Electronics Systems (938II)

Lecture 2.4

Building Blocks of Electronic Systems – Finite State Machine

Finite State Machine (FSM)

- A **Finite State Machine** (**FSM**) is a sequential circuit that:
  - Has a finite (limited) number of states
  - Is in exactly one of the finite states at any given time

- The evolution between the states depends on
  - Inputs
  - Current state
  - In fact, it is a sequential circuit

Finite State Machine (FSM)

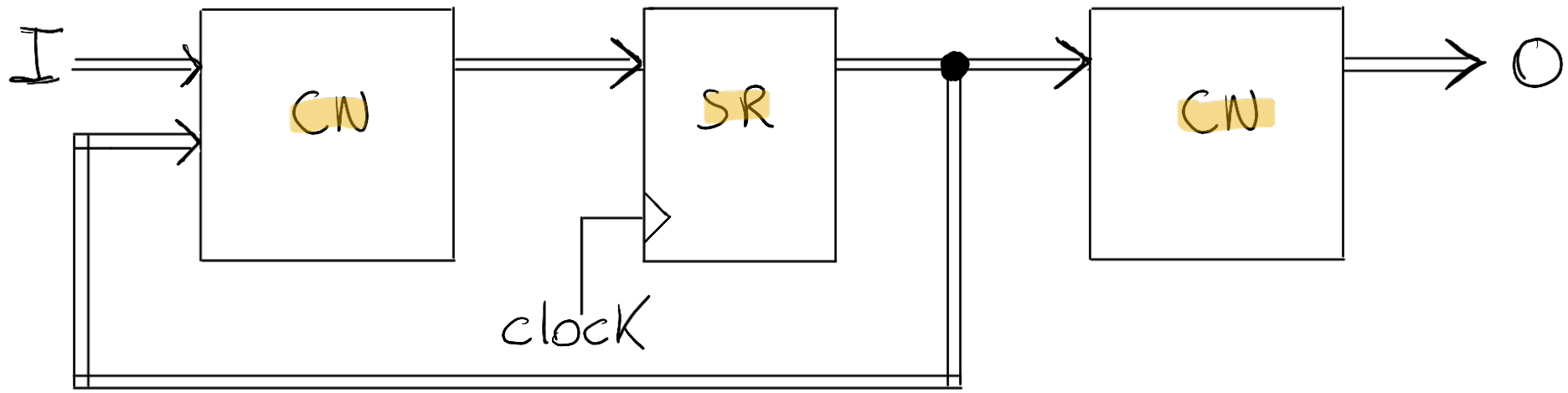
- Memory resources are required to store the current state
  - We have already seen an example
    - Latches
    - Flip-flops/Registers

  - If the FSM employs latches, it is asynchronous
  - If it employes a register, it is synchronous
    - By default, we will assume synchronous FSM in the next slides
    - This means that the state change occurs on the rising edge of the clock signal

Seq. networks: out vs assigned only on the edge of the clock if we have synchronous machines.

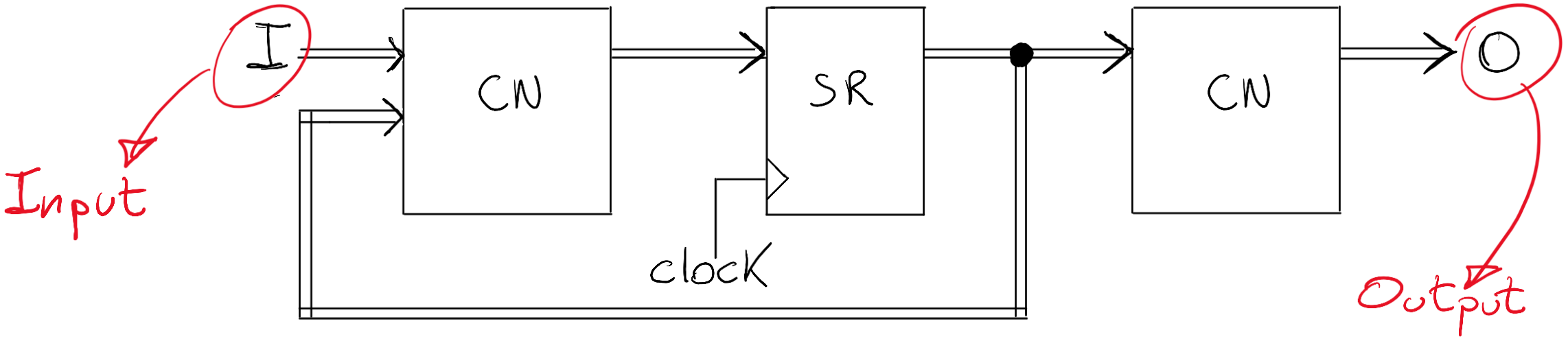
Finite State Machine (FSM)

- Outline of an FSM



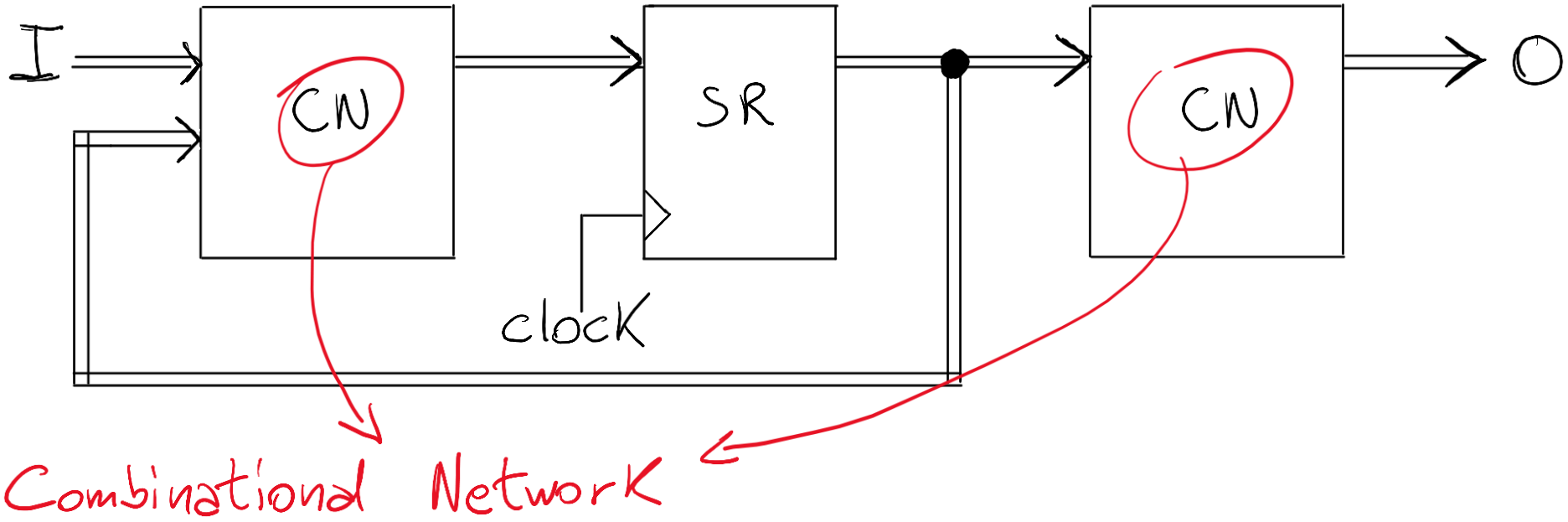
Finite State Machine (FSM)

- Outline of an FSM



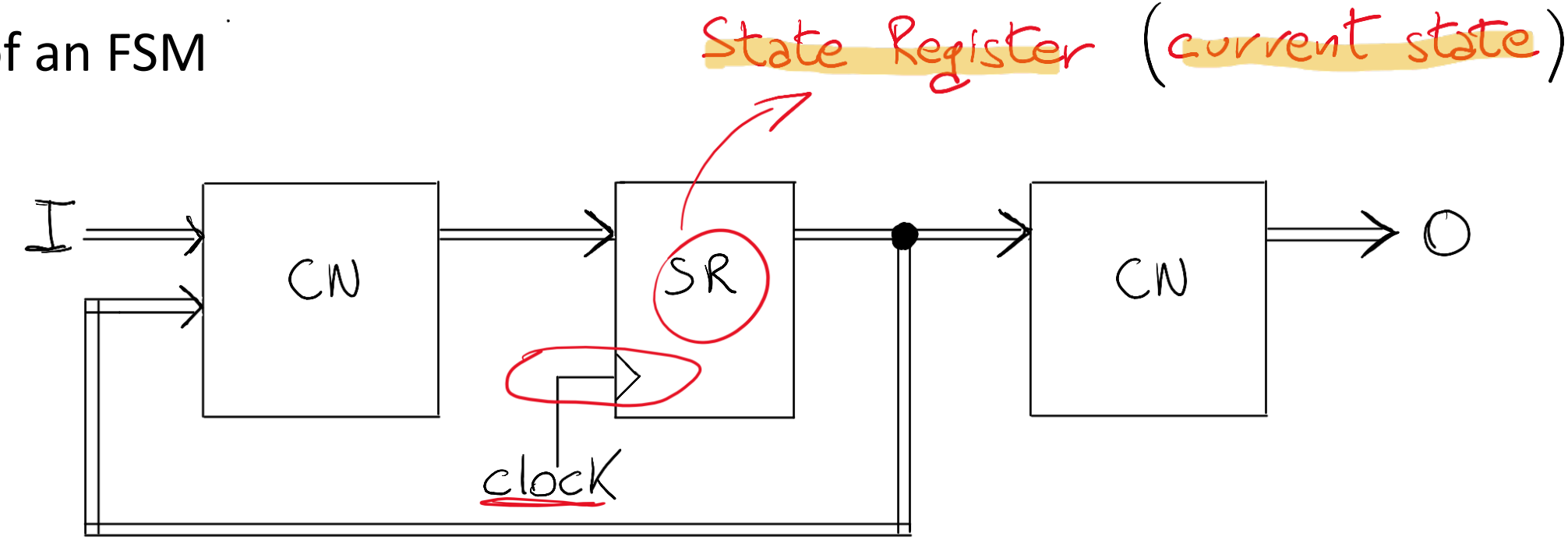
Finite State Machine (FSM)

- Outline of an FSM



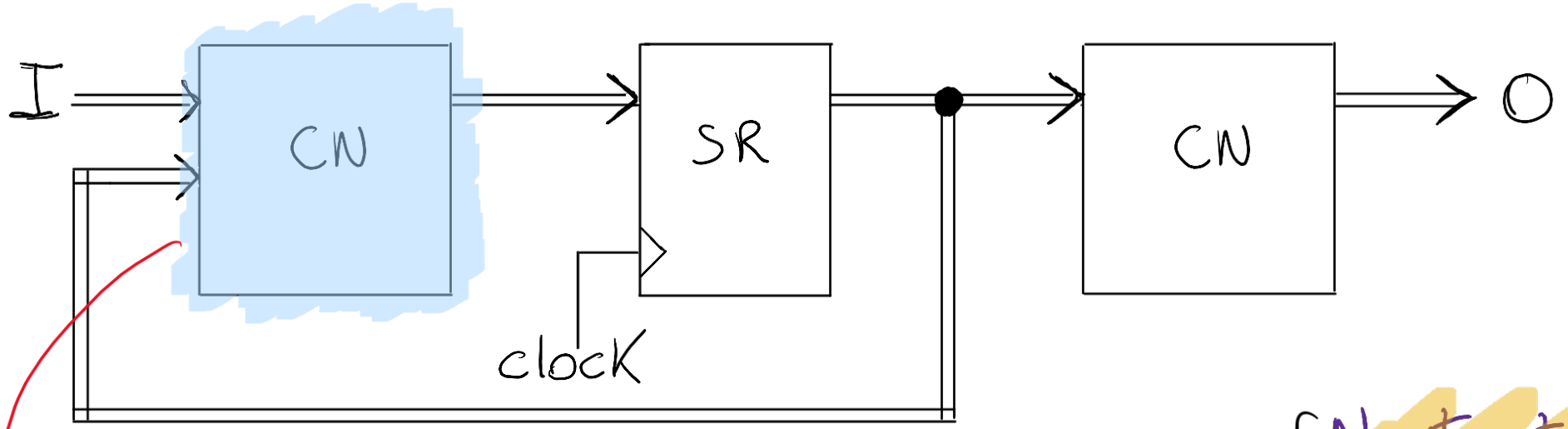
Finite State Machine (FSM)

- Outline of an FSM



Finite State Machine (FSM)

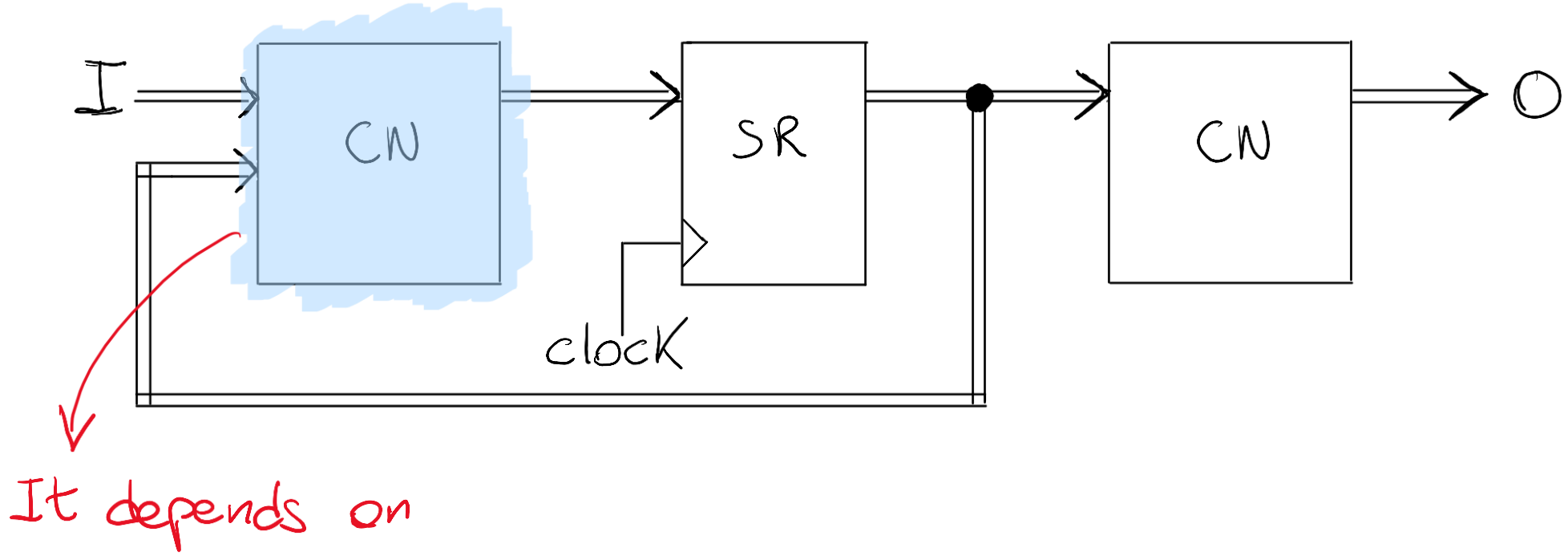
- Outline of an FSM



This is the circuit that defines the next state → Next state Logic

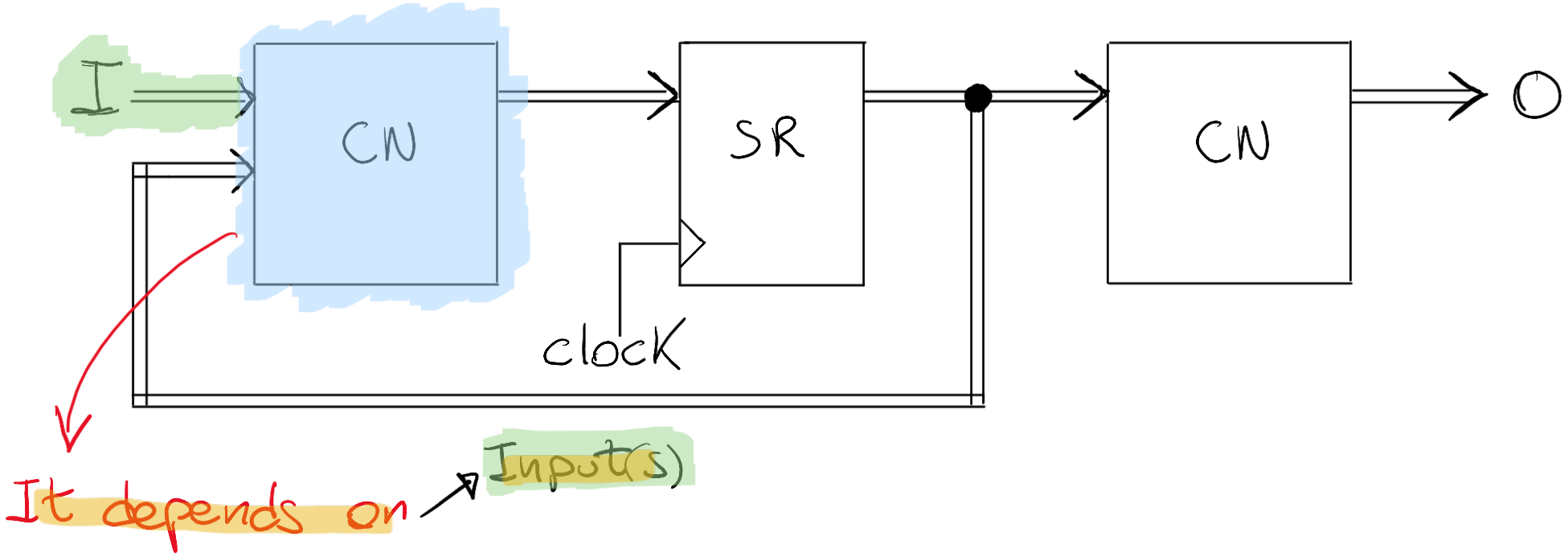
Finite State Machine (FSM)

- Outline of an FSM



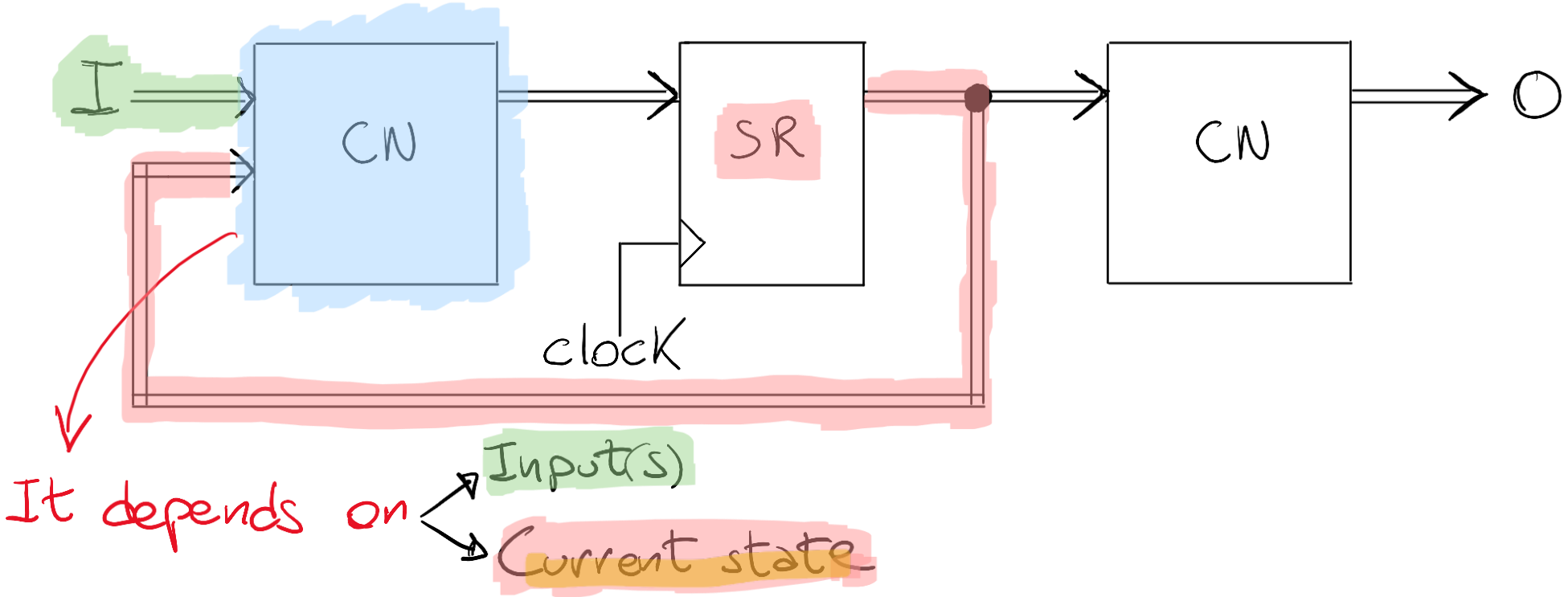
Finite State Machine (FSM)

- Outline of an FSM



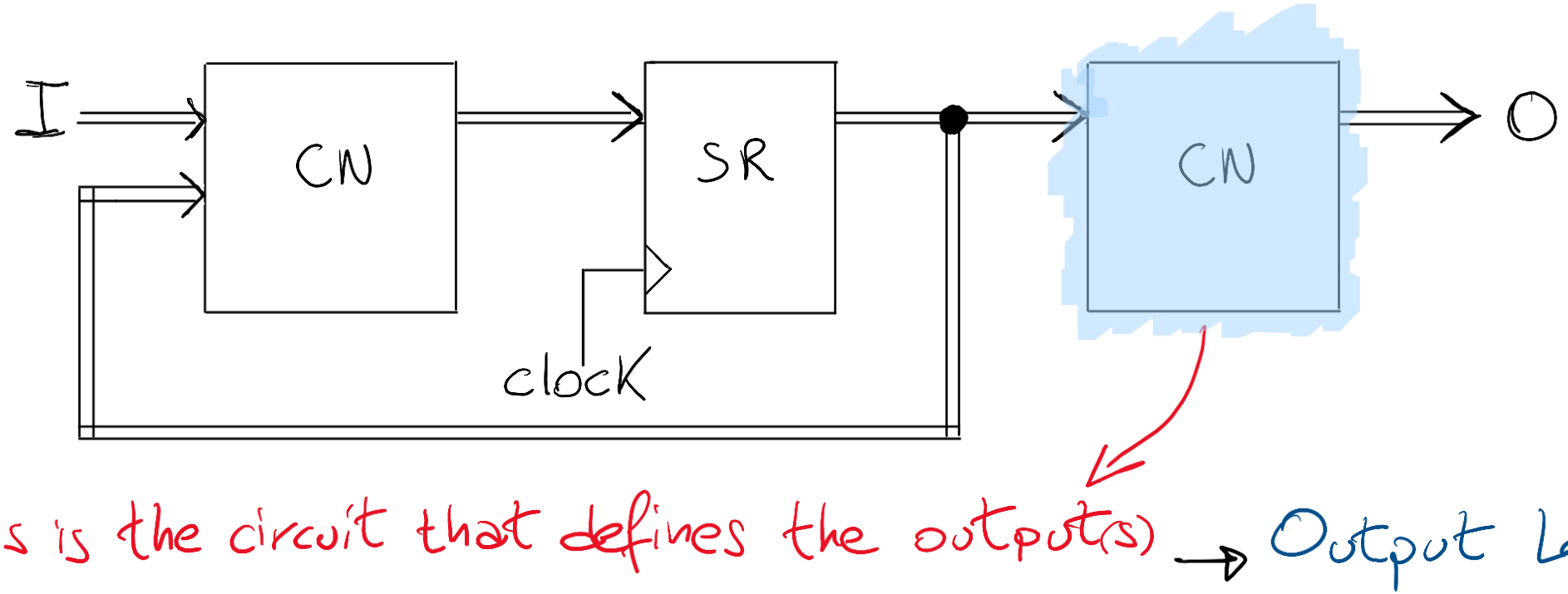
Finite State Machine (FSM)

- Outline of an FSM



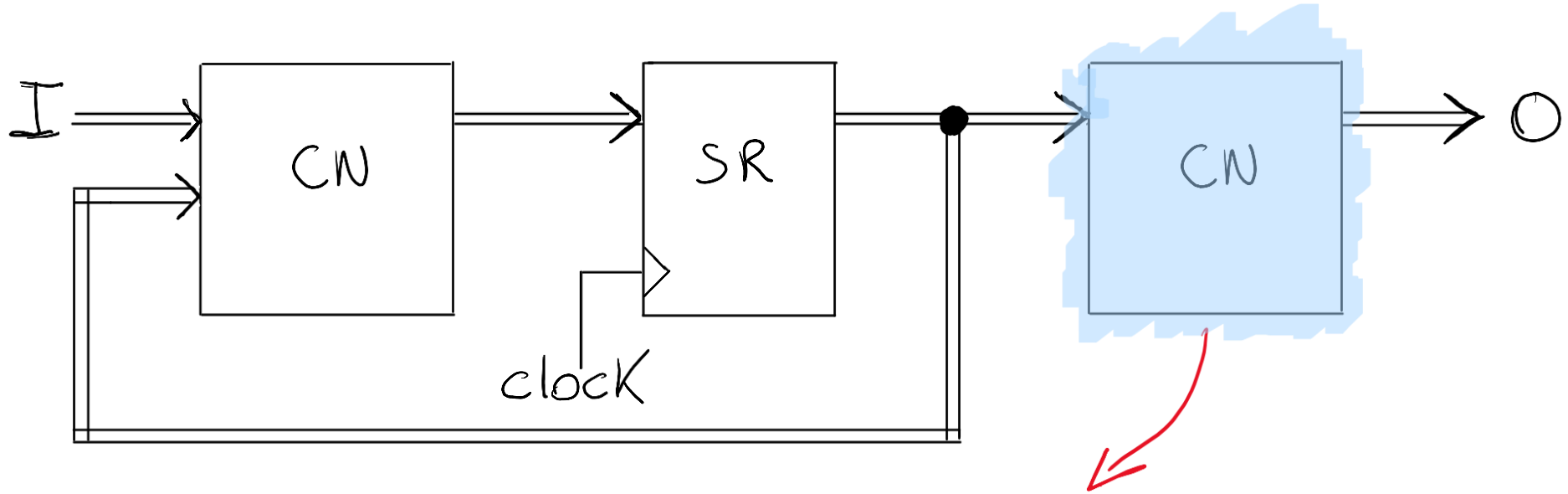
Finite State Machine (FSM)

- Outline of an FSM



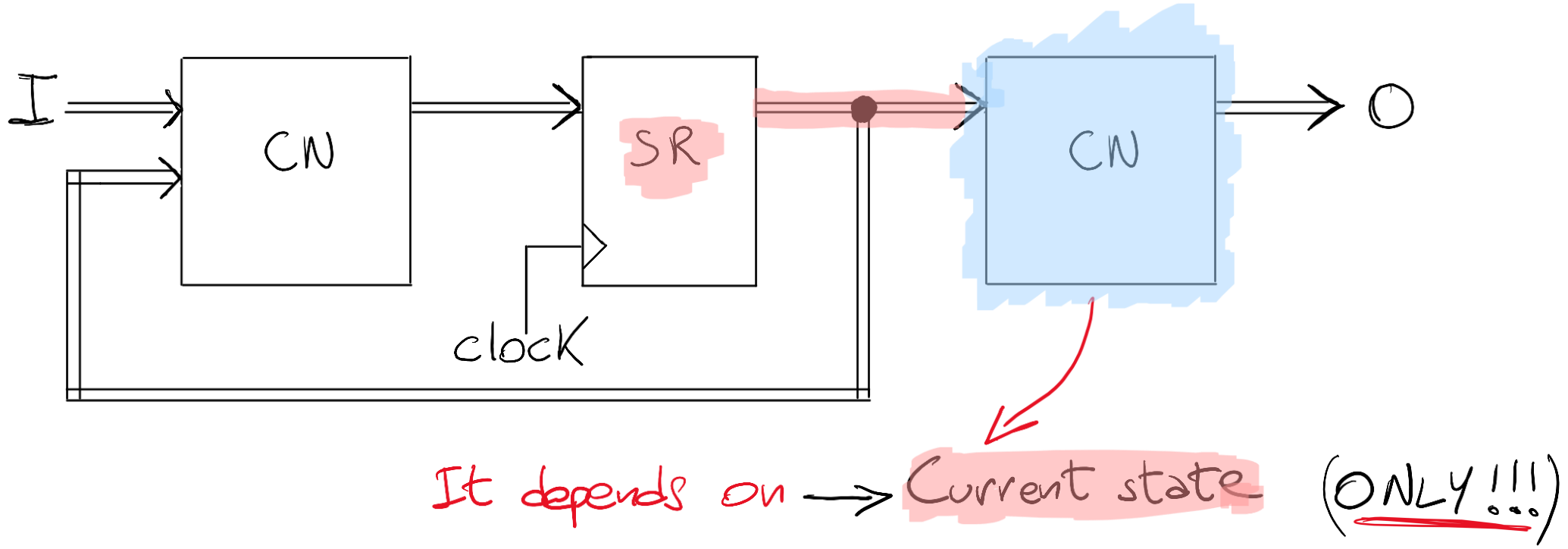
Finite State Machine (FSM)

- Outline of an FSM



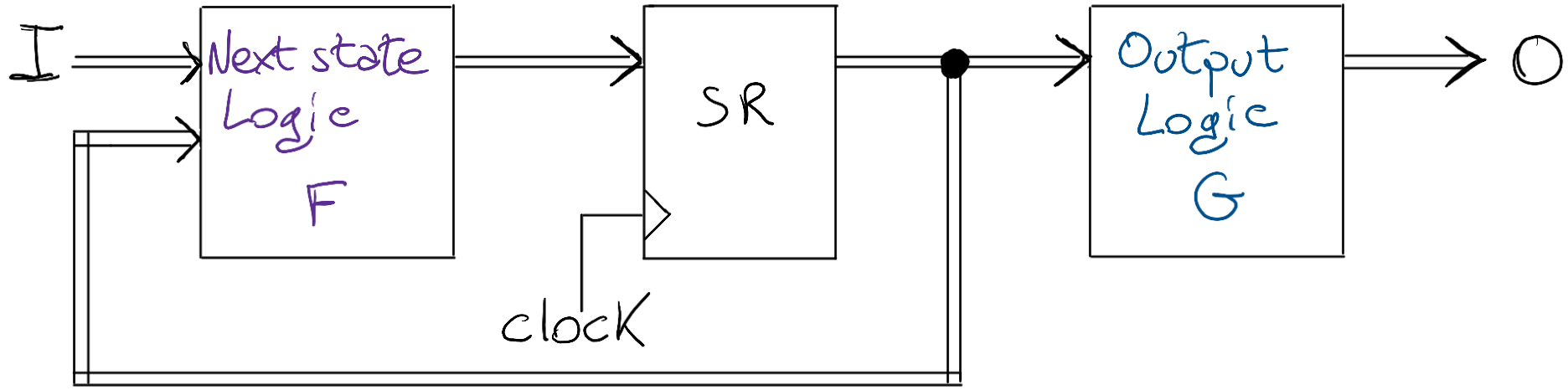
Finite State Machine (FSM)

- Outline of an FSM



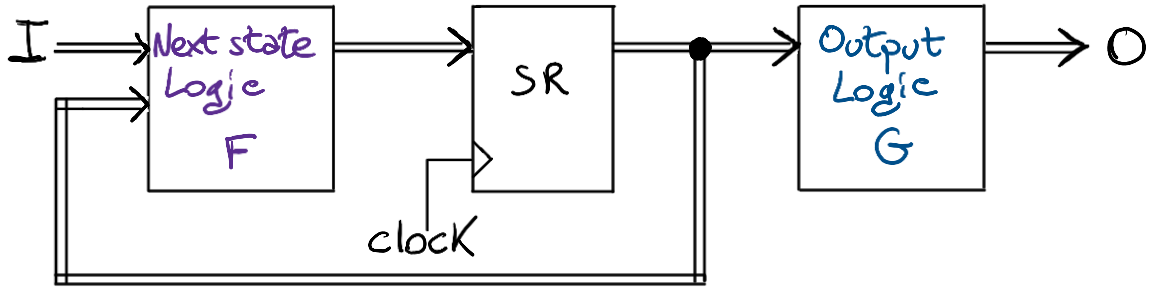
Finite State Machine (FSM)

- Outline of an FSM



Finite State Machine (FSM)

- Outline of an FSM



  - Next-state = F(input , current state)
  - Output: O = G(current state)
    - **Moore** FSM
    - A Moore FSM is an FSM in which the **output depends ONLY on the current state**

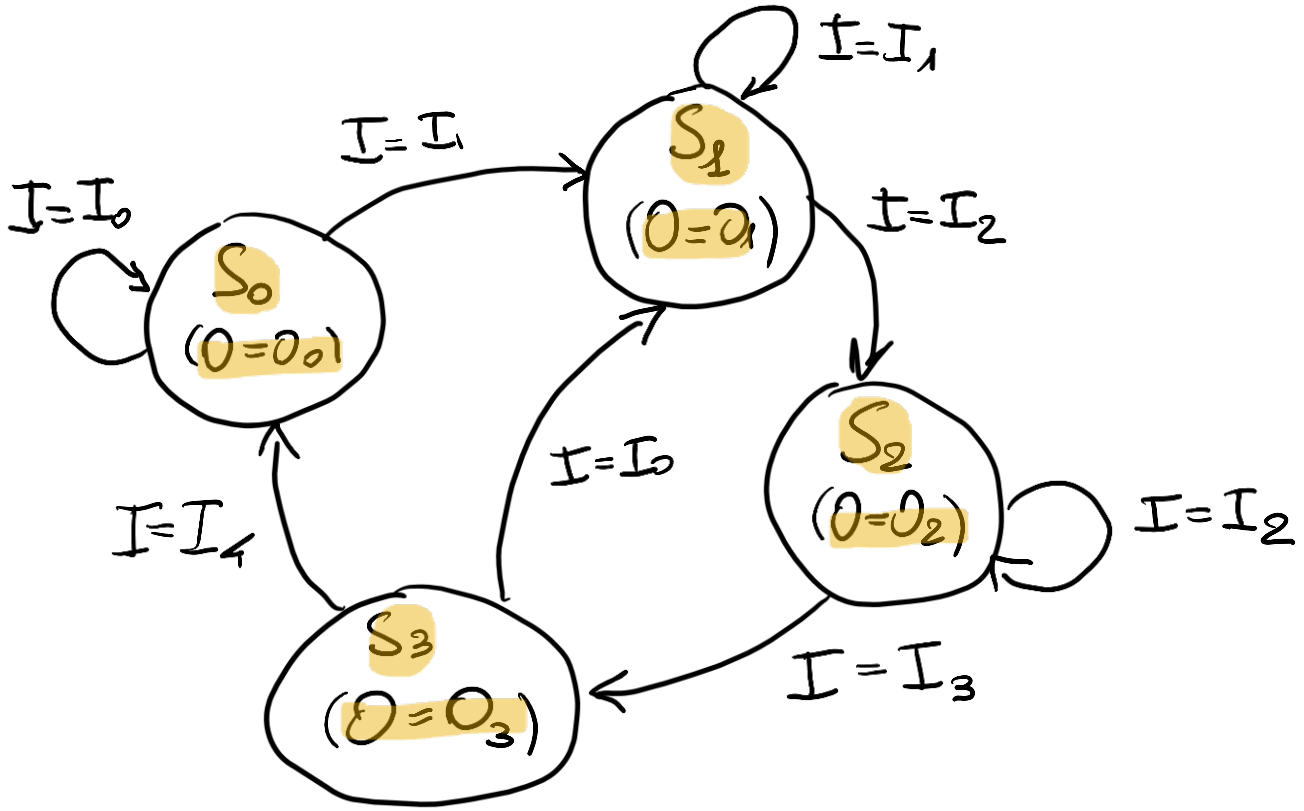
Finite State Machine (FSM)

- An FSM can be represented by
  - State diagram
  - State transition table
  - Output transition table

  - State and output transition tables can be merged in a unique table

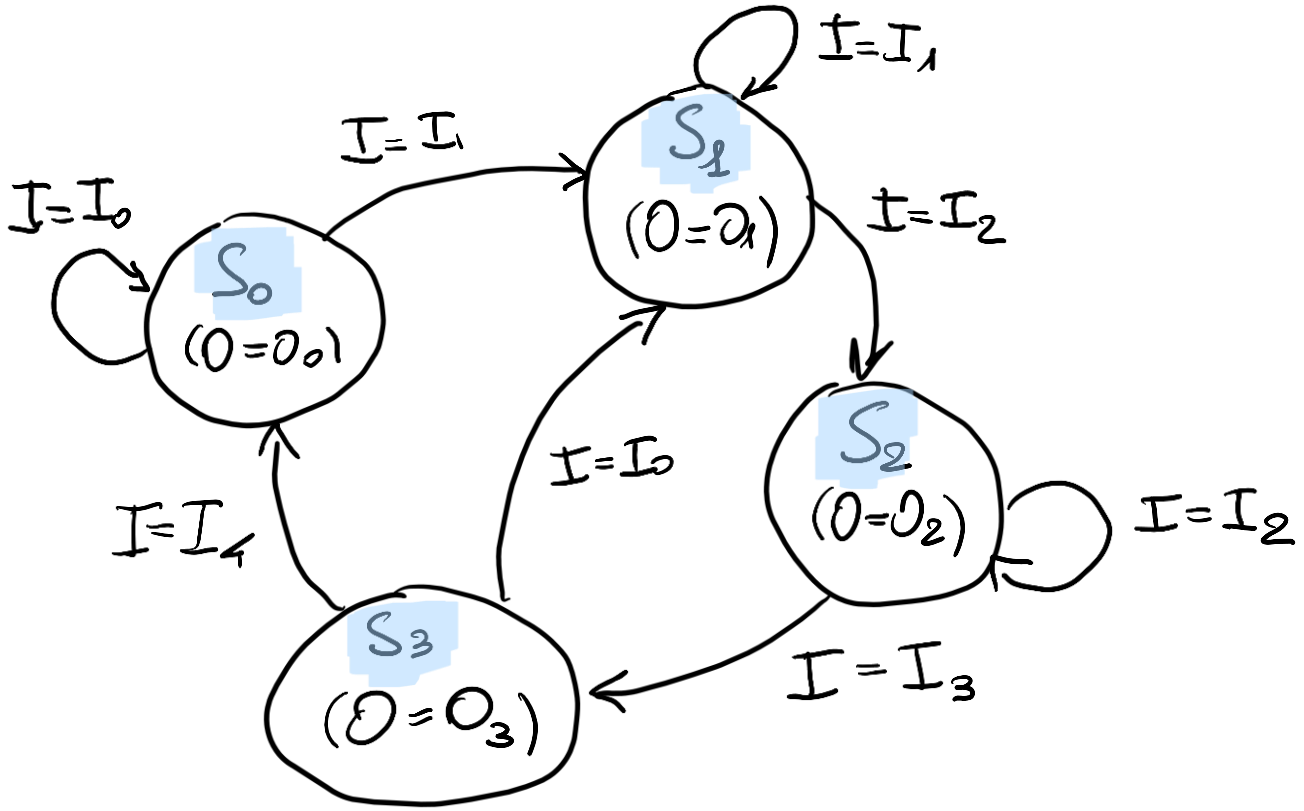
Finite State Machine (FSM)

- **State diagram** (Moore)



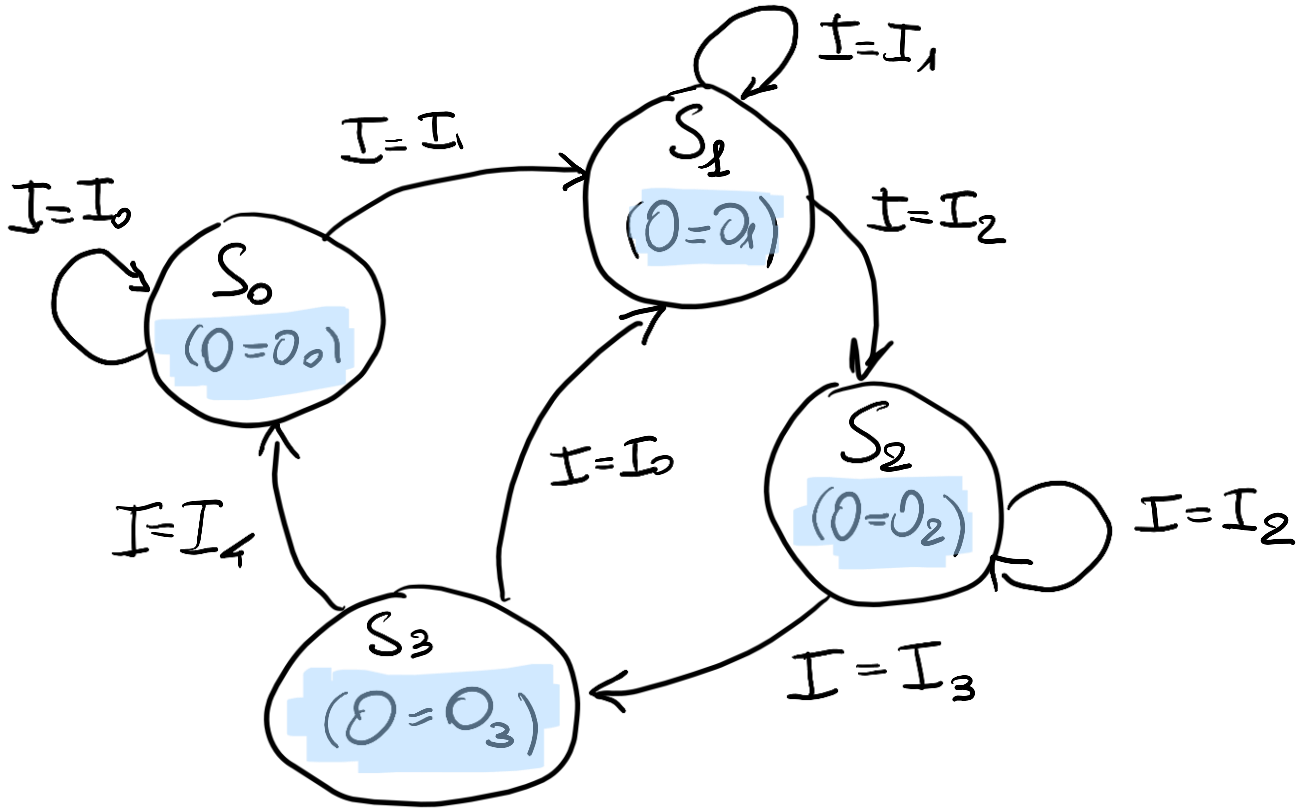
Finite State Machine (FSM)

- **State diagram (Moore)**
  - State



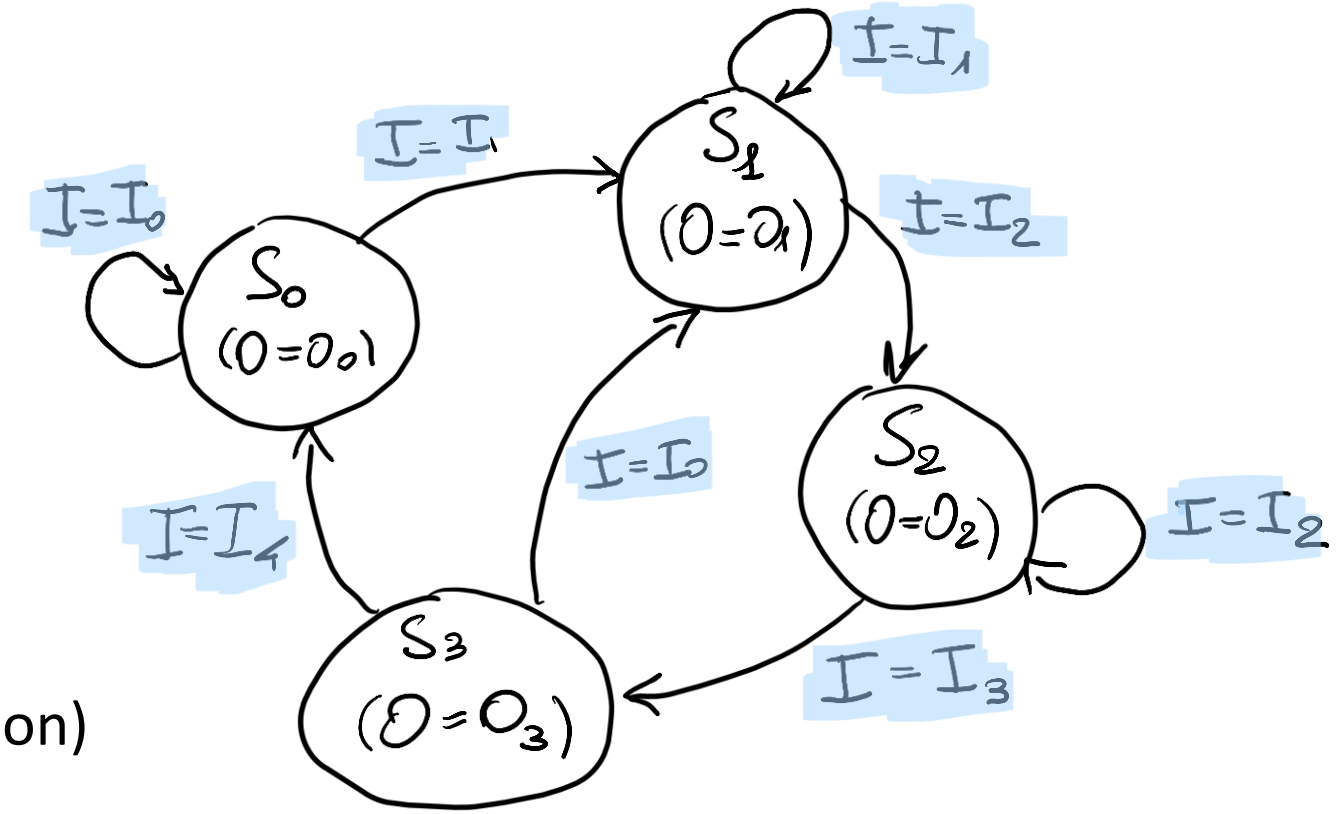
Finite State Machine (FSM)

- **State diagram (Moore)**
  - State
  - Output value (per state)



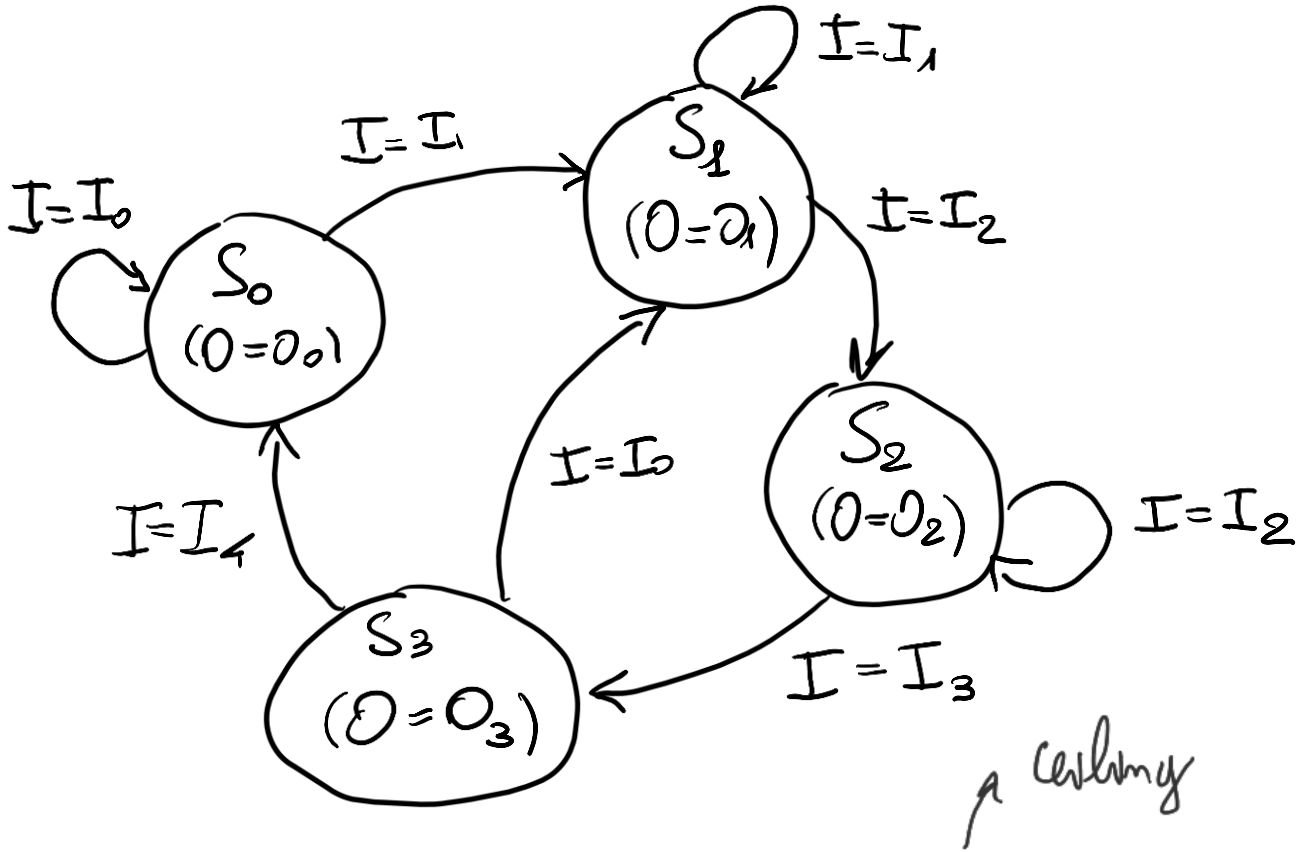
Finite State Machine (FSM)

- **State diagram (Moore)**
  - State
  - Output value (per state)
  - Input value (trigger for state transition)



Finite State Machine (FSM)

- **State diagram (Moore)**



- An FSM with **N** states requires a (status) register of $\boldsymbol{K = \lceil \log_2(N) \rceil}$ bits at least

Finite State Machine (FSM)

- At least $K = \lceil \log_2(N) \rceil$, because other coding style can be used
  - Example: One-hot encoding
    - The binary representation of each state has only one 1; all other bits are 0
    - N states → K = N bits
    - Higher resource consumption (more bits)
    - But higher robustness against disturbance

More bits change so disturbance can be bad (ex: 000 → 111)

One-hot encoding for a 4-state FSM

State	SR_0	SR_1	SR_2	SR_3
S0	1	0	0	0
S1	0	1	0	0
S2	0	0	1	0
S3	0	0	0	1

SR_i = ith bit of SR (Status Register)

Finite State Machine (FSM)

- **State transition table (Moore)**
  - Input: I (2-bit)
  - S = State (current)
    - Number of states = 4
  - S* = Next state

	I			
S	**00**	**01**	**10**	**11**
S0	S0	S0	S1	S1
S1	S2	S1	S2	S2
S2	S0	S3	S0	S3
S3	S1	S3	S0	S1
	S*			

Finite State Machine (FSM)

- **Output transition table (Moore)**
  - Output: O (3-bit)
  - S = State (current)
    - Number of states = 4

S	O
S0	011
S1	100
S2	110
S3	010

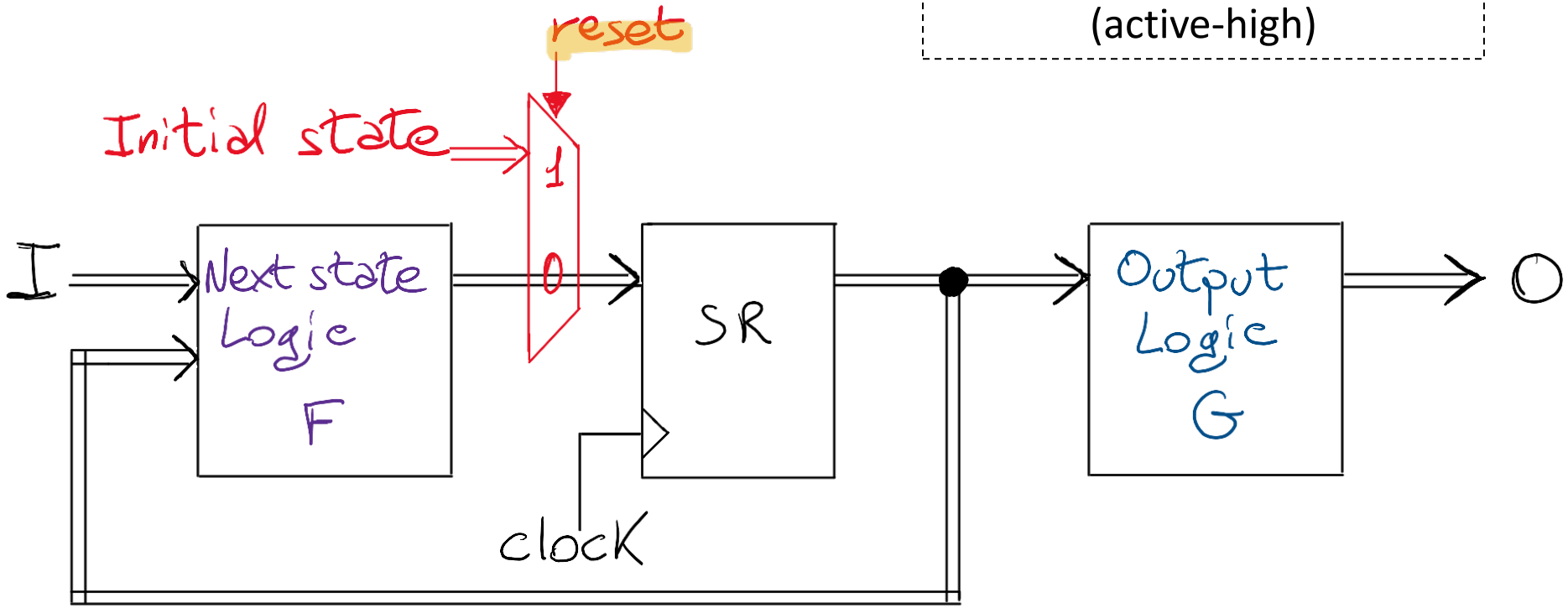
Finite State Machine (FSM)

- How to set the initial state of FSM?
  - At the power-on of the device/system, the logic value of elements is unpredictable
  - Unless it is addressed with specific techniques
  - Most diffused approach: Power-on Reset (PoR)
    - The main goal of **reset** is to bring a digital system into a **known state**!!!
    - Reset signal (and logic) must be added to the FSM circuit

Finite State Machine (FSM)

- Outline of an FSM

This is a synchronous reset (active-high)

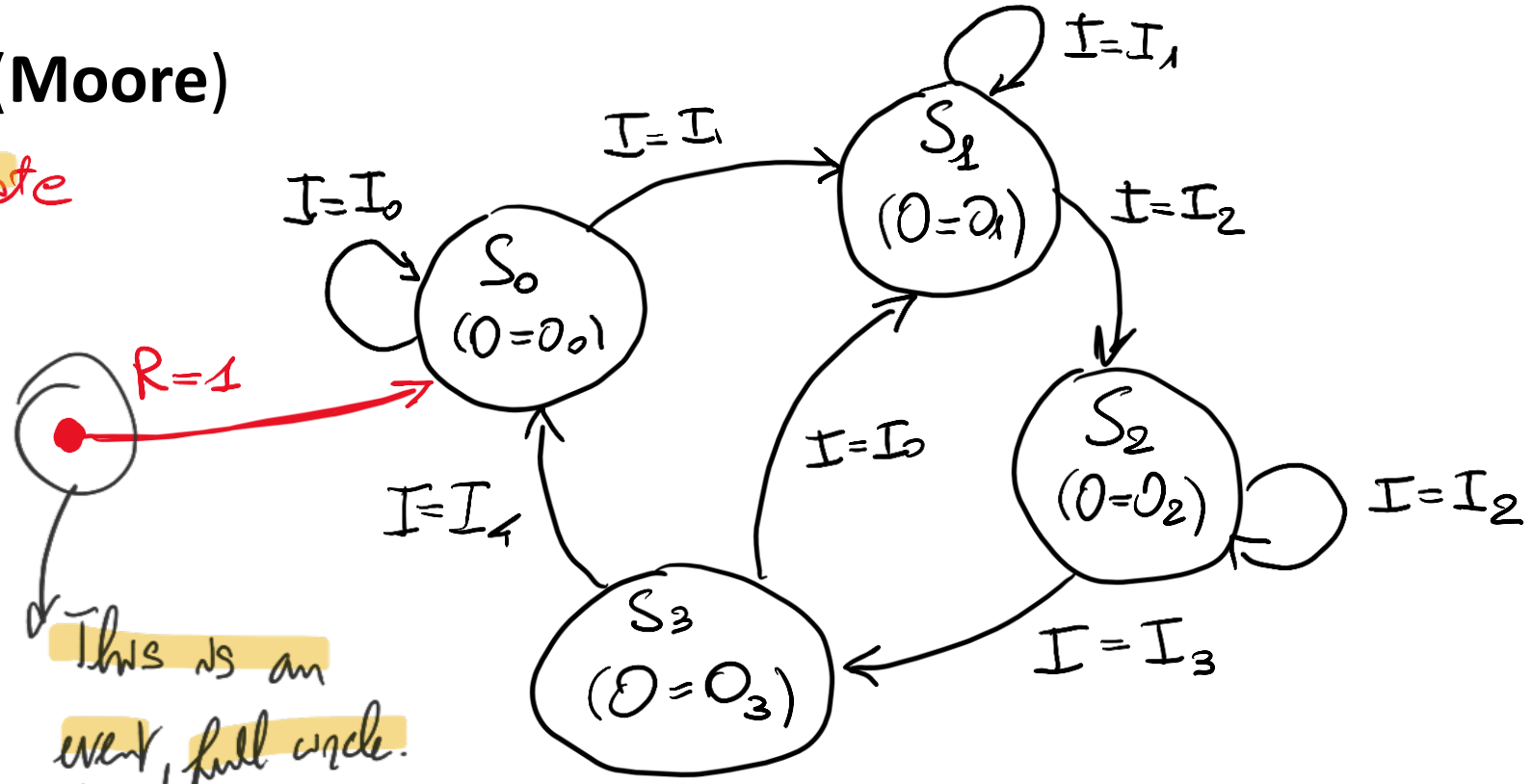


Finite State Machine (FSM)

- **State diagram (Moore)**

S_0 = Initial state

R = reset



This is an event, full circle.
In this case event = power on. I not specified because state doesn't depend on I

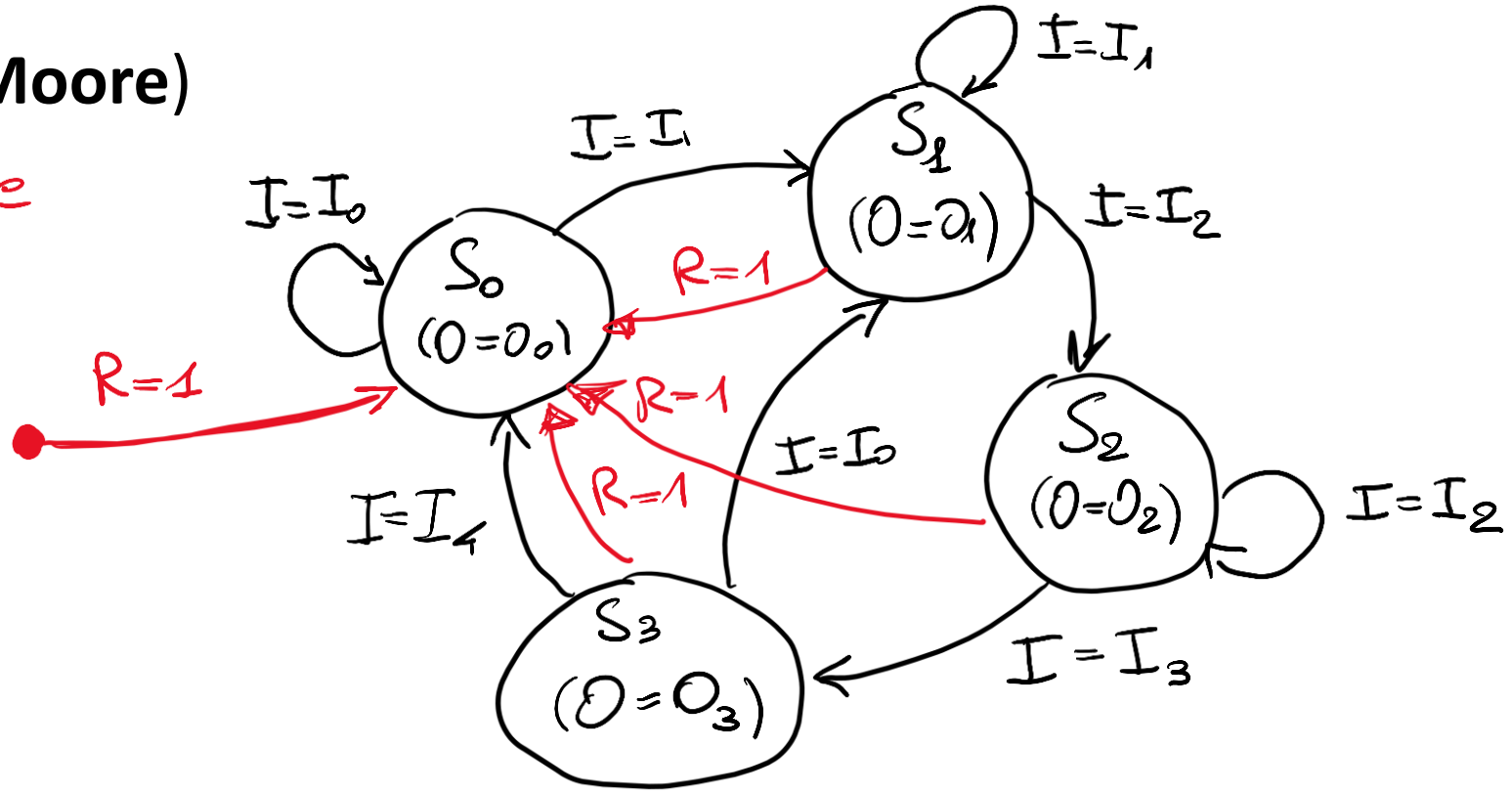
We should add a loop in S_0 for R because it is a synchronous input

Finite State Machine (FSM)

- **State diagram (Moore)**

S_0 = Initial state

R = reset

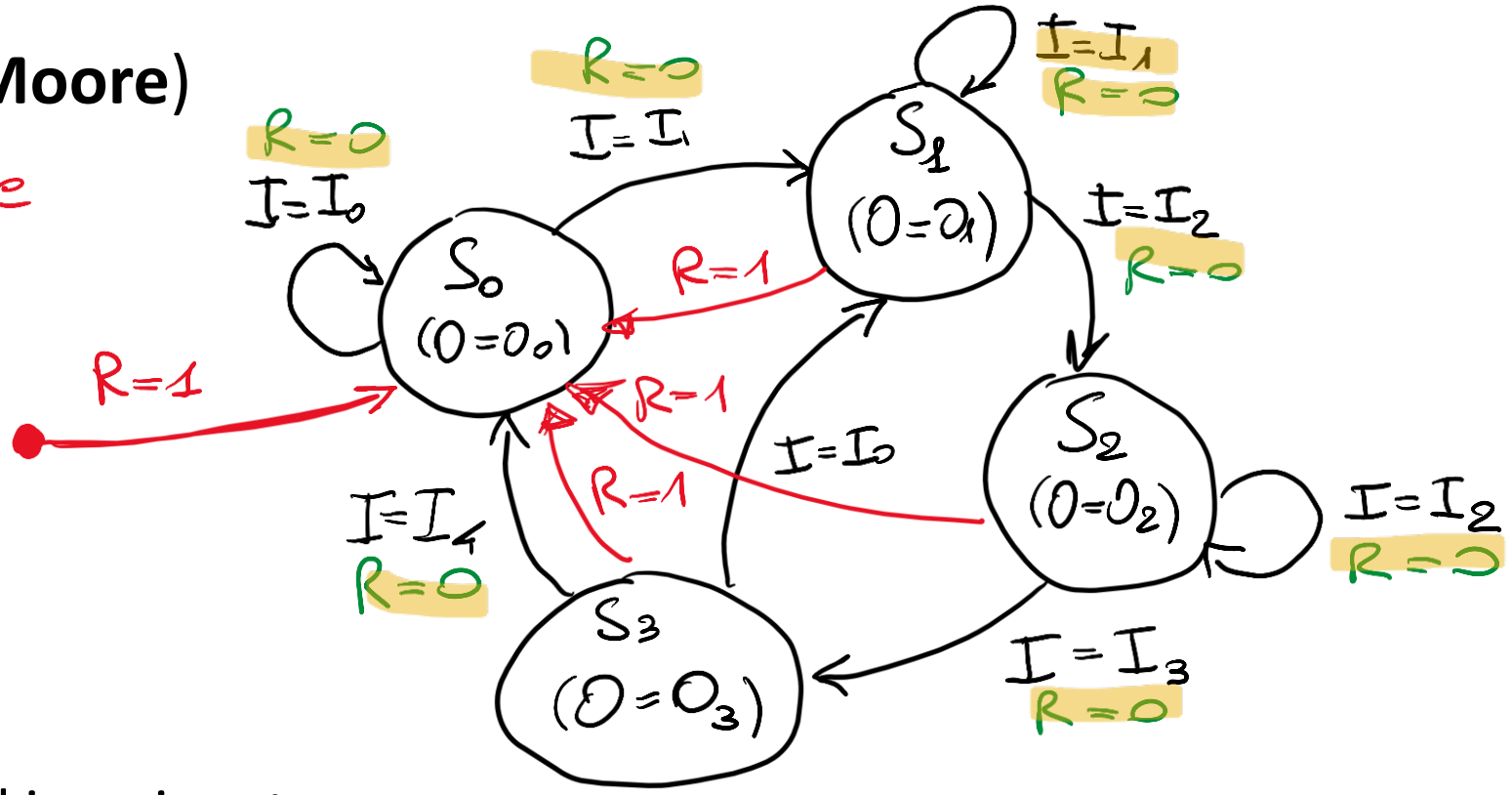


Finite State Machine (FSM)

- **State diagram (Moore)**

S_0 = Initial state

R = reset



  - Also reset signal is an input

Finite State Machine (FSM)

- **State transition table with reset (Moore)**
  - Inputs:
    - I (2-bit)
    - R = Reset (1-bit)
  - S = State (current)
    - Number of states = 4
  - S0 = Initial state (upon reset)
  - S* = Next state

	R,I				
S	**000**	**001**	**010**	**011**	**1xx**
S0	S0	S0	S1	S1	S0
S1	S2	S1	S2	S2	S0
S2	S0	S3	S0	S3	S0
S3	S1	S3	S0	S1	S0
	S*				

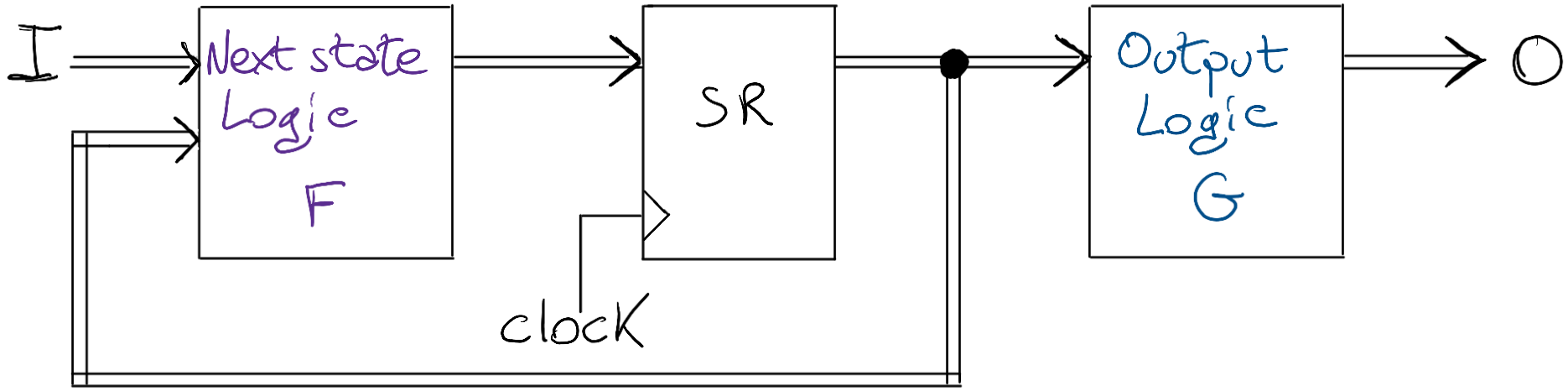
Finite State Machine (FSM)

- The Moore architecture is not the only possible architecture for FSM

- Two other architectures exist…
  - … or better, another main architecture
  - and an its modified version to overcome a problem that this second architecture may present

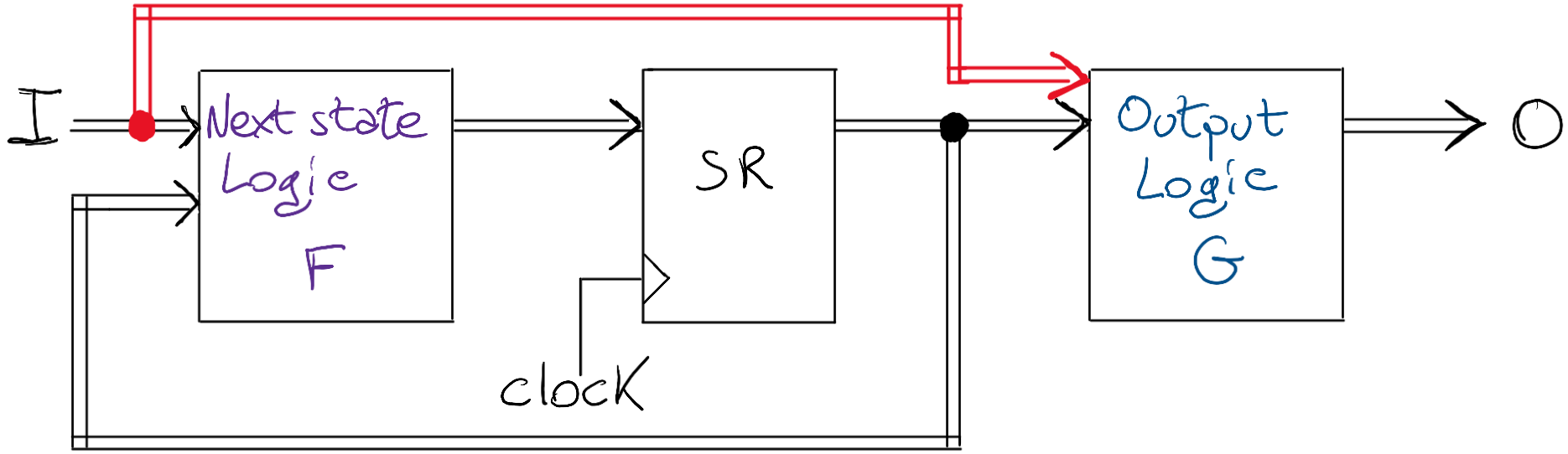
Finite State Machine (FSM)

- Recall: Moore FSM



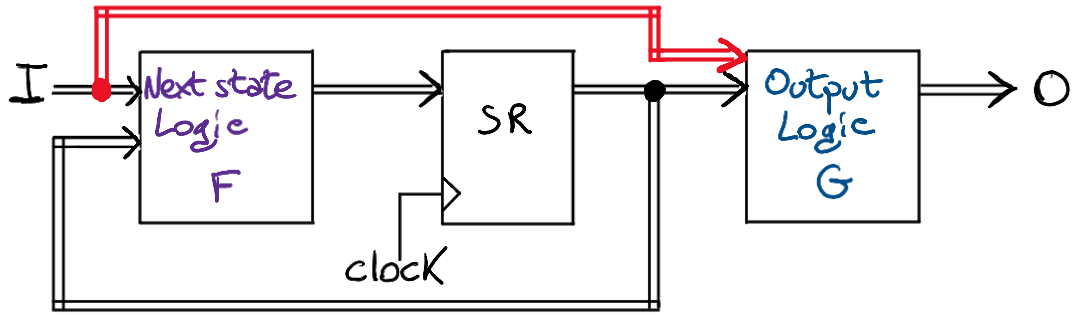
Finite State Machine (FSM)

- Mealy FSM



Finite State Machine (FSM)

- Mealy FSM



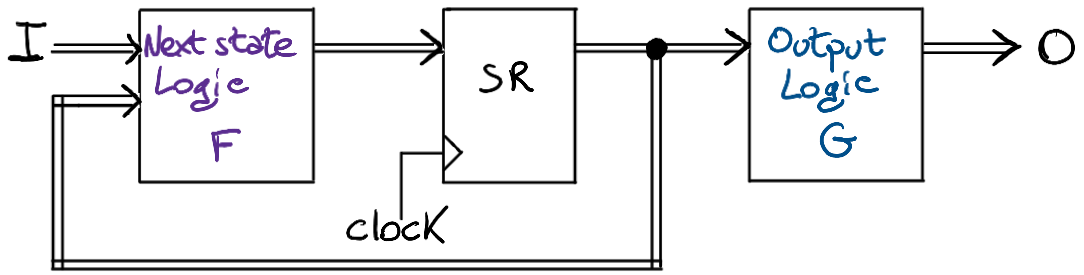
  - Next-state = F(input , current state)
    - Such as Moore FSM

  - Output: O = G(input, current state)
    - A **Mealy** FSM is an FSM in which the **output depends on the current state AND the input**

Finite State Machine (FSM)

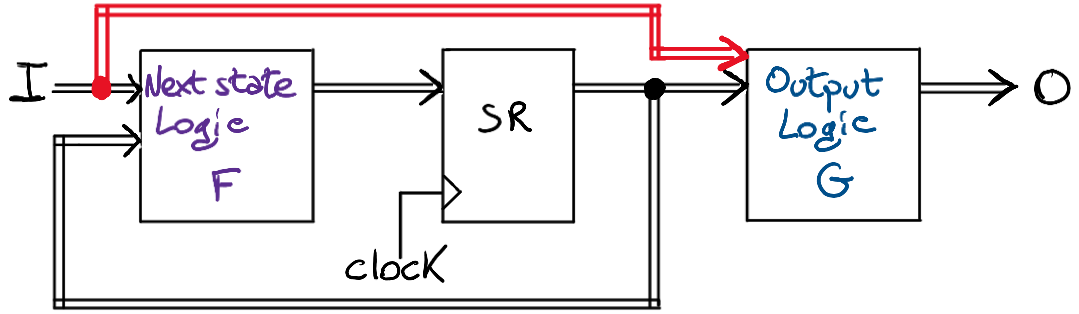
Moore FSM

- Next-state = F(input , current state)
- Output: O = G(current state)



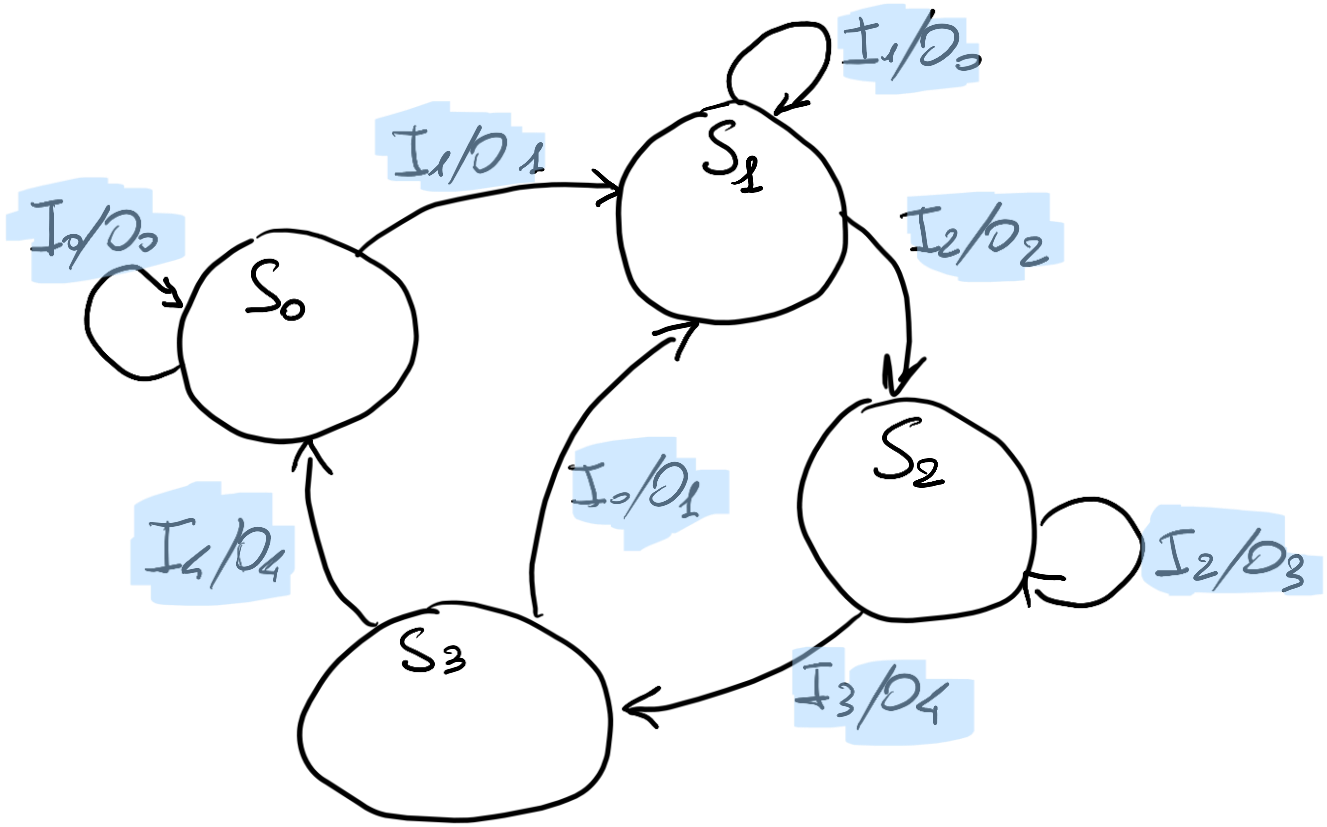
Mealy FSM

- Next-state = F(input , current state)
- Output: O = G(input, current state)



Finite State Machine (FSM)

- **State diagram (Mealy)**



  - Output is no more associated to the state, but is on the branch (with input)
    - I/O

Finite State Machine (FSM)

- State transition table is like the one in Moore's FSM
- Instead, output transition table changes with respect to the one in Moore's FSM, because output depends also on input(s)

Finite State Machine (FSM)

- **Output transition table (Mealy)**
  - Input: I (2-bit)
  - S = State (current)
    - Number of states = 4
  - Output: O (3-bit)

	I			
S	**00**	**01**	**10**	**11**
S0	001	010	010	111
S1	100	000	011	101
S2	111	110	101	100
S3	000	010	110	101

O

Finite State Machine (FSM)

- **State/output transition table (Mealy)**
  - Input: I (2-bit)
  - S = State (current)
    - Number of states = 4
  - S* = Next state
  - Output: O (3-bit)

	I			
S	**00**	**01**	**10**	**11**
S0	S0, 001	S0, 010	S1, 010	S1, 111
S1	S2, 100	S1, 000	S2, 011	S2, 101
S2	S0, 111	S3, 110	S0, 101	S3, 100
S3	S1, 000	S3, 010	S0, 110	S1, 101
	S*,O			

Finite State Machine (FSM)

- Mealy FSM
  - Since the output depends on both the current state (i.e. the State Register) and the input(s), the output is asynchronous
    - It does not change (only) on the clock edge as it happens in a Moore FSM
      - In a Moore FSM output is synchronous (with the clock)
    - The output may present spurious transitions, creating problems in some cases

↳ You choose if you need more robustness. If so, you use the register

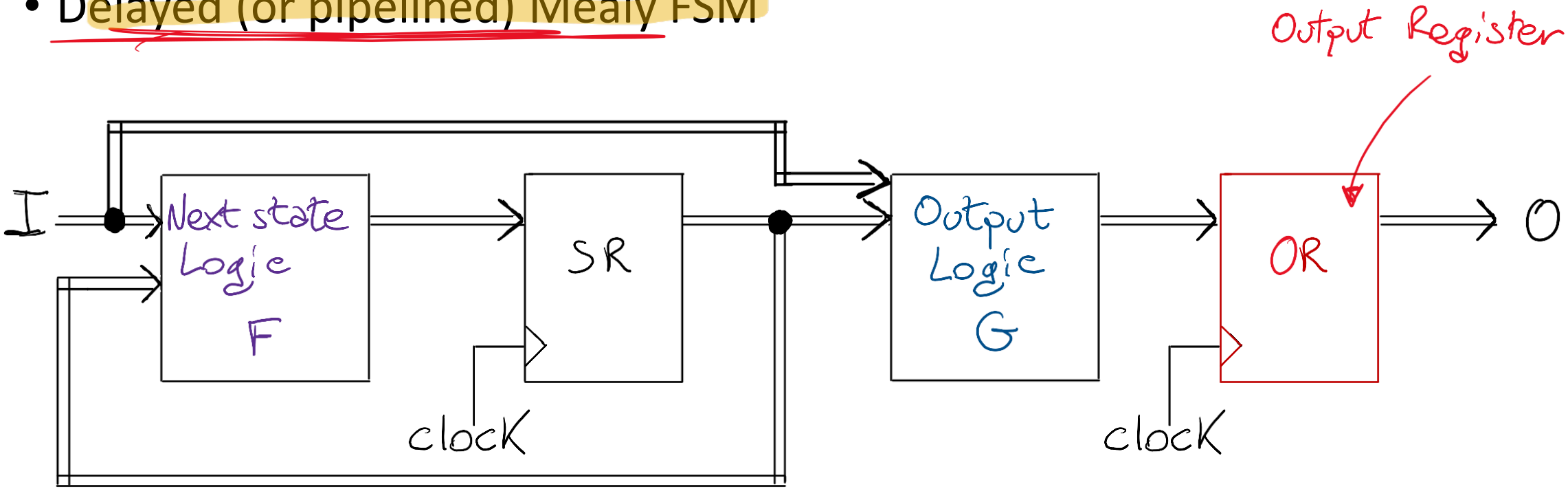
Finite State Machine (FSM)

- Mealy FSM
  - Since the output depends on both the current state (i.e. the State Register) and the input(s), the output is asynchronous
    - It does not change (only) on the clock edge as it happens in a Moore FSM
      - In a Moore FSM output is synchronous (with the clock)
    - The output may present spurious transitions, creating problems in some cases

  - If so, this problem can be solved by 'registering' the output ?
    - I.e., placing a register after the Output logic (G) block
    - In this way the output is again synchronous (with the clock)

Finite State Machine (FSM)

- Delayed (or pipelined) Mealy FSM



Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim
  - Input(s)
    - en (1-bit)
    - rst_n (asynchronous active-low reset)
  - Output(s)
    - out_code (2-bit)
  - States
    - 4 states: S0, S1, S2, S3
    - 2-bit state register

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim
  - Moore FSM
    - S = current state
    - S* = next state

State transition table

	rst_n, en		
S	**0x**	**10**	**11**
S0	S0	S0	S1
S1	S0	S1	S2
S2	S0	S2	S3
S3	S0	S3	S0
	S*		

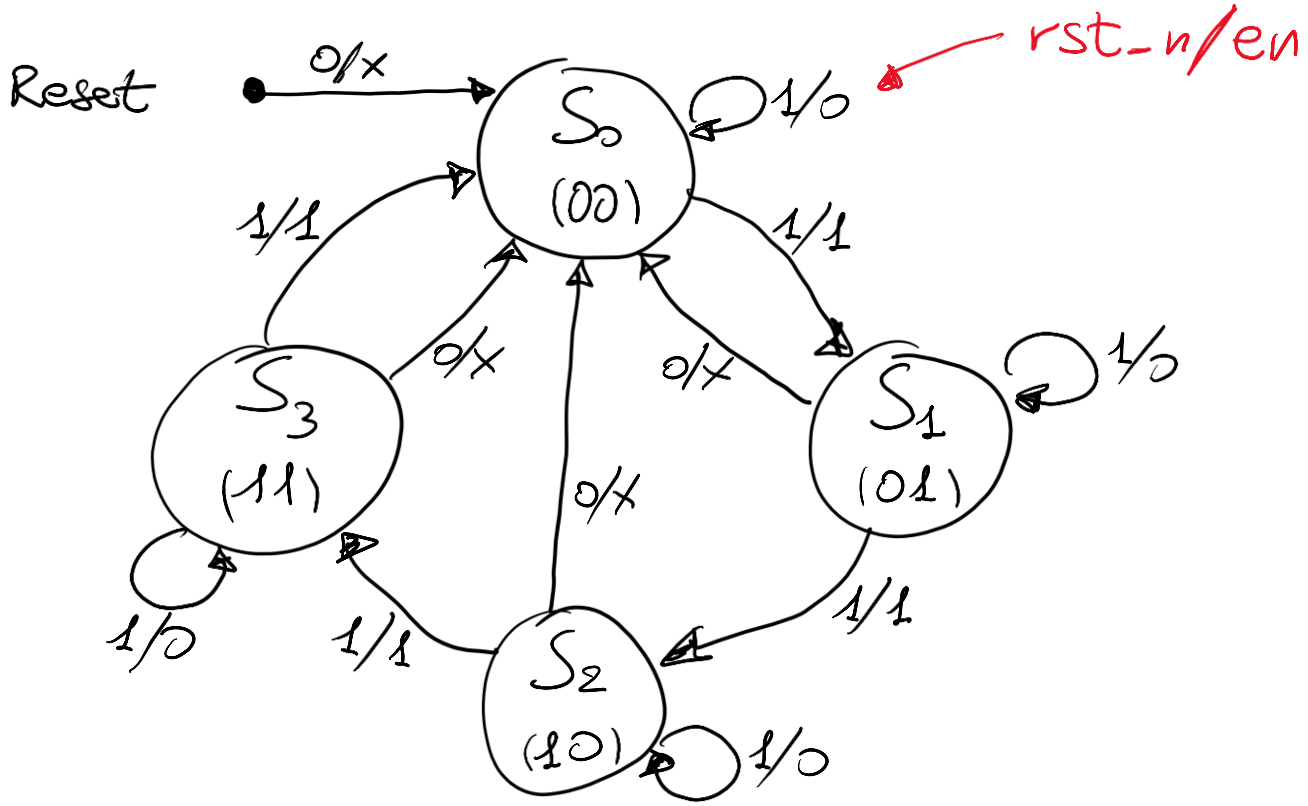
Output transition table

S	
S0	00
S1	01
S2	10
S3	11
	out_code

Exercise with SystemVerilog

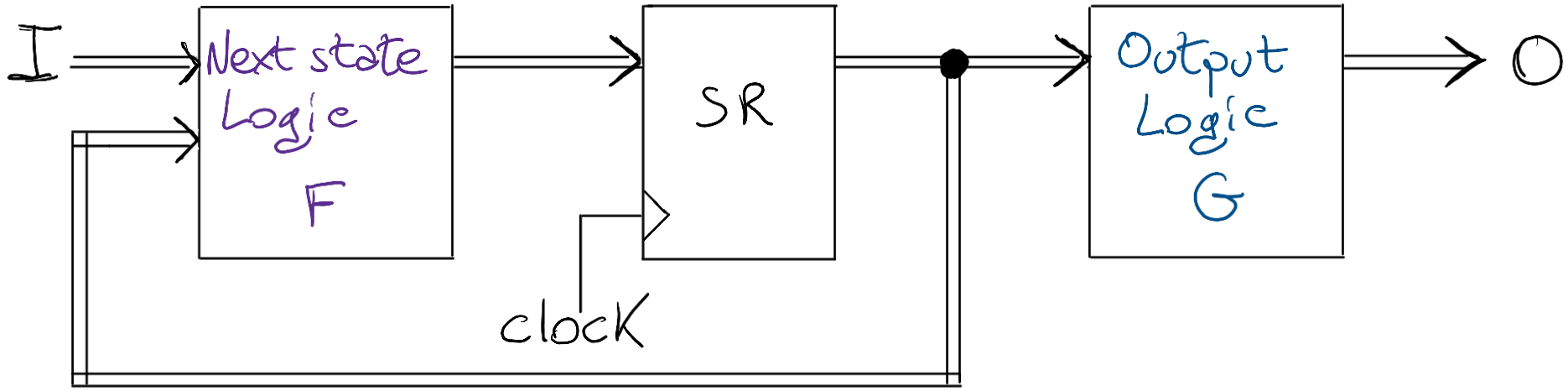
- Implementation of an FSM and simulation with Modelsim
  - Moore FSM

State diagram



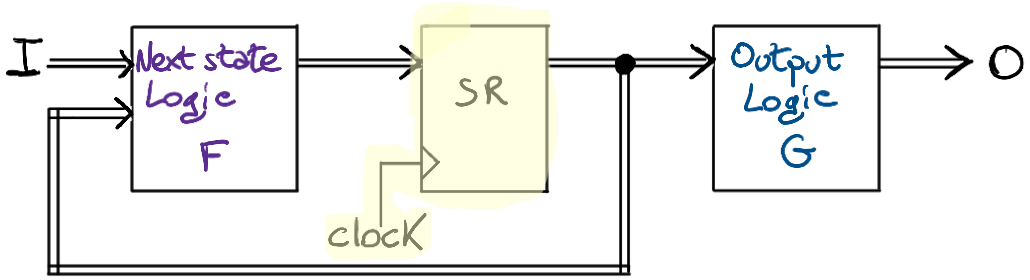
Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

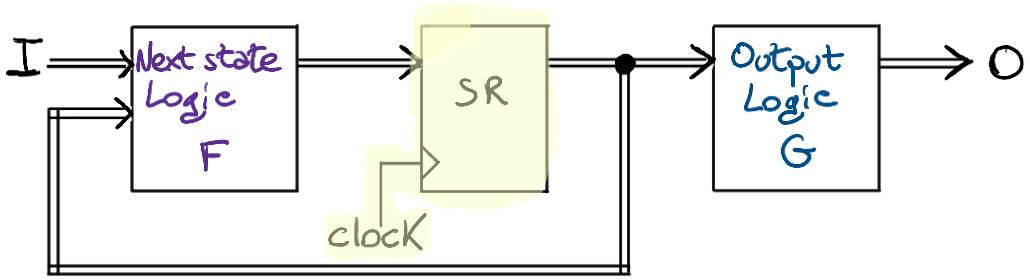


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment

```
/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

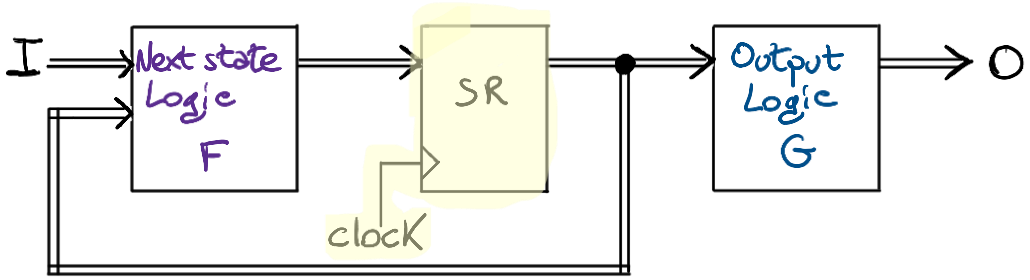


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment

```
/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

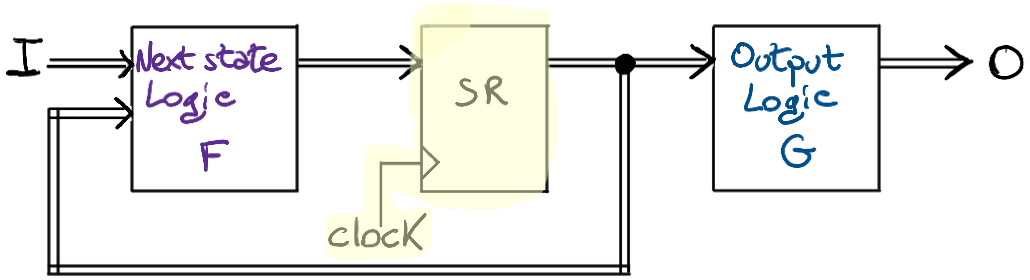


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment

```
/* Logic signals */
reg [1:0] current_state;

/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

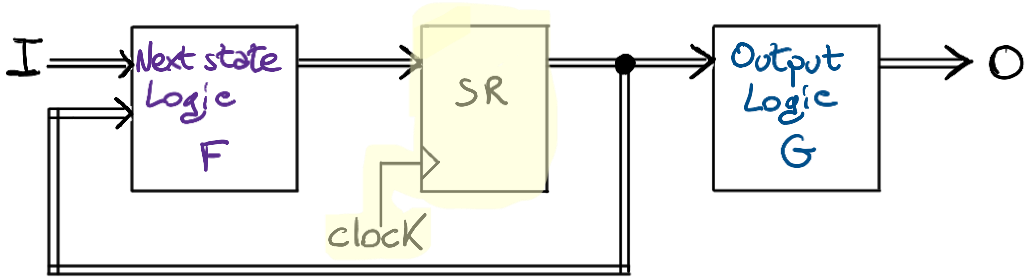


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment

```
/* Logic signals */
reg [1:0] current_state;

/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

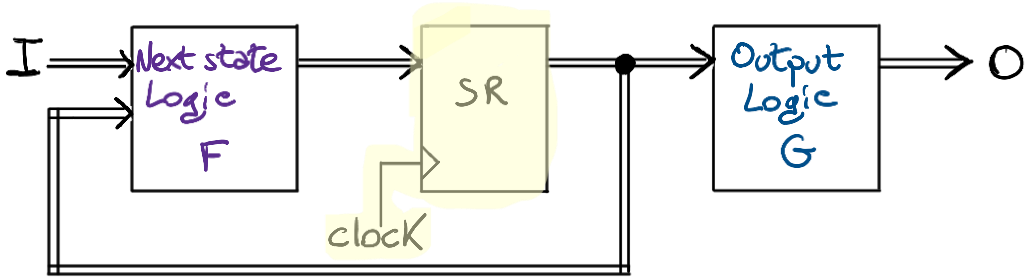


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment
- Signal internal to the module
  - Must be declared before its usage (in the module body)

```
/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

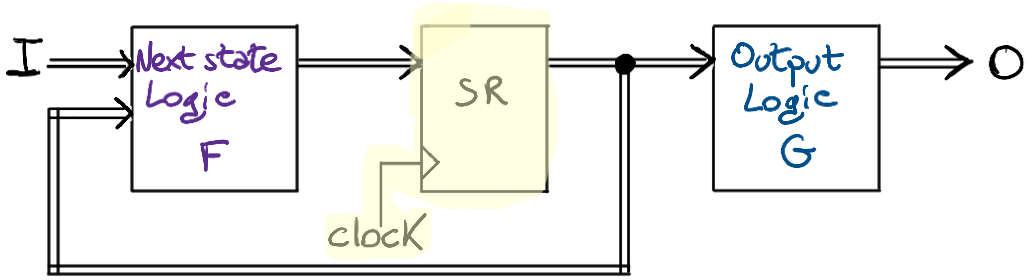


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment
- Signal internal to the module
  - Must be declared before its usage (in the module body)

```
module fsm (
  /* ports list */
);


/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;


endmodule
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

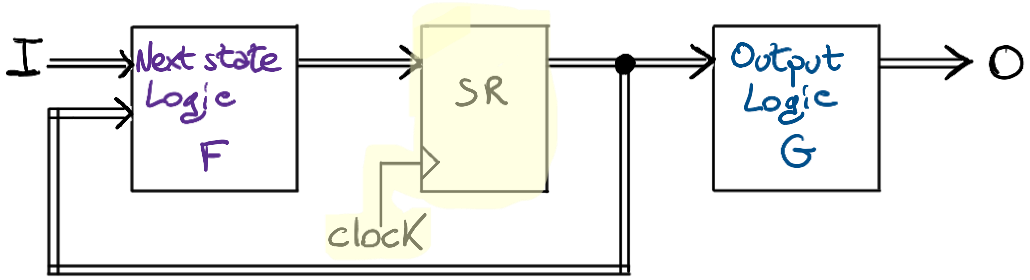


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment
- Signal internal to the module
  - Must be declared before its usage (in the module body)

```
module fsm (
  /* ports list */
);


/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;


endmodule
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

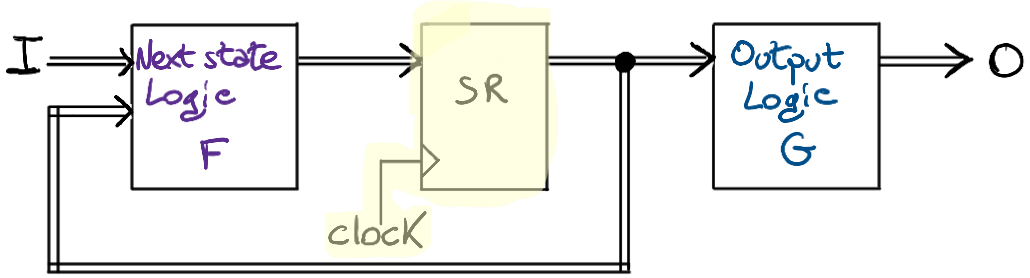


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment
- Signal internal to the module
  - Must be declared before its usage (in the module body)

```
module fsm (
  /* ports list */
);


/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;


endmodule
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

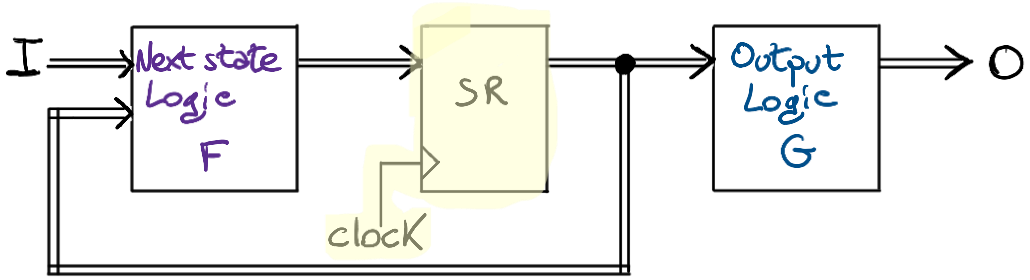


- State register = sequential logic
  1. always_ff block + trigger(s) edge
  2. reg signal
  3. non-blocking assignment
- Signal internal to the module
  - Must be declared before its usage (in the module body)

```
module fsm (
  /* ports list */
);

/* Logic signals */
reg [1:0] current_state;

/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;

endmodule
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

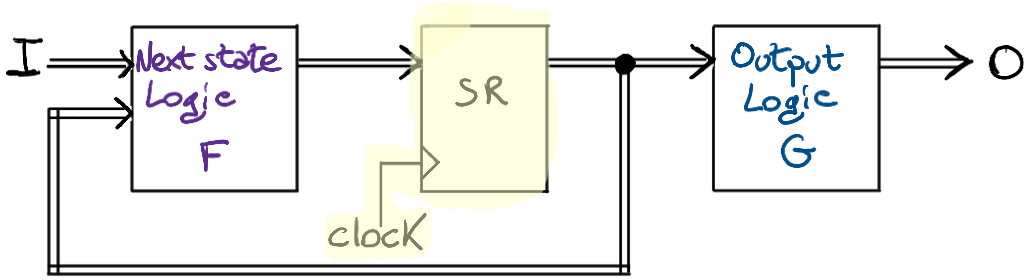


- States can be coded using labels

```
/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

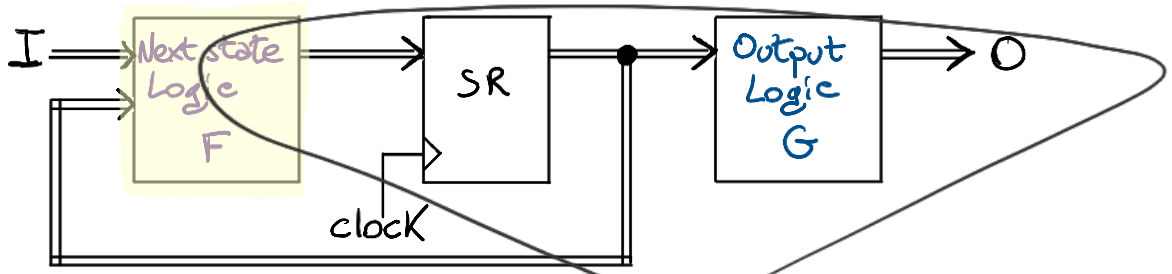


- States can be coded using labels
  - Local parameters
    - Syntax: **localparam <name> = <constant value>;**
  - Again: declare before usage!!!

```
/* State labels */
localparam S0 = 2'b00;
localparam S1 = 2'b01;
localparam S2 = 2'b10;
localparam S3 = 2'b11;

/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

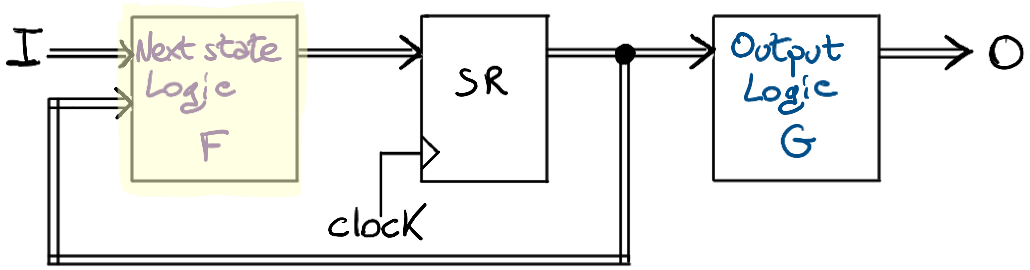


- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F

```
/* State labels */
localparam S0 = 2'b00;
localparam S1 = 2'b01;
localparam S2 = 2'b10;
localparam S3 = 2'b11;

/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



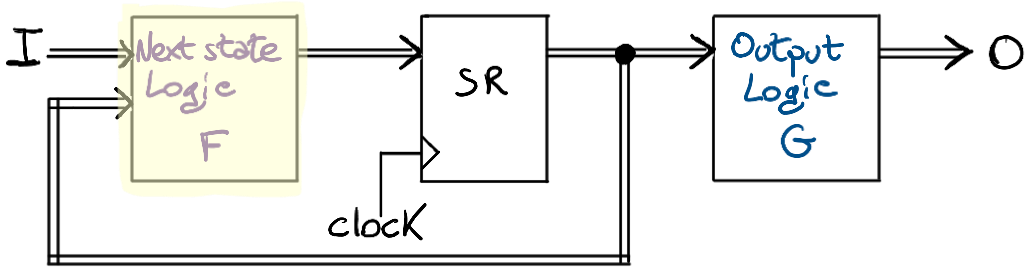
- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F
- F (next-state logic) is combinational
  - Case statement can be very useful, so …
    - always_comb
    - blocking assignment on **reg** signal

always comb = blocking assignment
↓
That works on reg though

```
/* State labels */
localparam S0 = 2'b00;
localparam S1 = 2'b01;
localparam S2 = 2'b10;
localparam S3 = 2'b11;

/* Logic signals */
reg [1:0] current_state;


/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

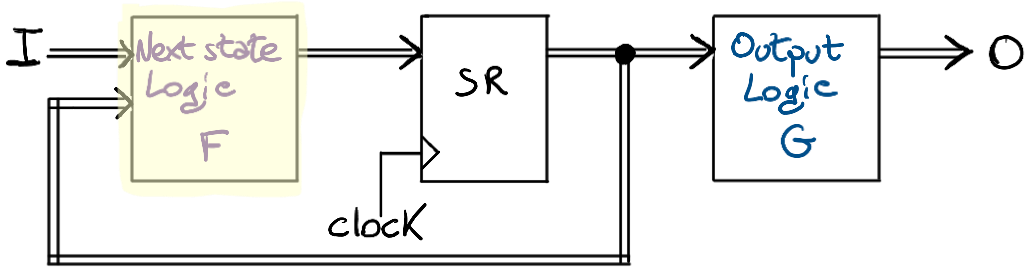


- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F
- F (next-state logic) is combinational
  - Case statement can be very useful, so …
    - always_comb
    - blocking assignment on **reg** signal

```
/* State labels */
localparam S0 = 2'b00;
localparam S1 = 2'b01;
localparam S2 = 2'b10;
localparam S3 = 2'b11;

/* Logic signals */
reg [1:0] current_state;
reg [1:0] next_state;

/* State Register (SR) */
always_ff @ (posedge clk or negedge rst_n)
  if(!rst_n)
    current_state <= S0;
  else
    current_state <= next_state;
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

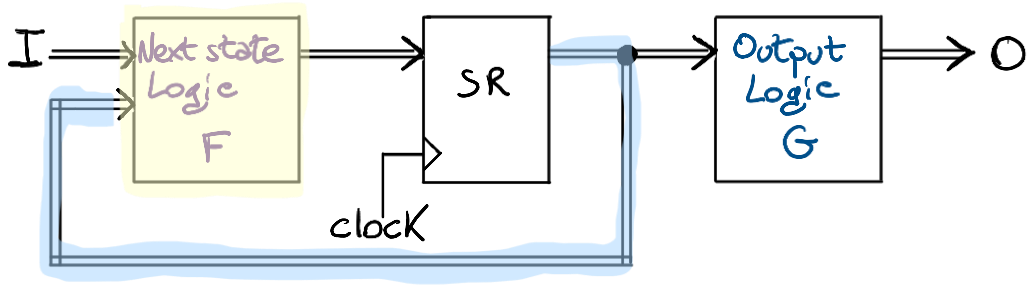


- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F
- F (next-state logic) is combinational
  - **Case** statement can be very useful, so …
    - **always_comb**
    - **blocking assignment** on reg signal

```
/* Next-state logic (F) */
always_comb

  case(current_state)
    S0: next_state = en ? S1 : S0;
    S1: next_state = en ? S2 : S1;
    S2: next_state = en ? S3 : S2;
    S3: next_state = en ? S0 : S3;
  endcase
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



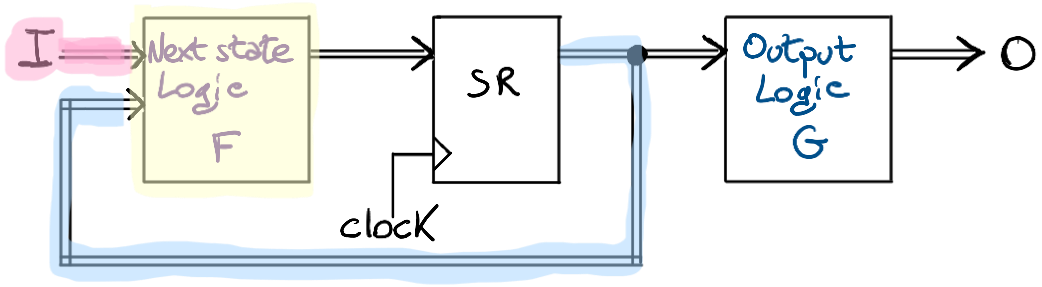
- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F
- F (next-state logic) is combinational
  - Case statement can be very useful, so …
    - always_comb
    - blocking assignment on reg signal

dependence on the current state

```
/* Next-state logic (F) */
always_comb

  case(current_state)
    S0: next_state = en ? S1 : S0;
    S1: next_state = en ? S2 : S1;
    S2: next_state = en ? S3 : S2;
    S3: next_state = en ? S0 : S3;
  endcase
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



- The always_ff block includes the dependence on the asynchronous active-low reset for the next state
  - Such signal can be omitted in the description of F
- F (next-state logic) is combinational
  - Case statement can be very useful, so …
    - always_comb
    - blocking assignment on reg signal

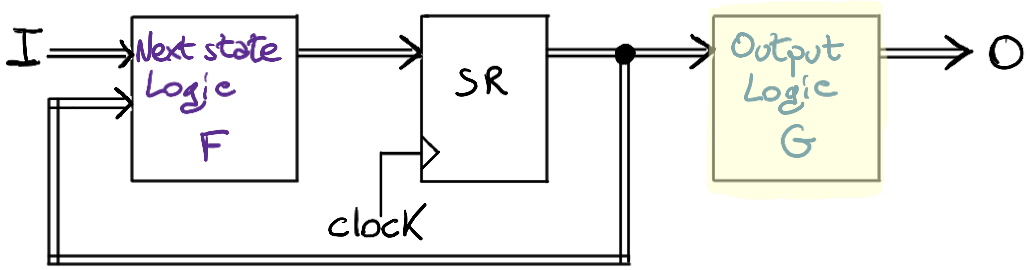
dependence on the current state

```
/* Next-state logic (F) */
always_comb

  case(current_state)
    S0: next_state = en ? S1 : S0;
    S1: next_state = en ? S2 : S1;
    S2: next_state = en ? S3 : S2;
    S3: next_state = en ? S0 : S3;
  endcase
```

dependence on input(s)

Exercise with SystemVerilog

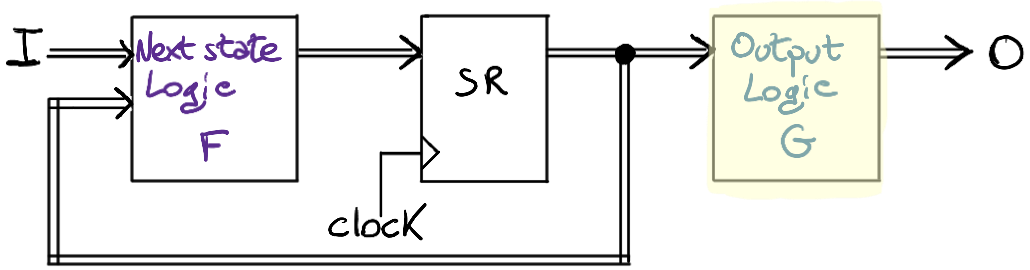
- Implementation of an FSM and simulation with Modelsim



  - G (output logic) can be implemented in the same way as F

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

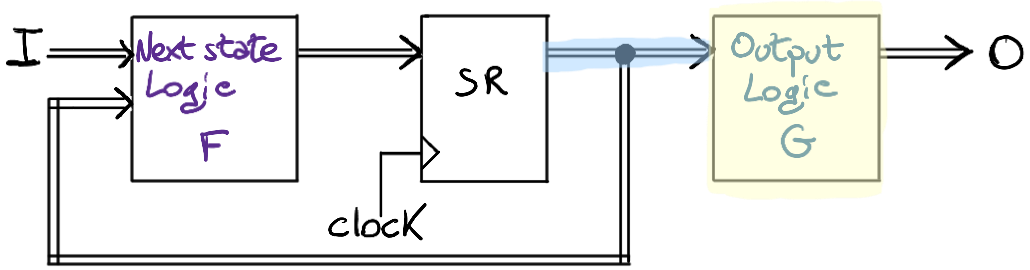


  - G (output logic) can be implemented in the same way as F

```
/* Output logic (G) */
always_comb

  case(current_state)
    S0: out_code = 2'd0;
    S1: out_code = 2'd1;
    S2: out_code = 2'd2;
    S3: out_code = 2'd3;
  endcase
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



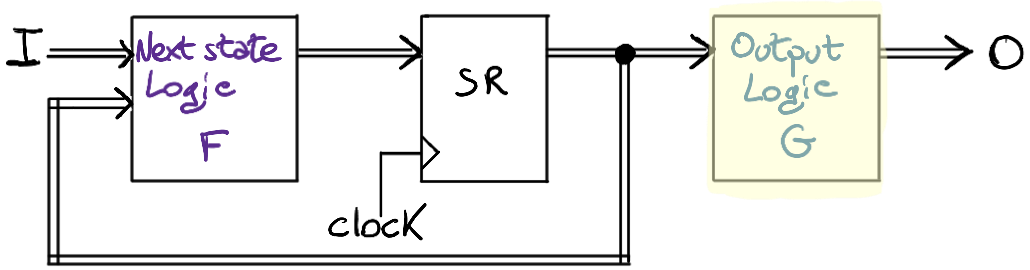
- G (output logic) can be implemented in the same way as F
  - It depends only on the current state

dependence on the current state

```
/* Output logic (G) */
always_comb

  case(current_state)
    S0: out_code = 2'd0;
    S1: out_code = 2'd1;
    S2: out_code = 2'd2;
    S3: out_code = 2'd3;
  endcase
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

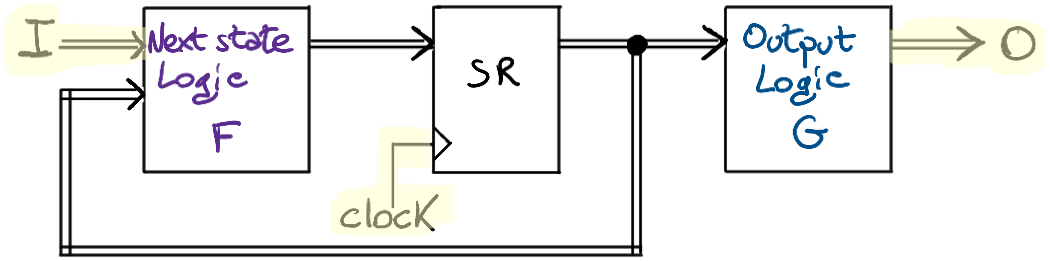



```
module fsm (
    input               clk
   ,input               rst_n
   ,input               en
   ,output reg [1:0]    out_code
);
```

- G (output logic) can be implemented in the same way as F
  - It depends only on the current state
  - Output (out_code) must be a **reg**

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim



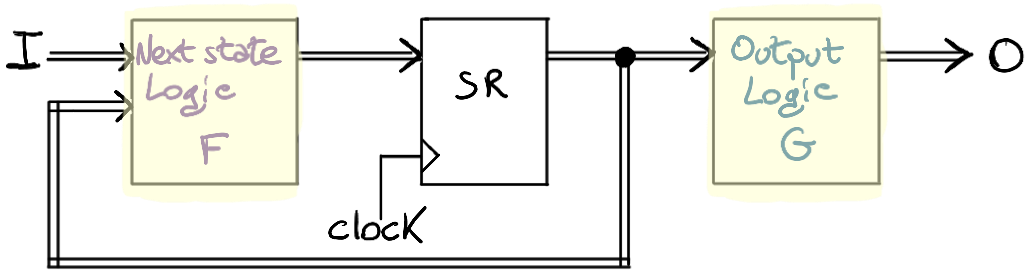
- List of ports (and module declaration)

```
module fsm (
    input               clk
   ,input               rst_n
   ,input               en
   ,output reg [1:0]    out_code
);
```

Every assignment is always REG <= non blocking assignment

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim

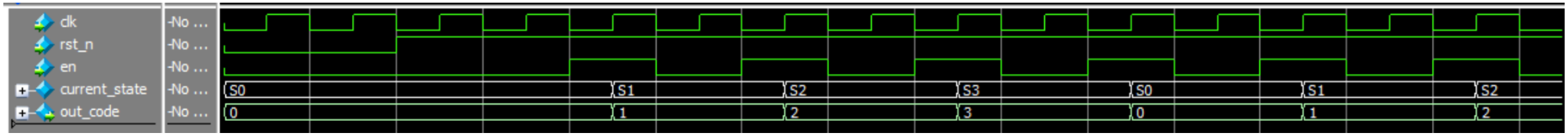


  - F and G can also be merged together

```
/* Next-state/Output logic (F/G) */
always_comb

  case(current_state)

    S0: begin
      out_code    = 2'd0;
      next_state  = en ? S1 : S0;
    end

    S1: begin
      out_code    = 2'd1;
      next_state  = en ? S2 : S1;
    end

    S2: begin
      out_code    = 2'd2;
      next_state  = en ? S3 : S2;
    end

    S3: begin
      out_code    = 2'd3;
      next_state  = en ? S0 : S3;
    end

  endcase
```

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.4
  - Try to simulate on your own

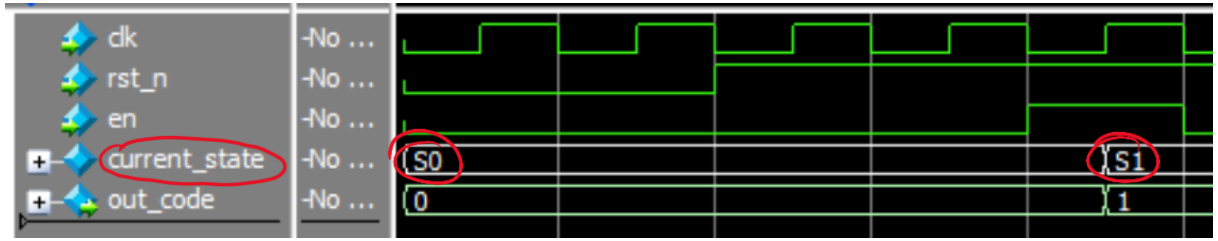


Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.4
  - Try to simulate on your own
  - However, I also included two .do files to automate the waveform and the simulation
    - wave.do and sim.do
    - Refer to README.txt file

Exercise with SystemVerilog

- Implementation of an FSM and simulation with Modelsim
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.4
  - Try to simulate on your own
  - However, I also included two .do files to automate the waveform and the simulation
    - wave.do and sim.do
    - Refer to README.txt file
    - Look at the wave.do file, if you are curious about how to display labels for the current state



Thank you for your attention

Luca Crocetti
(luca.crocetti@unipi.it)