //
  endcase
 end
end
```

Ensure state machine starts in sensible state

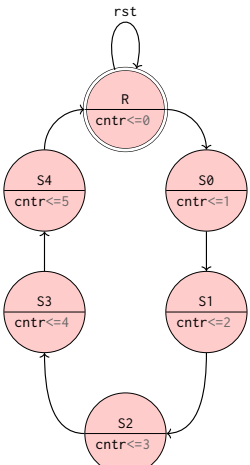
Example 1: Free-running count to 5




```
always@(posedge clk) begin: run_stmc
 if (rst) begin
  state <= R; //
 end else begin
  case (state)
   R       : state <= S0;
   S0      : state <= S1;
   S1      : state <= S2;
   S2      : state <= S3;
   S3      : state <= S4;
   S4      : state <= R;
   default : state <= R; //
  endcase
 end
end
```

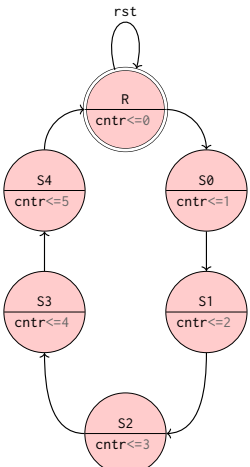
Default not necessary for enum types

Example 1: Free-running count to 5




```
always@(posedge clk) begin: compute
 if (rst) begin
  cntr <= 0;
 end else begin
  case (state)
   R       : cntr <= 0; //
   S0      : cntr <= 1;
   S1      : cntr <= 2;
   S2      : cntr <= 3;
   S3      : cntr <= 4;
   S4      : cntr <= 5;
   default : cntr <= 0;
  endcase
 end
end
```

Example 1: Free-running count to 5



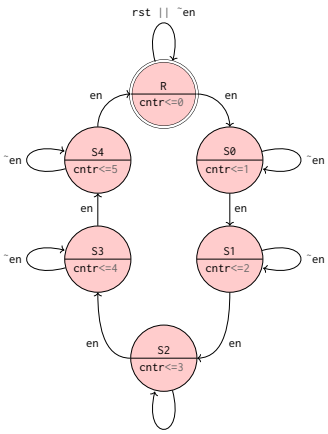

```
always@(posedg
 if (rst) b
  cntr <= 0;
 end else begin
  case (state)
   R       : cntr <= 0; //
   S0      : cntr <= 1;
   S1      : cntr <= 2;
   S2      : cntr <= 3;
   S3      : cntr <= 4;
   S4      : cntr <= 5;
   default : cntr <= 0;
  endcase
 end
end
```

Registered action per state

RTL Coding Guidelines for State Machines

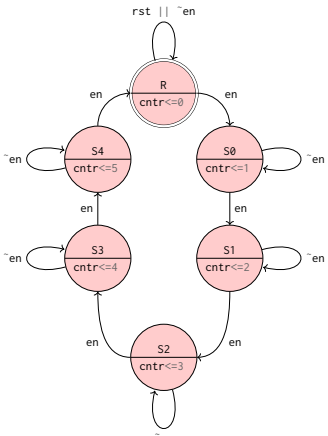
- An RTL state machine needs a `clk` and a `rst`
- It may have multiple (or no) inputs, but will generally have atleast one output
- Reset self loop need not be shown (assume its there)
- State decoder block is a separate always block → `run_stmc:` block in our code example
- Output decoder block is its own block → `compute:` block in our code example
- We used **`case`** statement since the state machine cannot be in two states at the same time
- No `others` clause required as the state enumeration is exhaustive → do not forget a state

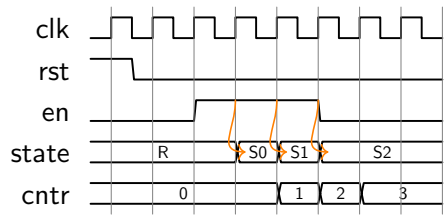
Example 2: Externally enabled count to 5



- **Problem**: design a counter that counts from 0 to 5, and restarts from 0, running forever. Count updates only on enable!
- Number of states = 6, 1-bit enable input, Output = 3-bit count
- Two kinds of state transitions
  - Normal advancement if en
  - Self loop if ~en
- Each state still has a simple action to assign a constant value to `cntr` output

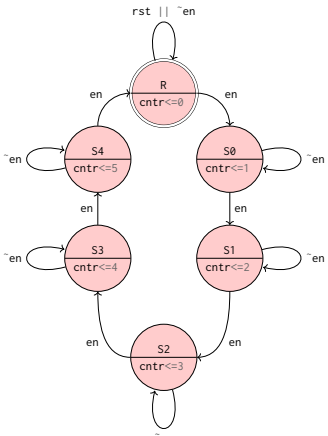
Example 2: Externally enabled count to 5

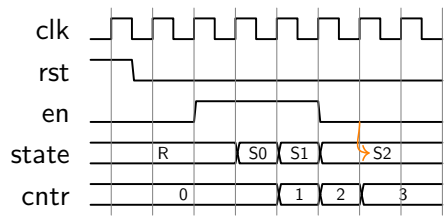




- Input en advances state machine
- State stops changing a cycle after en goes low
- Output trails the state by one cycle.
- Output stabilizes two cycles after en goes low

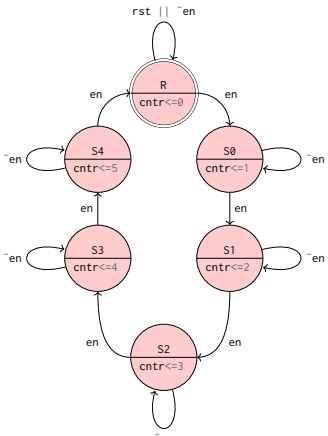
Example 2: Externally enabled count to 5

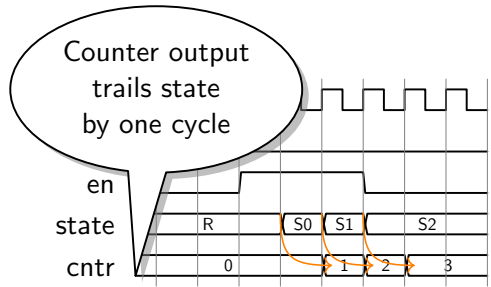




- Input en advances state machine
- State stops changing a cycle after en goes low
- Output trails the state by one cycle.
- Output stabilizes two cycles after en goes low

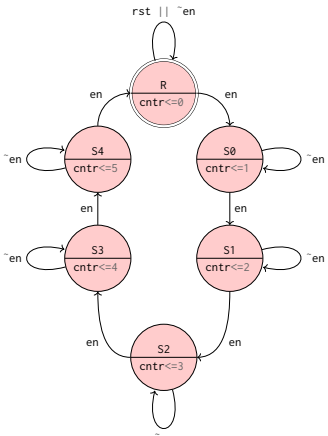
Example 2: Externally enabled count to 5

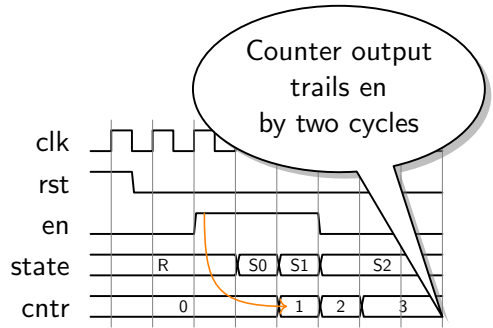




- Input en advances state machine
- State stops changing a cycle after en goes low
- Output trails the state by one cycle.
- Output stabilizes two cycles after en goes low

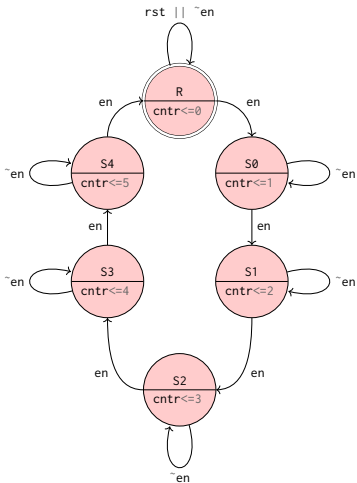
Example 2: Externally enabled count to 5





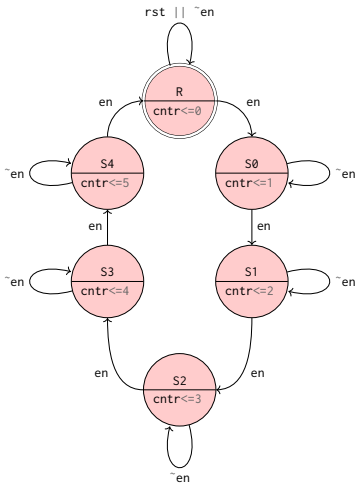
- Input en advances state machine
- State stops changing a cycle after en goes low
- Output trails the state by one cycle.
- Output stabilizes two cycles after en goes low

Example 2: Externally enabled count to 5



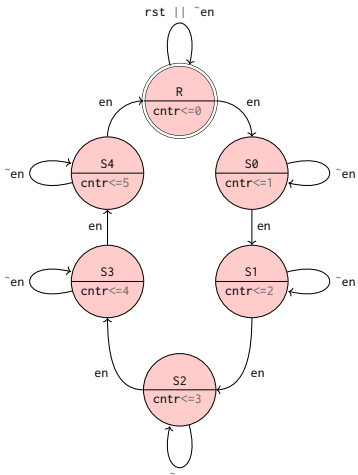

```
module cntr5en (
  input wire clk,
  input wire rst,
  input wire en,
  output reg [2:0] cntr
 );
```

Example 2: Externally enabled count to 5




```
always@(posedge clk) begin : run_stmc
 if (rst) begin
  state <= R;
 end else begin
  case (state)
   R       : if (en) state <= S0;
   S0      : if (en) state <= S1;
   S1      : if (en) state <= S2;
   S2      : if (en) state <= S3;
   S3      : if (en) state <= S4;
   S4      : if (en) state <= R;
   default : state <= R;
  endcase
 end
end
```

Example 2: Externally enabled count to 5

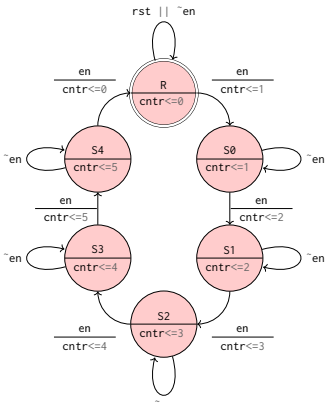



```
always@(posedge clk) begin : compute
 if (rst) begin
  cntr <= 0;
 end else begin
  case (state)
   R       : cntr <= 0;
   S0      : cntr <= 1;
   S1      : cntr <= 2;
   S2      : cntr <= 3;
   S3      : cntr <= 4;
   S4      : cntr <= 5;
   default : cntr <= 0;
  endcase
 end
end
```

RTL Coding Impact of en input

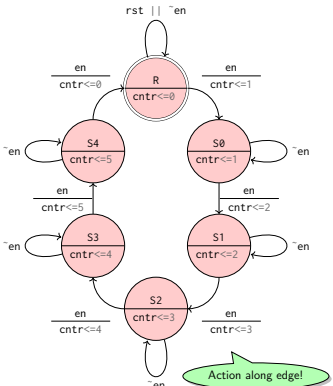
- Input is only read in the `run_stmc:` block
  - This block advances the state machine
  - Nested **`if`** statements inside a **`case`** statement
  - Processing of ~en is not explicit
  - Missing **`else`** will cause state to hold onto its current value.
- Output decoder block block → `compute:` is unchanged!!
- Separation of input processing and output generation
- Note the one cycle delay required to reflect input change on state
- And the second cycle delay required to reflect state change on output change

Example 3: Externally enabled count to 5 with fast out



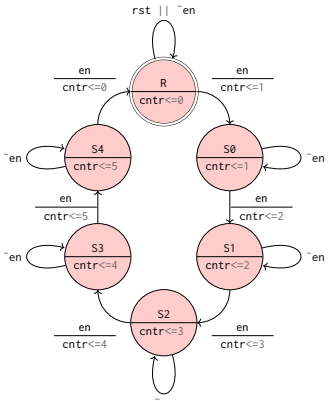
- **Problem**: design a counter that counts from 0 to 5, and restarts from 0, running forever. Count updates only on enable! Result is available sooner!
- Number of states = 6, 1-bit enable input, Output = 3-bit count
- Two kinds of state transitions
  - Normal advancement if en
  - Self loop if ~en
- Each state still has a simple action to assign a constant value to `cntr` output

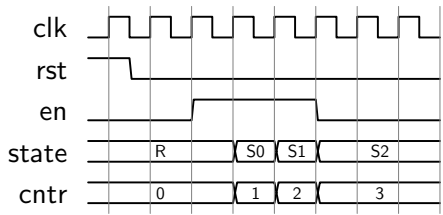
Example 3: Externally enabled count to 5 with fast out



- **Problem**: design a counter that counts from 0 to 5, and restarts from 0, running forever. Count updates only on enable! Result is available sooner!
- Number of states = 6, 1-bit enable input, Output = 3-bit count
- Two kinds of state transitions
  - Normal advancement if en
  - Self loop if ~en
- Each state still has a simple action to assign a constant value to `cntr` output

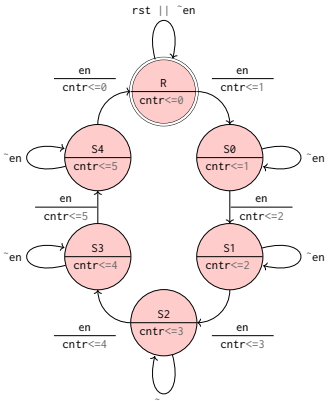
Example 3: Externally enabled count to 5 with fast out

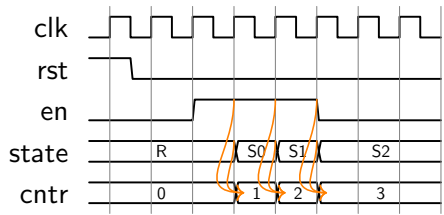




- Now have faster update to `cntr` in the timing diagram
- This is possible by modifying output decoder to check both input and state
  - By predicting the next state value, output can be sped up by one cycle to change in-sync with the state

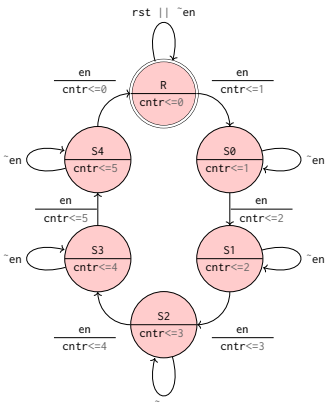
Example 3: Externally enabled count to 5 with fast out

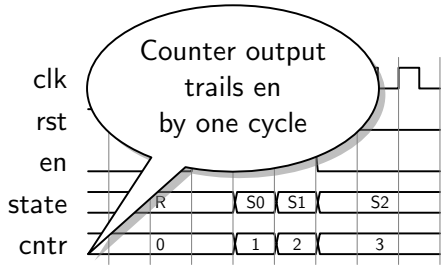




- Now have faster update to `cntr` in the timing diagram
- This is possible by modifying output decoder to check both input and state
  - By predicting the next state value, output can be sped up by one cycle to change in-sync with the state

Example 3: Externally enabled count to 5 with fast out

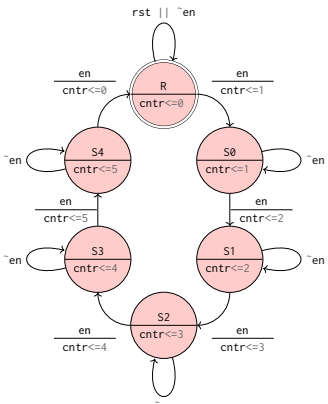




- Now have faster update to `cntr` in the timing diagram
- This is possible by modifying output decoder to check both input and state
  - By predicting the next state value, output can be sped up by one cycle to change in-sync with the state

Output **nsync** with state

Example 3: Externally enabled count to 5 with fast out




```
case(state)
     ...
 R  : if(en) state <= S0; //
     ...
endcase
```

Unconditional state transition

```
case(state)
     ...
 R : begin
  cntr <= 0;
  if(en) cntr <= 1;
 end //
```

Action while changing state!

RTL Coding Impact of faster output

- Output can be made to align with state transition
  - Requires modification to the `compute` block to include **`if`** condition
  - Again, nested **`if`** statements inside a **`case`** statement
  - Output will acquire default value assigned at start of state $\rightarrow$ hence **`else`** not needed and still OK!
  - Ignoring the **`else`** on the output will infer unwanted state on the output registers.
- Thus, single cycle delay from input to state as well as input to output.
- **Effect**: More complex output decoder design, can often be slower in frequency

Wrapup

- State Machines are a **compact** way to represent computation with some notion of memory or state (feedback)
- RTL offers a rich set of patterns to express state machines with clocked+combinational blocks
- Design challenge is to identify how many states you need, and to determine if actions happen inside a state, or along an edge