Computer Organization and Networks
(INB.06000UF, INB.07001UF)

Chapter 3 – State Machines
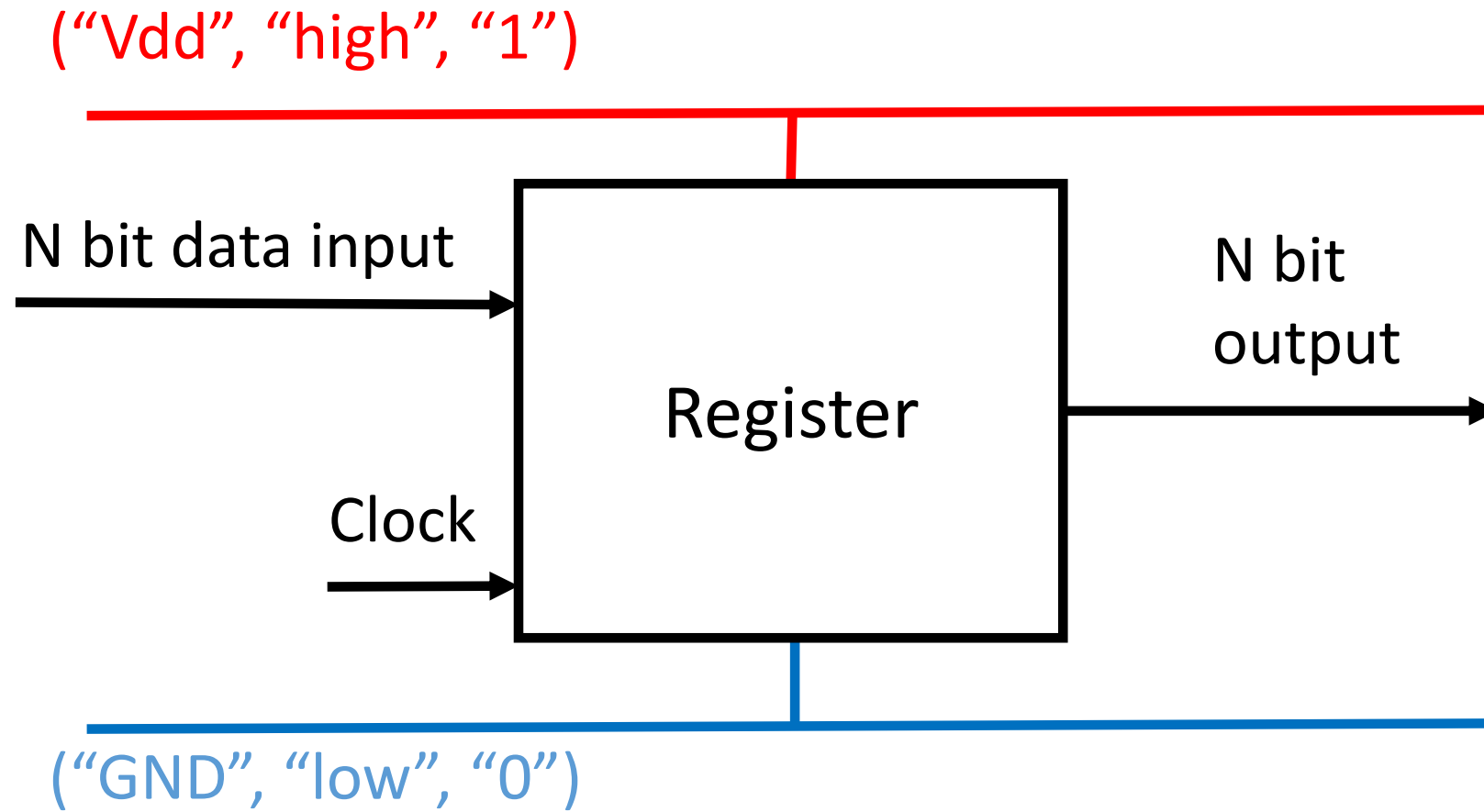
Winter 2021/2022

Stefan Mangard, www.iaik.tugraz.at

What About Storage?

- We have so far only considered functions

- We also want to store data and define sequences of computation

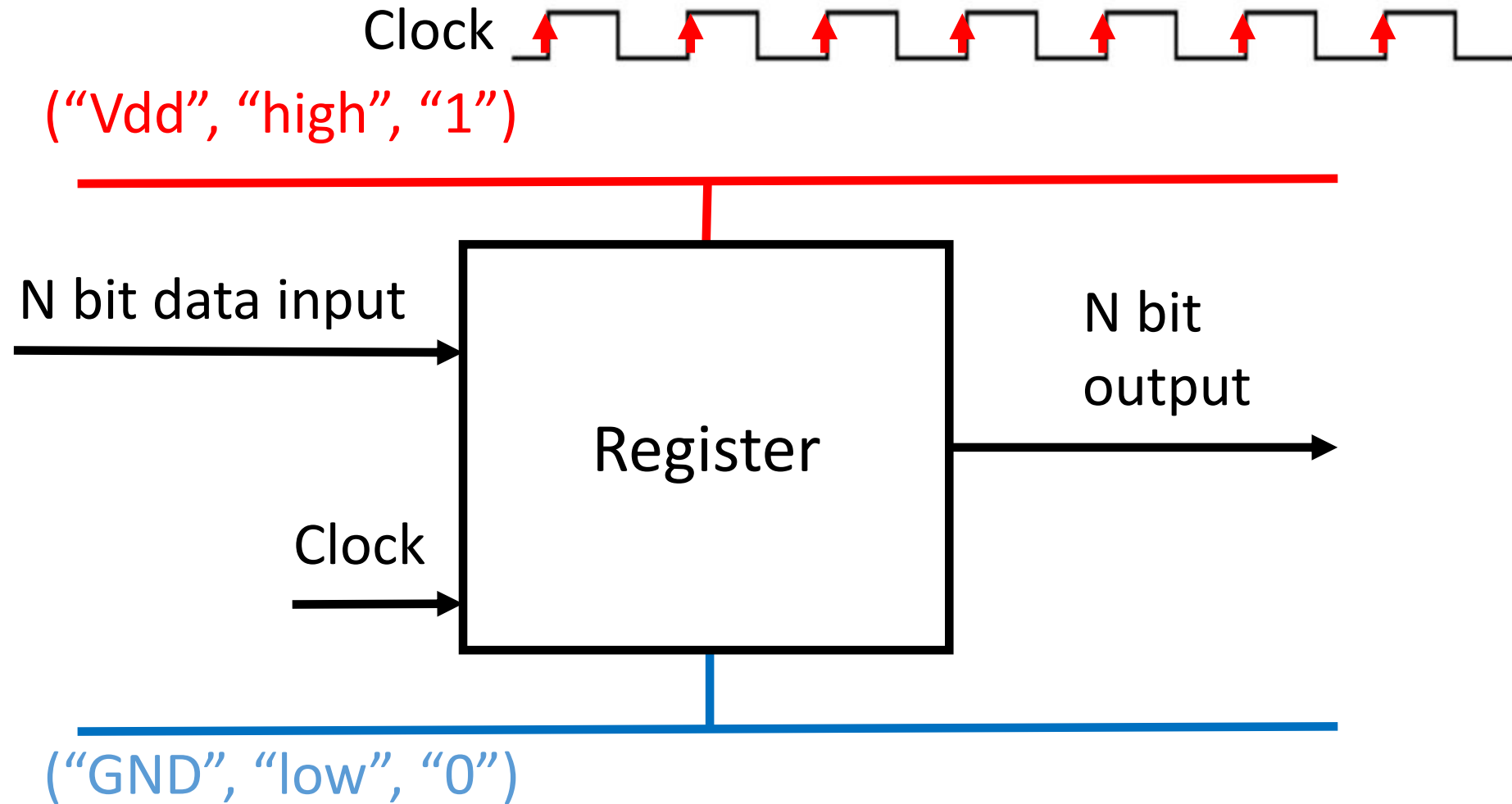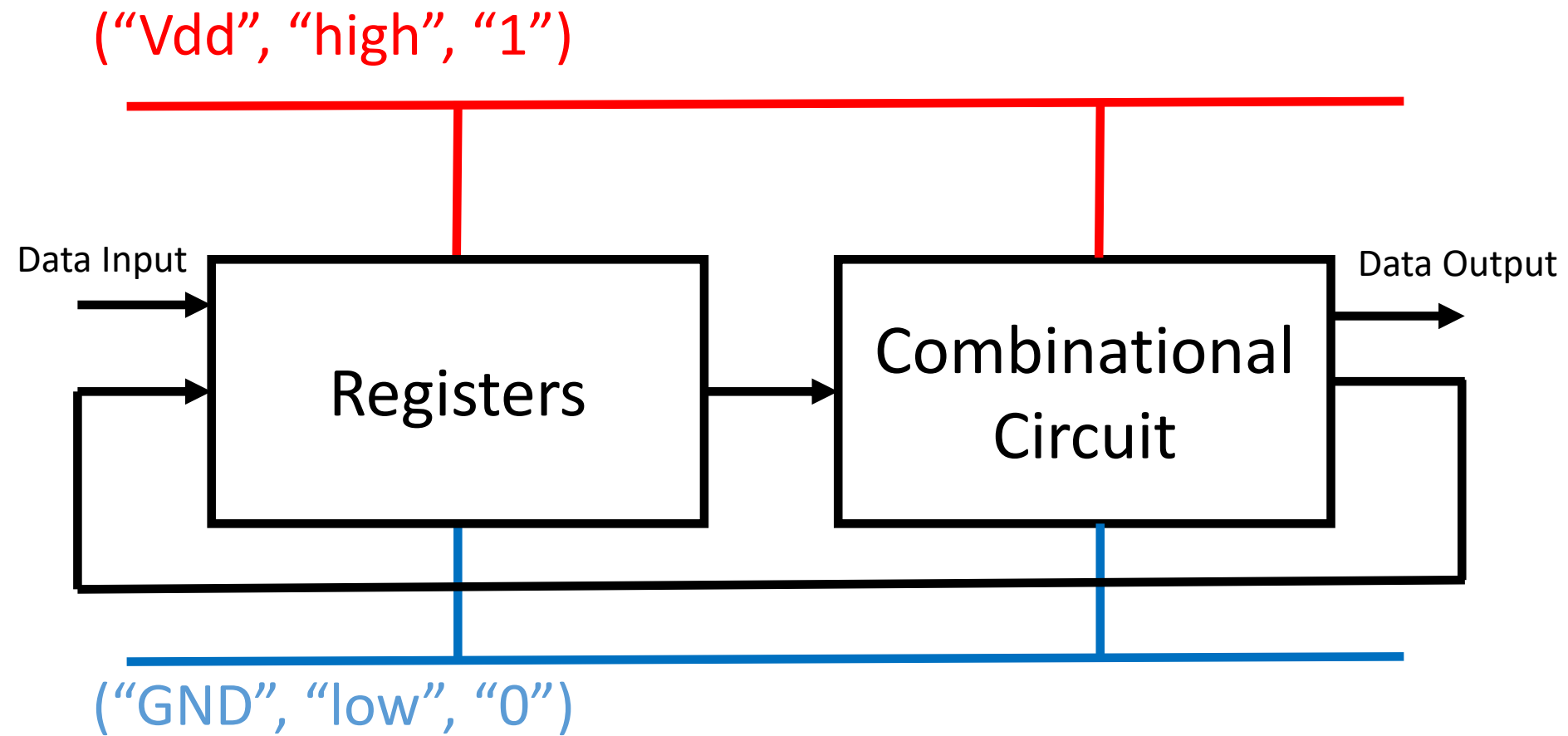- So far, we have not talked about storage or about time

Storage

Storage

("Vdd", "high", "1")

N bit data input

Clock

Register

N bit output

("GND", "low", "0")

Storage

Clock

("Vdd", "high", "1")

N bit data input
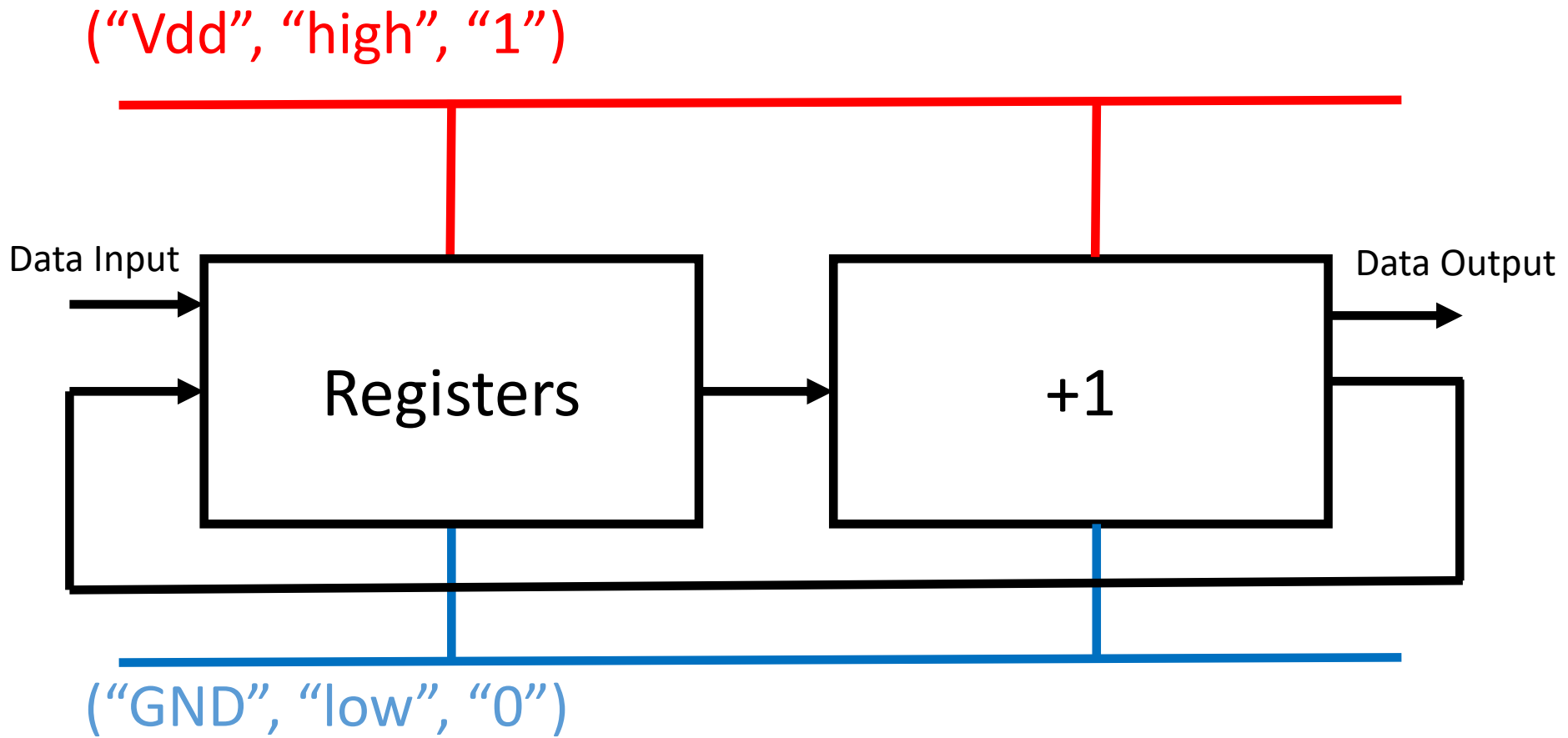
Register

Clock

N bit output

("GND", "low", "0")

Naming Conventions

- Register: An n-bit storage sampling data on the rising clock edge

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

Flip-Flop based on CMOS Gates
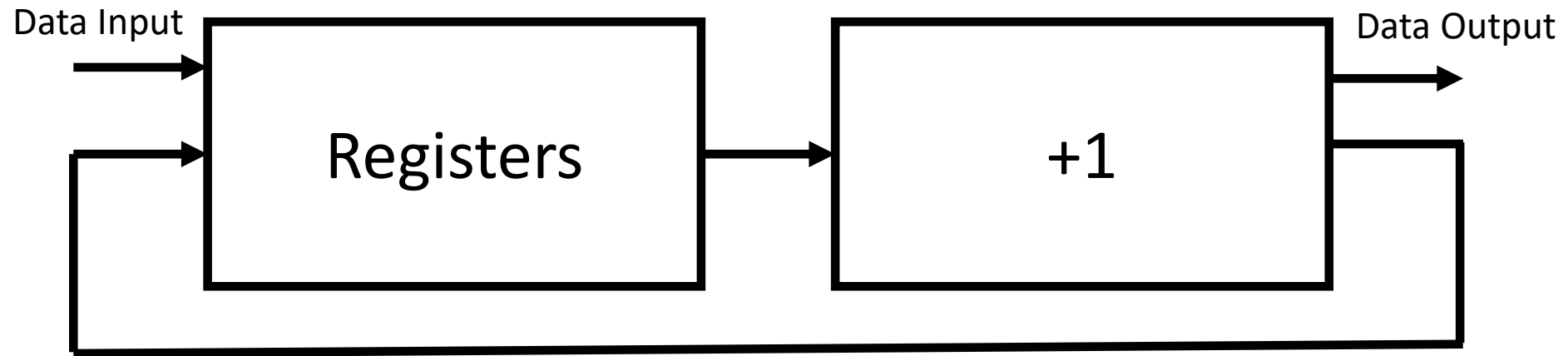
Combining

("Vdd", "high", "1")

Data Input

Registers

Combinational Circuit

Data Output

("GND", "low", "0")

Example Counter

("Vdd", "high", "1")

Data Input

| Registers | | +1 |

Data Output

("GND", "low", "0")

Example Counter

Let's Build This in Digital

- See examples con03 available at

  https://extgit.iaik.tugraz.at/con/examples-2021.git
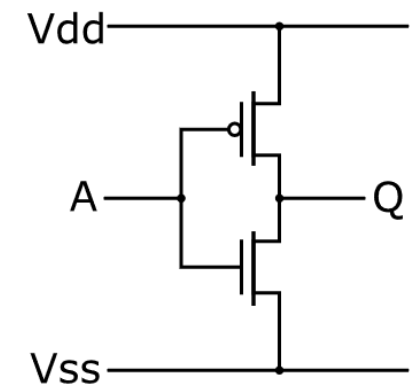
The Clock Frequency

- 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

Data Input → | Registers | → | Combinational Circuit | → Data Output
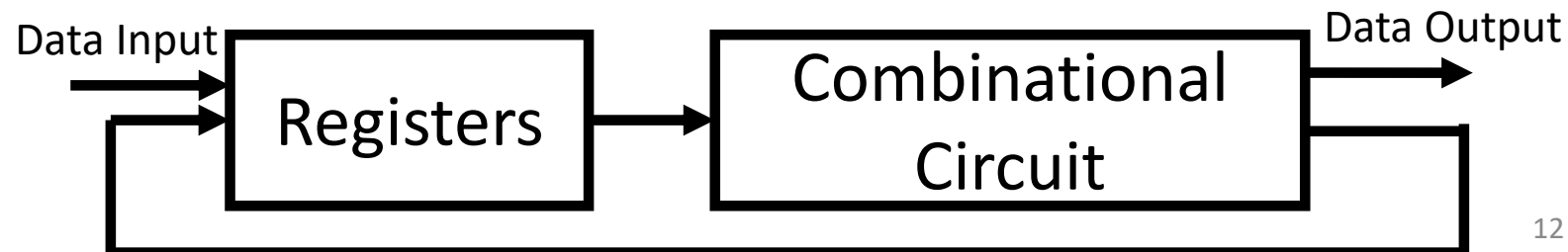
Temperature, Power Consumption

- 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

State Machines

Finite State Machines (FSMs)

- FSMs are the "work horse" in digital systems.

- FSMs can be implemented with hardware.

- We look at "synchronous" FSMs only:
  - The "clock signal" controls the action over time

- FSMs can be described with three main "views":
  - The functional view with the "state diagram"
  - The timing view with the "timing diagram"
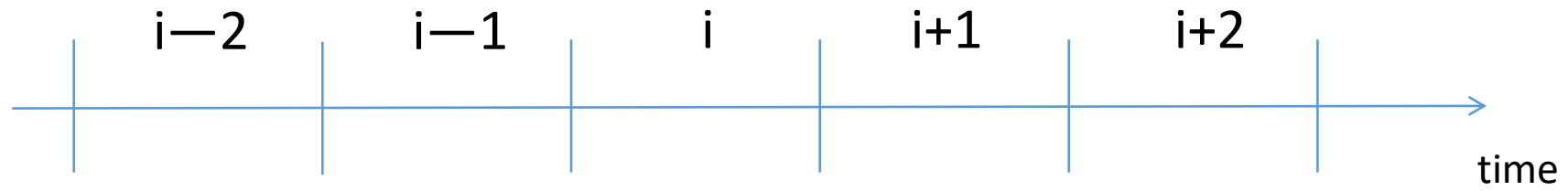 - The structural view with the "logic circuit diagram"

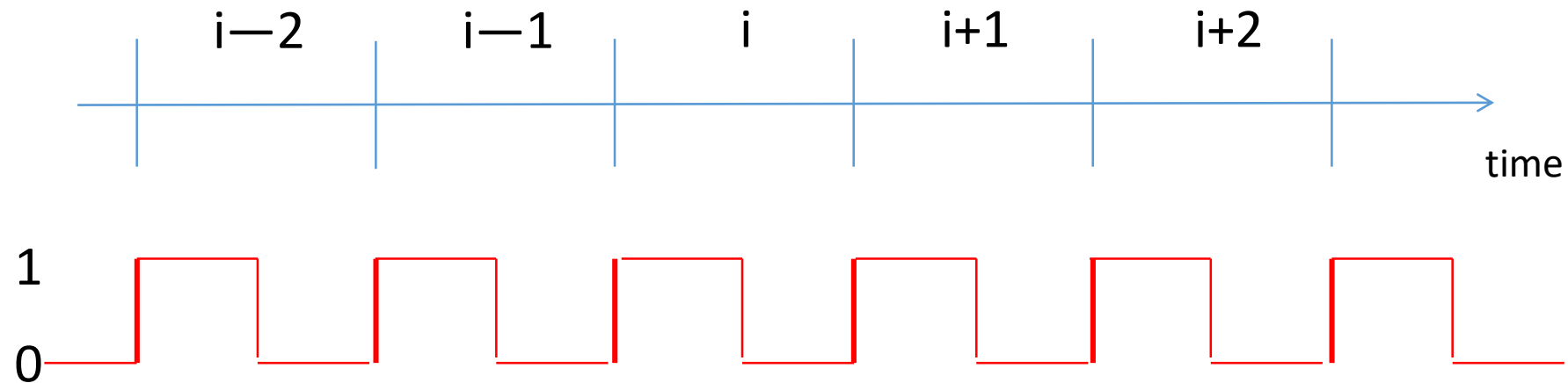# Time flows continuously

time

We cut time into slices
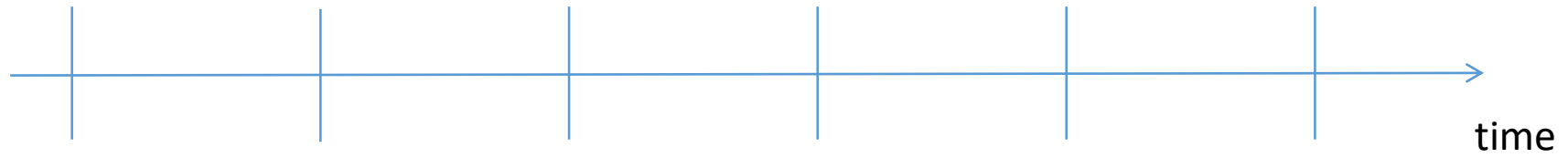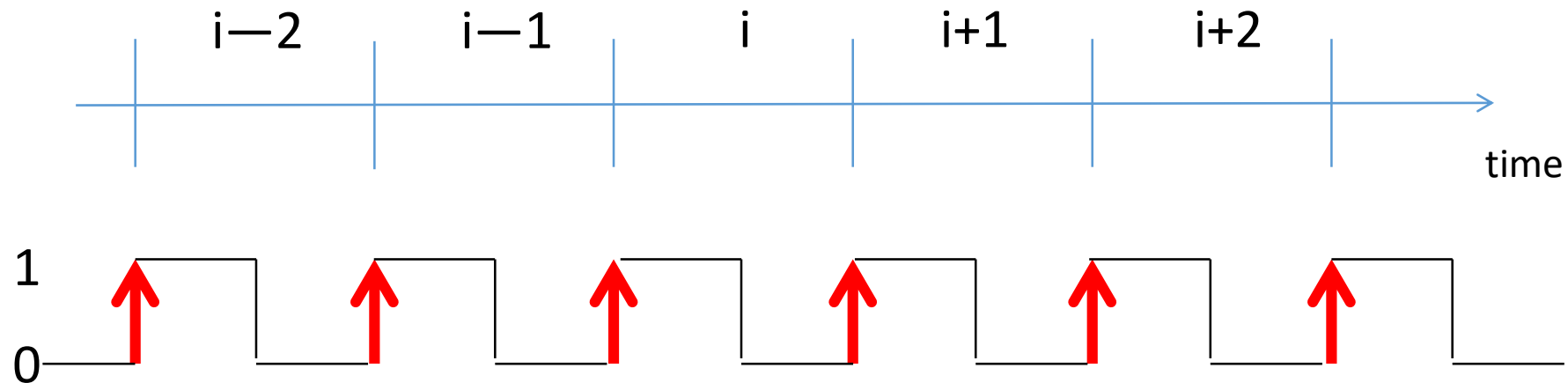
time

Time slices are strictly ordered

Clock signal

We use the clock signal ("clk") in order to advance time from slice to slice.
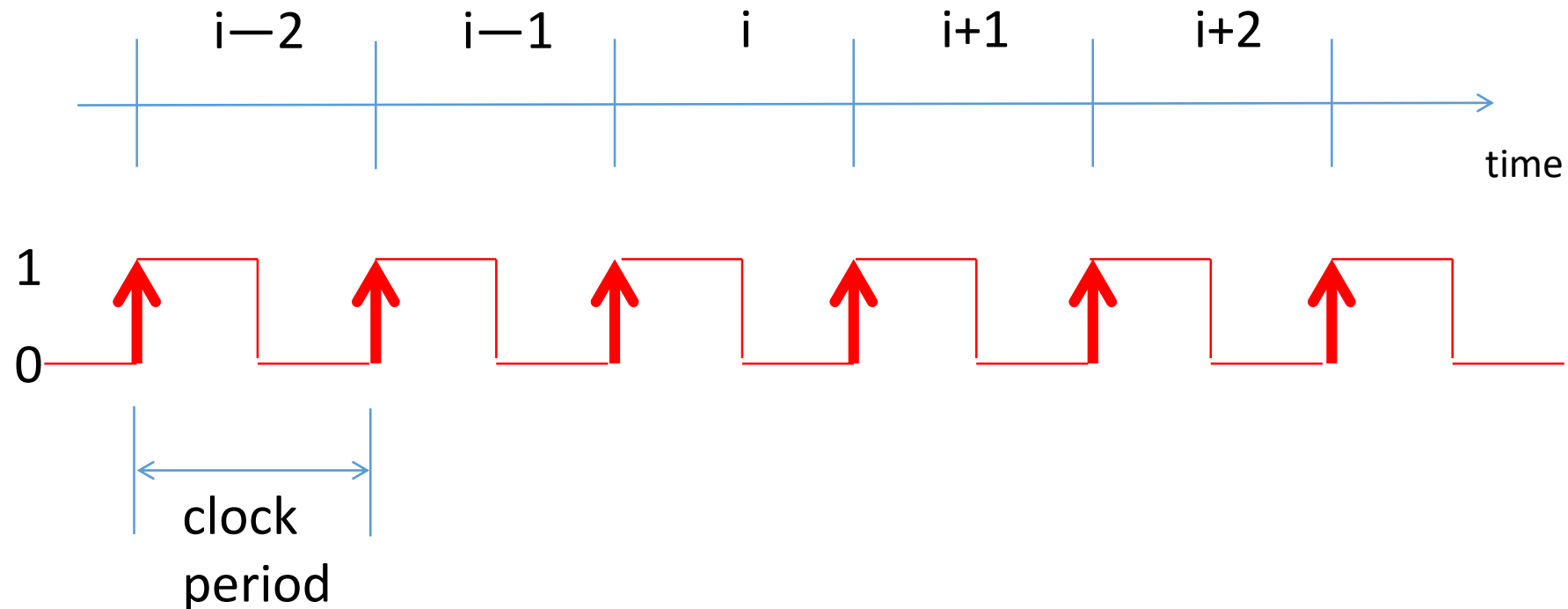
Rising clock edge

With **rising clock edges** we define the transition between neighboring time slices.
The negative clock edges have no importance.
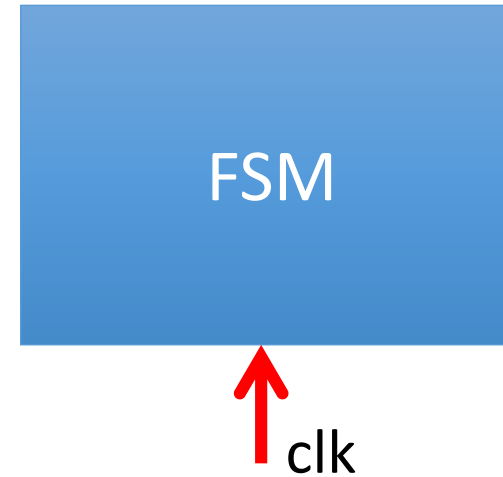
Clock period

i—2   i—1   i   i+1   i+2

time

1

0

clock
period

We call the time between two rising clock
edges also "clock period".
Most often, clock periods have the same
length. But this is not necessarily the case.

Synchronous finite state machine (= automaton)

FSM

clk

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.
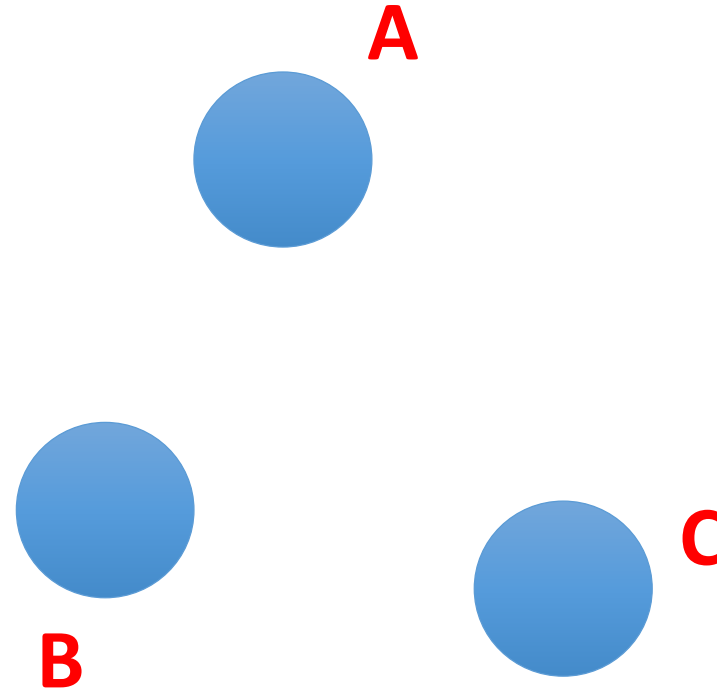
We are interested in "**finite state machines**" (FSMs).
FSMs have only a finite number of states.

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

State diagram:

A

B

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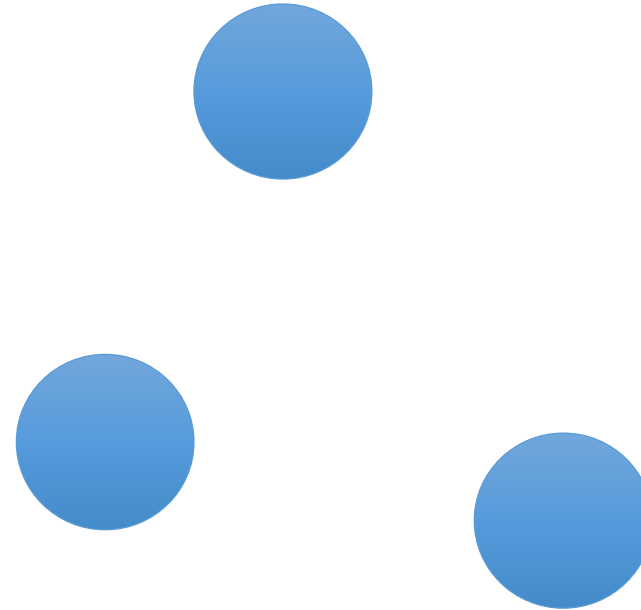
We denote the states with circles and give them symbolic names, e.g. A, B, and C.

State diagram:

A

B

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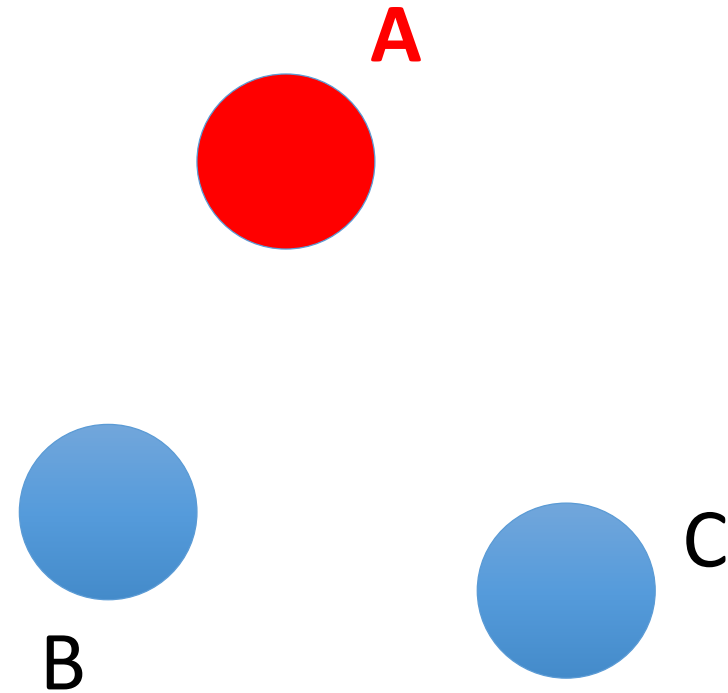
We define one of the states as the initial state.

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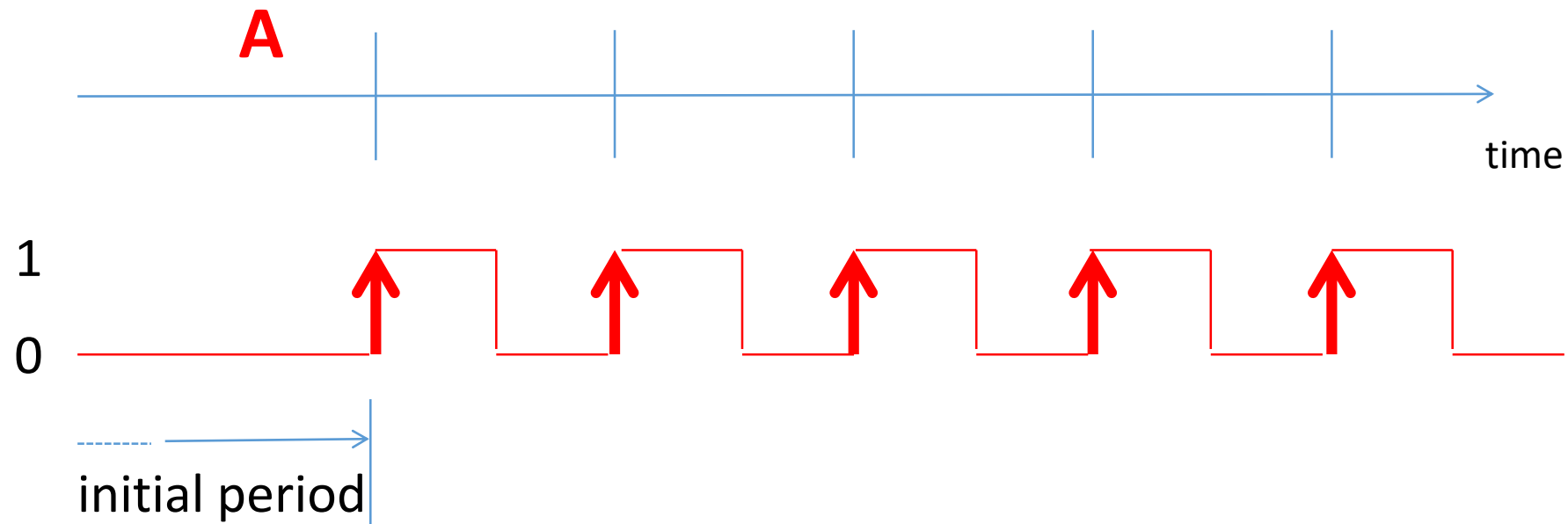
Initially, i.e. shortly after switching on the FSM and before the first rising edge of clock, there is the initial period. In this period, the FSM is in the "initial state".

State diagram:
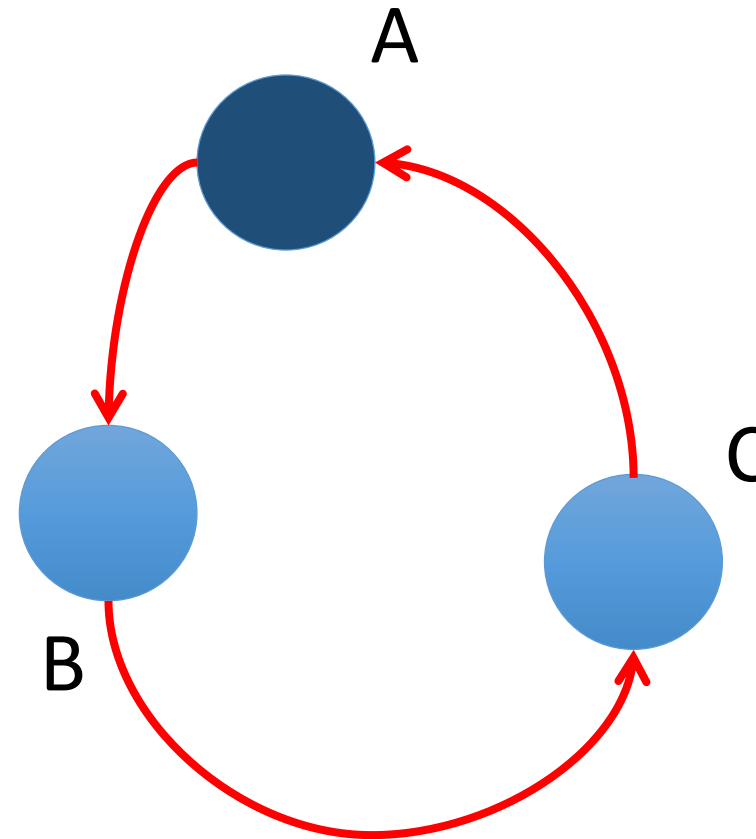
With arrows we define the sequence of states.

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

present state	next state
A	B
B	C
C	A

Timing diagram

time

clk

1

0

state

A   B   C   A   B   C
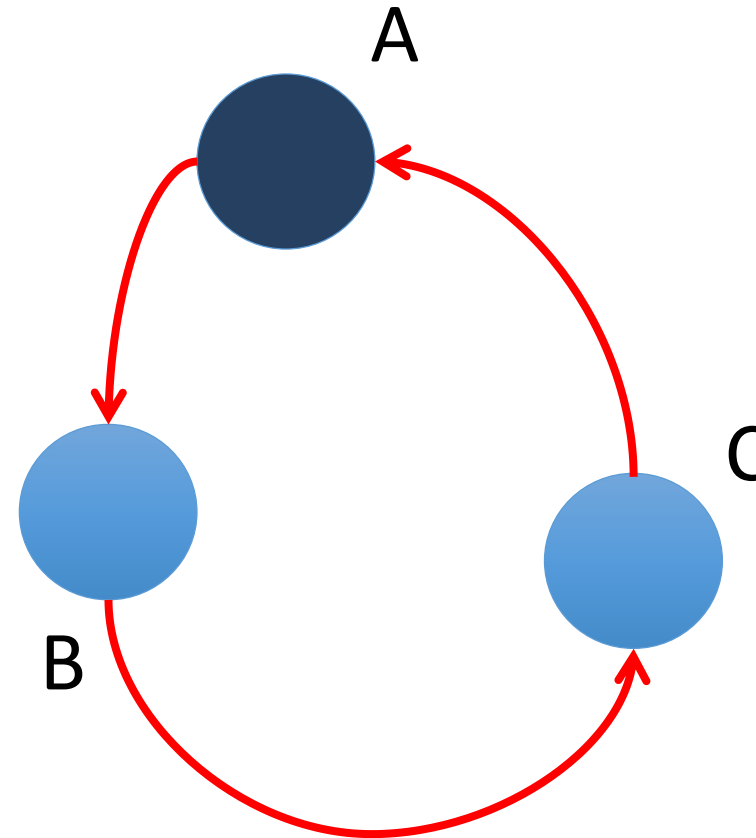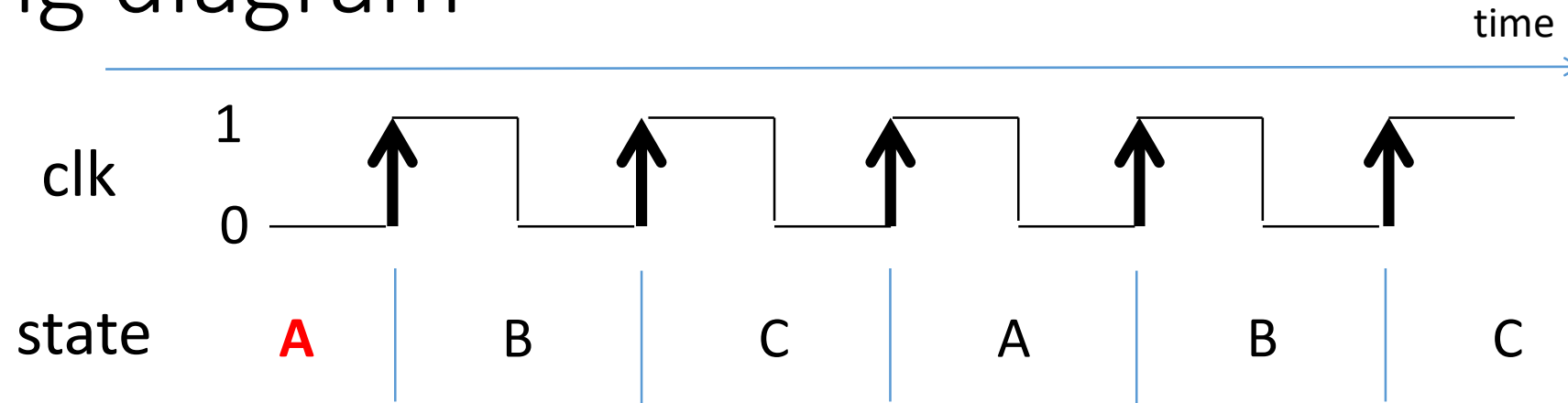
Initially, the FSM is in the initial state A.

With every positive clock edge, the next state becomes the current state.

An FSM has **always** a next state.

In order to technically realize ("implement") a state diagram,
we start by giving
each state
a unique number.
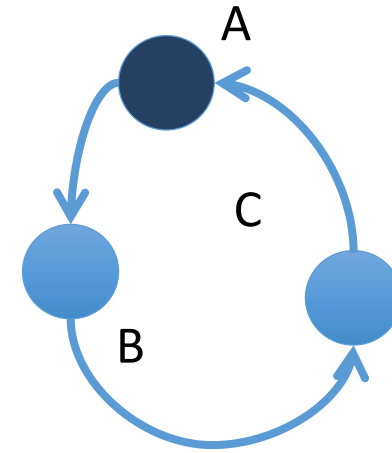
Popular state-encoding schemes

A

C

B
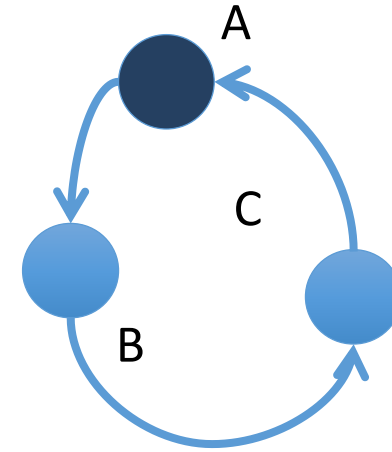
- **Binary encoding**
  - needs minimum amount of flip-flops.

- **One-hot encoding**
  - Tends to have a simpler next-state logic.

00

10

01
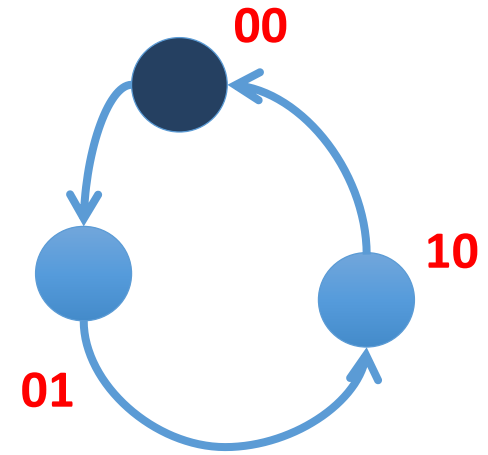
Binary encoding

state	number
A	**00**
B	**01**
C	**10**

33

00

10

01
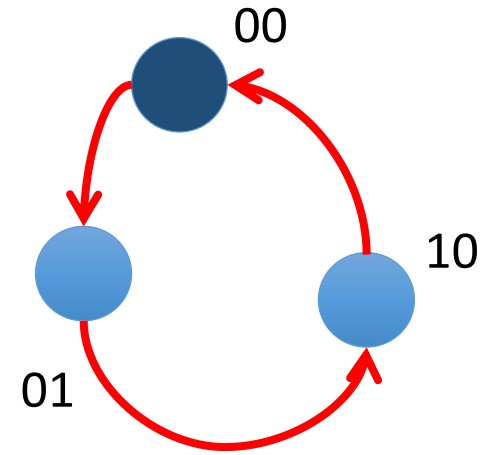
Binary encoding:
The state transition table
has also only binary numbers.

present state	next state
00	**01**
01	**10**
10	**00**

We also enter the unused combination "11".
This state does not exist.
"x" stands for "Don't care".
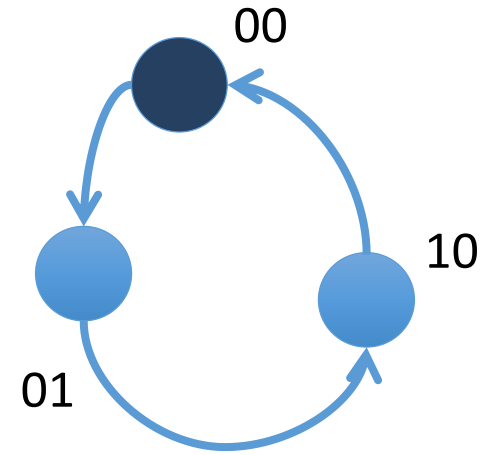
present state	next state
00	01
01	10
10	00
11	xx

We define names for the two state bits, e.g. s1, s0.
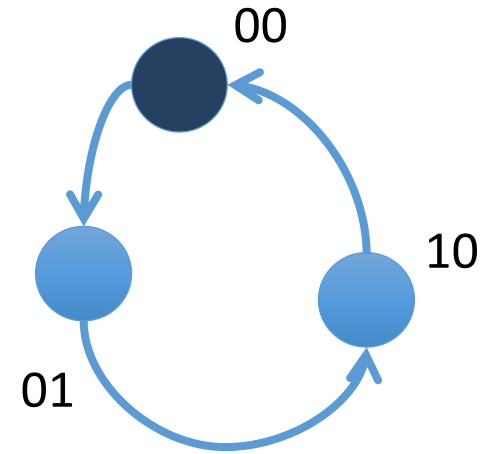
present		next	
s1	s0	s1	s0
0	0	0	1
0	1	1	0
1	0	0	0
1	1	x	x

Each state bit is stored in a flipflop

flipflop → stored value
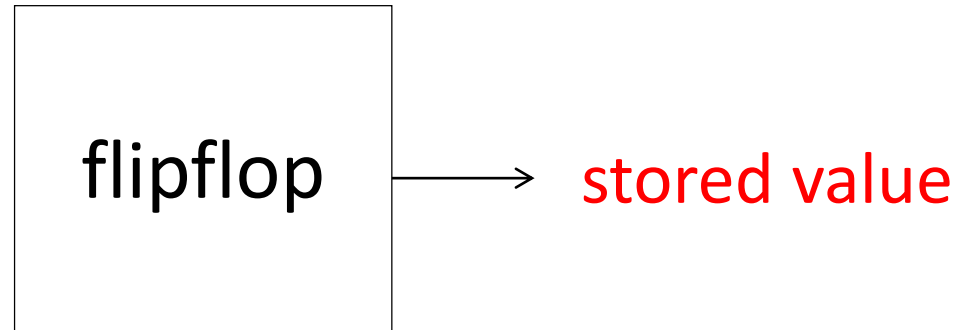
In our example, we need 2 flip-flops for storing the state bits.

next s0 ⟶ D0   Q0 ⟶ s0

next s1 ⟶ D1   Q1 ⟶ s1

clk
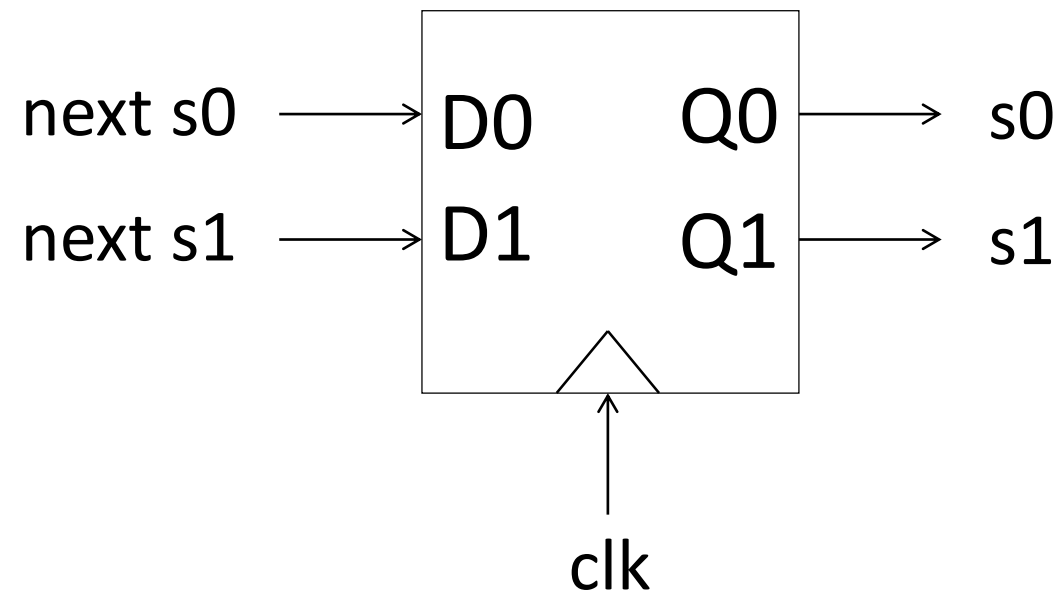
Flipflops which can store several bits
are also called "registers".

We use the state transition table as a "lookup table"…

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

next s0 → D0   Q0 → s0

next s1 → D1   Q1 → s1

clk
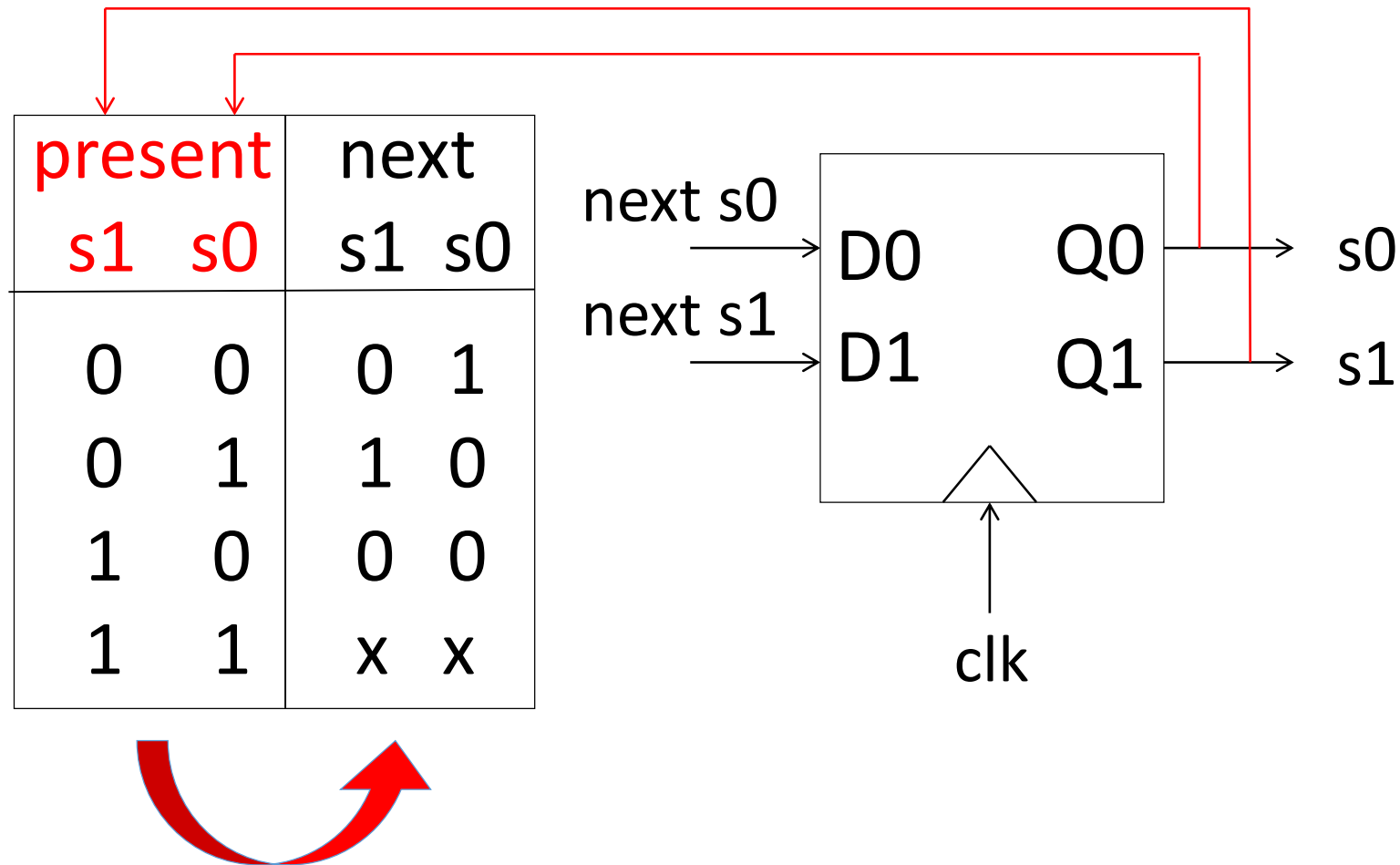
…and thus find out the next state

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

D0   Q0 → s0

D1   Q1 → s1

clk
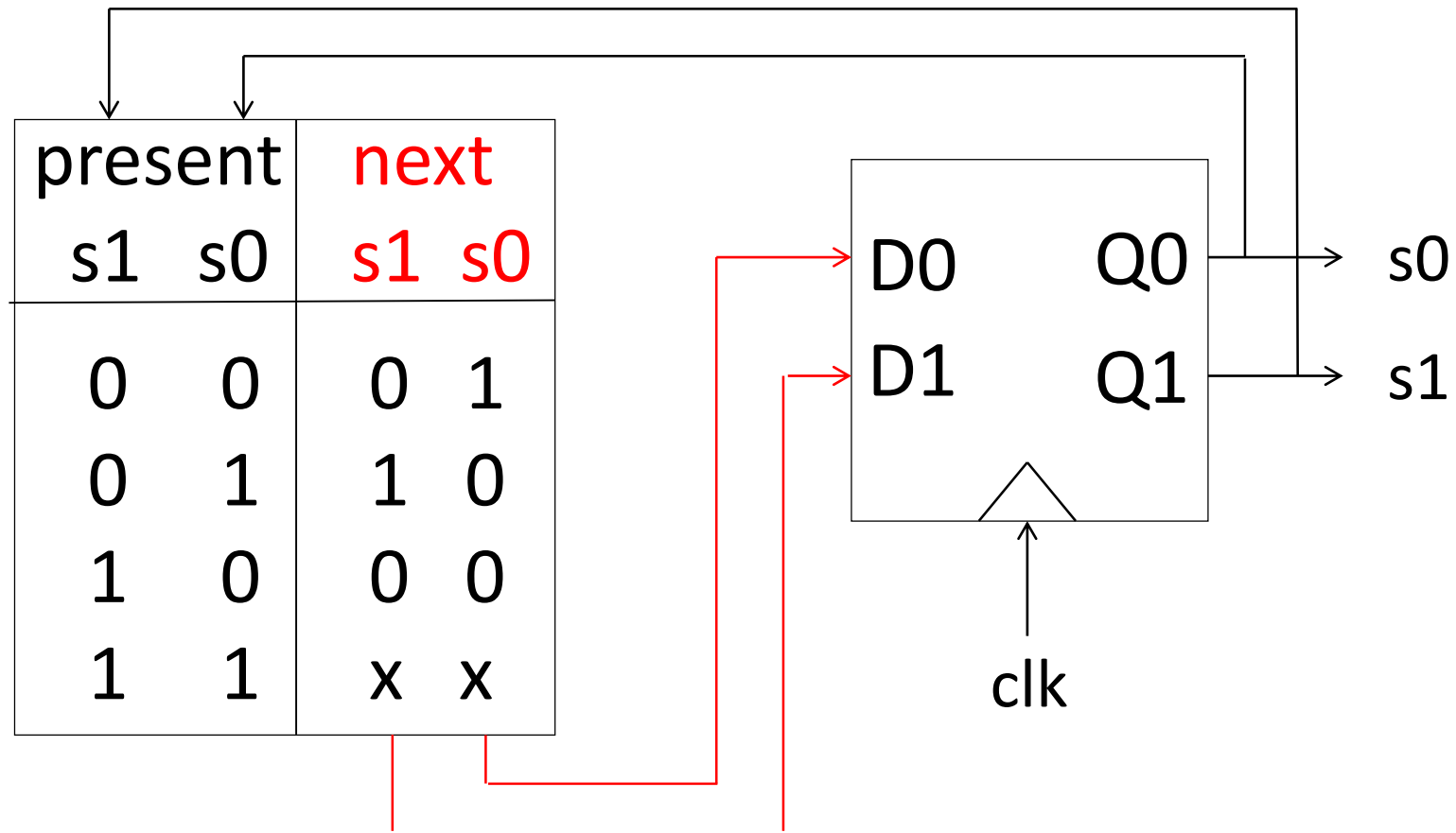
At each positive edge of clk, the next state gets stored as the current state.

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

D0   Q0 → s0

D1   Q1 → s1

clk
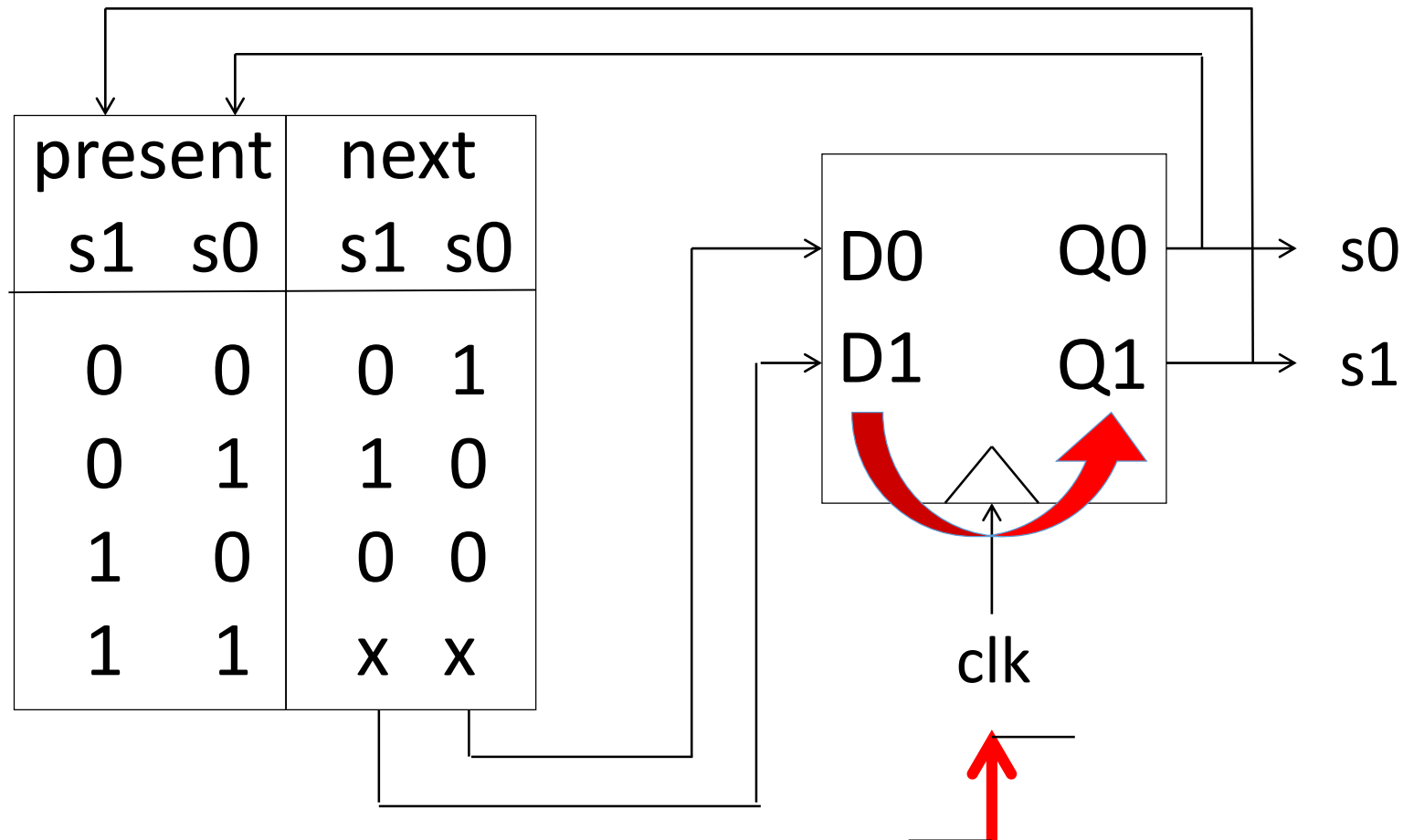
In order to get the initial state "00", we use flipflops with an "asynchronous reset input". Shortly after switching on the circuit, we apply "areset".

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

D0   Q0   → s0

D1   Q1   → s1

clk

areset
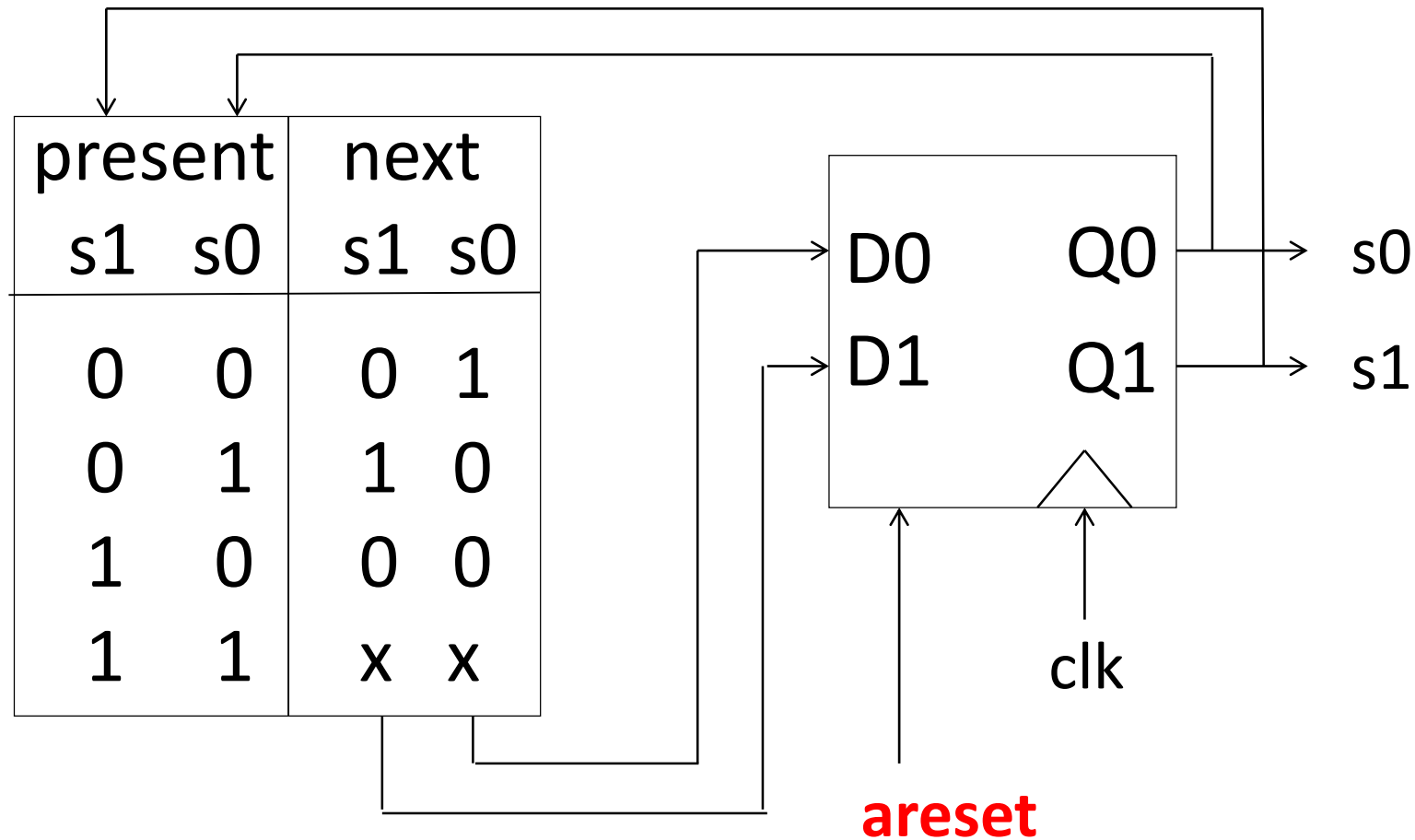
The state transition function in this example can be derived from the truth table:
next s0 = (~s1) & (~s0)

present		next	
s1	s0	s1	s0
0	**0**	0	**1**
0	1	1	0
1	0	0	0
1	1	x	0

D0   Q0   s0

D1   Q1   s1

clk

areset
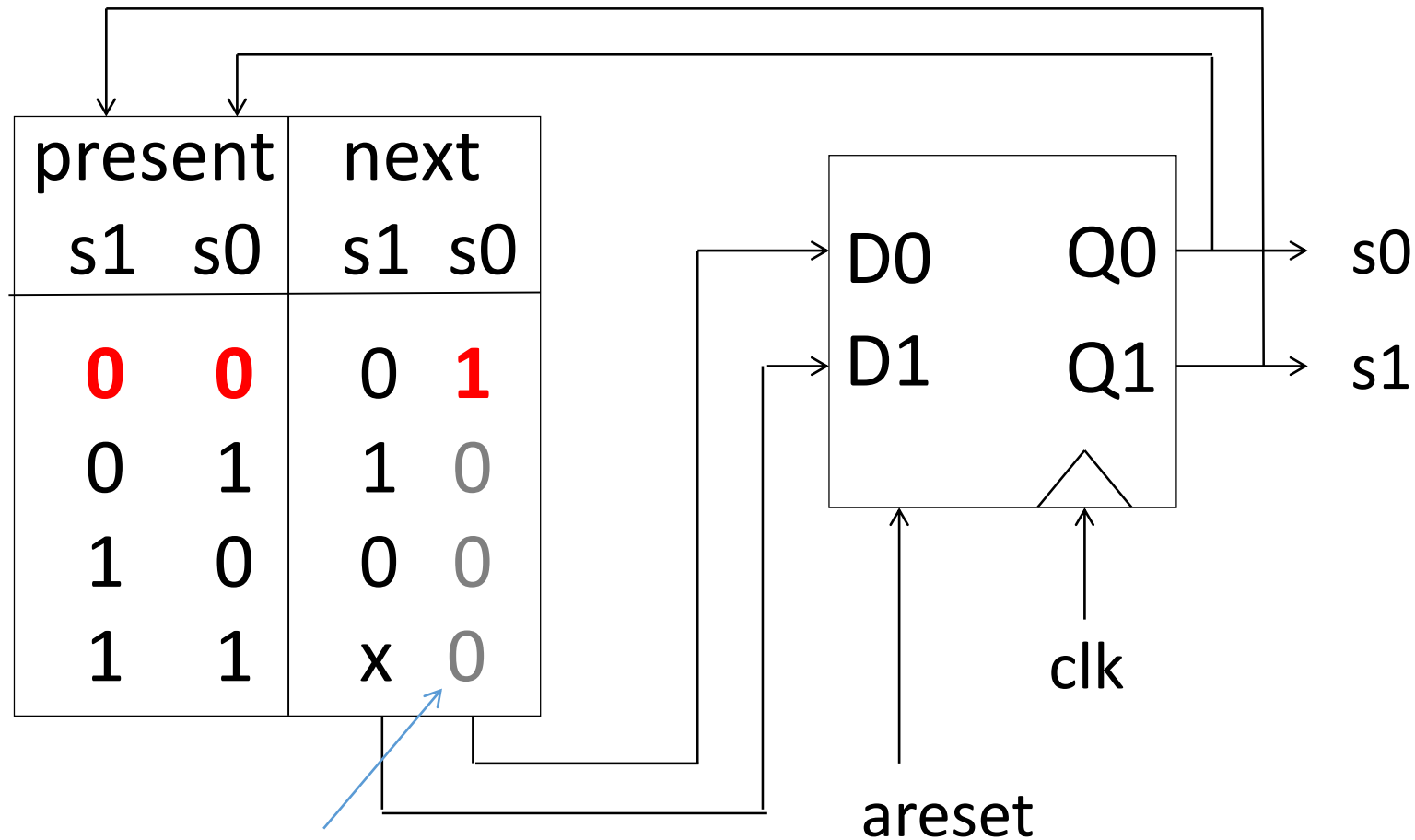
We set the don't care to 0.

The state transition function:
next s0 = (~s1) & (~s0)
next s1 = (~s1) & s0

present		next	
s1	s0	s1	s0
0	**0**	0	**1**
0	**1**	**1**	0
1	0	0	0
1	1	0	0

D0   Q0   s0
D1   Q1   s1

clk
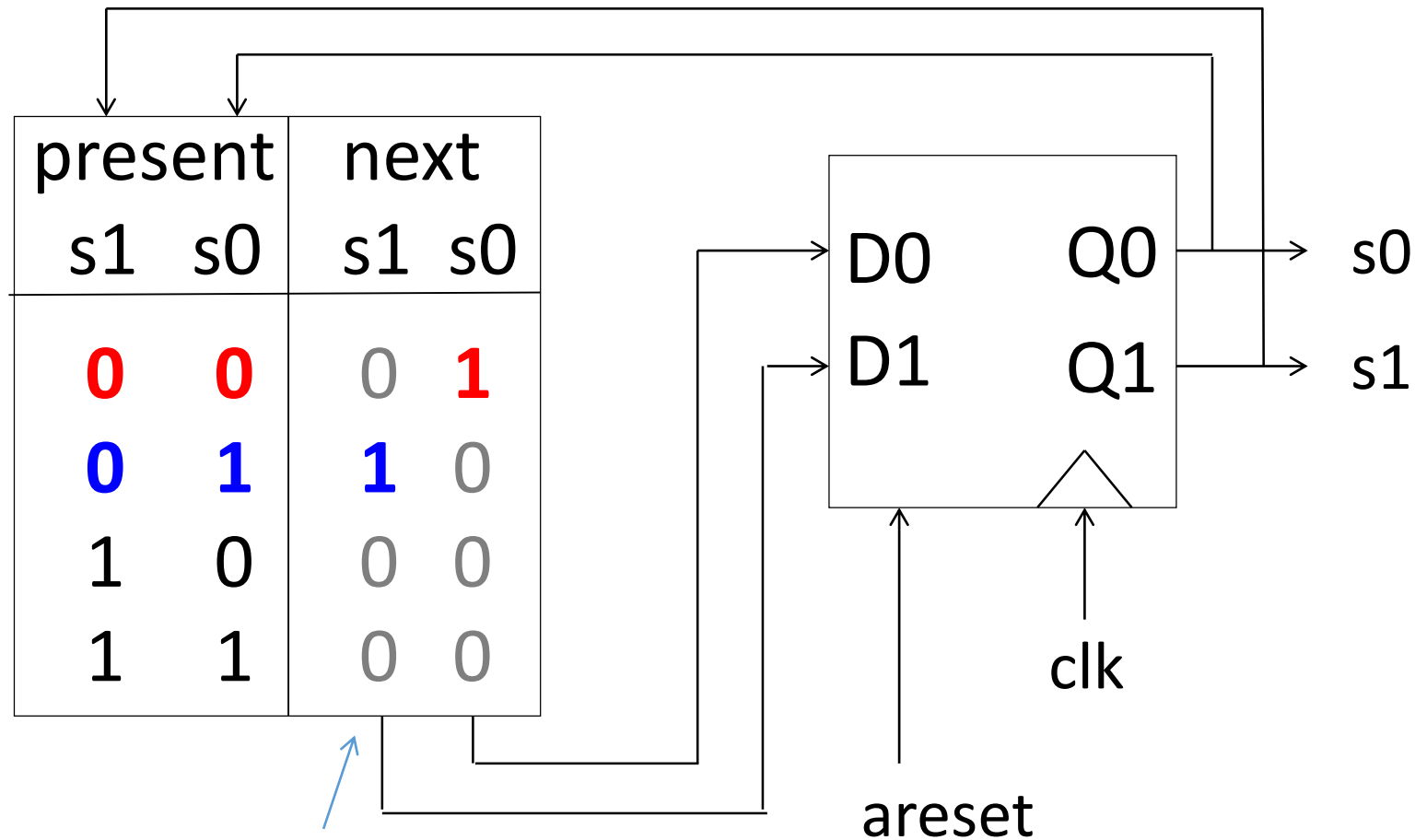
areset

We set this don't care value also to 0.

Structural diagram of the FSM:
State-transition function,
storage elements, and feedback of state.

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

next s1 = (~s1) & s0

next s0

next s1

D0 Q0 → s0

D1 Q1 → s1

areset clk
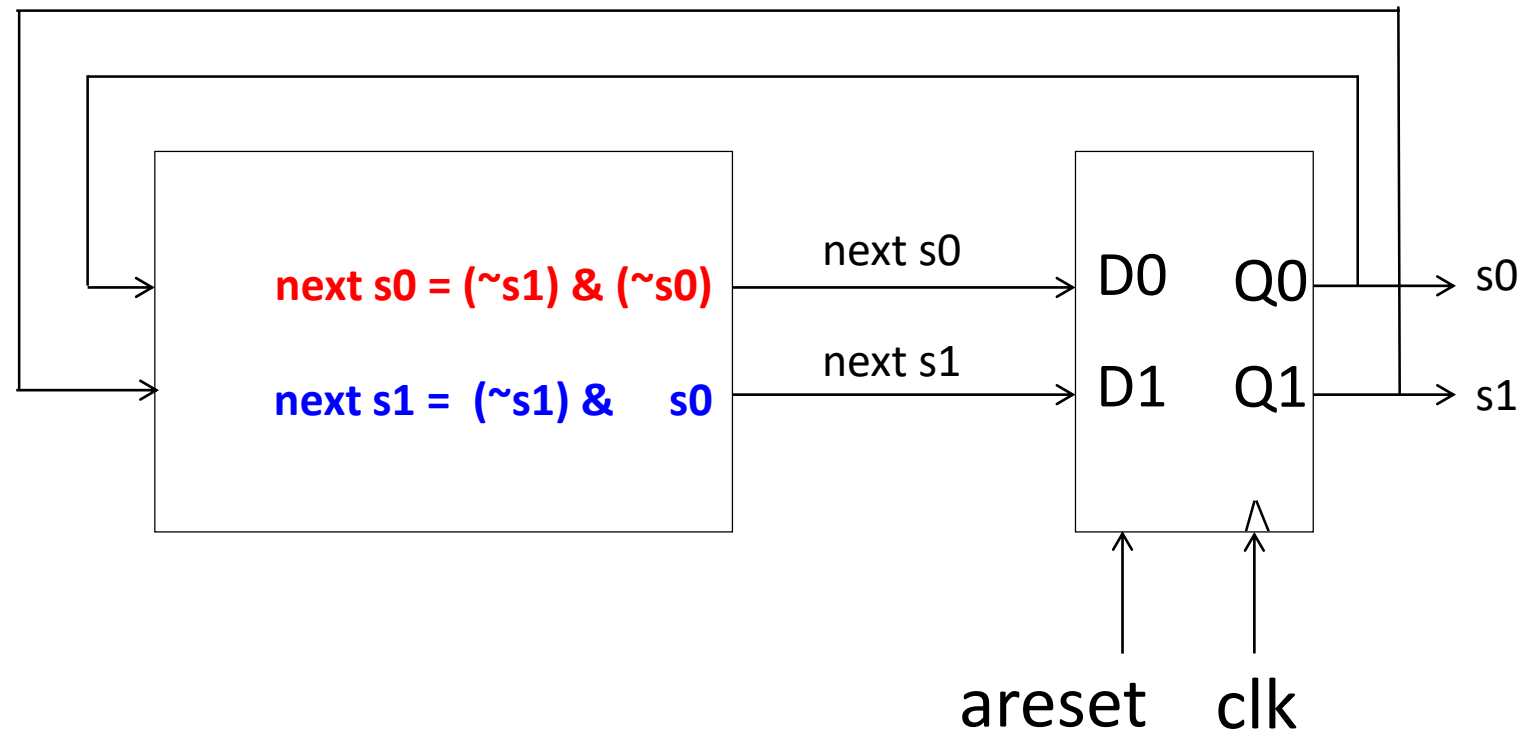
Essence so far

State diagram:

"next state" is a function of (present) "state"

State transitions:

the "next state" becomes (the present) "state"
on the rising edge of the clock

The FSM modeled and simulated with Digital
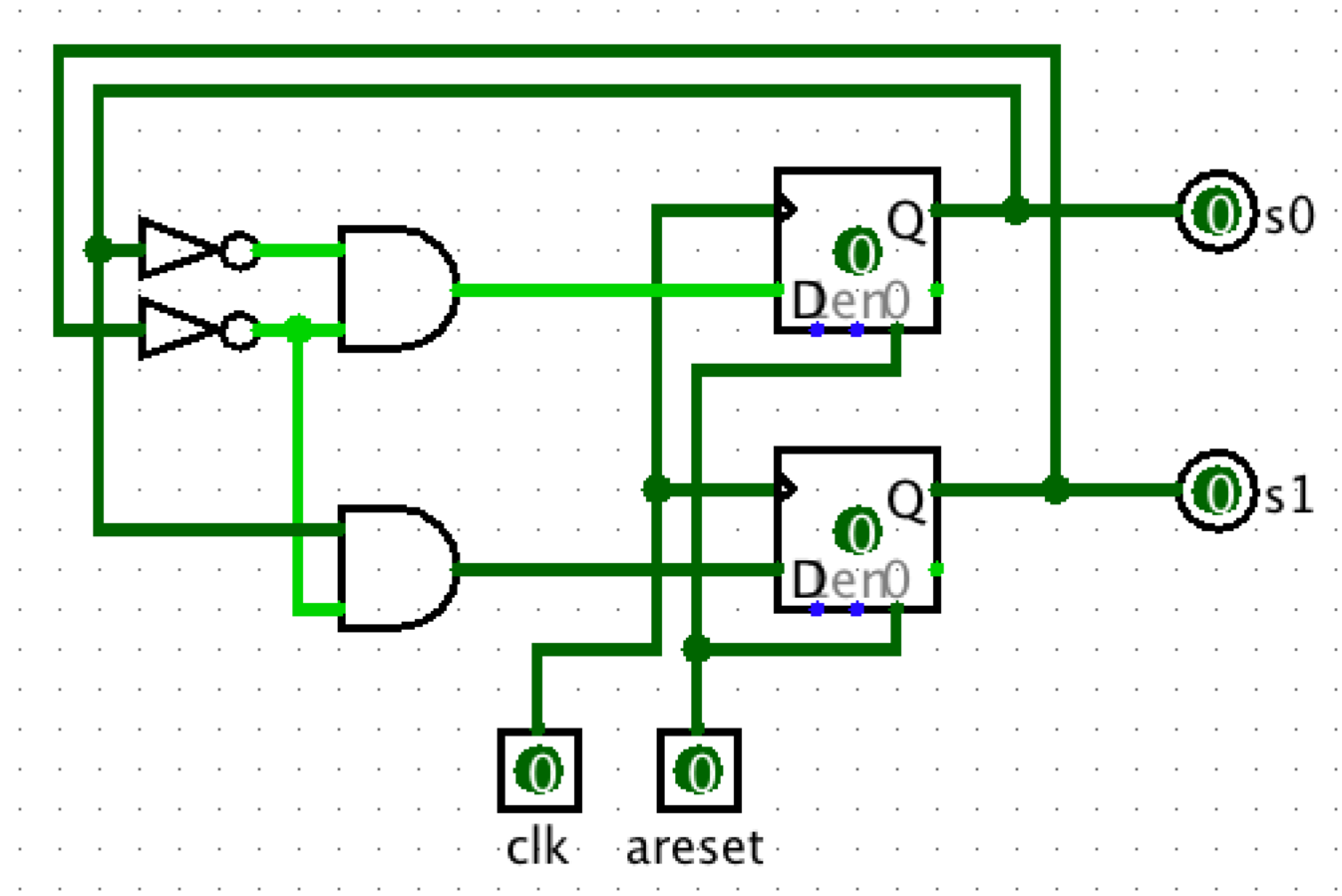
For an example of this FSM see

fsm_moore_no_input

Inputs

FSMs can 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.

A

in == 0

C

B

in == 1
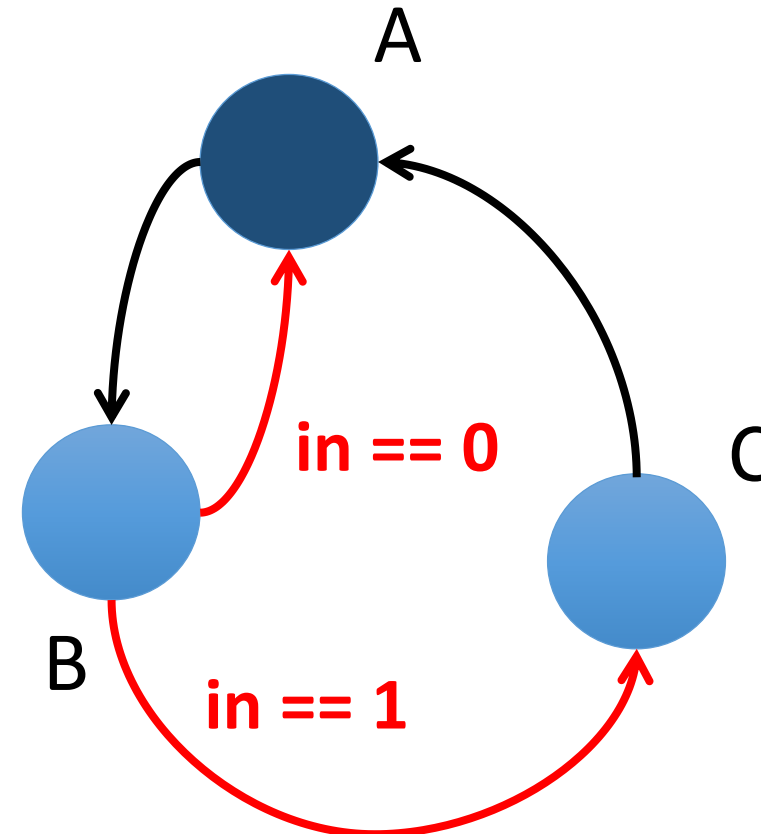
FSMs can also have **inputs** influencing the transition to the next state

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.
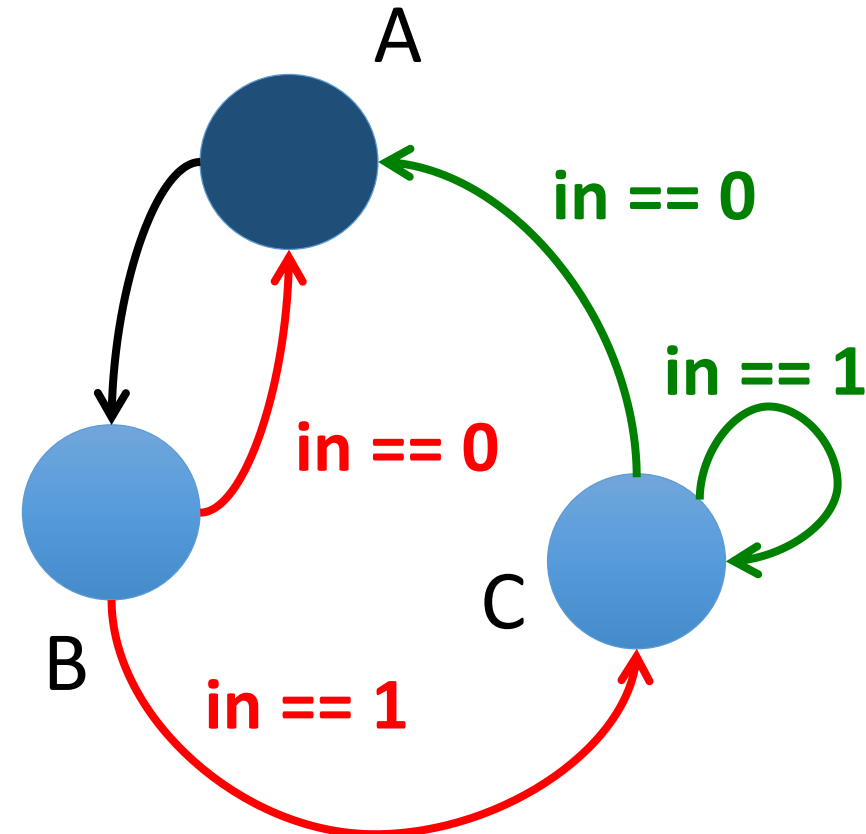
The state transition table

present in state		next state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	C
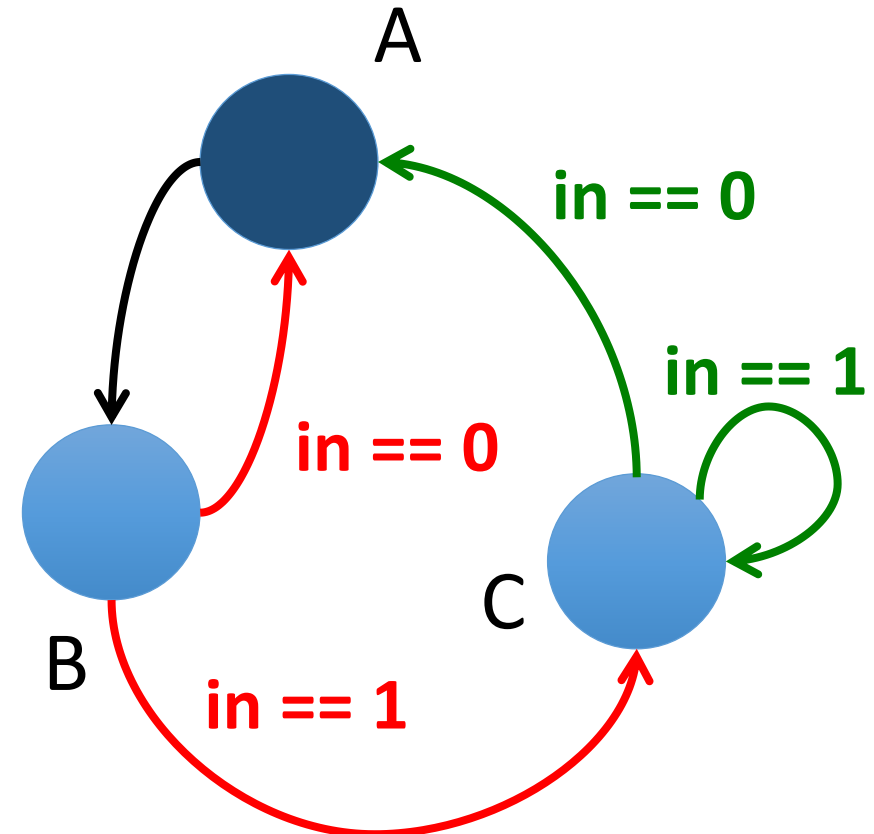
Timing diagram. Example 1
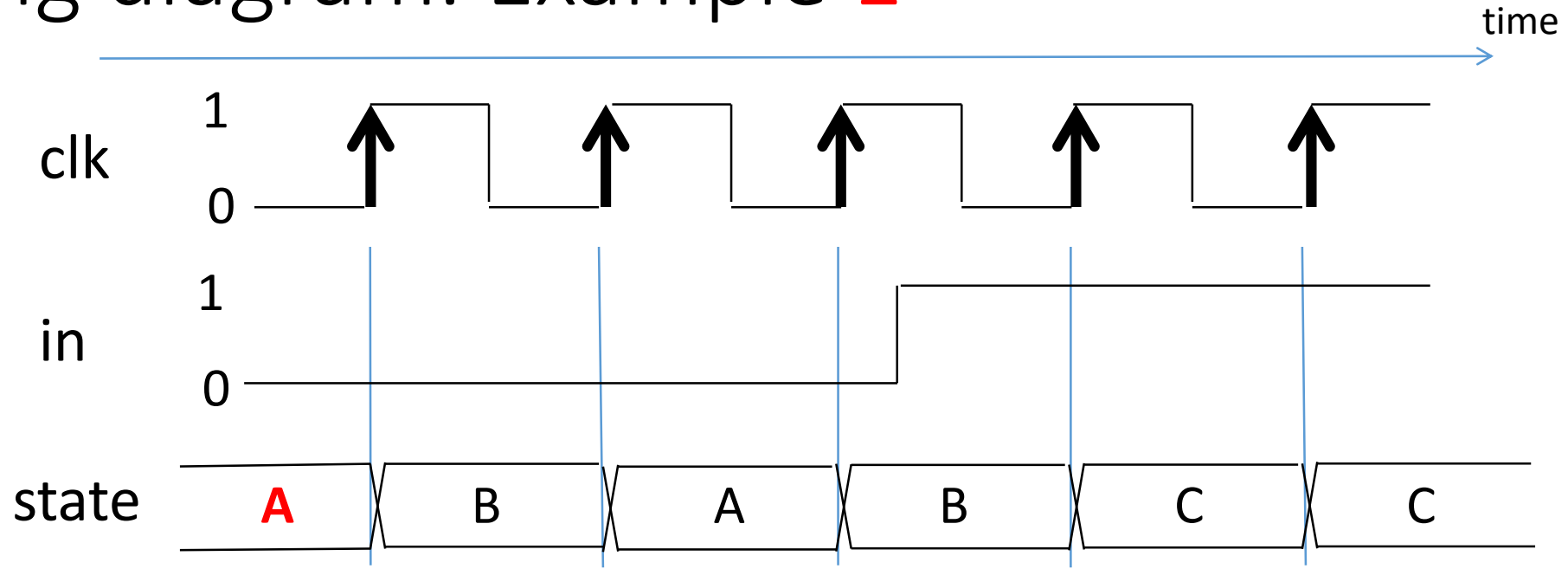
For a timing diagram, we need to choose values for the input signal "in". Only after some choice for "in" we can derive the sequence of states from the state diagram.

Timing diagram. Example 2
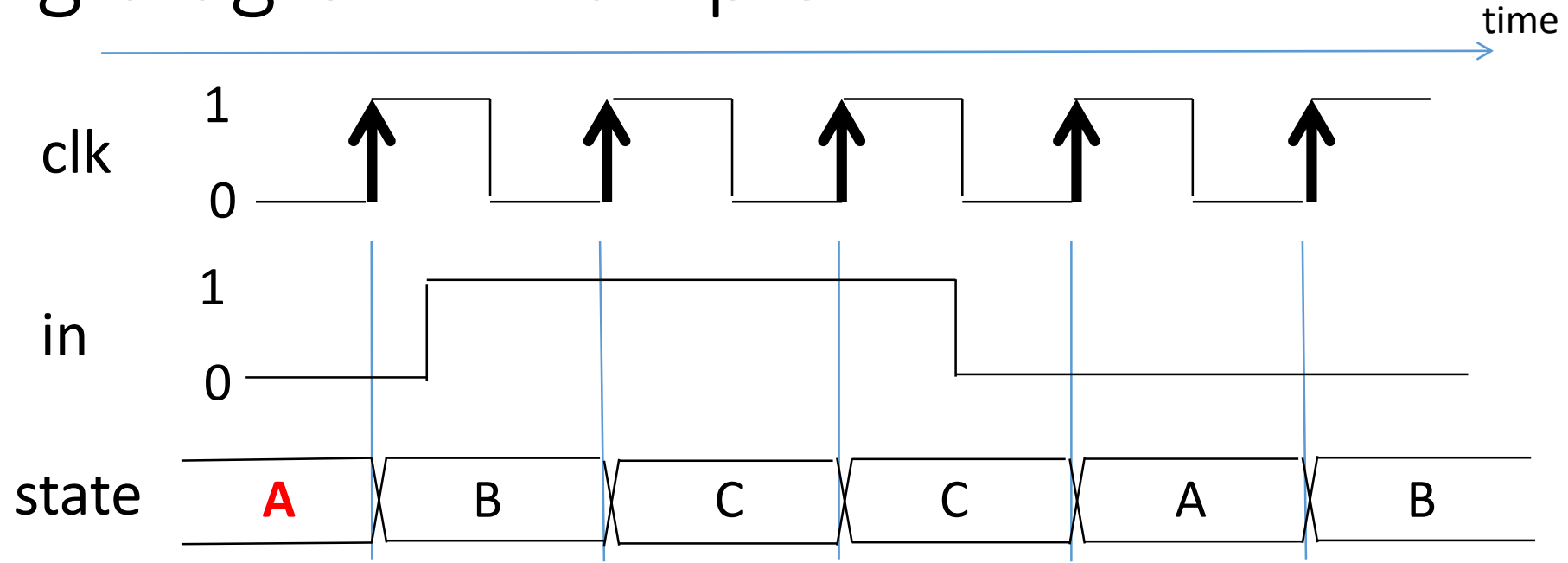
time

clk

1

0

in

1

0

state   **A**   B   C   C   A   B

For a timing diagram, we need to choose values for the input signal "in". Only after some choice for "in" we can derive the sequence of states from the state diagram.

Timing diagram. Example 3
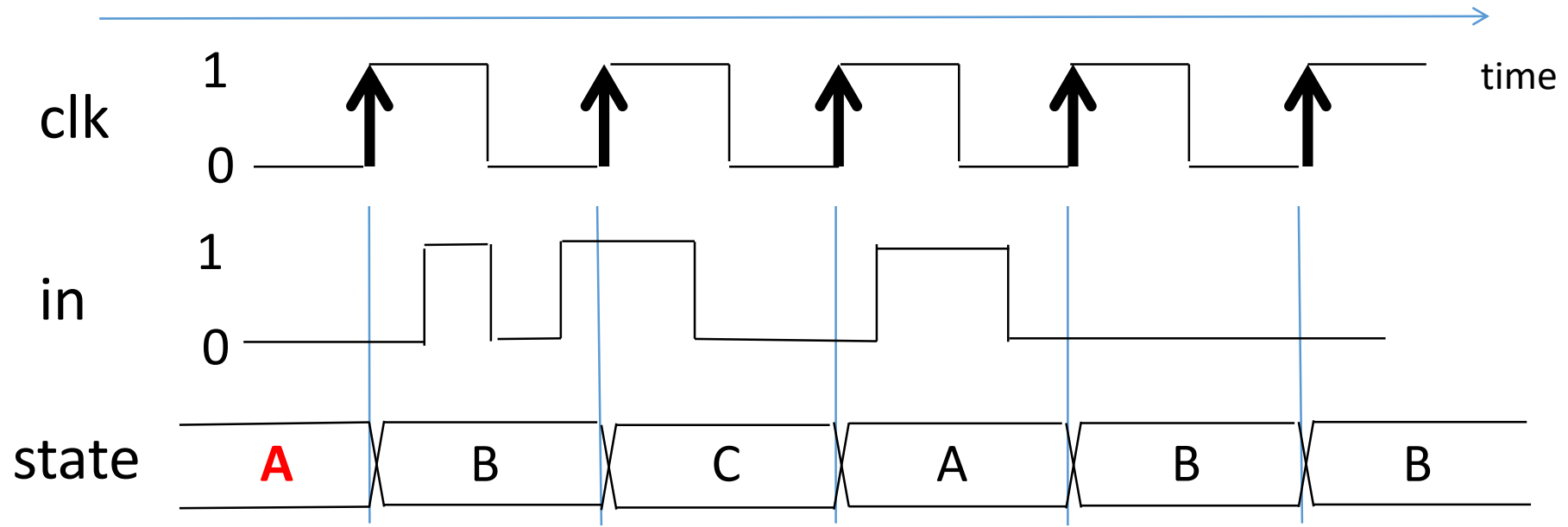
For a timing diagram, we need to choose values for the input signal "in". Only after some choice for "in" we can derive the sequence of states from the state diagram.

Binary state encoding: Instead of symbolic state names we use numbers

present state		in	next state	
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
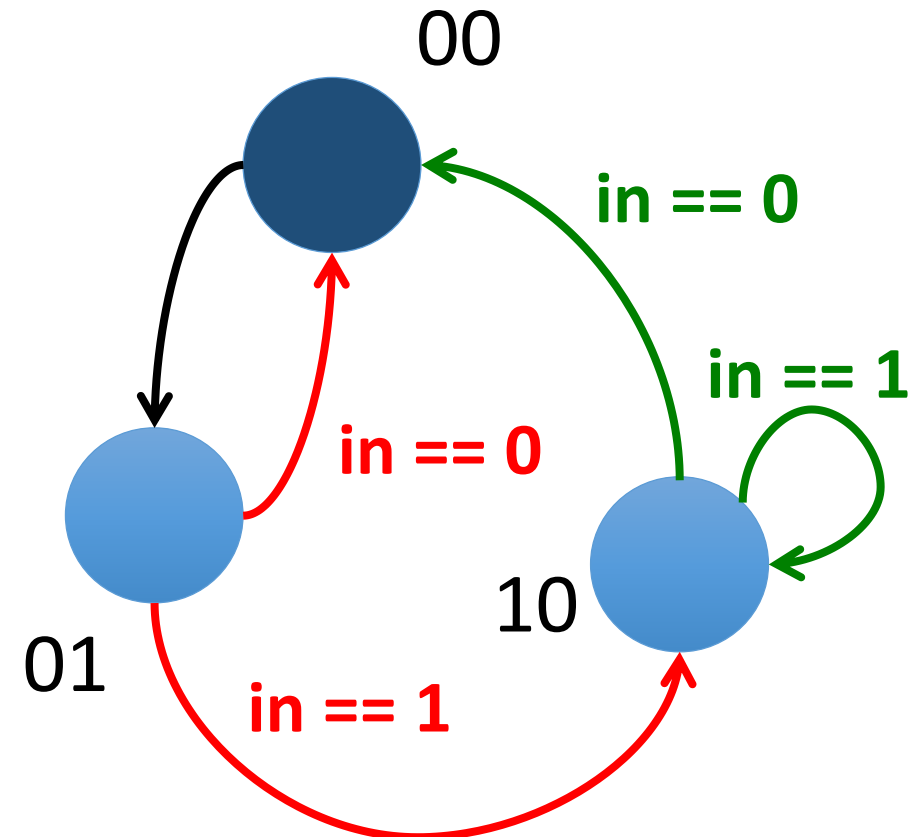
"11" does not exist: We use "Don't Care" as the following state

present state		in	next state	
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
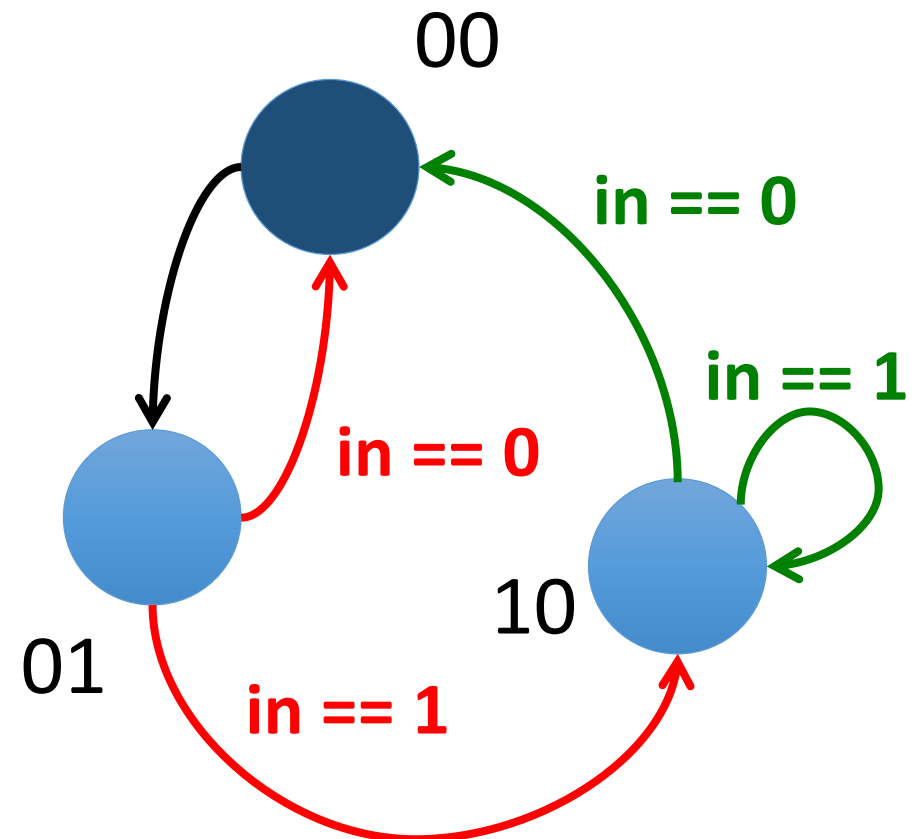
I have ordered the lines in the state transition table from 0 to 7. This makes it easier to "read" the table.

	present state		in	next state	
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	0	0
3	0	1	1	1	0
4	1	0	0	0	0
5	1	0	1	1	0
6	1	1	0	x	x
7	1	1	1	x	x

00

01

10

in == 0

in == 1

in == 0

in == 1
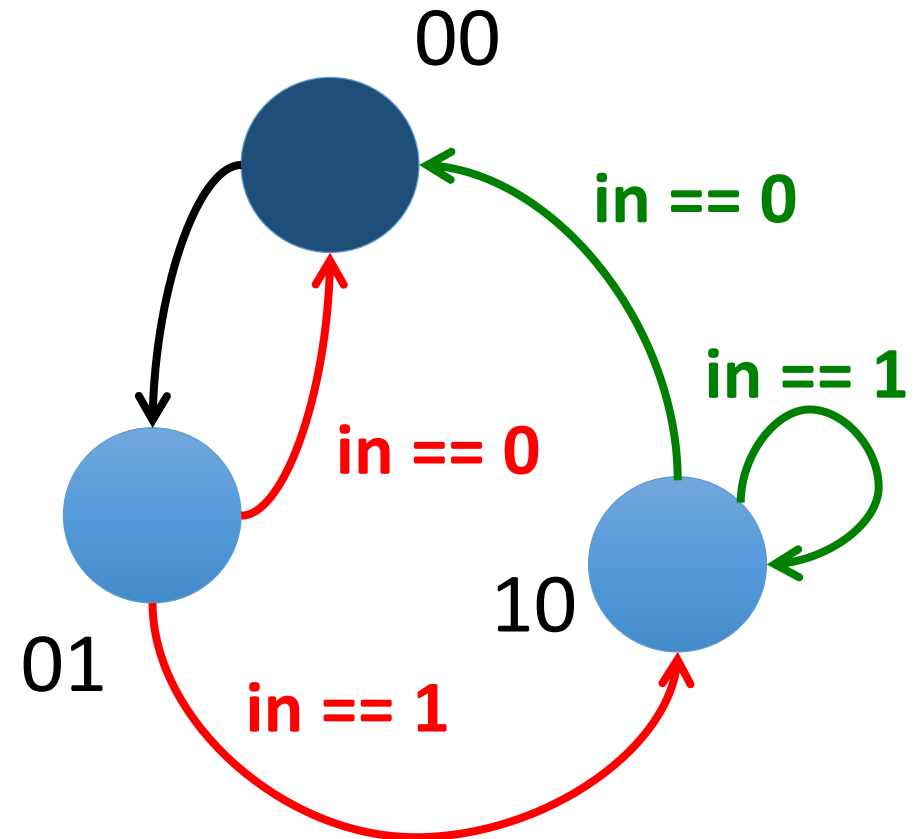
We call the state bits "s1" and "s0"

	present		in	next	
	s1	**s0**		**s1**	**s0**
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	0	0
3	0	1	1	1	0
4	1	0	0	0	0
5	1	0	1	1	0
6	1	1	0	x	x
7	1	1	1	x	x
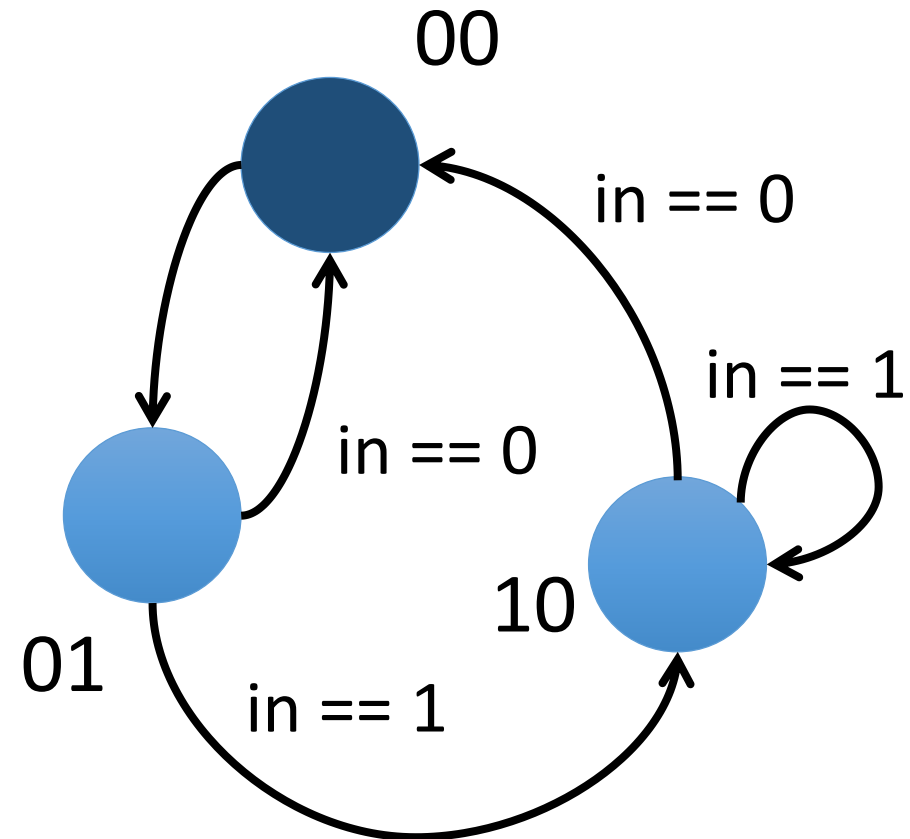
00

in == 0

in == 1

in == 0

01

10

in == 1

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

	present		in	next	
	s1	s0		s1	s0
0	**0**	**0**	**0**	0	**1**
1	**0**	**0**	**1**	0	**1**
2	0	1	0	0	0
3	0	1	1	1	0
4	1	0	0	0	0
5	1	0	1	1	0
6	1	1	0	x	0
7	1	1	1	x	0
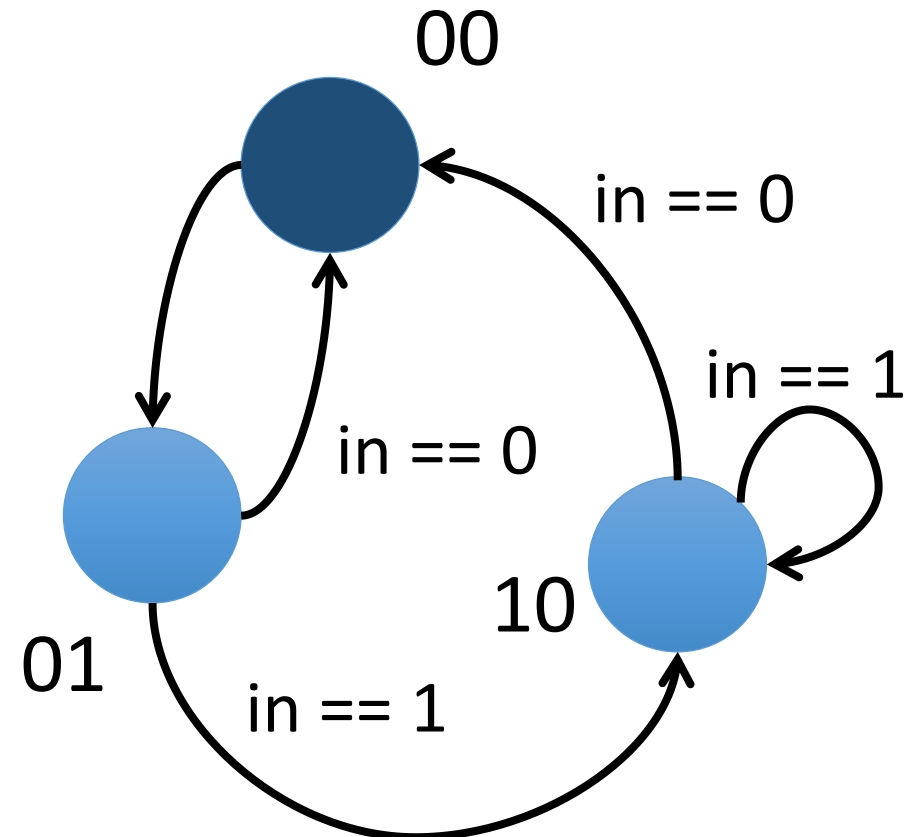
next s1 = ((~s1) &   s0 & in)
| (s1 & (~s0) & in)

	present s1 s0	in	next s1 s0
0	0 0	0	0 1
1	0 0	1	0 1
2	0 1	0	0 0
3	0 1	1	1 0
4	1 0	0	0 0
5	1 0	1	1 0
6	1 1	0	0 0
7	1 1	1	0 0

00

in == 0

in == 1

in == 0

01

10

in == 1

Structural diagram of the FSM

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

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

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

| (s1 & (~s0) & in)

next s0 → D0   Q0 → s0

next s1 → D1   Q1 → s1

in →

areset   clk
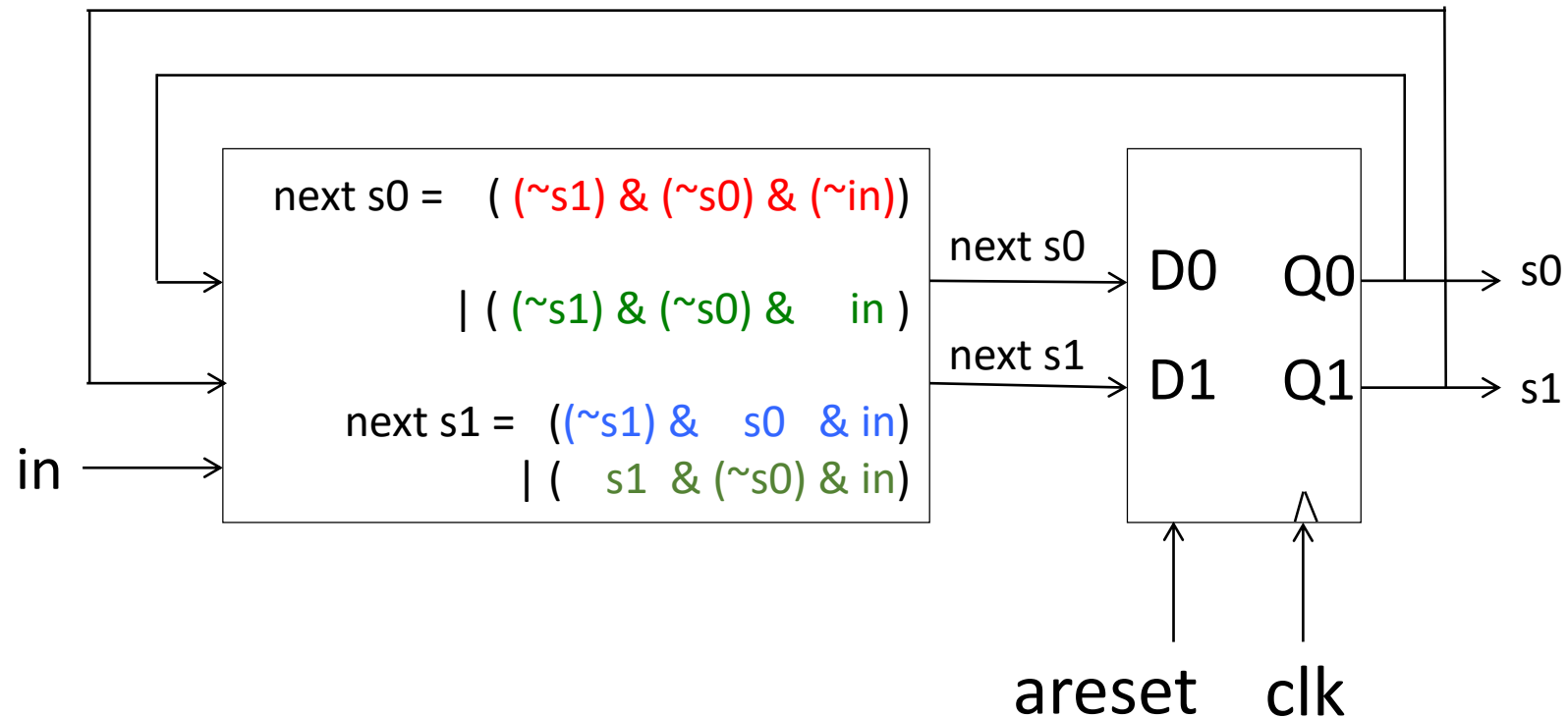
Implementation with Digital

For an example of this FSM see

fsm_moore_no_output_function

Outputs

FSMs typically also have outputs

In this example we see
that 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)
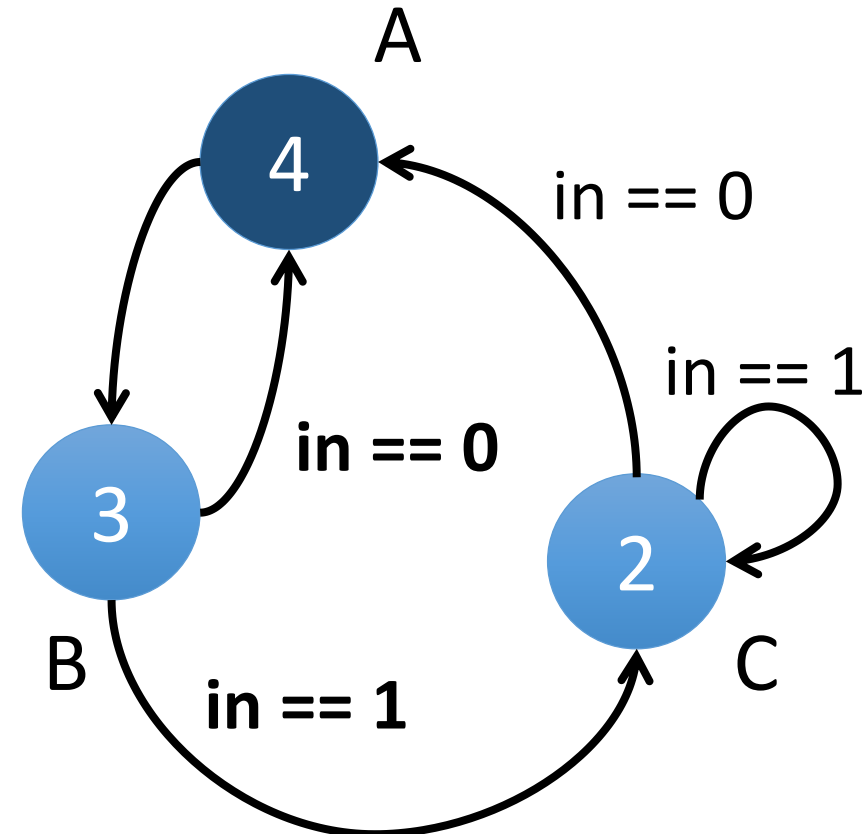
We define the outputs with the "output function"

Moore machines:

output = f(state)

state	output
A	4
B	3
C	2
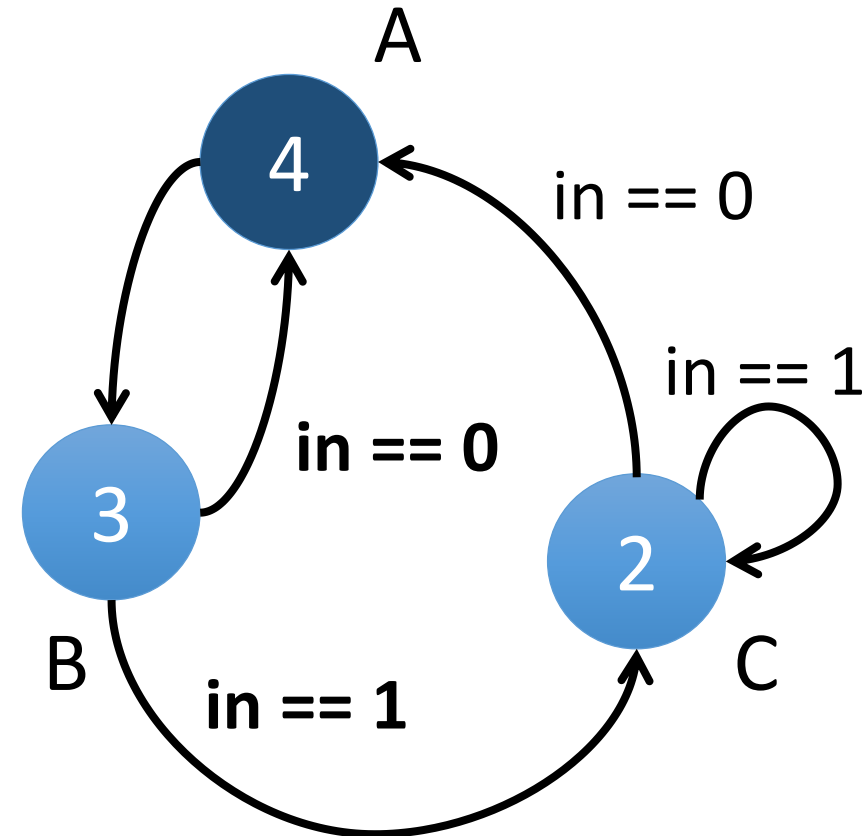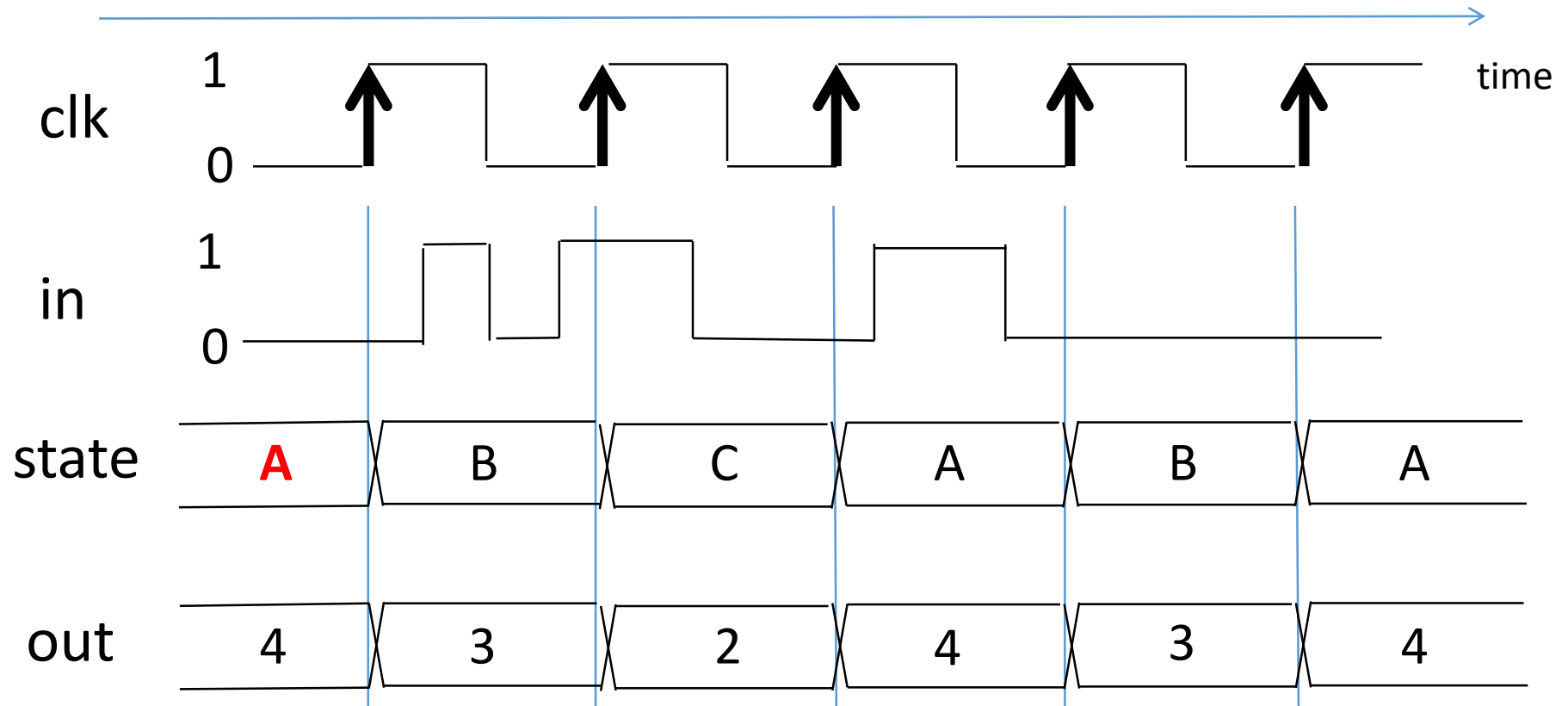
A

4

in == 0

in == 1

in == 0

3

2

B

C

in == 1

Timing diagram. Example 3

For a timing diagram, we need to choose values for the input signal "in". Only after some choice for "in" we can derive the sequence of states from the state diagram.

We define the outputs with binary values

Moore machines:

output = f(state)

state	output
A	100	(=4)
B	011	(=3)
C	010	(=2)

A

100

in == 0

in == 1

in == 0

011

010

B

C

in == 1
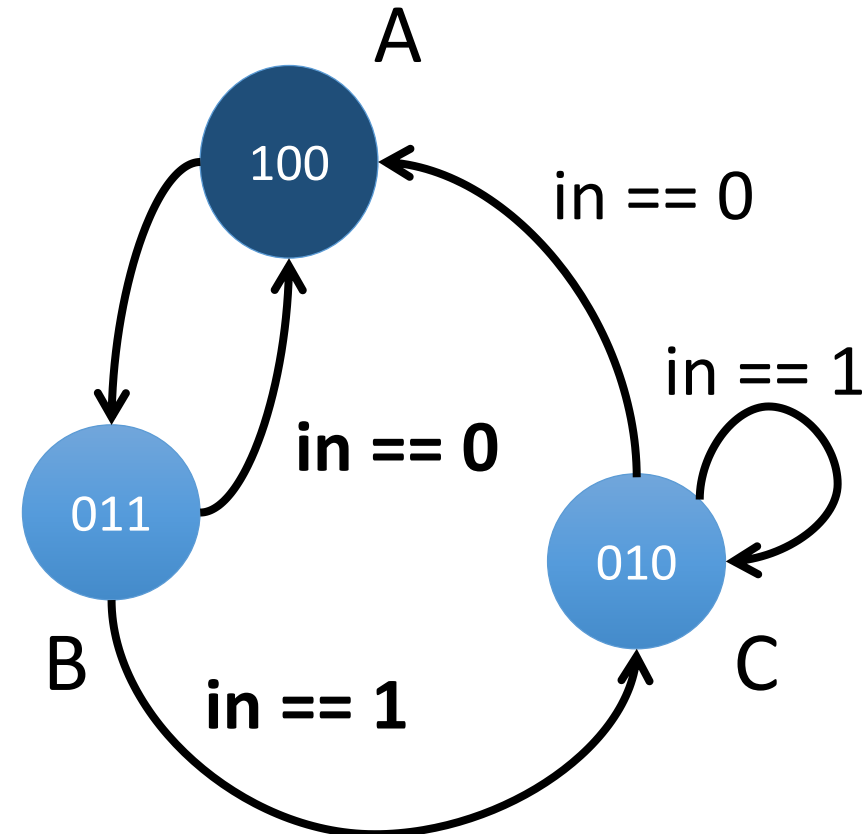
We define the outputs with binary values

Moore machines:

output = f(state)

state	o2 o1 o0
A	1 0 0
B	0 1 1
C	0 1 0

A

100

in == 0

in == 0

011

in == 1

010

B

C

in == 1
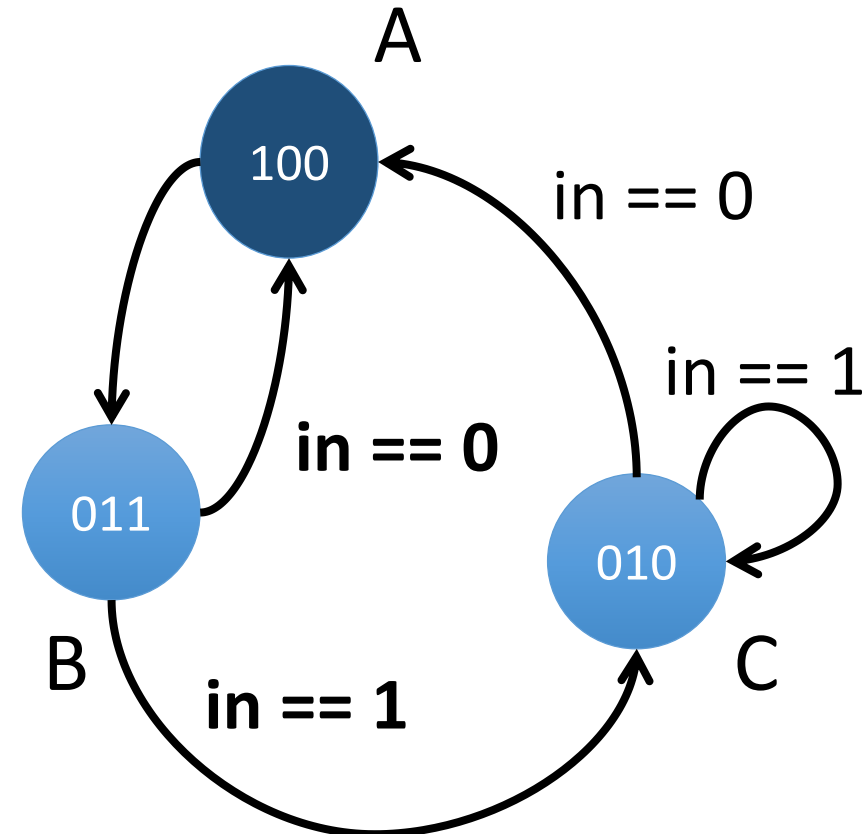
For an example of this FSM see

fsm_moore_with_output_function

Binary state encoding: We define the outputs with binary values

state	o2 o1 o0
0 0	1 0 0
0 1	0 1 1
1 0	0 1 0

A
100

in == 0

in == 1

011
in == 0

B
in == 1

010
C
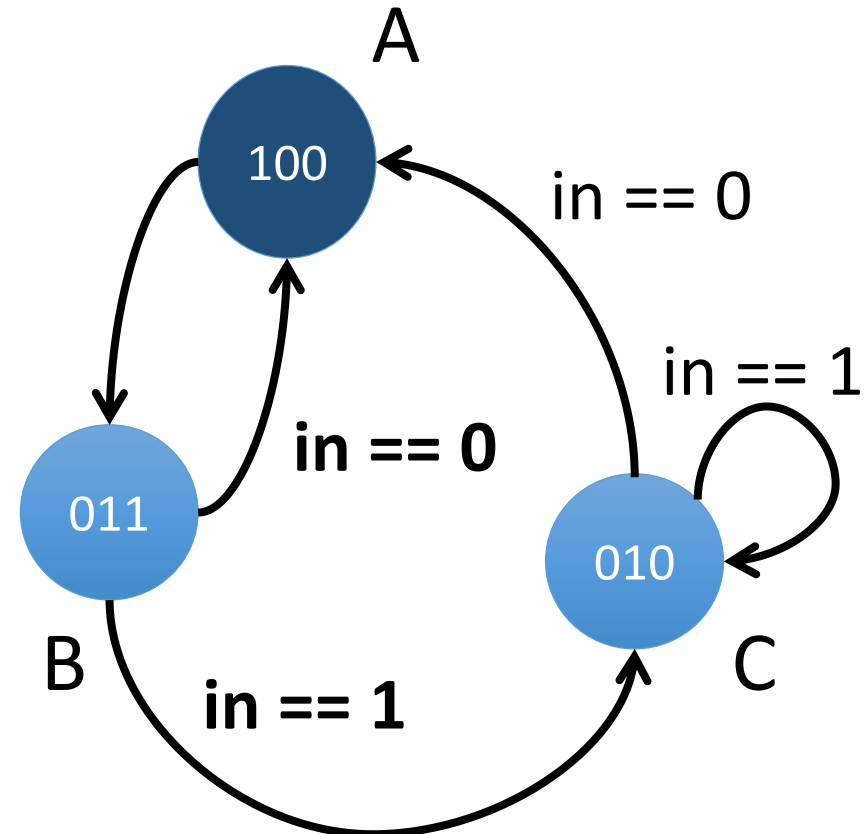
We define the outputs with binary values

o2 = ~s1 & ~s0

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

A

100

in == 0

in == 1

in == 0

011

010

B

in == 1

C
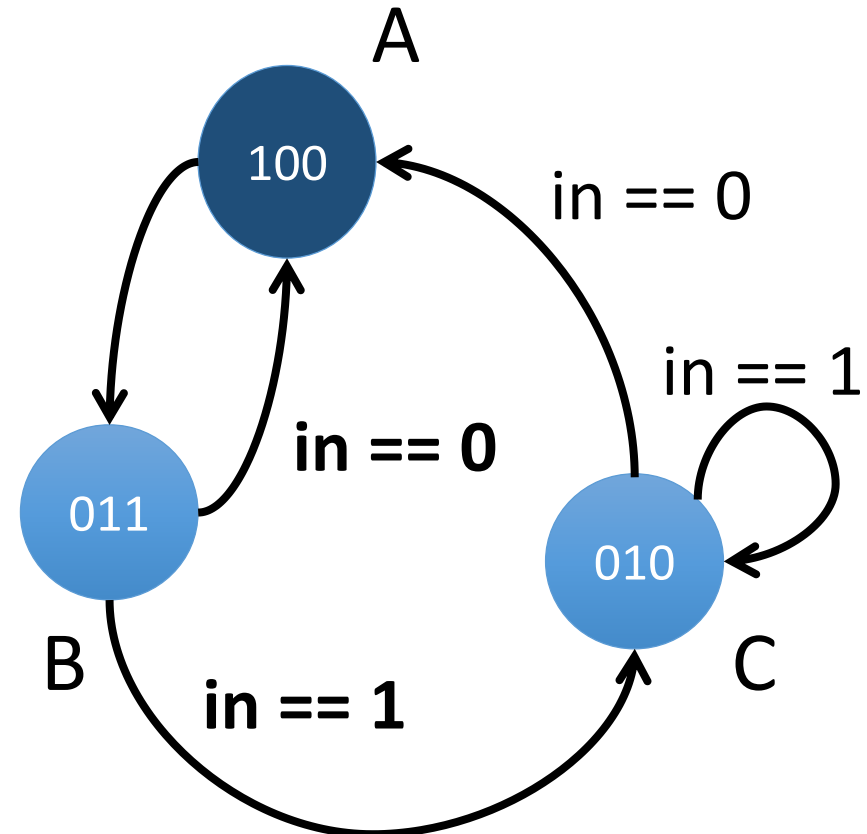
We define the outputs with binary values

o2 = ~s1 & ~s0

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

s1	s0	o2	**o1**	o0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	0
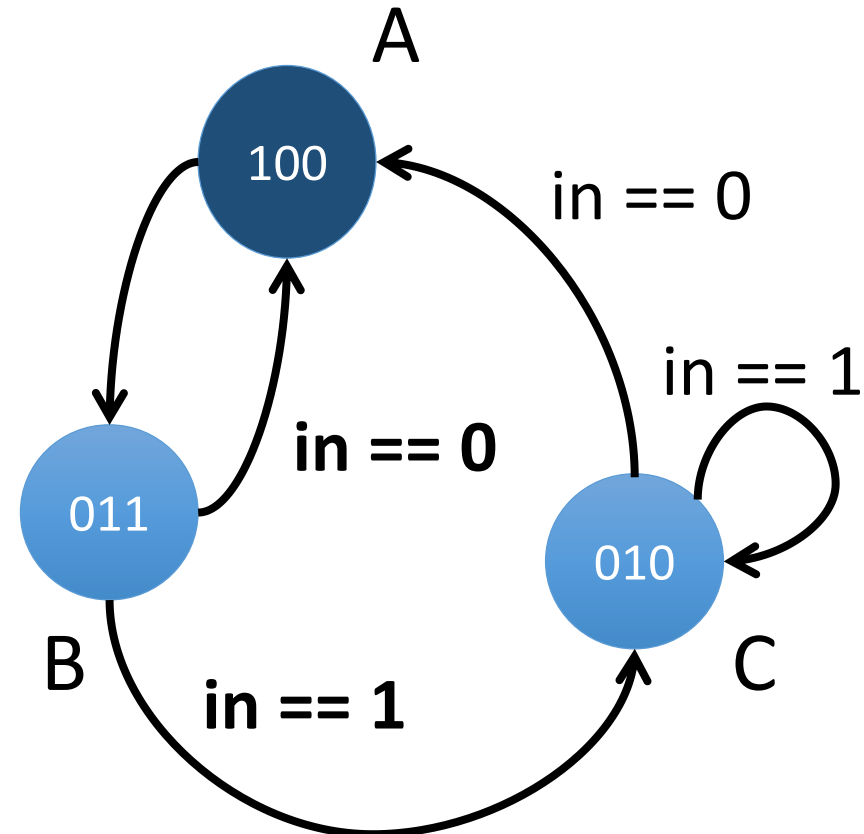
We define the outputs with binary values

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

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

A
100

in == 0

in == 1

011

in == 0

B

010

C

in == 1
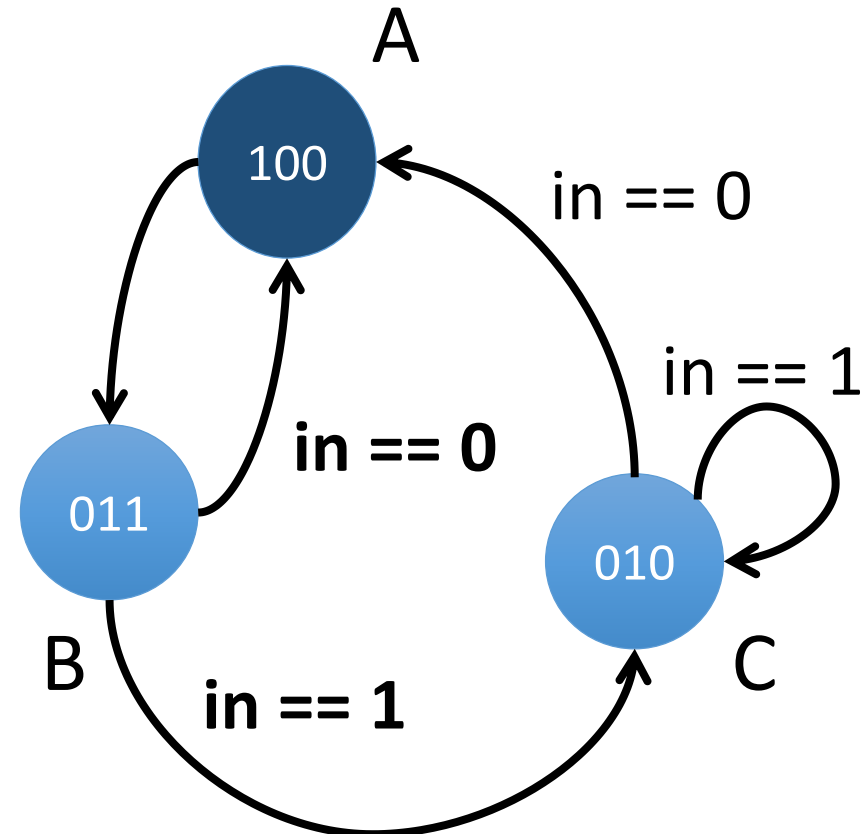
Structural diagram of the FSM
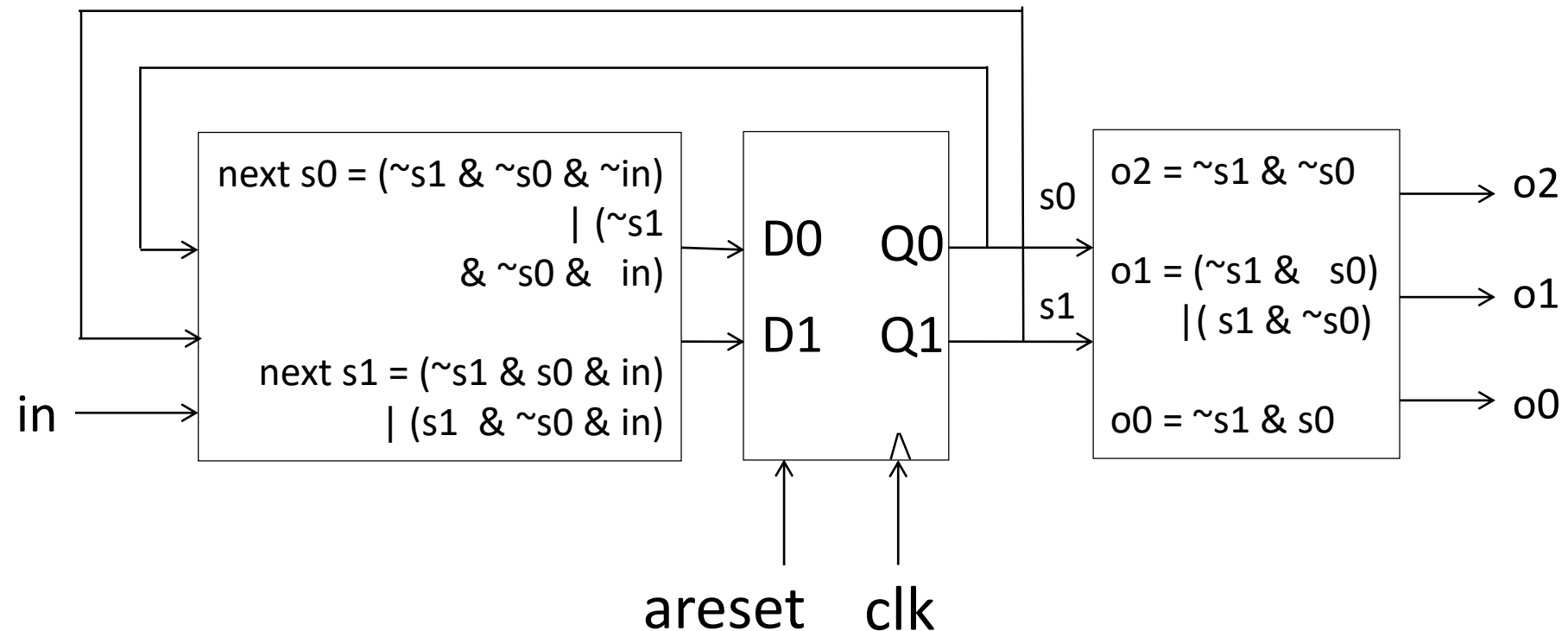
Implementation with Digital
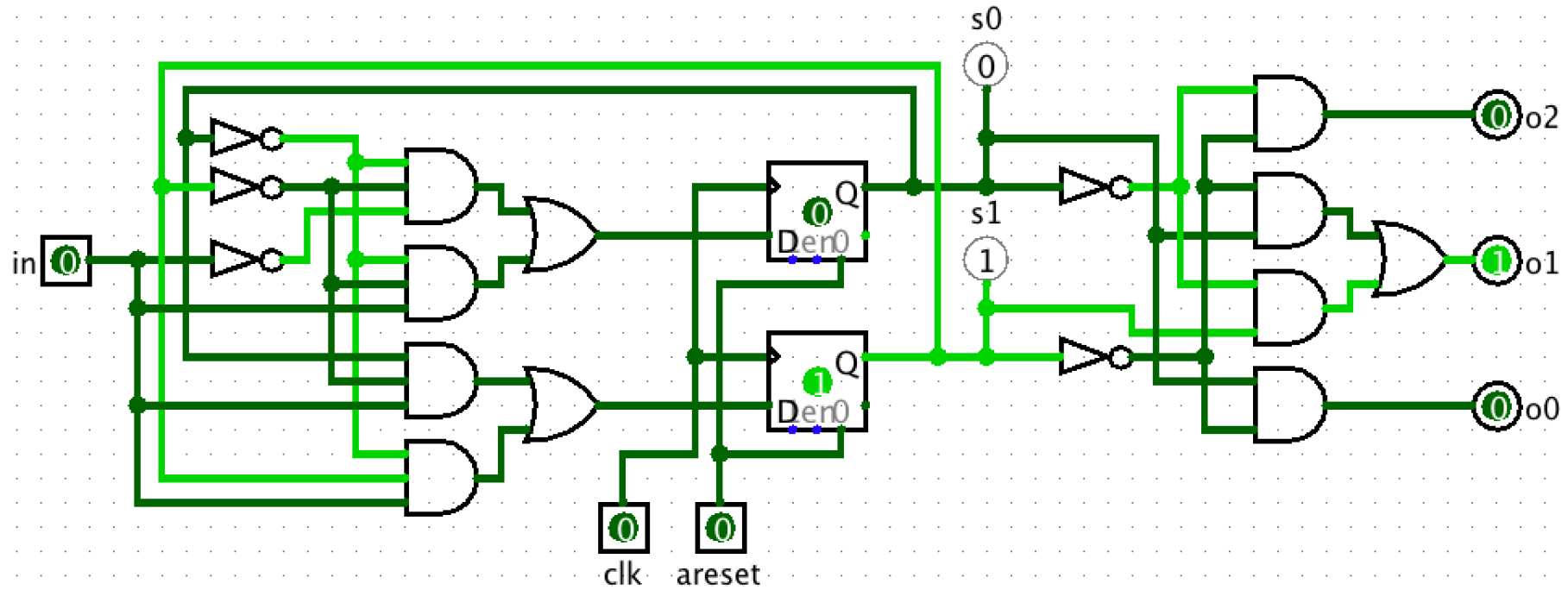
Essence of Moore Machines
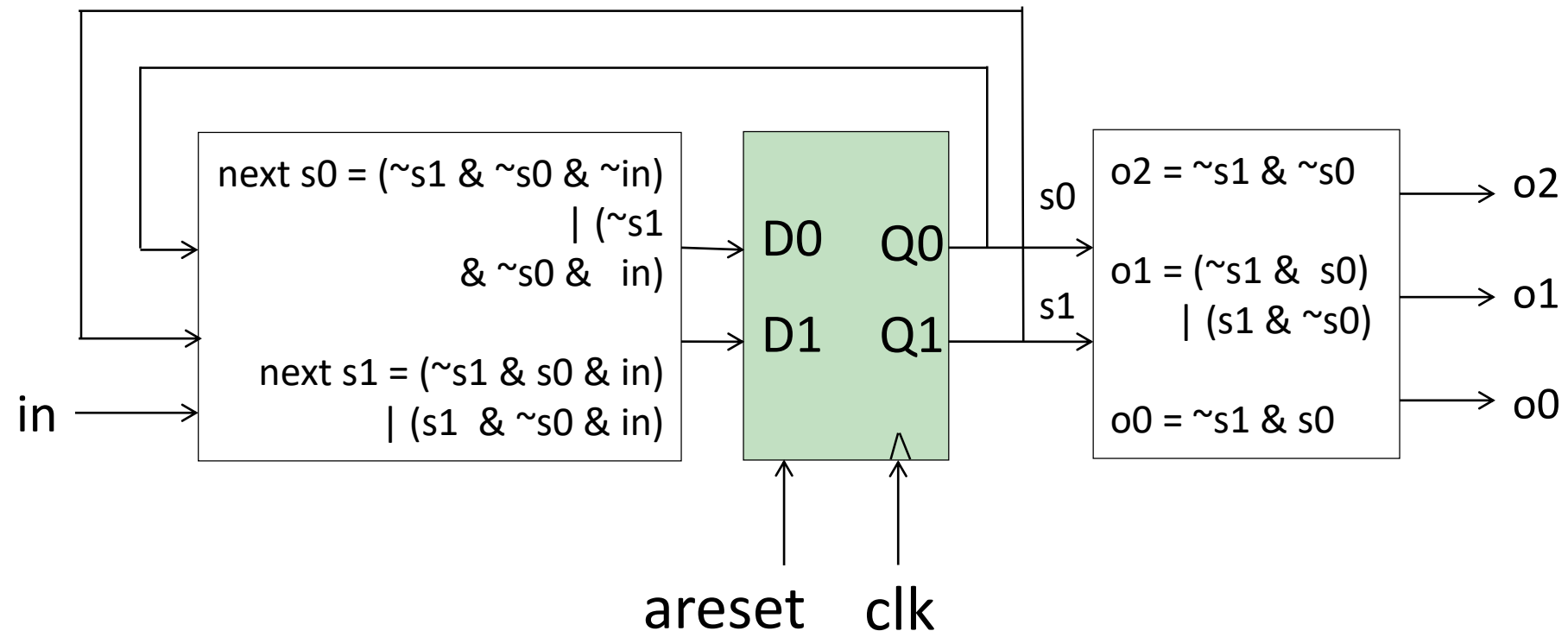
the state is stored in a register

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

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

in

D0   Q0

D1   Q1

areset   clk

s0

s1

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

o2

o1

o0

Essence of Moore Machines

the state is stored in a register

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

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

in

D0   Q0   s0

D1   Q1   s1

o2 = ~s1 & ~s0   o2

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

o0 = ~s1 & s0   o0

areset   clk

a state transition is caused by the clock signal.

Essence of Moore Machines

**the state is
stored
in a register**

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

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

in

D0   Q0

D1   Q1

s0

s1

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

o2

o1

o0
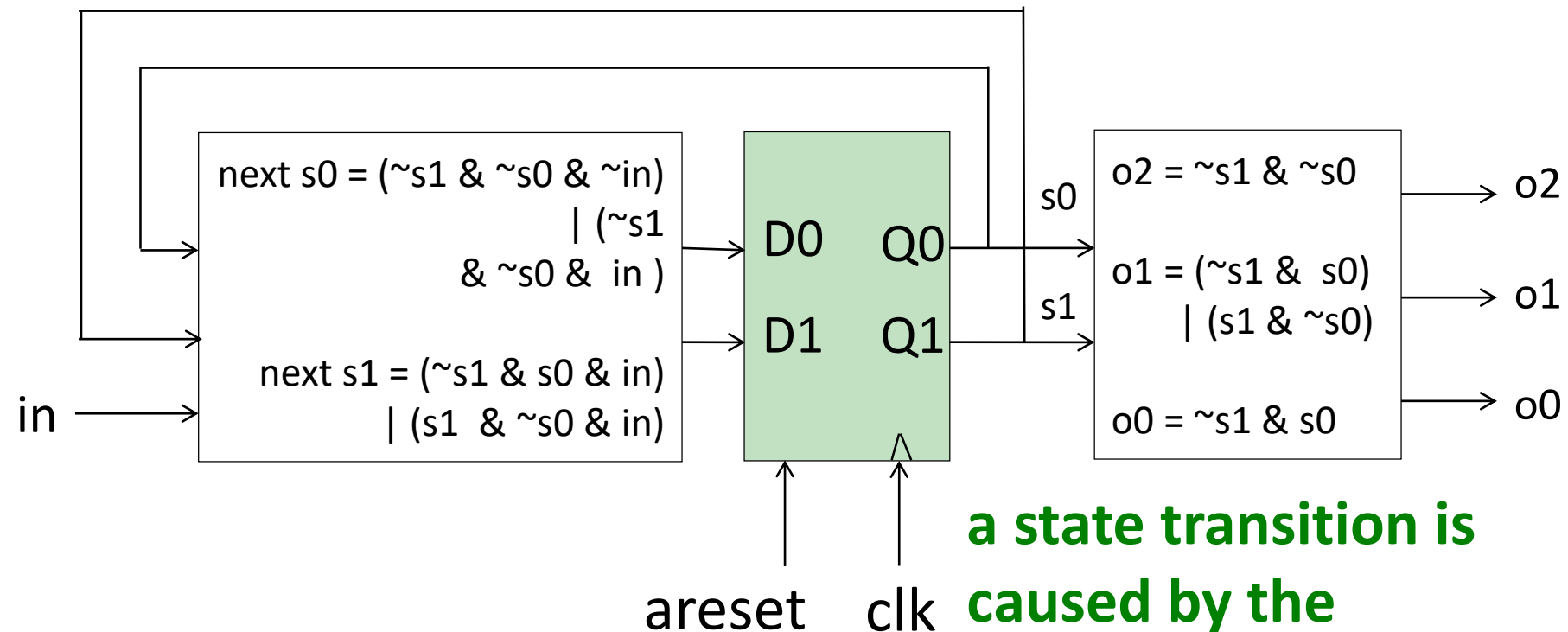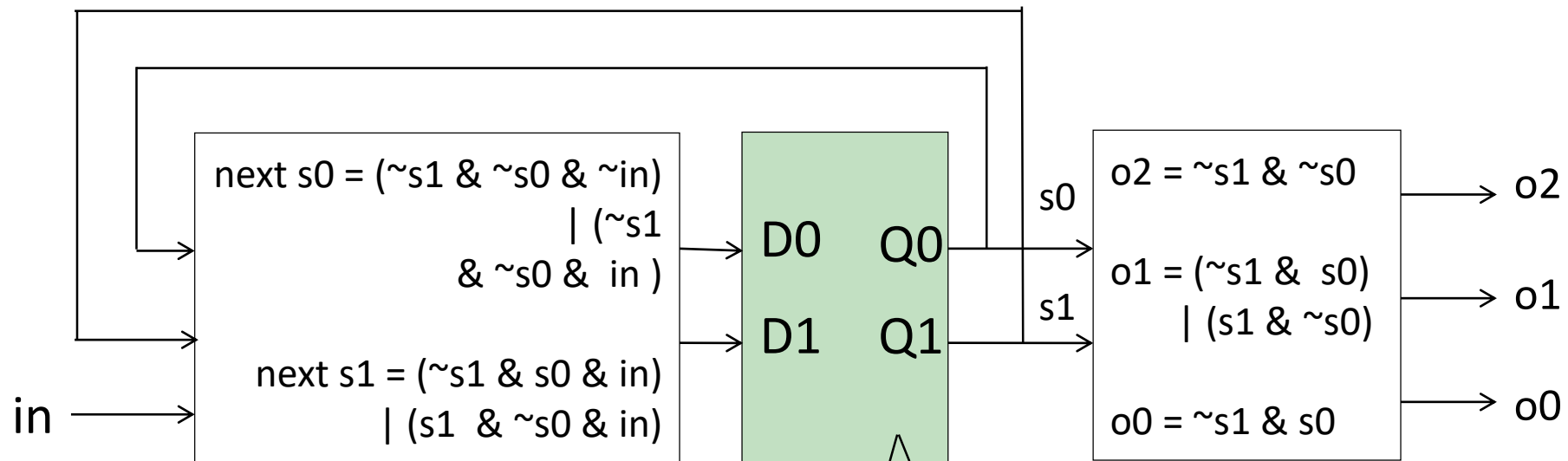
**With "areset" we can
initialize the ASM
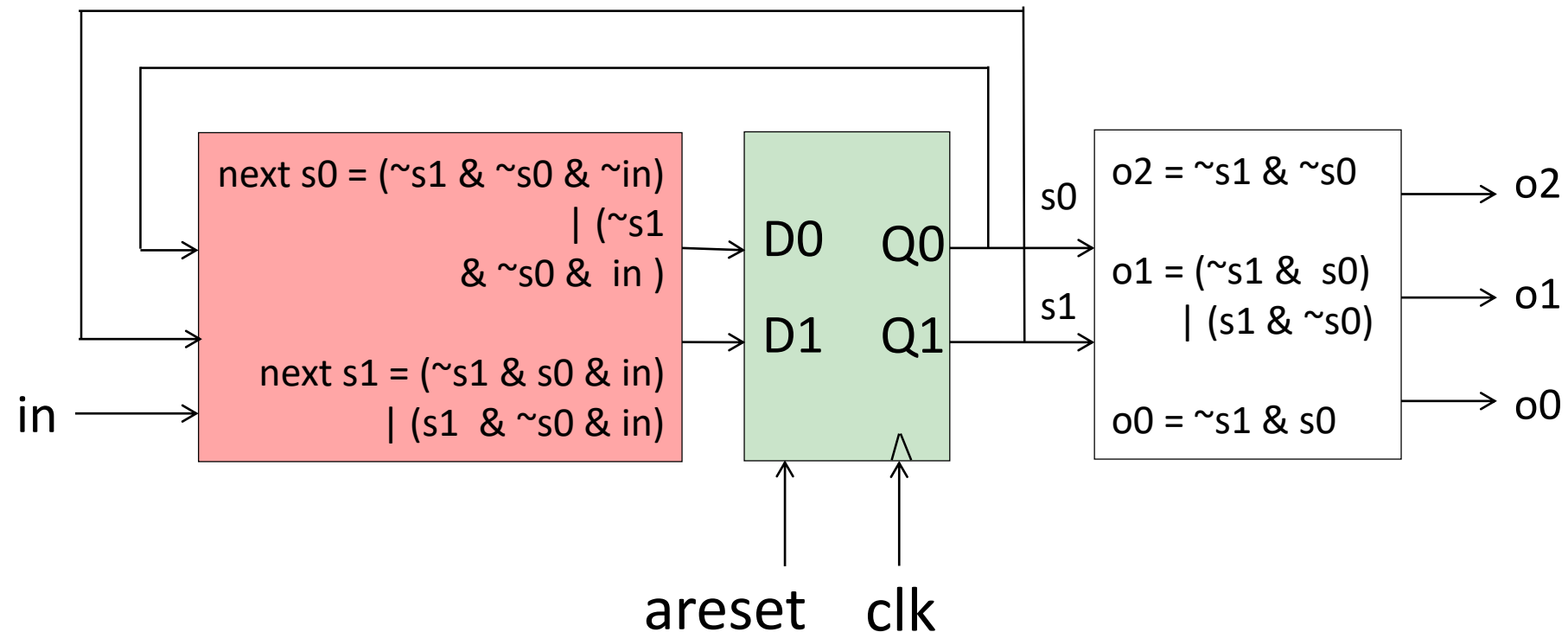("initial state").**

areset   clk

**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)**

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

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

in

D0   Q0

D1   Q1

s0

s1

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

o2

o1

o0

areset   clk

Essence of Moore Machines

With the output function we compute the output values:

output = g(state)
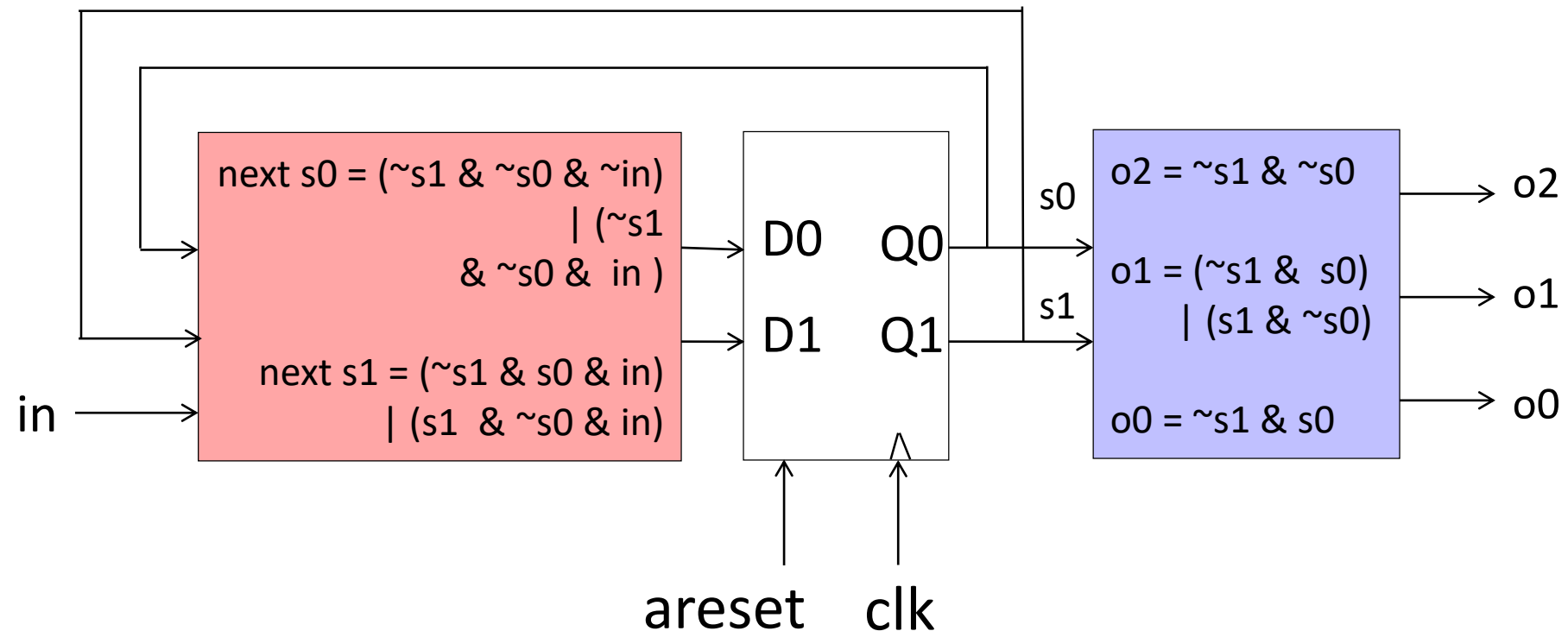
Essence of Moore Machines

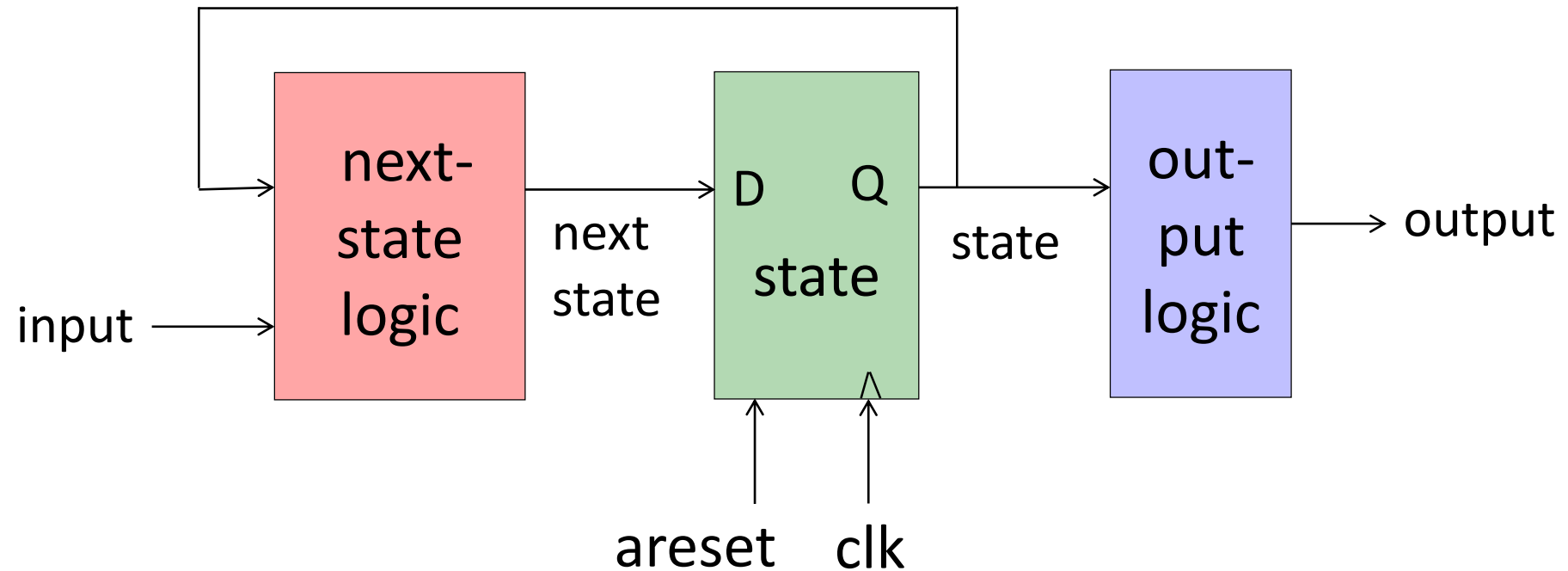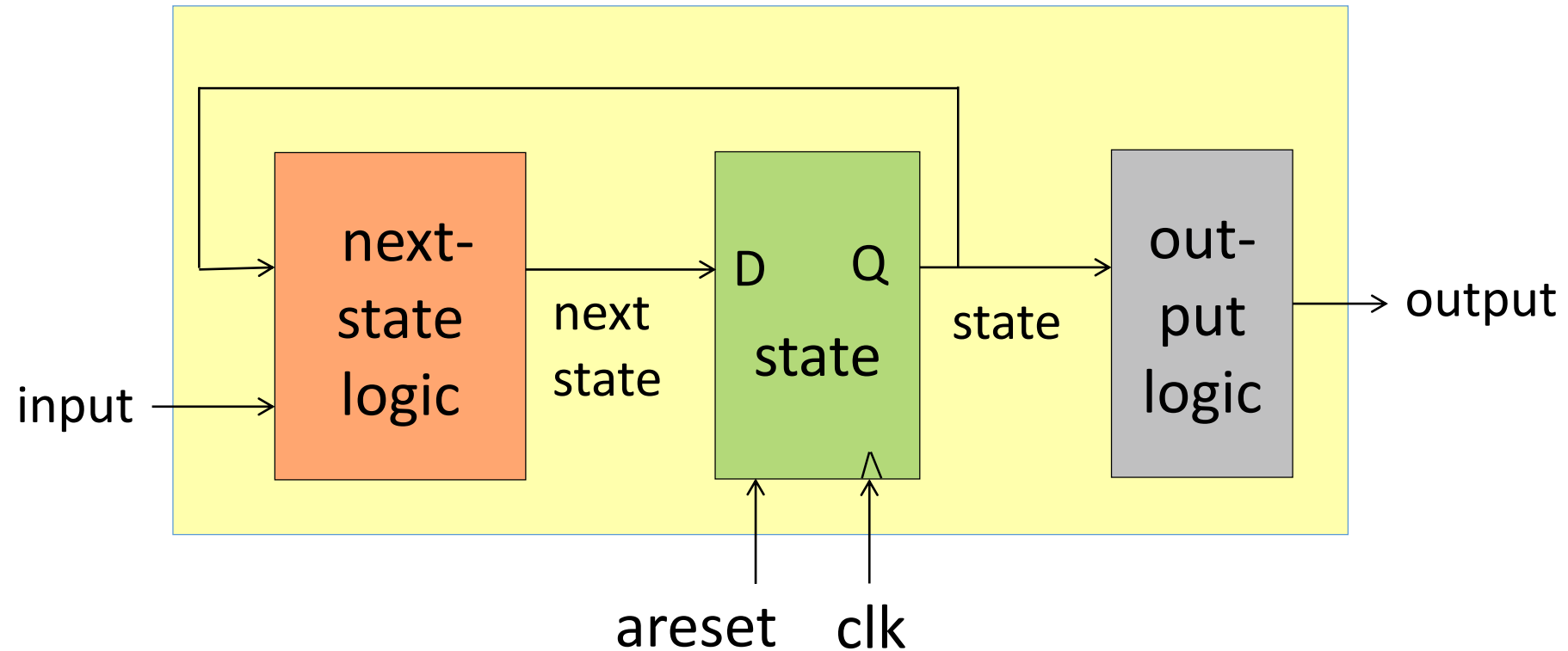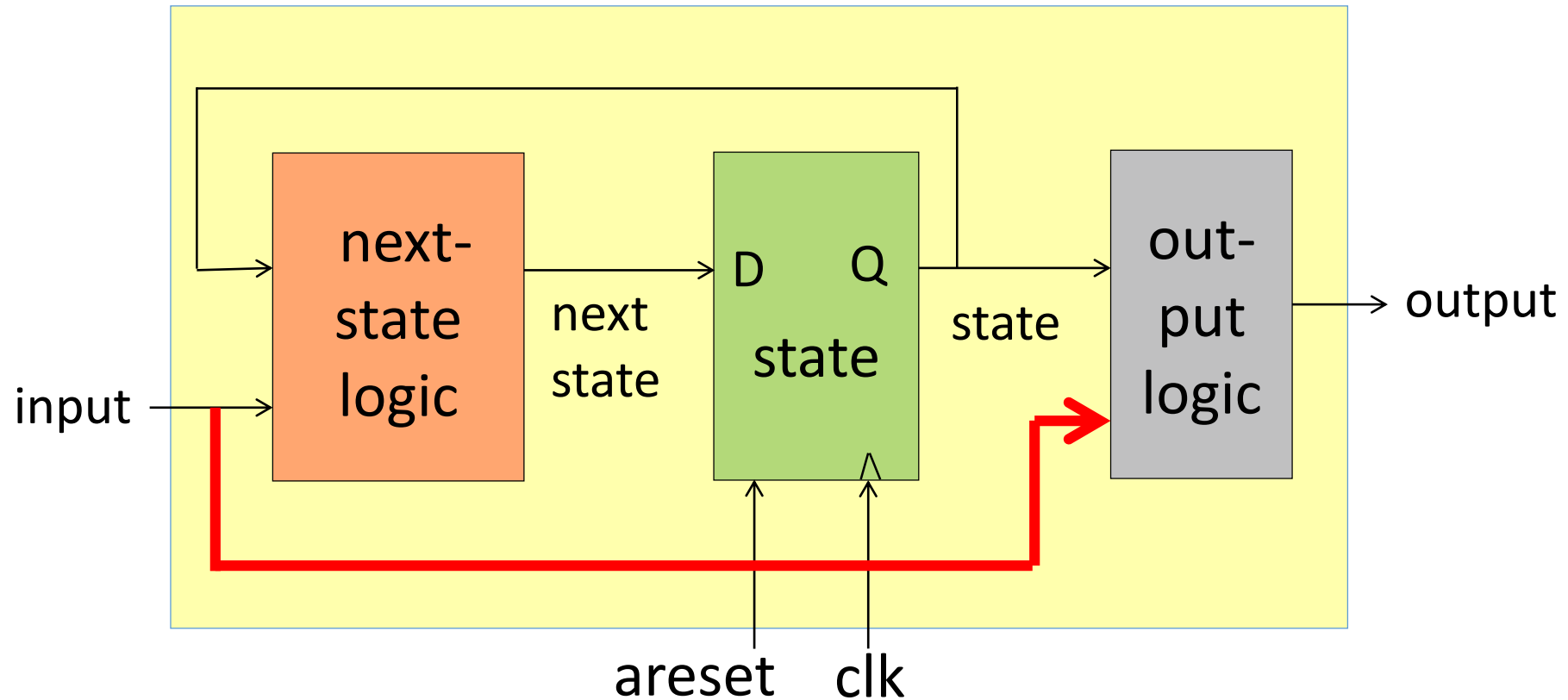# There exist 2 types of machines: check out the LITTLE but IMPORTANT difference

- 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.

A

in == 0
out = 2

out = 4

in == 0
out = 3

in == 1
out = 0

B

C

in == 1

out = 1
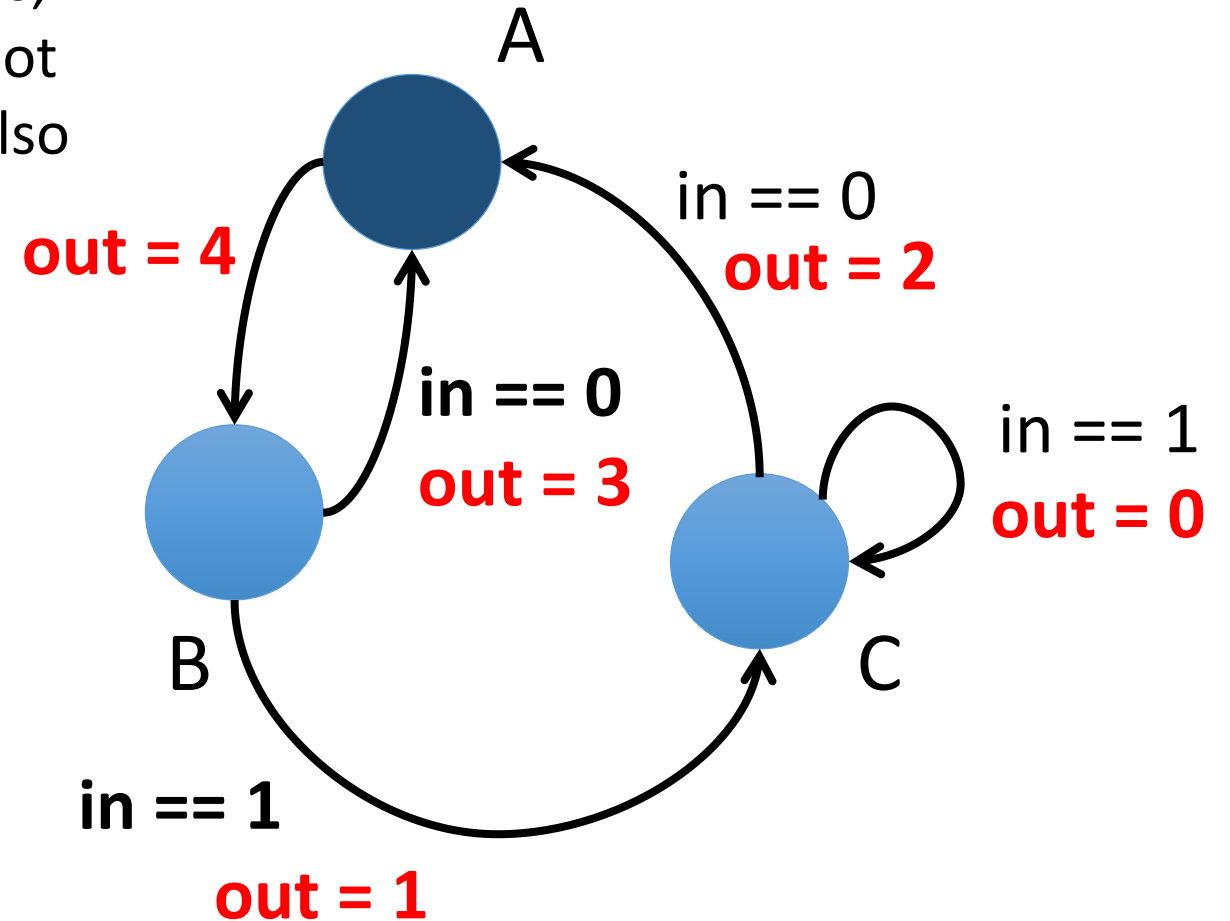
The state transitions are the same as in the previous example with the Moore Machine

present state	in	next state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	C
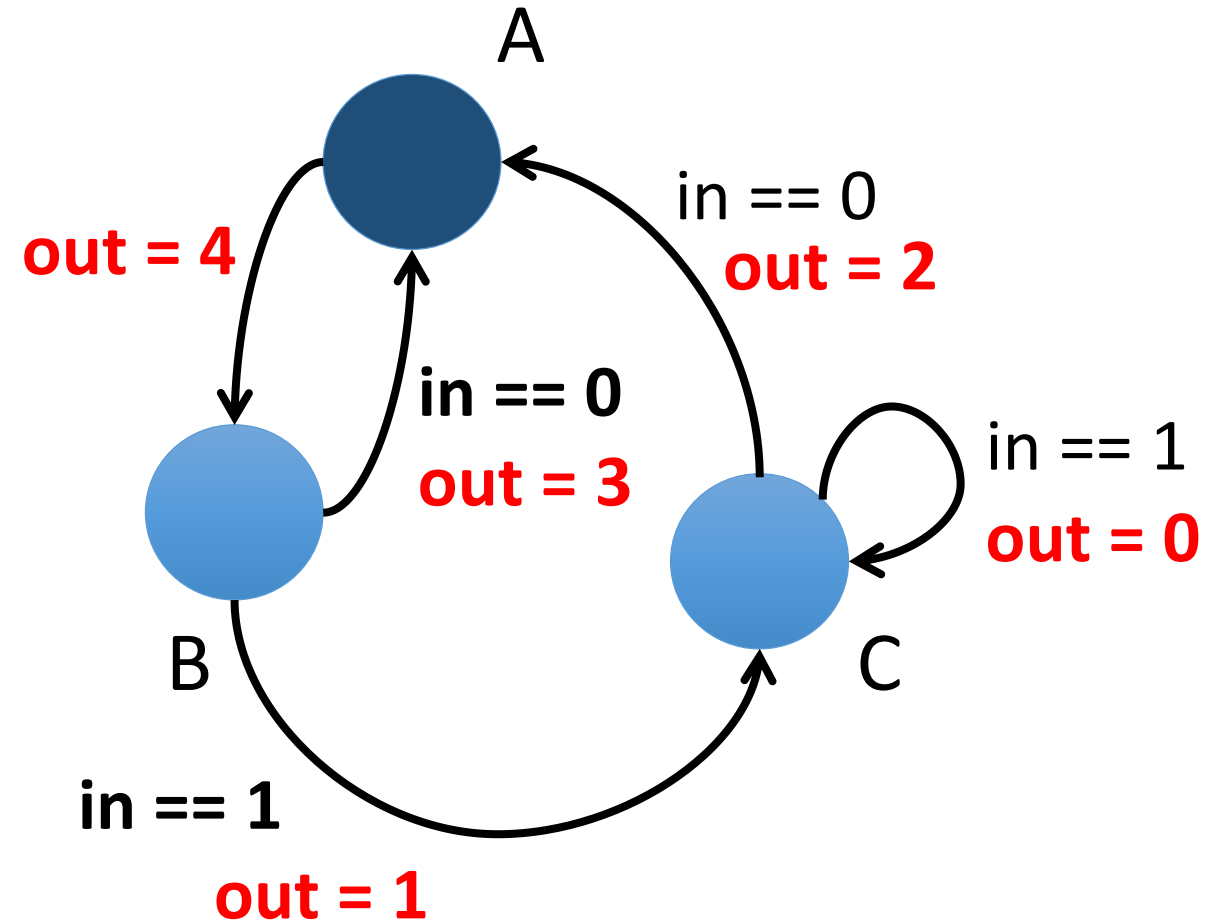
The output function

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
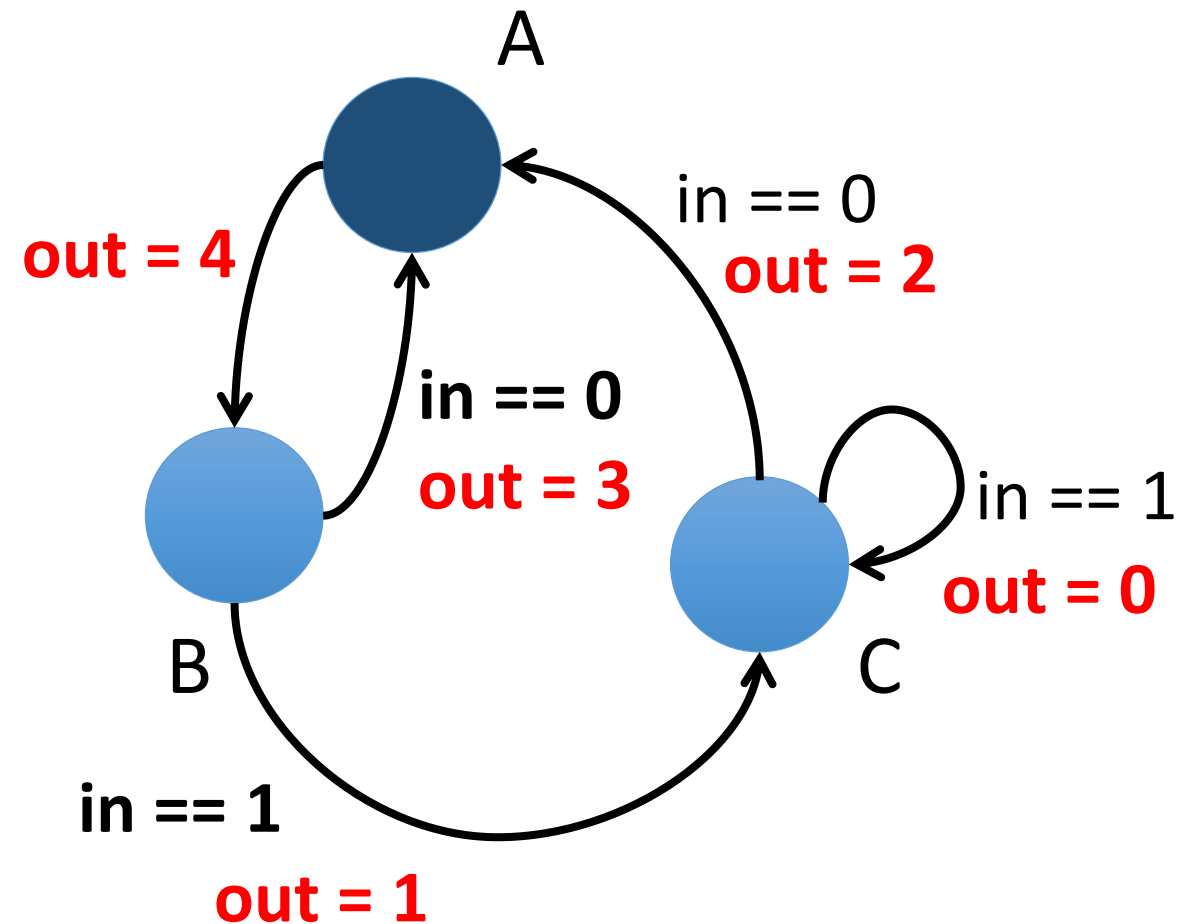
Timing diagram
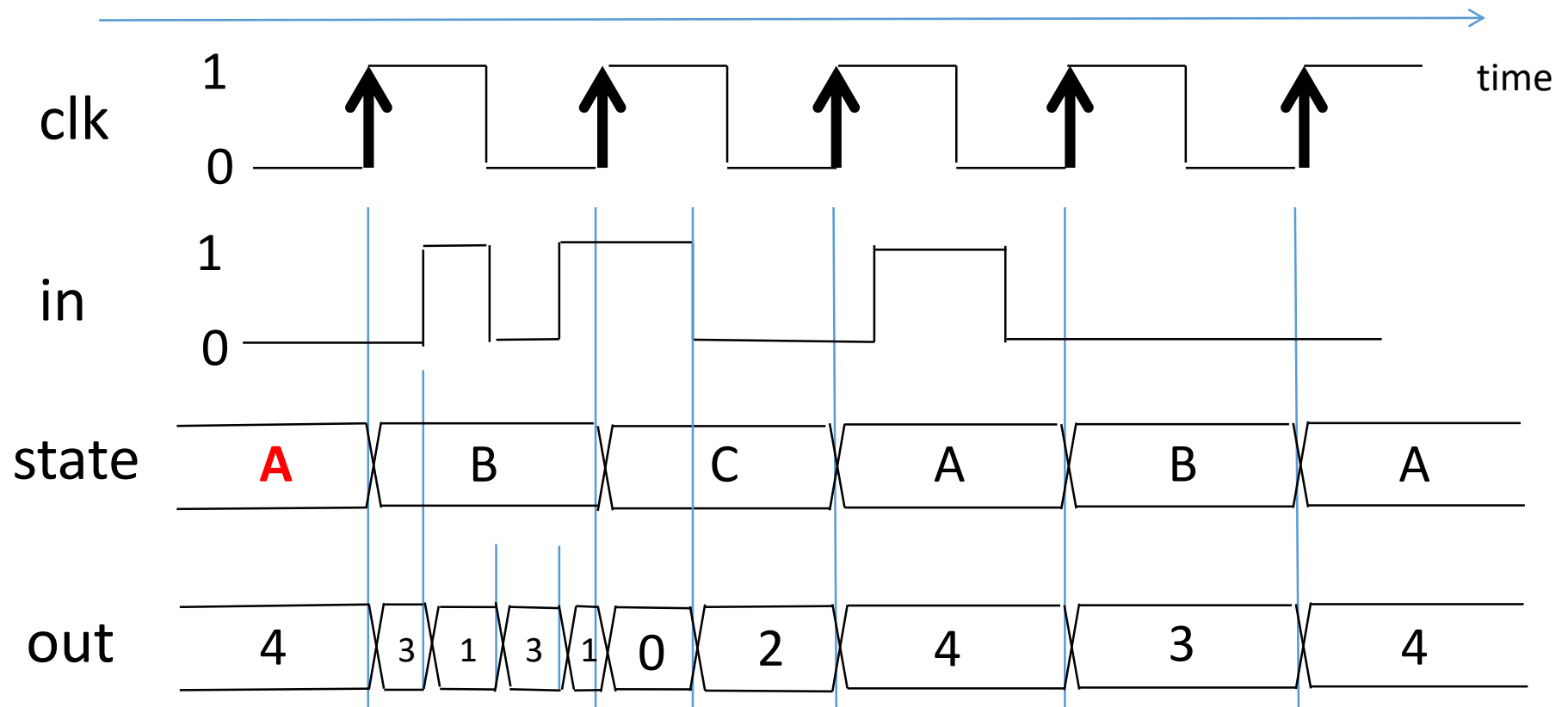
For a timing diagram, we need to choose values for the input signal "in". Only after some choice for "in" we can derive the sequence of states from the state diagram. We see here also, how the value of "in" immediately influences the value of "out".

91

Binary encoding: Re-writing the output table with binary values.

And completing the table with the unused bit combinations.

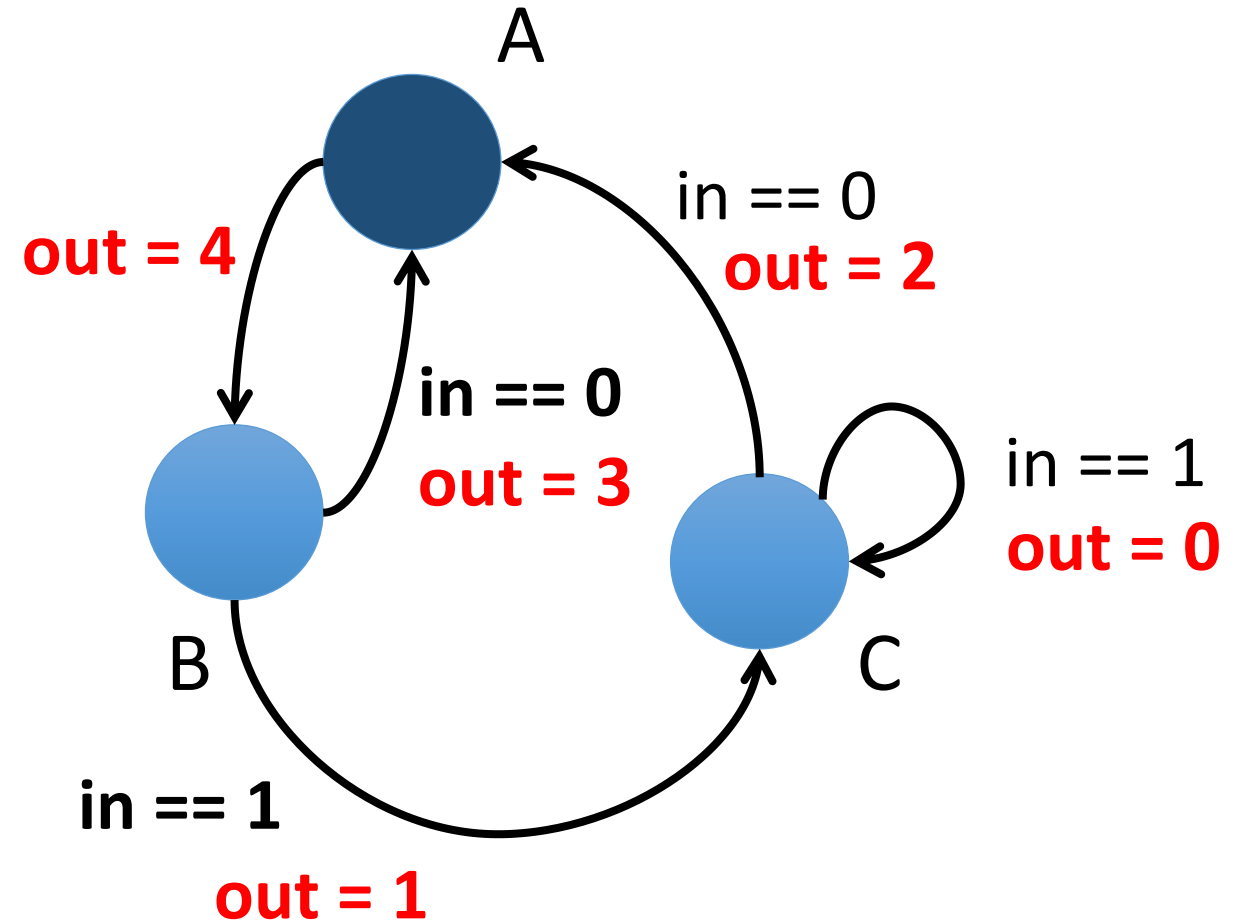| s1	s0	in	o2	o1	o0
0	0	0	1	0	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

A

out = 4

in == 0
out = 2

in == 0
out = 3

in == 1
out = 0

B

C

in == 1

out = 1

We can derive the logic functions for o2, o1, and o0

s1	s0	in	o2	o1	o0
0	0	0	**1**	0	0
0	0	1	**1**	0	0
0	1	0	0	1	1
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

o2 = ~s1 & ~s0

We can derive the logic functions for o2, o1, and o0

s1	s0	in	o2	o1	o0
0	0	0	**1**	0	0
0	0	1	**1**	0	0
0	1	0	0	**1**	1
0	1	1	0	0	1
1	0	0	0	**1**	0
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

o2 = ~s1 & ~s0

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

We can derive the logic functions for o2, o1, and o0

s1	s0	in	o2	o1	o0
0	0	0	**1**	0	0
0	0	1	**1**	0	0
0	1	0	0	**1**	**1**
0	1	1	0	0	**1**
1	0	0	0	**1**	0
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

The result

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

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

D0   Q0

D1   Q1

s0

s1

o2 = ~s1 & ~s0

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

o0 = ~s1 & s0

in

areset   clk

o2

o1

o0
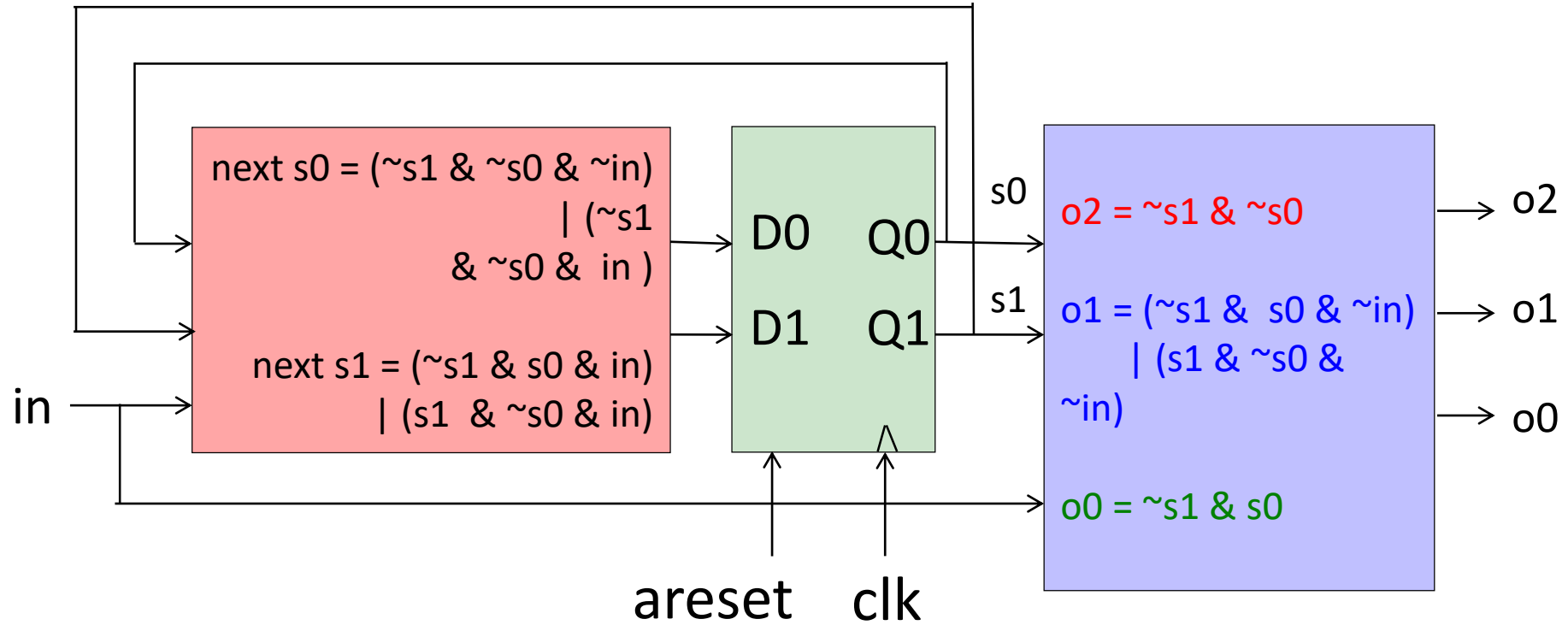
Modeling with Digital
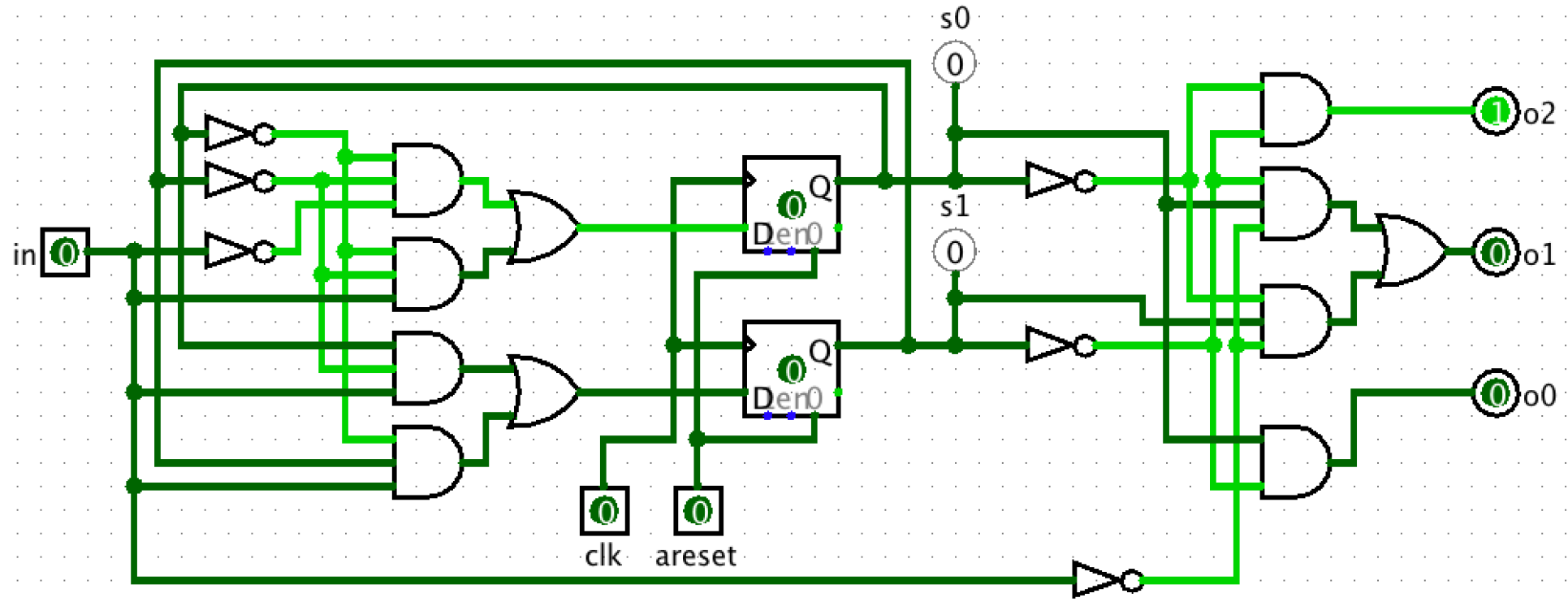
For a SystemVerilog example of this FSM see

con03_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.

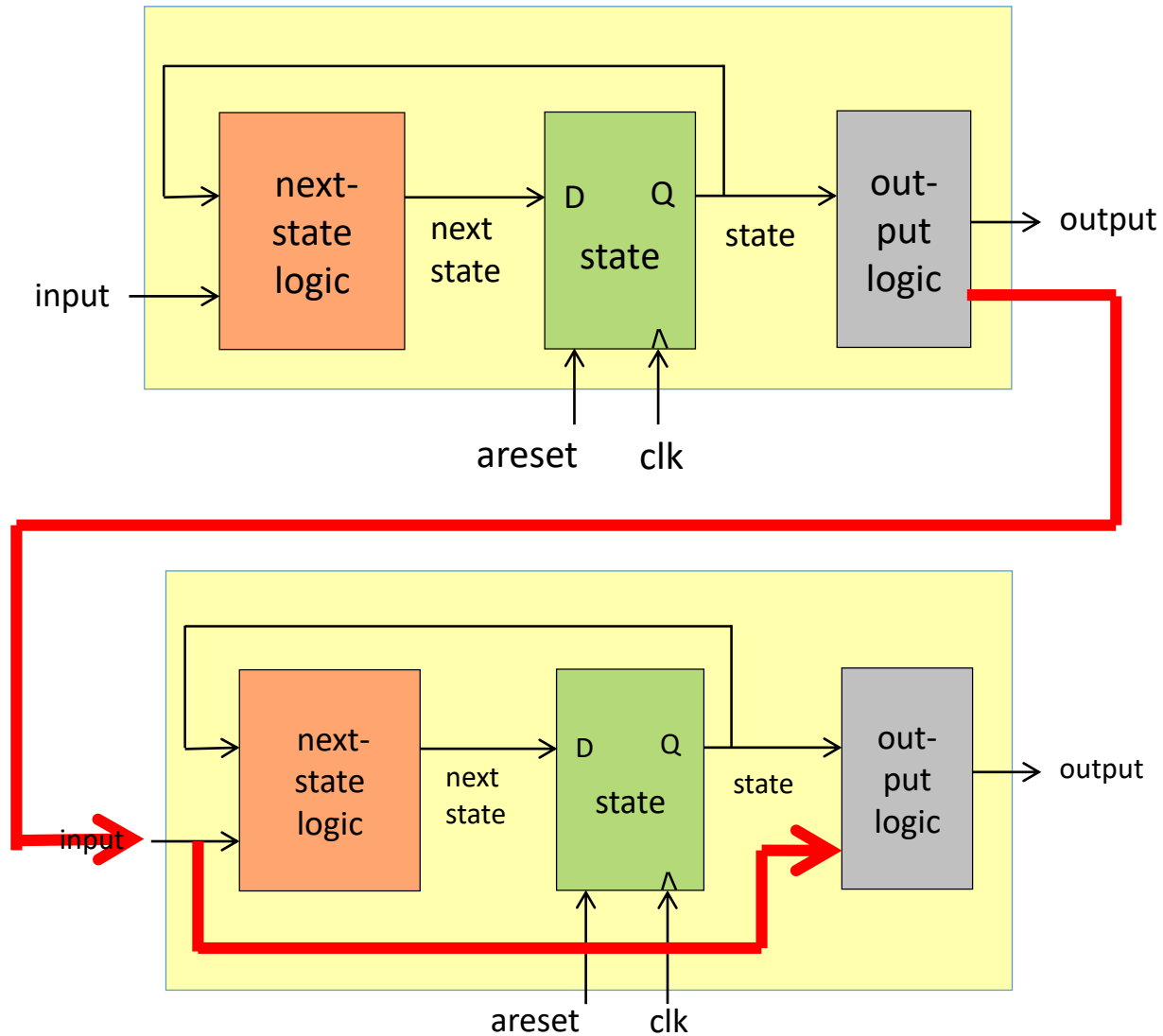- Combining two Mealy Machines can cause troubles: **One needs to avoid combinational loops**!

The combination of two Moore Machines creates again a (more complex) Moore Machine
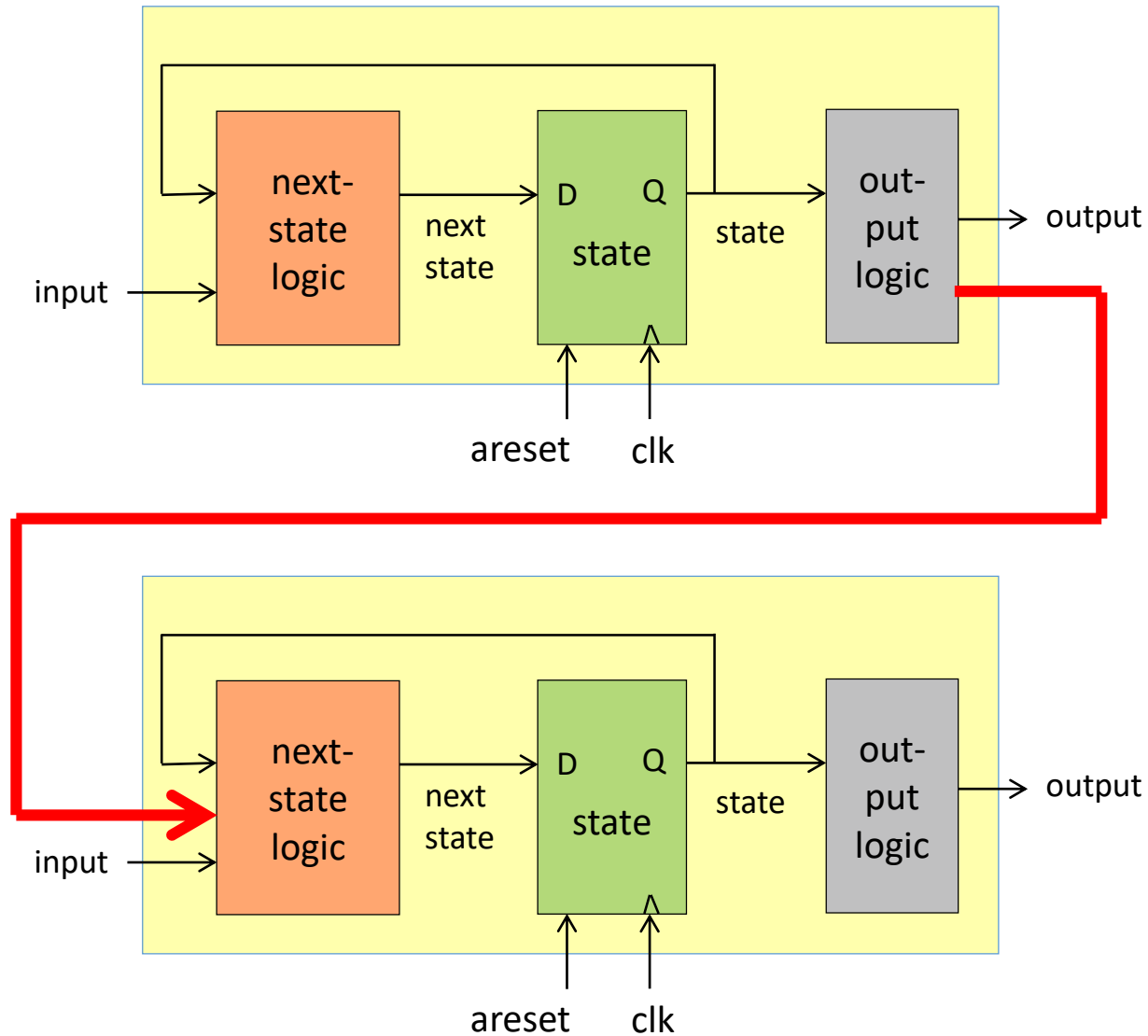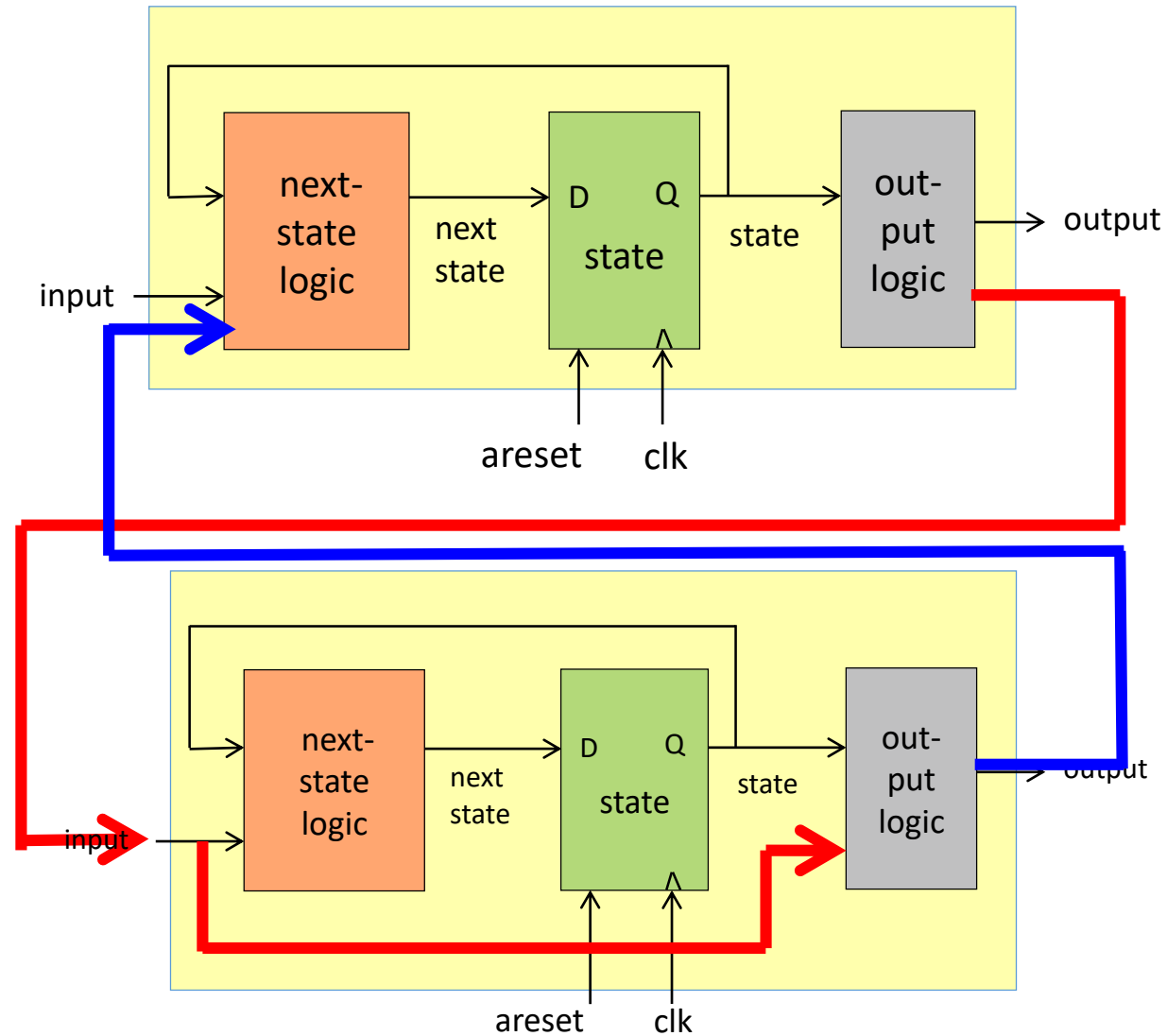
We can even connect More Machines in a loop-like fashion
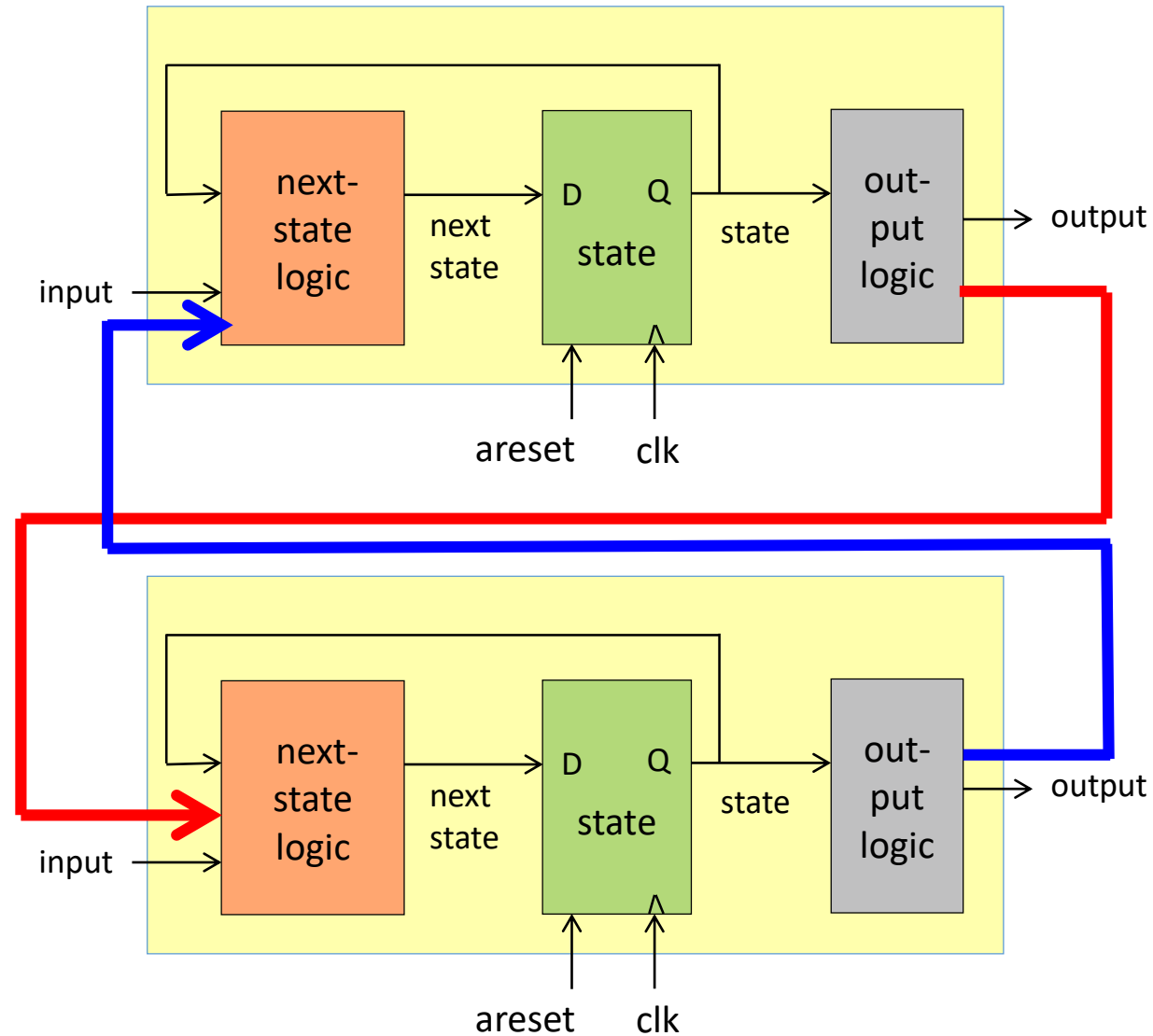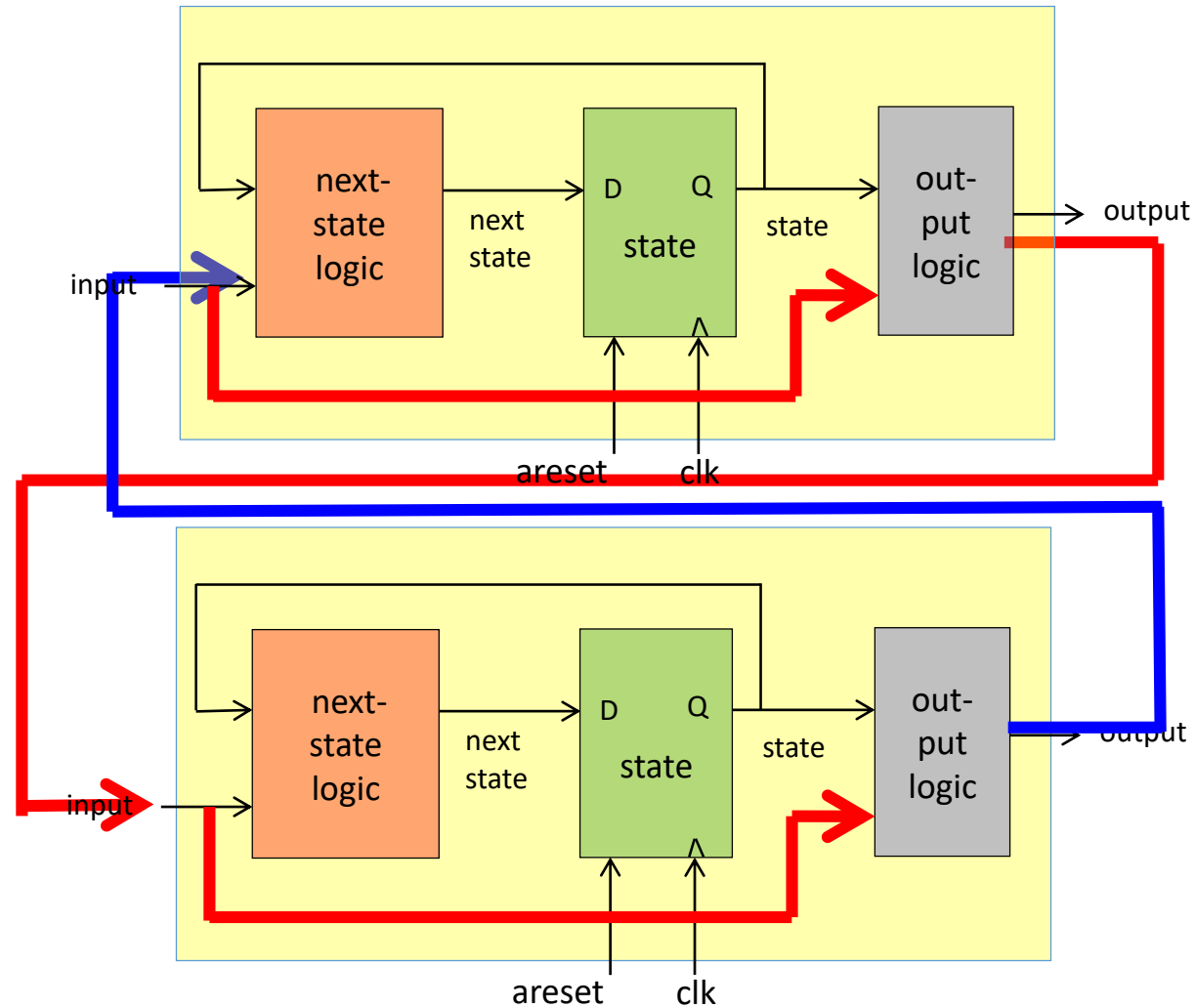
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine
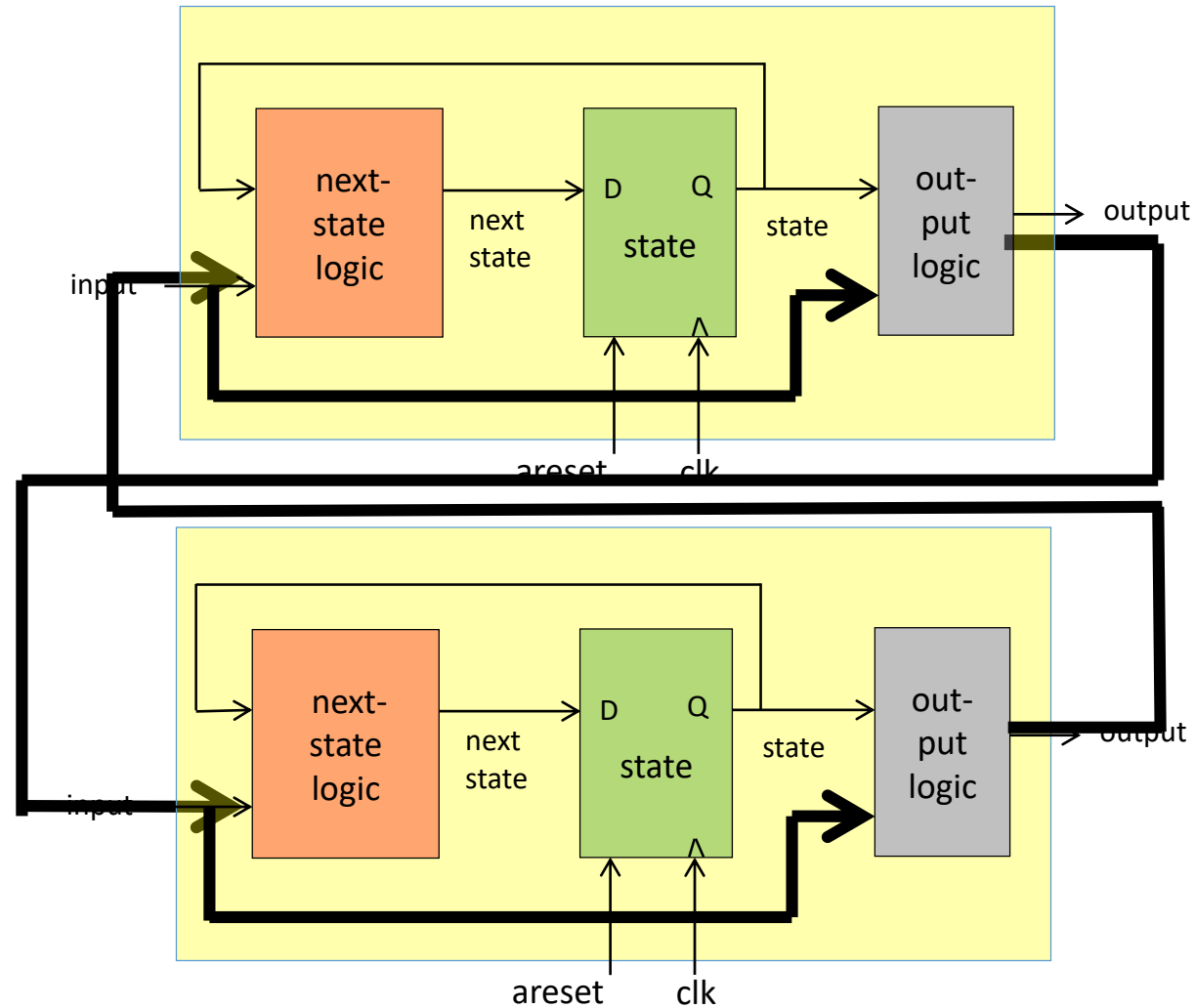
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine

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

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

- You always start by defining the function with a state diagram.

- If you choose values for the input signal(s), then you can derive the timing diagram by using the state diagram.

- From a state diagram, you can always derive a circuit diagram.
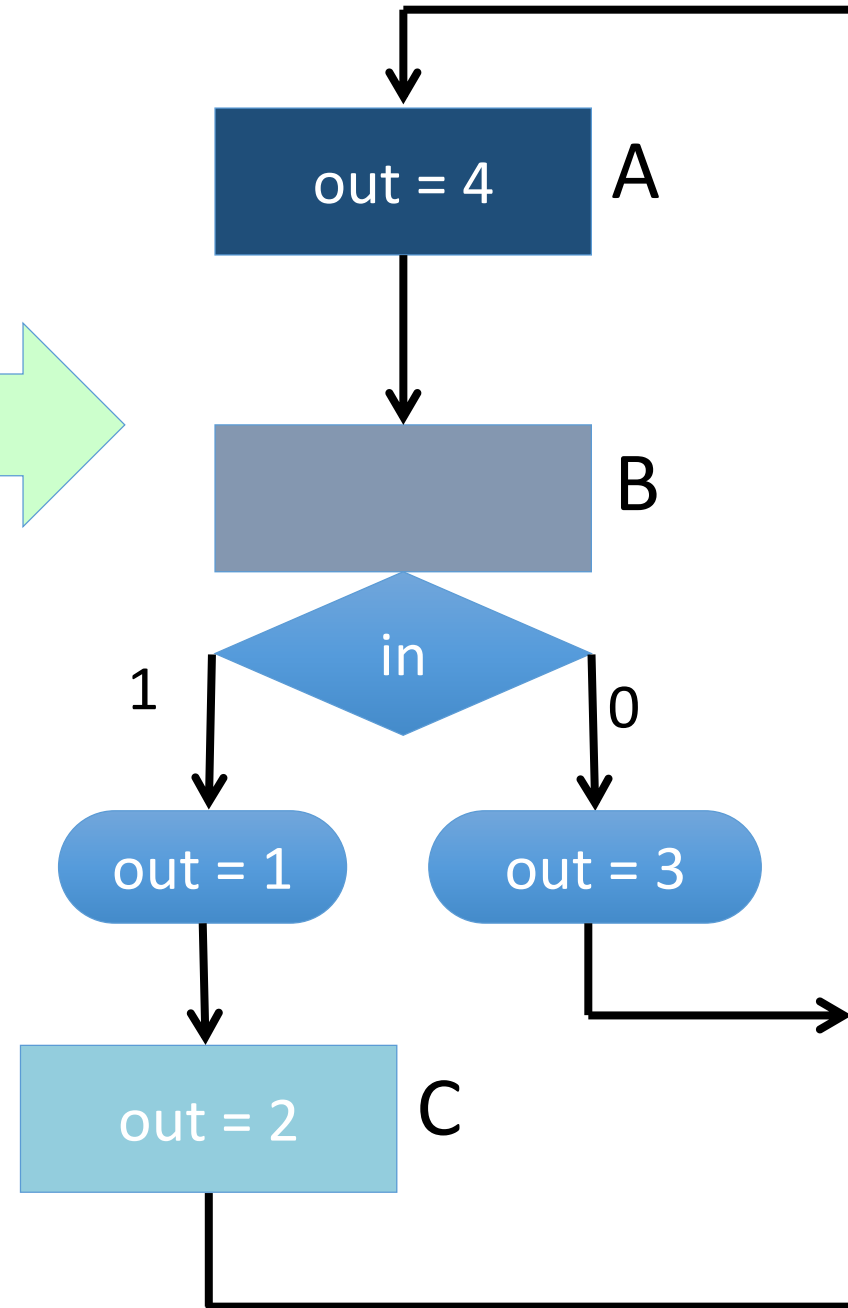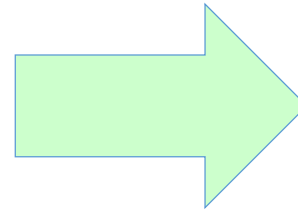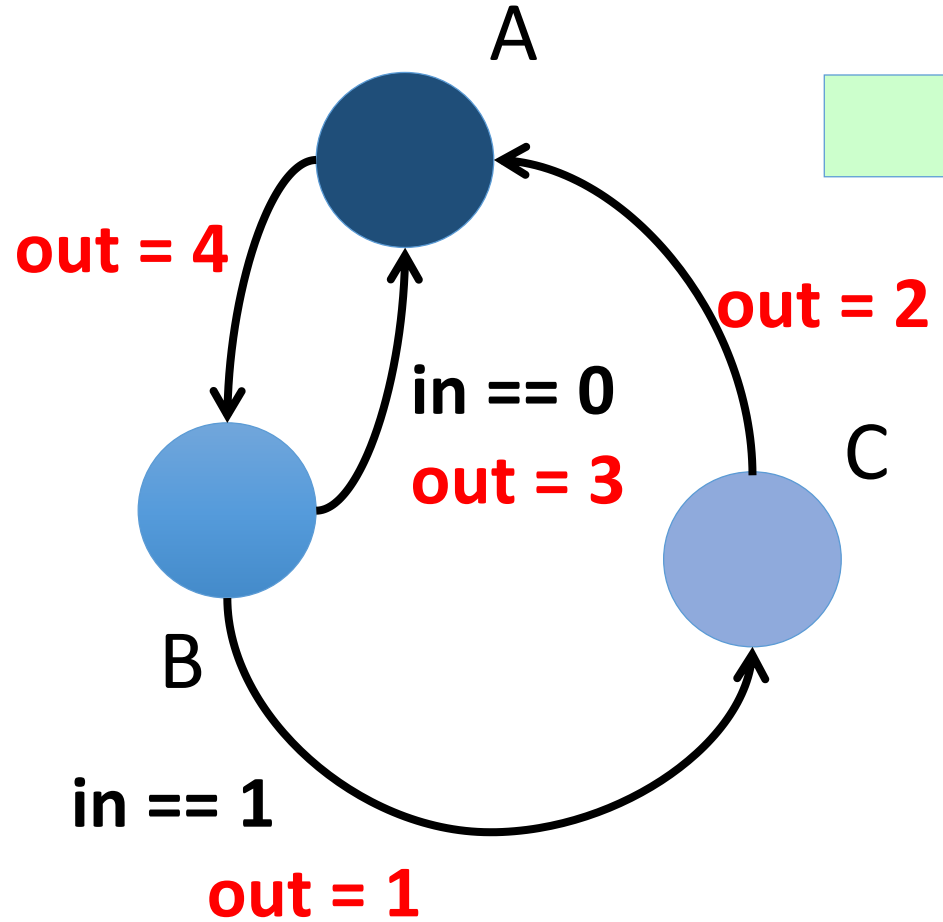
Algorithmic State Machines

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
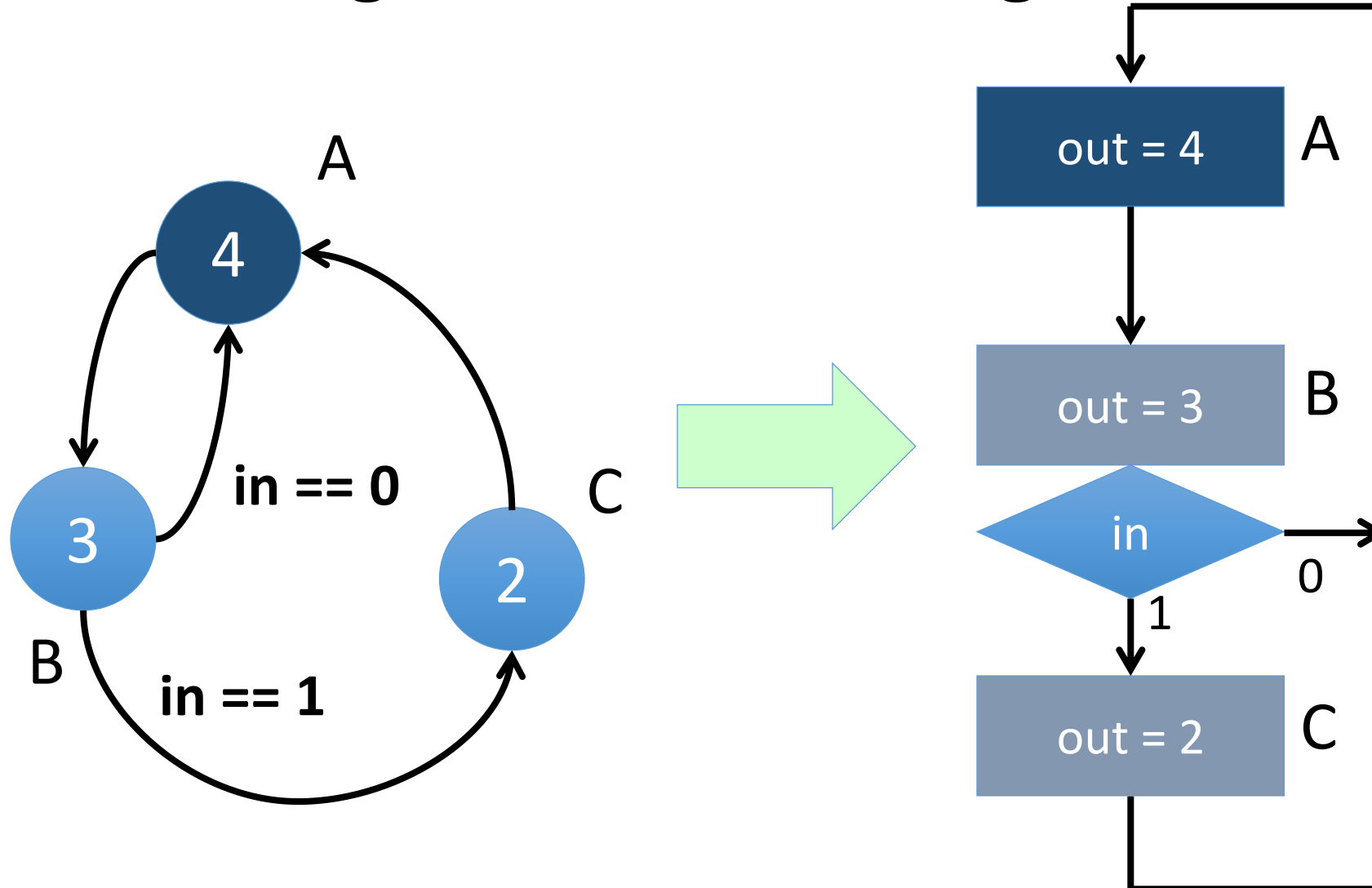
Mealy Machines

Mealy Machines
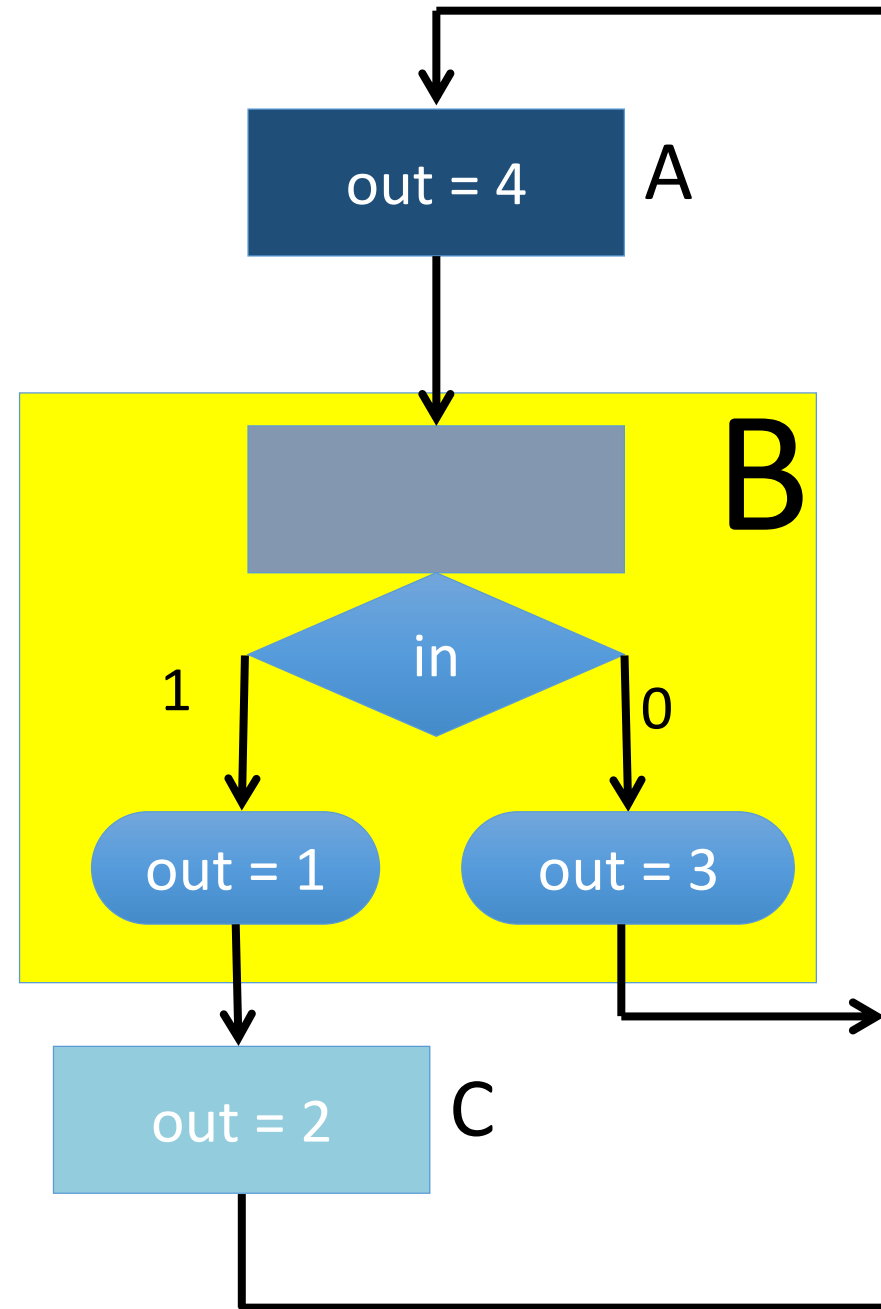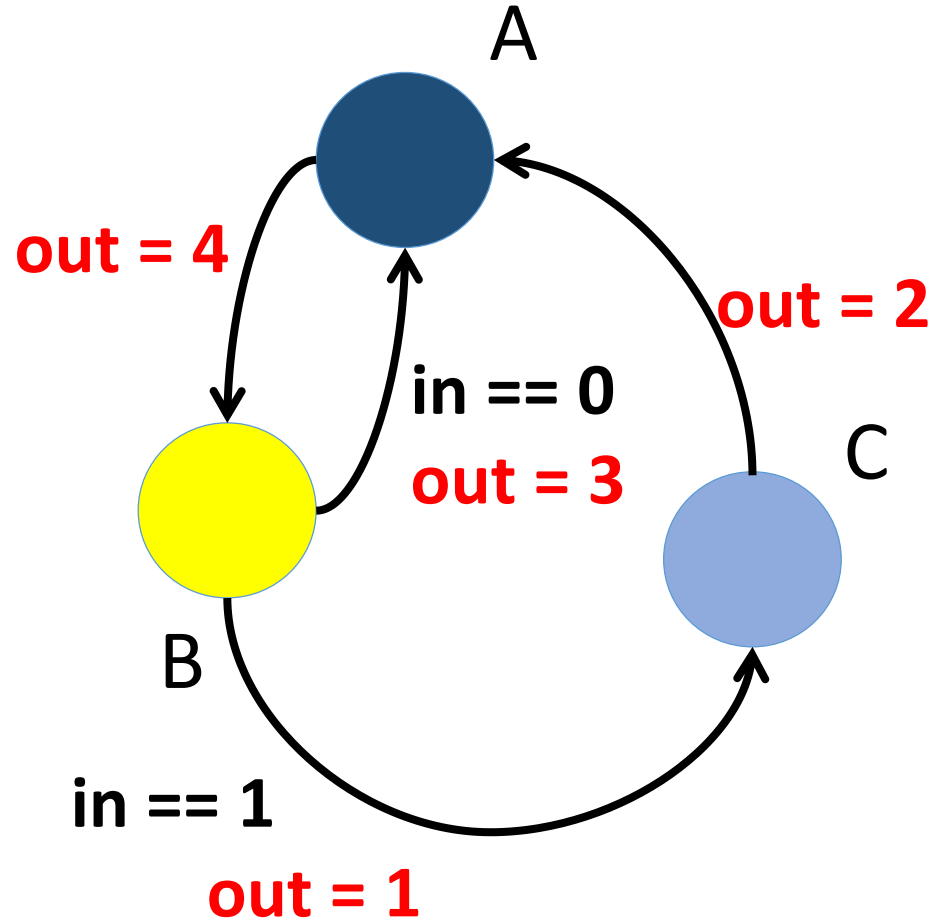
Register-Transfer Statements

- Register-transfer statements define the change of a value stored in a register.

- Values in registers can only change at the active (= rising) edge of clock.

- We denote "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.

- We can specify register-transfer statement in an ASM diagram.

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.
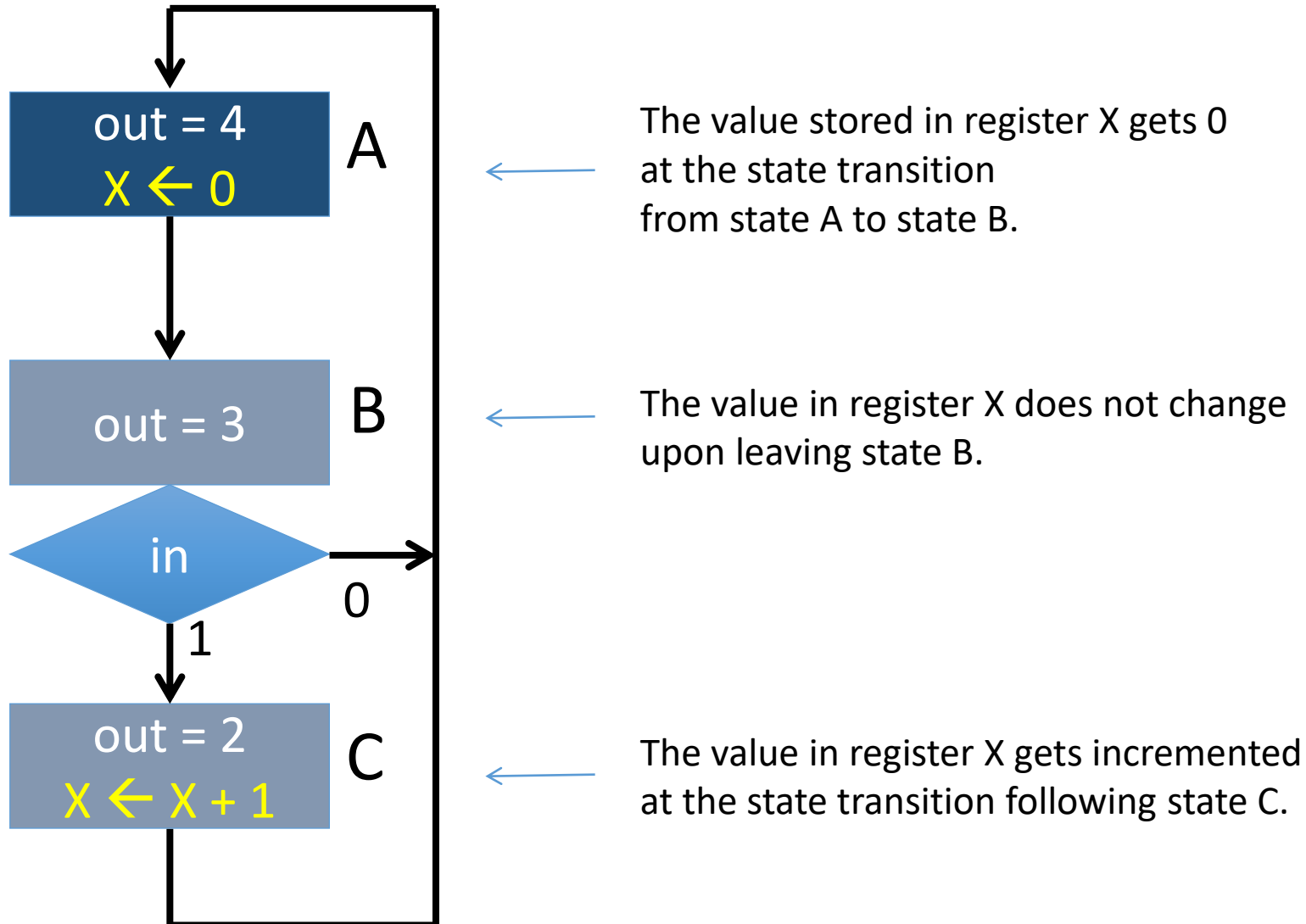
Timing diagram
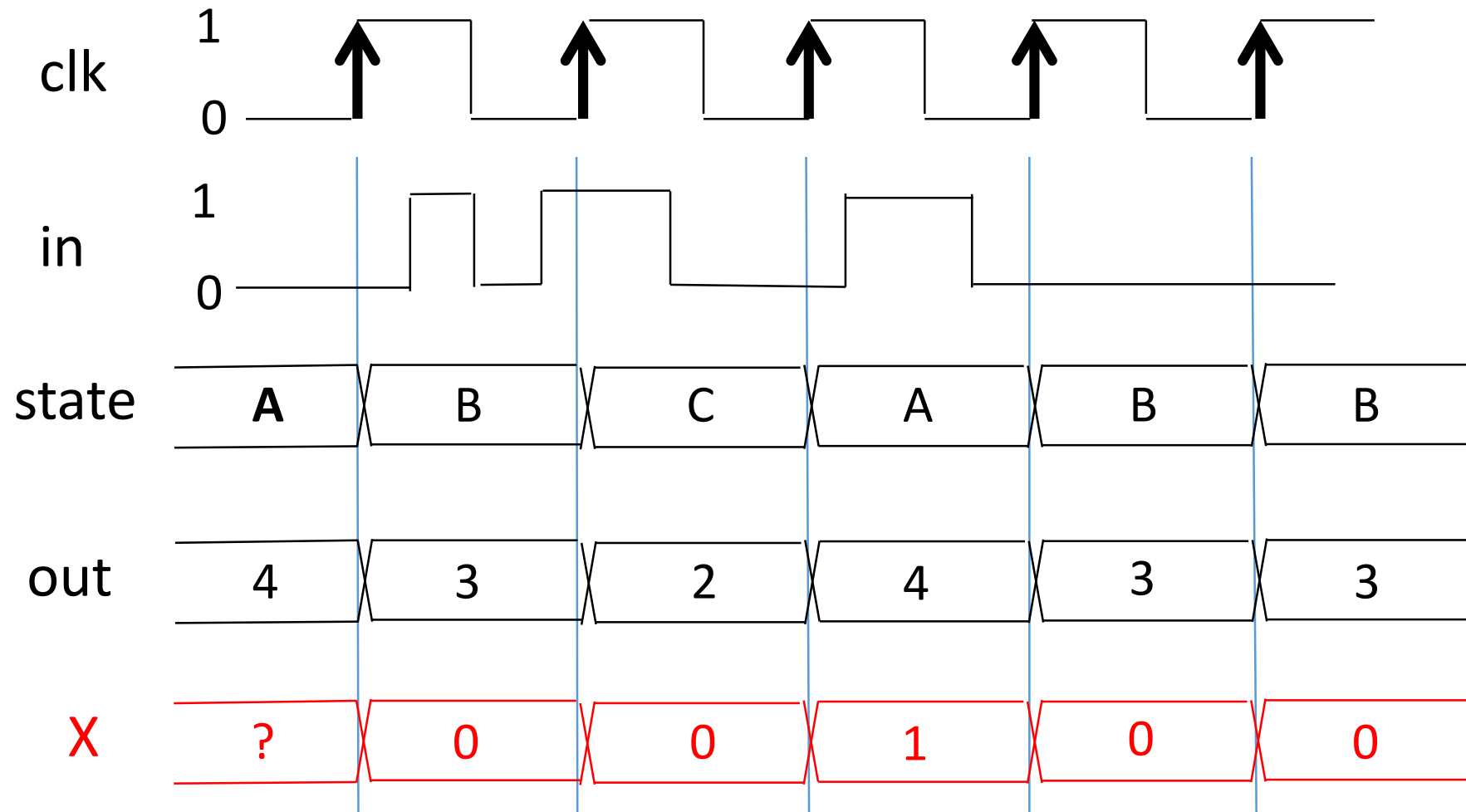
"=" versus "←"

- With the equal sign ("=") we denote that the output of the FSM has a certain value during a particular state.

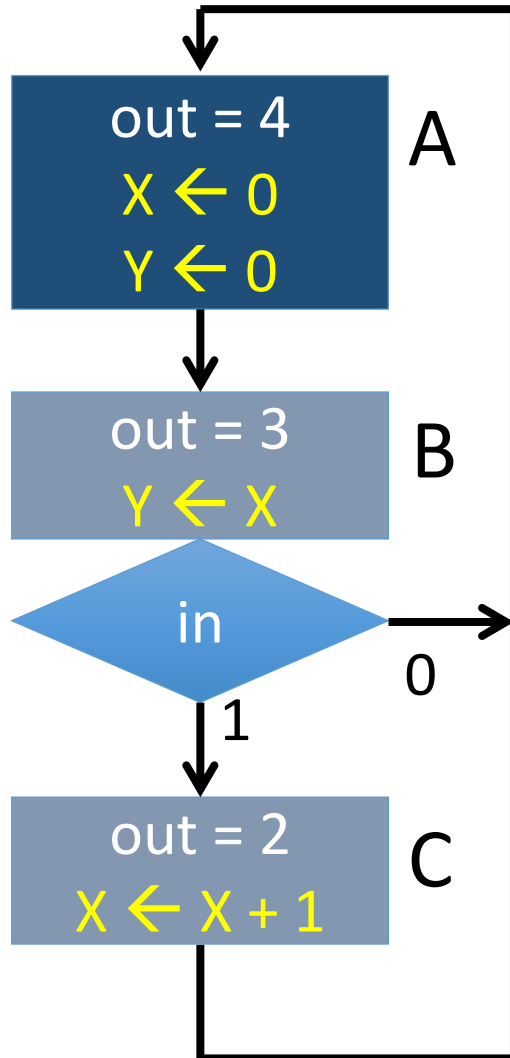- With the left-arrow ("←") we denote a register-transfer statement: The register value left of the arrow changes to whatever is defined right of the arrow upon the next active (= rising) edge of clock.

Several register-transfer statements can be specified within one state

A

out = 4

X ← 0

Y ← 0

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

B

out = 3

Y ← X

in

0

1

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.

C

out = 2

X ← X + 1

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

Several register-transfer statements can be specified within one state

A
out = 4
$X \leftarrow 0$
$Y \leftarrow 0$

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

B
out = 3
$Y \leftarrow X$

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.

in 0 1

C
out = 2
$X \leftarrow X + 1$
$Y \leftarrow X$

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.

Separating Control and Data Path

Register-Transfer Statements Define the Data Path

out = 4
A
$X \leftarrow 0$
$Y \leftarrow 0$

out = 3
B
$Y \leftarrow X$

in

0

1

out = 2
C
$X \leftarrow X + 1$
$Y \leftarrow X$

Operations for register X:

$$X \leftarrow 0$$
$$X \leftarrow X$$
$$X \leftarrow X + 1$$

Operations for register Y:

$$Y \leftarrow 0$$
$$Y \leftarrow X$$
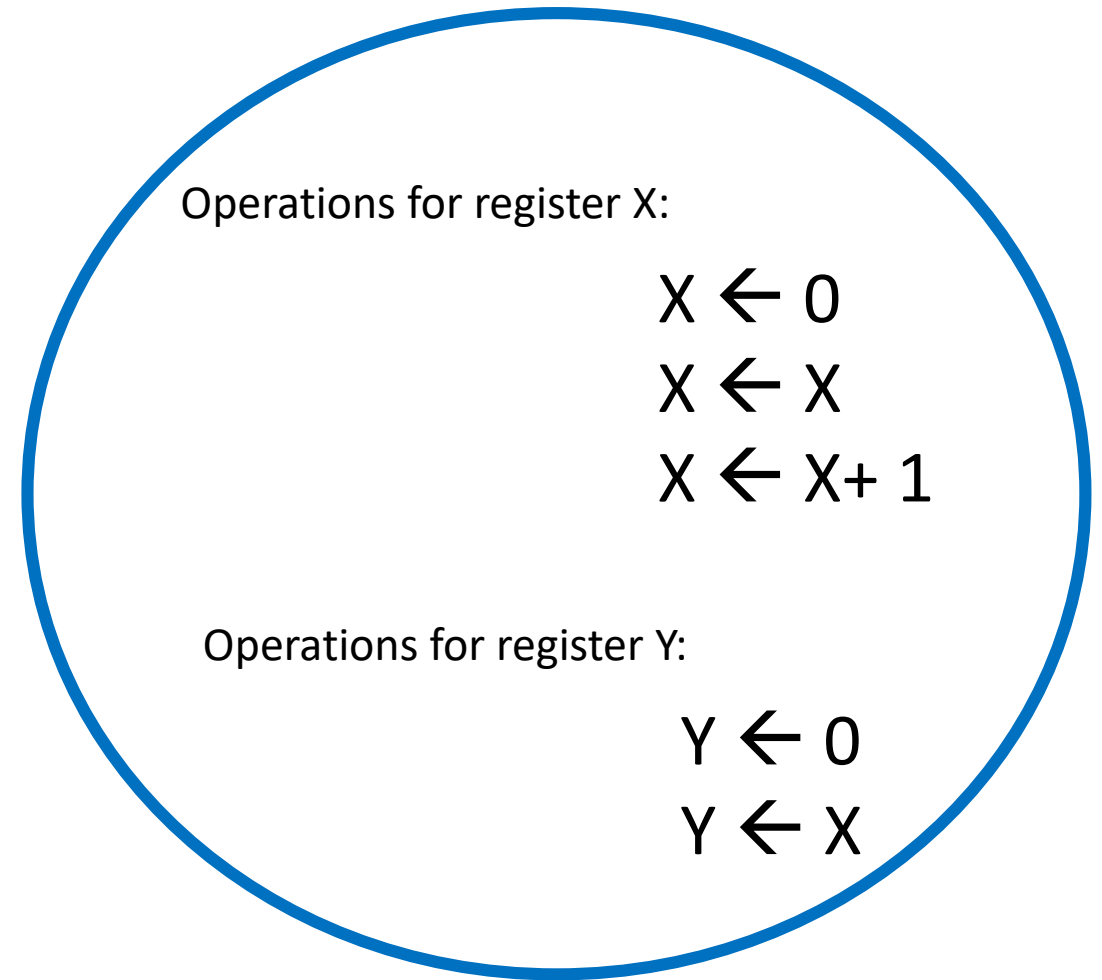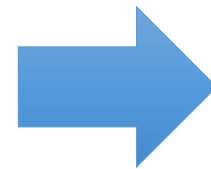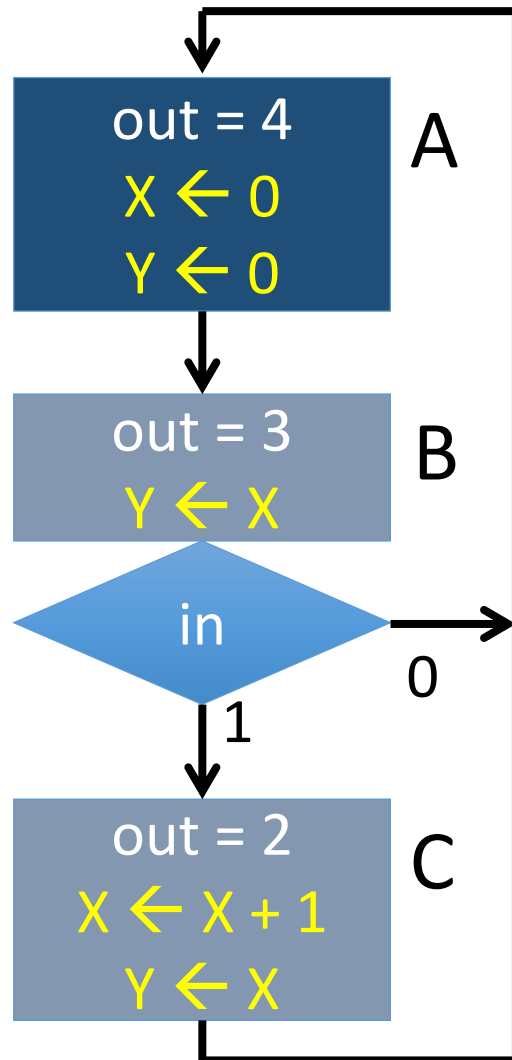
These are the actions that our system is able to perform on
The data registers X and Y

Control Unit

Data Path

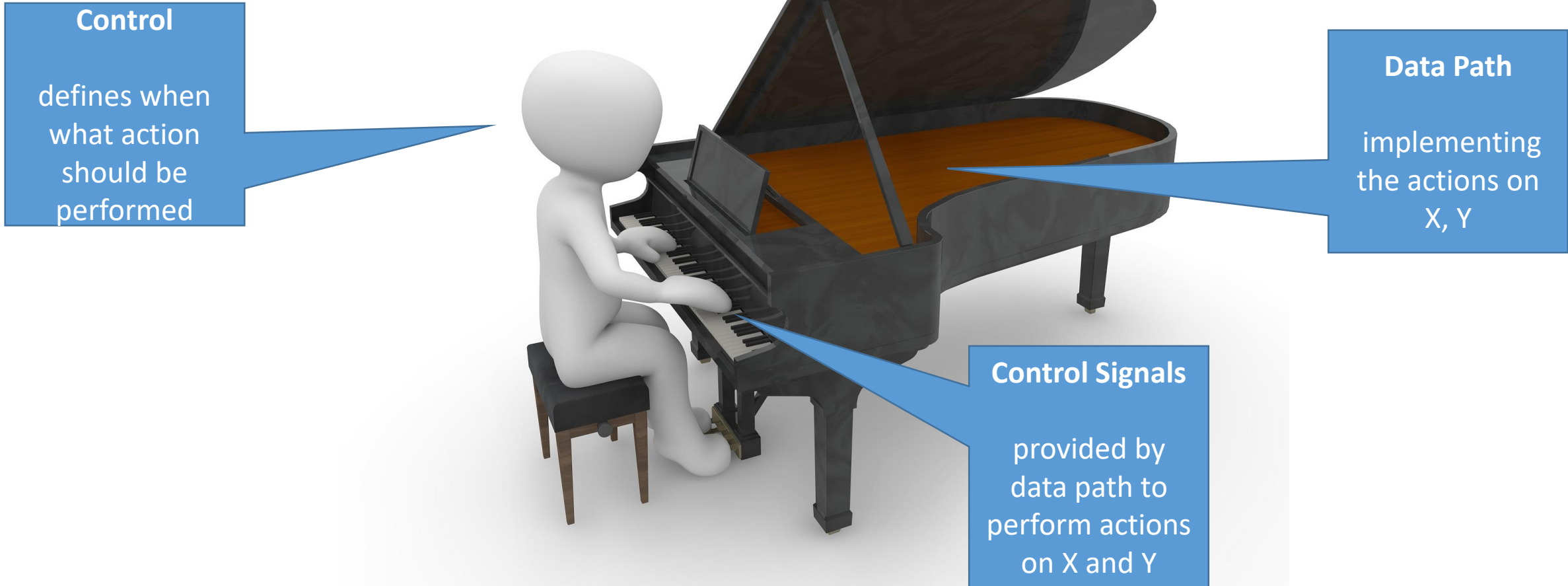- State machine generating control signals for the data path

- 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
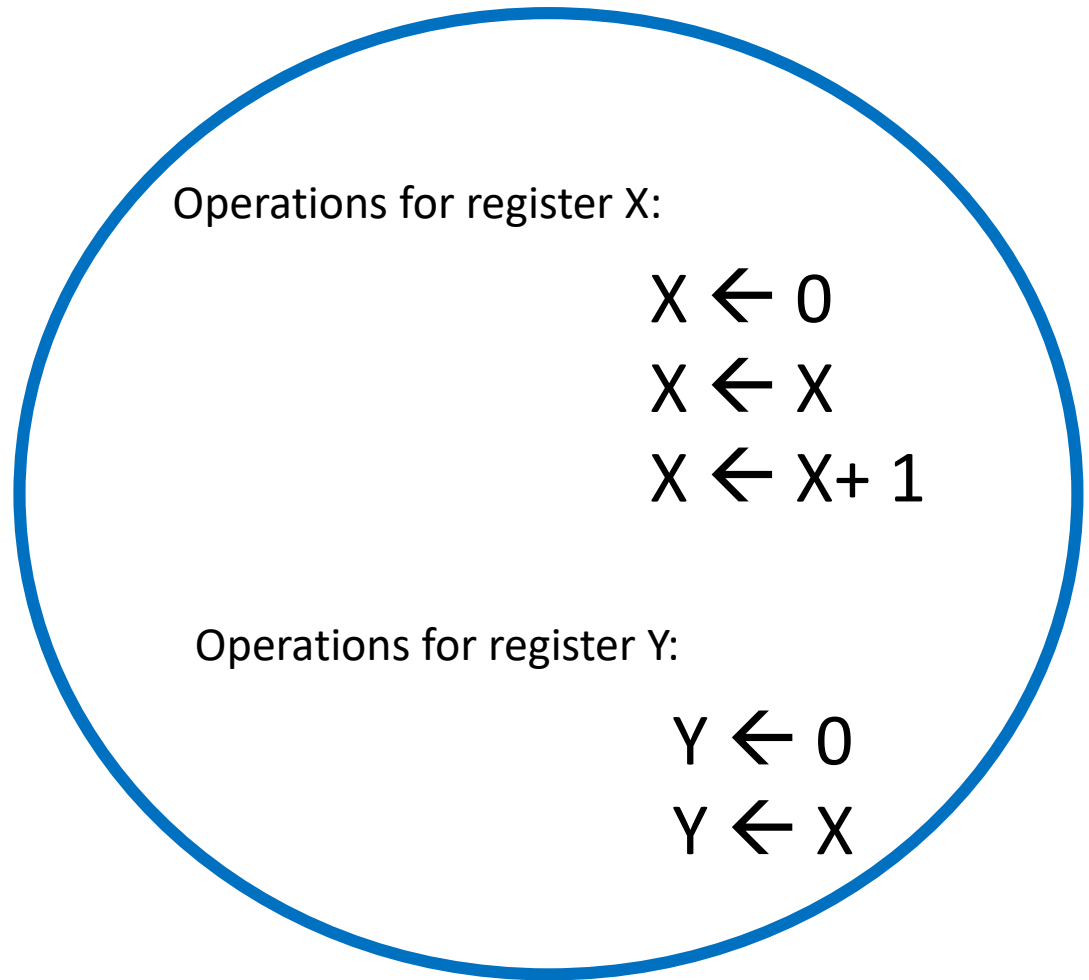

"Piano Player"

"Piano"

Music

Control

defines when what action should be performed

Data Path

implementing the actions on X, Y

Control Signals

provided by data path to perform actions on X and Y

Register-Transfer Statements Define the Data Path


```
out = 4    A
X ← 0
Y ← 0

out = 3    B
Y ← X

   in      0

   1

out = 2    C
X ← X + 1
Y ← X
```

Operations for register X:

$$X \leftarrow 0$$
$$X \leftarrow X$$
$$X \leftarrow X+1$$

Operations for register Y:

$$Y \leftarrow 0$$
$$Y \leftarrow X$$
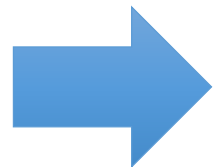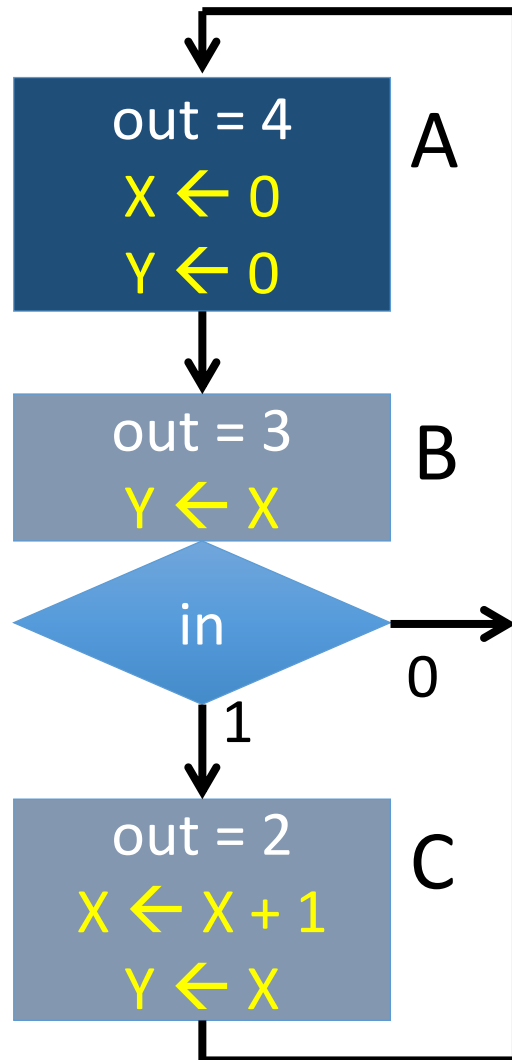
These are the operations that our data path implements

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.

The neighborhood of register X
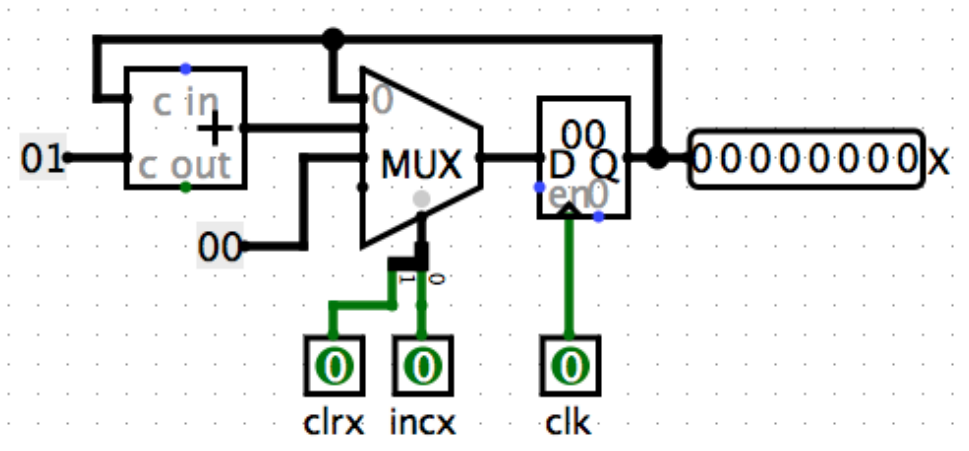
clrx	incx	action
0	0	$X \leftarrow X$
0	1	$X \leftarrow X + 1$
1	0	$X \leftarrow 0$

We use binary notation and name the two binary select variables.

The neighborhood of register X

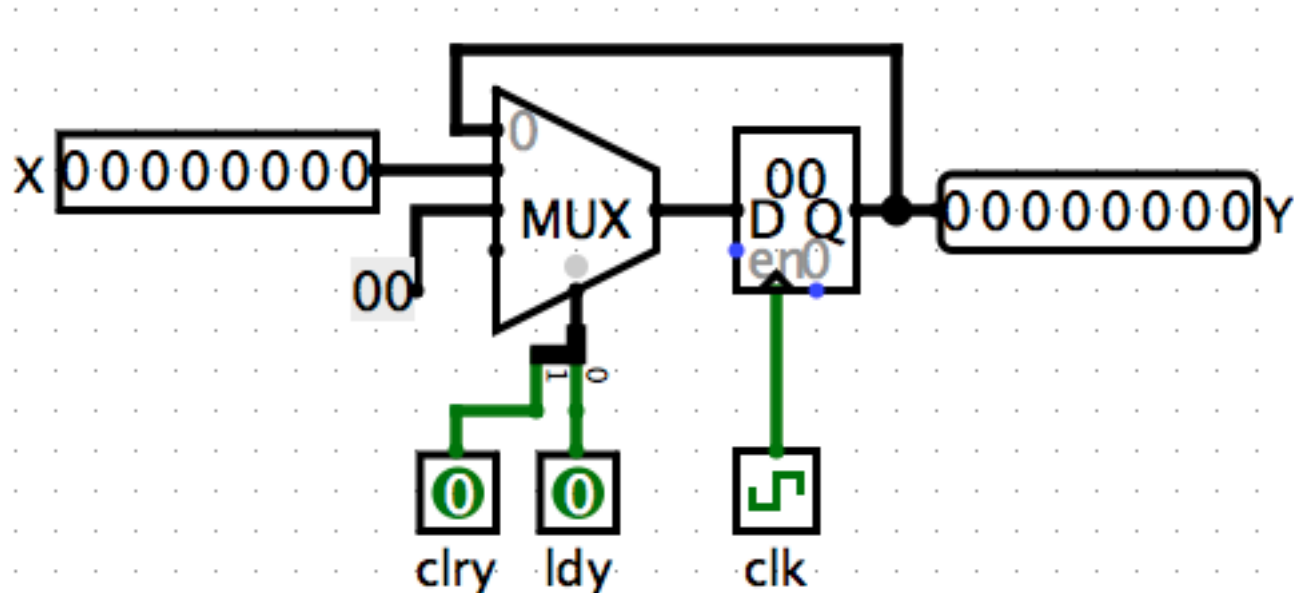| clrx	incx	action
0	0	X ← X
0	1	X ← X+ 1
1	0	X ← 0

With Logisim we can
model the neighborhood of
Register and also simulate.

The neighborhood of register Y

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

In a similar way, we can model the neighborhood of Y.

The datapath
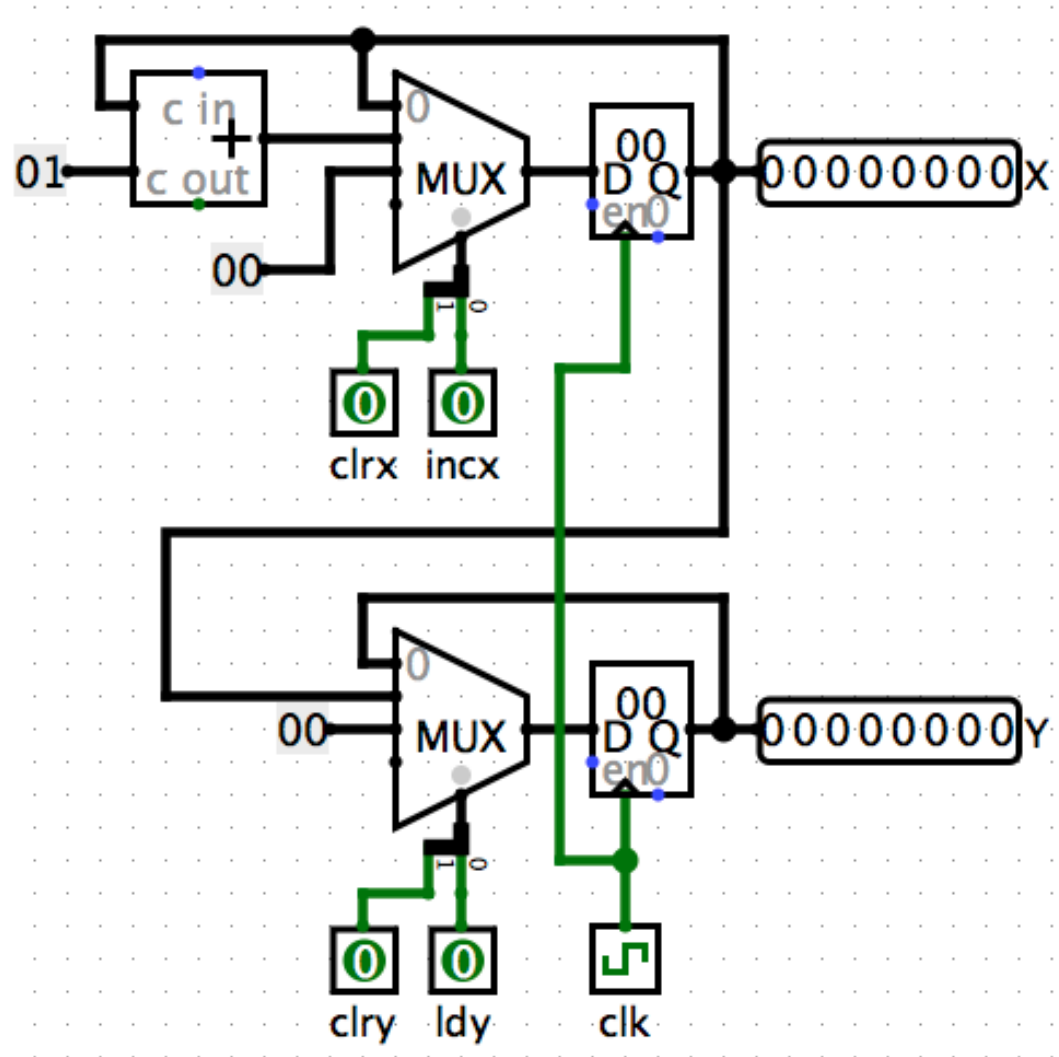
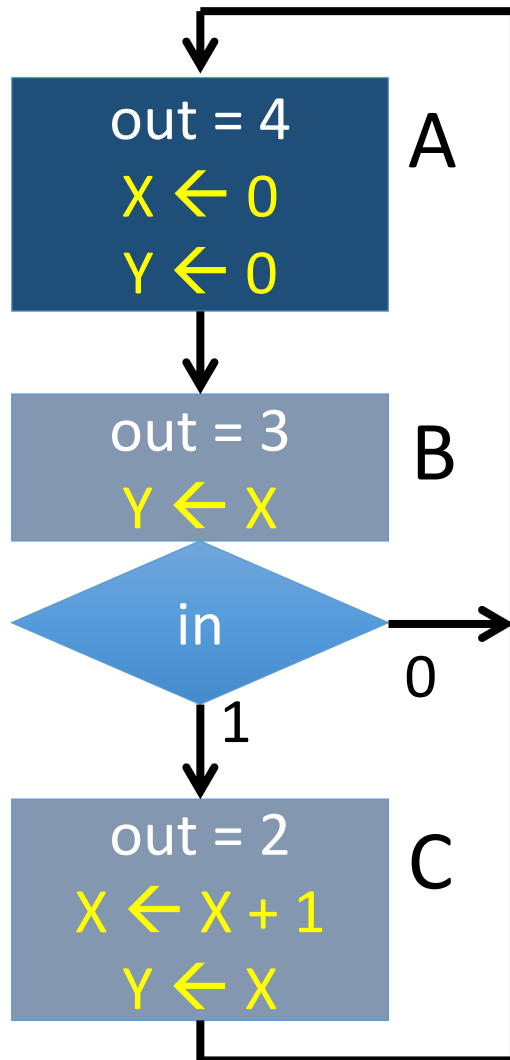We combine the two neighborhoods.

Note that both neighborhoods are Moore machines.
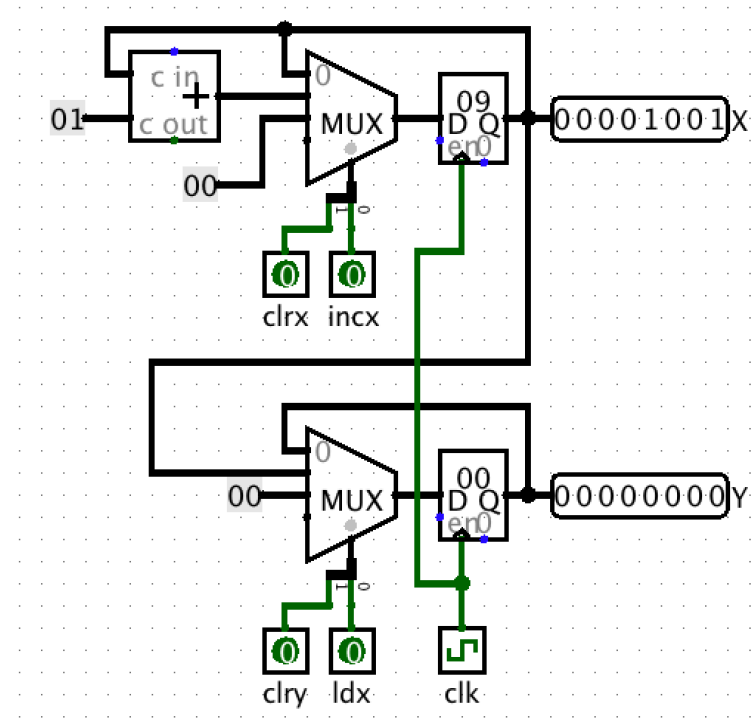
The Moore machine for X has 2 inputs: clrx and incx. Since we have chosen an 8-bit register for X, we have 256 possible states. The output function is the identity function.

The connection of the two is again a Moore machine. Thus, The datapath is a Moore machine.

127
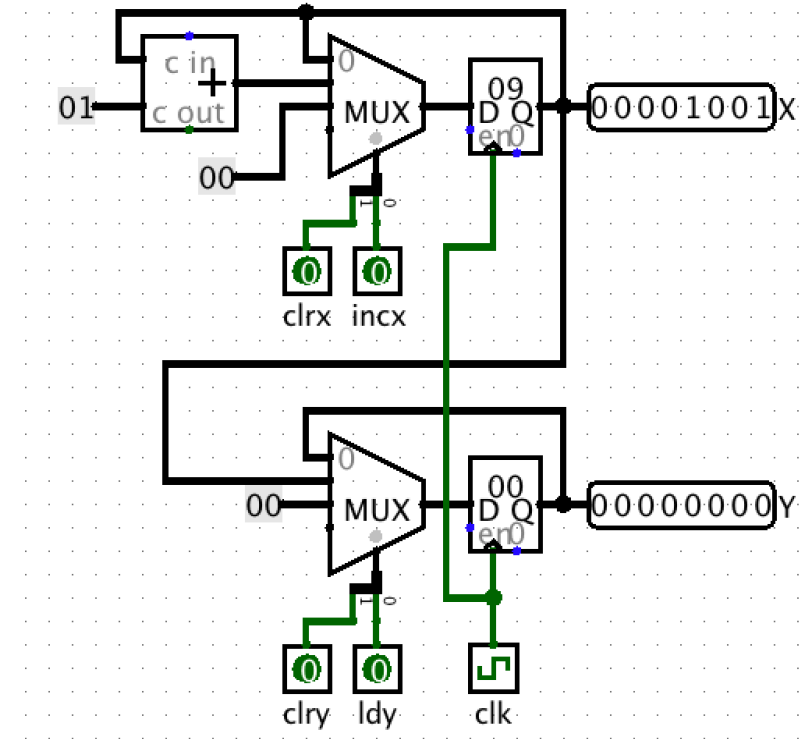
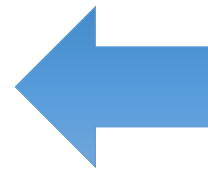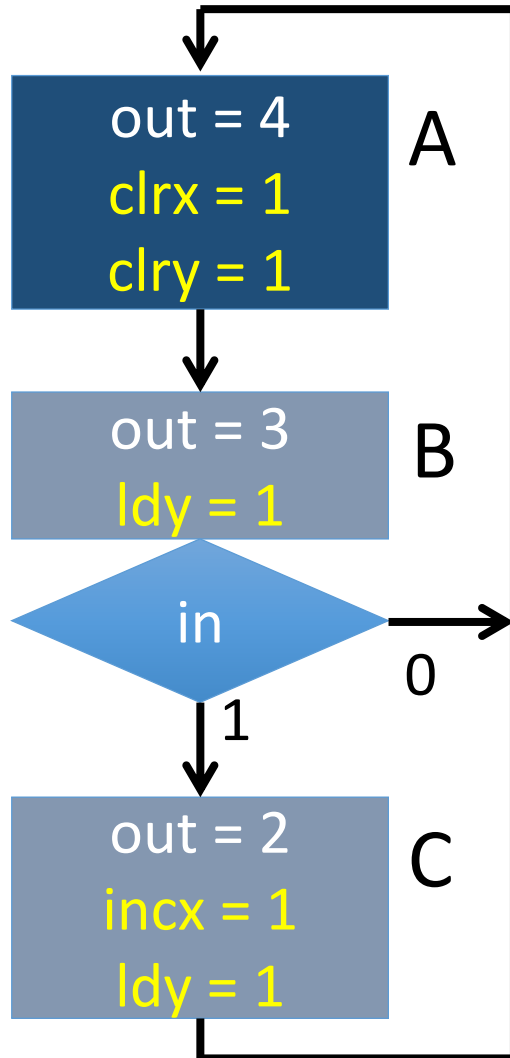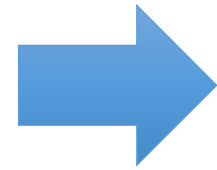# Register-transfer statements define the data path

The control logic needs to provide the control signals

Truth table of output logic of controller
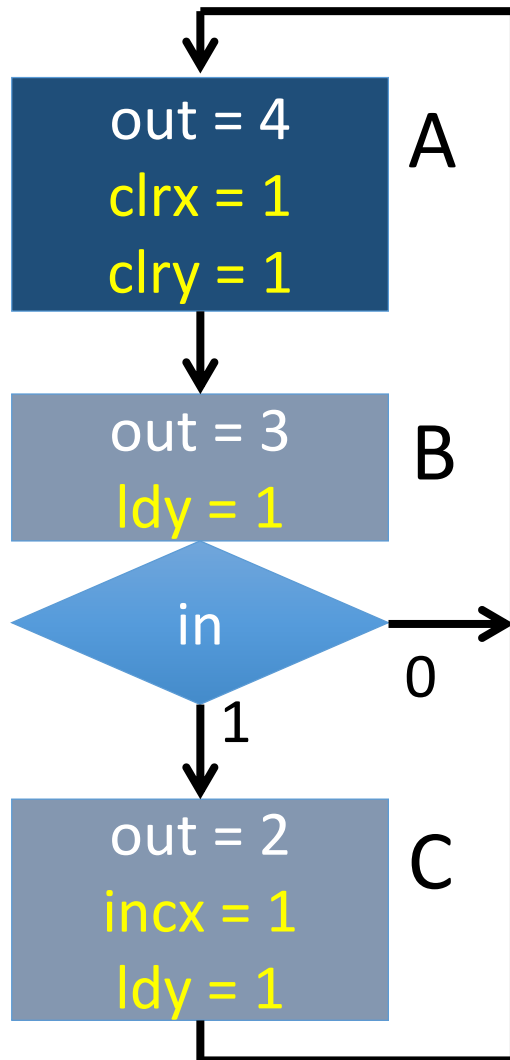
state	clrx	incx	clry	ldy	out
0 0	1	0	1	0	100
0 1	0	0	0	1	011
1 0	0	1	0	1	010

A   00
B   01
C   10

Truth table of next-state logic of controller

| out = 4 |
| clrx = 1 | A |
| clry = 1 |

| out = 3 | B |
| ldy = 1 |

◇ in

0

1

| out = 2 | C |
| incx = 1 |
| ldy = 1 |
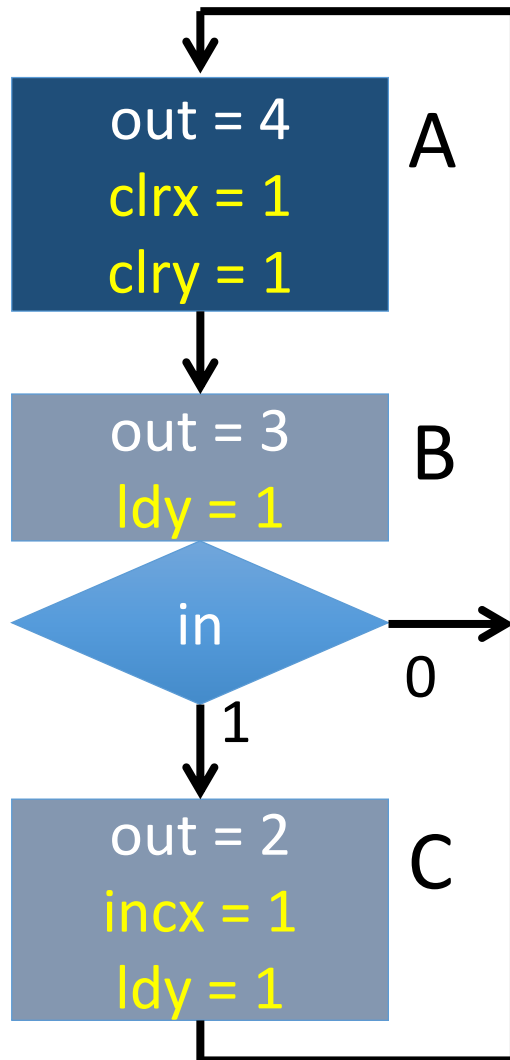
state	clrx	incx	clry	ldy	out
0 0	1	0	1	0	100
0 1	0	0	0	1	011
1 0	0	1	0	1	010

state	in	next_state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	A

Truth table of next-state logic of controller

out = 4	A
clrx = 1	
clry = 1	

| out = 3 | B |
| ldy = 1 | |

in

0

1

out = 2	C
incx = 1	
ldy = 1	
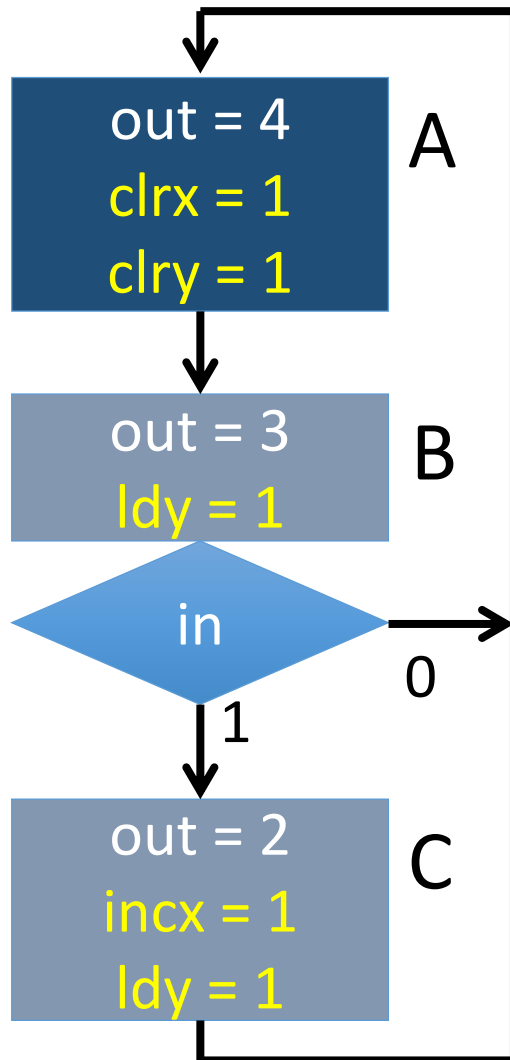
s1 s0	clrx	incx	clry	ldy	out
0 0	1	0	1	0	100
0 1	0	0	0	1	011
1 0	0	1	0	1	010
1 1	x	x	x	x	xxx

A   00
B   01
C   10

s1	s0	in	ns1	ns0
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	0	0
1	1	0	x	x
1	1	1	x	x

From truth table to implementation

outl:

nsl:
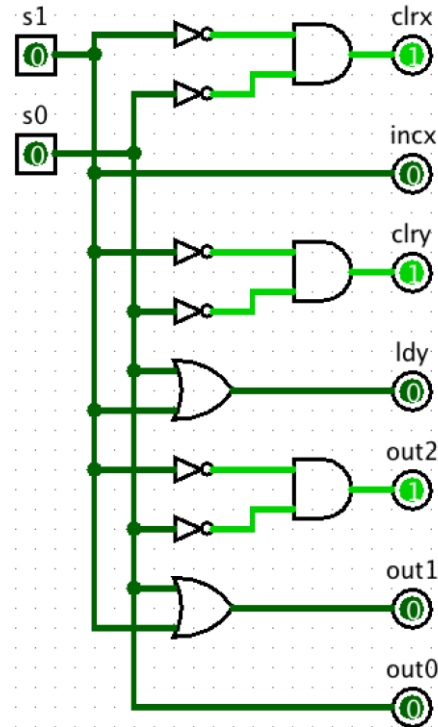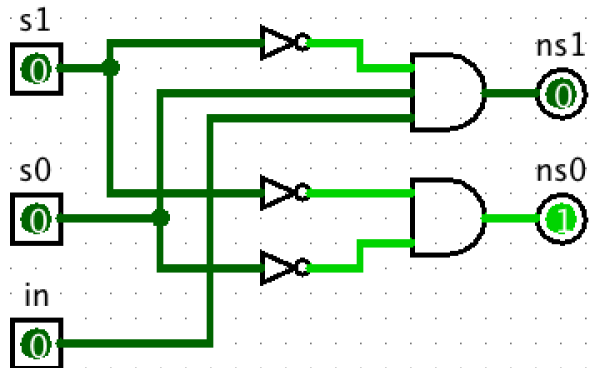
s1 s0	clrx	incx	clry	ldy	out
0 0	1	0	1	0	100
0 1	0	0	0	1	011
1 0	0	1	0	1	010
1 1	x	x	x	x	xxx

s1	s0	in	ns1	ns0
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	0	0
1	1	0	x	x
1	1	1	x	x

The controller and the data path
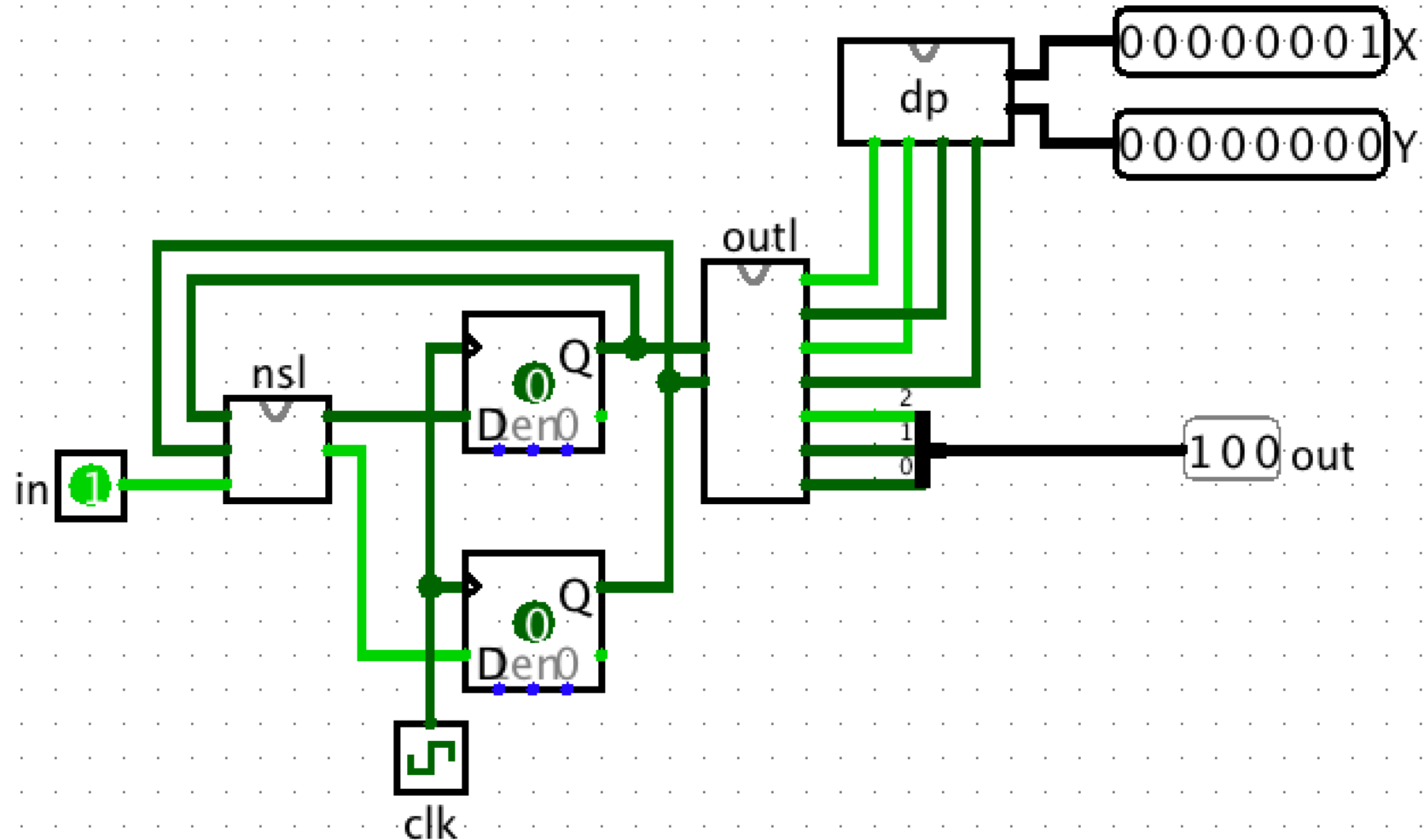
SystemVerilog

(Watch the tutorials for the practicals for details)

SystemVerilog coding style

- Suffix _o for module outputs, _i for module inputs
- Register variables with suffix _p for previous 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!

Overview of Coding Guidelines
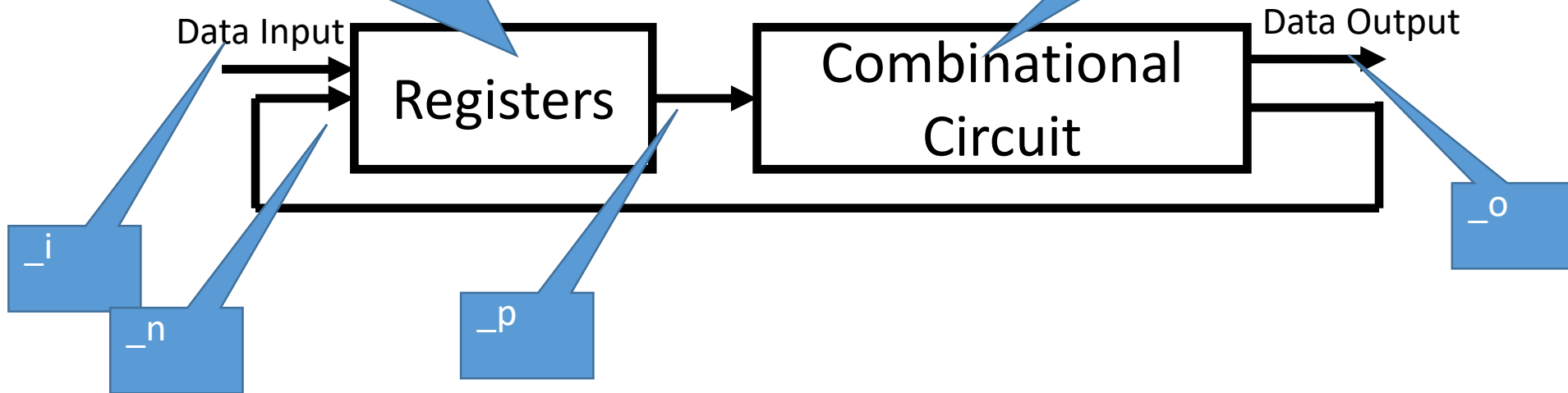
always_ff @(posedge clk_i or posedge reset_i)

Use Non-Blocking Assignments
a_p <= a_n;

Always_comb

Use Blocking Assignments
a = b;

Data Input

Registers
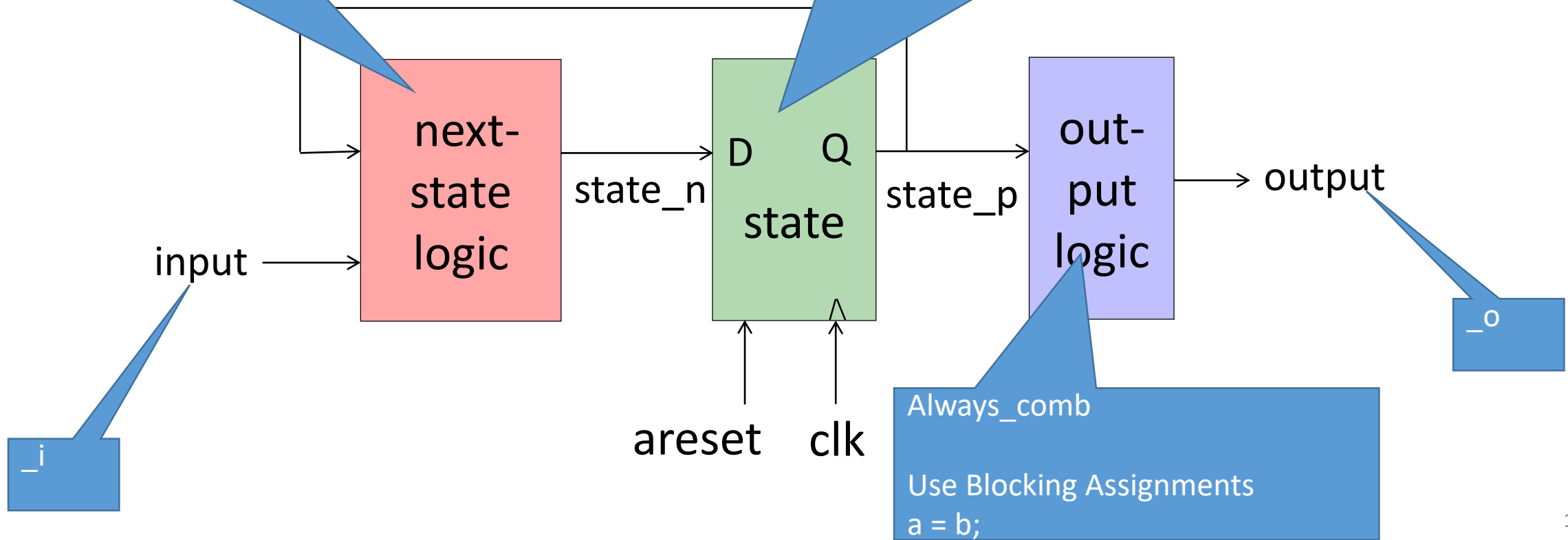
Combinational Circuit

Data Output

_i

_n

_p

_o

Essence of Moore Machines (Verilog)

always_comb

Use Blocking Assignments
a = b;

always_ff @(posedge clk_i or posedge reset_i)

Use Non-Blocking Assignments
state_p <= state_n;

next-state logic

D   Q
state

out-put logic

input

_i

state_n

state_p

output

_o

areset   clk

Always_comb

Use Blocking Assignments
a = b;

SystemVerilog Examples

- All FSMs of this lecture are available in SystemVerilog as well