Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 3 – State Machines

Winter 2023/2024

Stefan Mangard, www.iaik.tugraz.at

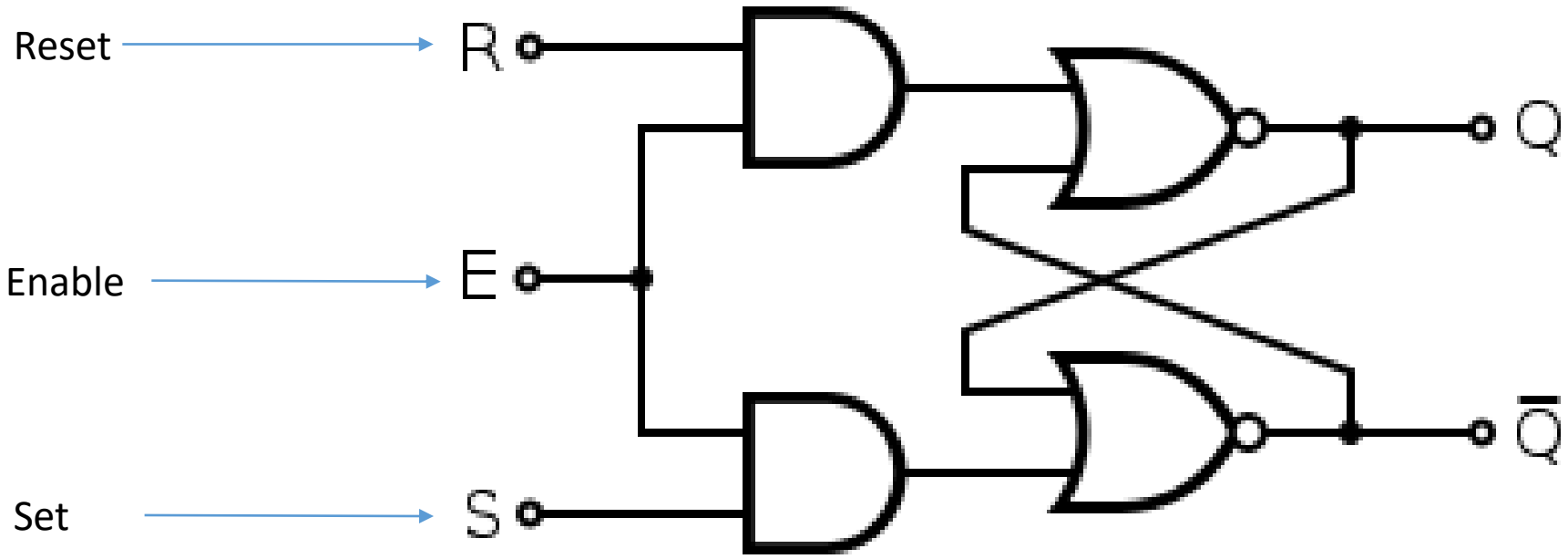
Sequential Circuits

(How to store data)

From Combinational Circuits to Sequential Circuits

- The circuits that we have discussed so far did not contain storage
- A change of an input has directly led to a change at the output
- We now build storage elements from logic gates
  - The basic idea to achieve storage is to create a feedback loop

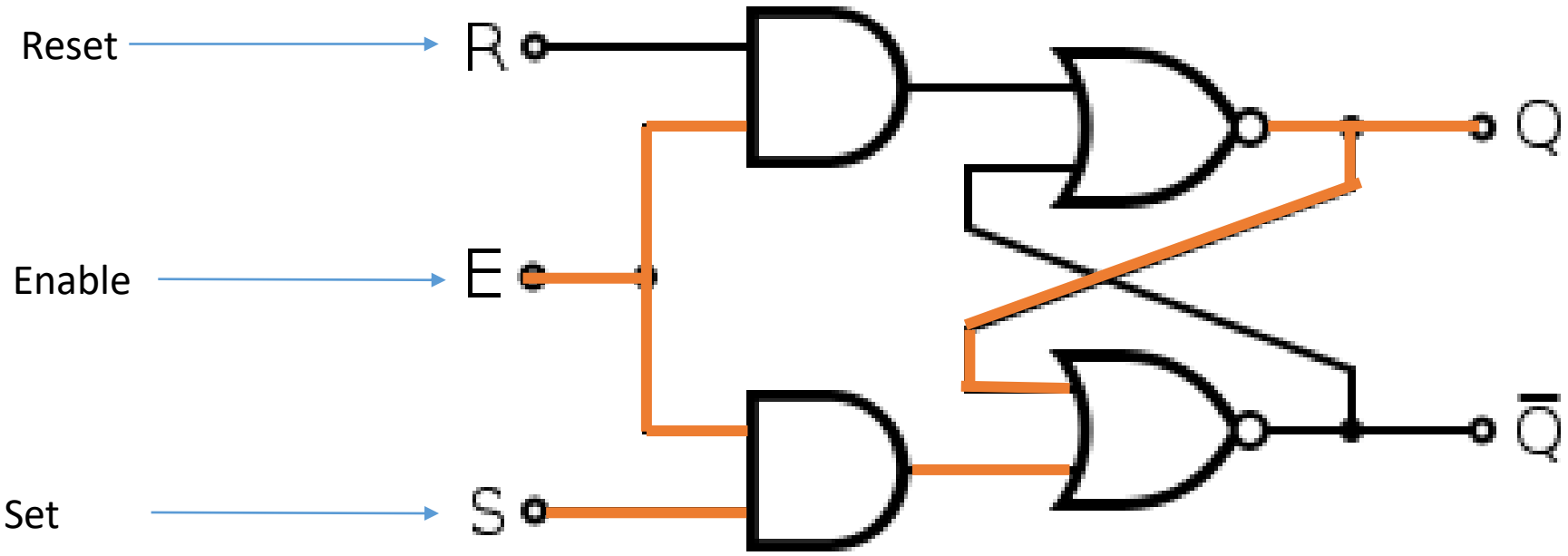
A Simple Set-Reset Latch (NOR Version)



We set the “set” input

Truth Table of NOR

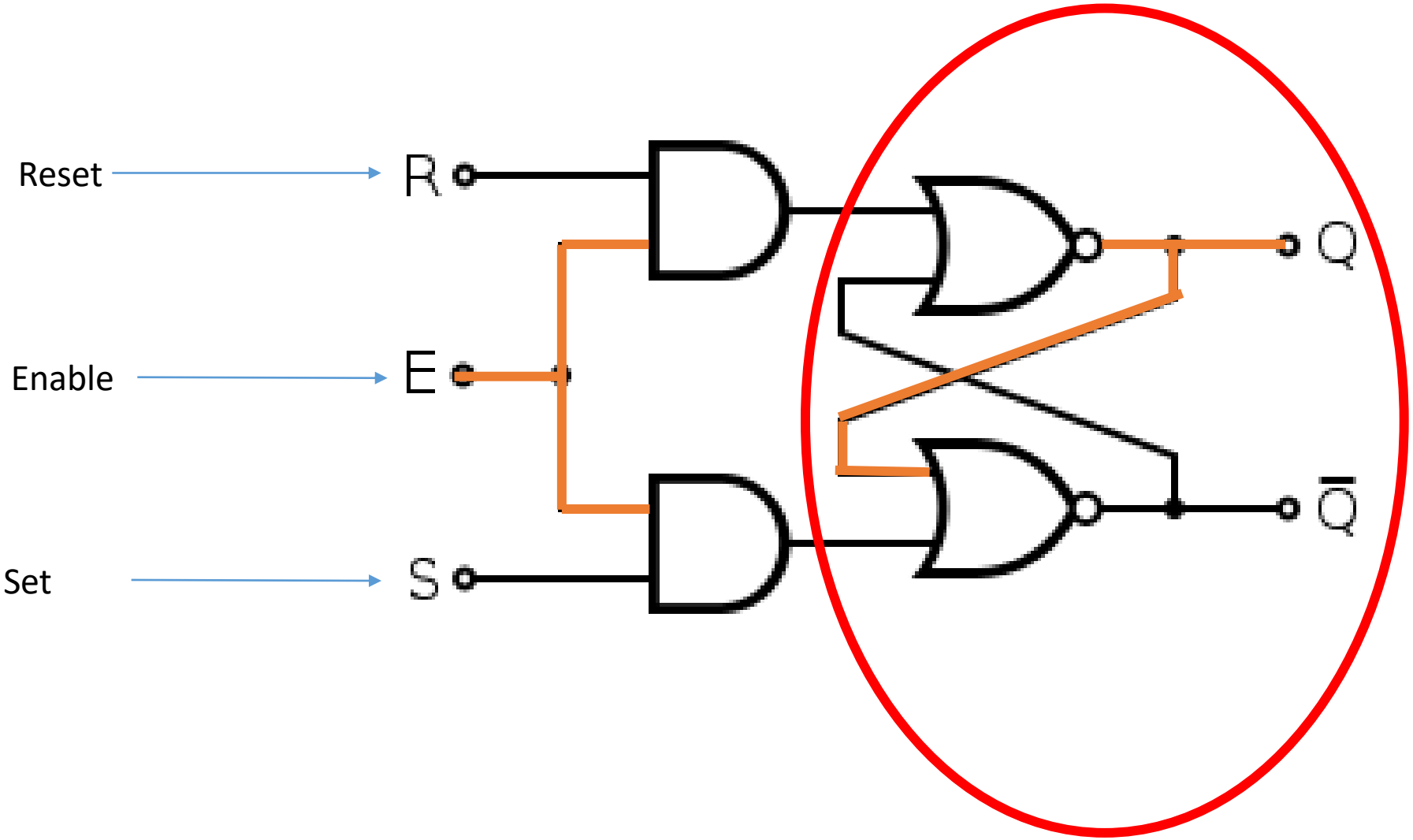
a	b	q
0	0	1
0	1	0
1	0	0
1	1	0



We release the “set” input

Truth Table of NOR

a	b	q
0	0	1
0	1	0
1	0	0
1	1	0

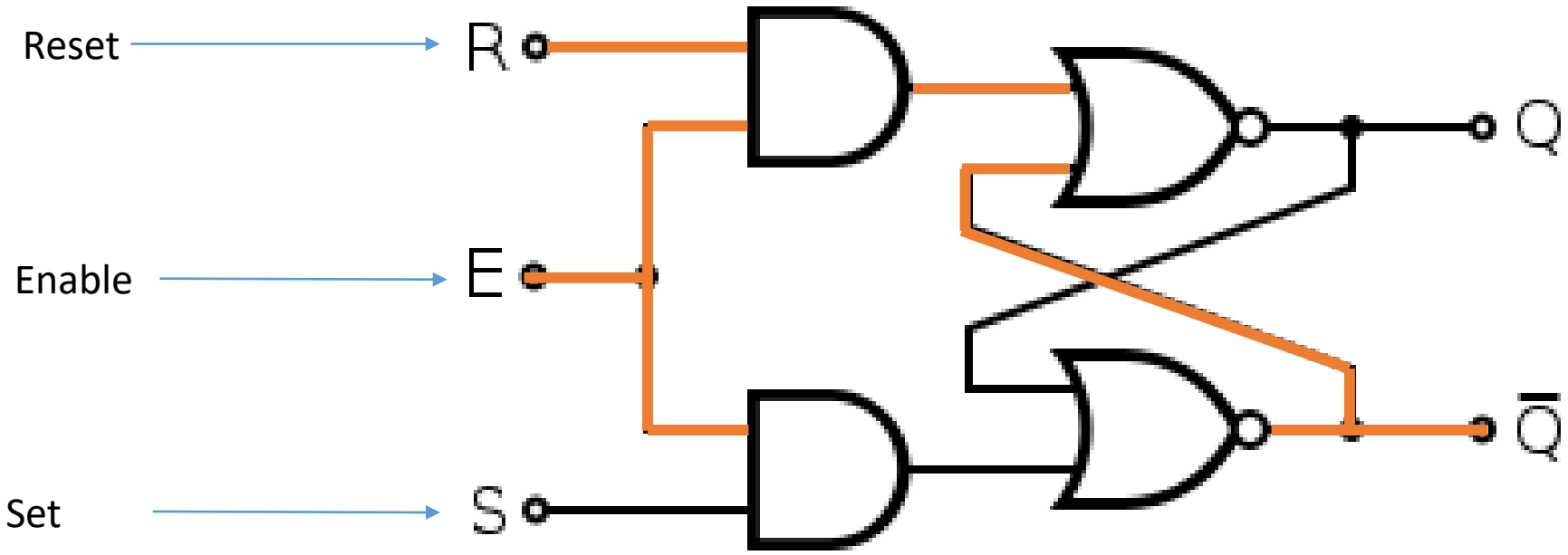


The feedback loop is in a stable state; The output value is kept – even if the “set” input is set to 0

We set the “reset” input

Truth Table of NOR

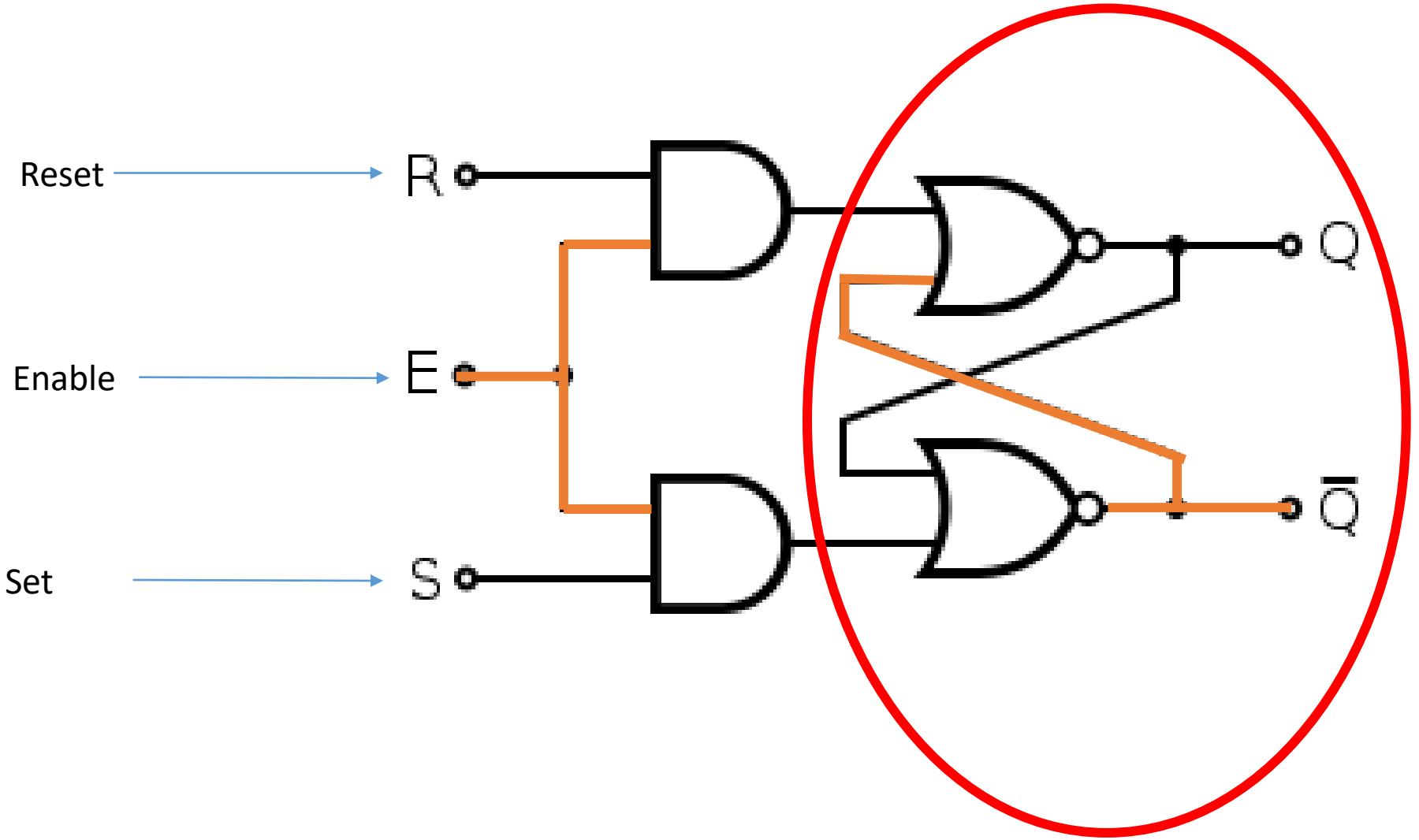
a	b	q
0	0	1
0	1	0
1	0	0
1	1	0



We release the “reset” input

Truth Table of NOR

a	b	q
0	0	1
0	1	0
1	0	0
1	1	0

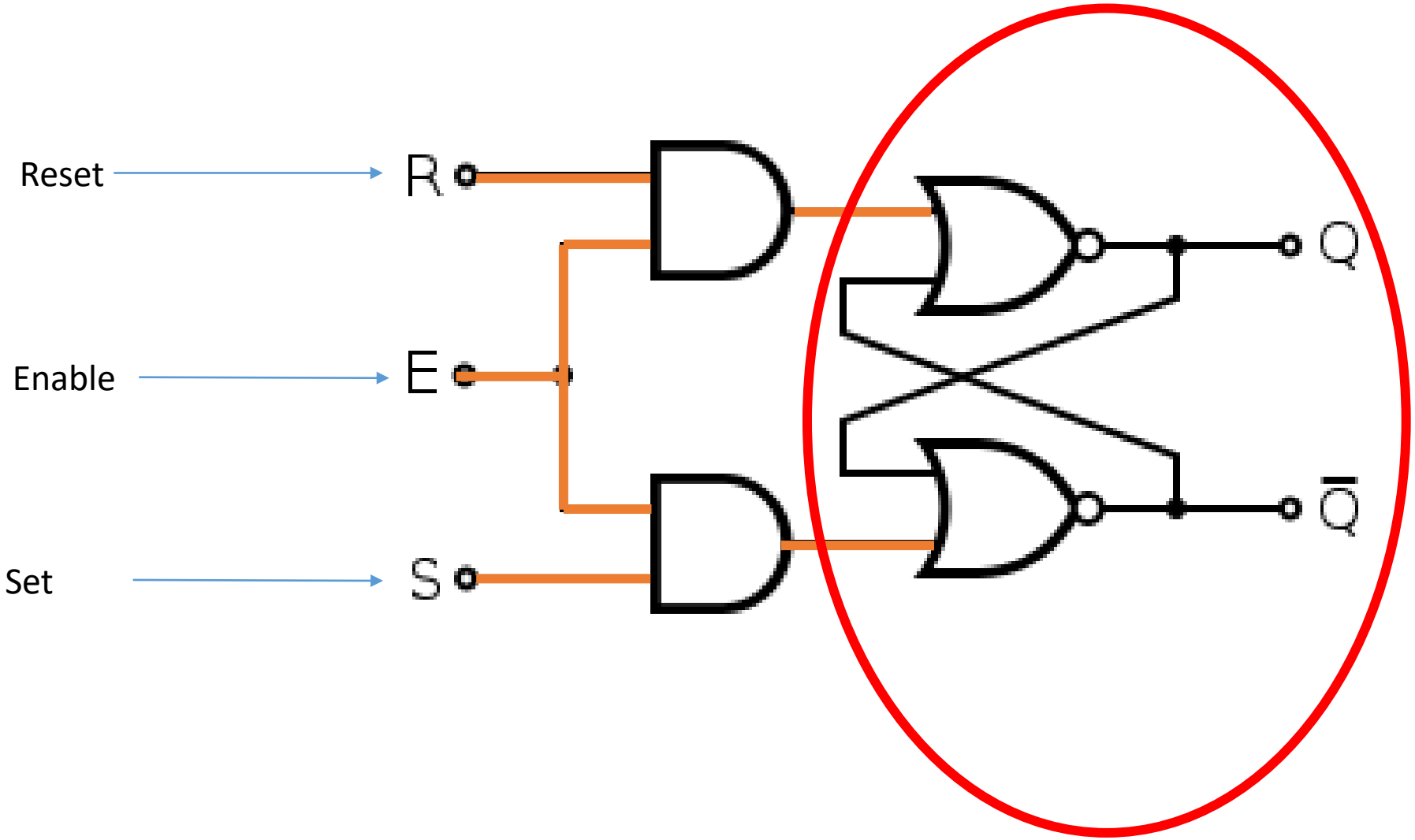


Again, the output value is kept

Illegal Action: We set “Set” and “Reset”

Truth Table of NOR

a	b	q
0	0	1
0	1	0
1	0	0
1	1	0



Invalid output state

Combining Computation and Storage

- There are many ways to combine gates for computation and for storage

  → It has turned out that only few scale to large circuit designs

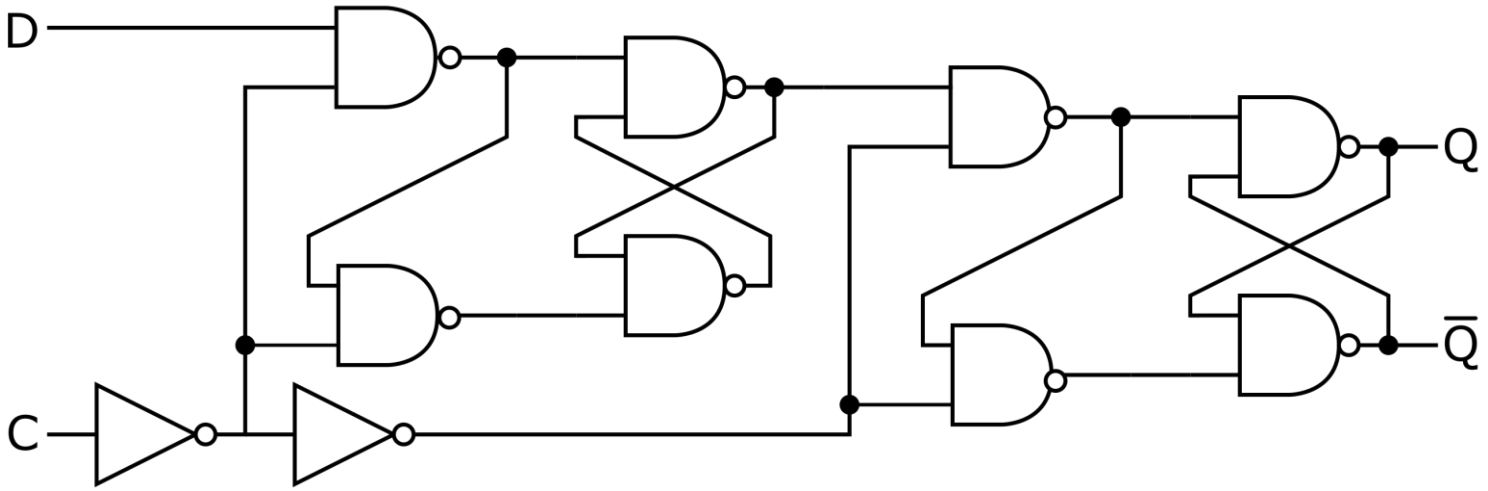
- Nearly all digital circuits are built as **synchronous circuits** with a global clock signal

  → These circuits **don't use latches** as storage, **but Flip-Flops** that are connected to a clock signal

  → This course focuses on synchronous circuits only

- There is also a design methodology for **asynchronous circuits** (self-timed circuits), but they are a nice topic

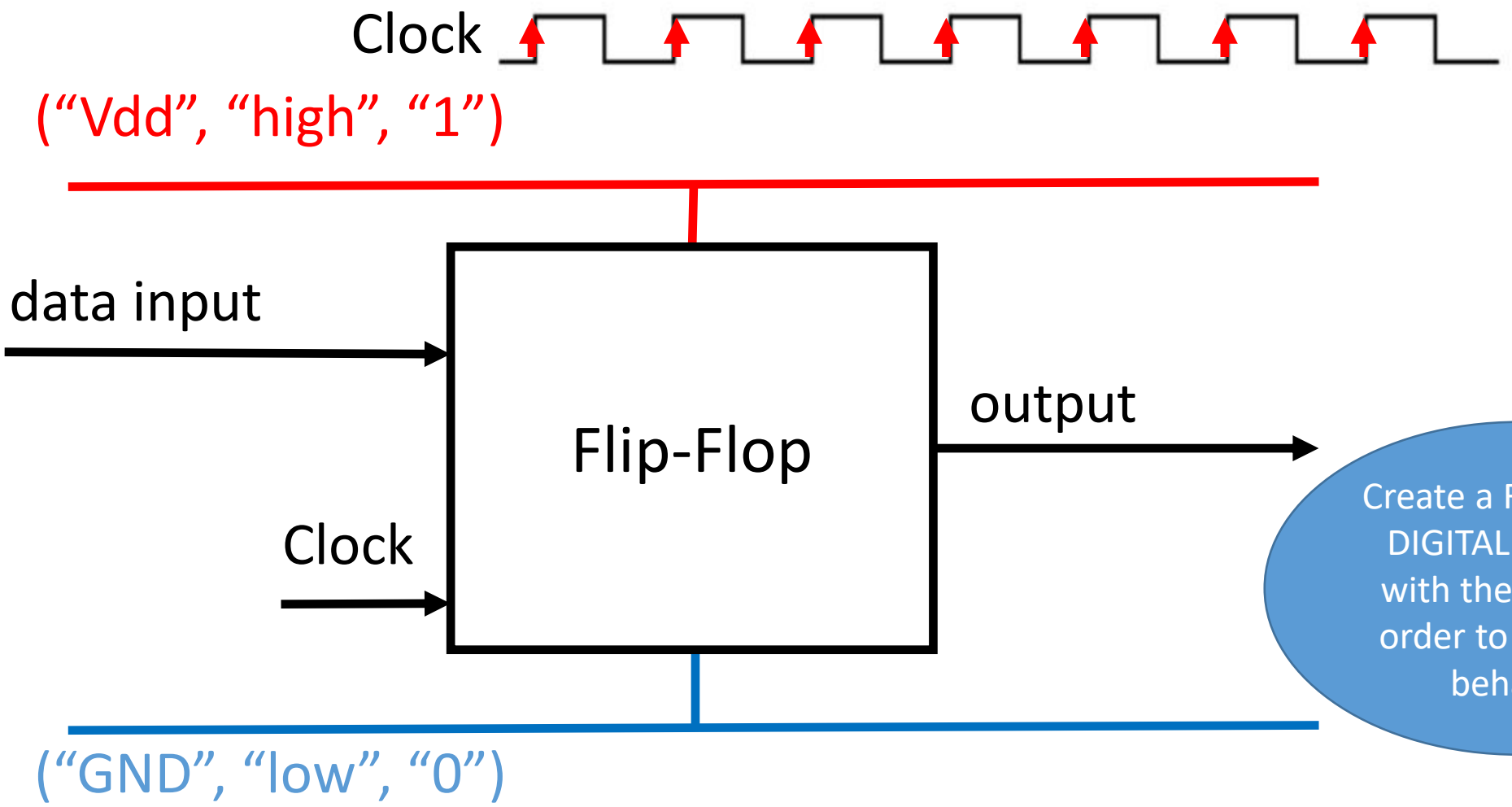
Flip-Flop based on CMOS Gates



Note: A flip-flop simply consists of two latches

Storage

- The flip-flop sets output = input when the clock switches from low to high;
- In all other cases, the input is ignored; the last “sampled” value is kept at the output

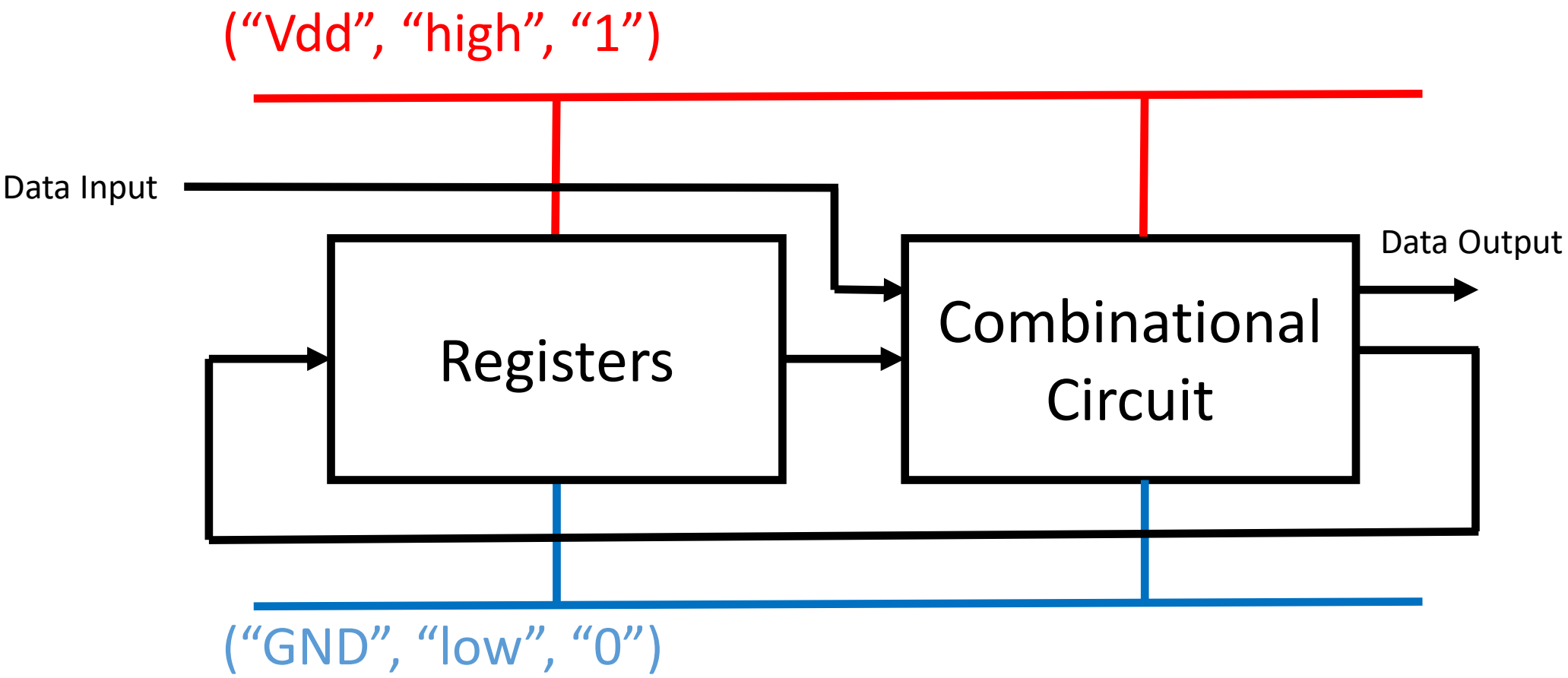


Create a Flip-Flop in DIGITAL and play with the inputs in order to learn the behavior

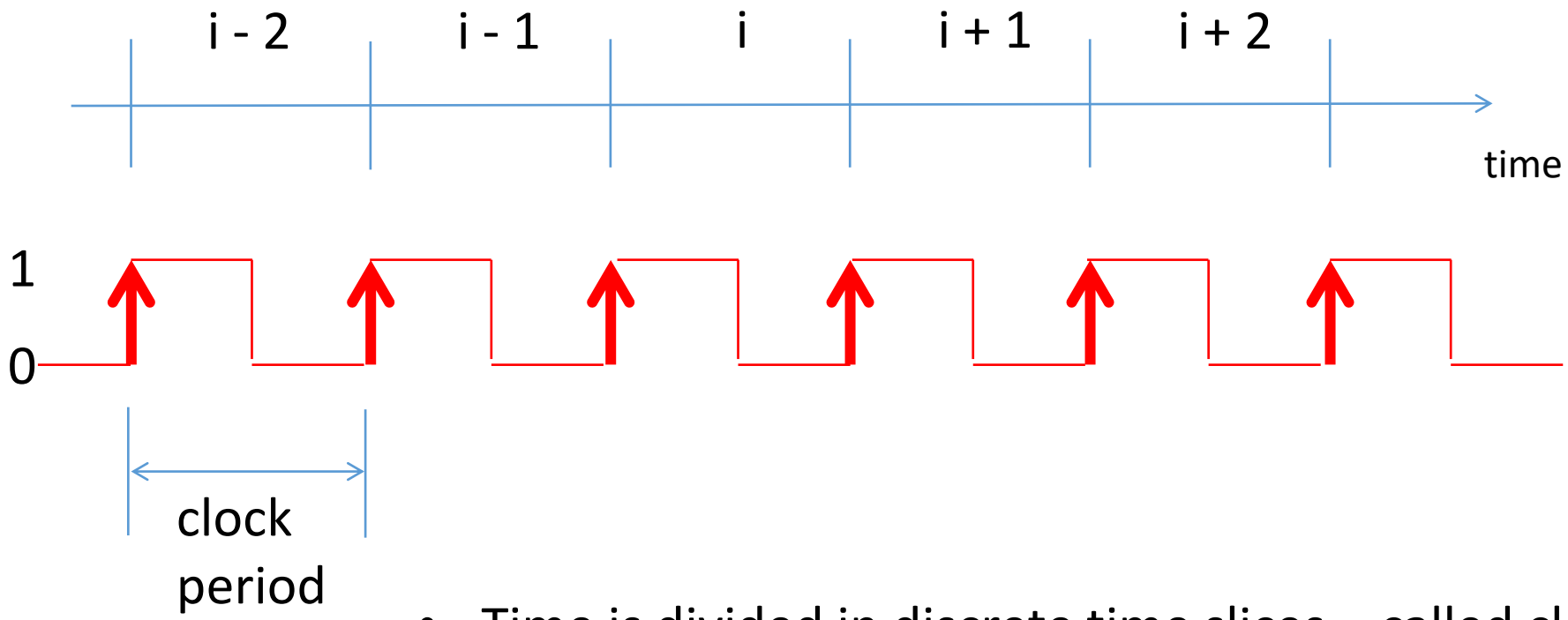
Naming Conventions

- Flip-Flop: A 1-bit storage sampling data on the rising clock edge
- Register: An n-bit storage sampling data on the rising clock edge

Combining


("Vdd", "high", "1")
Data Input
Registers
Combinational Circuit
Data Output
("GND", "low", "0")

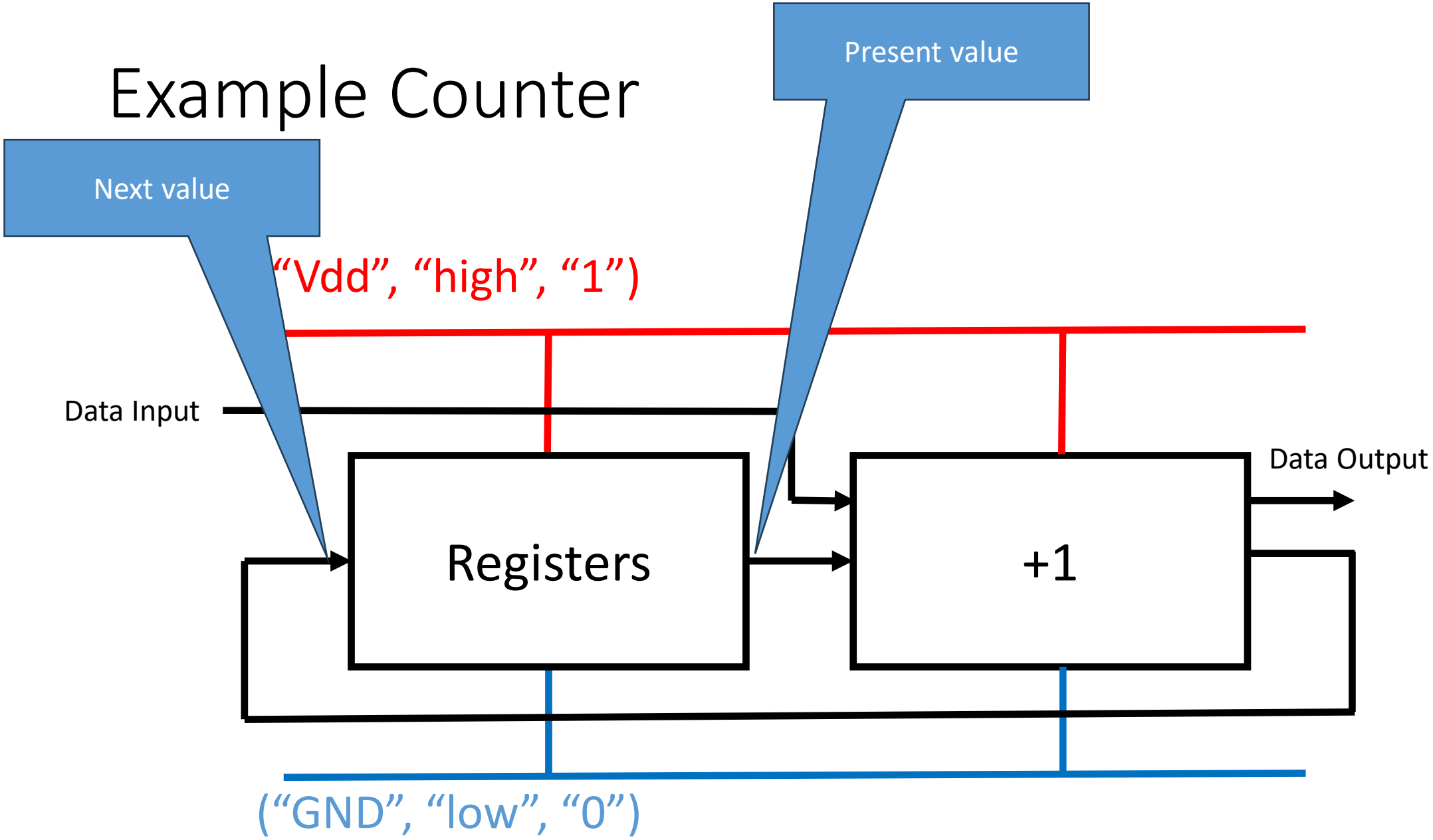
Sequential Logic Splits Time in Discrete Slices



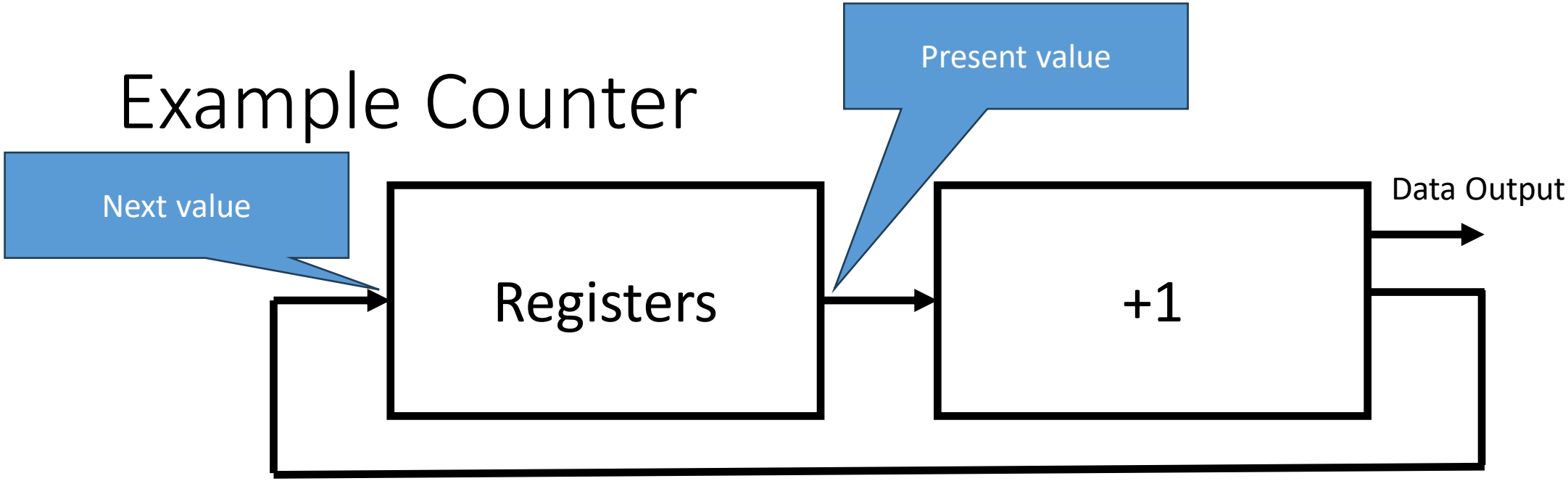
- Time is divided in discrete time slices – called clock cycles
- We call this time between two rising clock edges also “clock period”.

- On the rising clock edge the “next value” of every flip flop becomes the “preset value”

Example Counter



Example Counter



Clock

Register 0 1 2 3 4 5 6 7 …..

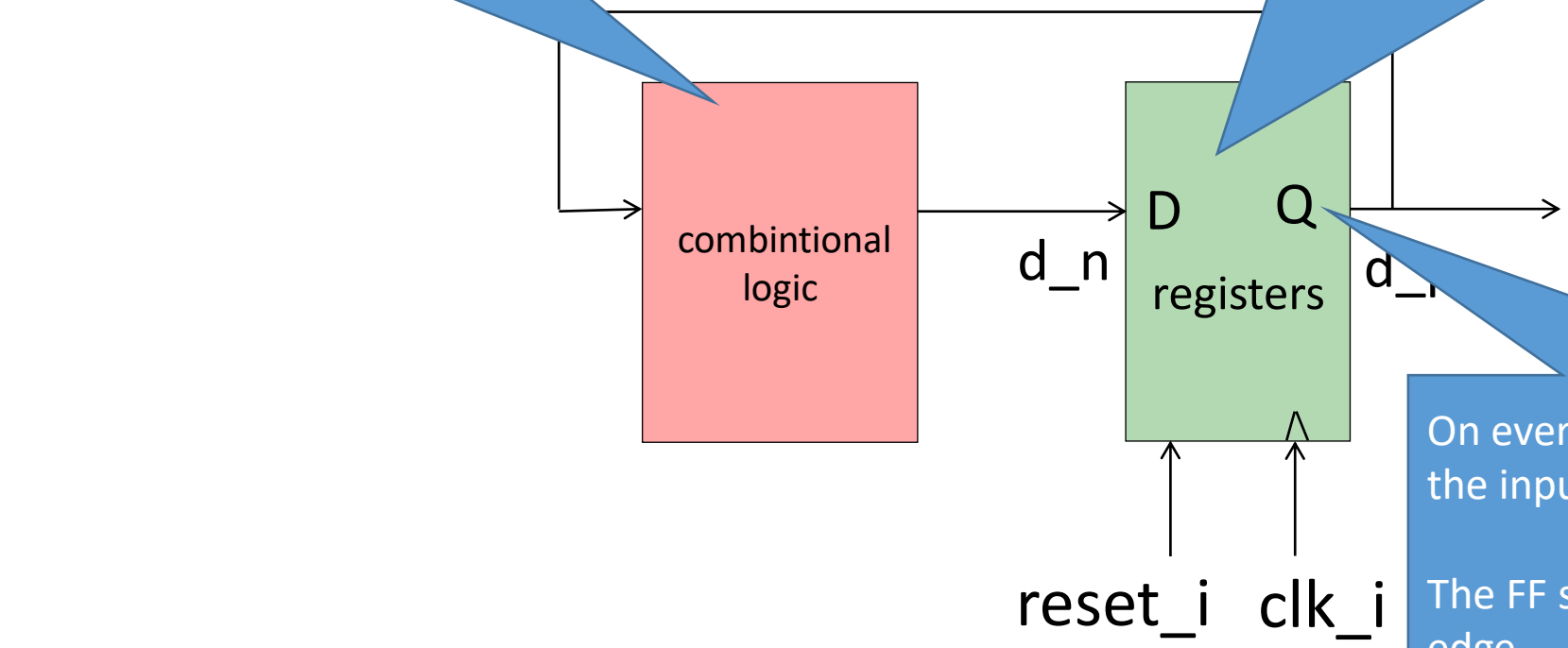
SystemVerilog

always_comb

Describes combinationl logic

always_ff @(posedge clk_i or posedge reset_i)

Descibes sequential logic (flip-flops, registers)



On every rising clock edge, the registers sample the input and provide it to the output

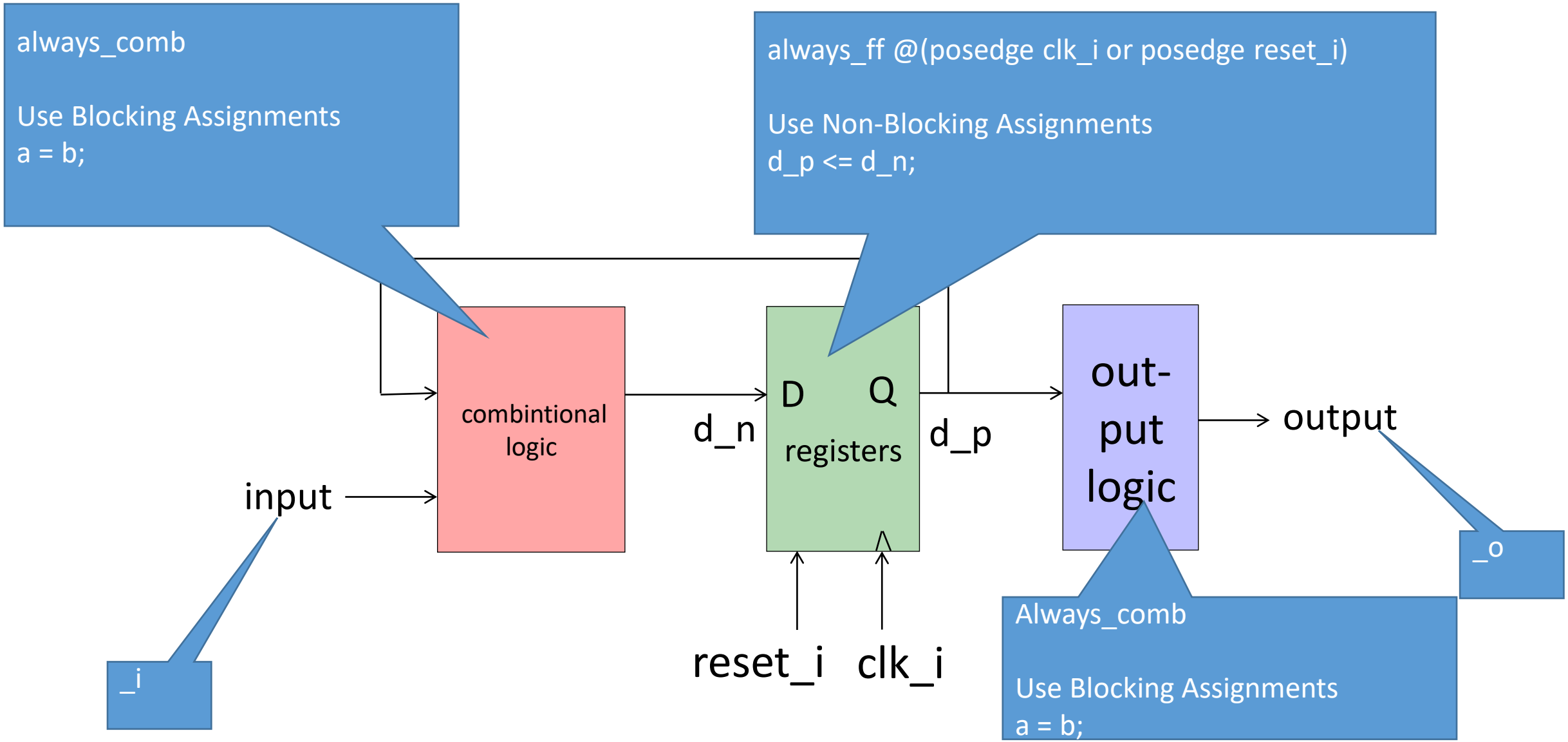
The FF sets d_p equal to d_n on the rising clock edge

Let’s Build This in SystemVerilog

- See example con03.02_addsub

https://extgit.iaik.tugraz.at/con/examples-2023.git

Coding Guidelines in SystemVerilog



SystemVerilog: Blocking vs. Non-Blocking Assignments

- Blocking Assignments (=) are done immediately and impact the subsequent operations. The following operations are only executed after the assignment has been completed.

- Non-Blocking Assignments (<=) do not impact subsequent operations in the same always block. These assignments describe parallel behaviour. Using them for FFs means describing the output of the FFs after the next rising clock edge

See example con03.03_blocking_vs_nonblocking

https://extgit.iaik.tugraz.at/con/examples-2023.git

Do not write code like in this example!

Do not mix blocking and non-blocking assignments in the same always block

Another Note on SystemVerilog

- The synthesis tool builds exactly what you describe – be careful not to create latches in your combinational circuit

  →CHECK what happens, if you remove the else branch from the example

  con01.03_simple_circuit_with_mux

  https://extgit.iaik.tugraz.at/con/examples-2023.git

  (You need to replace **always_comb** by **always @(*)** to see the creation of latches → the always_comb statement of SystemVerilog helps to avoid latches in com)

Summary of CON SystemVerilog Coding Style

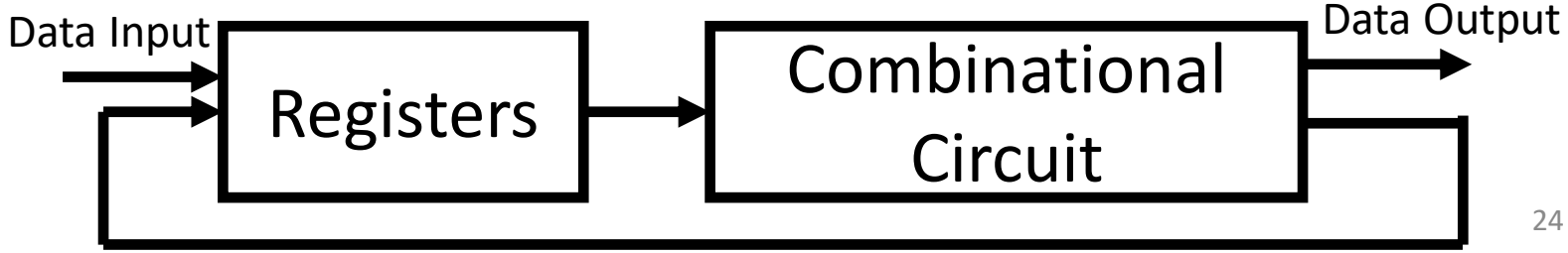
- Suffix _o for module outputs, _i for module inputs
- Register variables with suffix _p for present and _n for next value
- Array range with [MSB:LSB], like e.g. [31:0]
- Clocked processes use non-blocking (<=) others use blocking assignments (=)
- Clocked processes only update registers, everything else has to be done in combinational blocks
- Filename corresponds to module name: module MyDesign in file mydesign.sv
- Module instantiation always with named assignments (.A(C))

- **With significant implications beyond style:**
  - Always use default assignments (e.g. state_n = state_p)
  - Always use default branches (default:) in case statements

  → If you do not assign the output of a combinational block for all input conditions, latches are created for data storage!

The Clock Frequency

- Can we increase the clock frequency arbitrarily?
- The clock frequency is limited by the time the combinational circuit needs to compute its outputs.
- The critical path is the path with the longest propagation delay in the combinational circuit. It defines the maximum clock rate



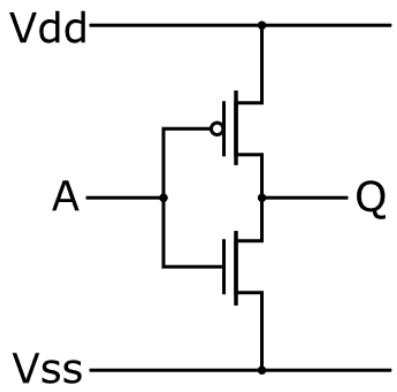
Temperature, Power Consumption

- The higher the temperature, the slower the transistors become and the lower becomes the maximum clock rate

  → The lower the temperature,
  the higher clock rates are possible

- Why does a CPU produce heat?
  - Every time a logic gate switches, NMOS and PMOS transistors are open at the same time → there is a short current.
  - Upon a switch, there is also current flowing to charge and discharge parasitics

  → The more transistors are switching, the more heat is produced



Clock Frequency Too High

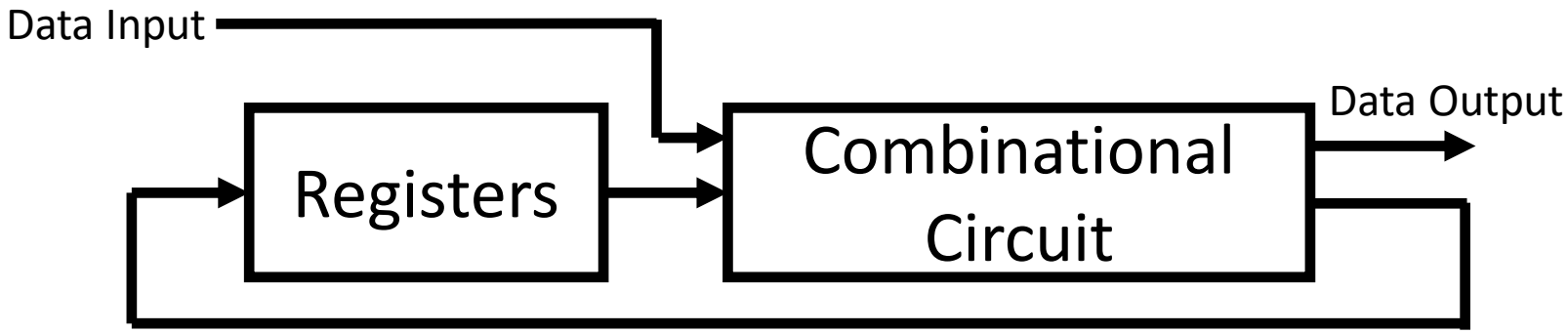
What happens, if the clock frequency is too high?

- The circuit stores an intermediate state of the combinational circuit in the registers.

- The intermediate state depends on the physical layout, the temperature, fabrication details, … → hard to predict; overclocking a processor too much typically leads to a crash

Observations

- What we have discussed the basics of combinational and sequential circuits



- In order to build large systems composed of registers and combinational logic, we need a structured approach and more tools and theory to describe our systems

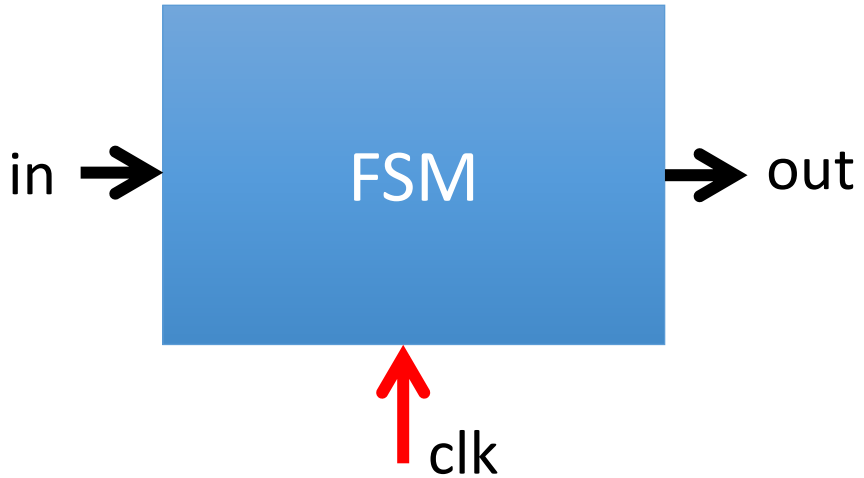
State Machines

Finite State Machines (FSMs)

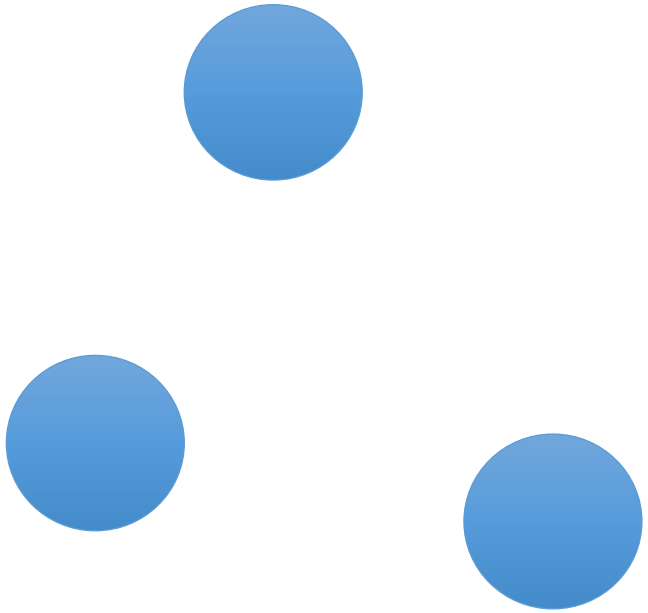
- FSMs are the “work horse” in digital systems.
- We look at “synchronous” FSMs only:
  - The “clock signal” controls the action over time
- FSMs can be described with different “views”:
  - The functional view with the “state diagram”
  - The timing view with the “timing diagram”
  - The structural view with the “logic circuit diagram”
  - The behavioral view with “SystemVerilog”

Finite State Machine (= automaton)

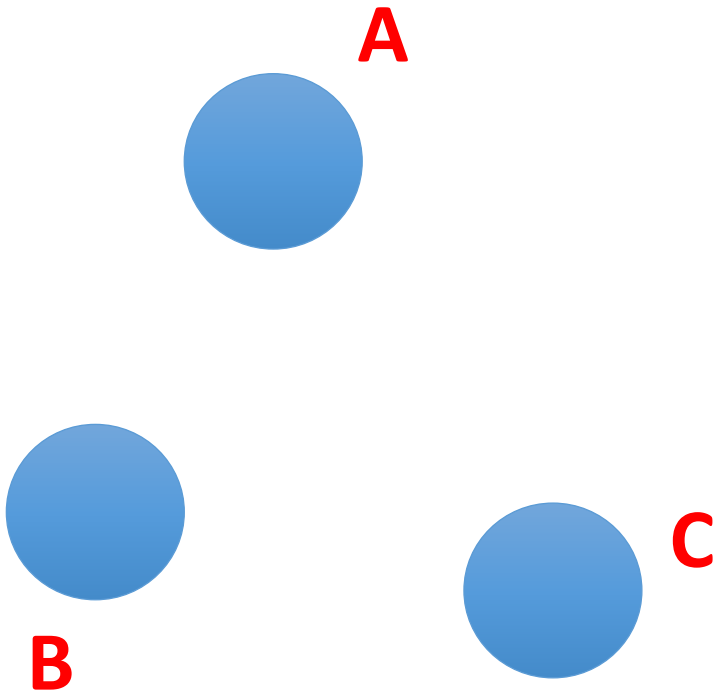
- A **synchronous FSM** is clocked by a clock signal (“clk”)
- In each clock period, the machine is in a defined (current) **state**.
- With each rising edge of the clock signal, the machine advances to a defined next state.



The sequence of states can be defined in a **state diagram**.

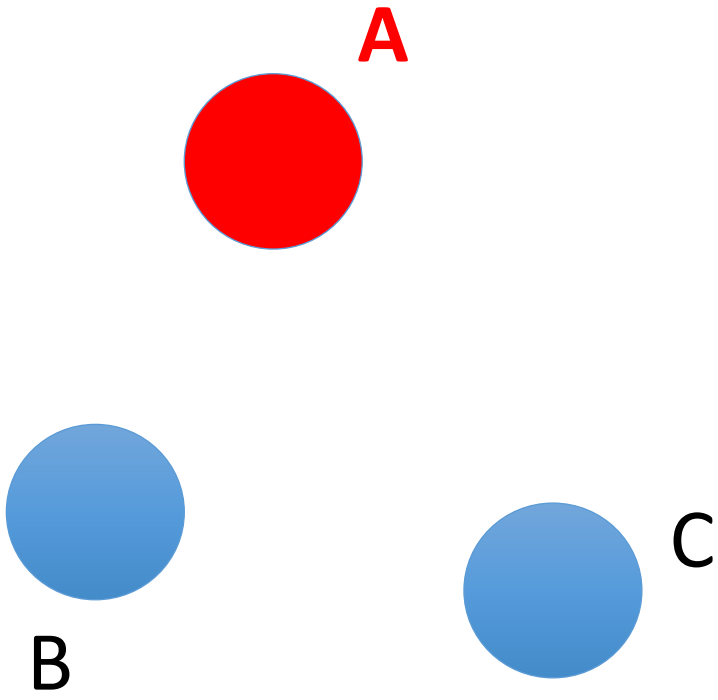


State diagram:



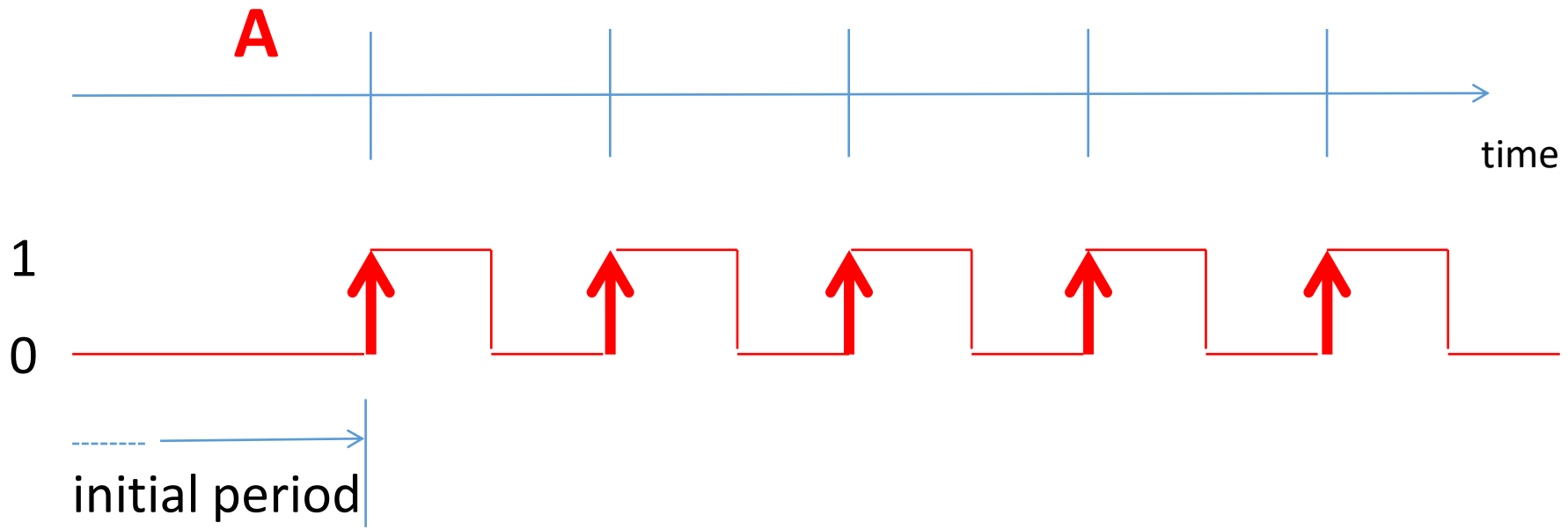
We denote the states with circles and give them **symbolic names**, e.g. A, B, and C.

State diagram:



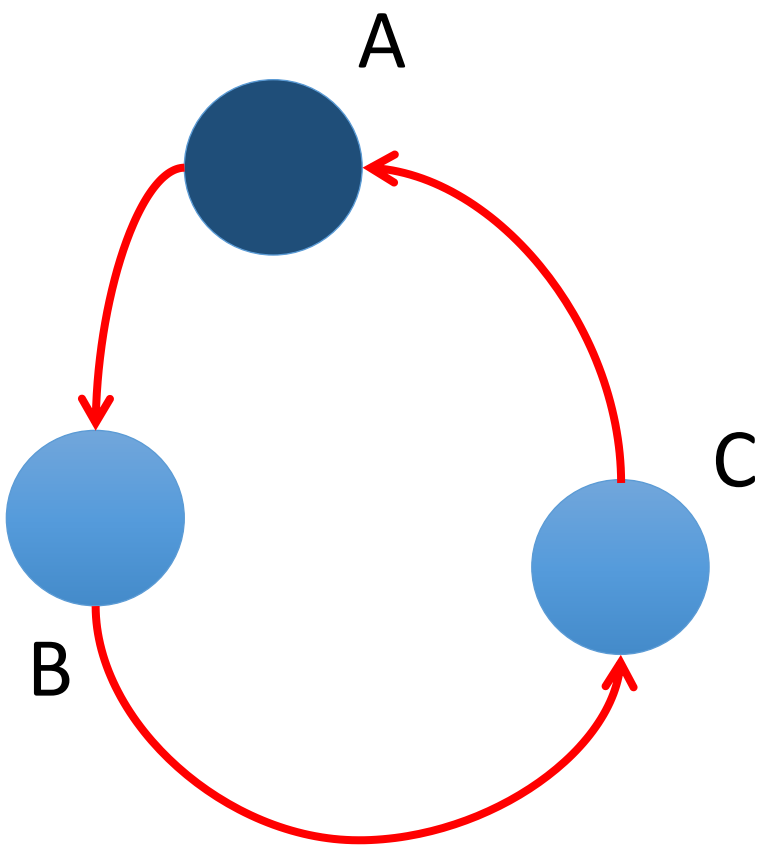
We define one of the states as the **initial state**.

In the beginning…



Initially, i.e. shortly after switching on the FSM and before the first rising edge of clock, there is the initial period. In this period, the FSM is in the "initial state".

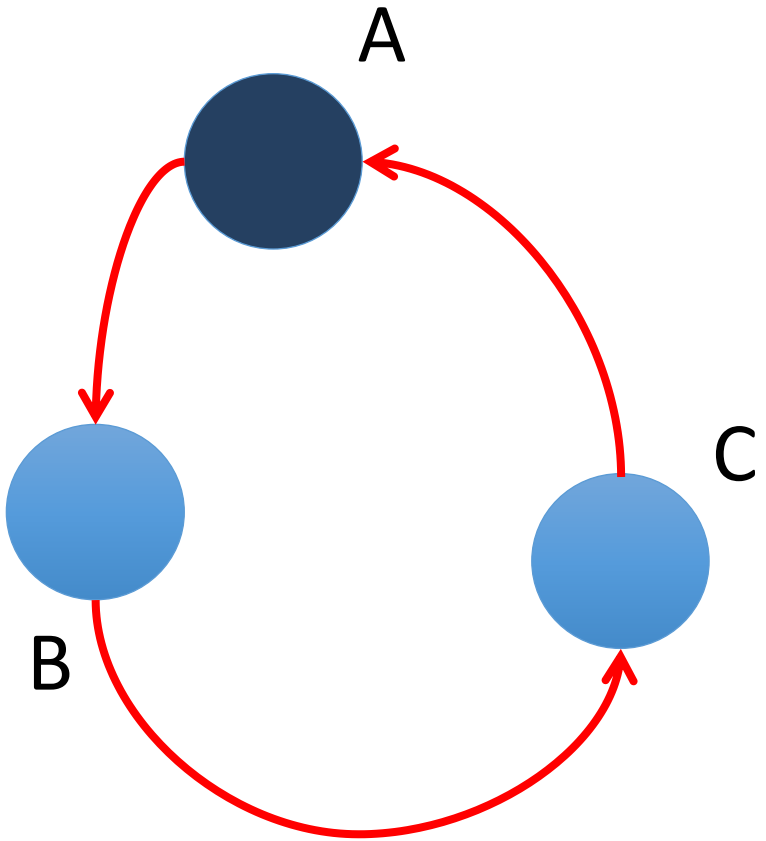
State diagram:



With arrows we define the sequence of states.

The sequence of states can also be defined in a **state transition table**.

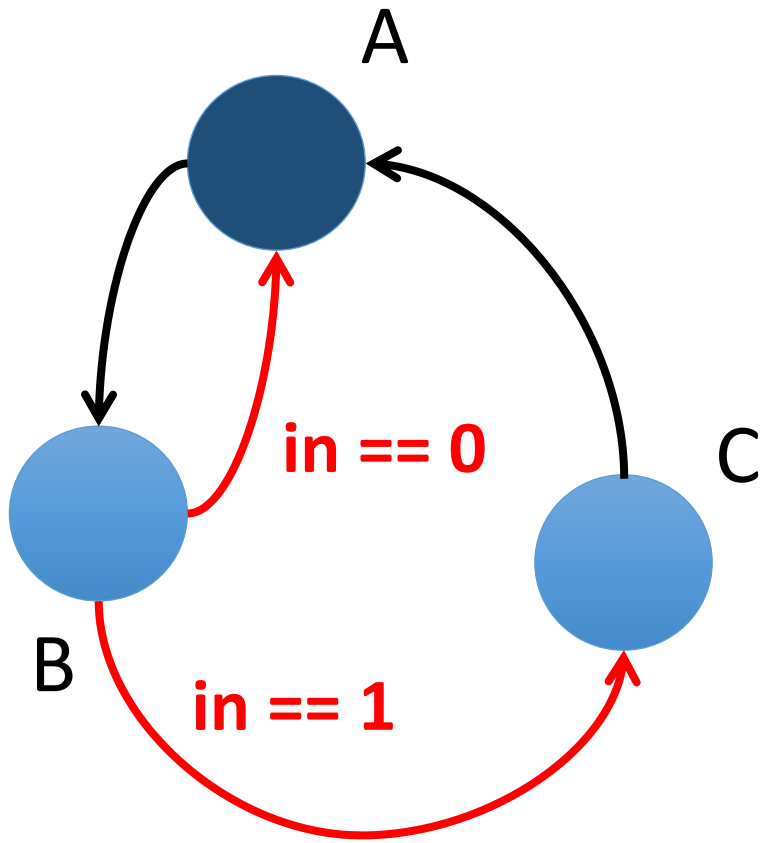
present state	next state
A	B
B	C
C	A



FSMs typically also have inputs influencing the transition to the next state

next state = f(state, input)

In this example we see that the one-bit input “in” influences the choice of the state after B.

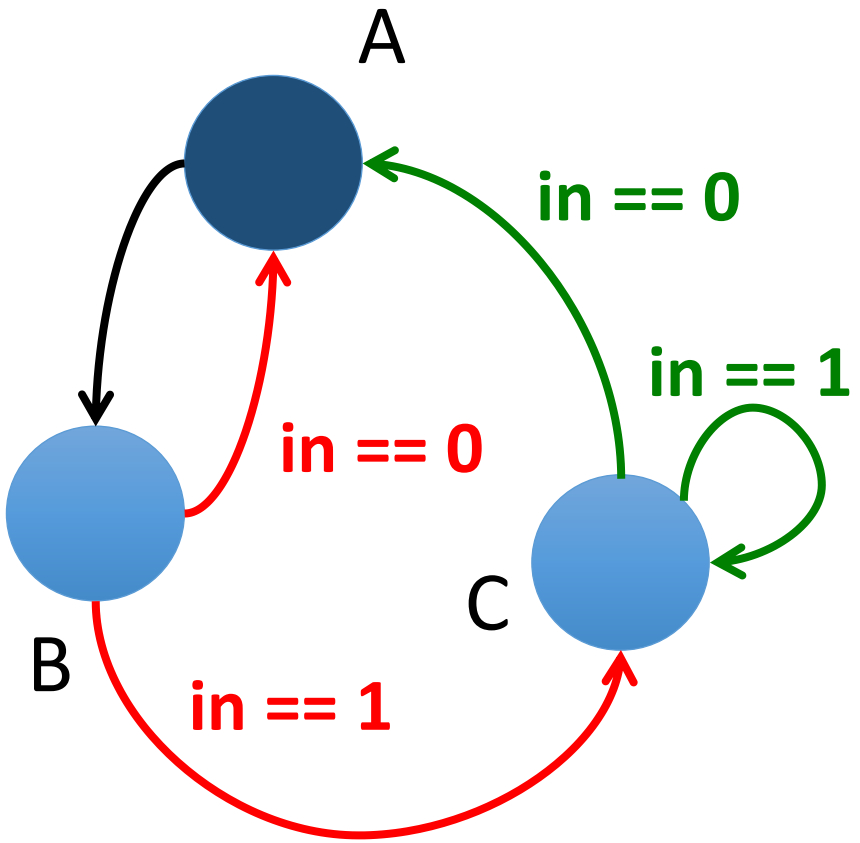


FSMs typically also have inputs influencing the transition to the next state

next state = f(state, input)

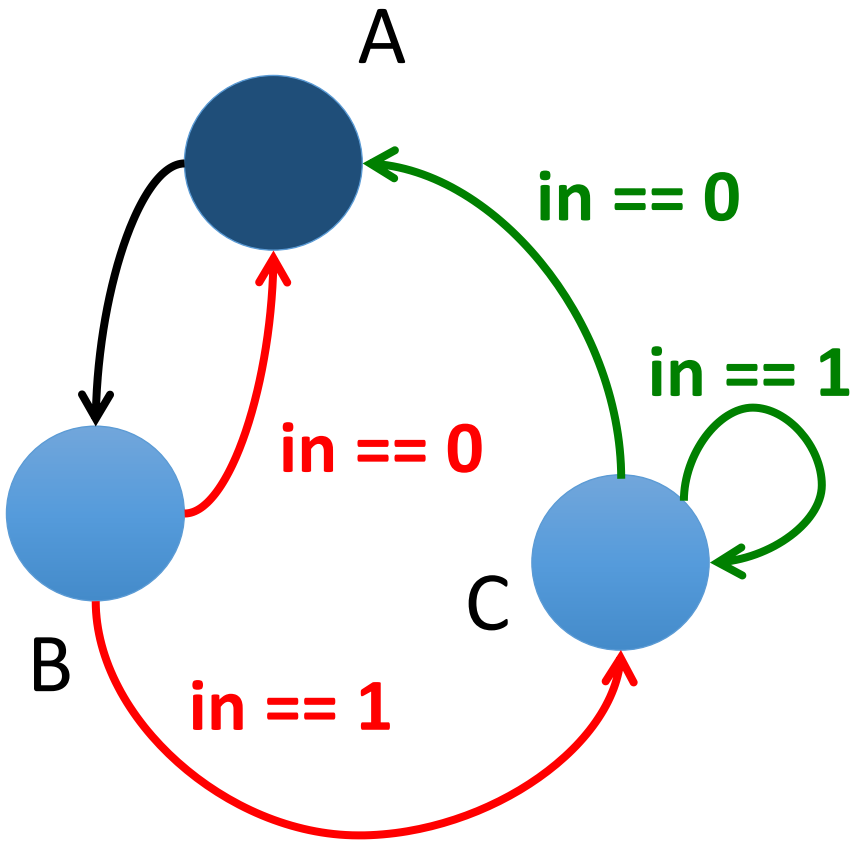
In this example we see that the one-bit input "in" influences the choice of the state after B.

The following state can also be the same as the current state.

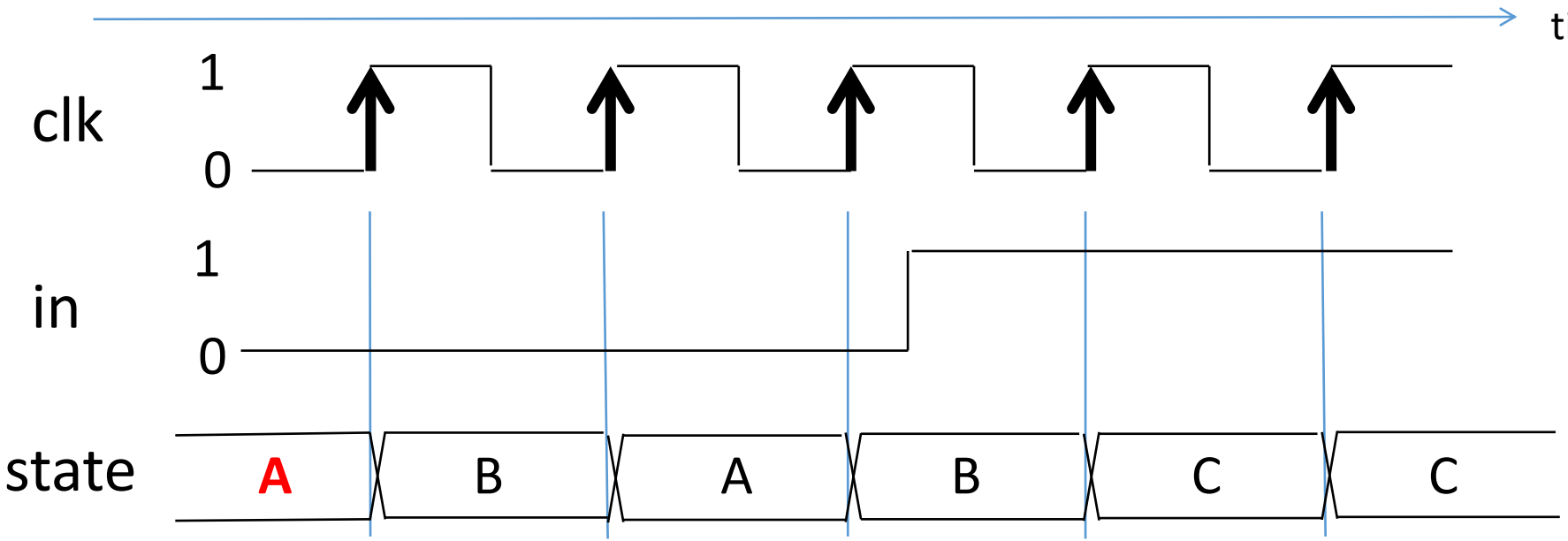


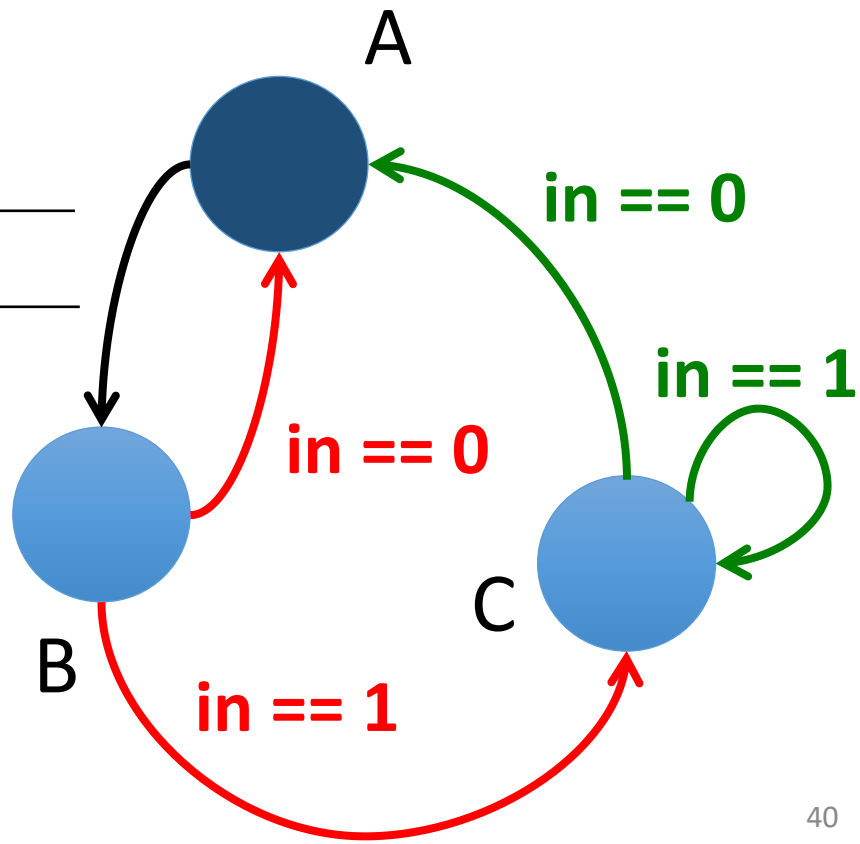
The State Transition Table

present state	in	next state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	C

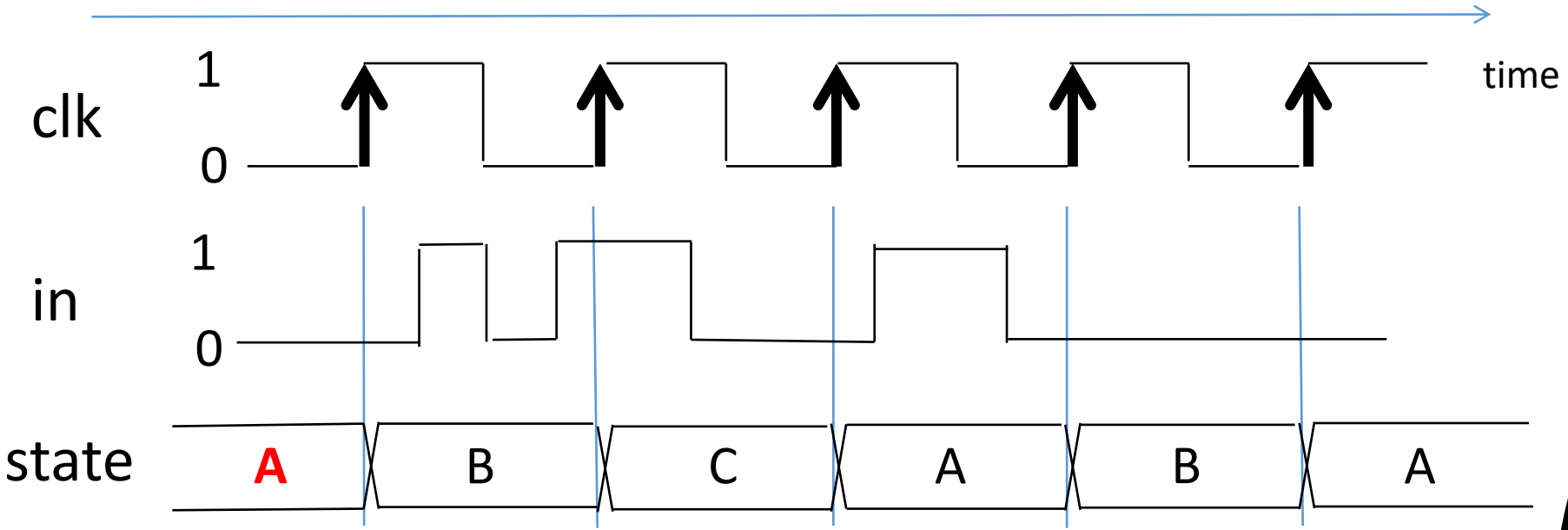


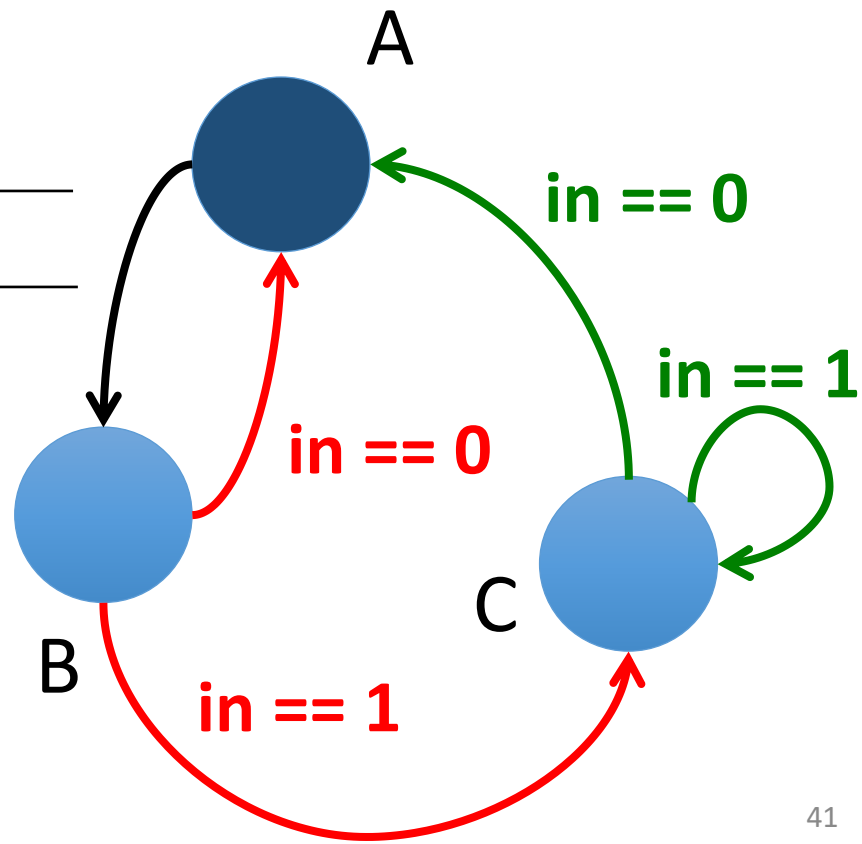
Timing Diagram – Example 1





Timing Diagram – Example 2

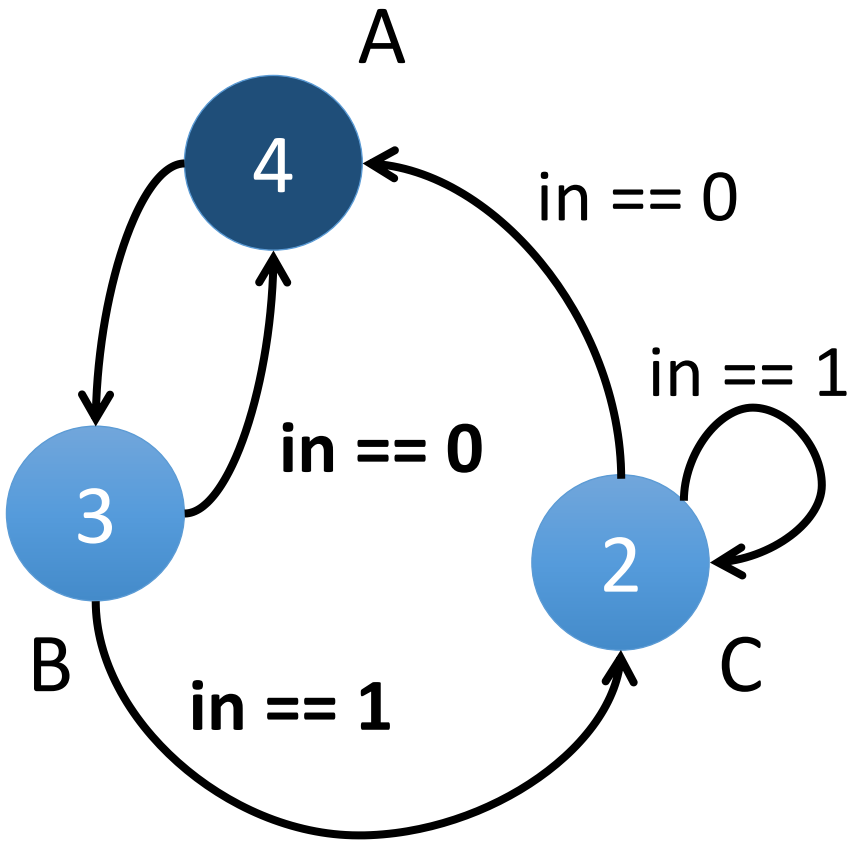




FSMs typically also have outputs

In this example the outputs are a function of the state. We write the output values into the circles.

We call such machines also **"Moore machines"**:



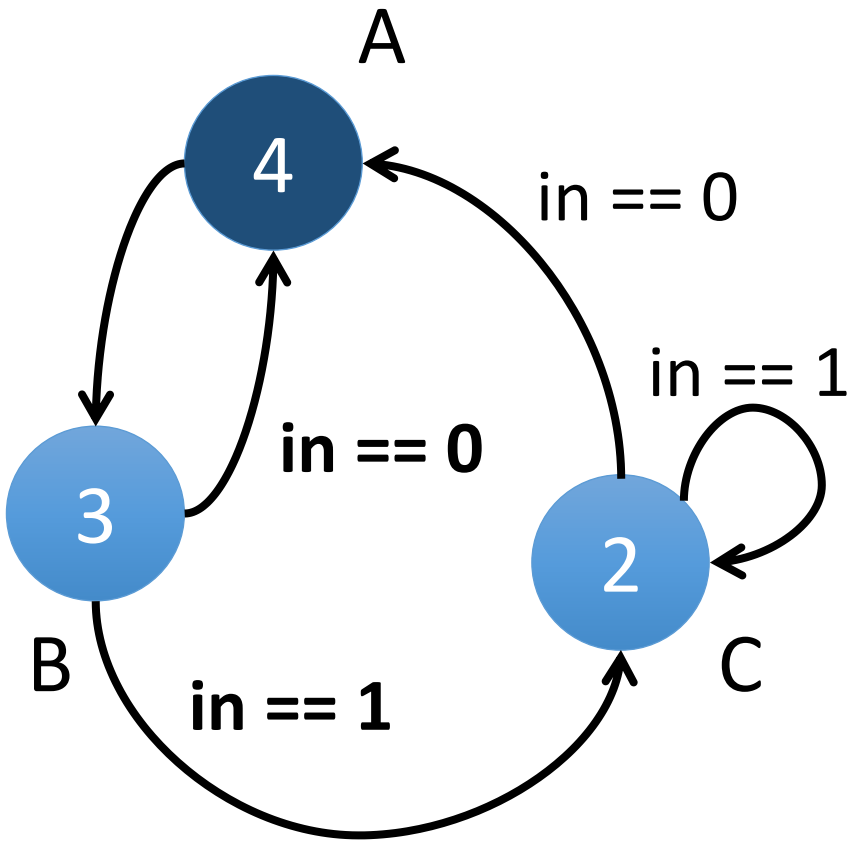
output = f(state)

We define the outputs with the “output function”

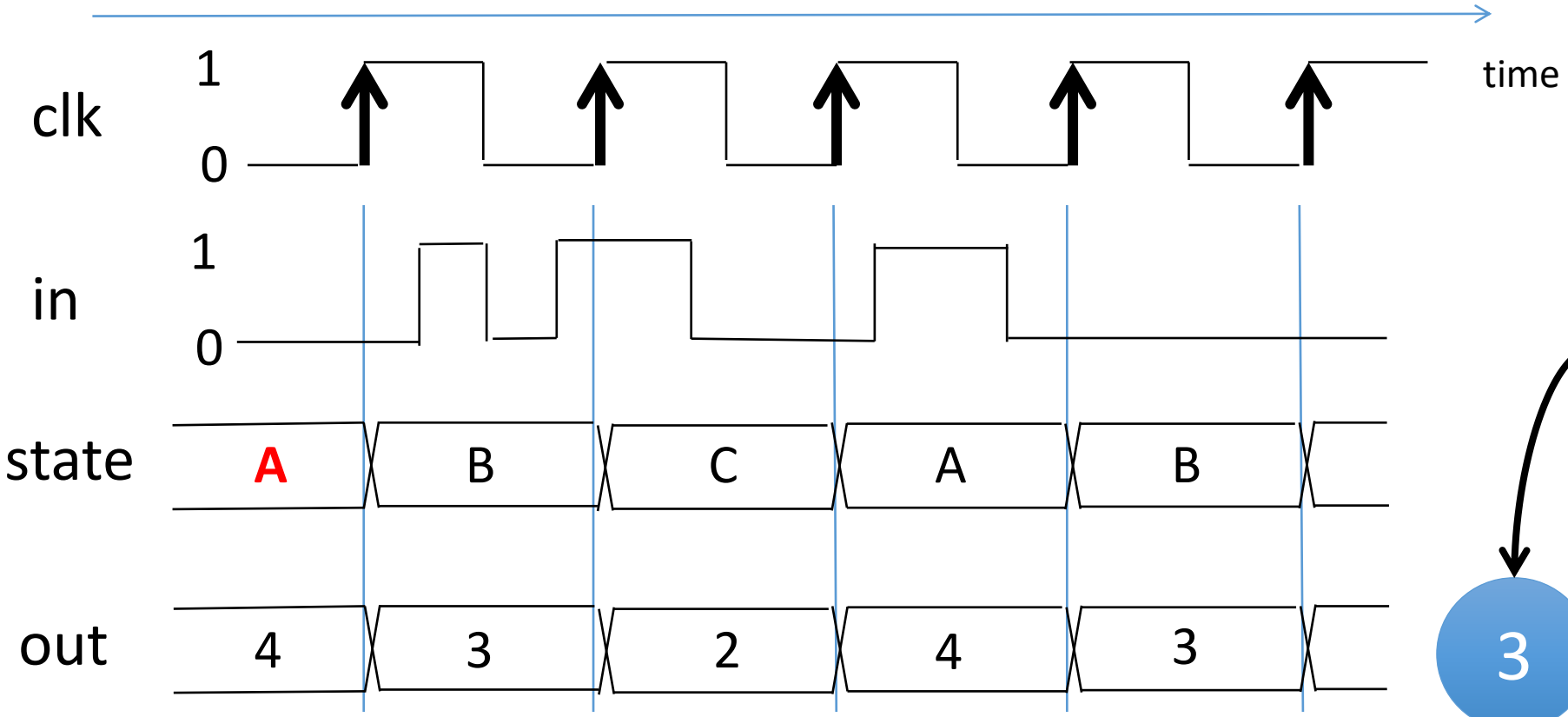
Moore machines:

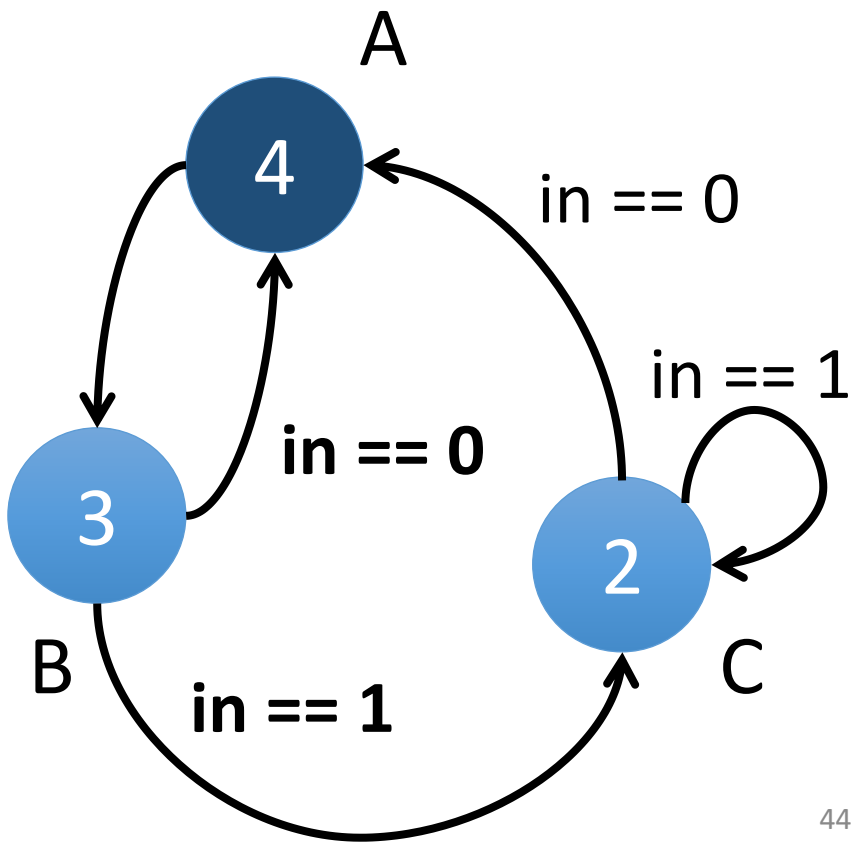
output = f(state)

state	output
A	4
B	3
C	2

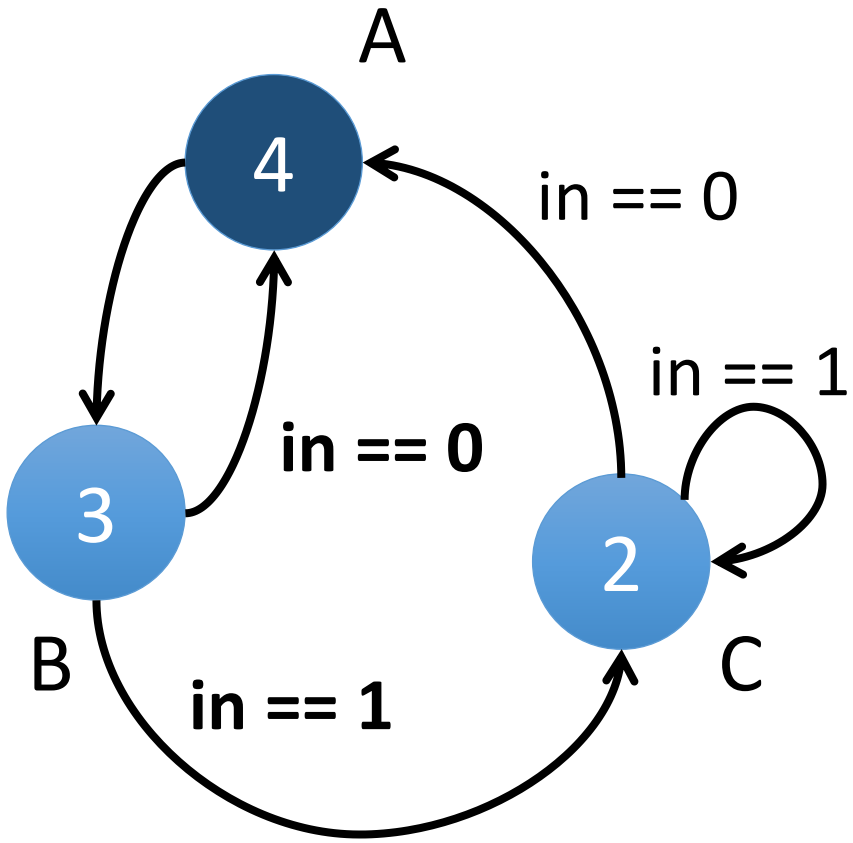


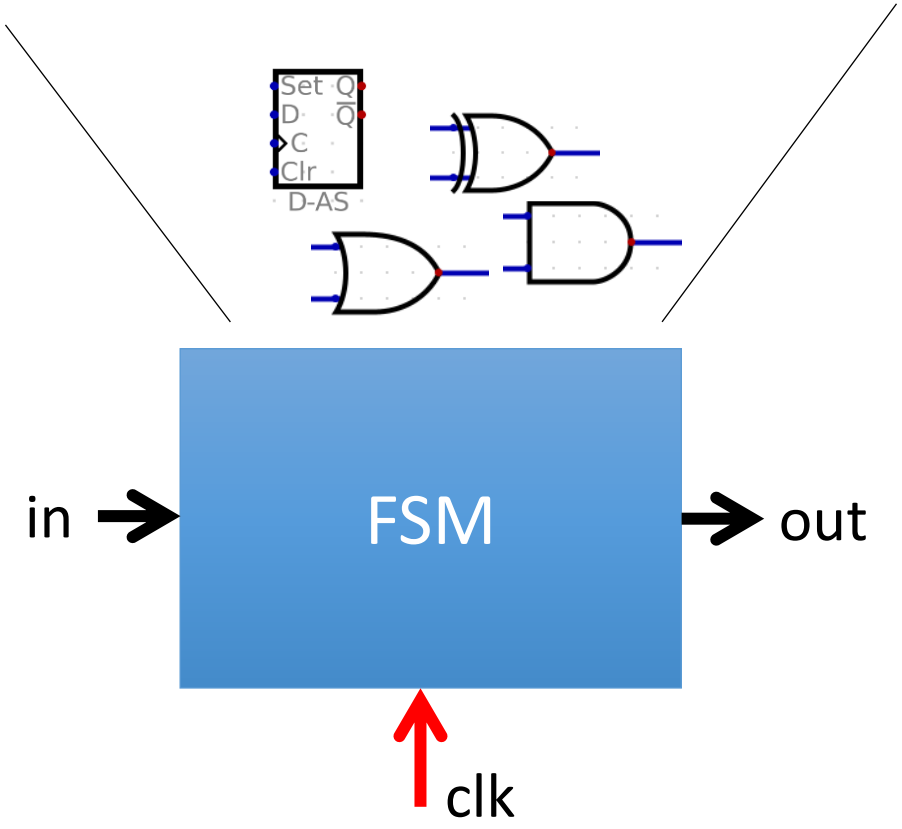
Timing Diagram – Example 3



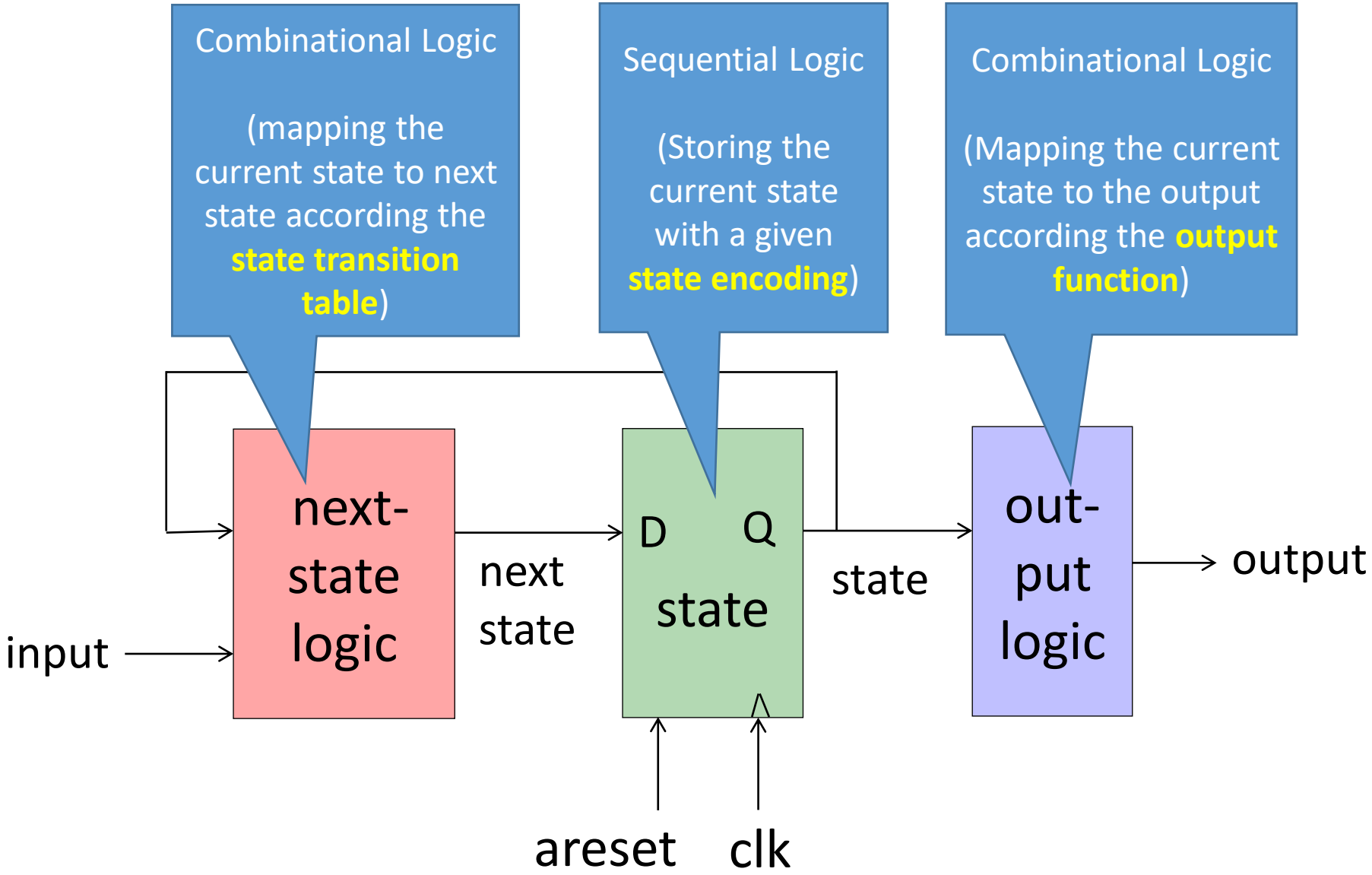


Mapping a State Diagram to Hardware





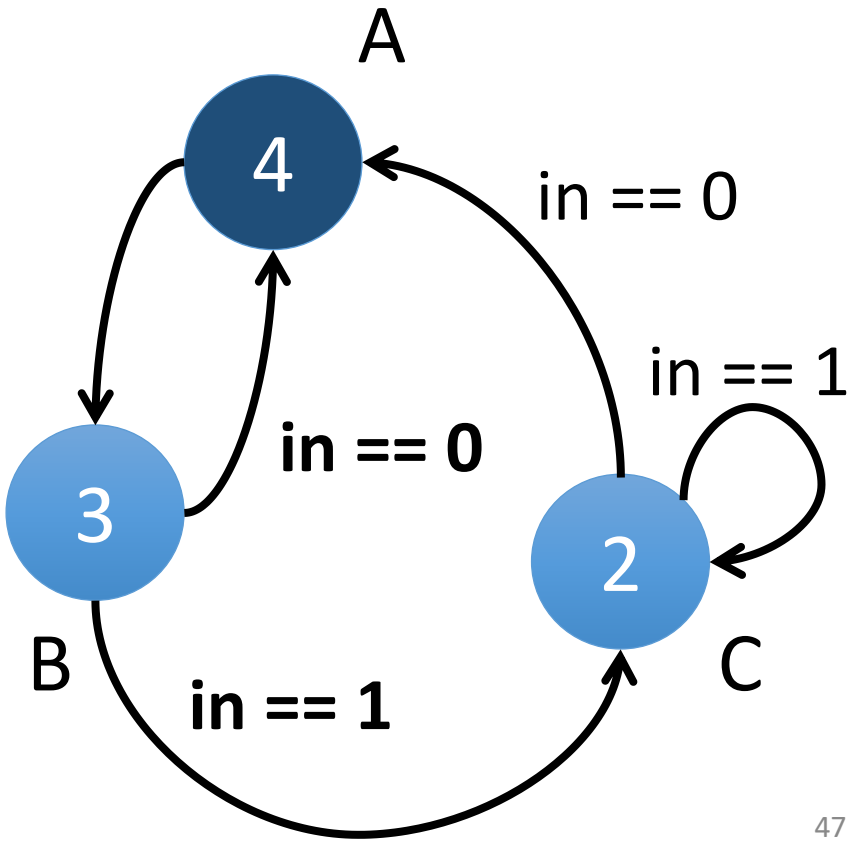
Essence of Moore Machines



State Encoding

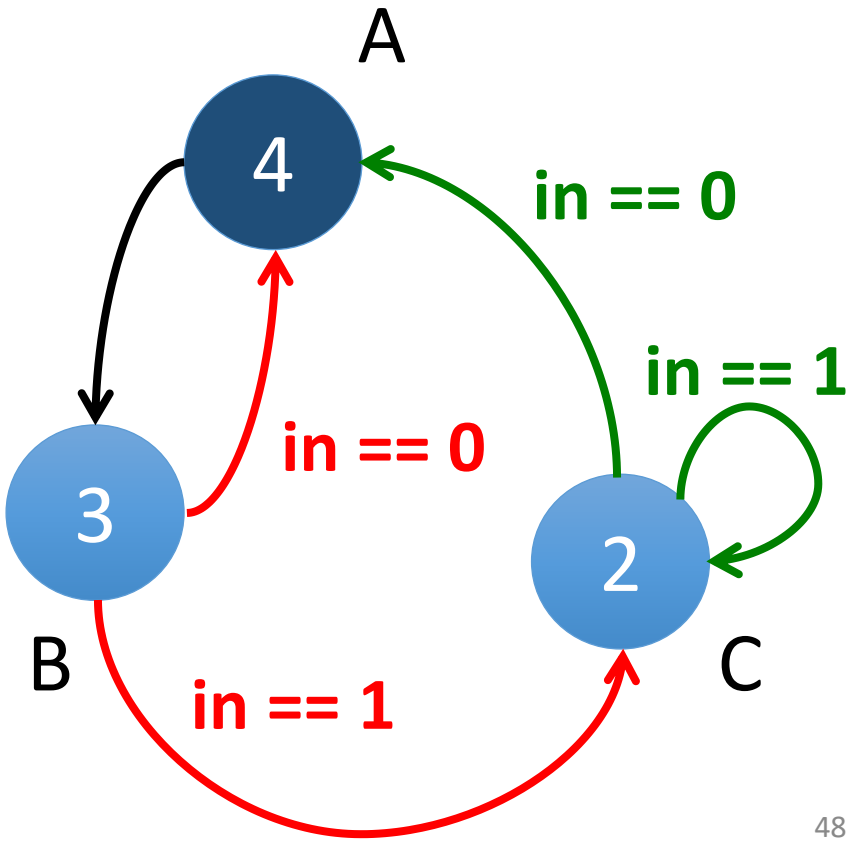
We use binary enconding → we need two bits to encode the three states A, B, C

state	encoding
A	00
B	01
C	10



State Transition Table

present state	in	next state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	C

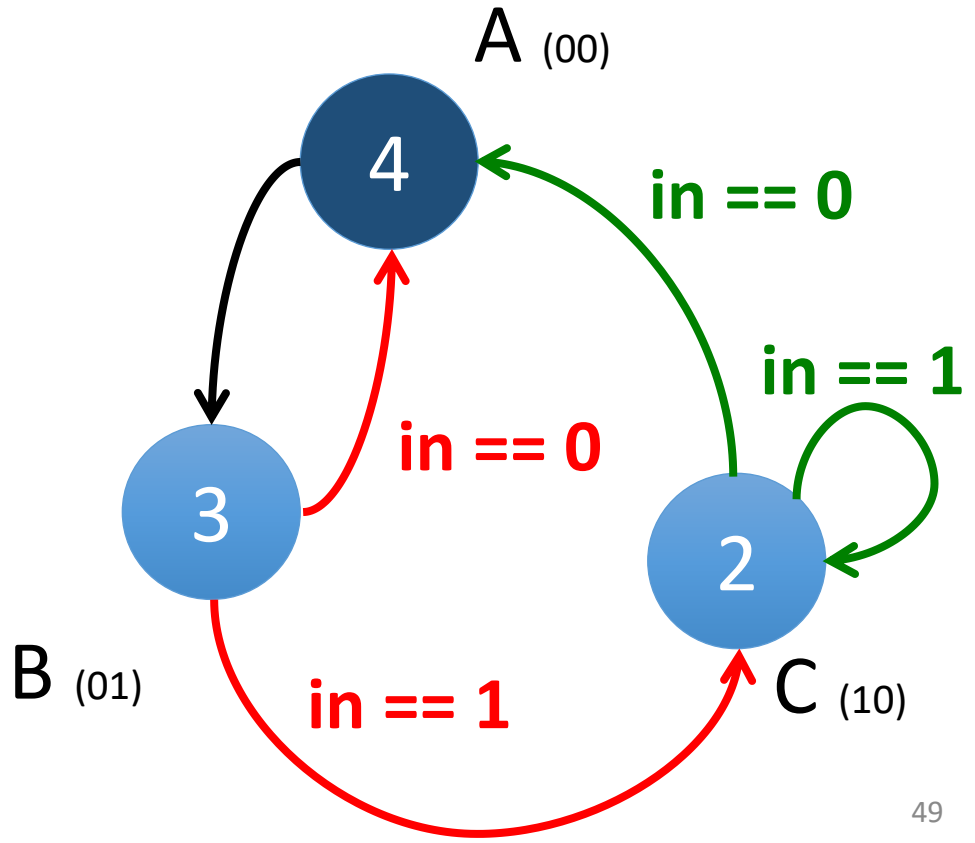


“11” does not exist: We use “Don’t Care” as the following state

present s1	present s0	in	next s1	next s0
0	0	0	0	1
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	x	x
1	1	1	x	x

next s0 = ((~s1) & (~s0) & (~in)) | ((~s1) & (~s0) & in)

next s1 = ((~s1) & s0 & in) | (s1 & (~s0) & in)



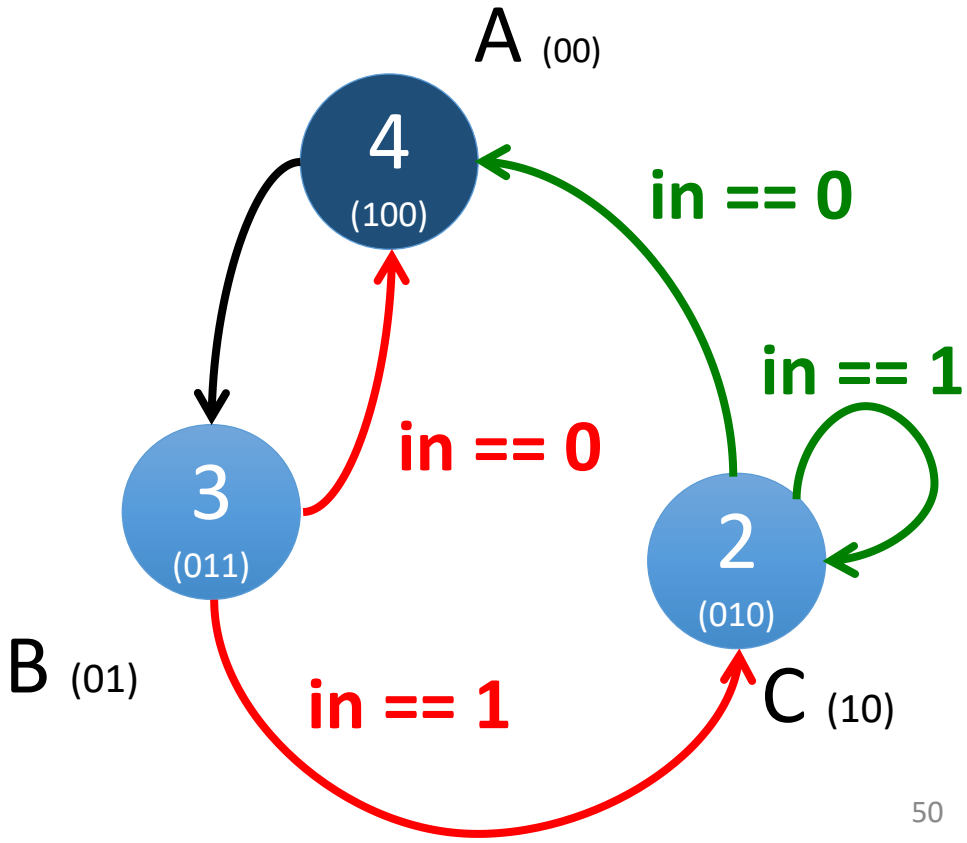
Output Function

s1	s0	o2	o1	o0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	0

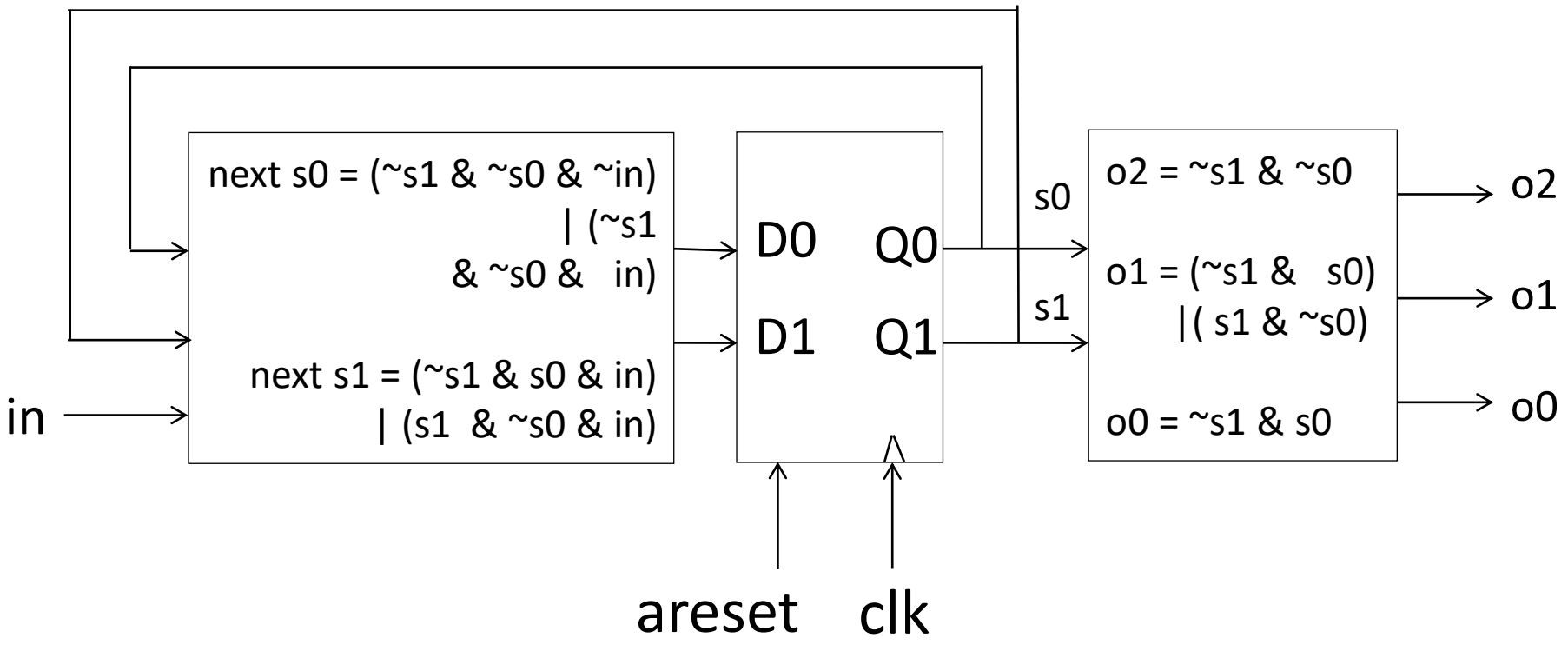
o2 = ~s1 & ~s0

o1 = (~s1 & s0) | (s1 & ~s0)

o0 = ~s1 & s0

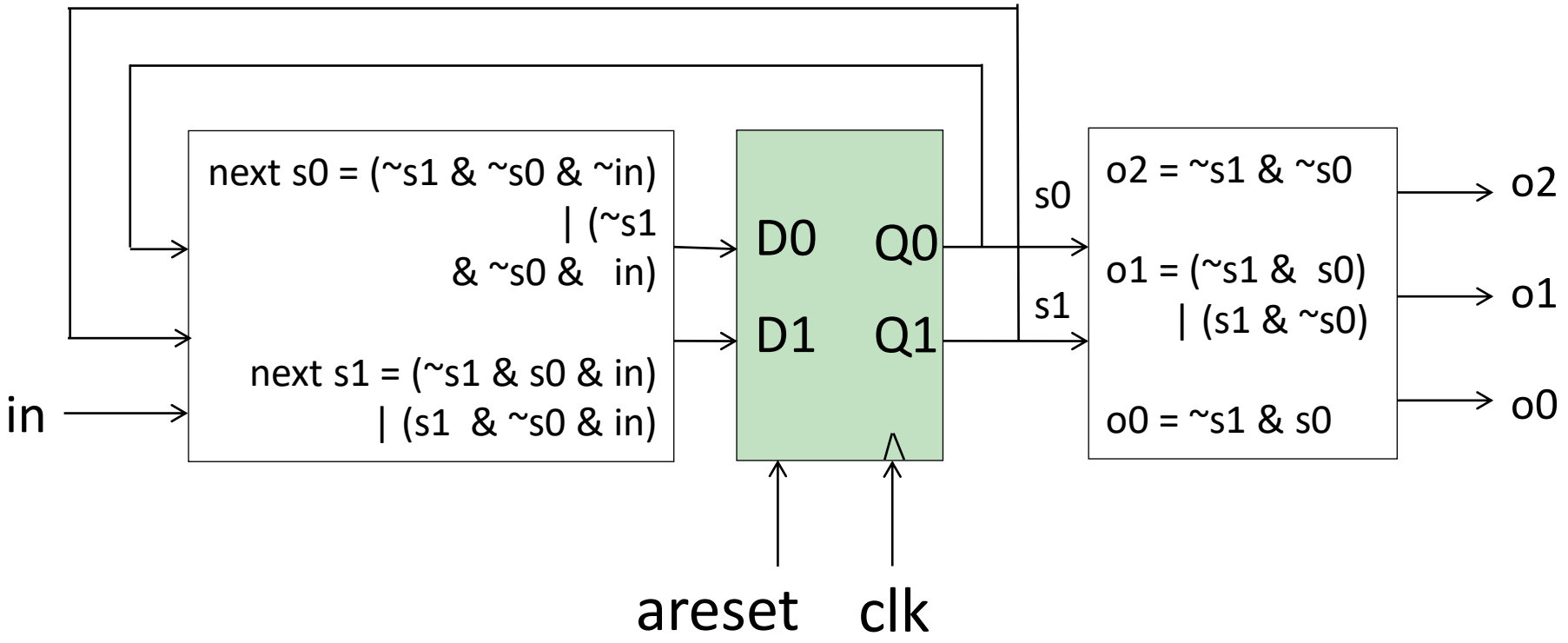


Structural diagram of the FSM



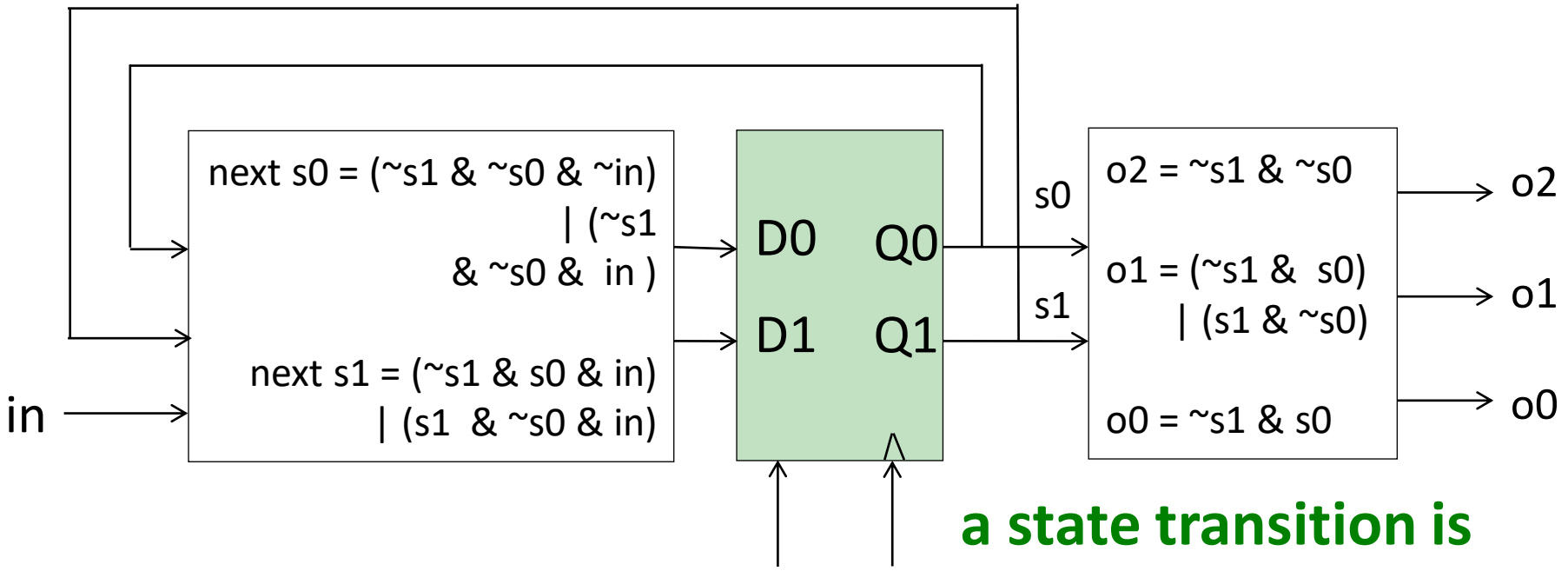
Essence of Moore Machines

the state is stored in a register



Essence of Moore Machines

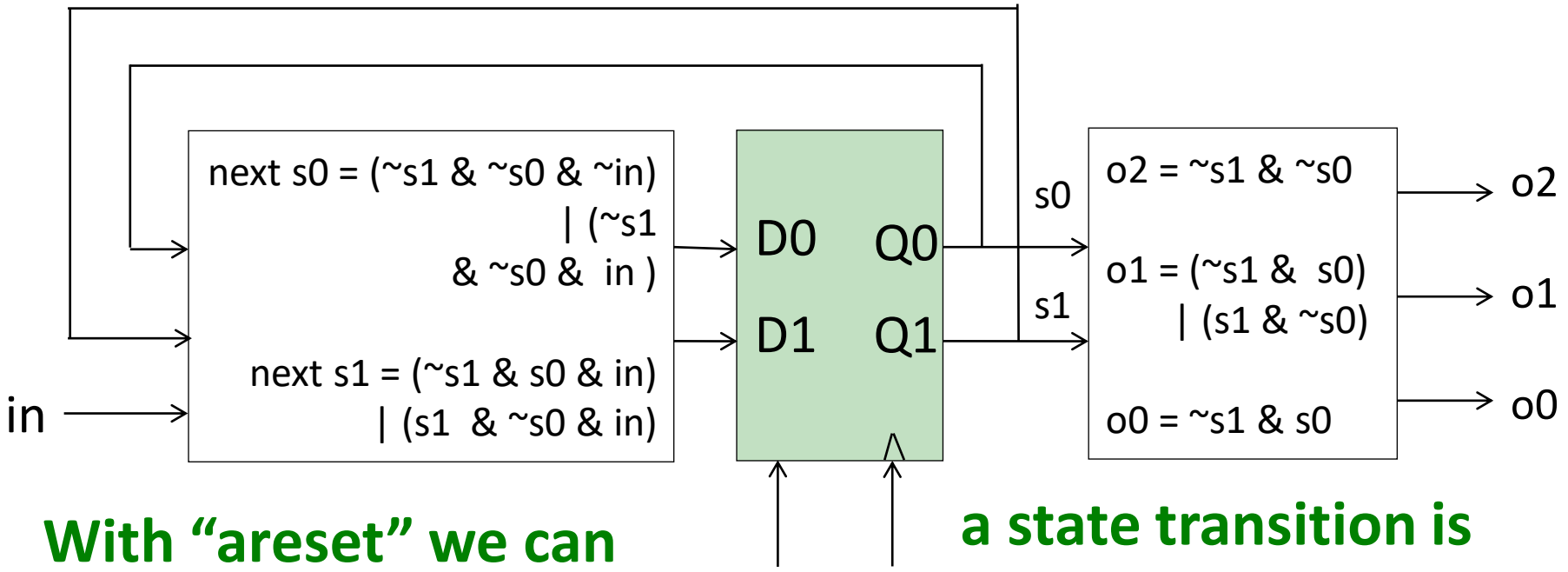
the state is stored in a register



a state transition is caused by the clock signal.

Essence of Moore Machines

the state is stored in a register

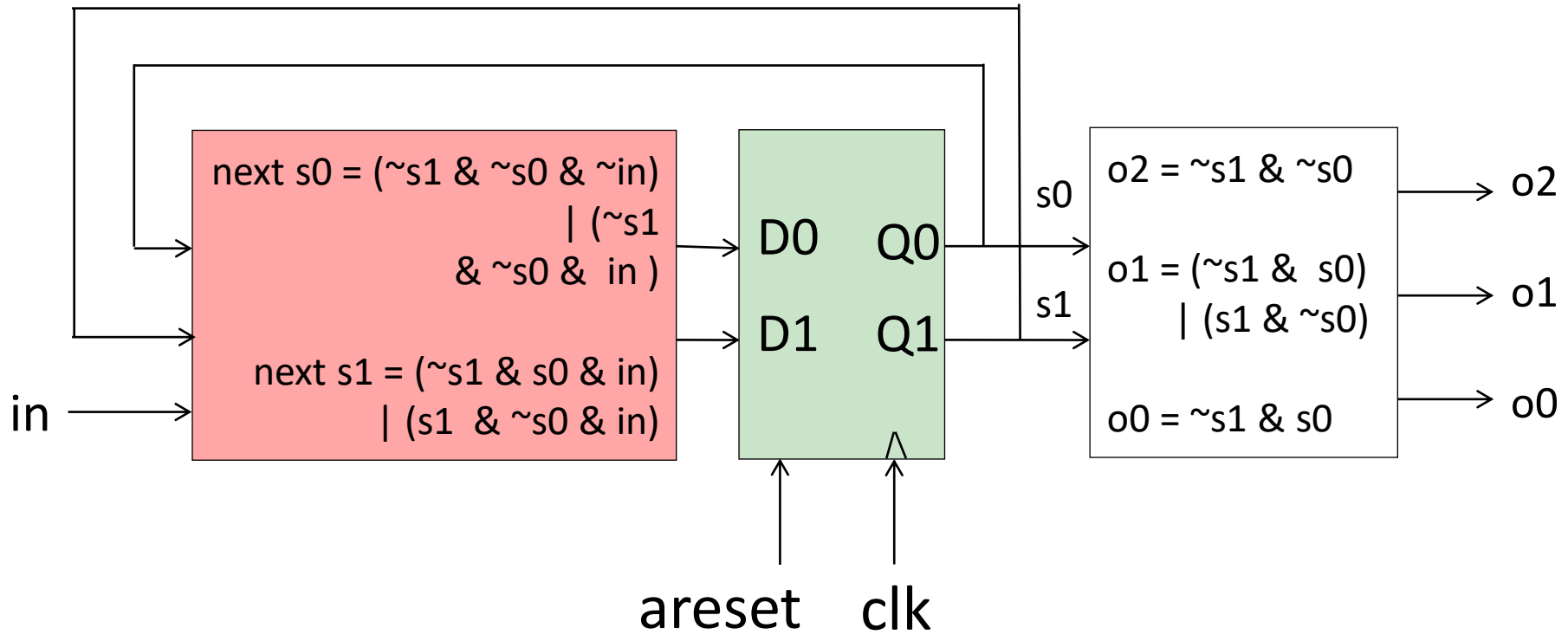


With "areset" we can initialize the ASM ("initial state").

a state transition is caused by the clock signal.

Essence of Moore Machines

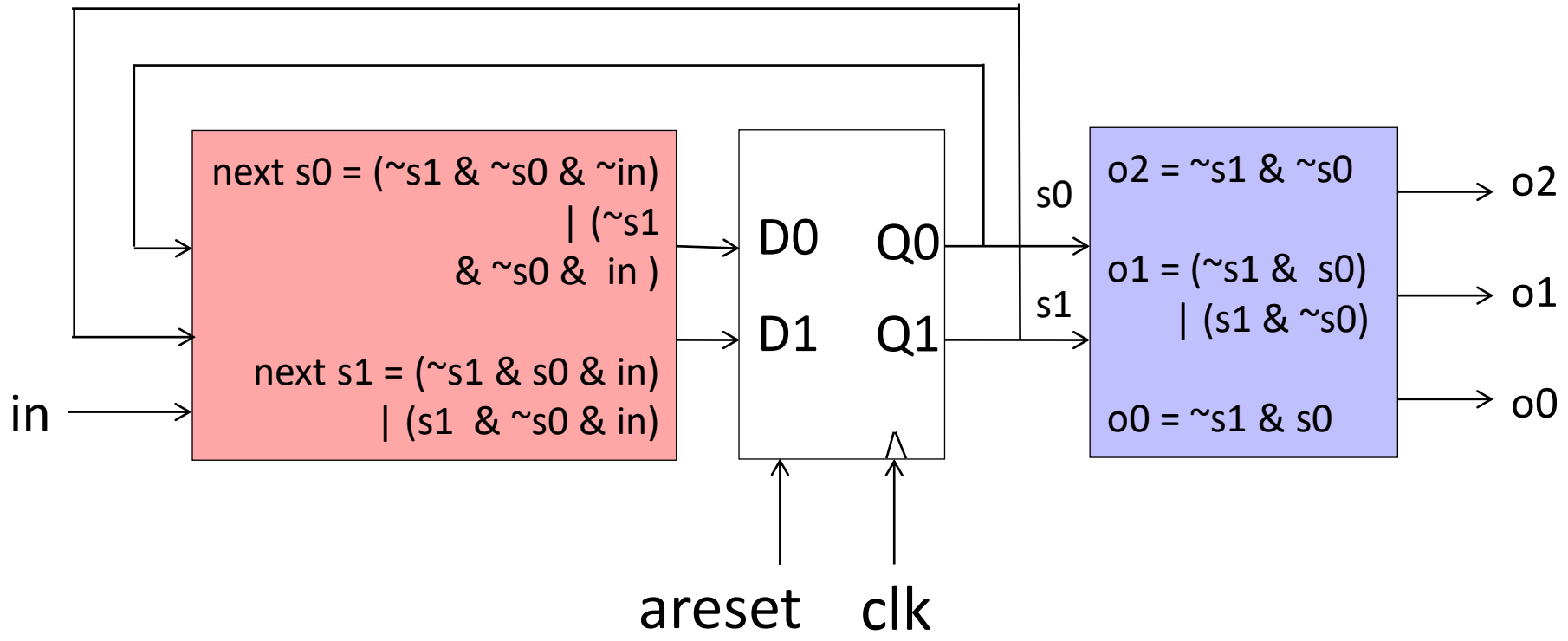
With the next-state function f
we compute the next state:
next state = f(state, input)



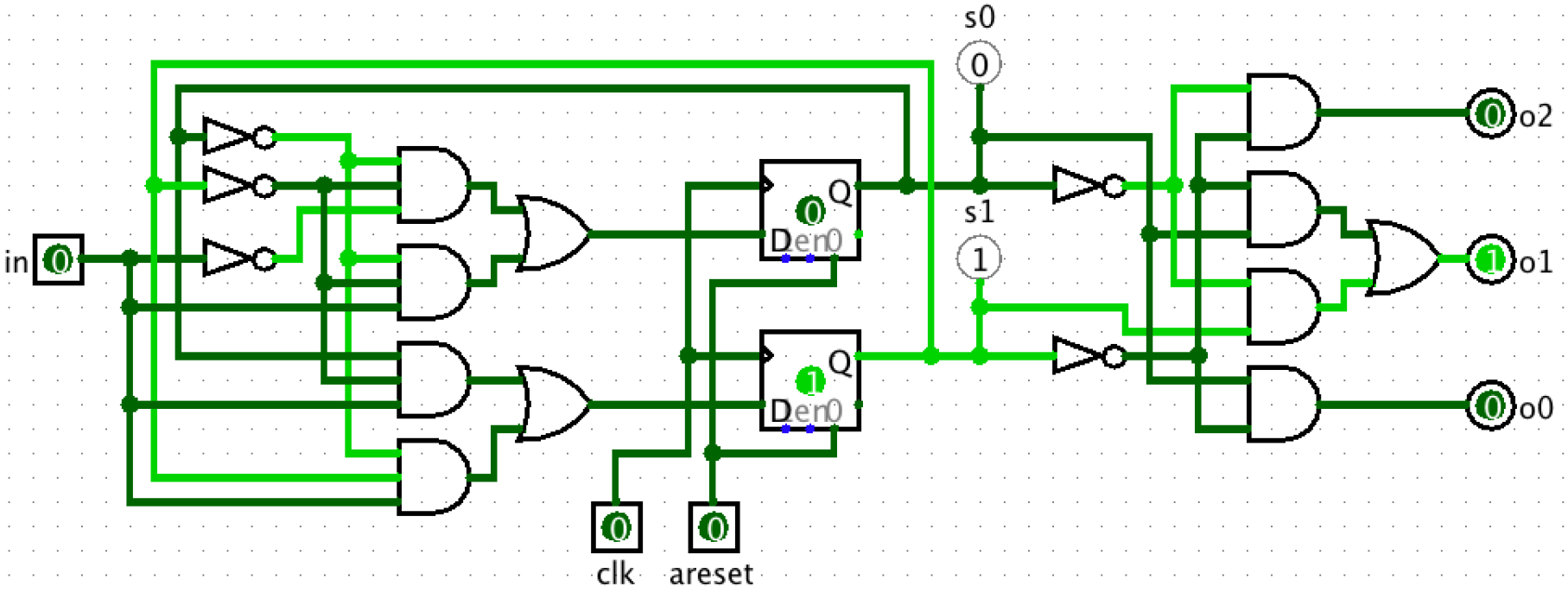
Essence of Moore Machines

With the output function we compute the output values:

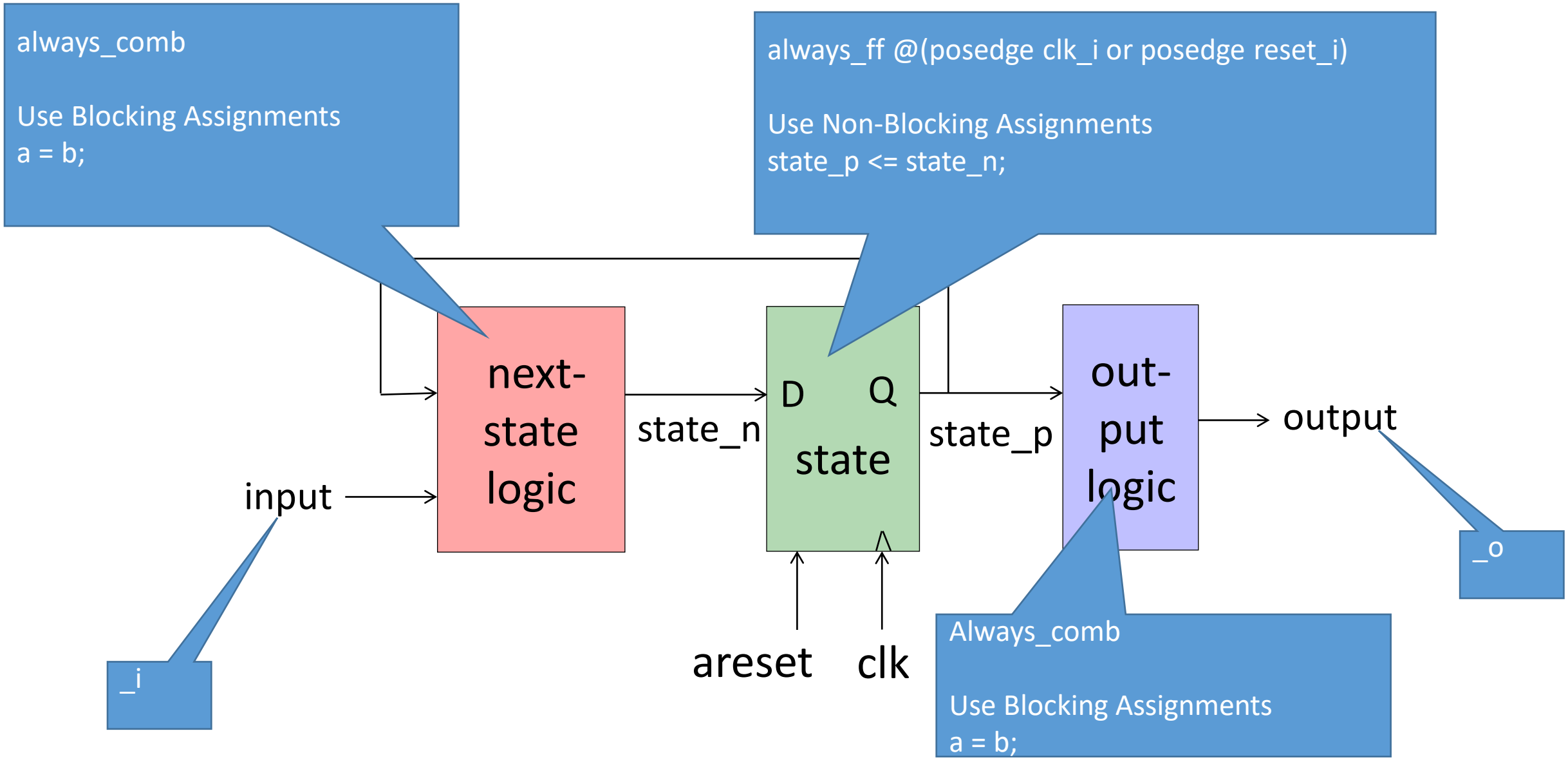
output = g(state)



Implementation with Digital

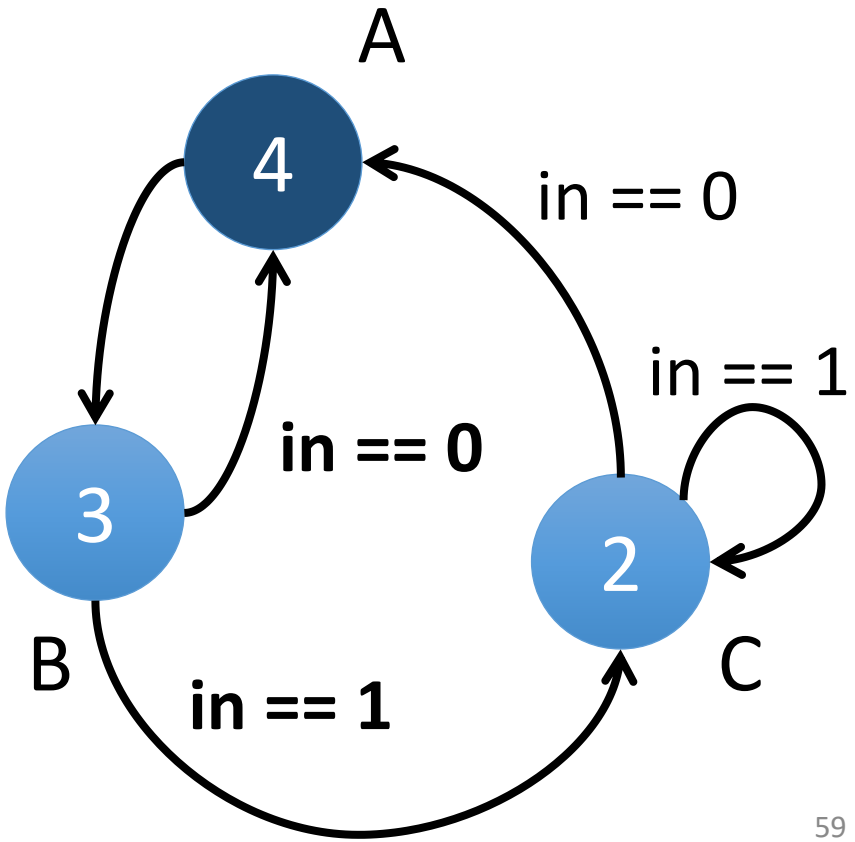


Coding Guidelines in SystemVerilog - Moore Machines



Modeling with SystemVerilog

See example con03.04_moore_fsm

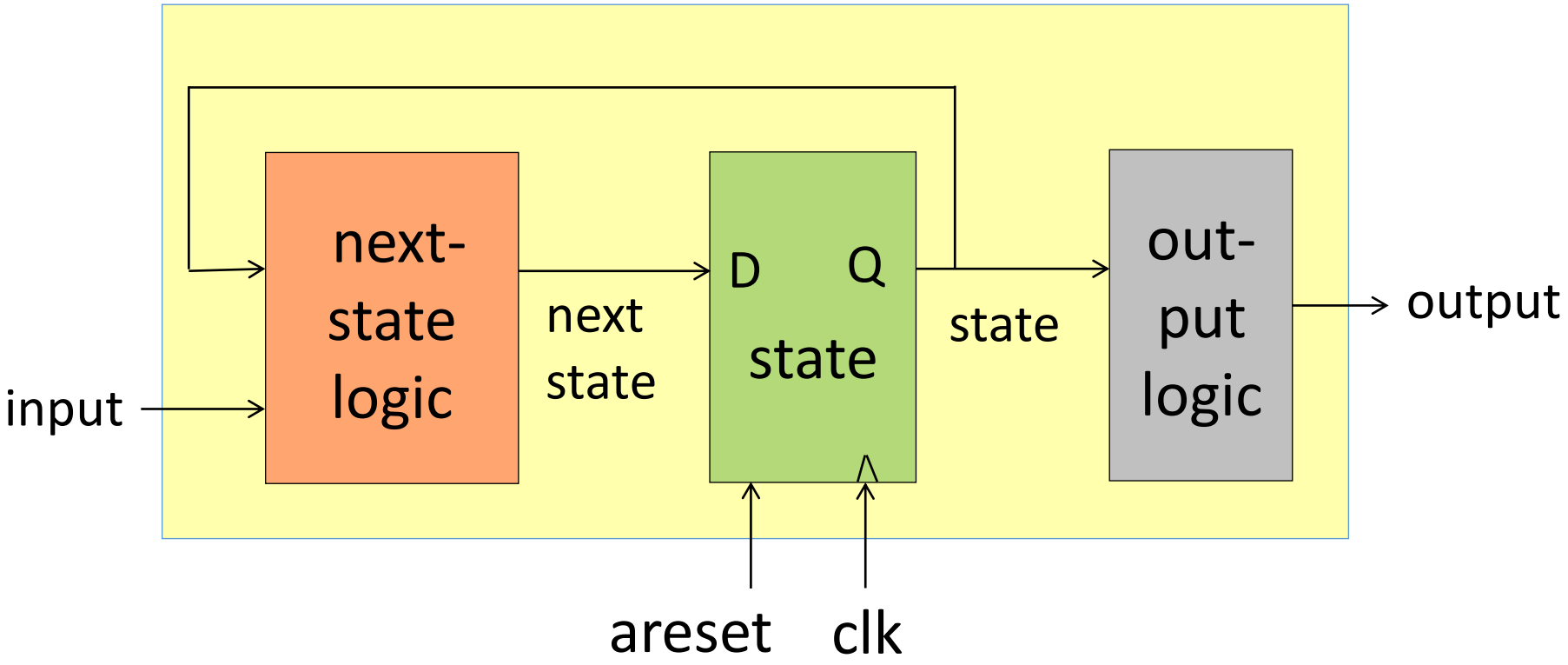


There exist 2 types of machines - check out the LITTLE but IMPORTANT difference

- Moore Machines
  - next state = function of present state and input
  - output = function of present state

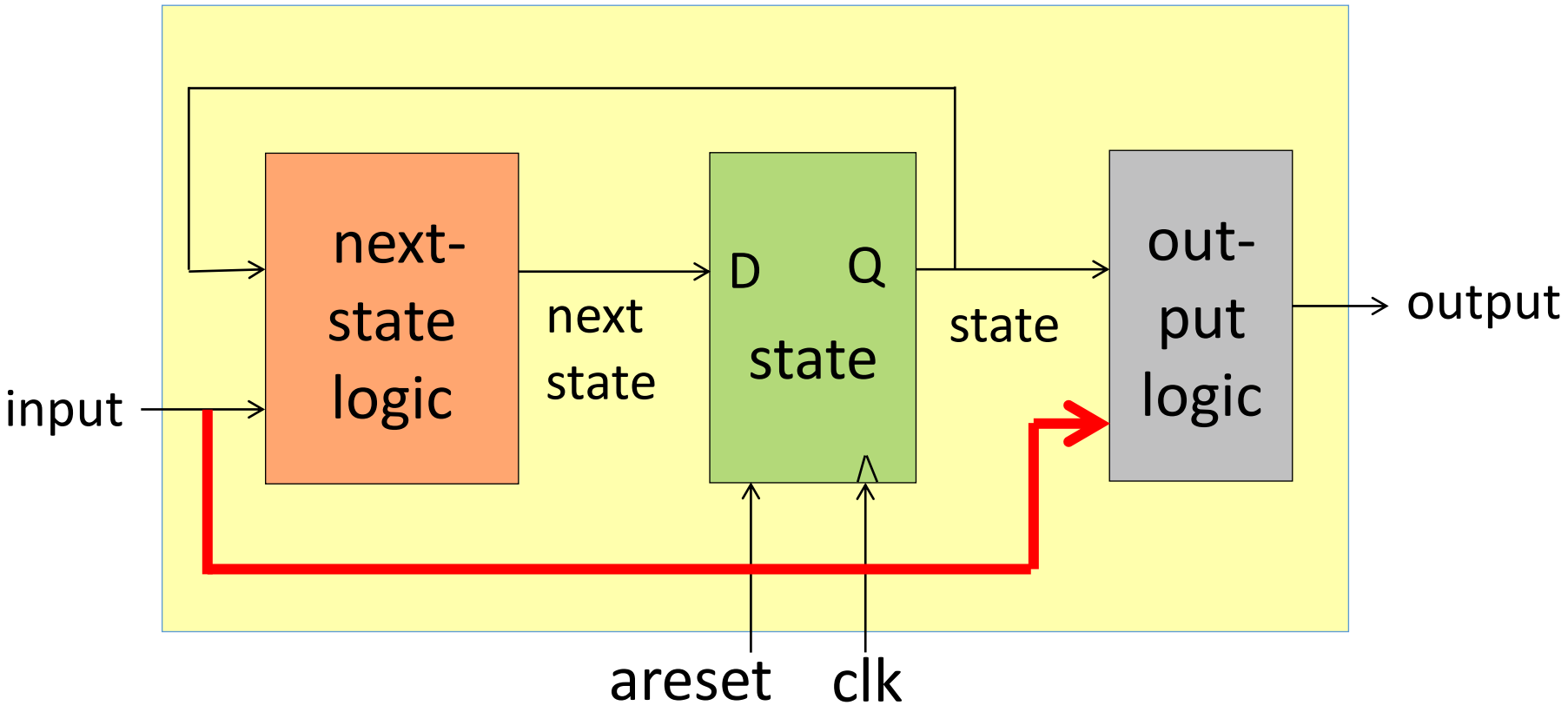
- Mealy Machines
  - next state = function of present state and input
  - output = function of present state **and input**

Essence of Moore Machines



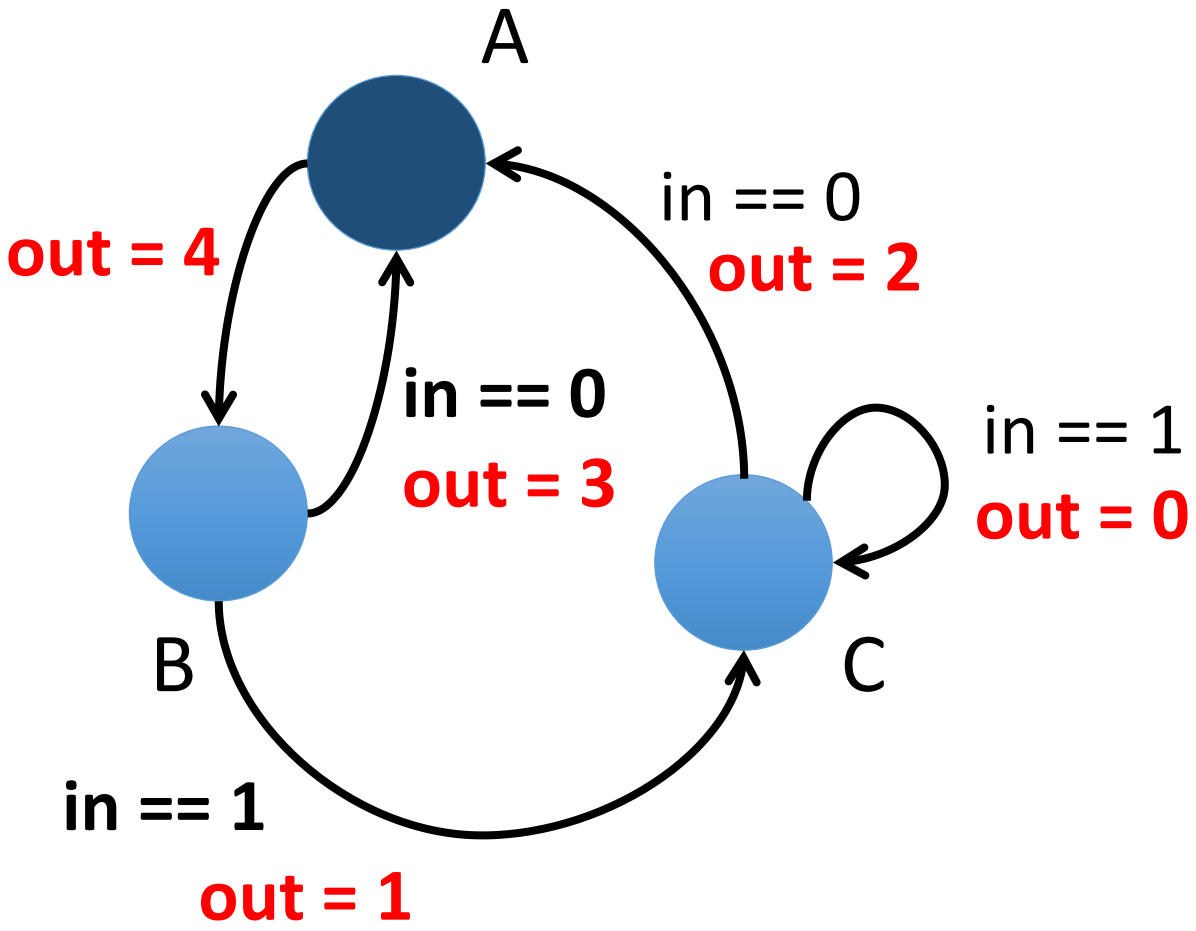
Essence of Mealy Machines

output = g(state, input)



An example for a Mealy Machine

We write the **output values** next to the transition arrows, since the output depends not only on the state, but can also depend on the input.

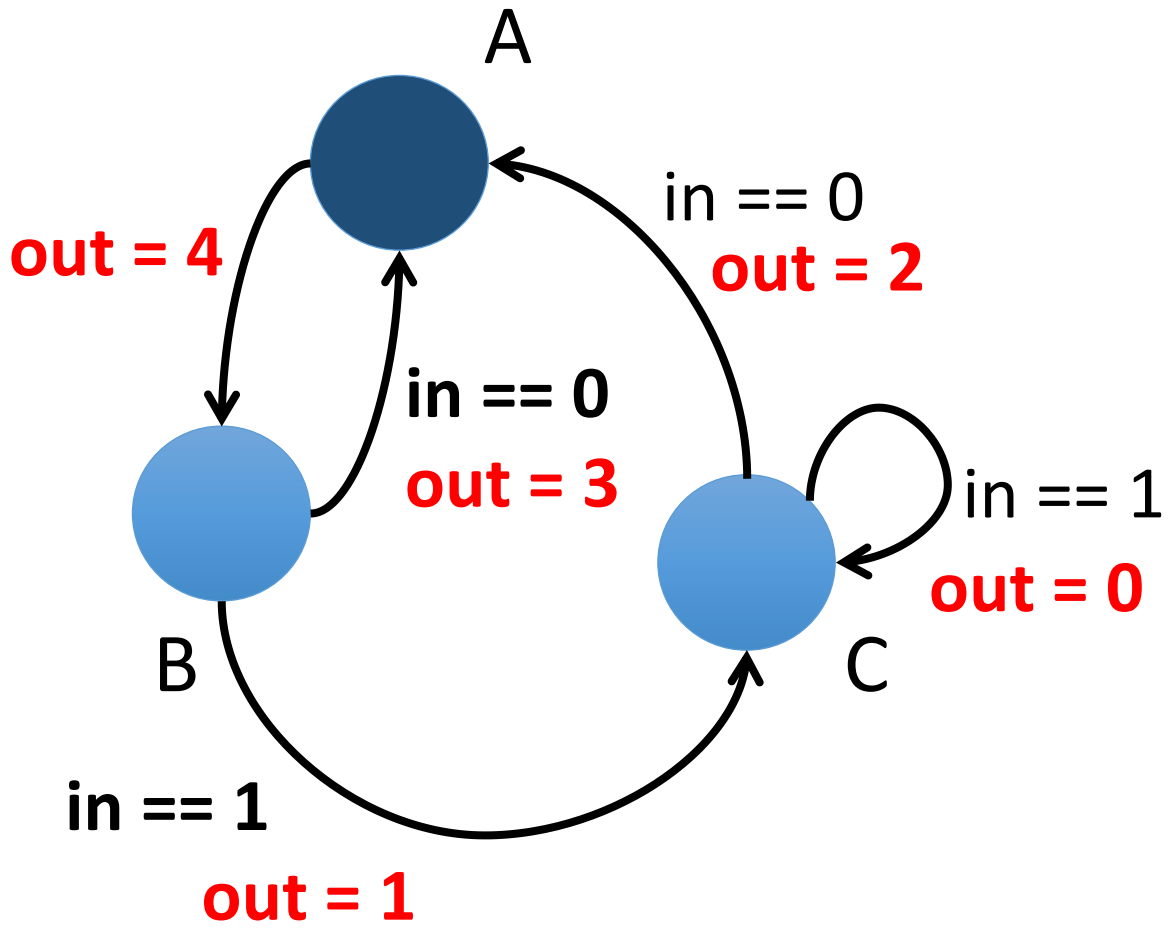


The output function

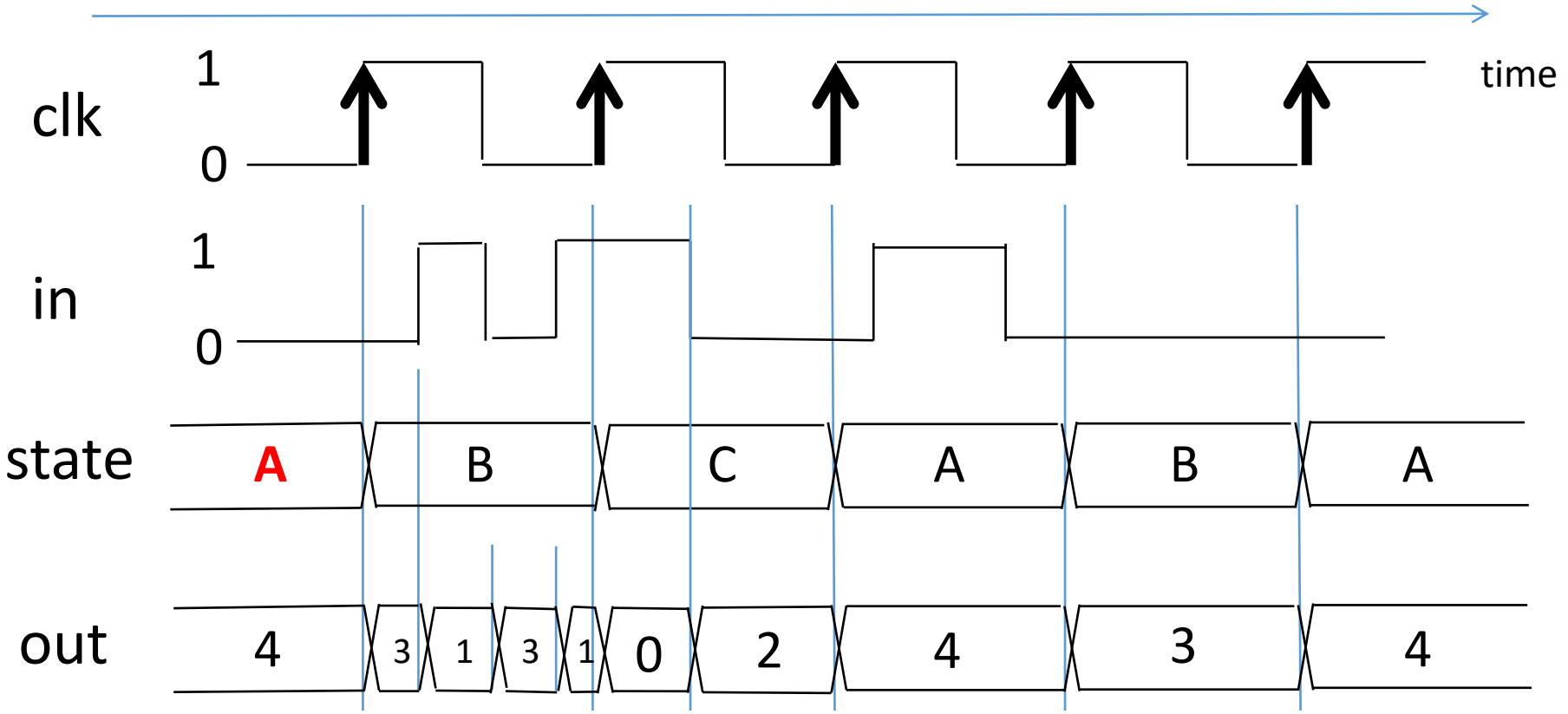
The output function can be derived from the state diagram.

output = g(state, input)

state	in	output
A	0	4
A	1	4
B	0	3
B	1	1
C	0	2
C	1	0



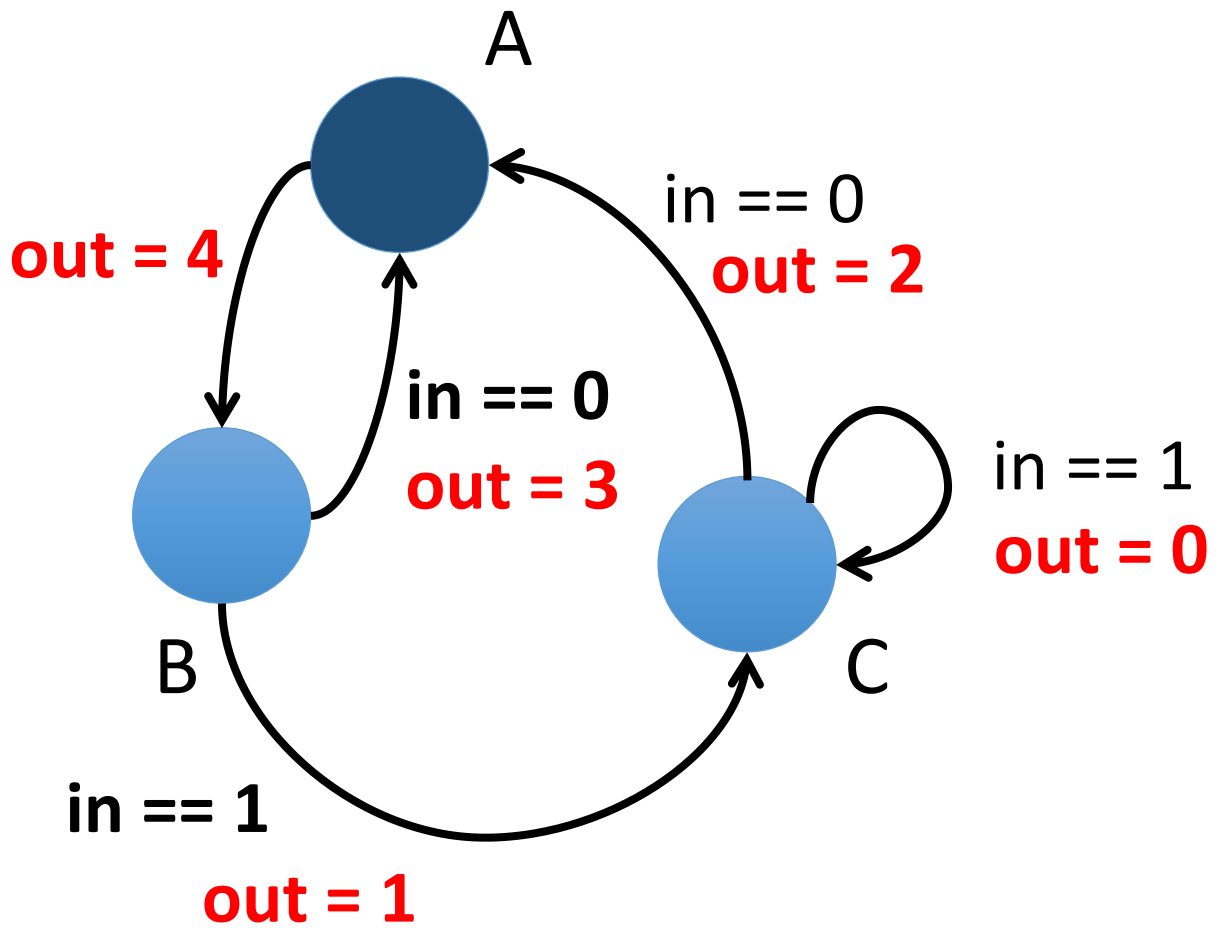
Timing diagram



Note how the value of “in” immediately influences the value of “out”.

Modeling with SystemVerilog

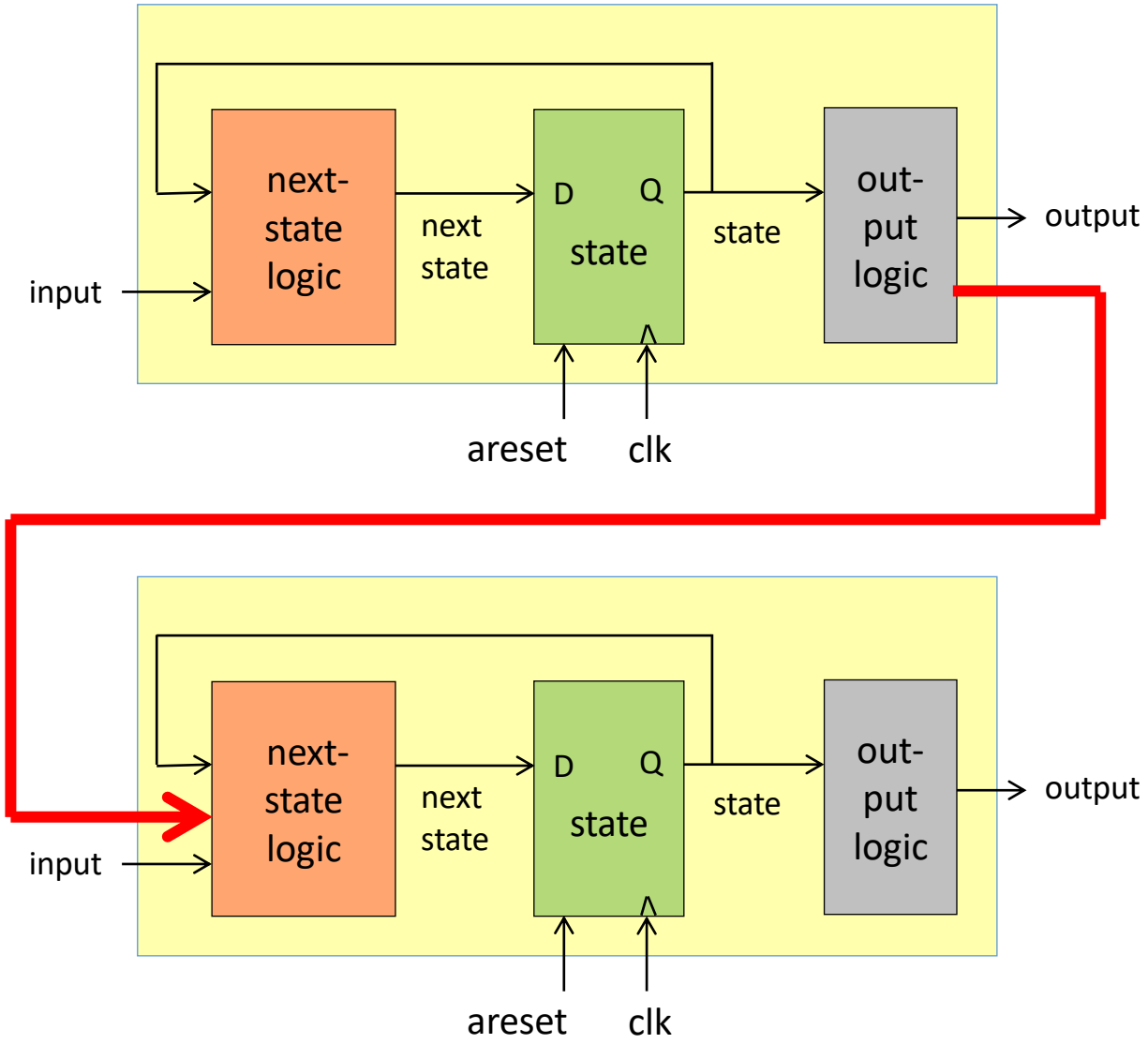
See example con03.05_mealy_fsm



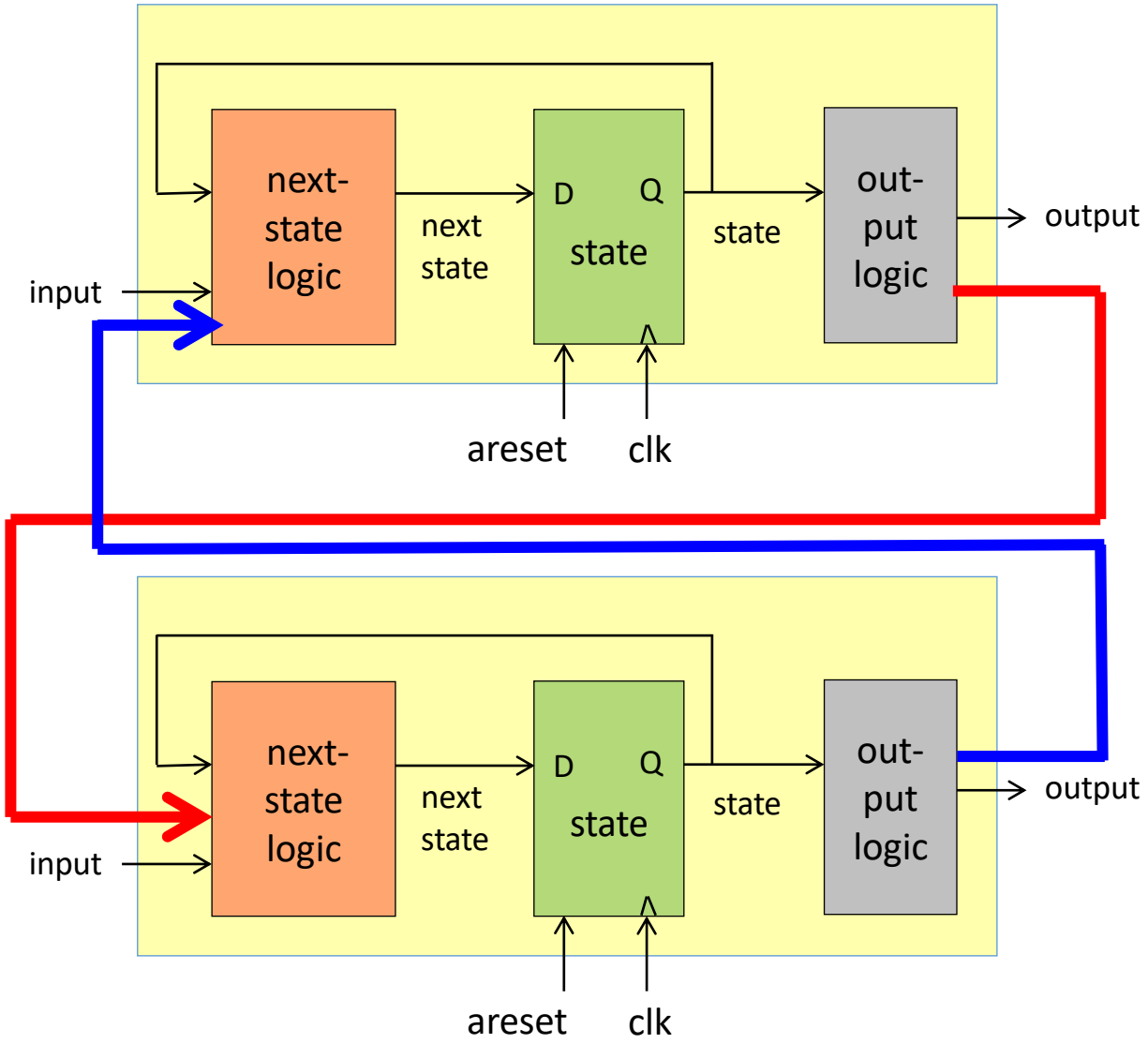
We can combine machines

- Combining Moore Machines causes no problem. We get another Moore Machine.
- Combining a Moore Machine with a Mealy Machine causes also no problem. We get a Moore Machine or a Mealy Machine.
- Combining two Mealy Machines can cause troubles: **One needs to avoid combinational loops**!

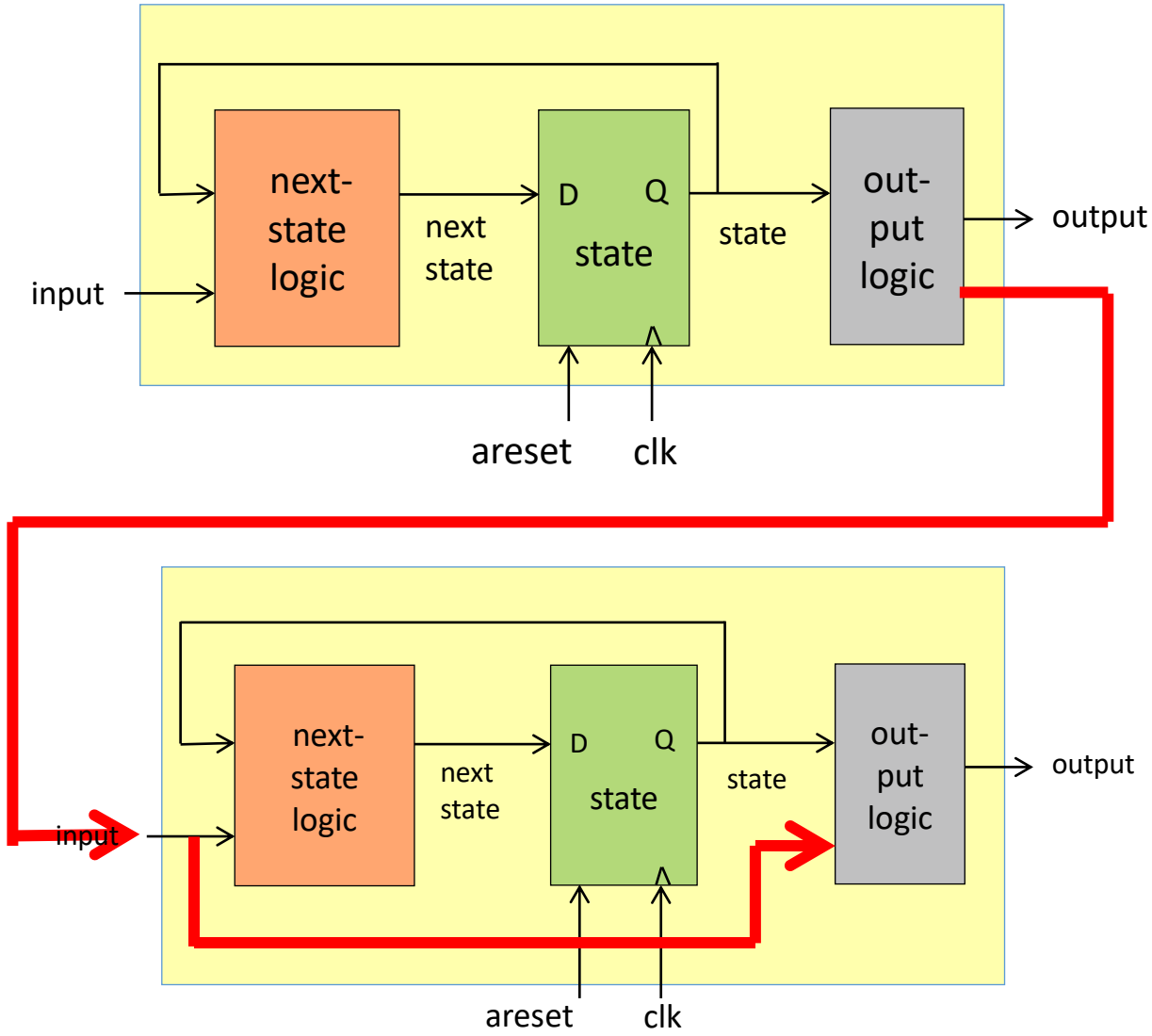
The combination of two Moore Machines creates again a (more complex) Moore Machine



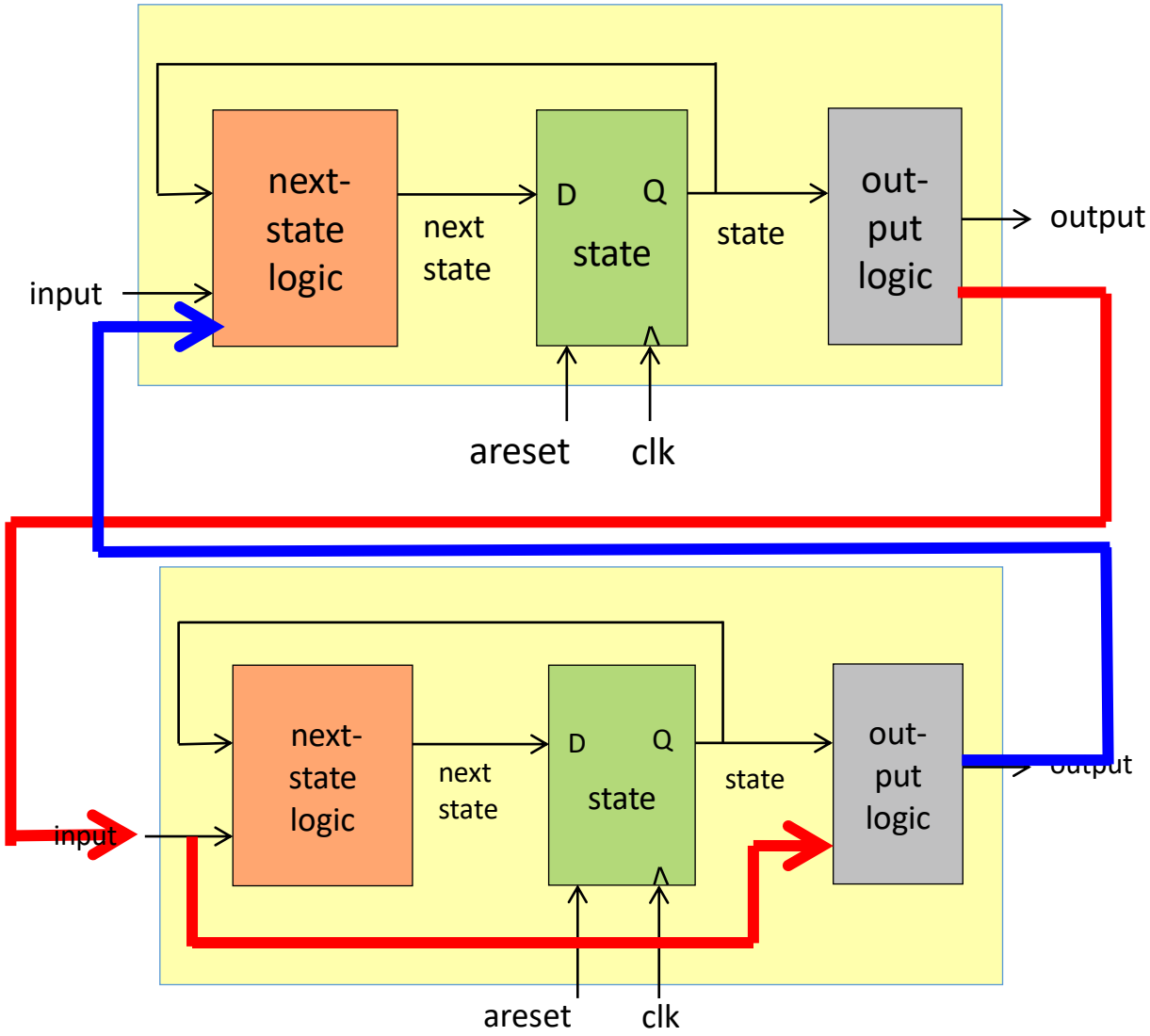
We can even connect More Machines in a loop-like fashion



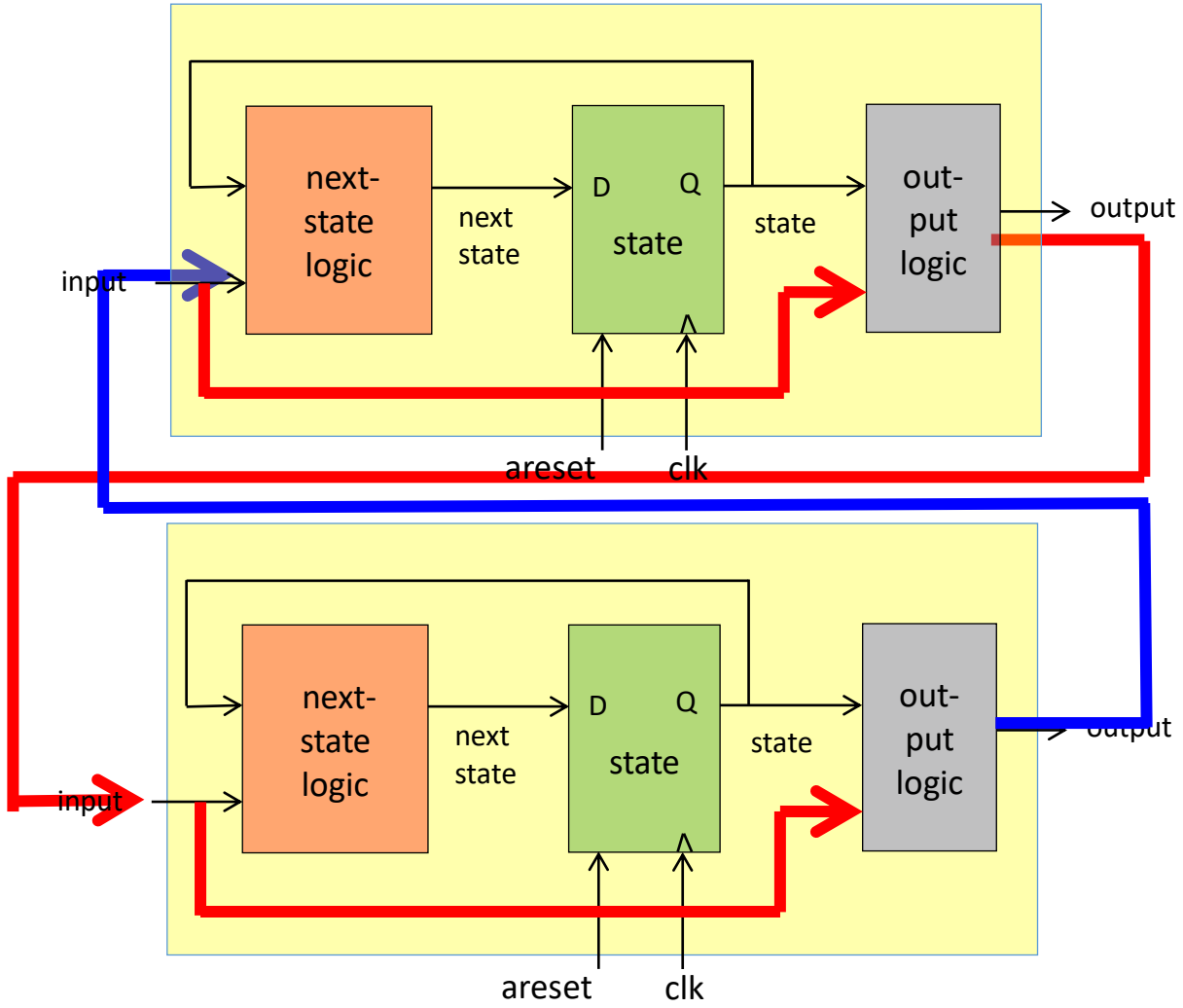
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine



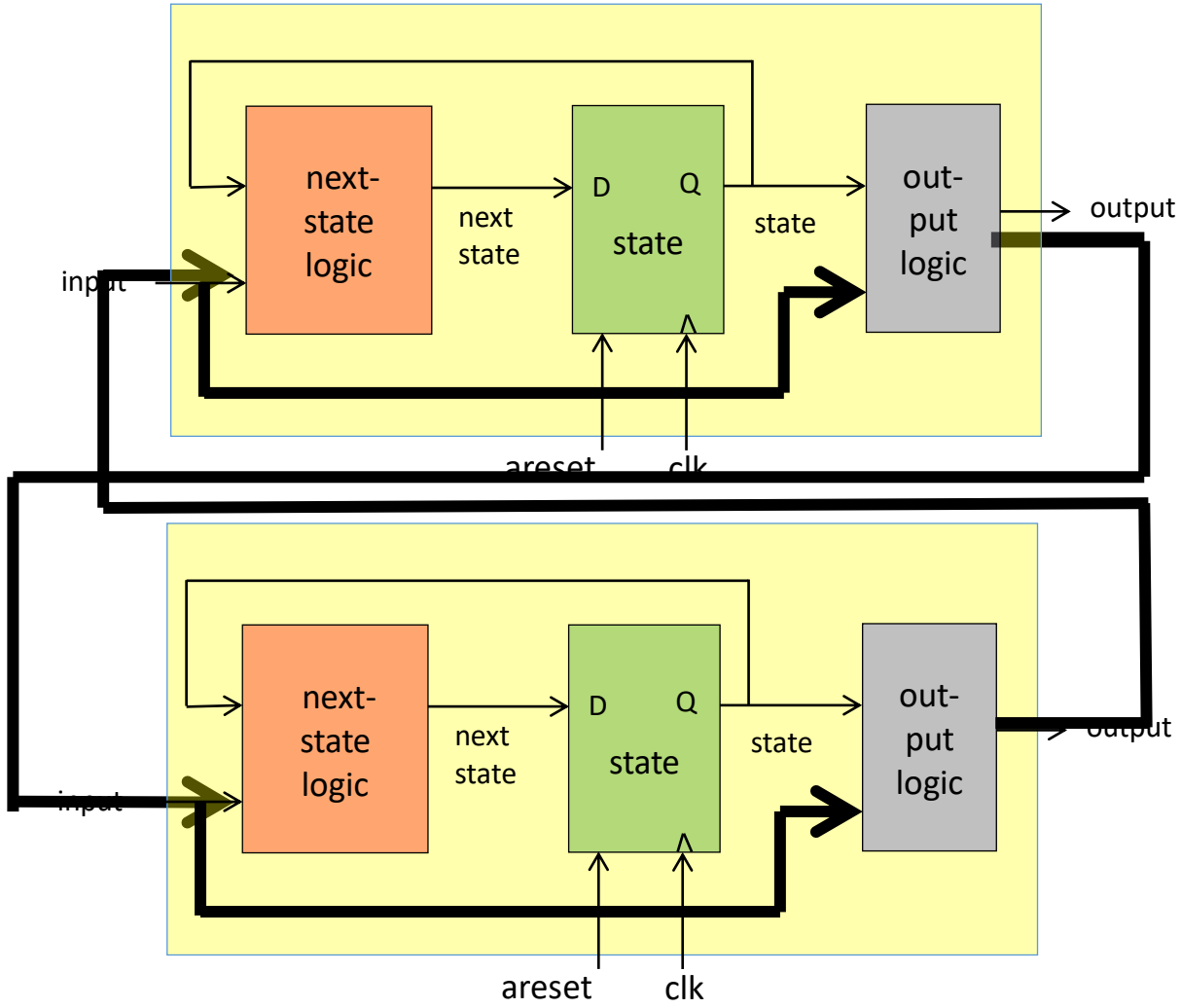
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine



The combination of two Mealy Machines is “dangerous”: You need to avoid “combinational loops”



The combination of two Mealy Machines is “dangerous”: You need to avoid “combinational loops”



Summary

Not for every circuit the modelling as Moore/Mealy Machine is the best.

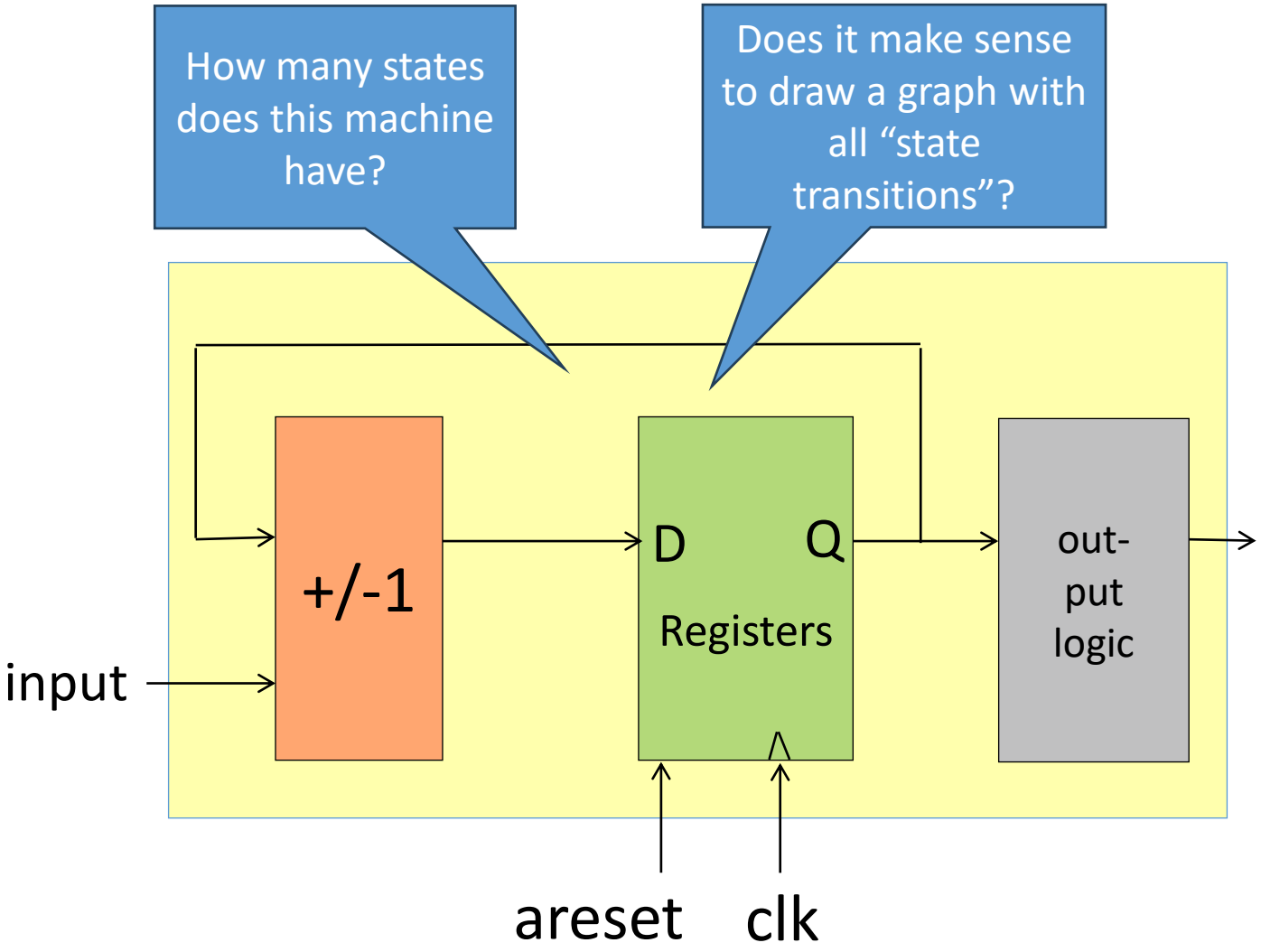
- All digital logic can in principle be built with Moore Machines and Mealy Machines.

- You go from one representation to the other (in both directions from each representation to each other):

State Diagram – Timing Diagram – SystemVerilog Code – Circuit Diagram

Separating Control and Data Path

Rembember our AddSub Machine




```
logic [3:0] value_p, value_n;

// sequential logic
always_ff @(posedge clk_i or posedge reset_i) begin
    value_p <= value_n;
    if (reset_i == 1) begin
        value_p <= 0;
    end
end

// combinational logic
always_comb begin
    value_n = value_p + 1;
    if (addsub_i == 1) begin
        value_n = value_p - 1;
    end

end

// combinational logic / output logic
always_comb begin
    out_o = value_p;
end
```

Our AddSub Machine is a Datapath

- A Datapath “contains everything that is related to data processing”
  - consists of
    - Functional units doing computations
    - Data registers
    - Wires and multiplexers connecting the registers und functional units
  - takes as Inputs
    - Control signals defining “which computations are performed and where data is stored”
  - provides as outputs
    - Data values and derived data values (e.g. overflow, underflow, result of compares, …)

→ Often visualized based on block diagrams; state diagrams are not suitable

Control Units are the Counterpart

- A Control Unit/Controller/FSM “contains everything that is related managing what happens when and under what condition”
  - consists of
    - Next-state logic
    - State registers
    - Wires connecting the nextstate logic and registers
  - takes as inputs
    - Control signals and status flags (system conditions)
  - provides as outputs
    - Control signals connected to a data path taking them as input

→ Best visualized based on state diagrams

Control Unit

- State machine generating control signals for the data path

“Piano Player”

Data Path

- Contains all functional units and registers related to data processing
- Receives control signals to perform operations on the data.
- Provides status signals to the control-related data to the control unit

“Piano”

Mapping to SystemVerilog

Control Unit

- **Always_comb** Block describing nextstate logic
- **Always_FF** Block describing the state registers

Data Path

- **Always_comb** Block describing the functional units of the data path
- **Always_FF** Block describing the data registers

Depending on the implemented system, it is useful to implement the “always blocks” of control and data separate or together

IMPORTANT Announcement: No points will be reduced in Task 1 if there is a joint always_comb and a joint always_FF block for control and data

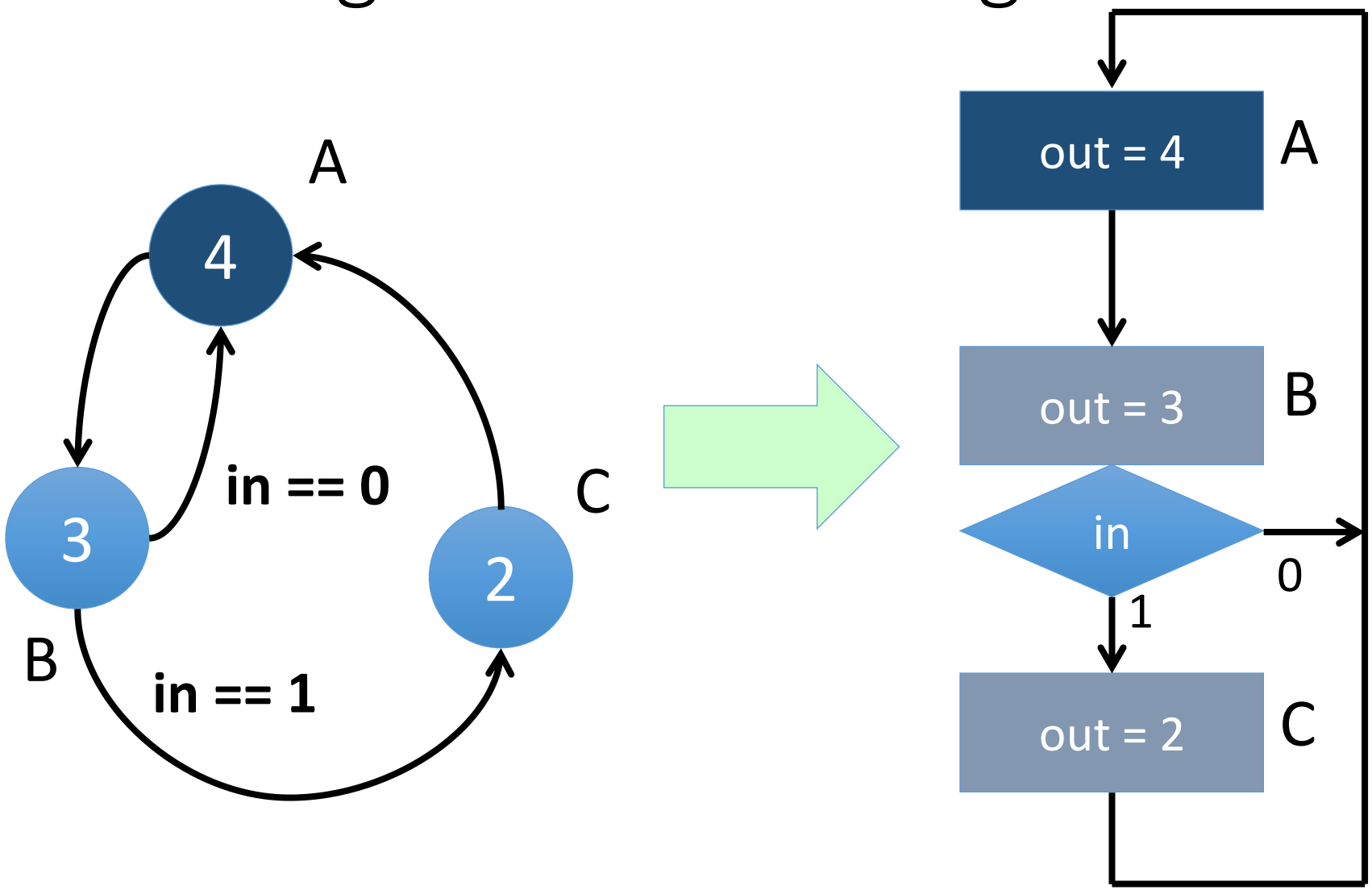
Algorithmic State Machines

(Modelling Control and Data Path in a Single Graph)

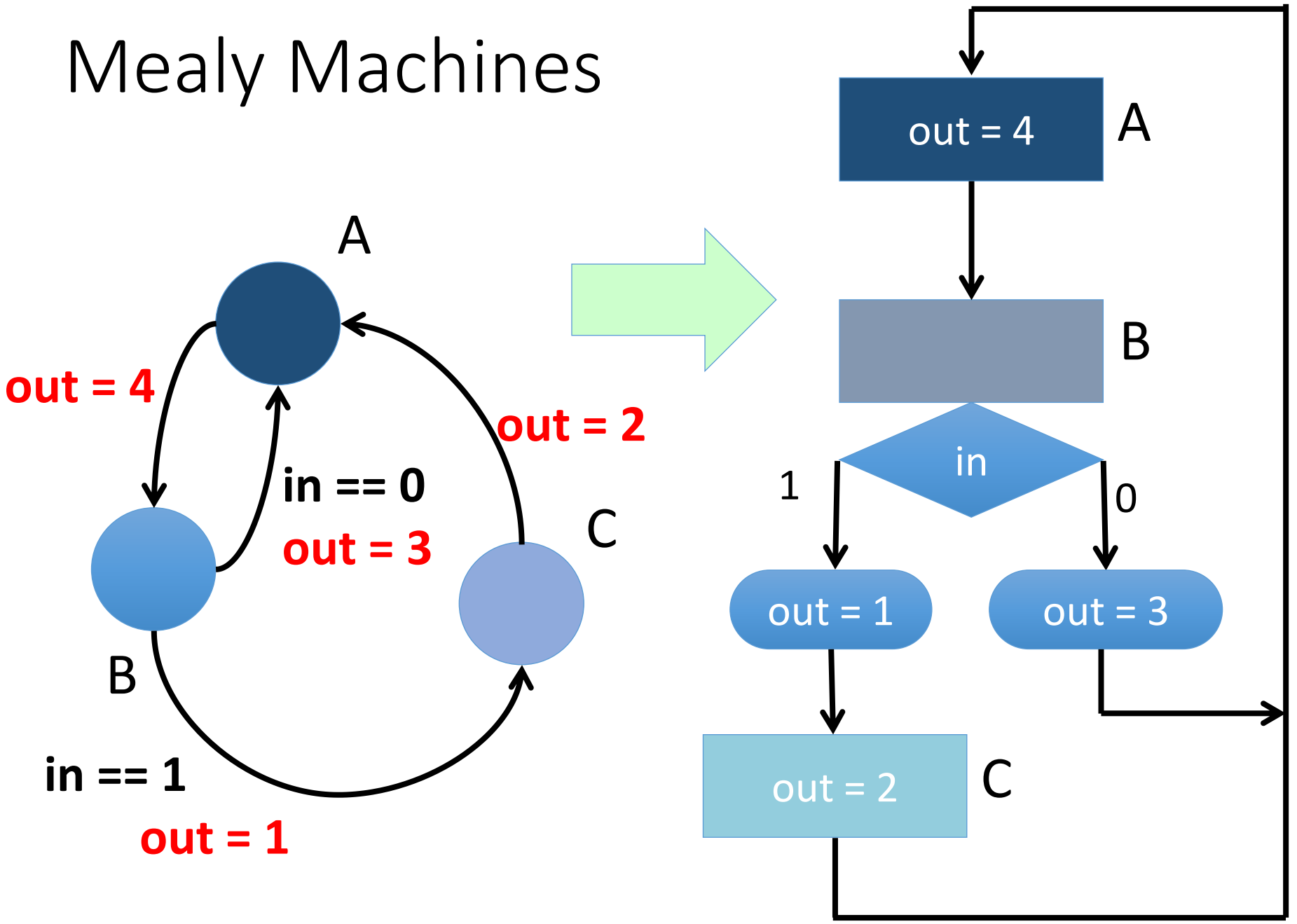
Algorithmic State Machines (ASMs)

- ASMs are a useful extension to finite state machines
- ASMs allow to specify a system consisting of a data path together with its control logic
- All FSM state diagrams have an equivalent ASM diagram

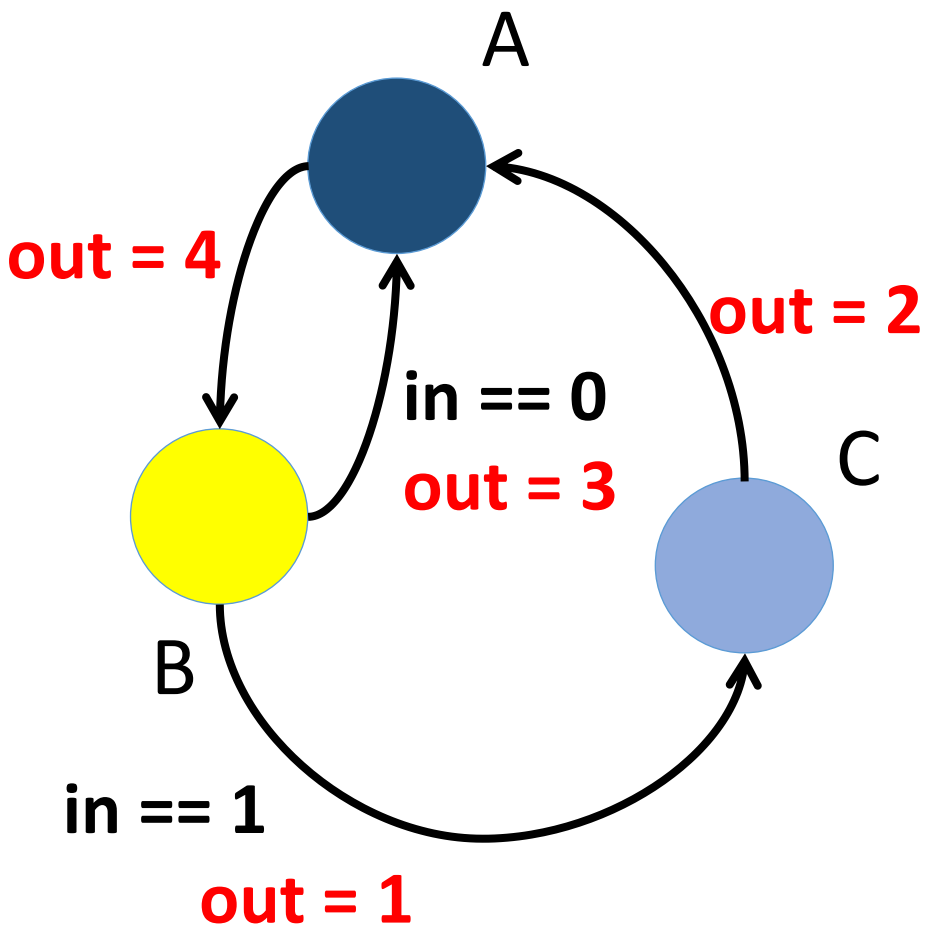
FSM state diagram → ASM diagrams

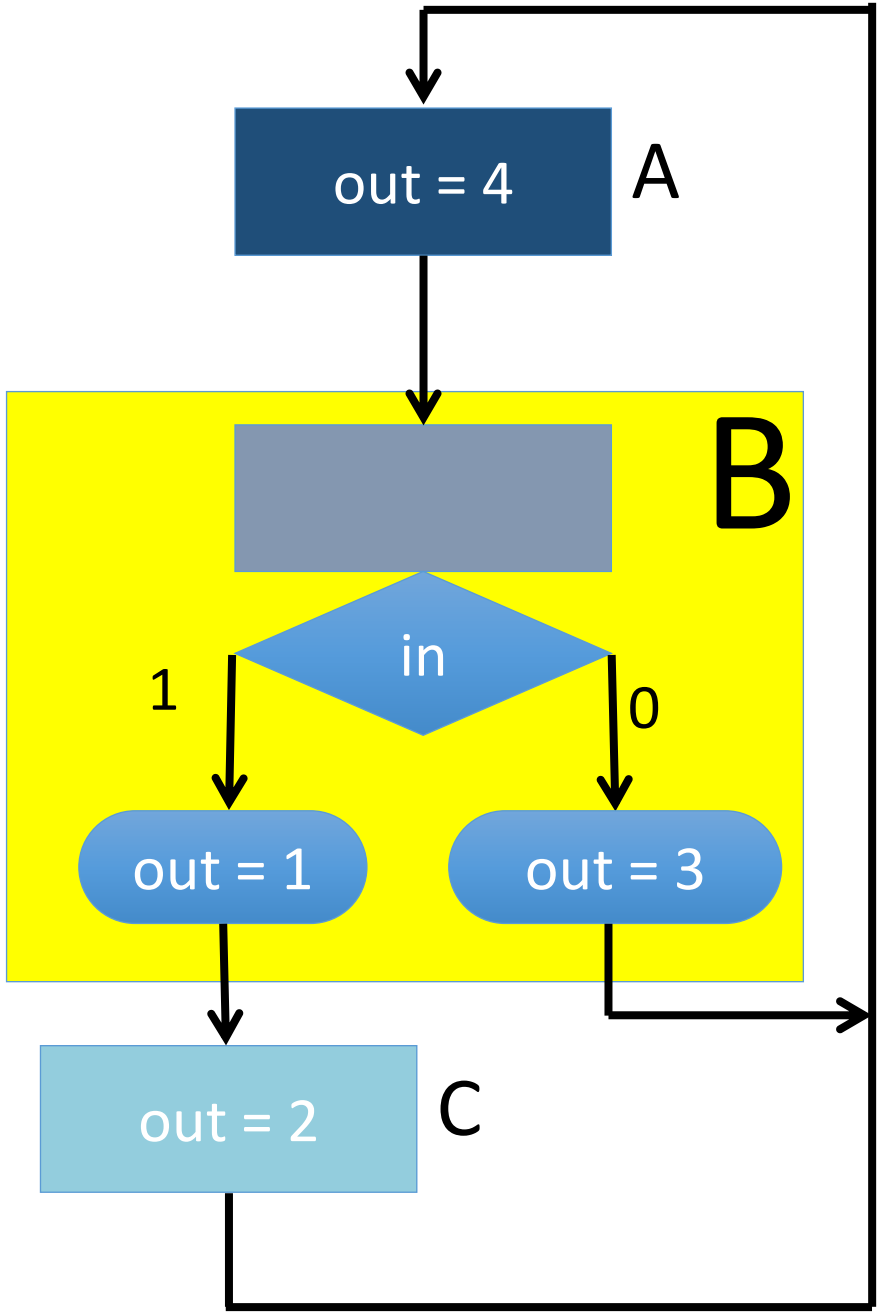


Mealy Machines



Mealy Machines





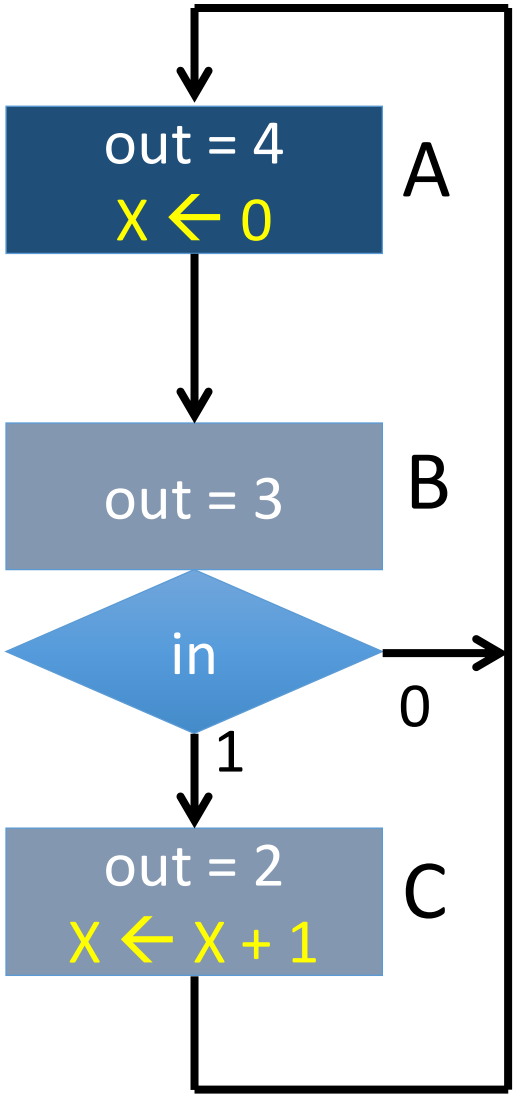
Adding the Datapath Using Register-Transfer Statements

- Register-transfer statements define the change of a value stored in a register of the data path
- Values in registers can only change at the active (= rising) edge of clock.
- “Register-transfer statements” with a “left arrow” (“←”)
- Example: “a ← x” means that the value in the register “a” gets the value of “x” at the “next” active (= rising) edge of clock.

“=“ versus “←” in an ASM

- The symbols “=“ versus “←” in an ASM align with the symbols “=“ and “<=“ in Systemverilog
  - With the equal sign (“=“) we denote that the output of the FSM has a certain value during a particular state (SystemVerilog: output logic – combinational)
  - With the left-arrow (“←”) we denote a register-transfer statement: The register value left of the arrow changes to whatever is defined right of the arrow upon the next active (= rising) edge of clock.

ASM Diagram With two Register-Transfer Statements

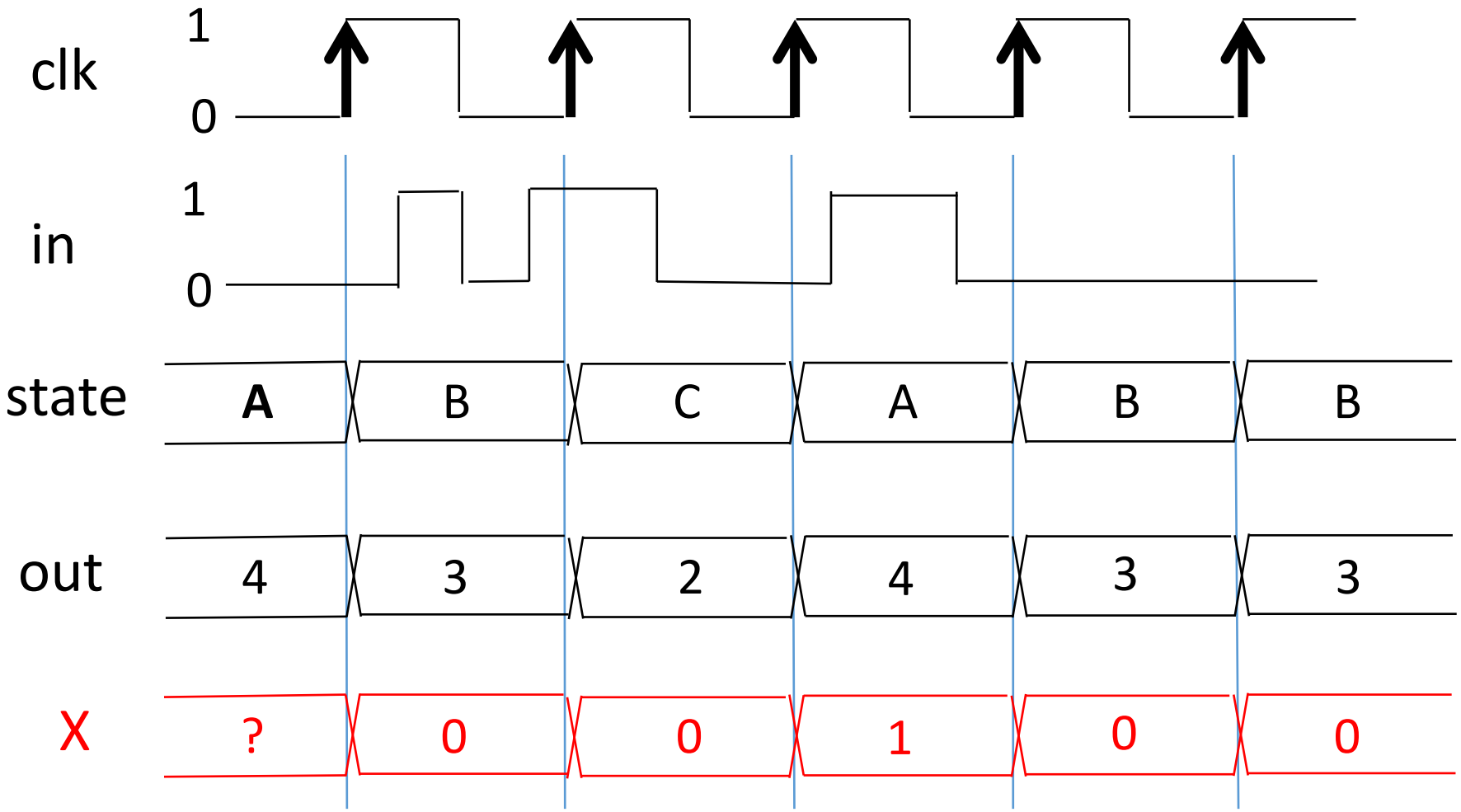


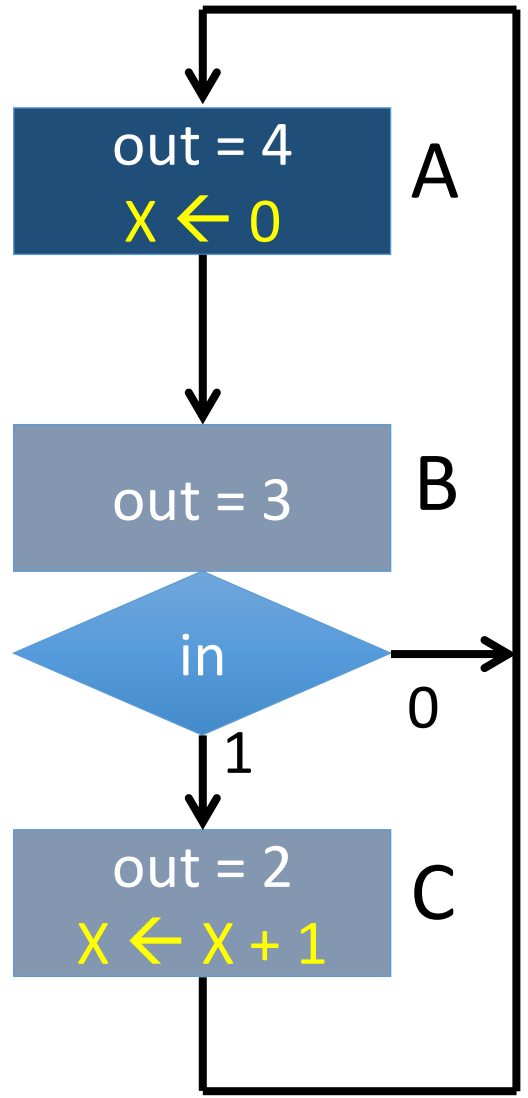
The value stored in register X gets 0 at the state transition from state A to state B.

The value in register X does not change upon leaving state B.

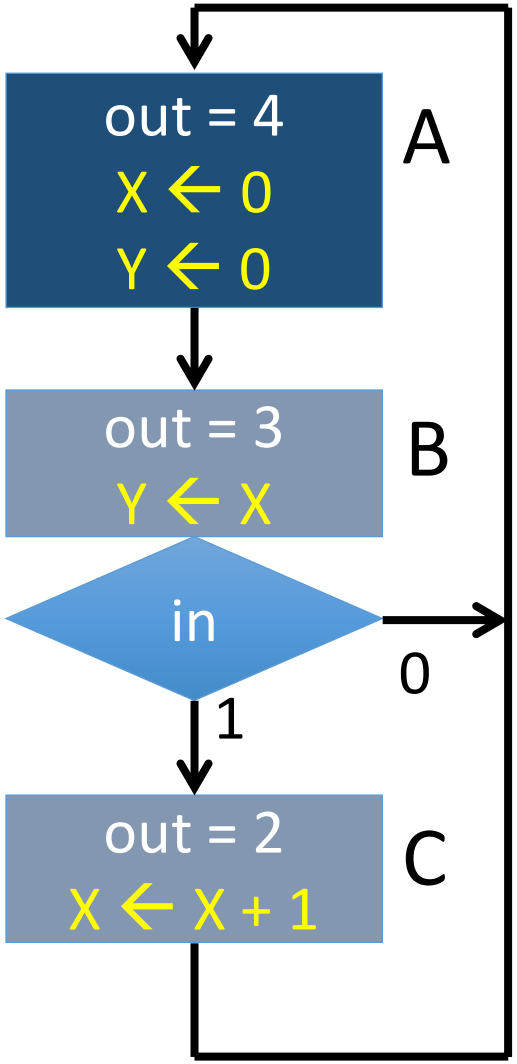
The value in register X gets incremented at the state transition following state C.

Timing diagram





Several register-transfer statements can be specified within one state

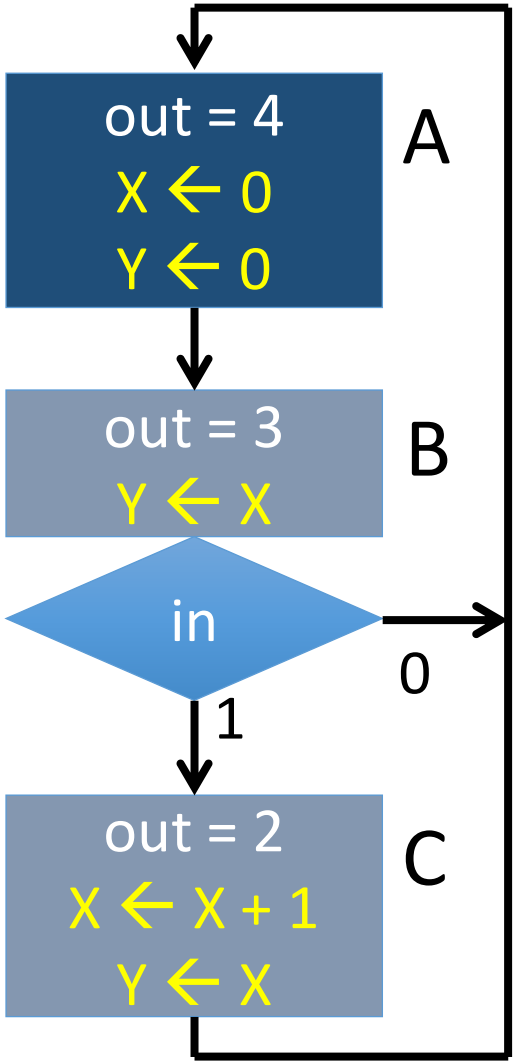


The values stored in register X and register Y become 0 at the state transition from state A to state B.

The value in register X does not change upon leaving state B. The value stored in register Y gets the value of register X upon leaving state B.

The value in register X gets incremented at the state transition following state C.

Several register-transfer statements can be specified within one state



The values stored in register X and register Y become 0 at the state transition from state A to state B.

The value in register X does not change upon leaving state B. The value stored in register Y gets the value of register X upon leaving state B.

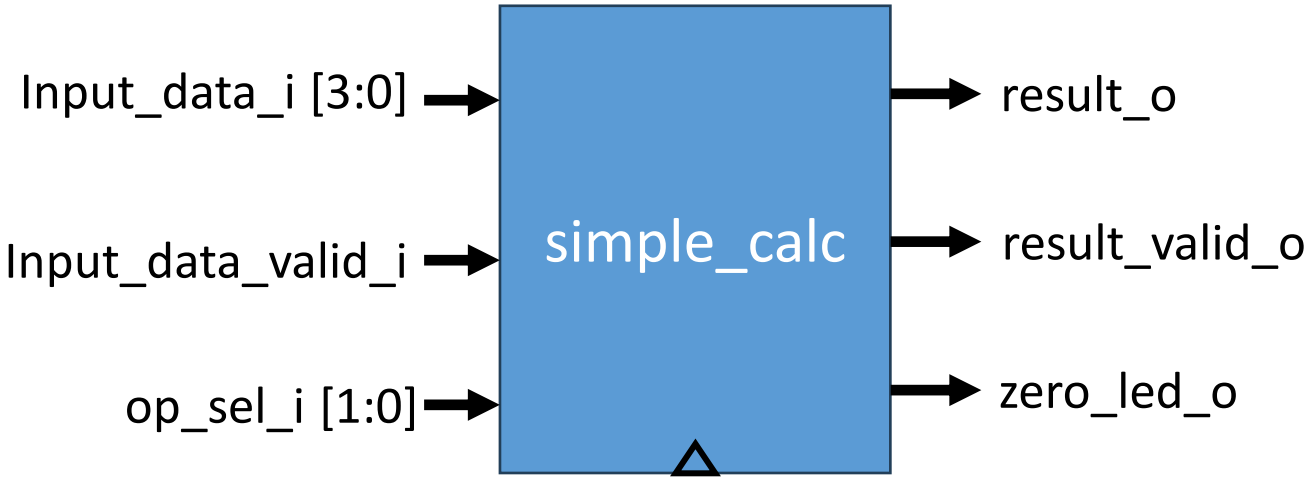
The value in register X gets incremented at the state transition following state C. Register Y gets the "old" value from X; i.e the value before X gets incremented.

Example in SystemVerilog

See example con03.06_asm_generic_example

https://extgit.iaik.tugraz.at/con/examples-2023.git

Example 2
A “Simple Calculator”



Simple_calc

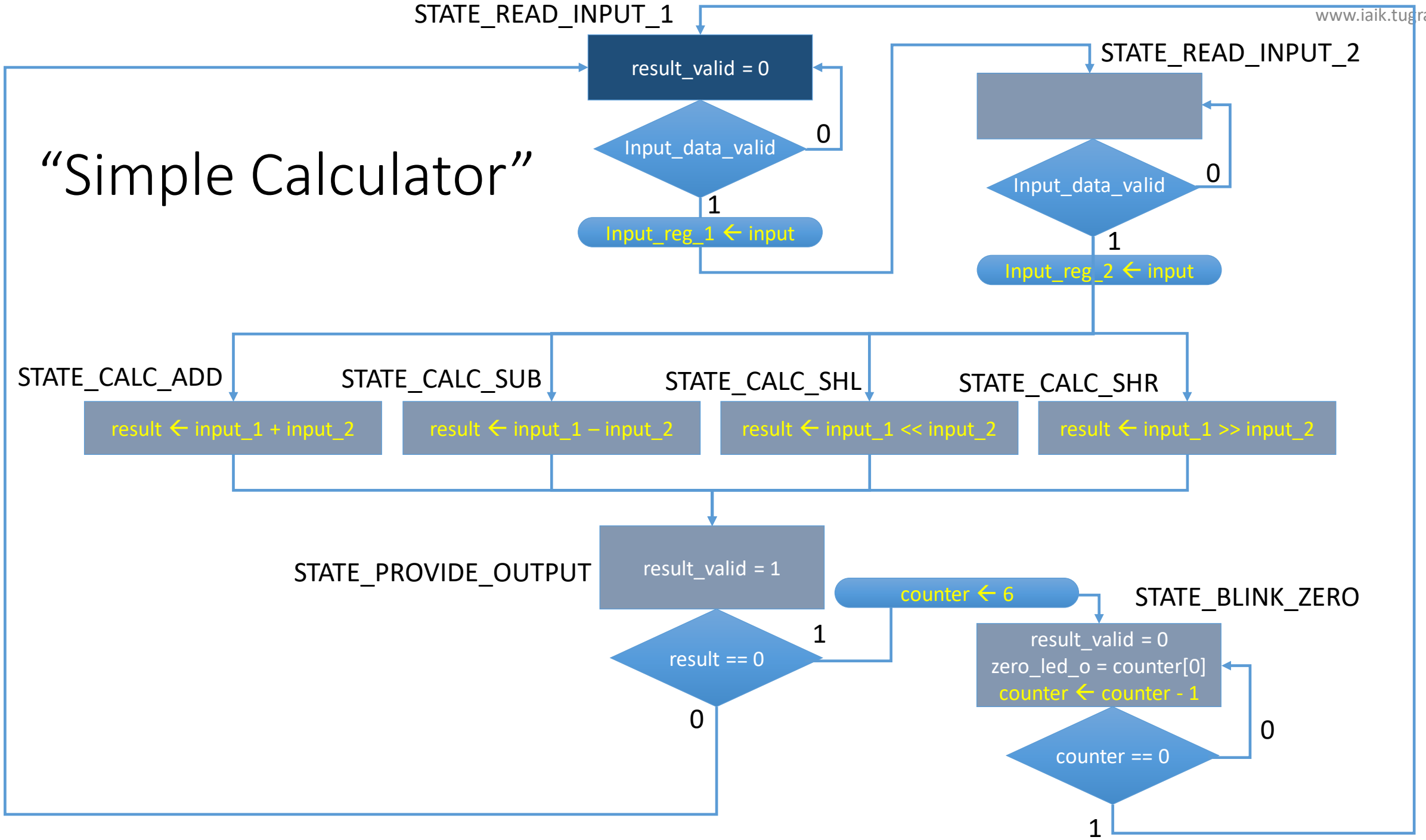
1. Reads two inputs via the input port input_data_i
(Read is done one after the other in the clock cycles when input_data_valid_i == 1)

2. Performs one out of four operations on the inputs (add,sub, shl, shr) selected by op_sel_i

3. Provides the result at result_o for one clock cycle (result_valid_o is set during this clock cycle)

4. It provides a blinking LED output in case the result is zero and then starts from the beginning

“Simple Calculator”



Example in SystemVerilog

See example con03.07_simple_calc

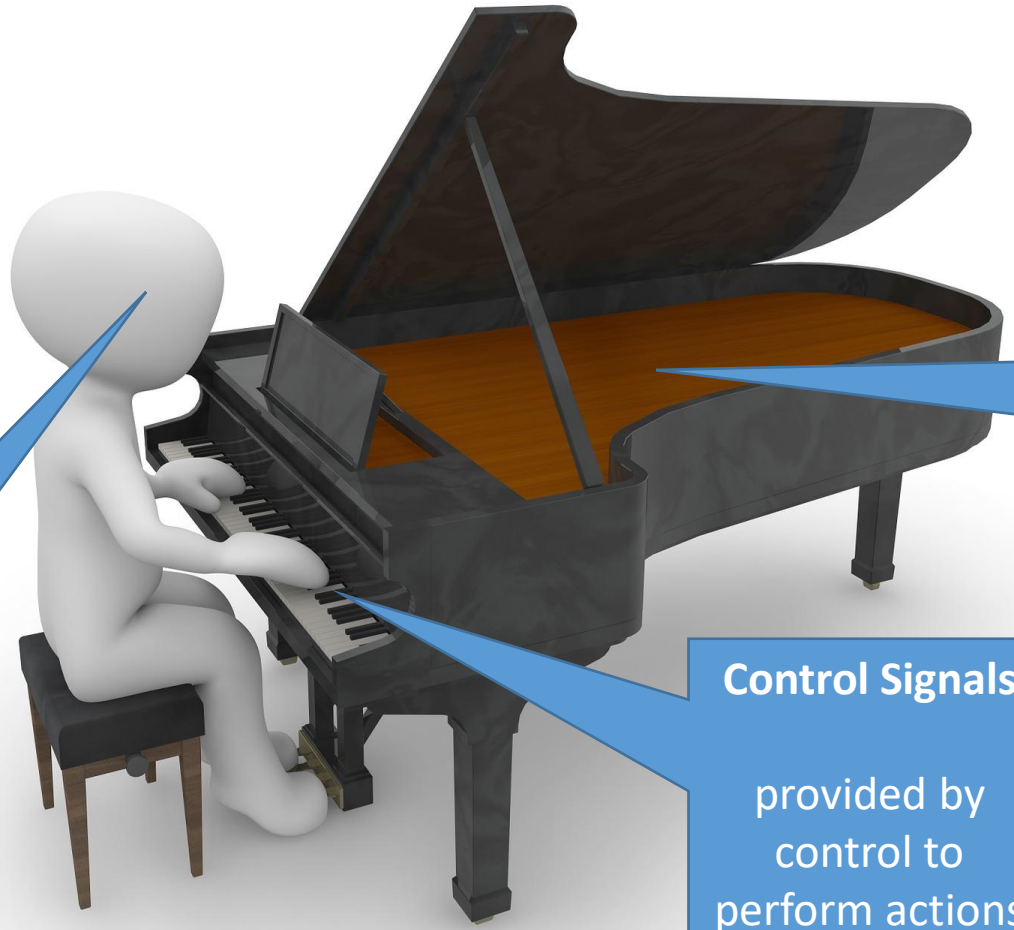
https://extgit.iaik.tugraz.at/con/examples-2023.git

Synthesis of the Data Path and its Control Signals

The Interface Between Control and Data Path

Control

defines when what action should be performed



Data Path

implementing the actions on data registers

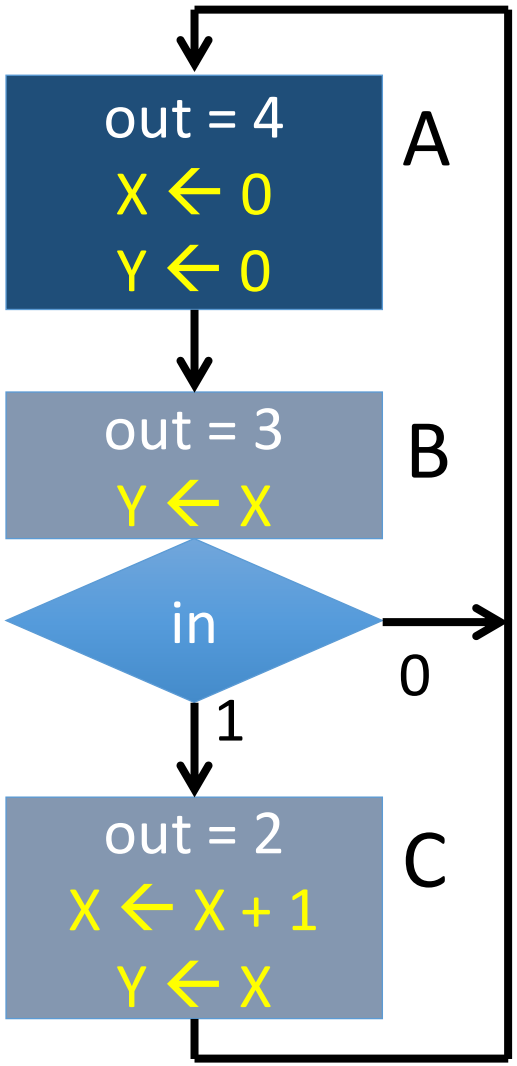
Status Signals

provided by data path to inform control

Control Signals

provided by control to perform actions on data path

Register-Transfer Statements Define the Data Path



Operations for register X:

$X \leftarrow 0$

$X \leftarrow X$

$X \leftarrow X+ 1$

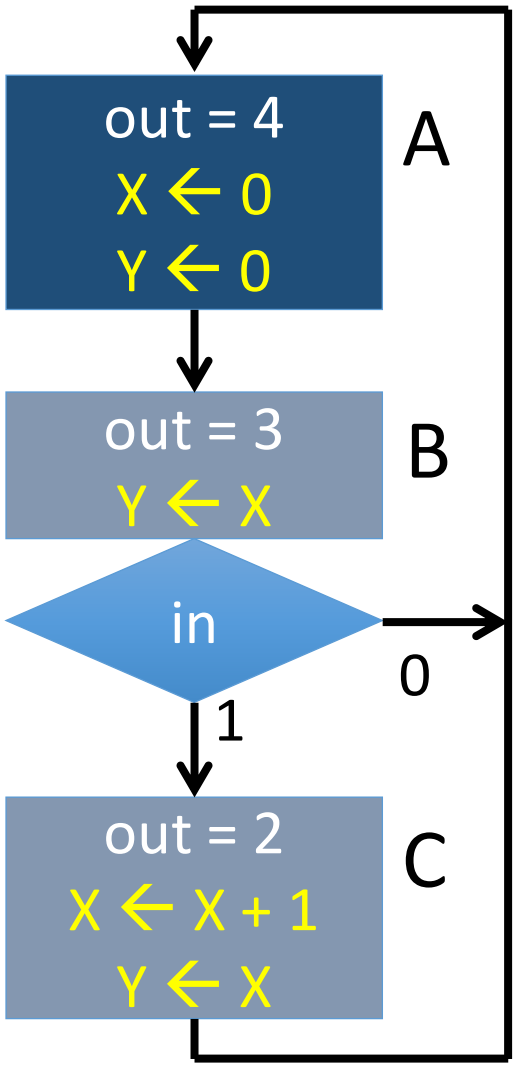
Operations for register Y:

$Y \leftarrow 0$

$Y \leftarrow X$

These are the actions that our system is able to perform on
The data registers X and Y

Register-Transfer Statements Define the Data Path



Operations for register X:

$X \leftarrow 0$

$X \leftarrow X$

$X \leftarrow X+ 1$

Operations for register Y:

$Y \leftarrow 0$

$Y \leftarrow X$

These are the operations that our data path implements

Operations for register X

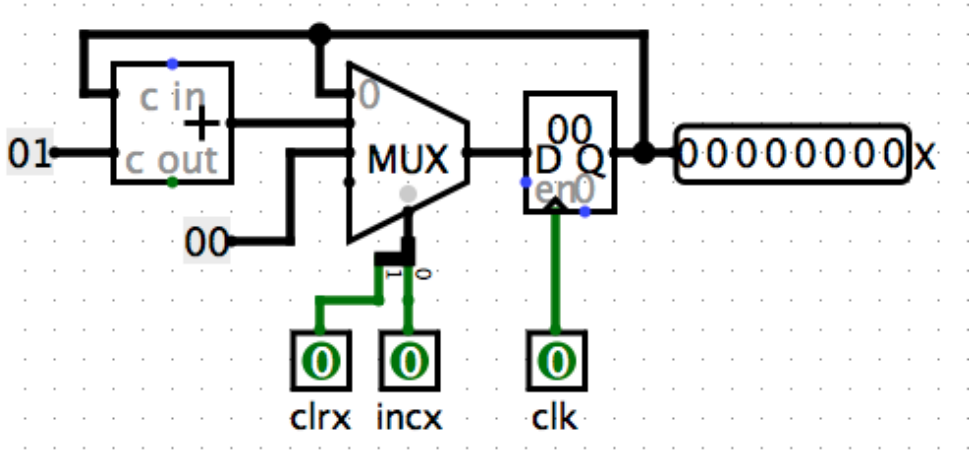
Case 0: X ← X
Case 1: X ← X+ 1
Case 2: X ← 0

We need to distinguish between 3 cases.

→ A one bit control signal is not enough. We need two control signals.

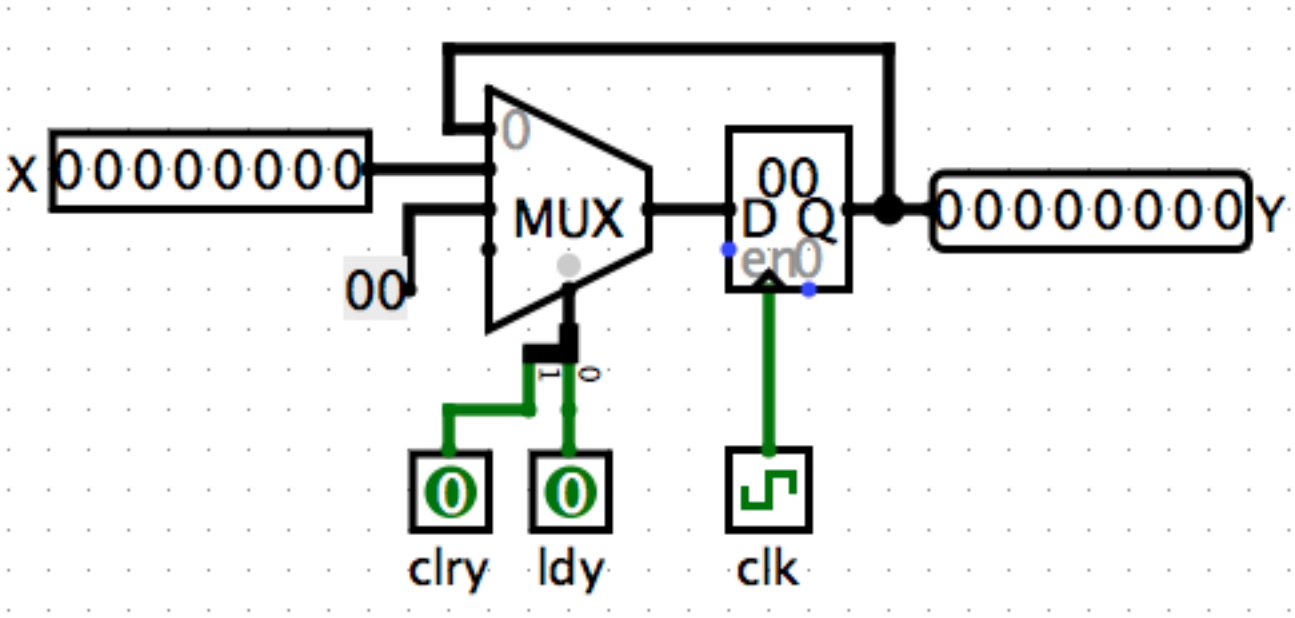
Control Signals and Datapath for the Actions on Register X

clrx	incx	action
0	0	X ← X
0	1	X ← X+ 1
1	0	X ← 0

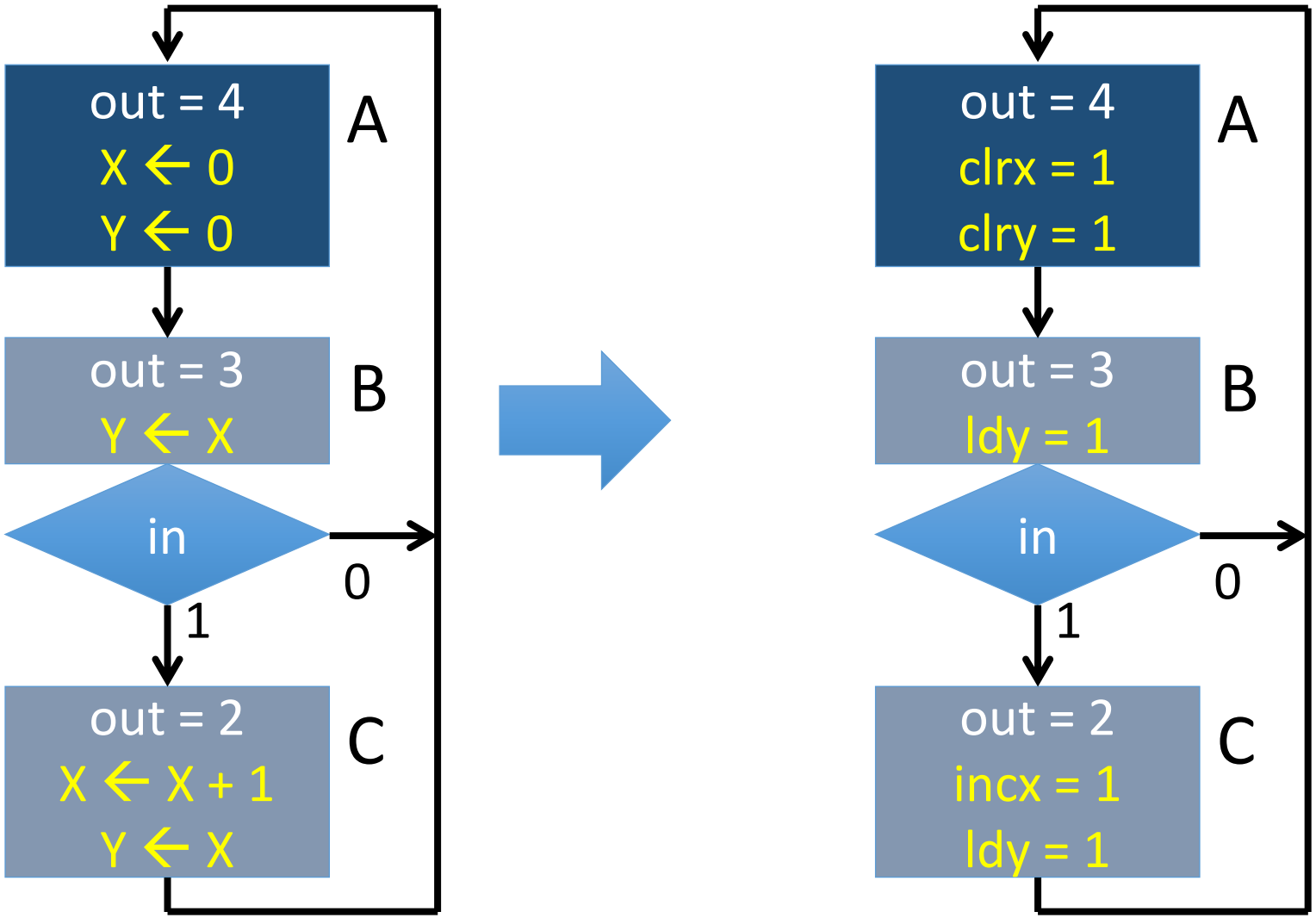


Control Signals and Datapath for the Actions on Register Y

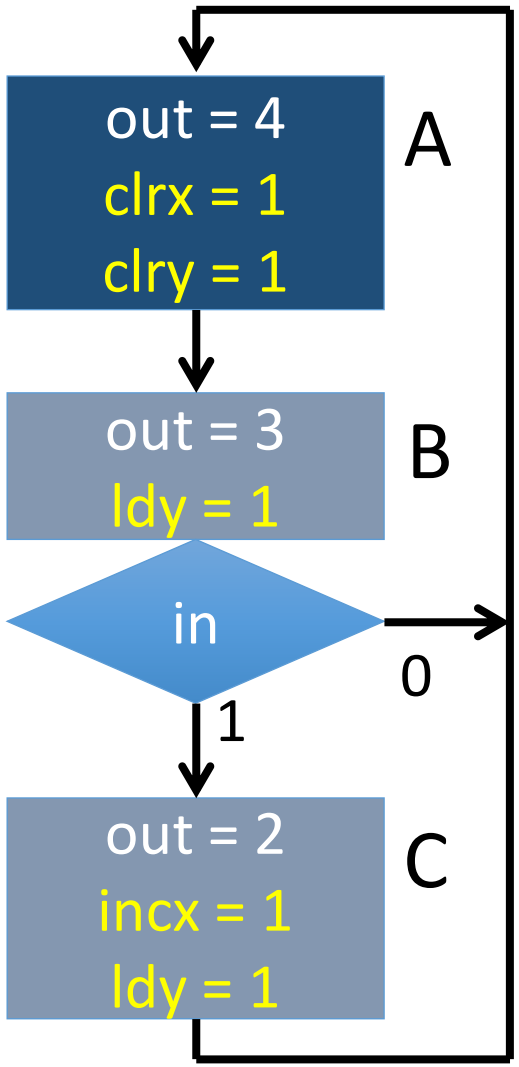
clry	ldy	action
0	0	Y ← Y
0	1	Y ← X
1	0	Y ← 0



Register-transfer statements become assignment of control signals in the controller

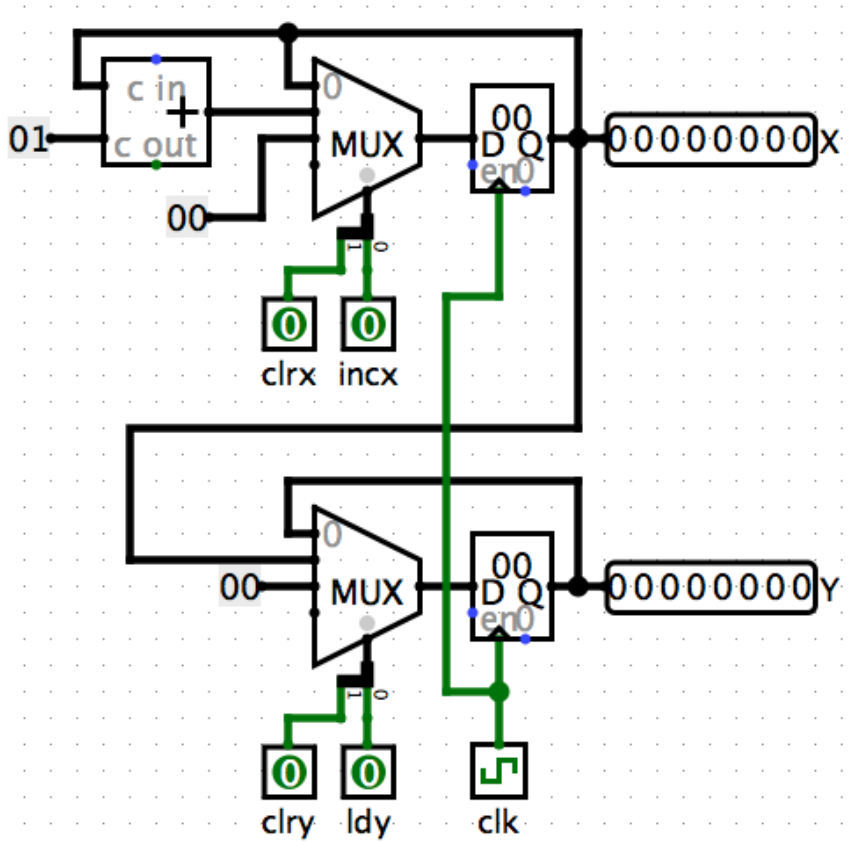


Separated Control & Datapath




```
// Combinational logic of the data path
always_comb begin
  x_n = x_p;
  y_n = y_p;

  // operations for register X
  if (incx)
    x_n = x_p +1;

  if (clrx)
    x_n = 3'b000;

  // operations for register Y
  if (ldy)
    y_n = x_p;

  if (clry)
    y_n = 3'b000;
end
```

```
// registers of the data path
always_ff @(posedge clk_i or posedge reset_i) begin
  if (reset_i) begin
    x_p <= 3'b000;
    y_p <= 3'b000;
  end else begin
    x_p <= x_n;
    y_p <= y_n;
  end
end
```



Example in SystemVerilog

See example
con03.08_asm_generic_example_with_separate_datapath

https://extgit.iaik.tugraz.at/con/examples-2023.git