Computer Organization and Networks

Chapter 3: Finite State Machine

IAIK TU Graz

http://www.iaik.tugraz.at

Why should you read this primer?

With this set of slides I guide you through the essence of designing finite state machines.

With each slide, you get drawn into the topic step by step.

I use very simple examples in order to concentrate on the essence.

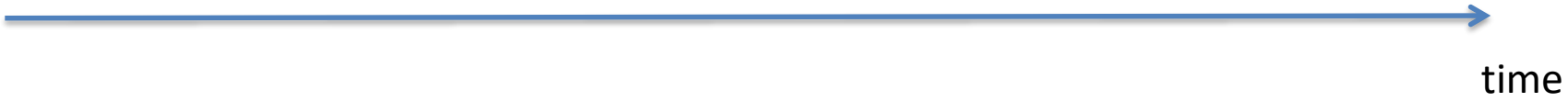
I assume that you know…

- what the term “function” means in mathematics.
- that in logic functions we use only 0s and 1s.
- that a logic function can be uniquely described with a truth table.
- that a truth table lists all possible input combinations in an ordered fashion.
- that the symbol “~” is used for logic negation.
- that the symbol “&” is used for logic ANDing.
- that the symbol “|” is used for logic ORing.

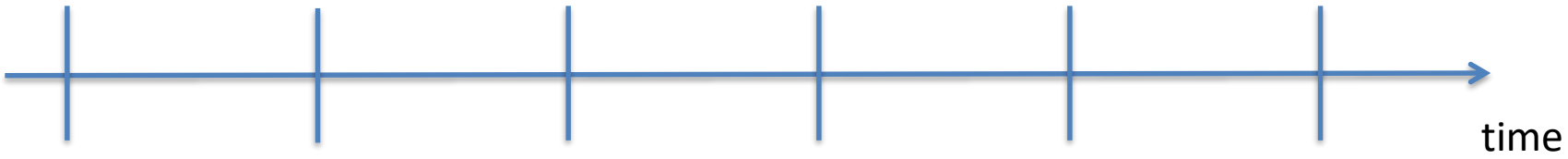
Finite State Machines (FSMs)

- 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”

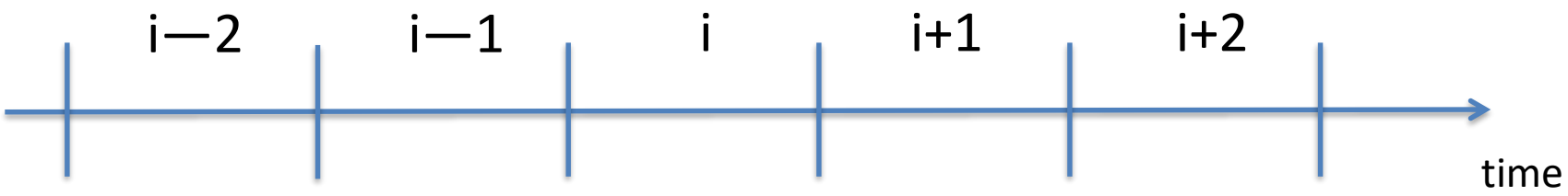
Time flows continuously



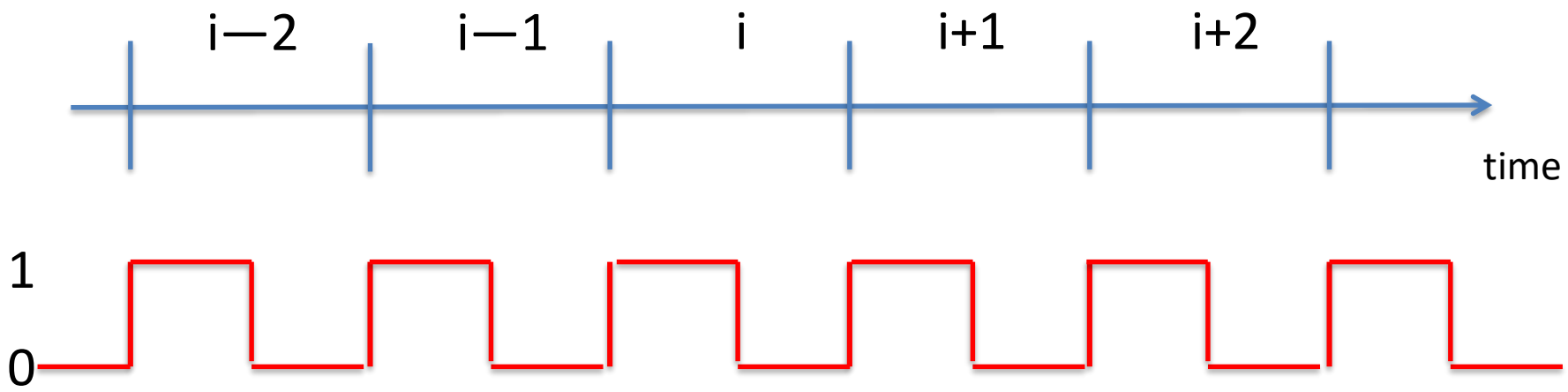
We cut time into slices



Time slices are strictly ordered

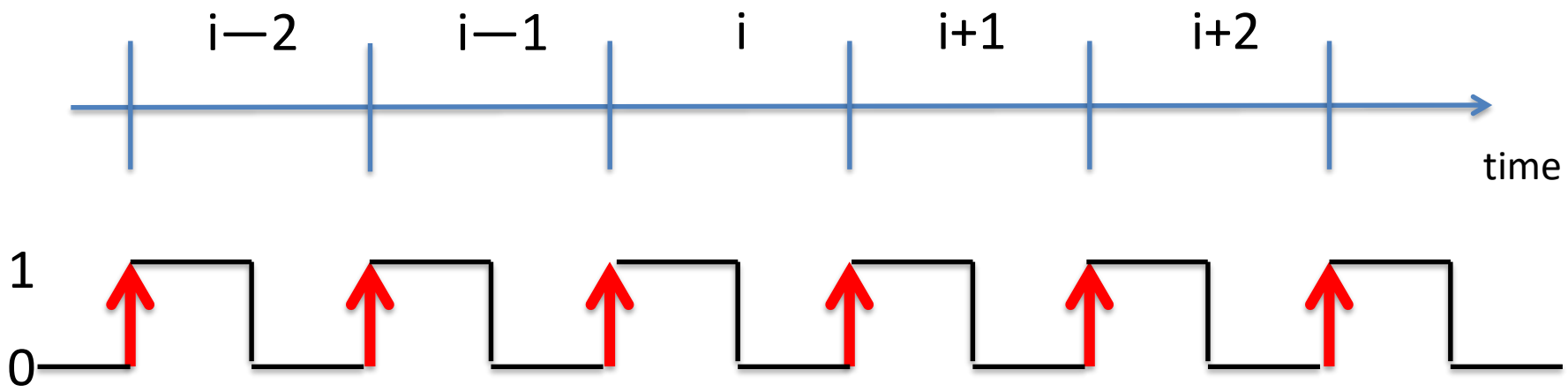


Clock signal



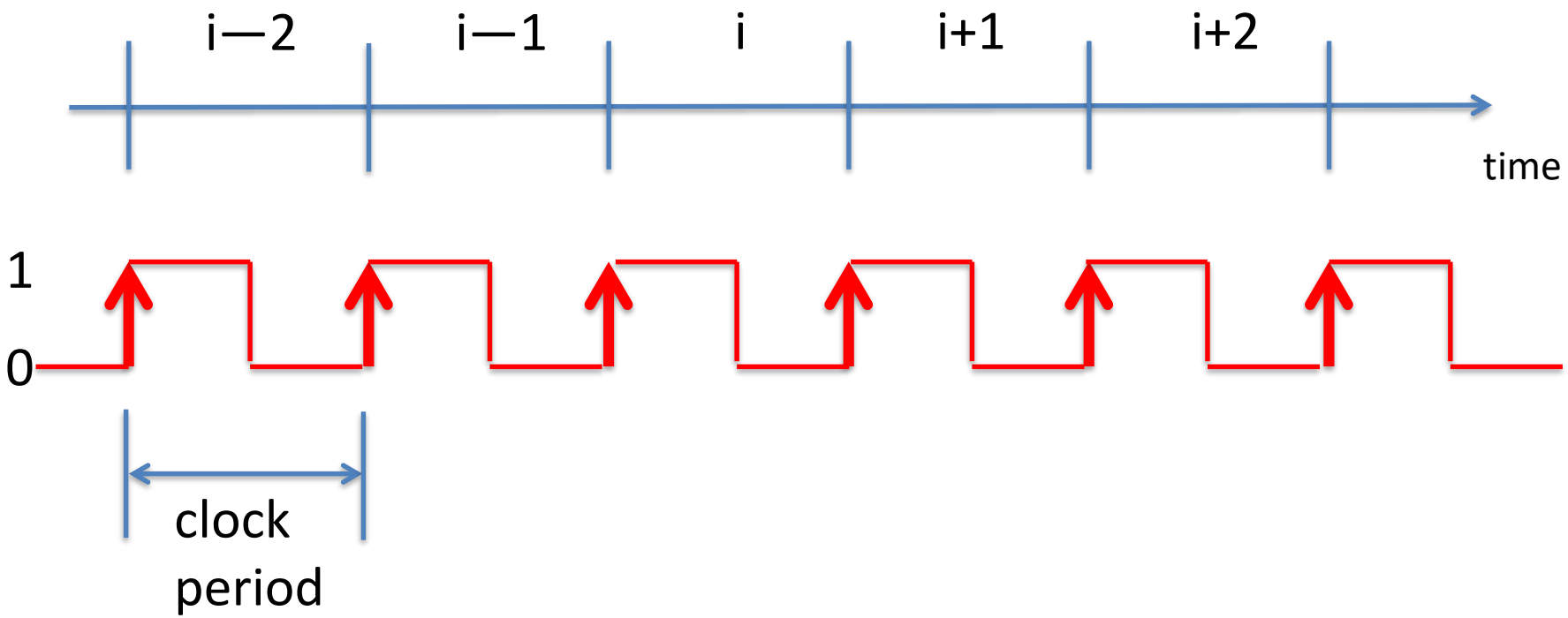
We use the clock signal (“clk”) in order to advance time from slice to slice.

Rising clock edge



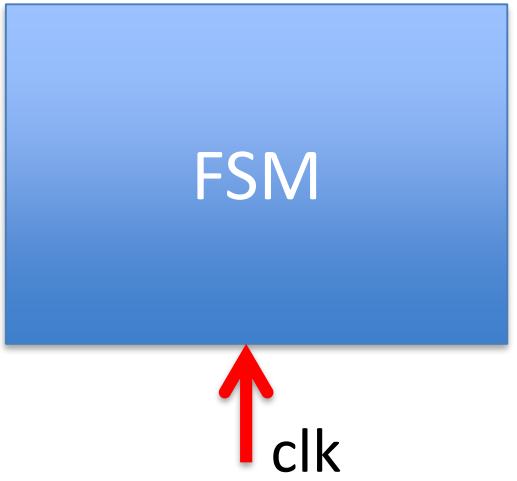
With **rising clock edges** we define the transition between neighboring time slices.
The negative clock edges have no importance.

Clock period



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.

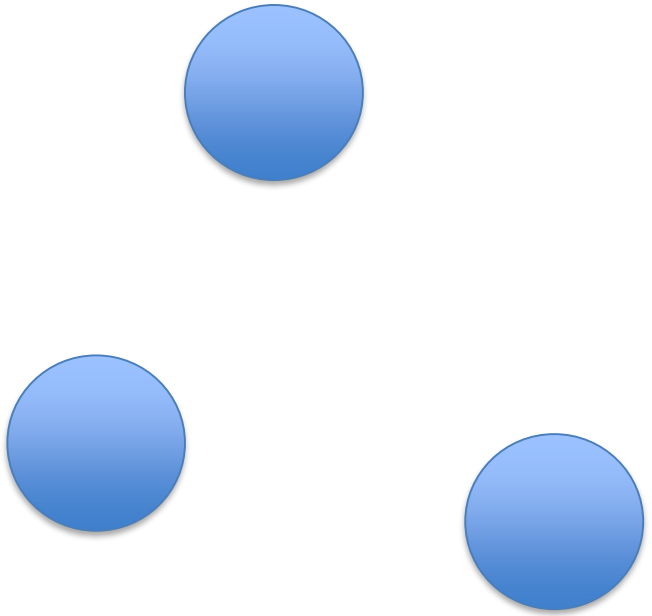
Synchronous finite state machine (= automaton)



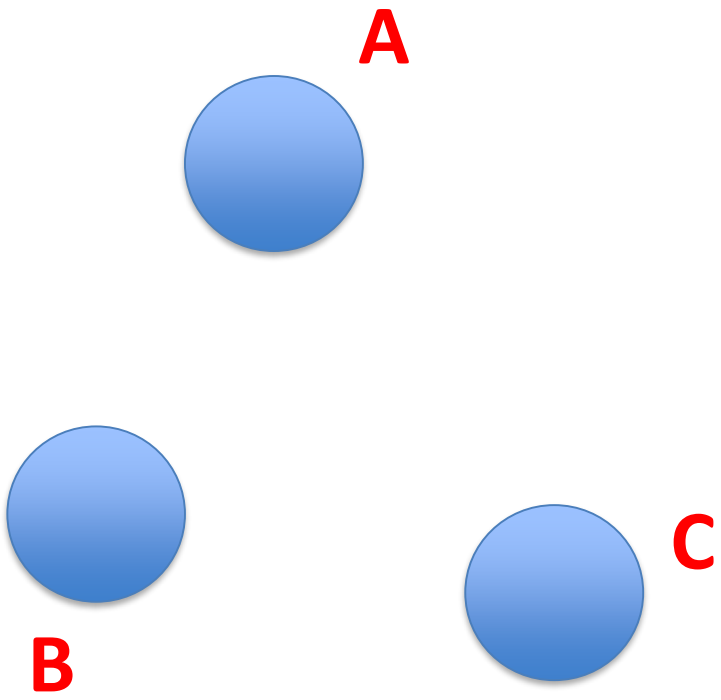
A **synchronous FSM** is clocked by a clock signal ("clk").
In each clock period, the machine is in a defined (current) **state**.
With each rising edge of the clock signal, the machine advances to a defined next state.

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**.

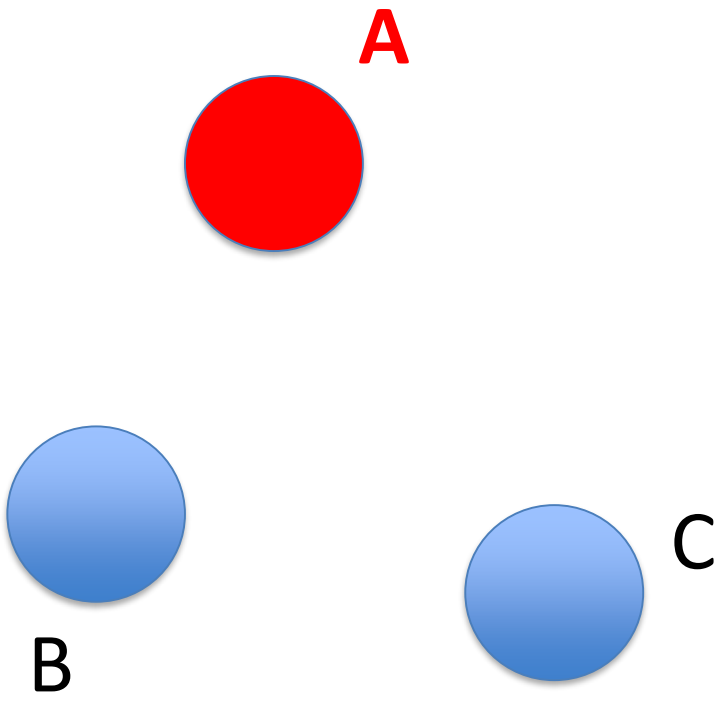


State diagram:



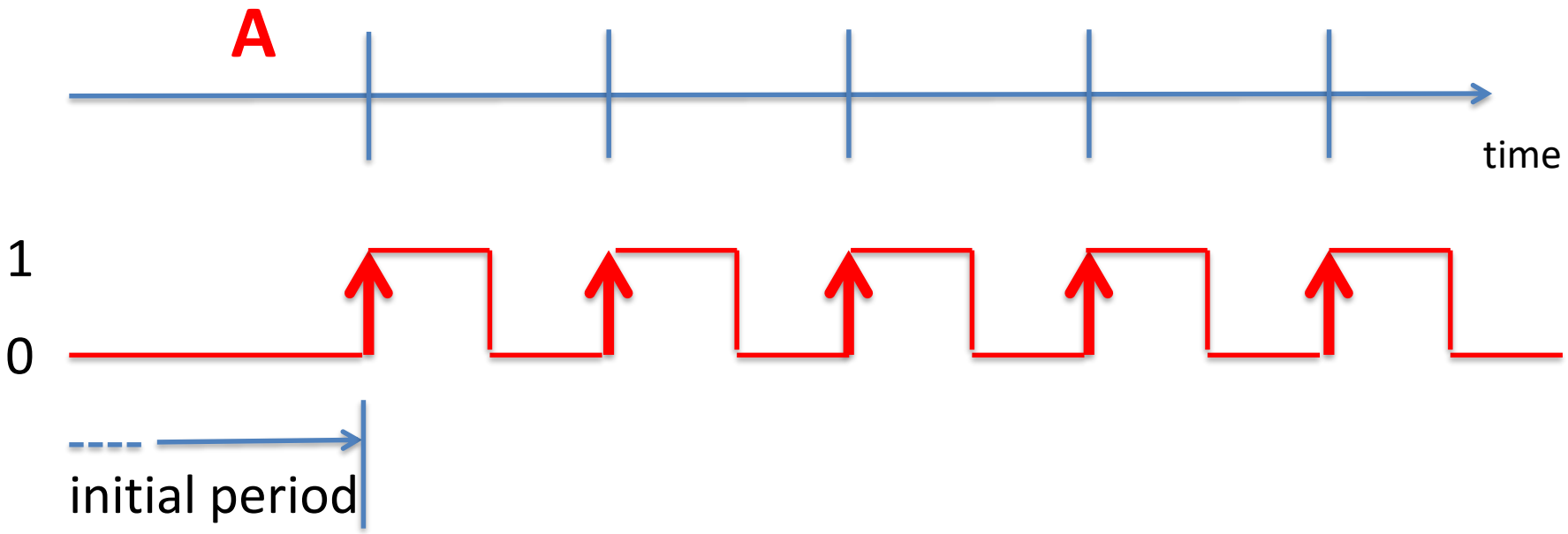
We denote the states with circles and give them **symbolic names**, e.g. A, B, and C.

State diagram:



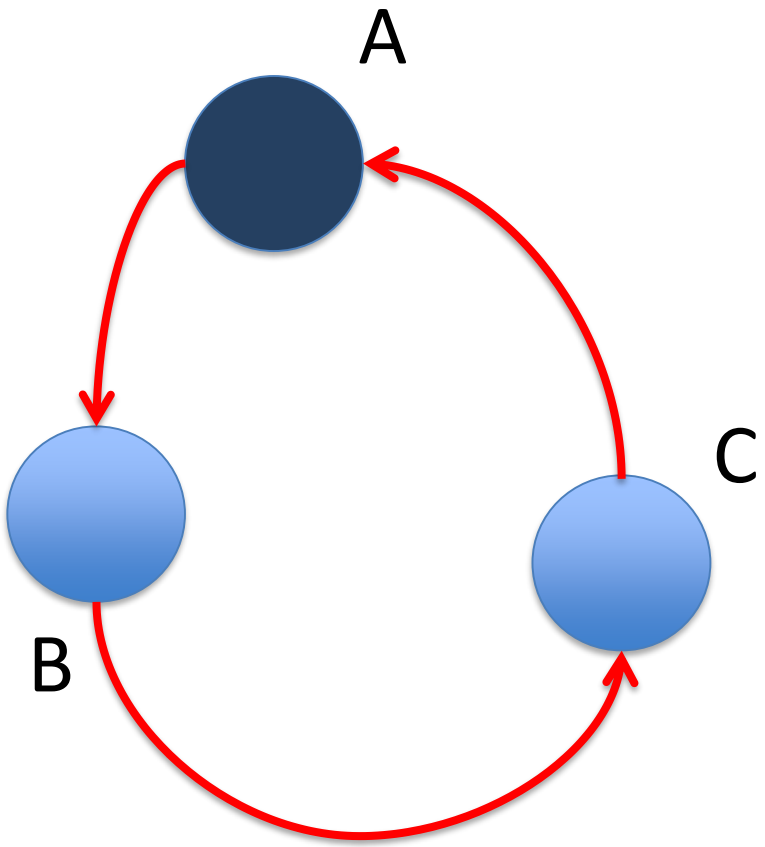
We define one of the states as the **initial state**.

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".

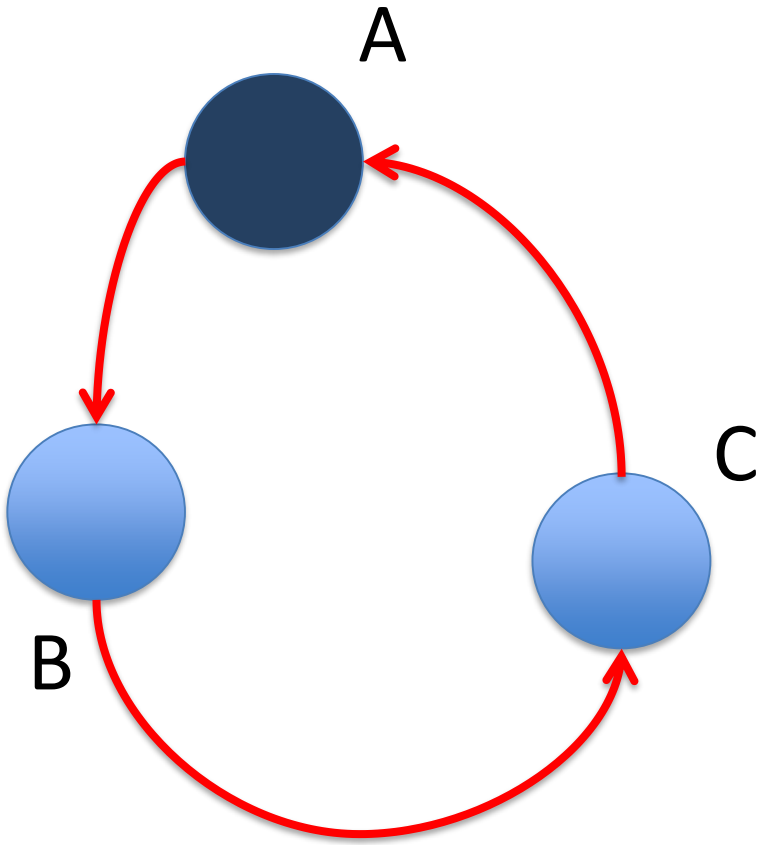
State diagram:



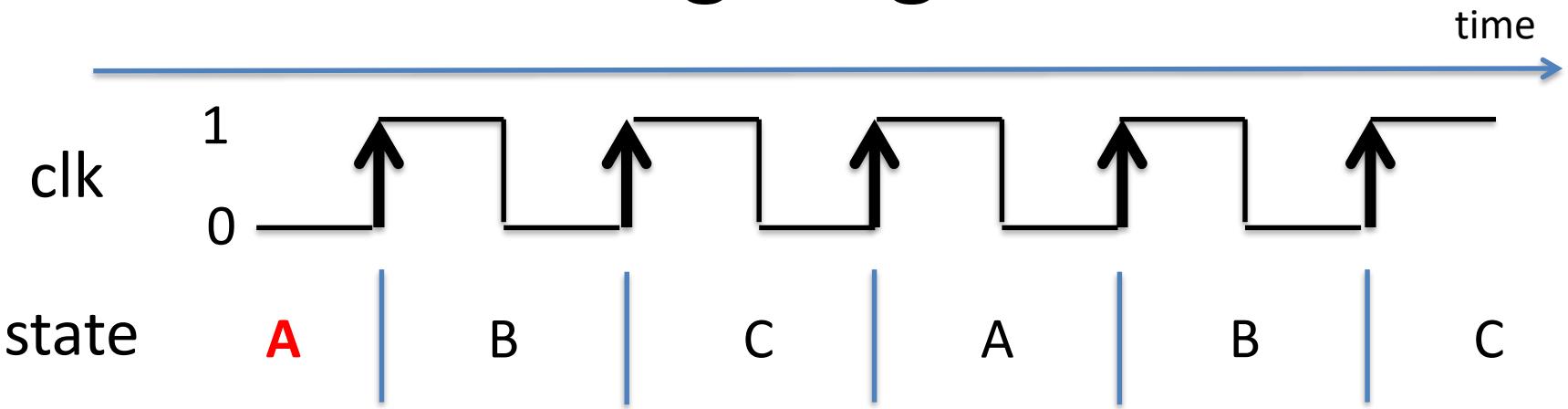
With arrows we define the sequence of states.

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

current state	next state
A	B
B	C
C	A



Timing diagram



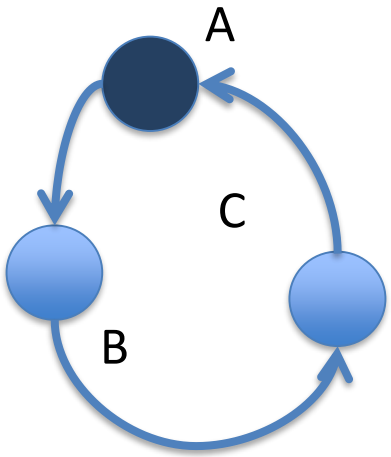
Initially, the FSM is in the initial state A.

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

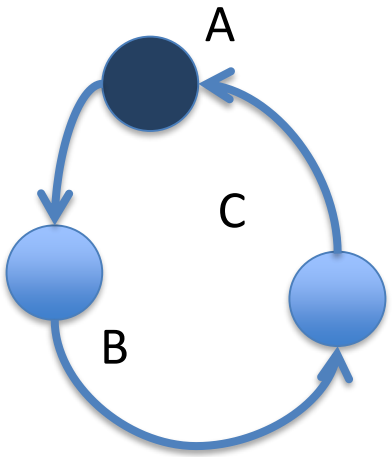
An FSM has **always** a next state.

In order to technically realize ("implement") a state diagram,

we start by giving each state a unique number.

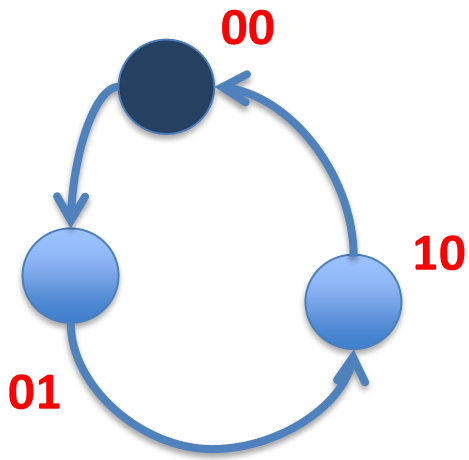


Popular state-encoding schemes



- Binary encoding
  - needs minimum amount of flip-flops.
- One-hot encoding
  - Tends to have a simpler next-state logic.

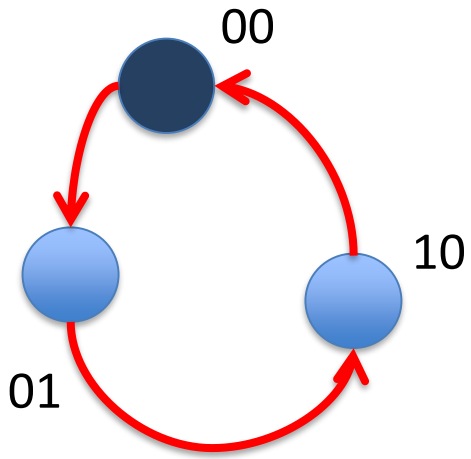
Binary encoding



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

For a Verilog example of this FSM see

chapter_03/example_00/01_fsm_moore_no_input.v

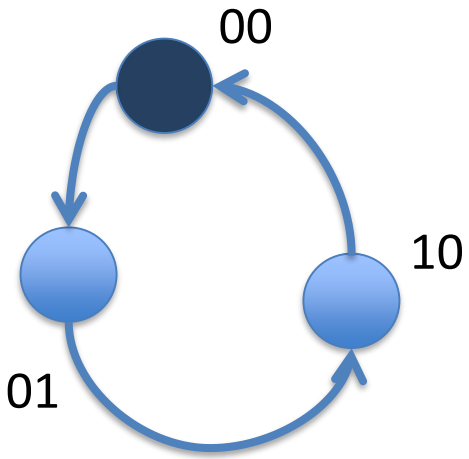


Binary encoding:

The state transition table has also only binary numbers.

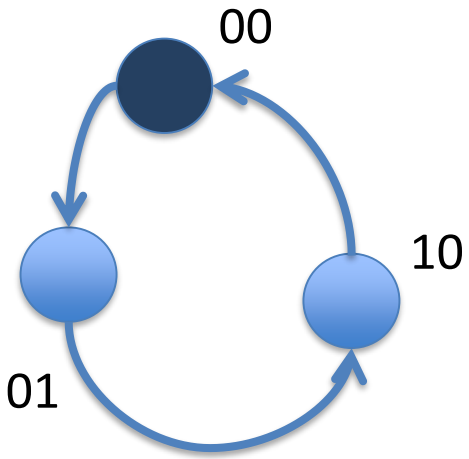
current state	next state
00	**01**
01	**10**
10	**00**

We also enter the unused combination “11”.
This state does not exist.
“x” stands for “Don’t care”.



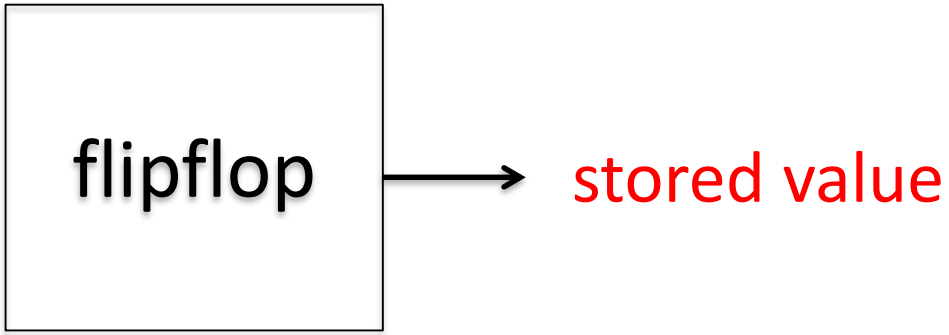
current state	next state
00	01
01	10
10	00
11	xx

We define names for the two state bits, e.g. s1, s0.

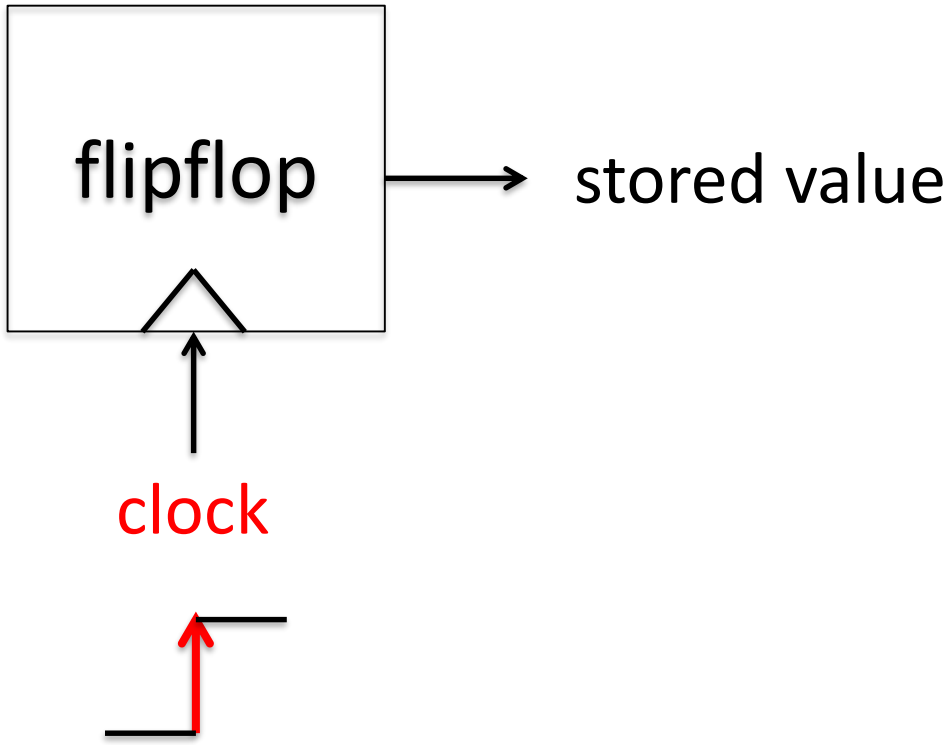


current		next	
s1	s0	s1	s0
0	0	0	1
0	1	1	0
1	0	0	0
1	1	x	x

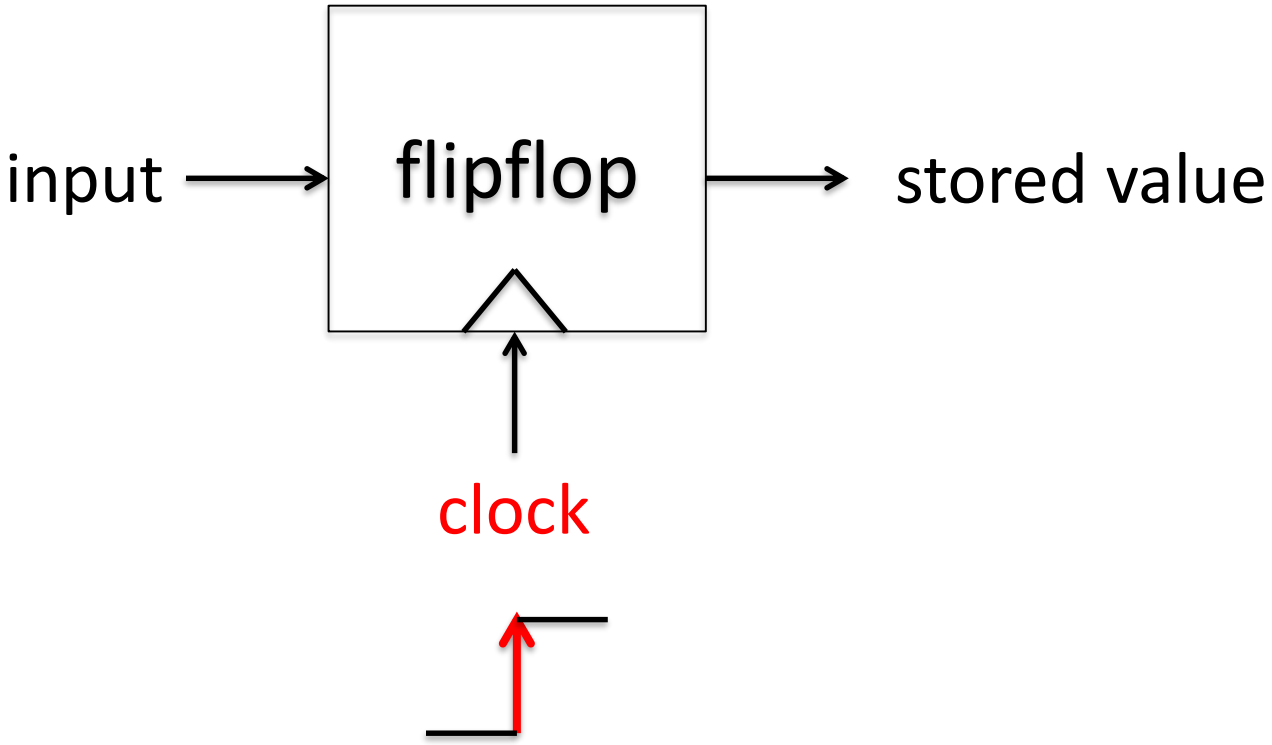
Each state bit is stored in a flipflop



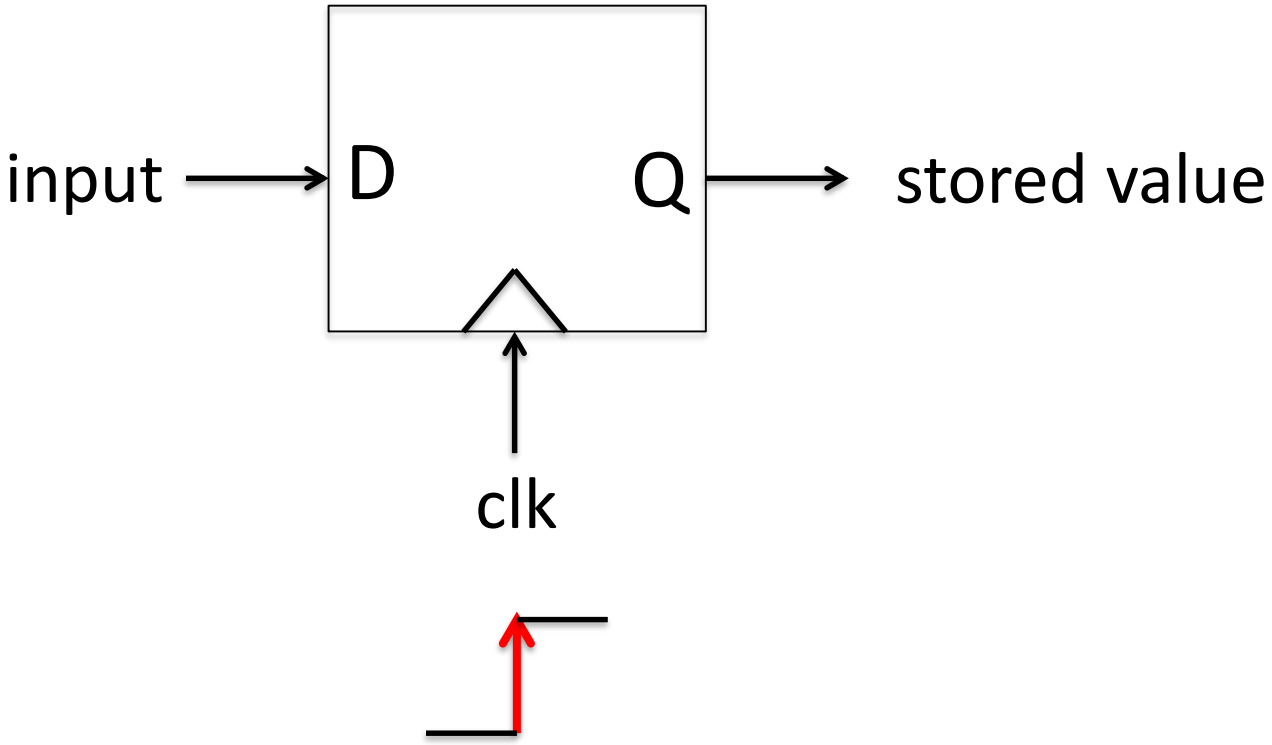
A flipflop stores a new value, when a rising edge occurs on the clock input



The flipflop stores the value, which it sees on its input.

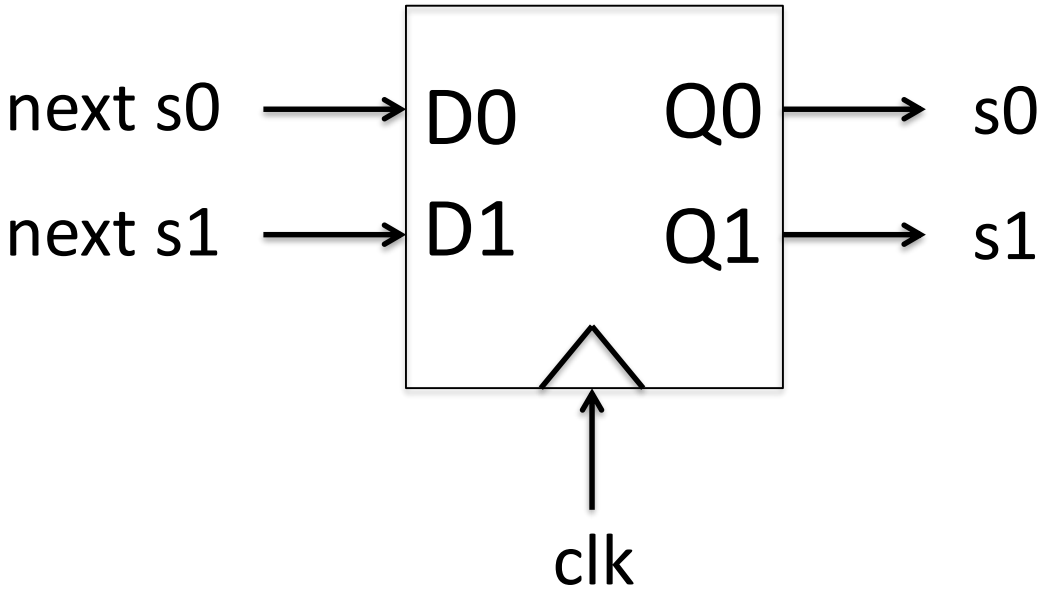


Summary: A D-flip-flop samples its input value D and stores this value when a rising edge of clk occurs



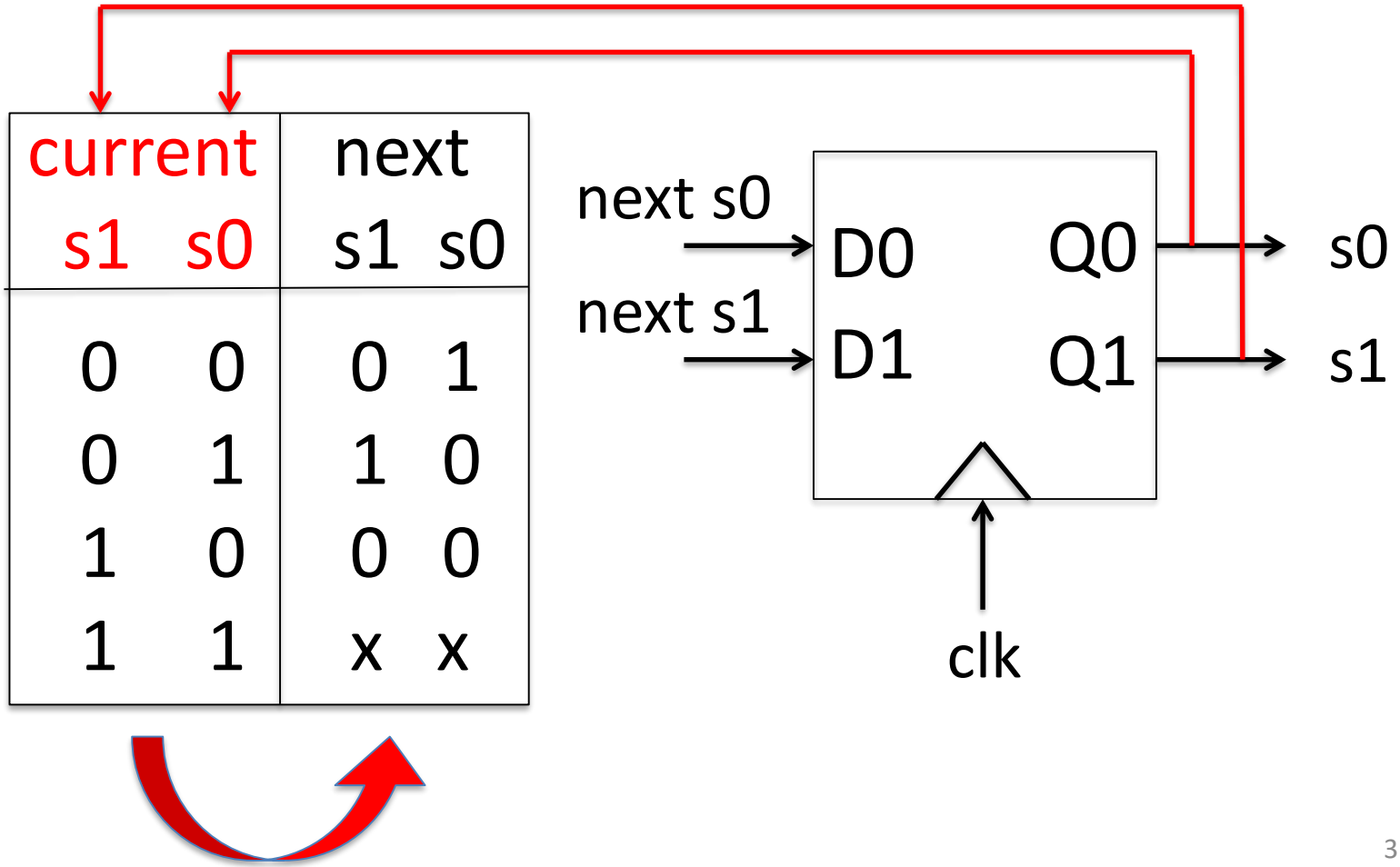
Thus, a D-type flipflop can be seen as a 1-bit photo camera.

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

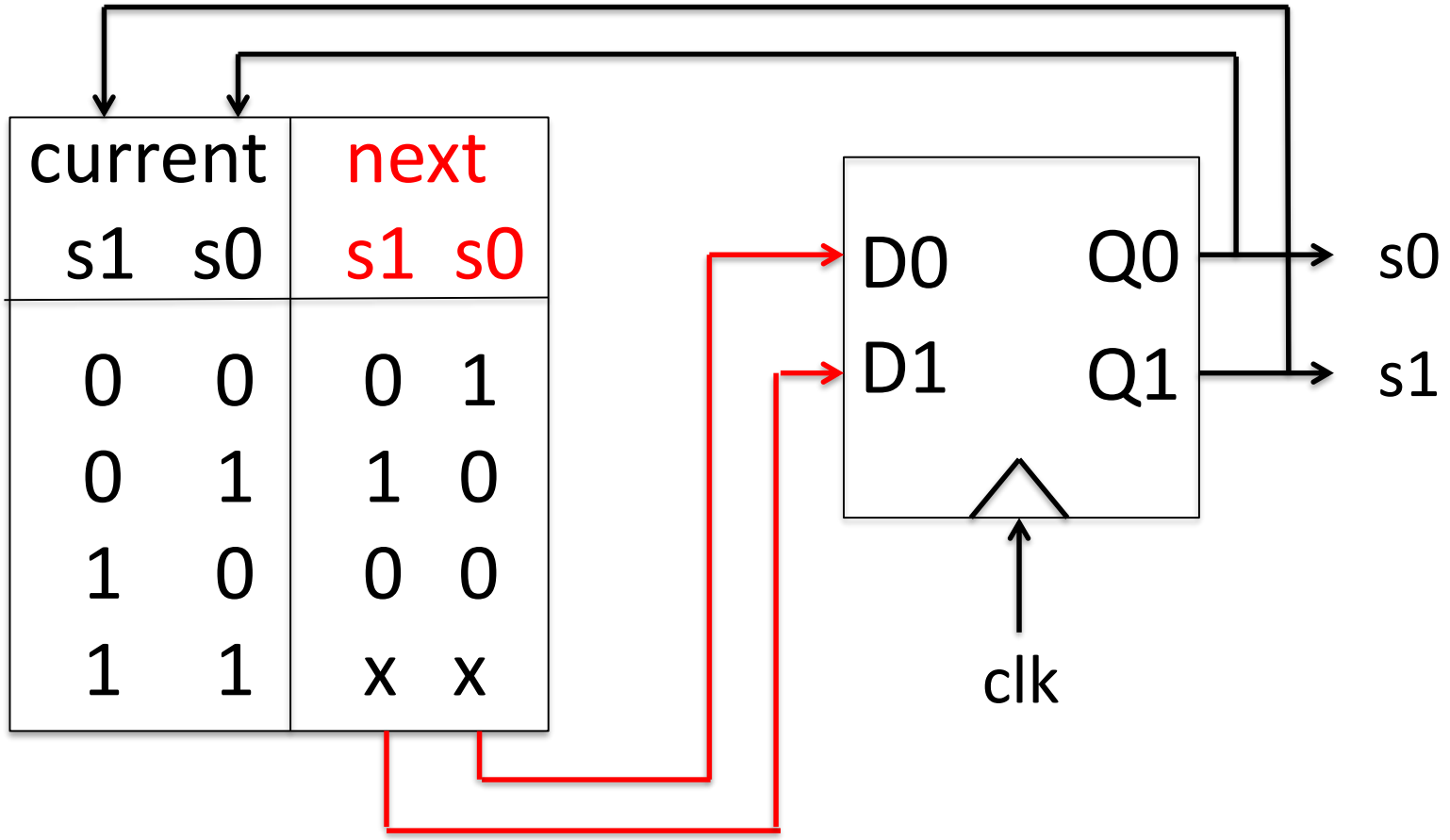


Flipflops which can store several bits are also called “registers”.

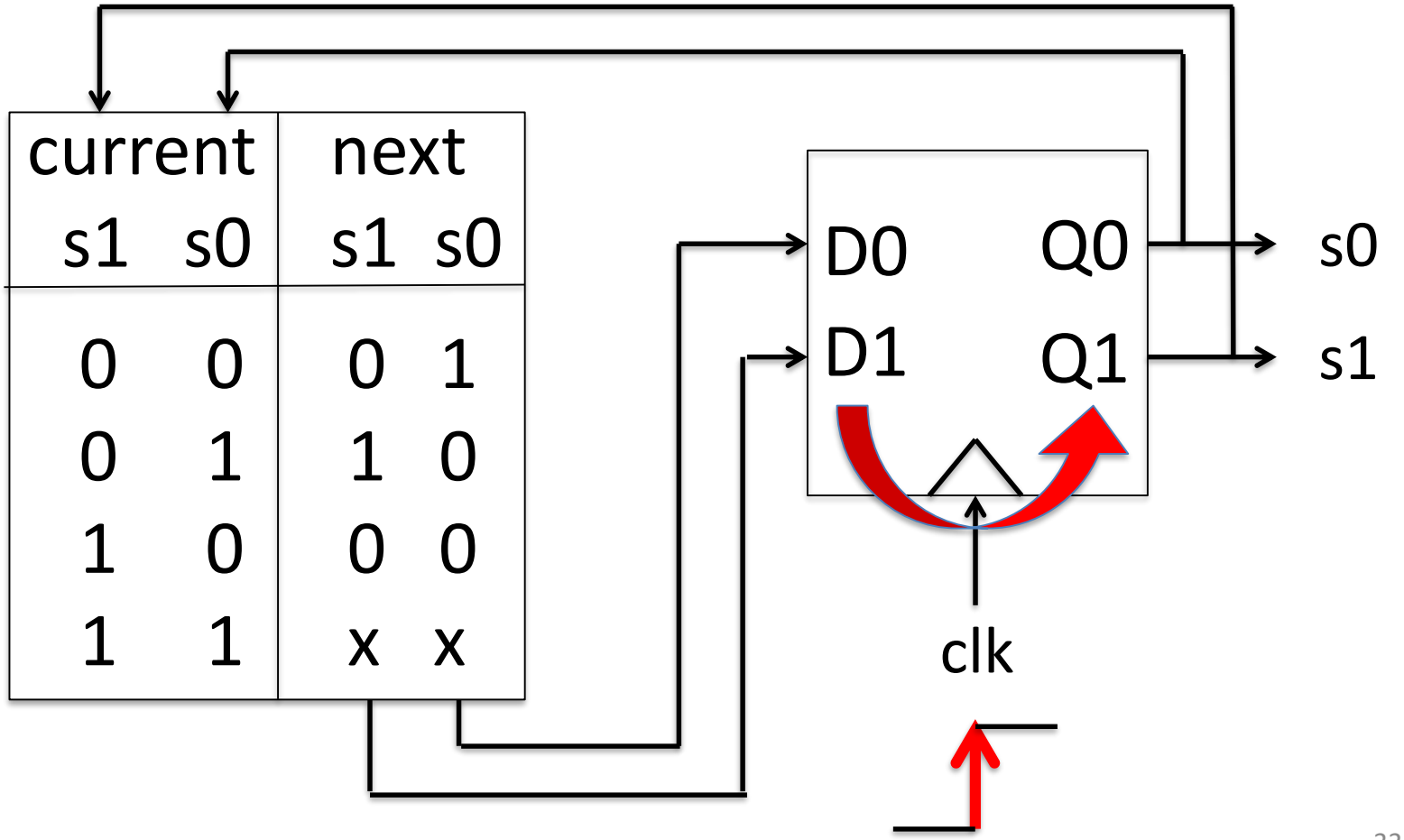
We use the state transition table as a "lookup table"…



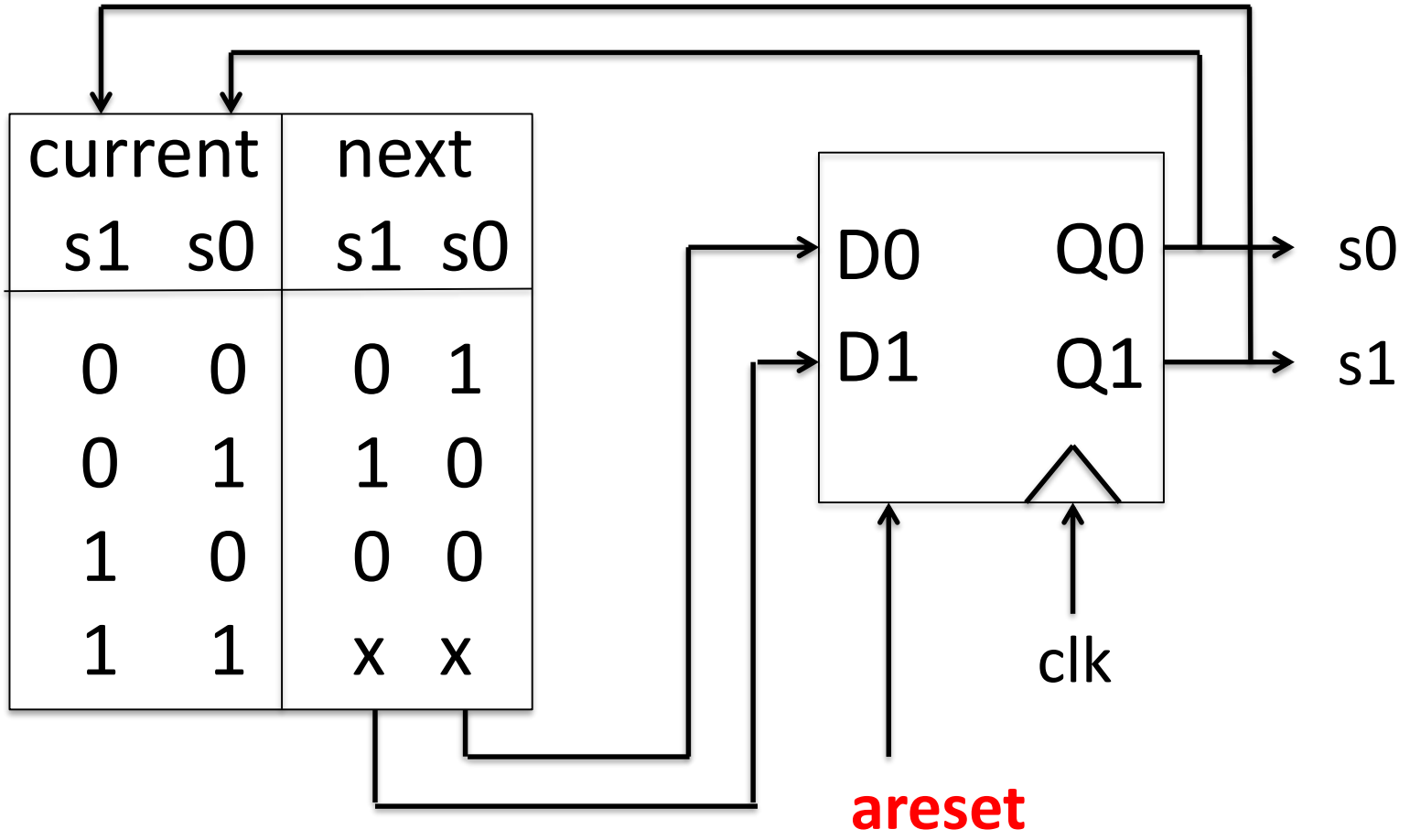
…and thus find out the next state



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

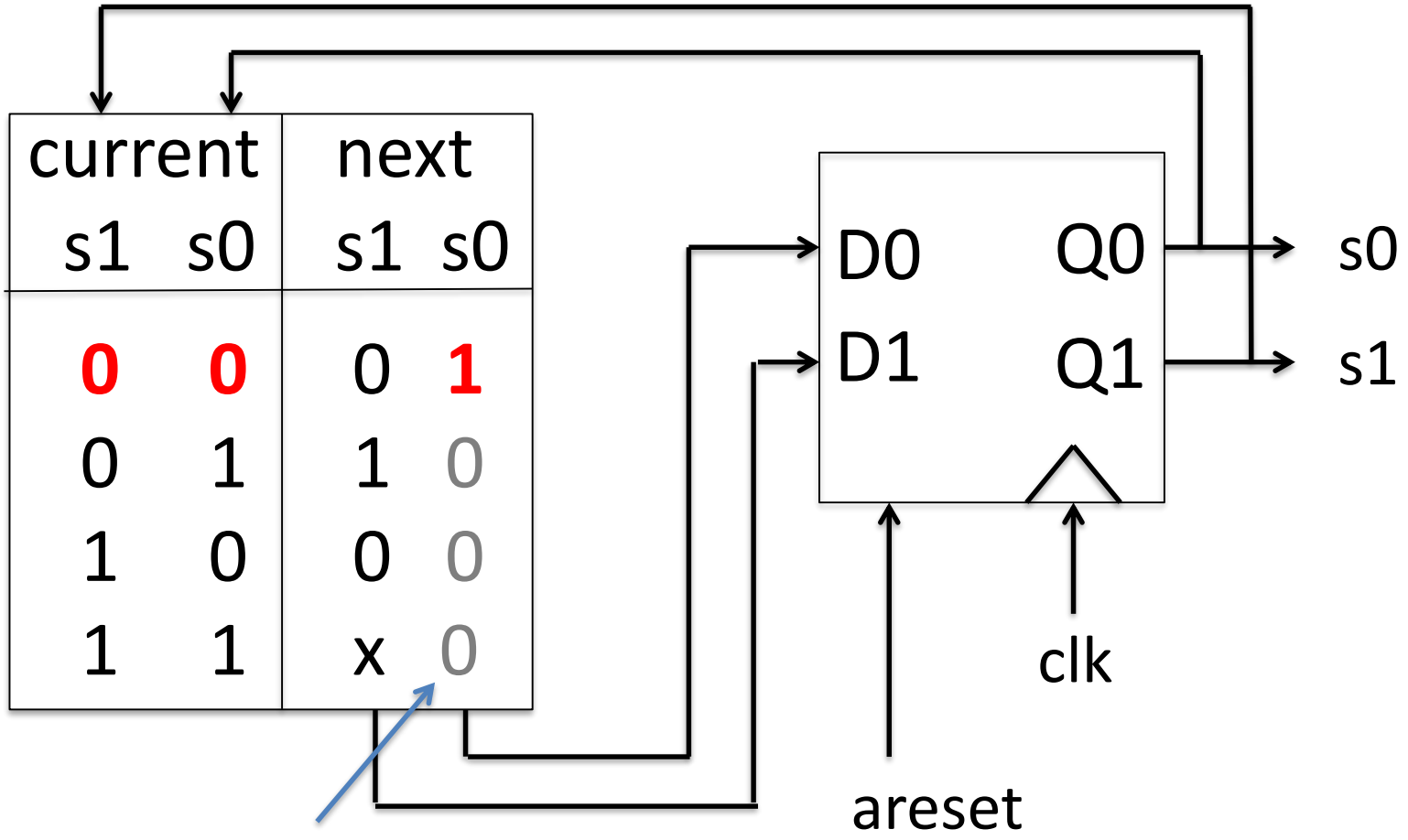


In order to get the initial state “00”, we use flipflops with an “asynchronous reset input”. Shortly after switching on the circuit, we apply “areset”.



The state transition function in this example can be derived from the truth table:

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

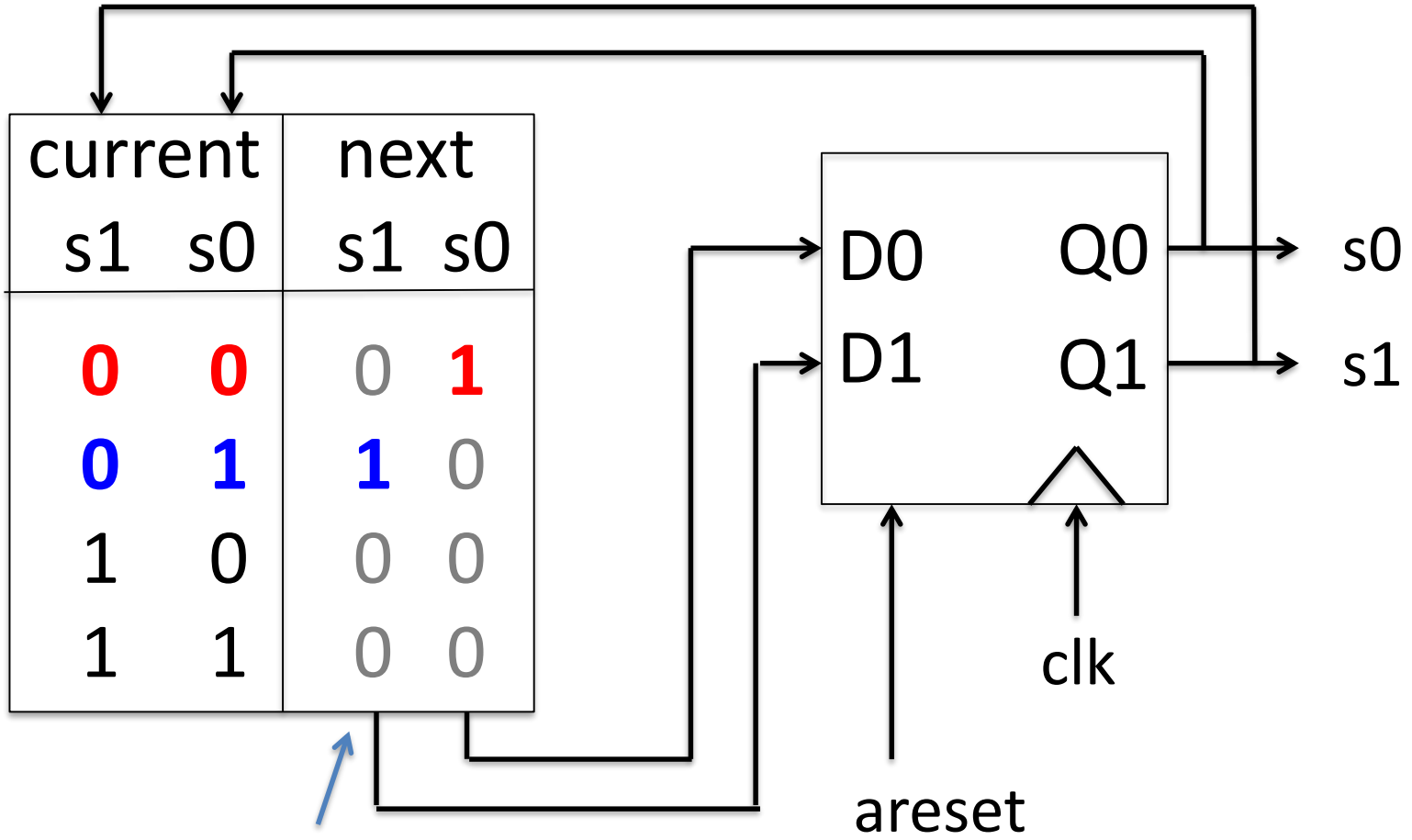


We set the don't care to 0.

The state transition function:

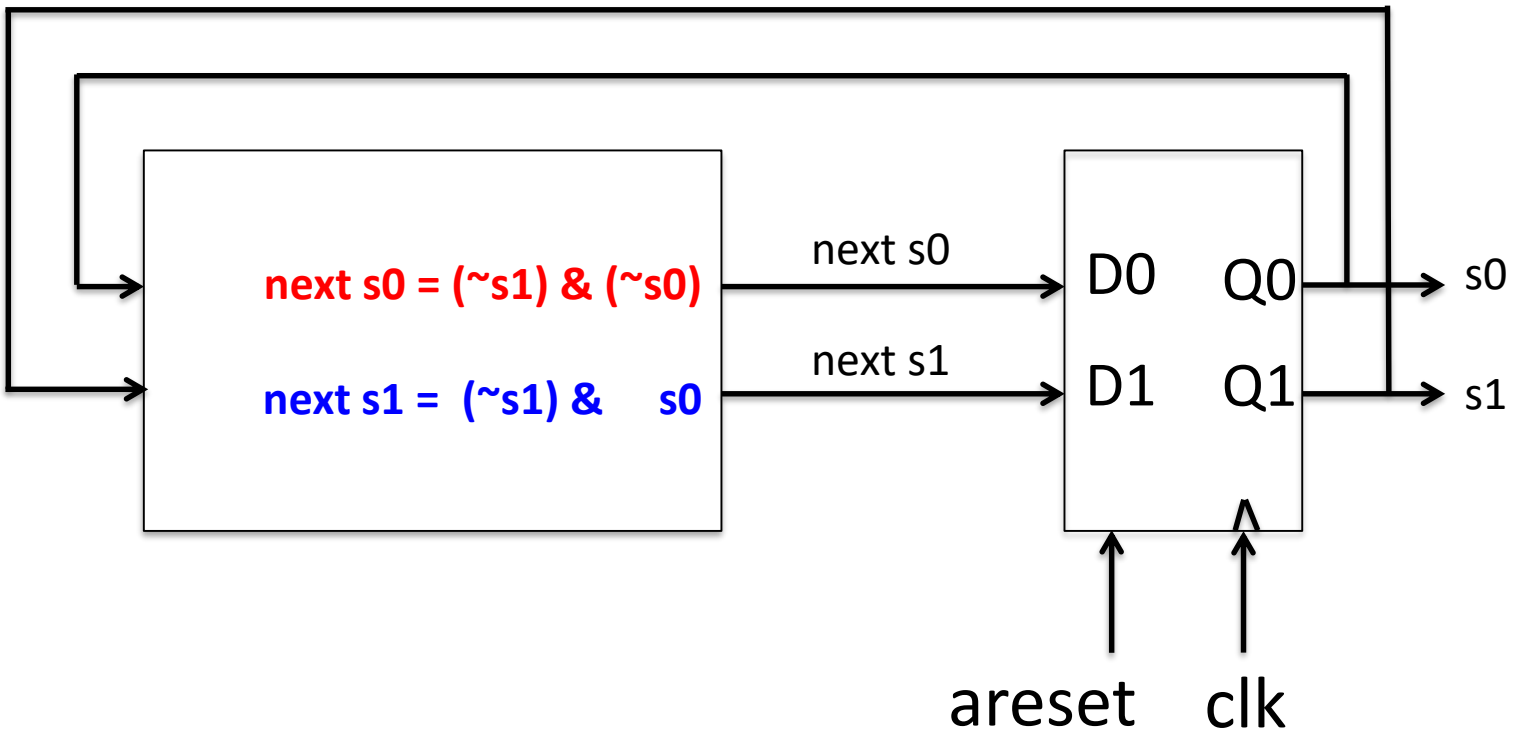
next s0 = (~s1) & (~s0)

next s1 = (~s1) & s0



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

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



Essence so far

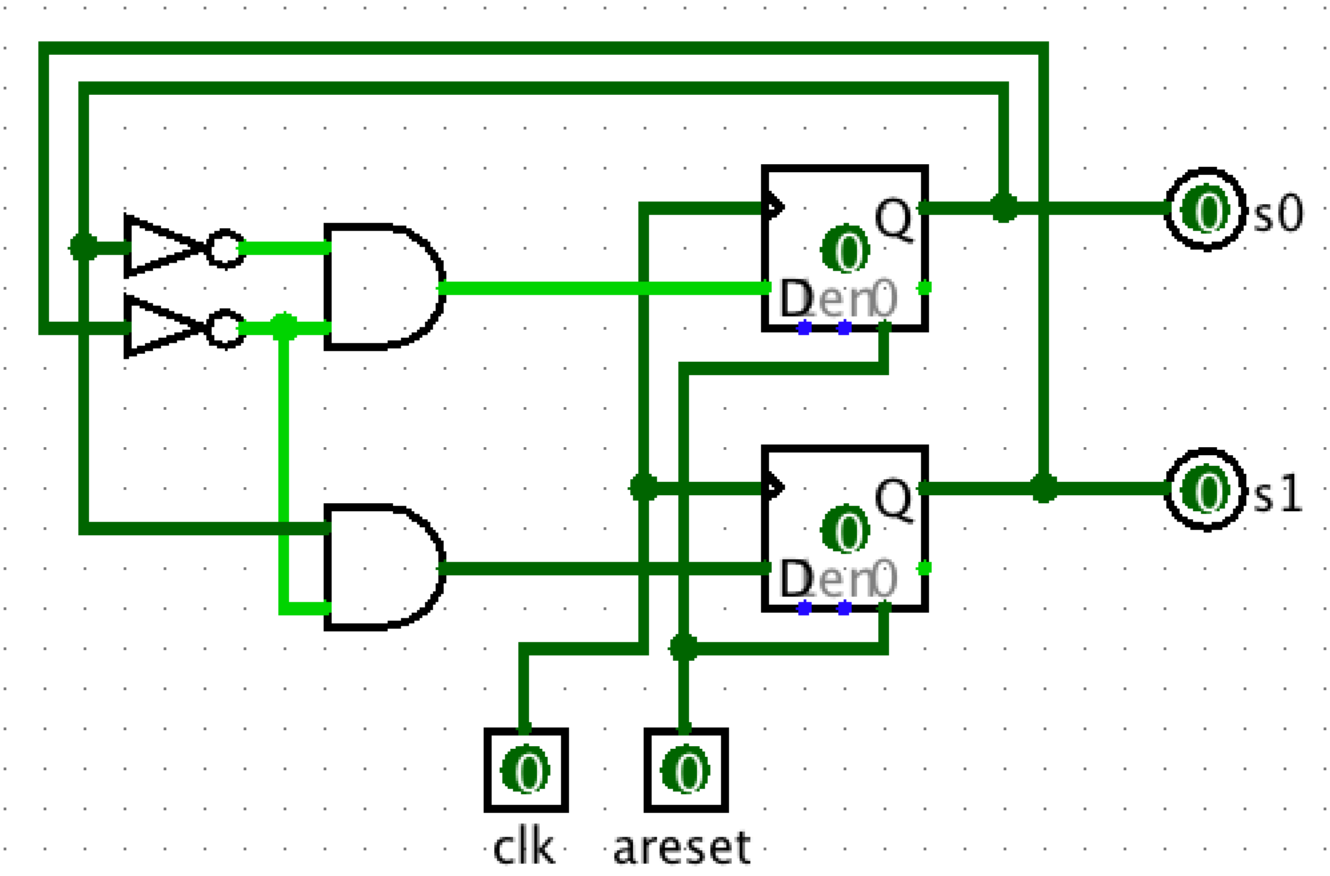
State diagram:

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

State transitions:

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

The FSM modeled and simulated with Logisim

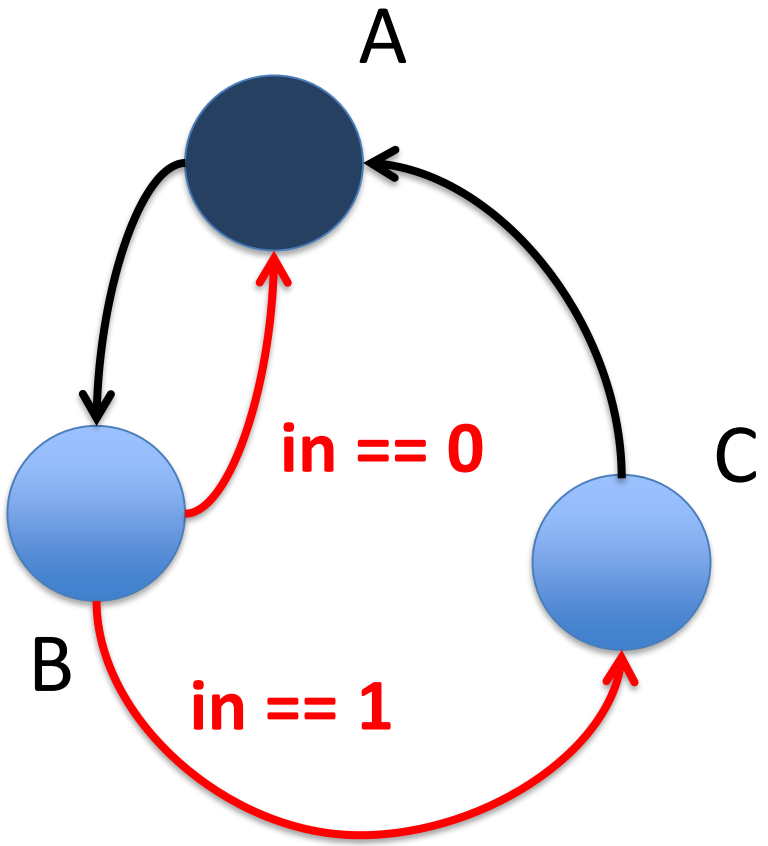


Inputs

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

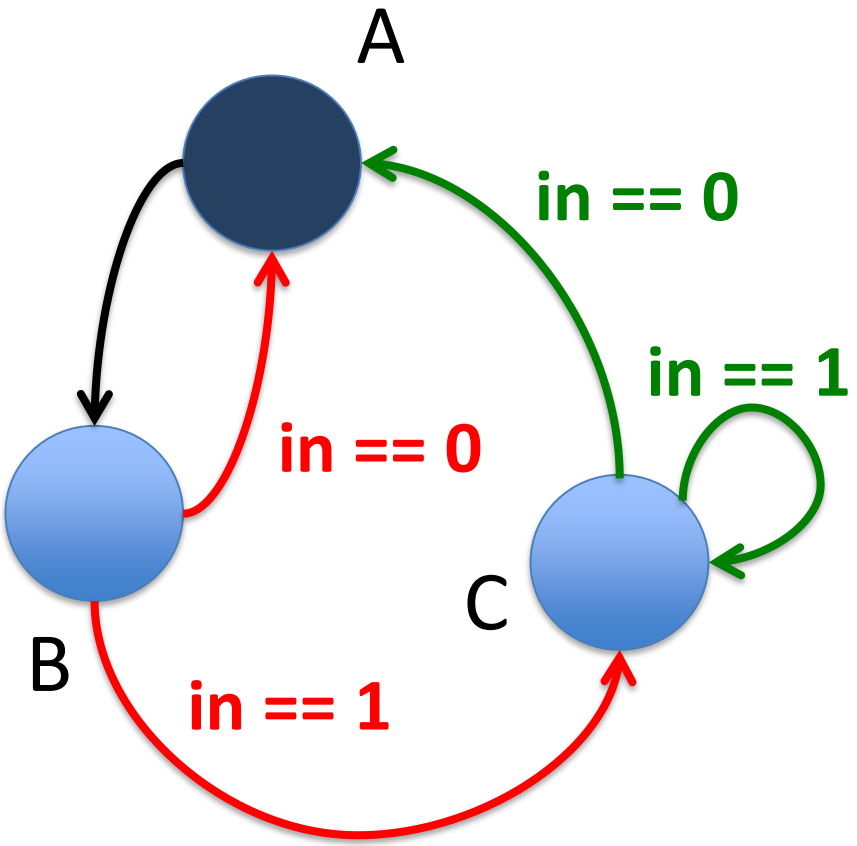
next state = f(state, input)

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



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

next state = f(state, input)

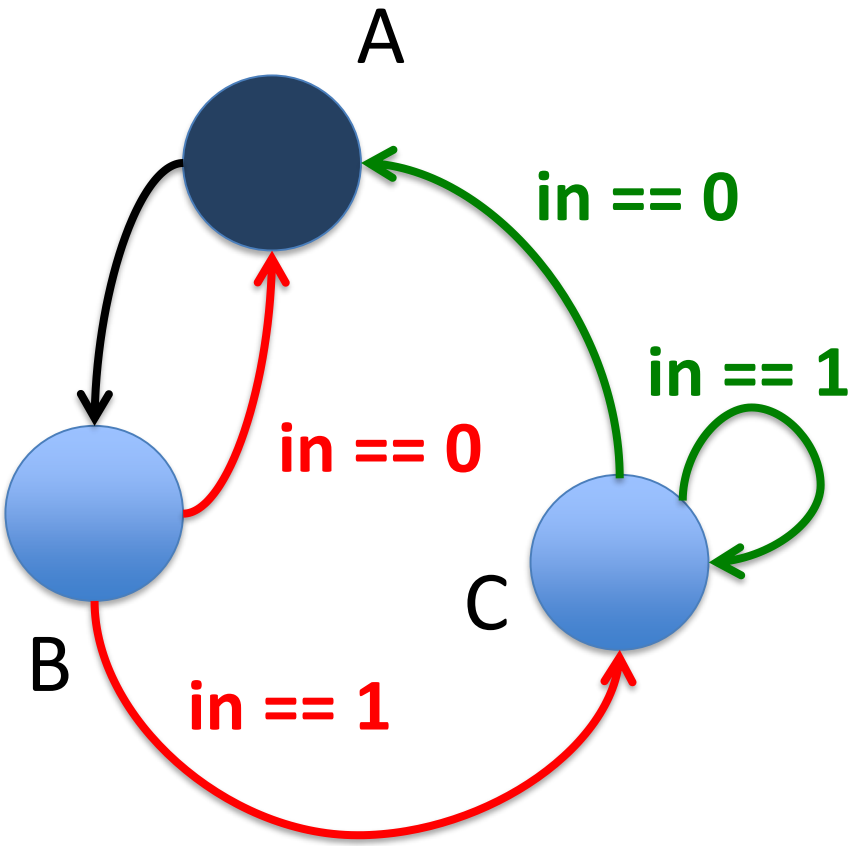


In this example we see that the one-bit input "in" influences the choice of the state after B.

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

The state transition table

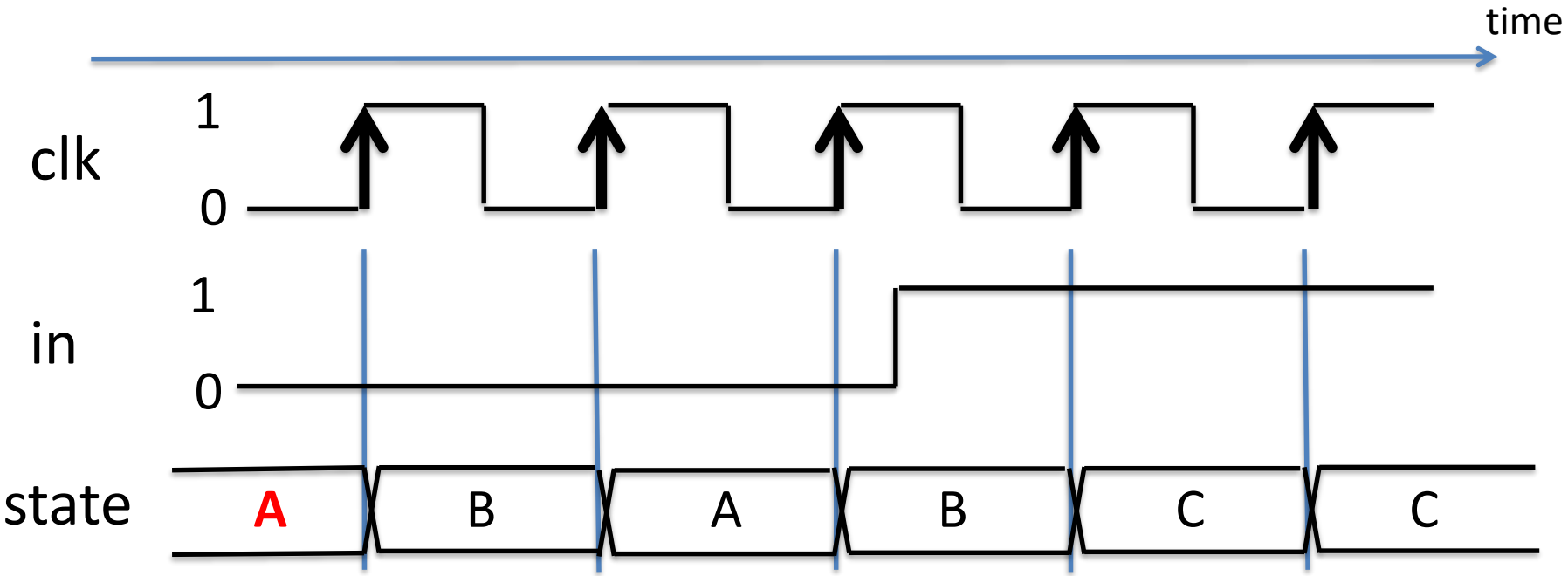
current state	in	next state
A	0	B
A	1	B
B	0	A
B	1	C
C	0	A
C	1	C



For a Verilog example of this FSM see

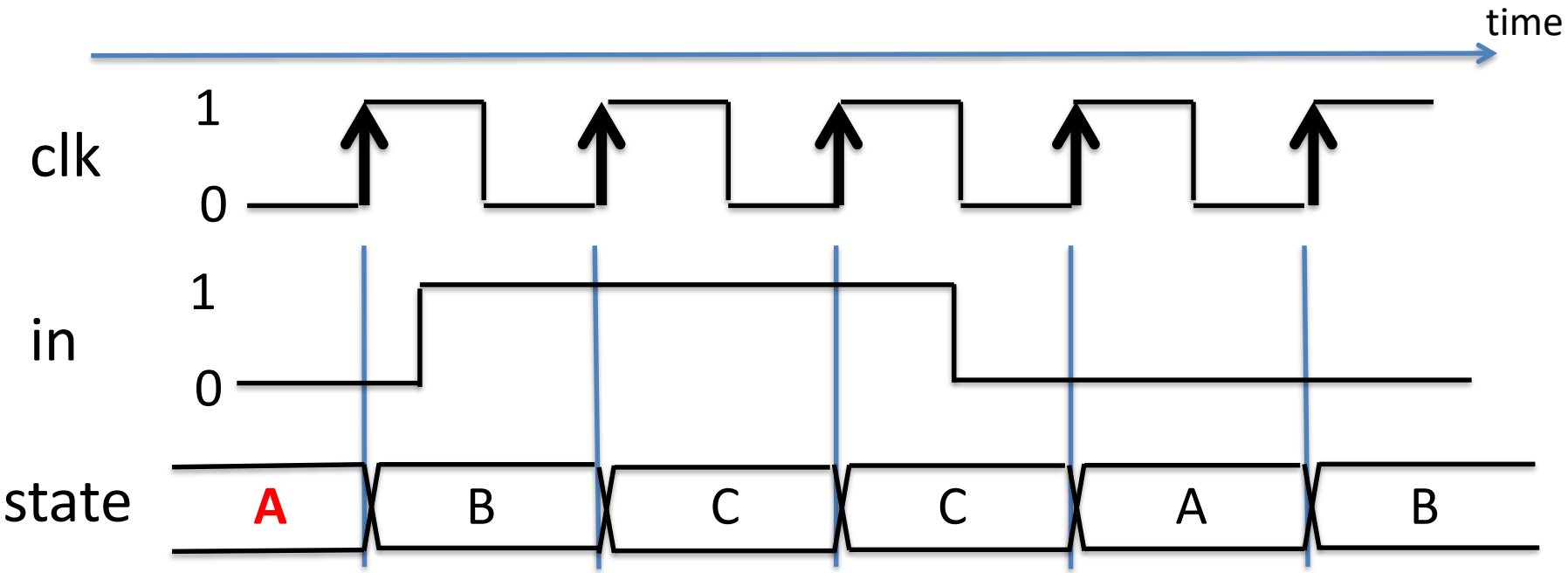
chapter_03/example_00/01_fsm_moore_no_output.v

Timing diagram. Example 1



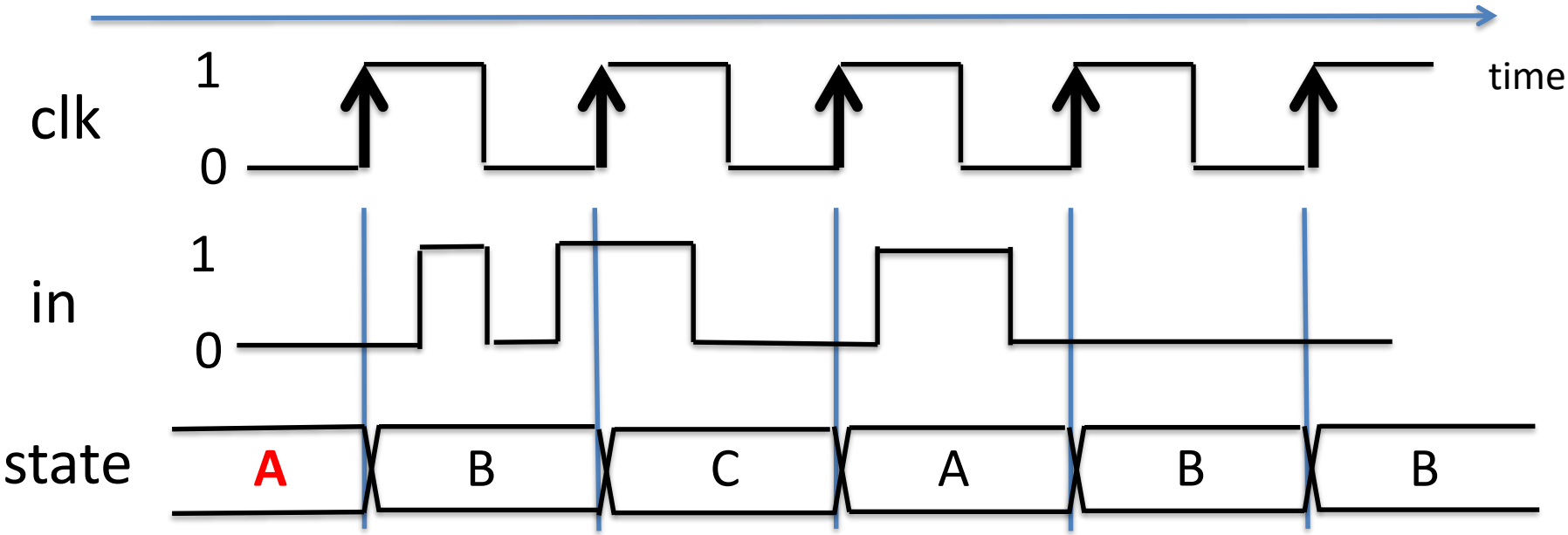
For a timing diagram, we need to choose values for the input signal “in”. Only after some choice for “in” we can derive the sequence of states from the state diagram.

Timing diagram. Example 2



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.

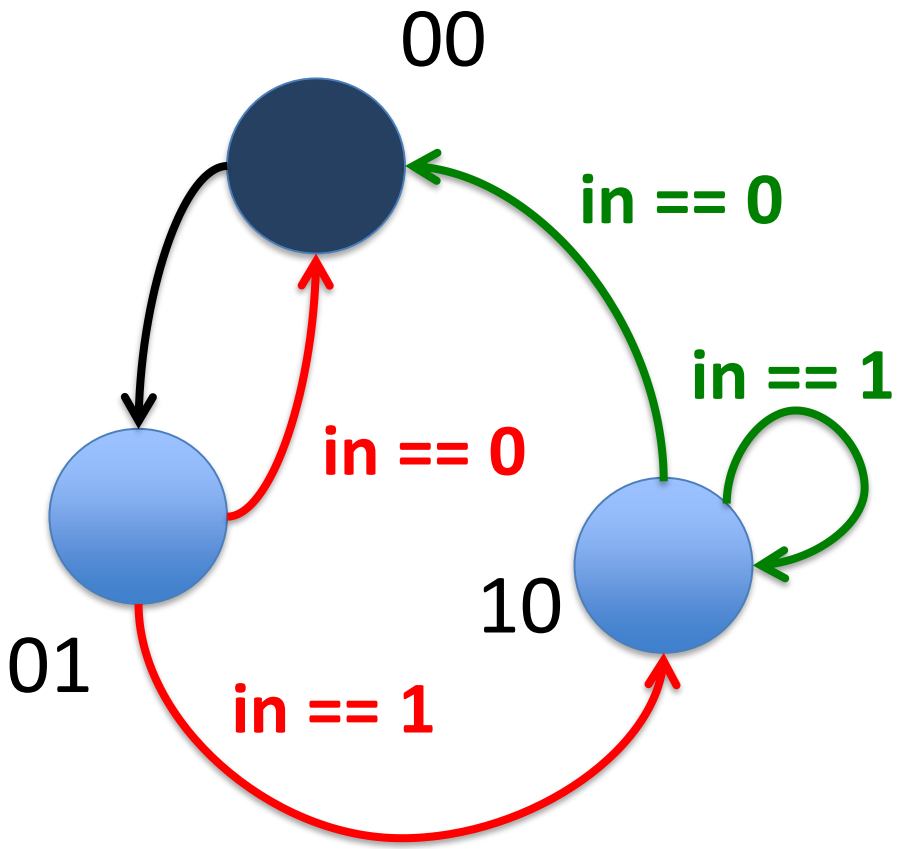
Timing diagram. Example 3



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.

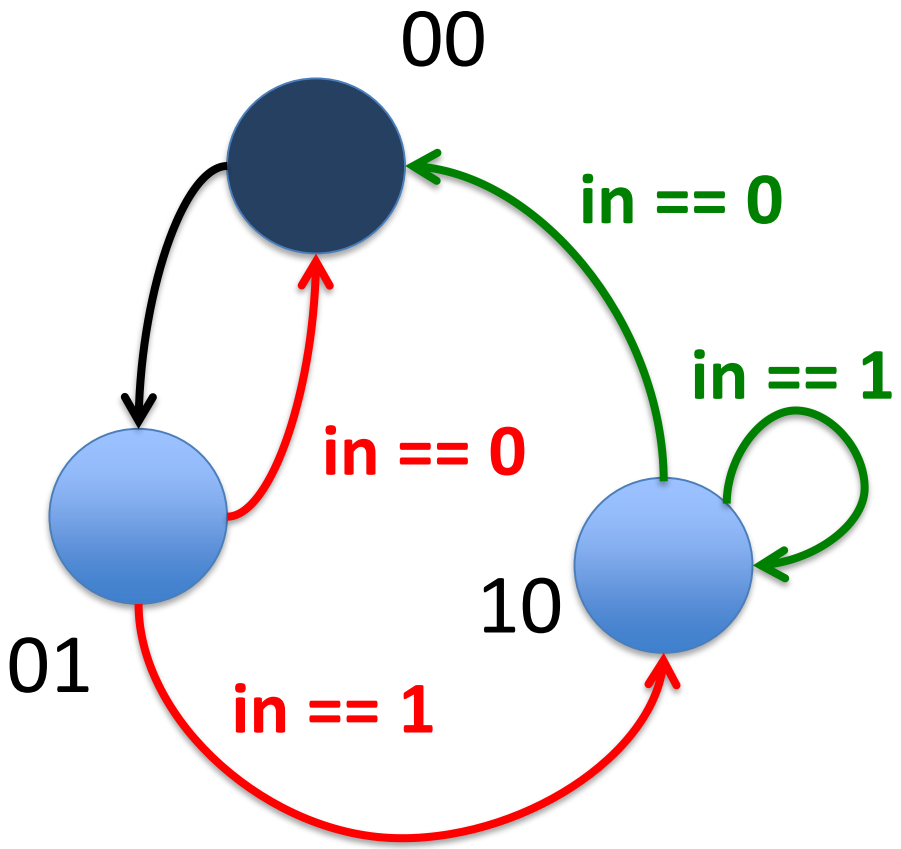
Binary state encoding: Instead of symbolic state names we use numbers

current 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



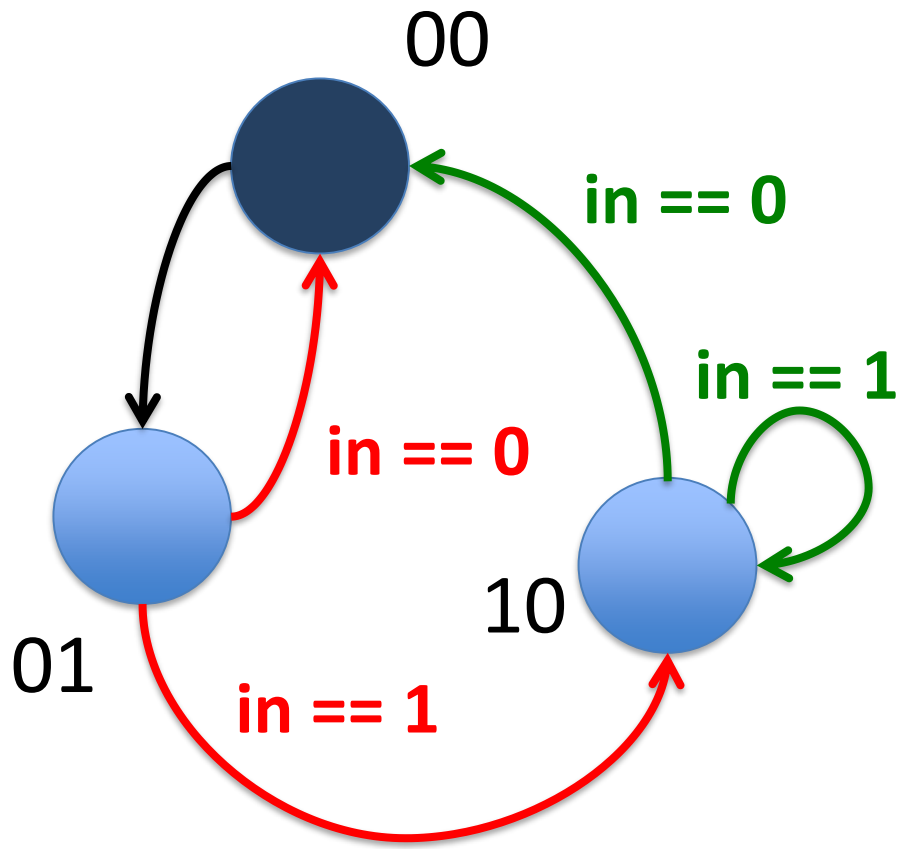
“11” does not exist: We use “Don’t Care” as the following state

current 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



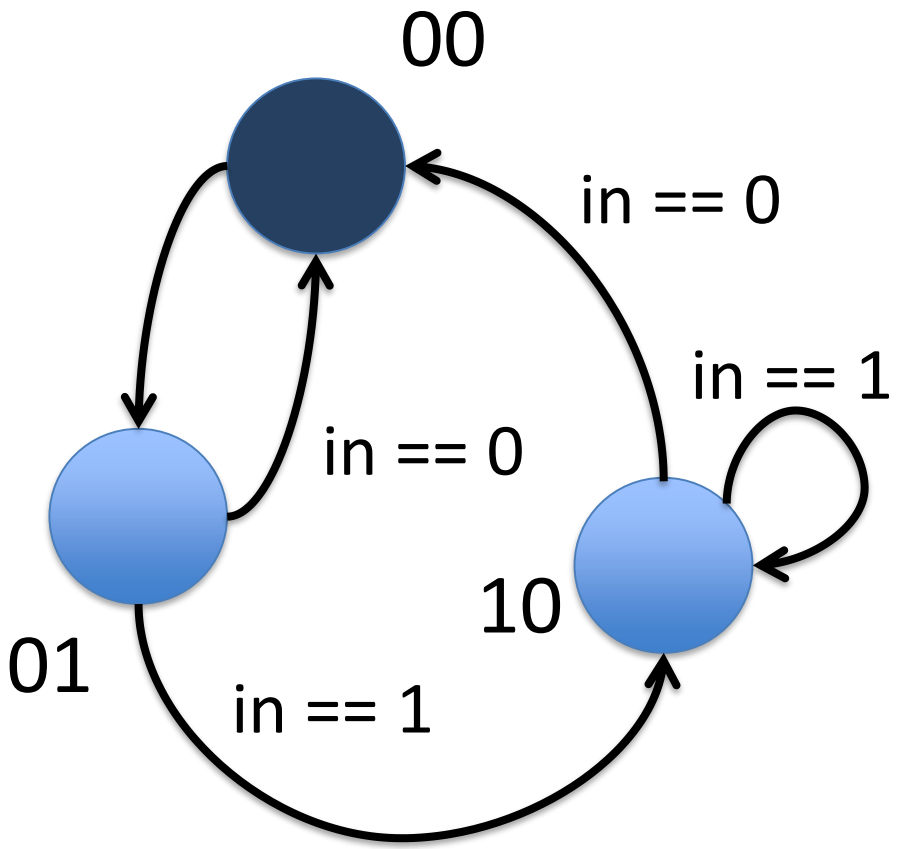
I have ordered the lines in the state transition table from 0 to 7. This makes it easier to “read” the table.

	current 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



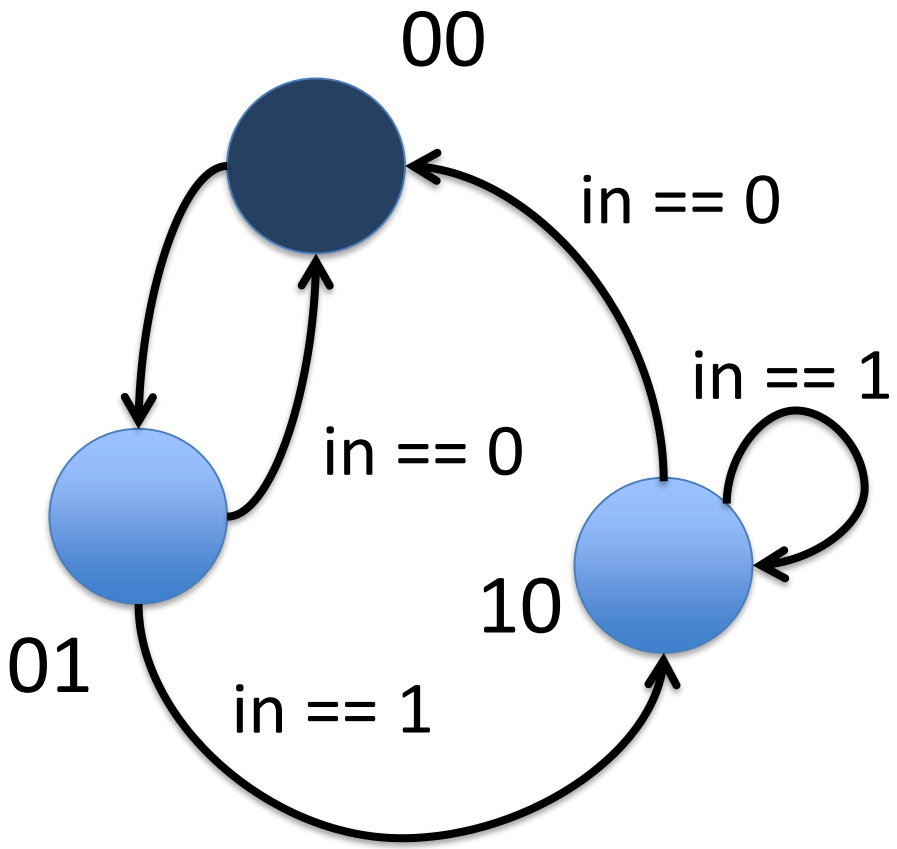
We call the state bits “s1” and “s0”

	current s1	current s0	in	next s1	next 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



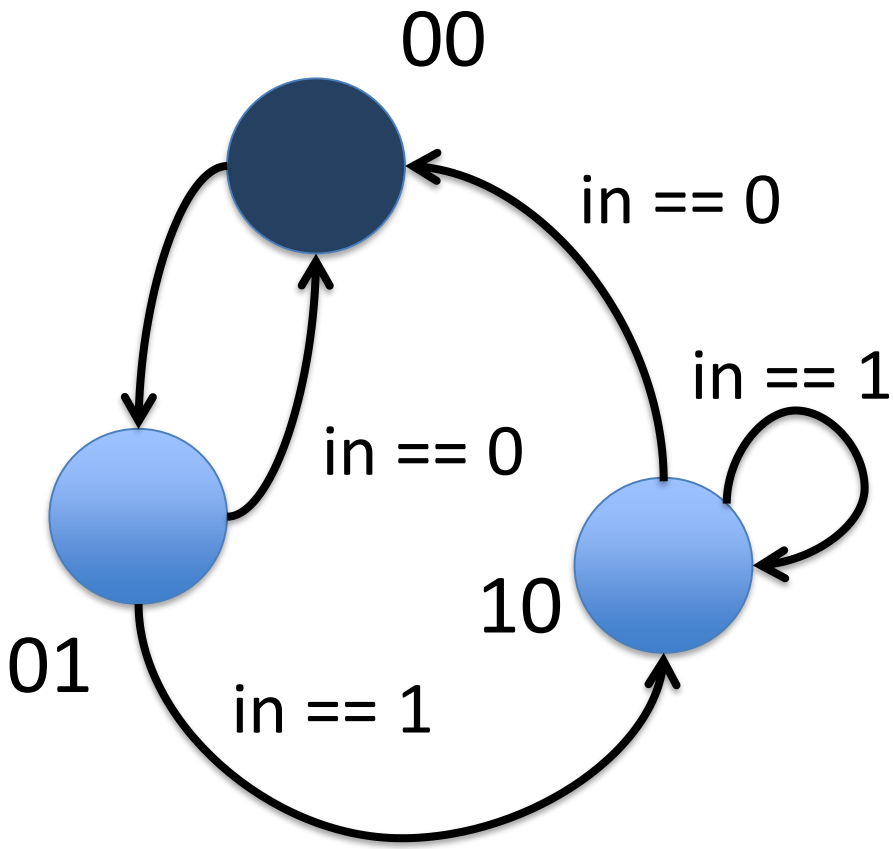
```
next s0 =   ( (~s1) & (~s0) & (~in))
          | ( (~s1) & (~s0) &    in )
```

	current s1	current s0	in	next s1	next 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

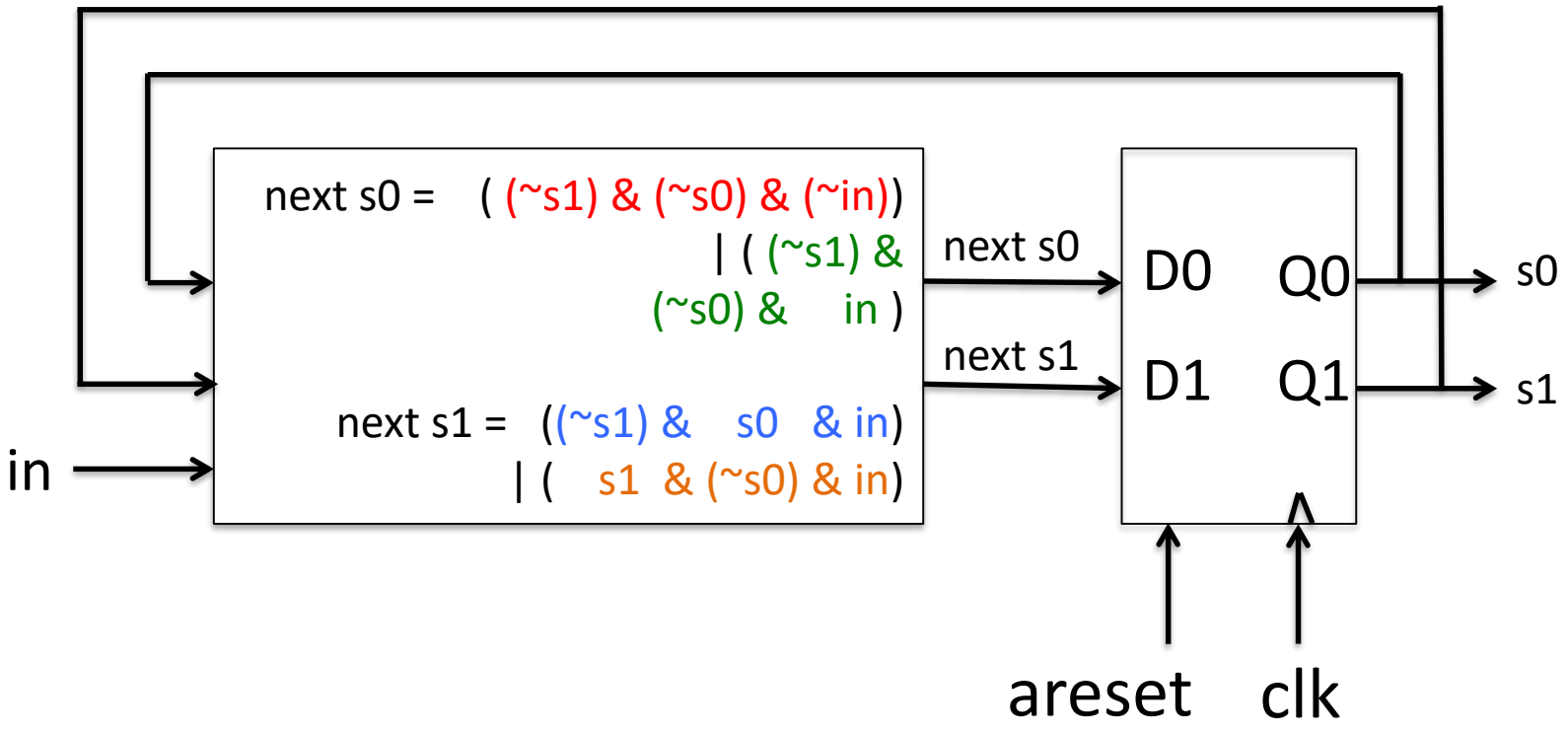


next s1 = ((~s1) & s0 & in)
| (s1 & (~s0) & in)

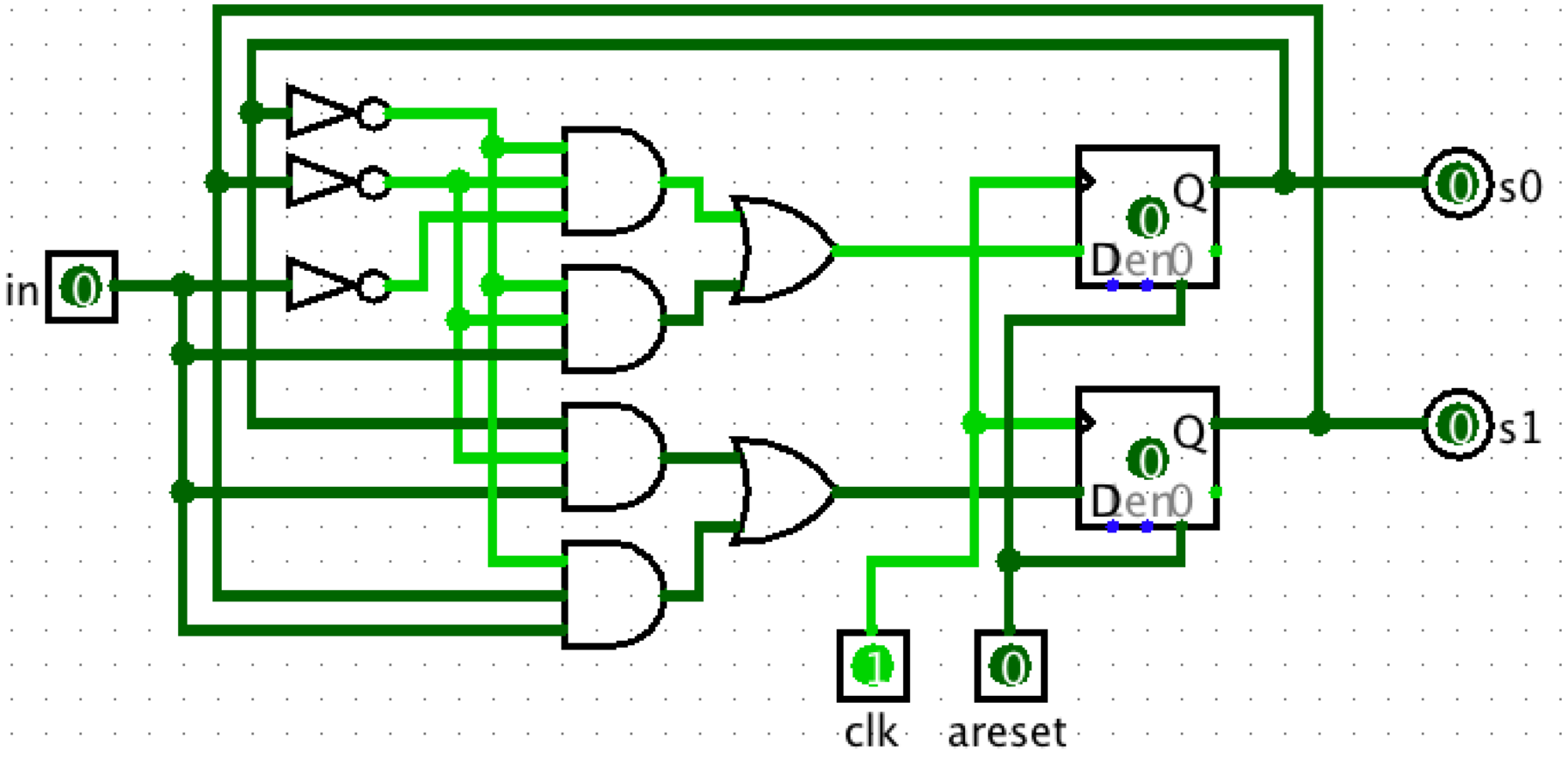
	current s1	current s0	in	next s1	next s0
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	0	0
3	0	1	1	1	0
4	1	0	0	0	0
5	1	0	1	1	0
6	1	1	0	0	0
7	1	1	1	0	0



Structural diagram of the FSM



Implementation with Logisim



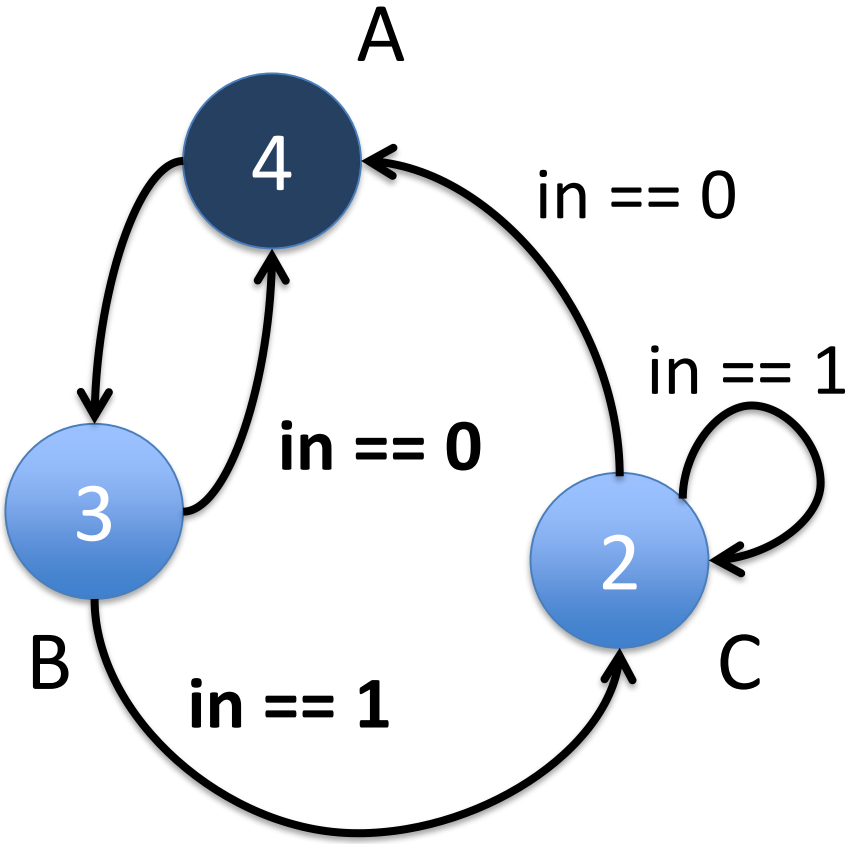
Outputs

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)

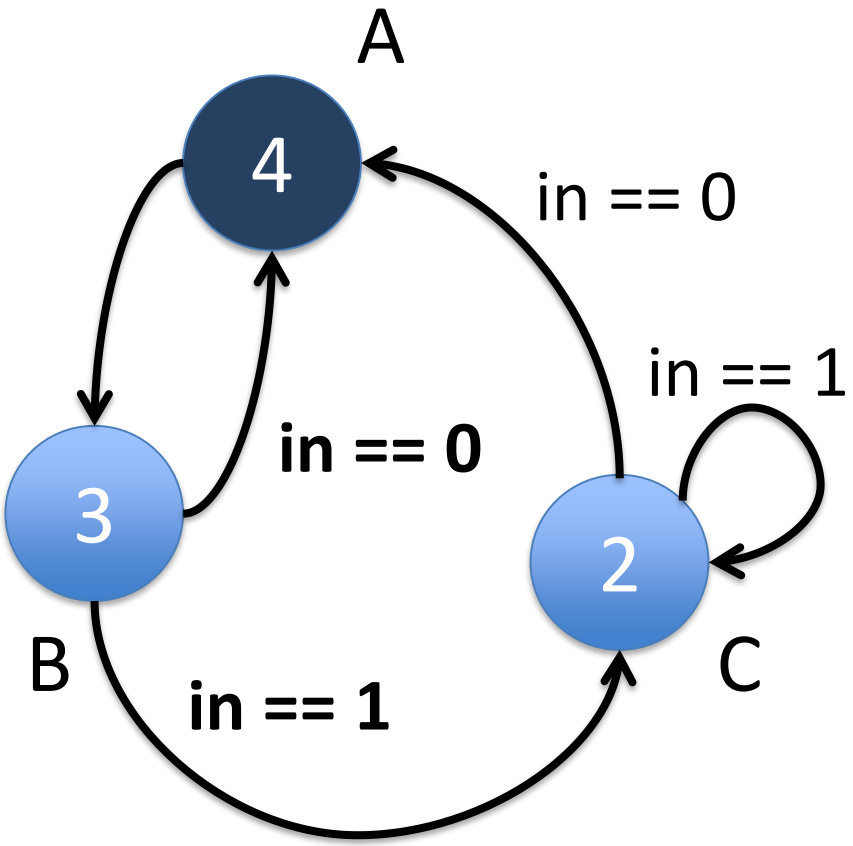


We define the outputs with the "output function"

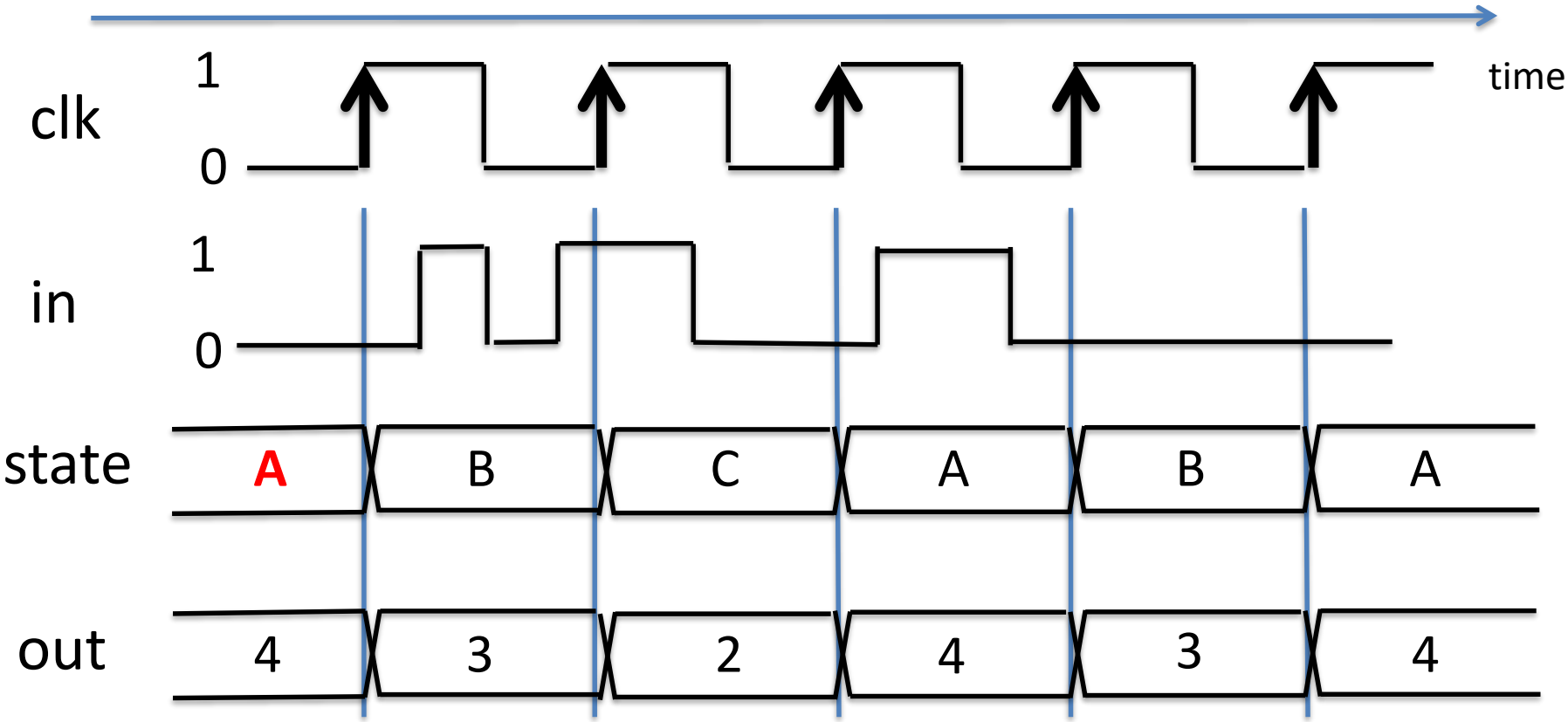
Moore machines:

output = f(state)

state	output
A	4
B	3
C	2



Timing diagram. Example 3

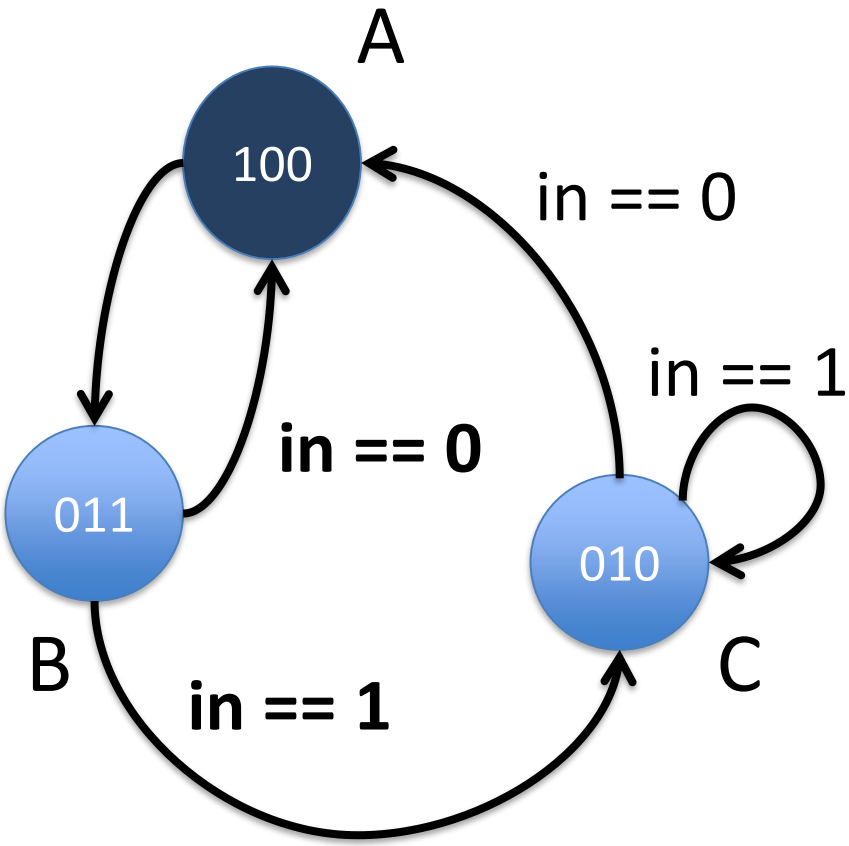


For a timing diagram, we need to choose values for the input signal “in”. Only after some choice for “in” we can derive the sequence of states from the state diagram.

We define the outputs with binary values

Moore machines:

output = f(state)



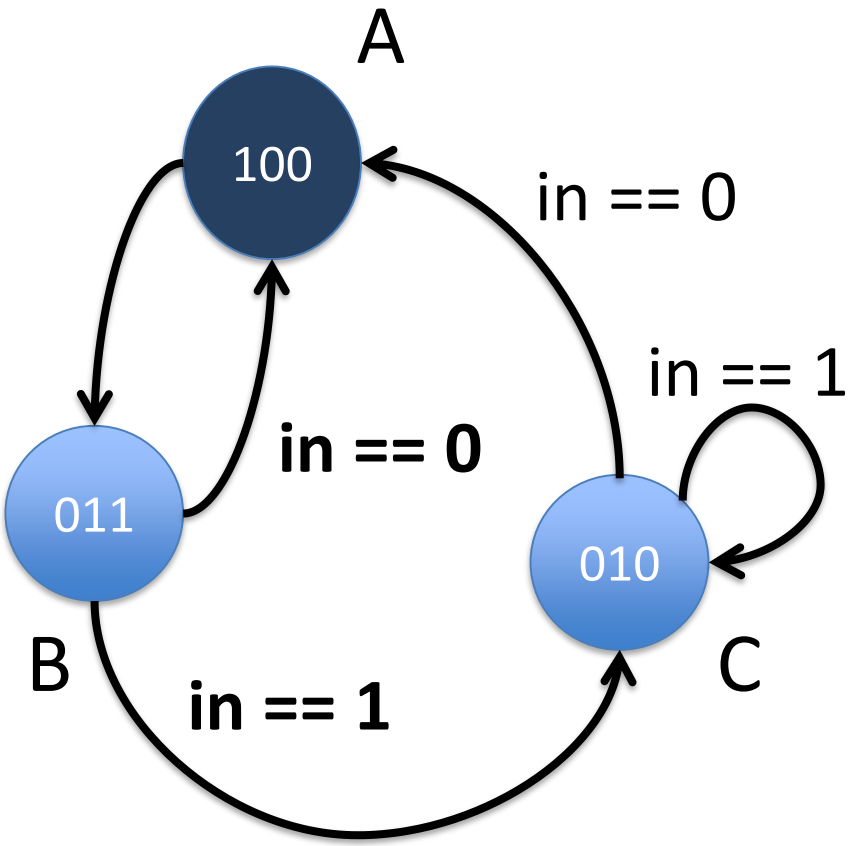
state	output	
A	100	(=4)
B	011	(=3)
C	010	(=2)

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

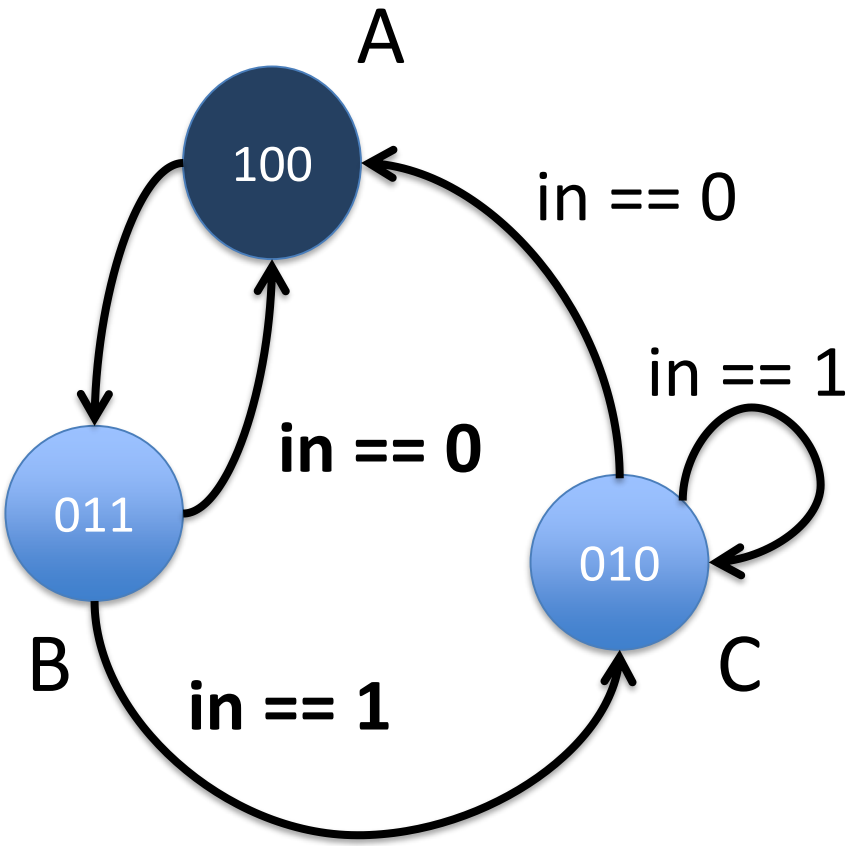


For a Verilog example of this FSM see

chapter_03/example_00/01_fsm_moore_with_output_function.v

Binary state encoding: We define the outputs with binary values

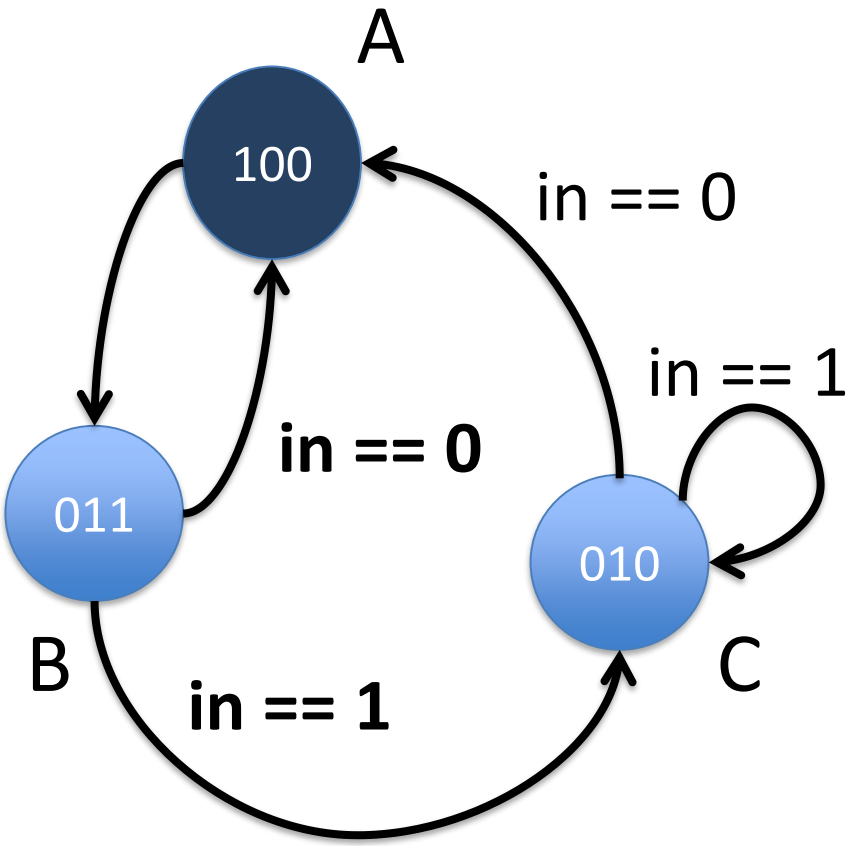
state		o2	o1	o0
0	0	1	0	0
0	1	0	1	1
1	0	0	1	0



We define the outputs with binary values

o2 = ~s1 & ~s0

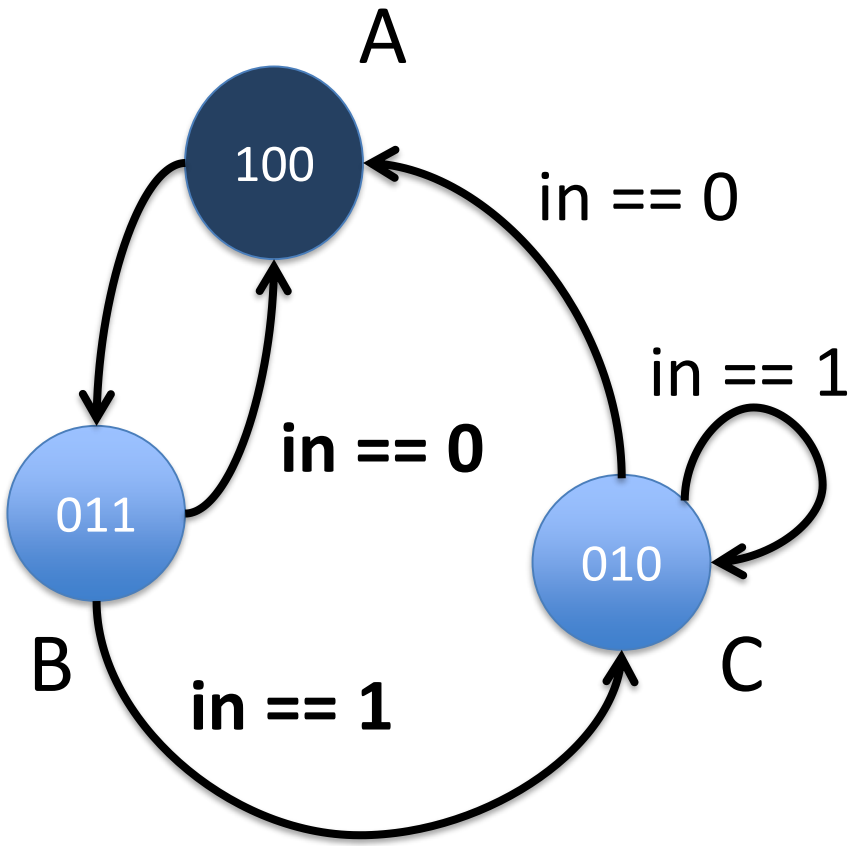
s1	s0	**o2**	o1	o0
0	**0**	**1**	0	0
0	1	0	1	1
1	0	0	1	0



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

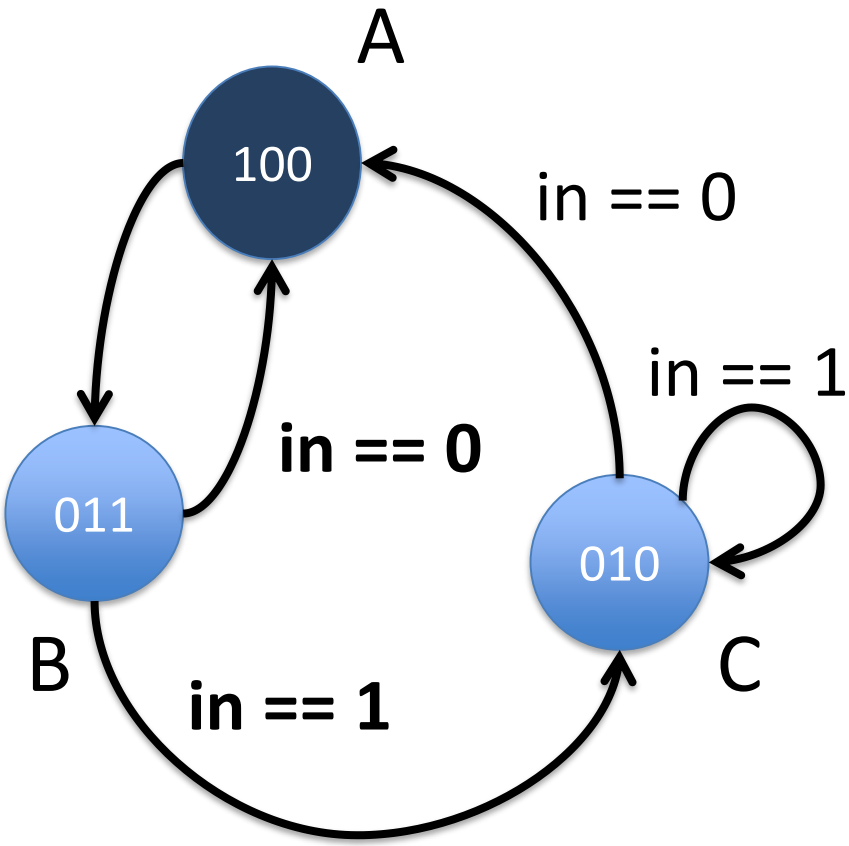
We define the outputs with binary values

o2 = ~s1 & ~s0

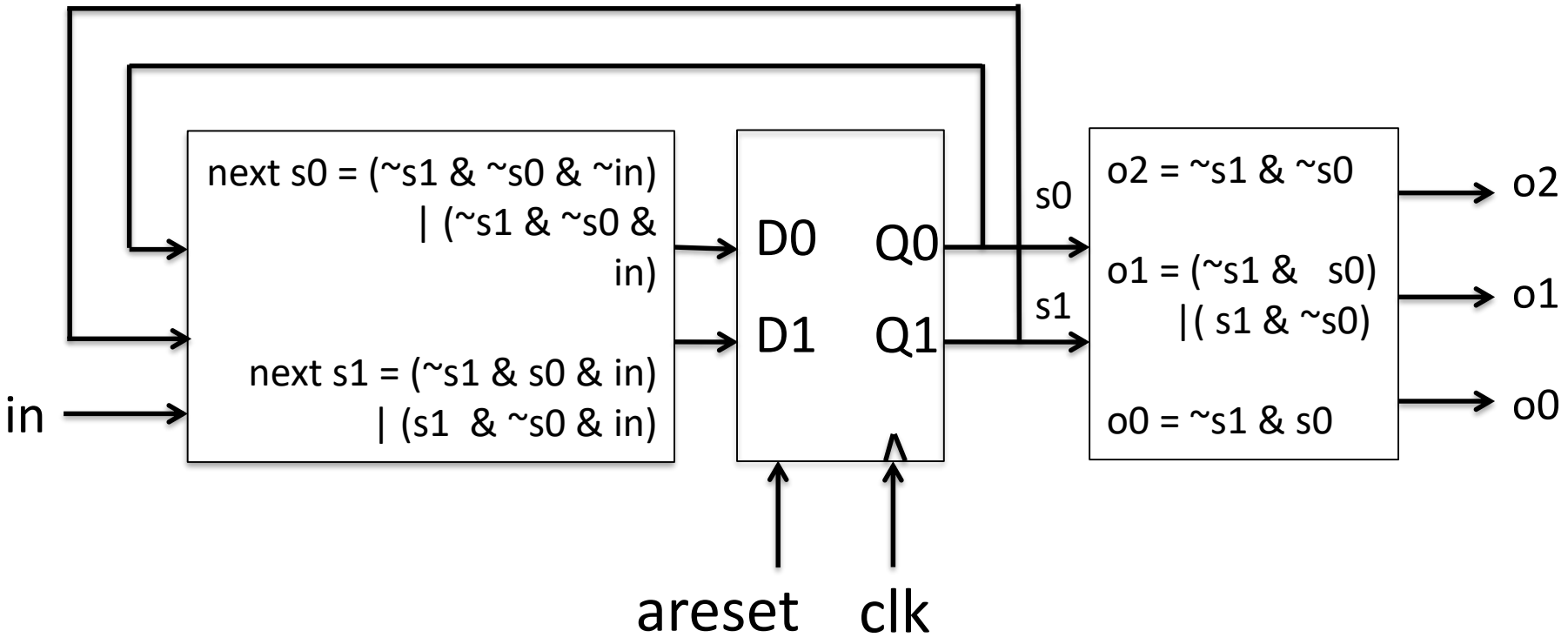
o1 = (~s1 & s0) | (s1 & ~s0)

o0 = ~s1 & s0

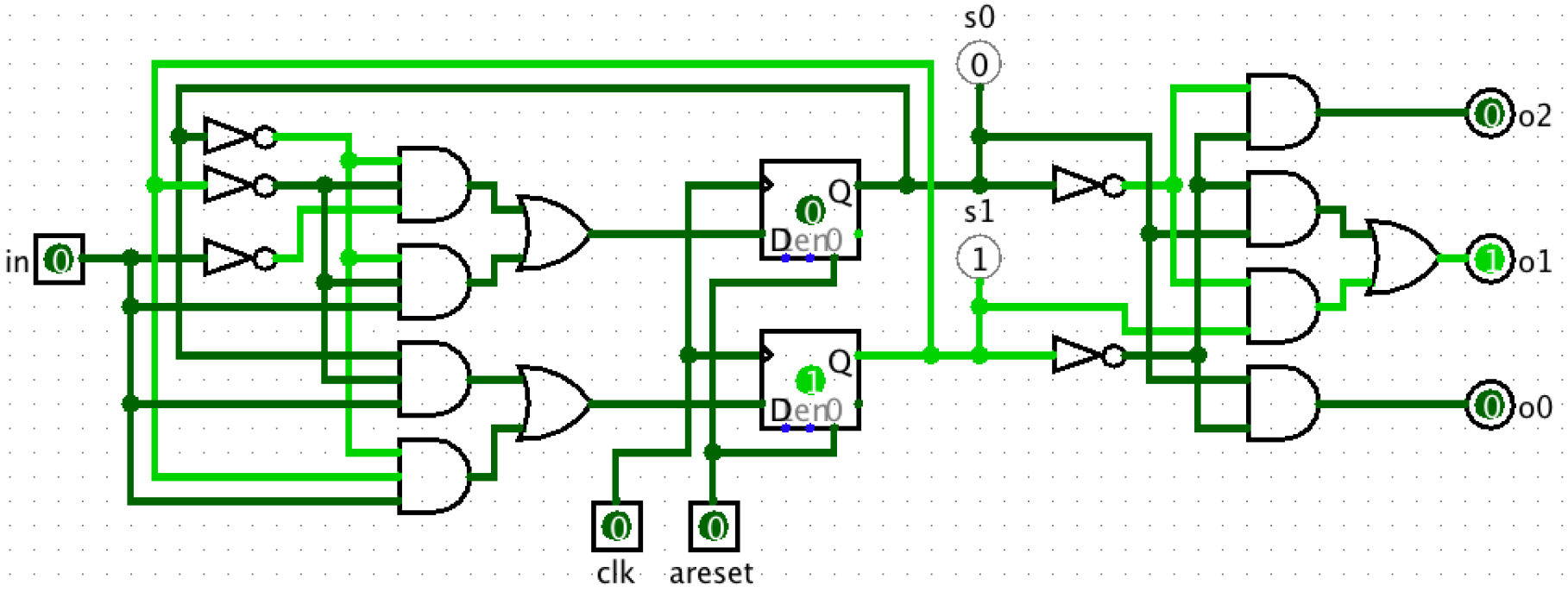
s1	s0	o2	o1	o0
0	0	1	0	0
0	**1**	0	1	**1**
1	0	0	1	0



Structural diagram of the FSM

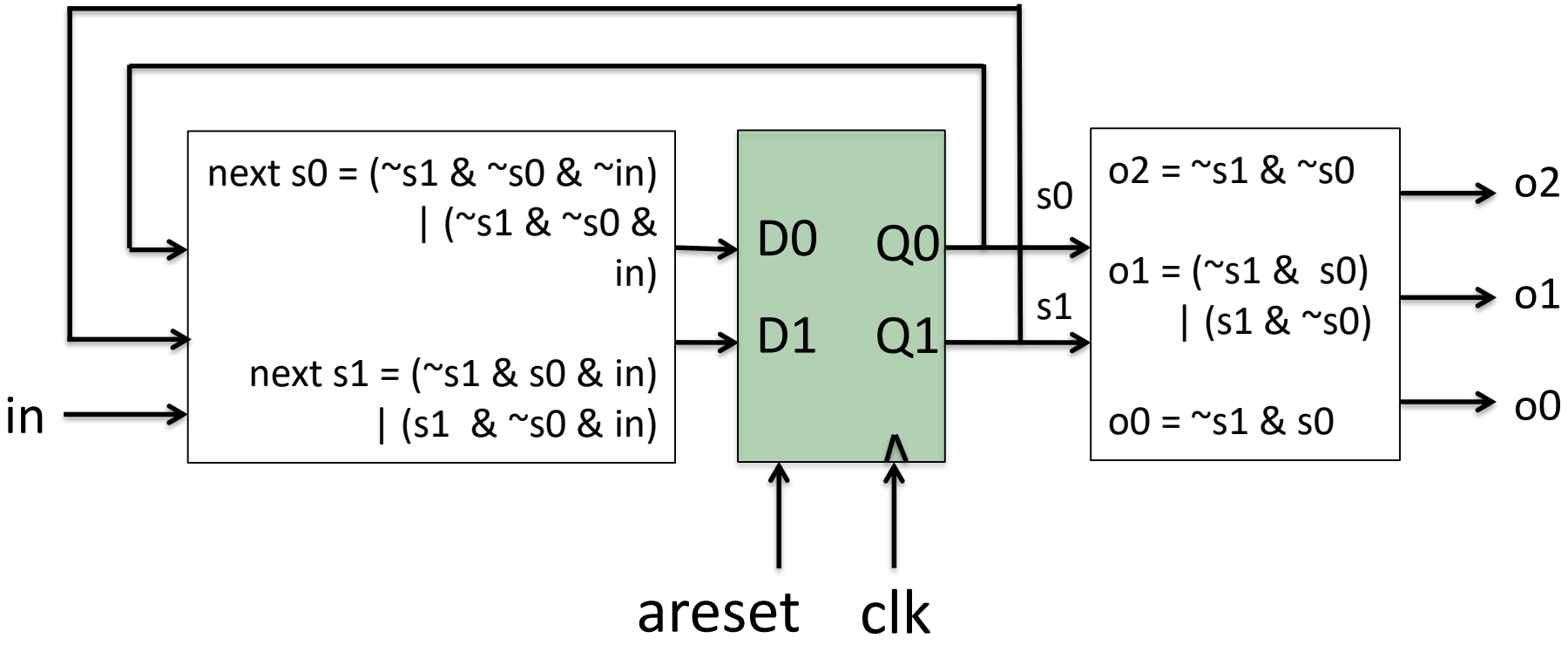


Implementation with Logisim



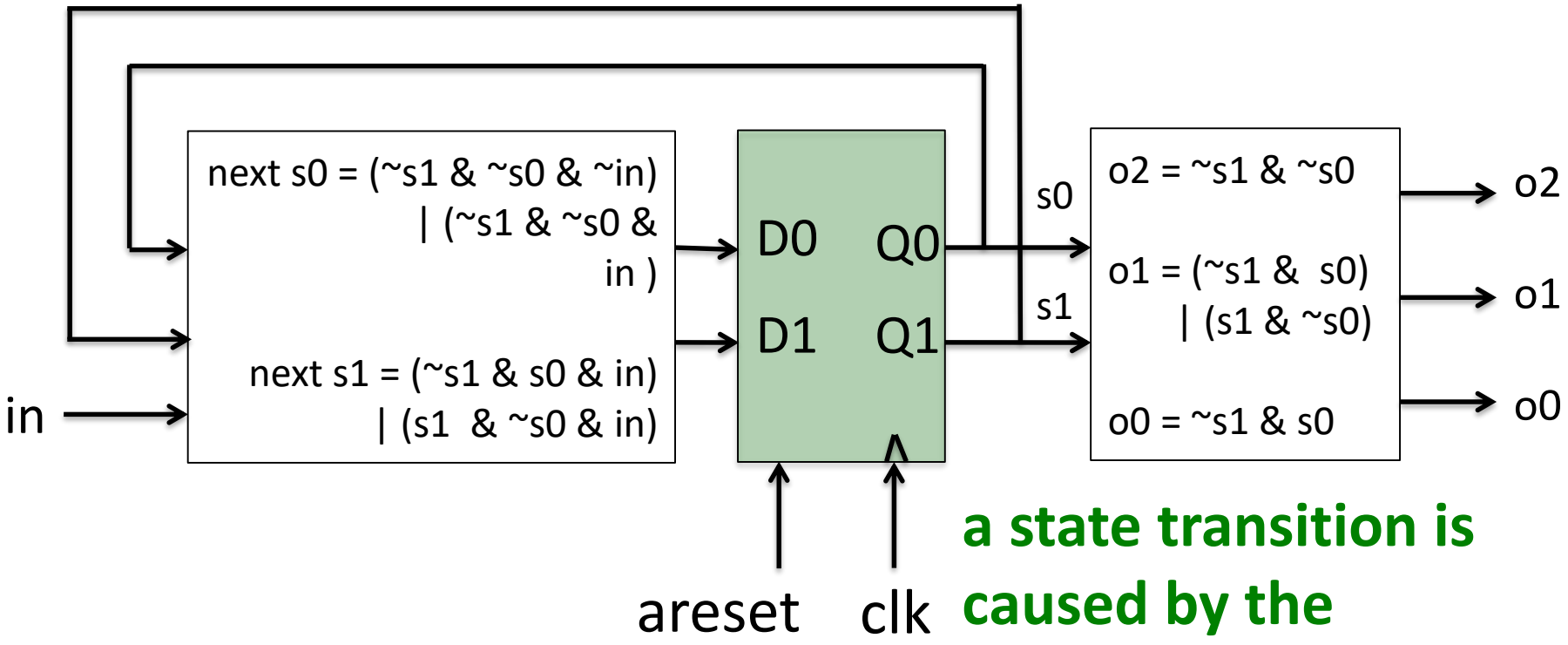
Essence of Moore Machines

the state is stored in a register



Essence of Moore Machines

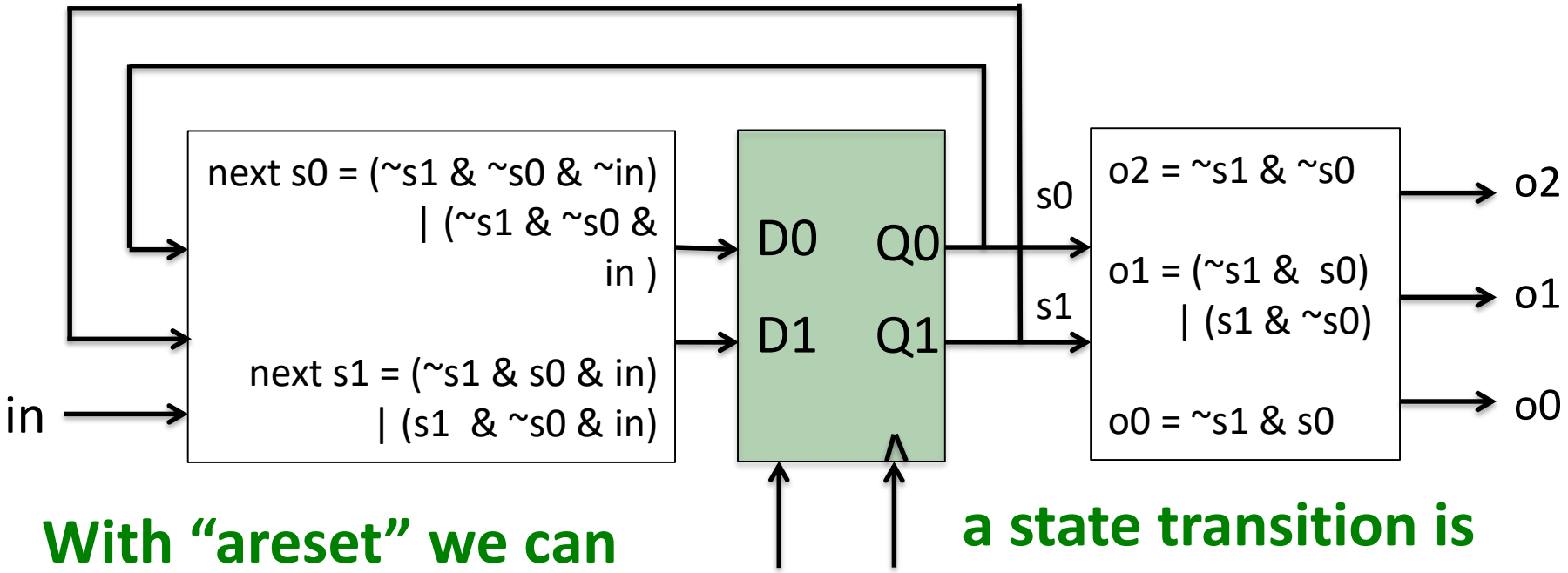
the state is stored in a register



a state transition is caused by the clock signal.

Essence of Moore Machines

the state is stored in a register

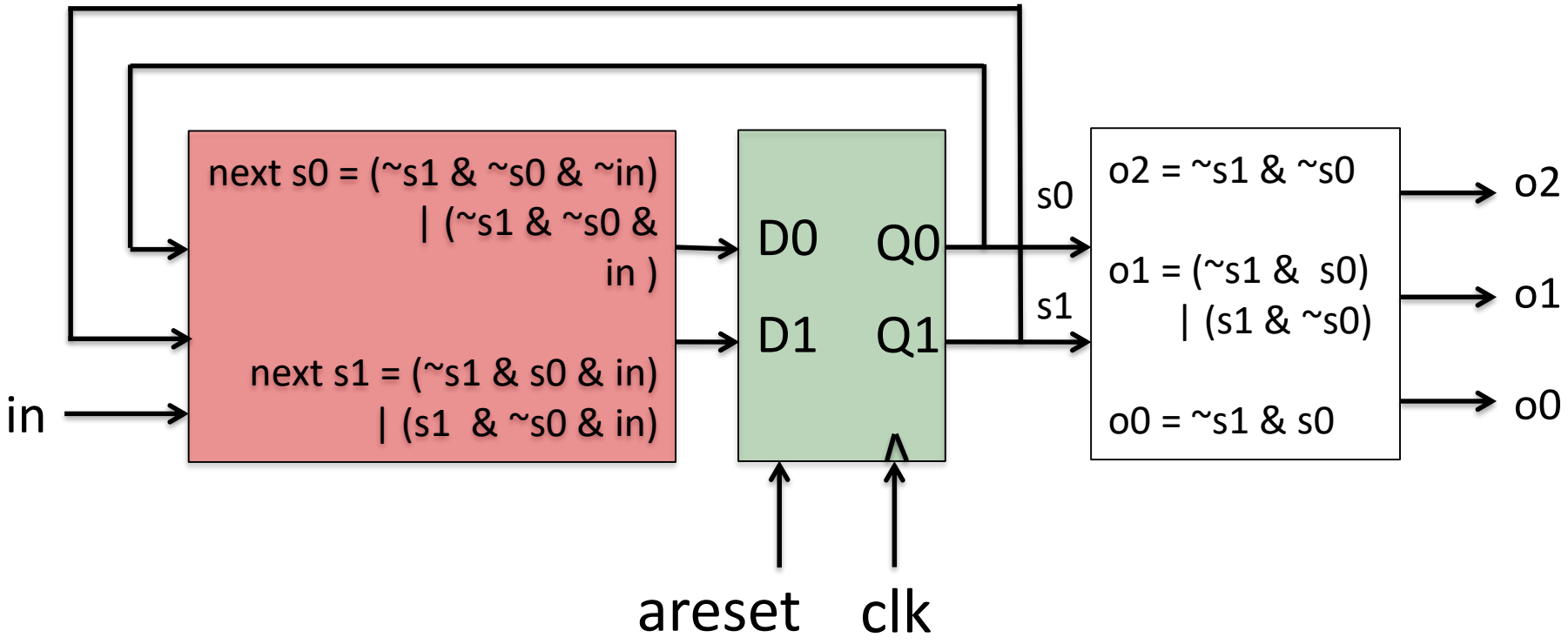


With "areset" we can initialize the ASM ("initial state").

a state transition is caused by the clock signal.

Essence of Moore Machines

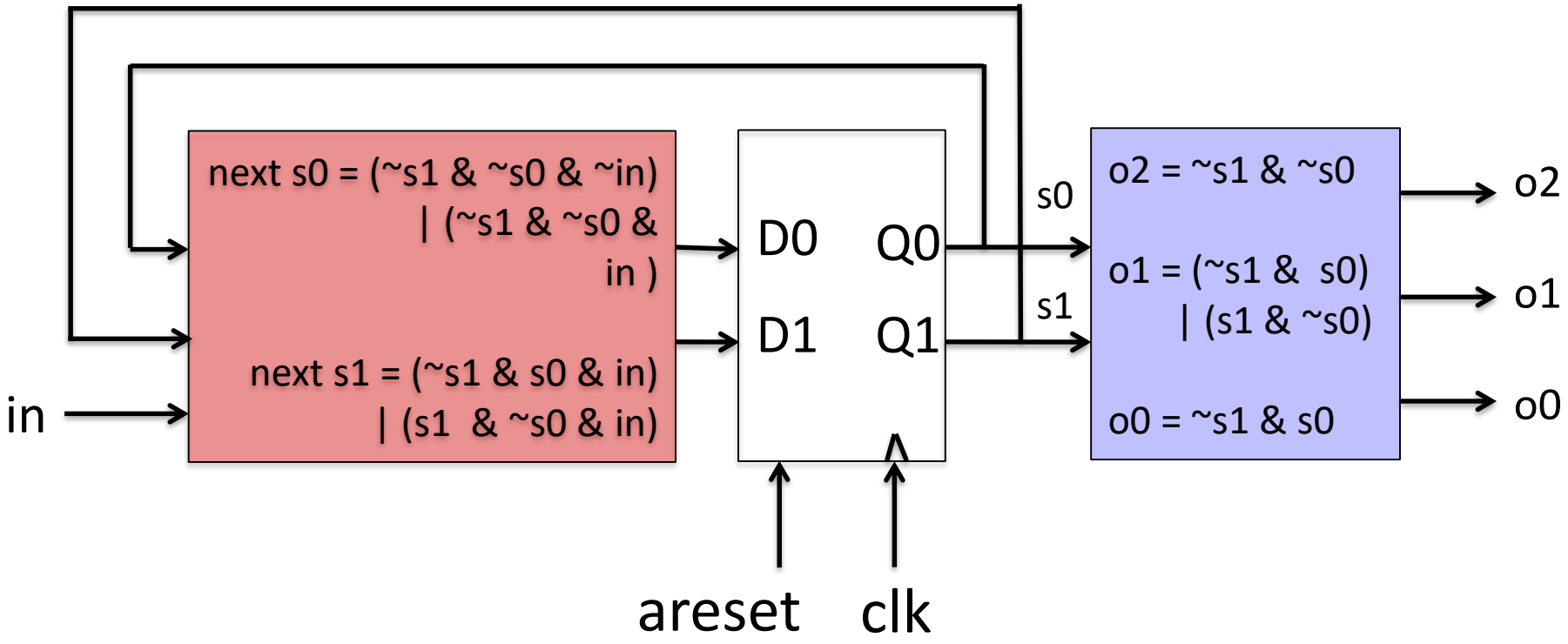
With the next-state function f
we compute the next state:
next state = f(state, input)



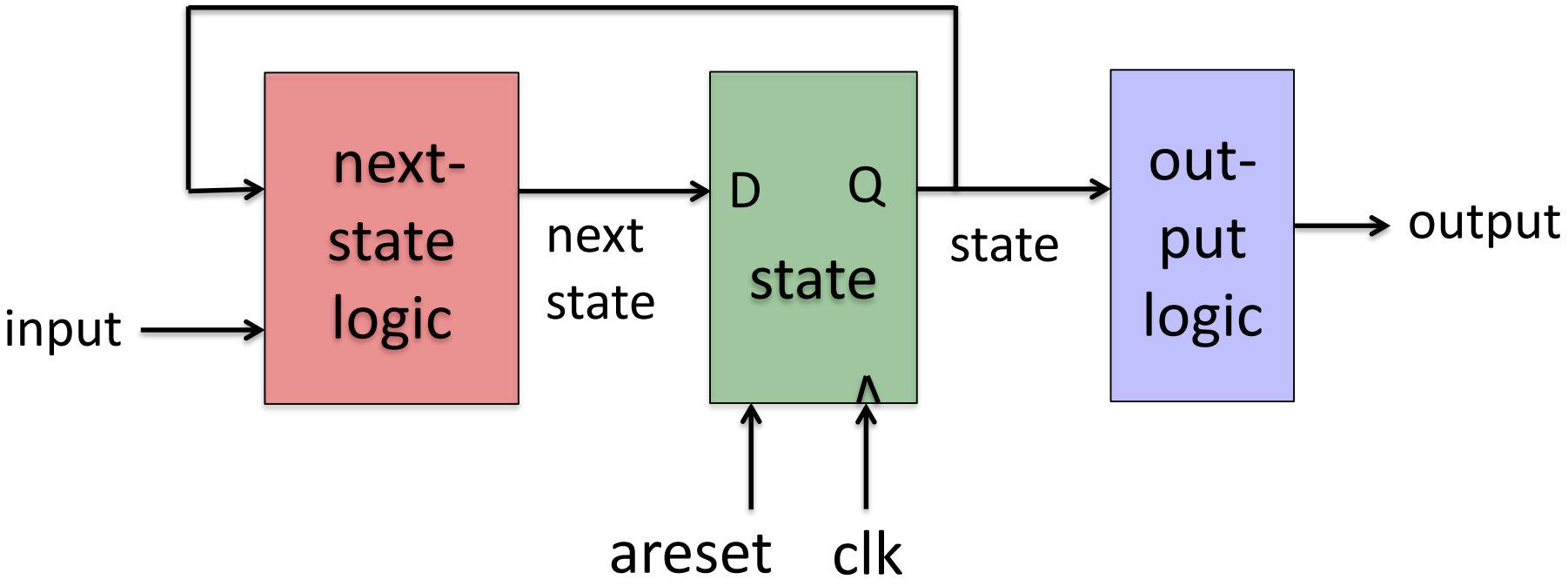
Essence of Moore Machines

With the output function we compute the output values:

output = g(state)



Essence of Moore Machines

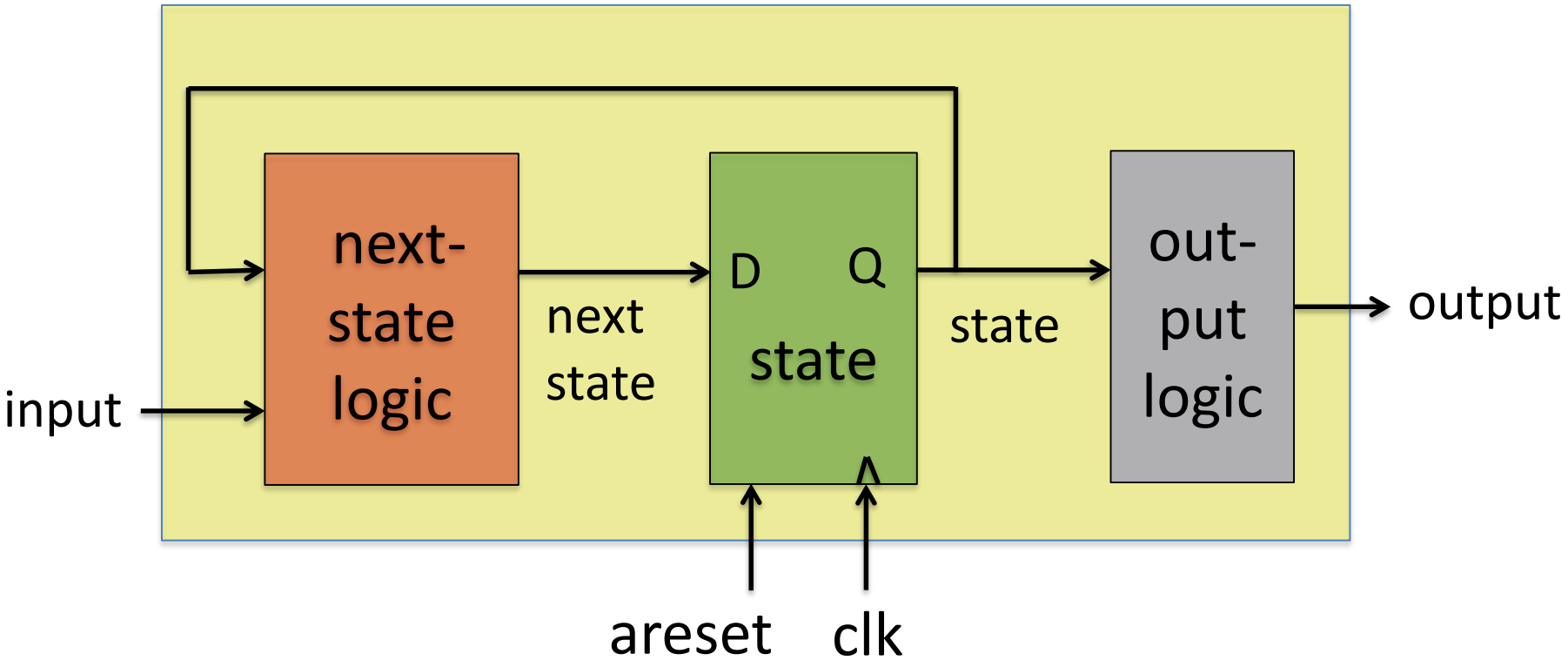


There exist 2 types of machines: check out the LITTLE but IMPORTANT difference

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

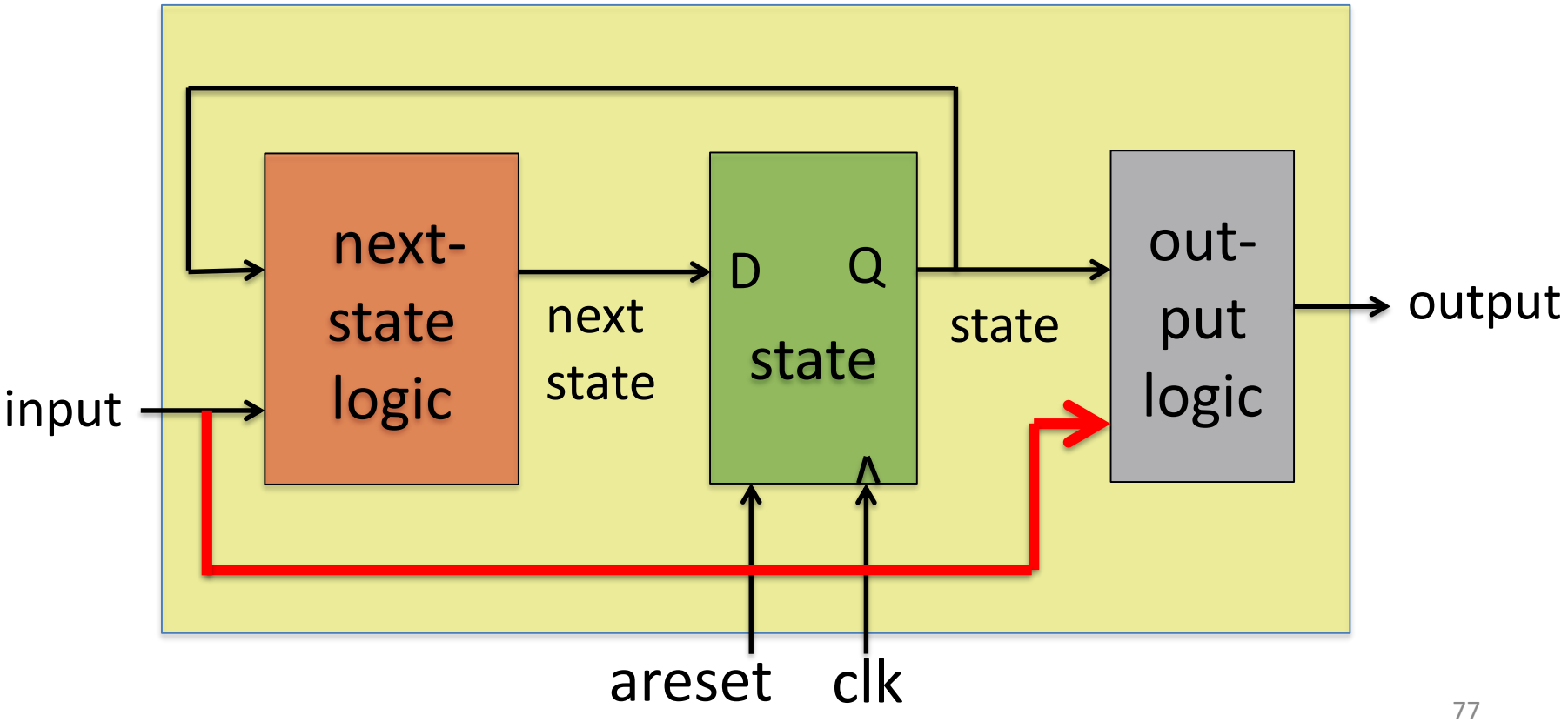
- Mealy Machines
  - next state = function of state and input
  - output = function of state **and input**

Essence of **Moore** Machines



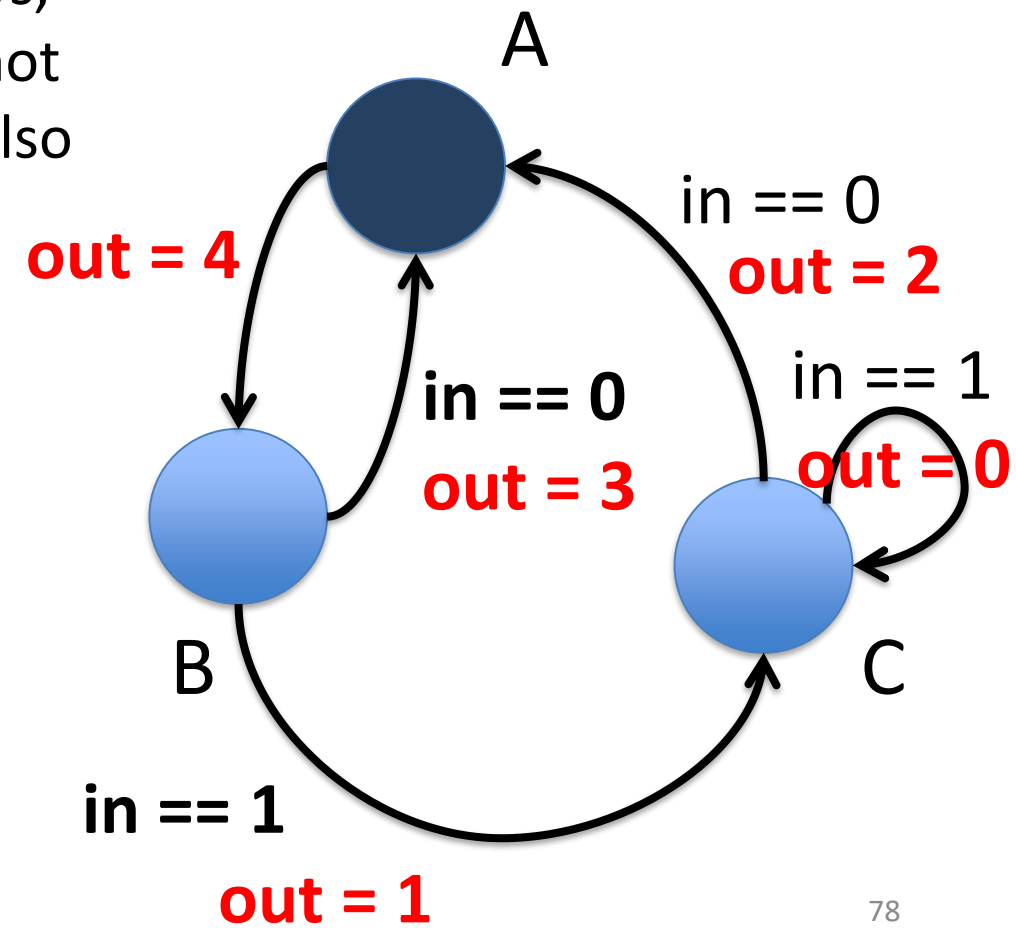
Essence of **Mealy** Machines

output = g(state, input)



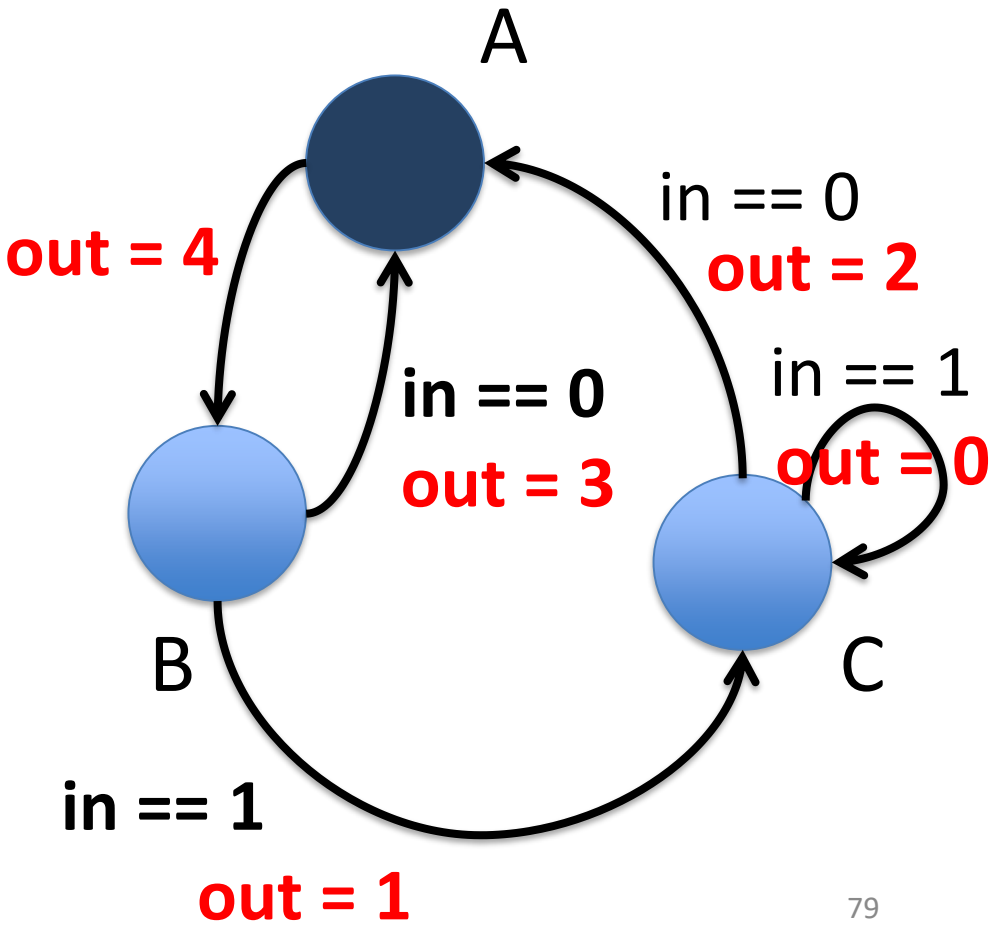
An example for a Mealy Machine

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 state transitions are the same as in the previous example with the Moore Machine

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

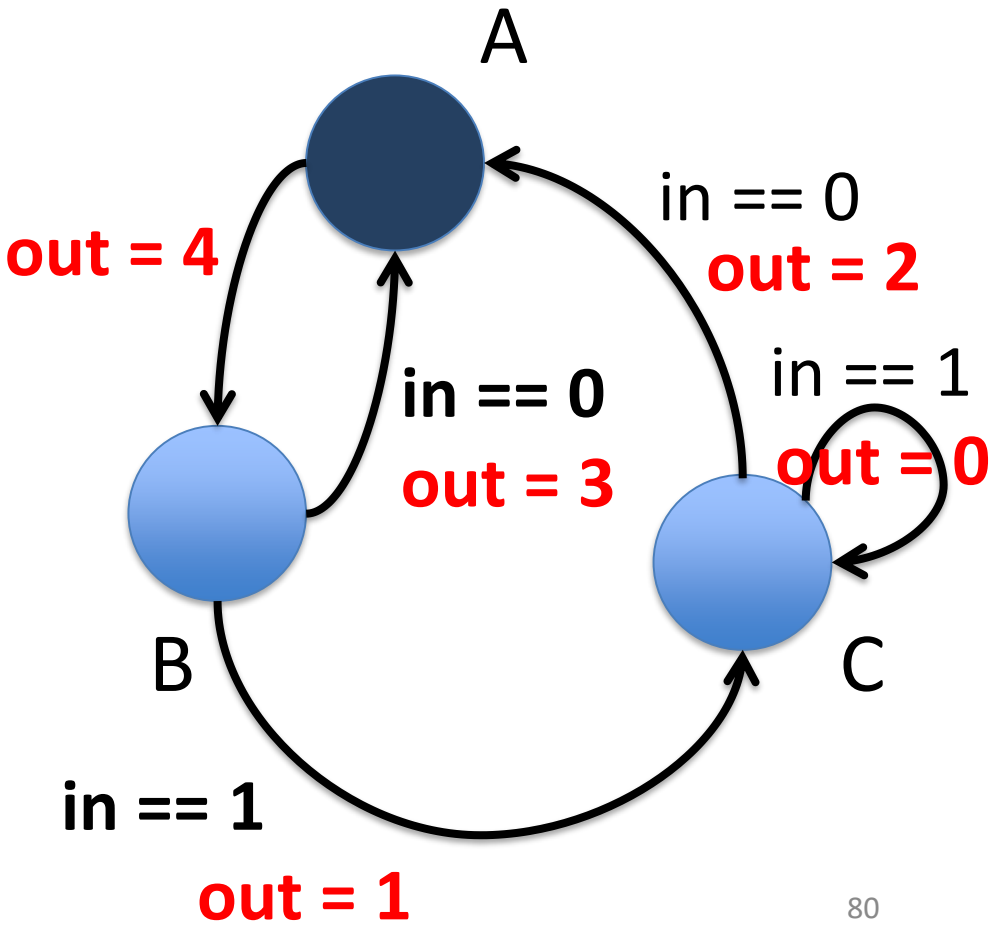


The output function

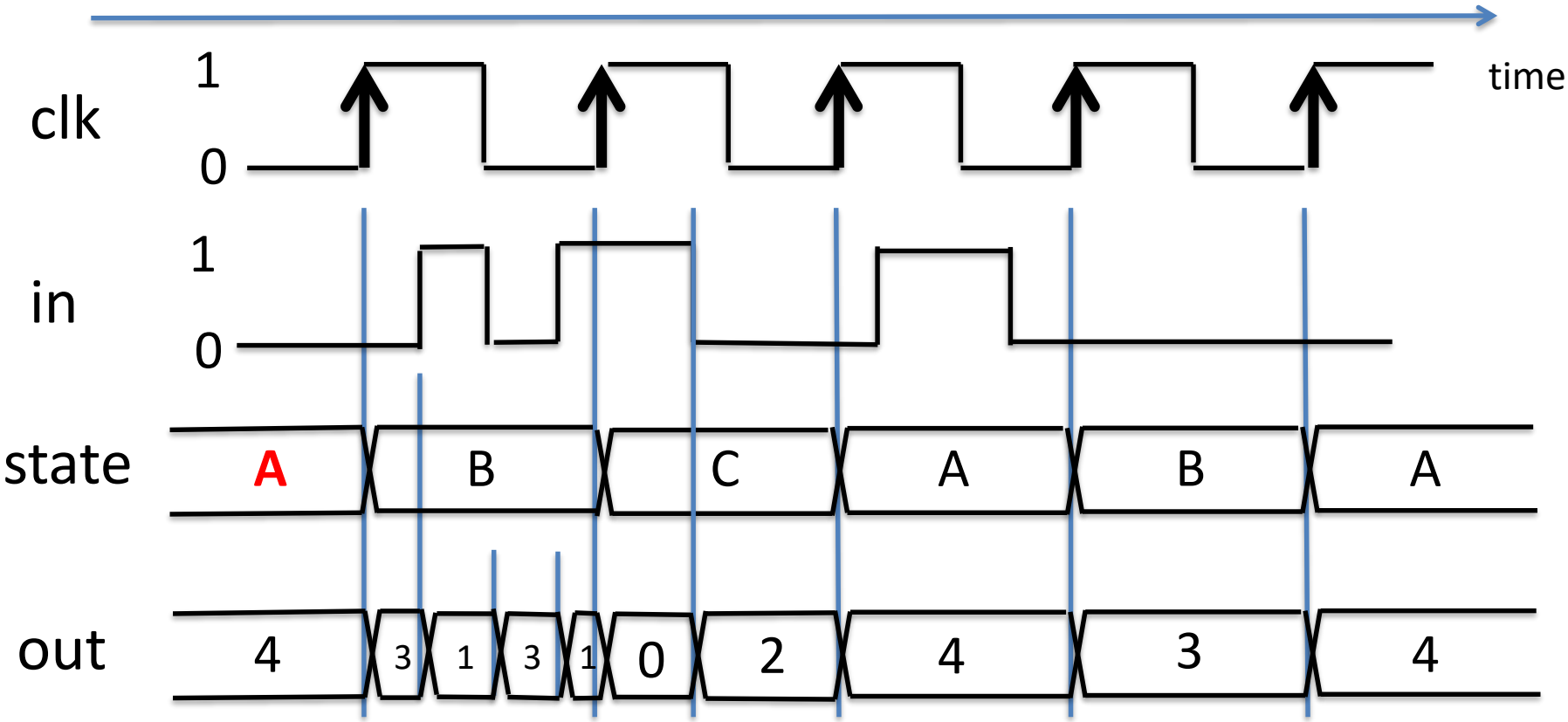
The output function can be derived from the state diagram.

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”.

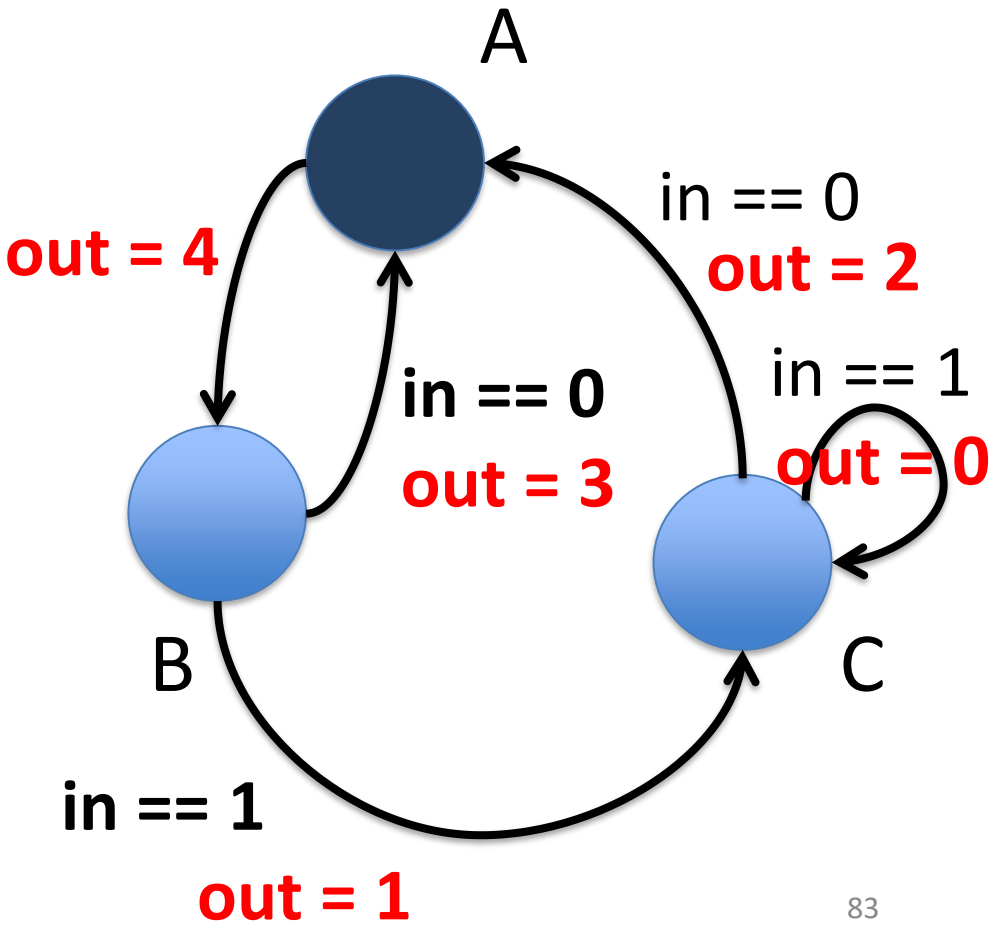
For a Verilog example of this FSM see

chapter_03/example_00/04_fsm_mealy.v

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



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

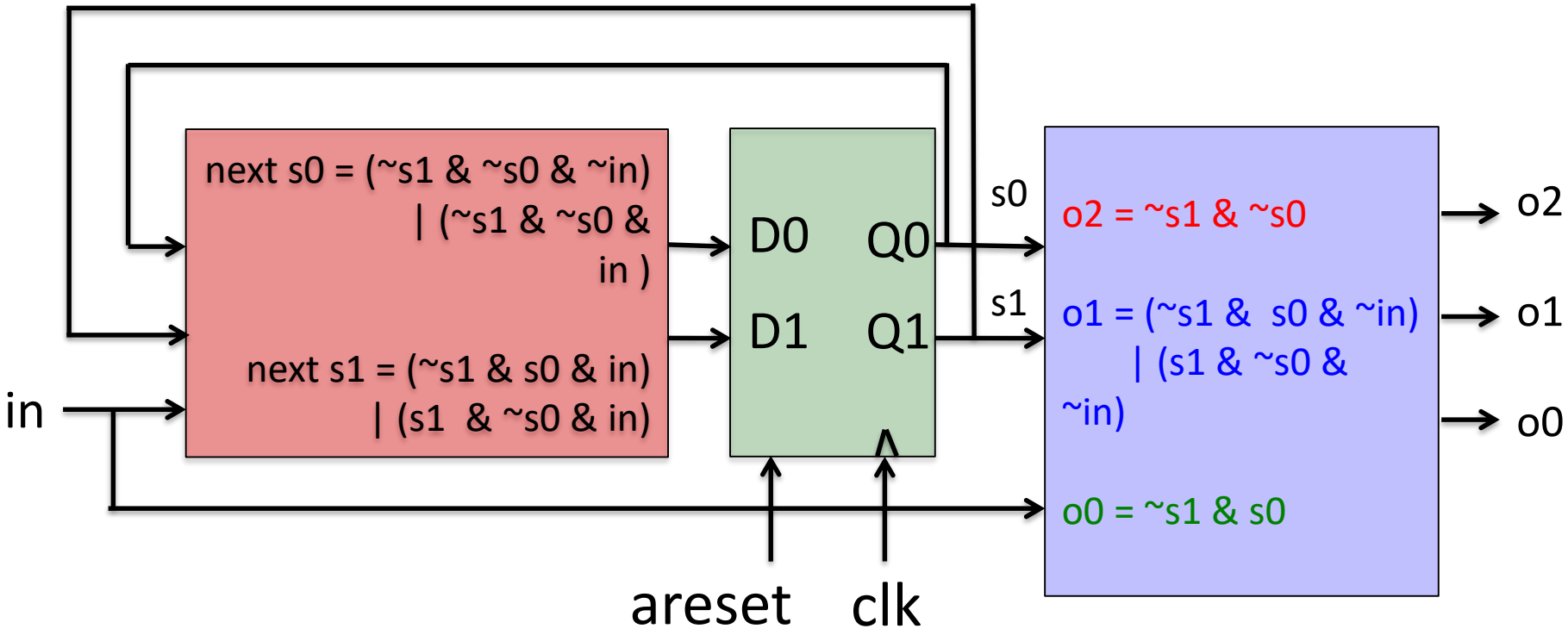
s1	s0	in	o2	o1	o0
0	0	0	**1**	0	0
0	0	1	**1**	0	0
0	1	0	0	**1**	**1**
0	1	1	0	0	**1**
1	0	0	0	**1**	0
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

o2 = ~s1 & ~s0

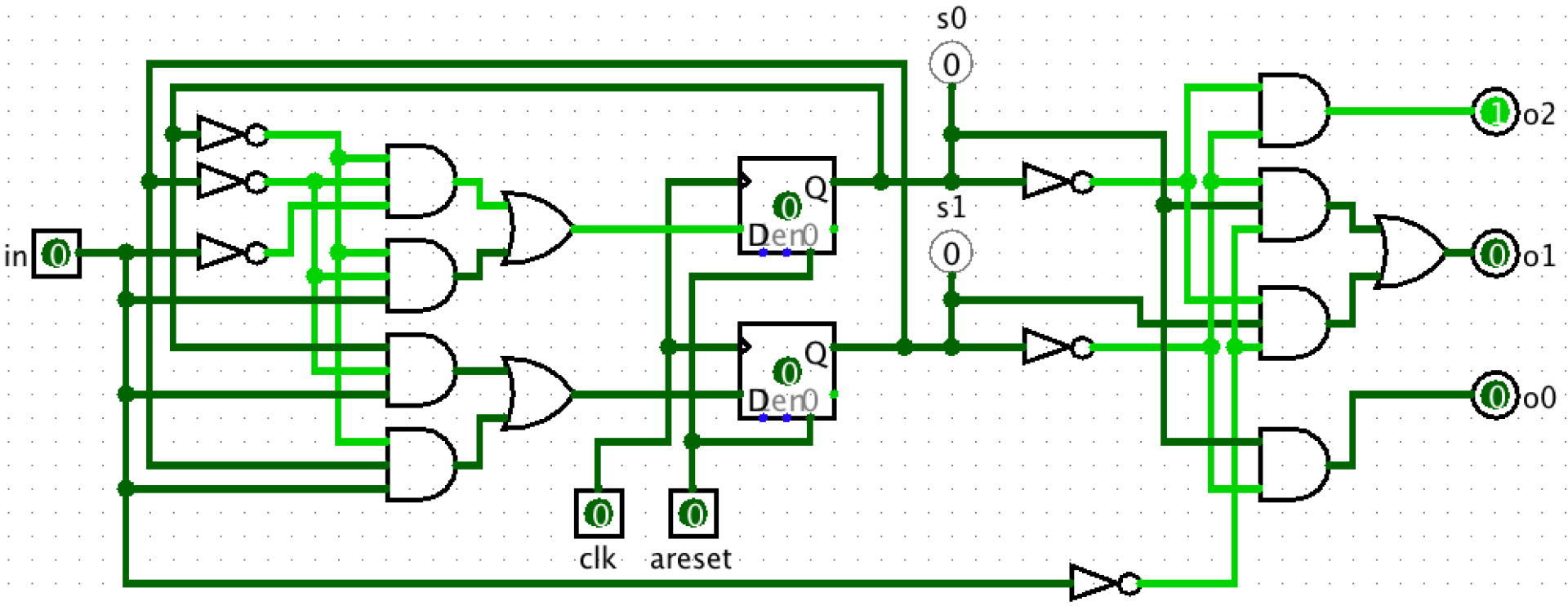
o1 = (~s1 & s0 & ~in)
| (s1 & ~s0 & ~in)

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
o2
o1
o0
in
areset
clk

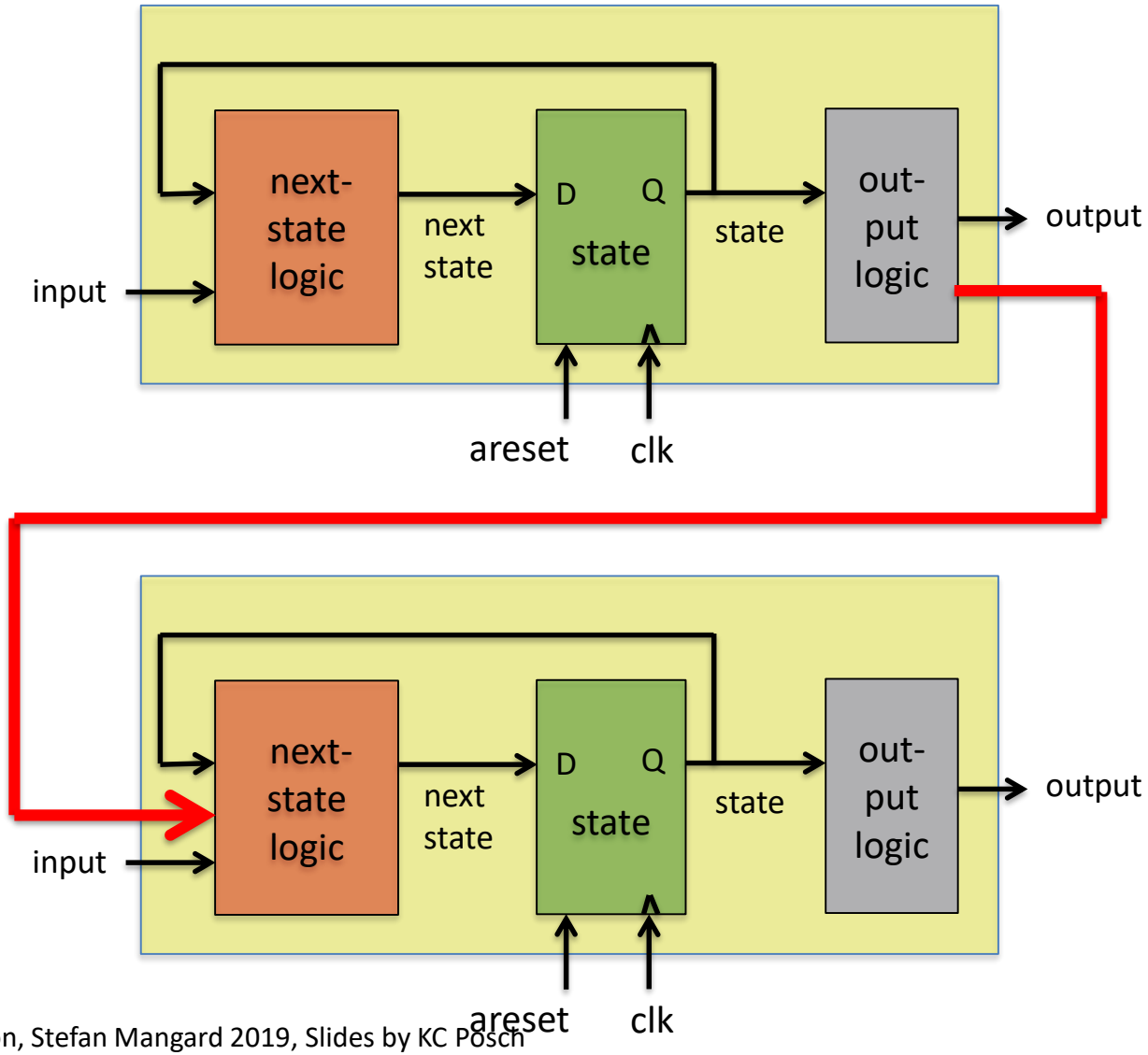
Modeling with Logisim



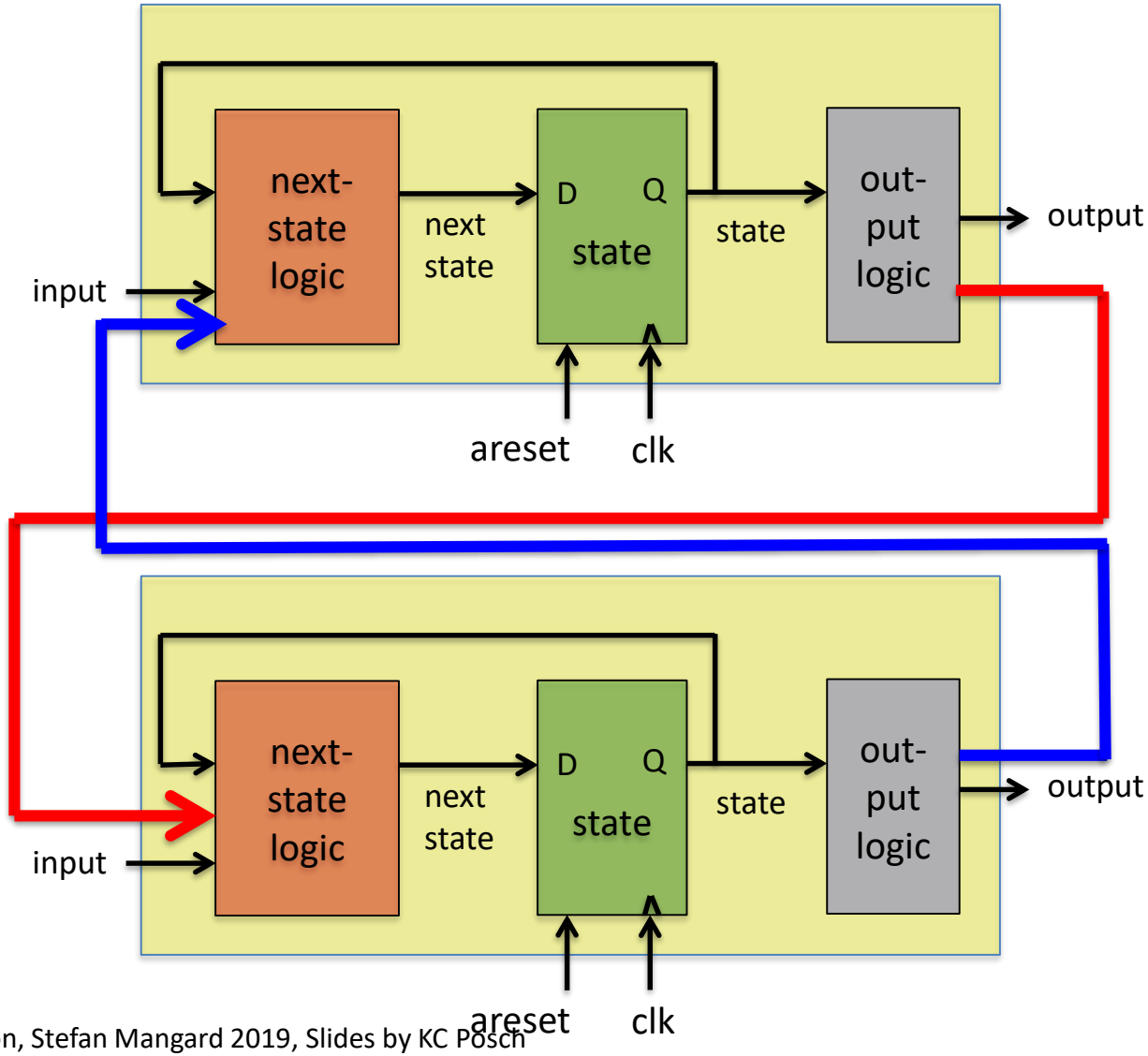
We can combine machines

- 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**!

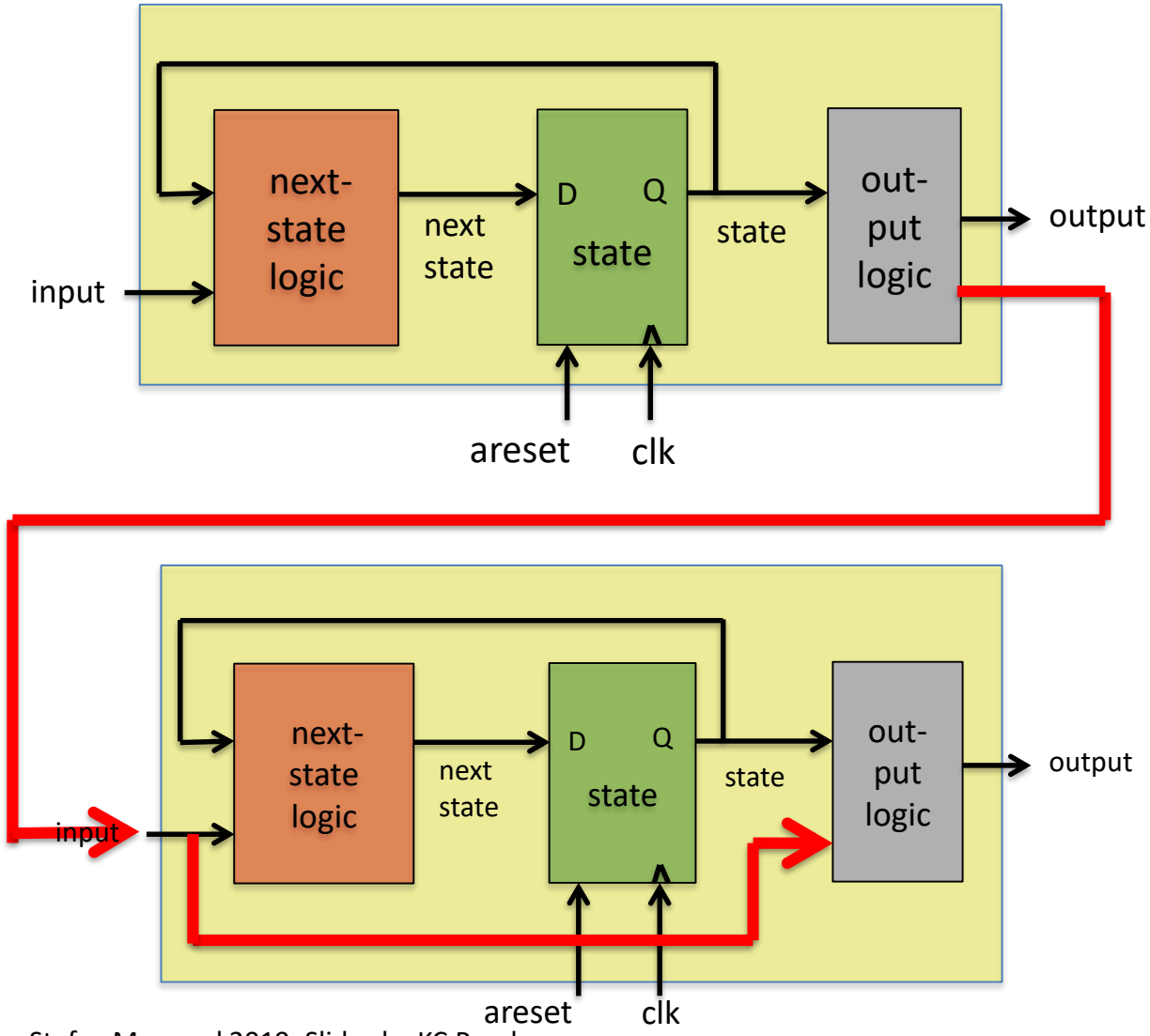
The combination of two Moore Machines creates again a (more complex) Moore Machine



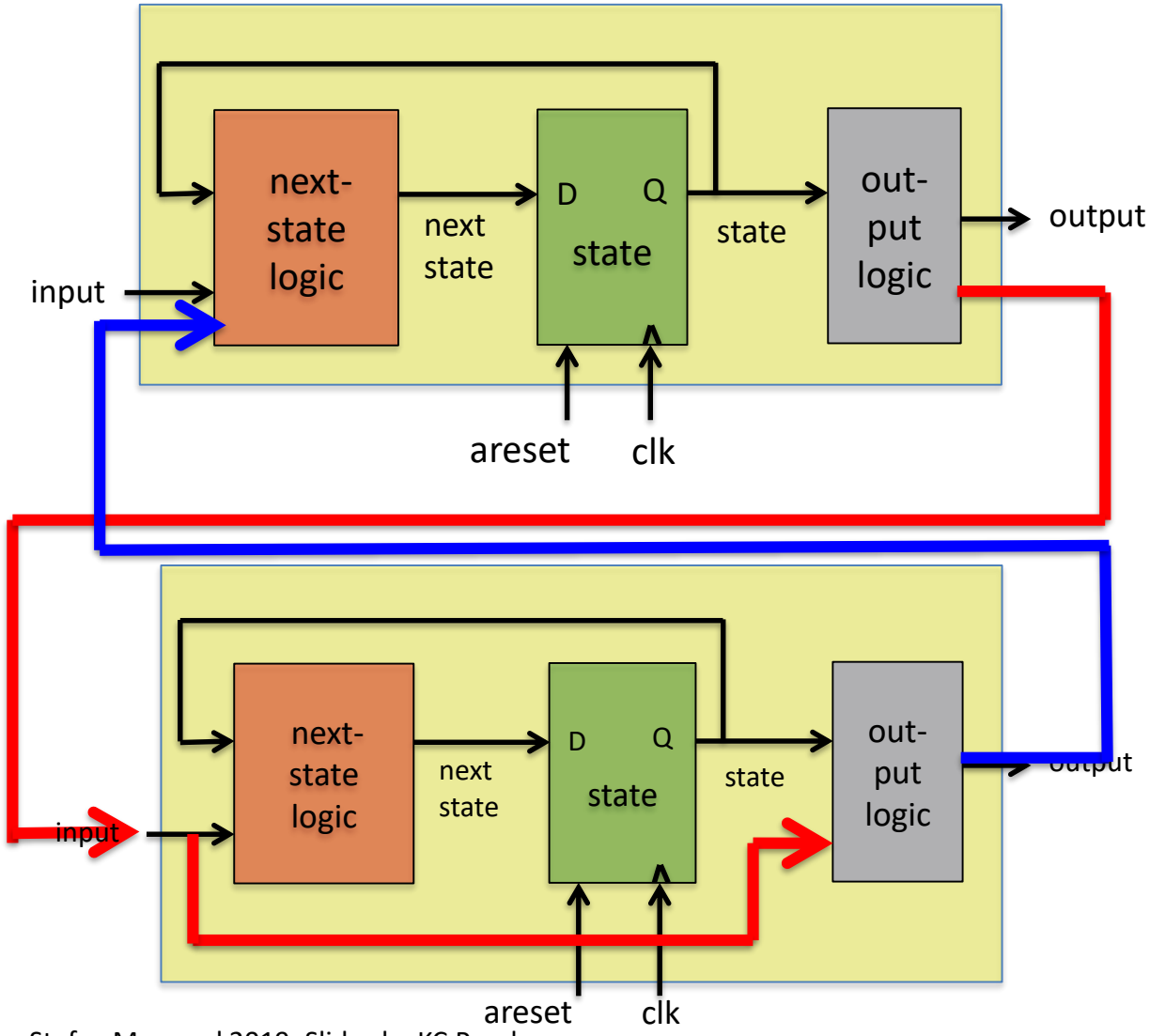
We can even connect More Machines in a loop-like fashion



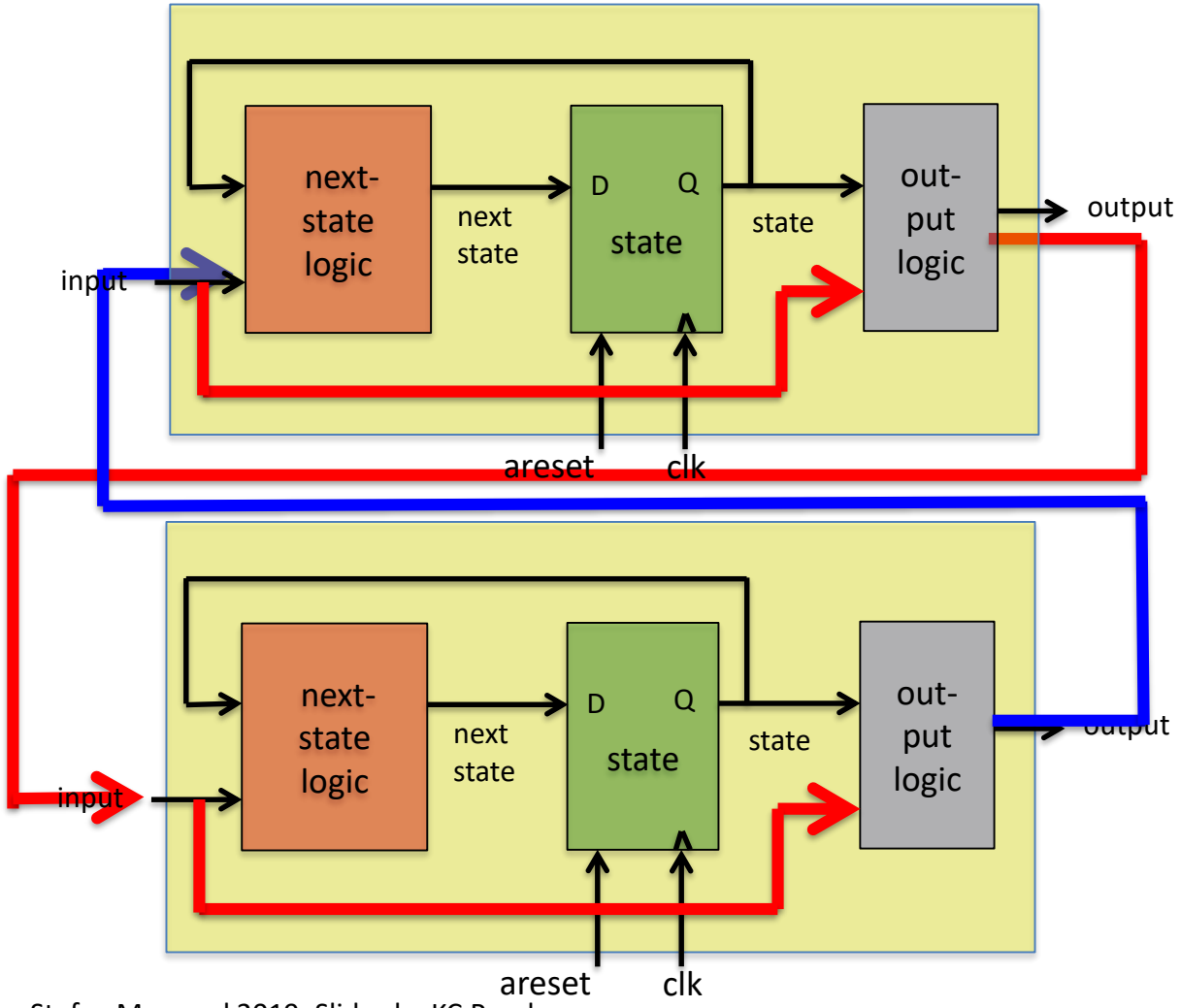
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine



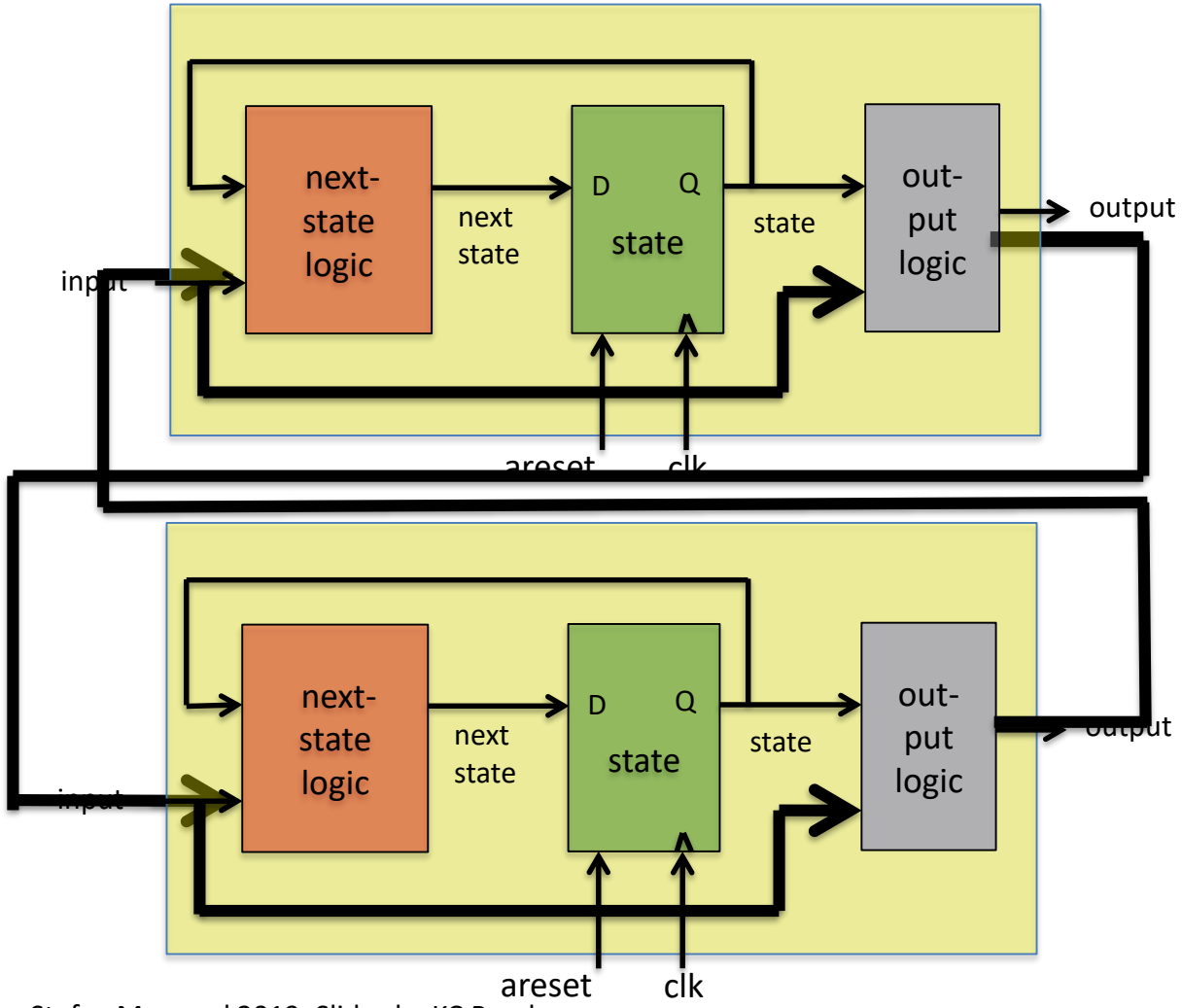
The combination of a Moore Machine with a Mealy Machine creates a Moore Machine or a Mealy Machine



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

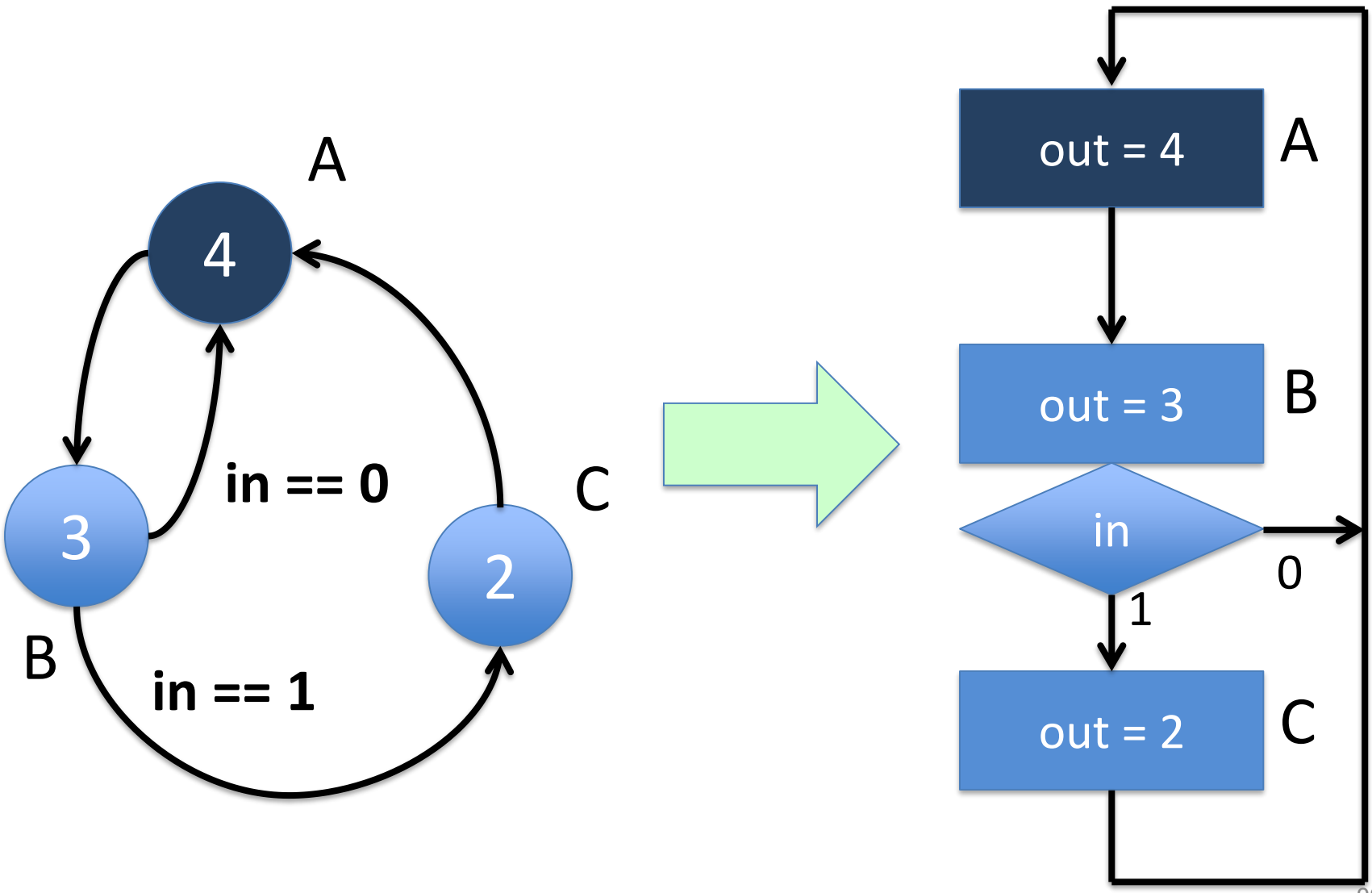
- All digital logic can in principle be built with Moore Machines and Mealy Machines.
- You always start by defining the function with a state diagram.
- If you choose values for the input signal(s), then you can derive the timing diagram by using the state diagram.
- From a state diagram, you can always derive a circuit diagram.

Algorithmic State Machines

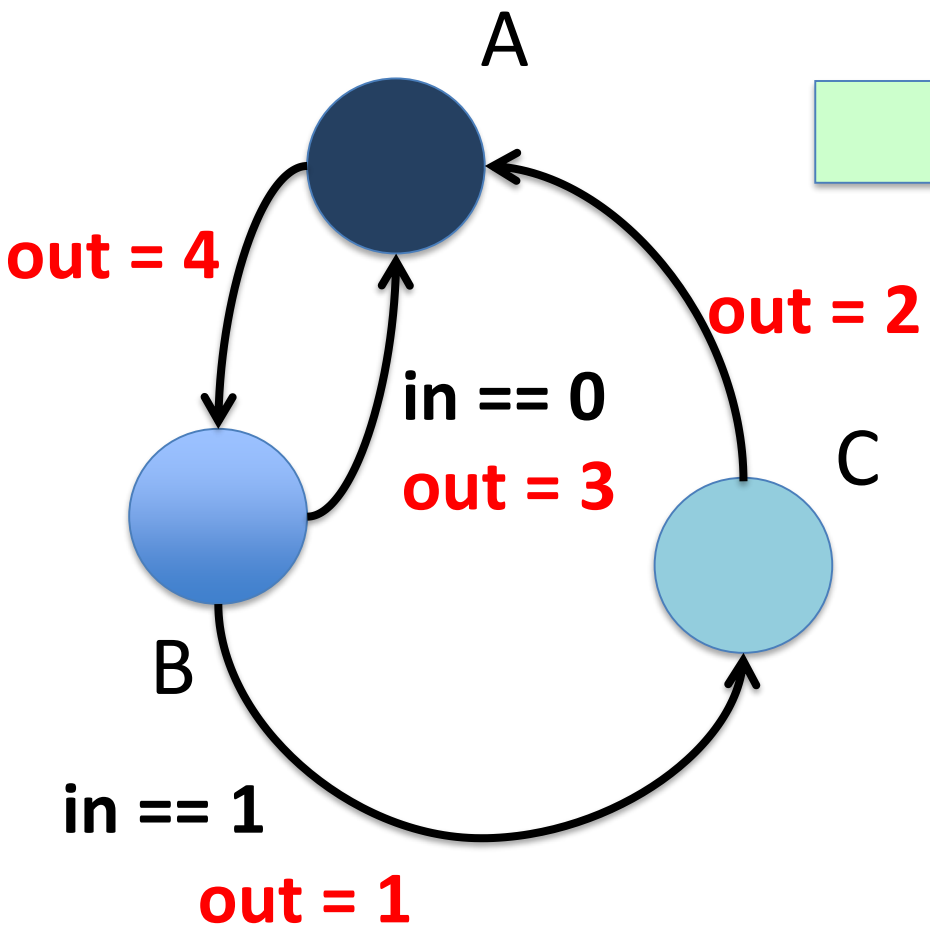
Algorithmic State Machines (ASMs)

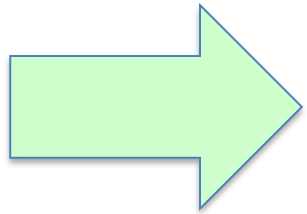
- ASMs are a useful extension to finite state machines
- ASMs allow to specify a system consisting of a data path together with its control logic
- All FSM state diagrams have an equivalent ASM diagram

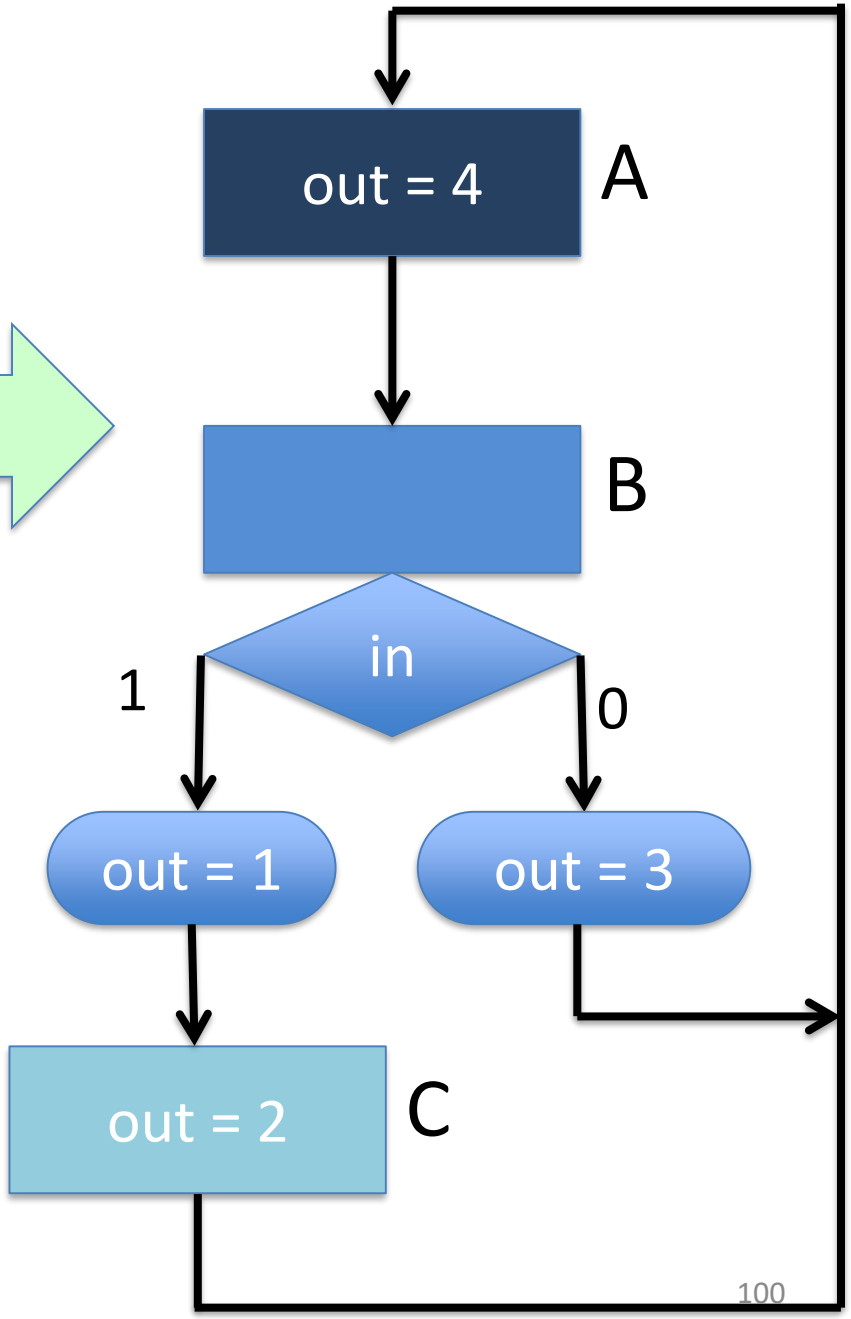
FSM state diagram → ASM diagrams



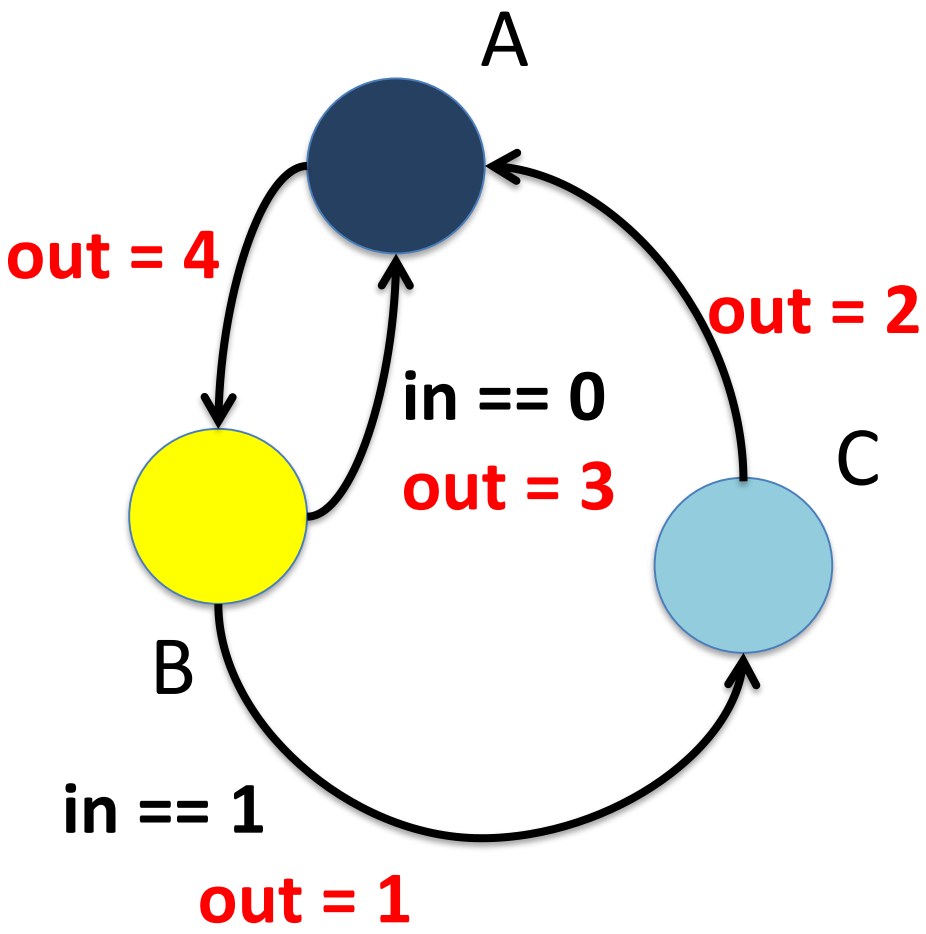
Mealy Machines

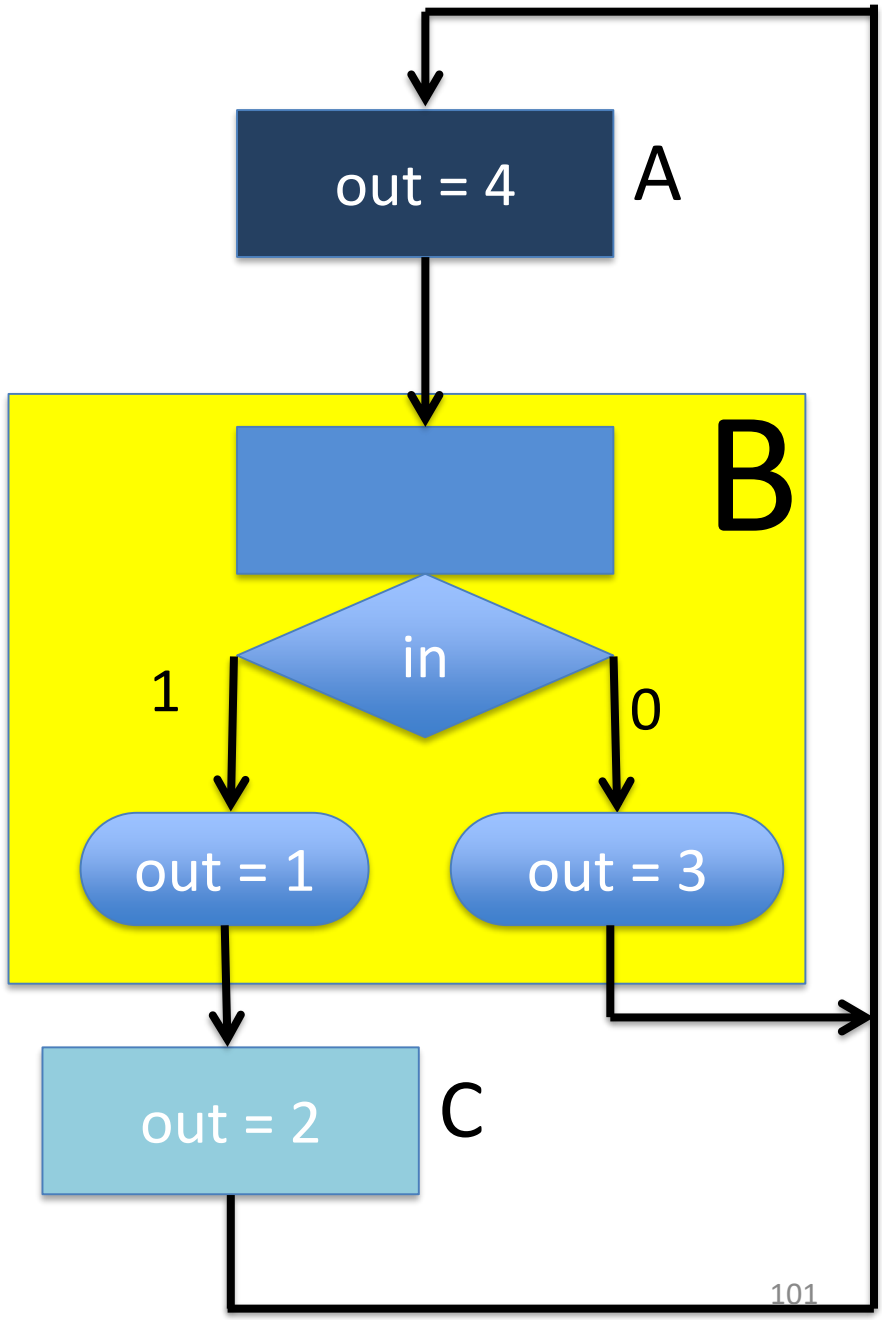






Mealy Machines

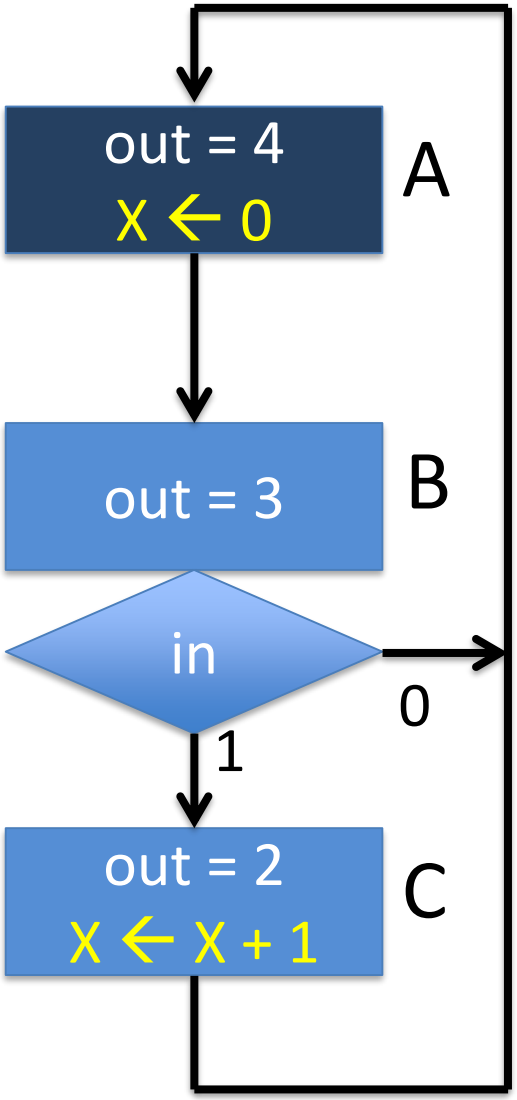




Register-Transfer Statements

- 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

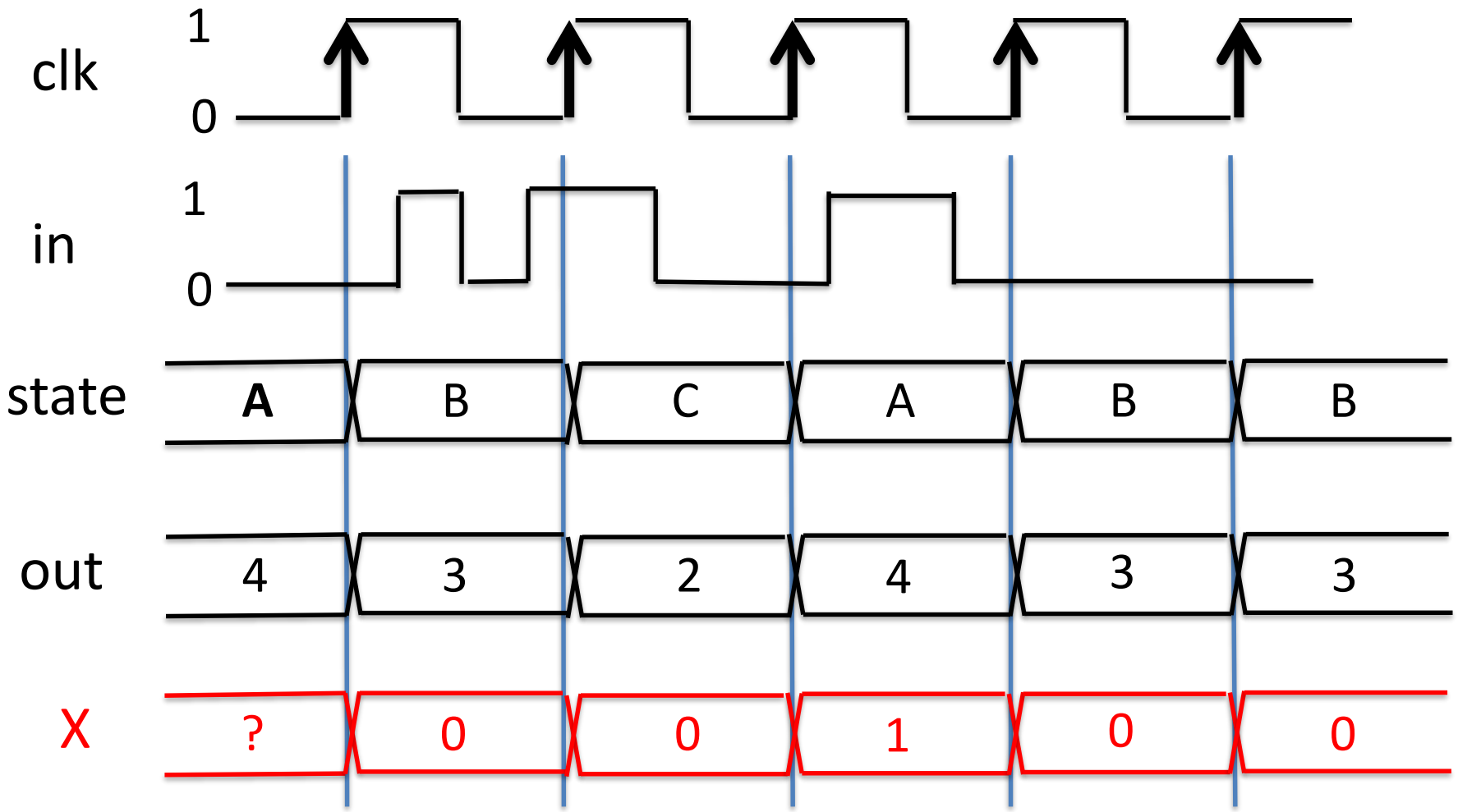


The value stored in register X gets 0 at the state transition from state A to state B.

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

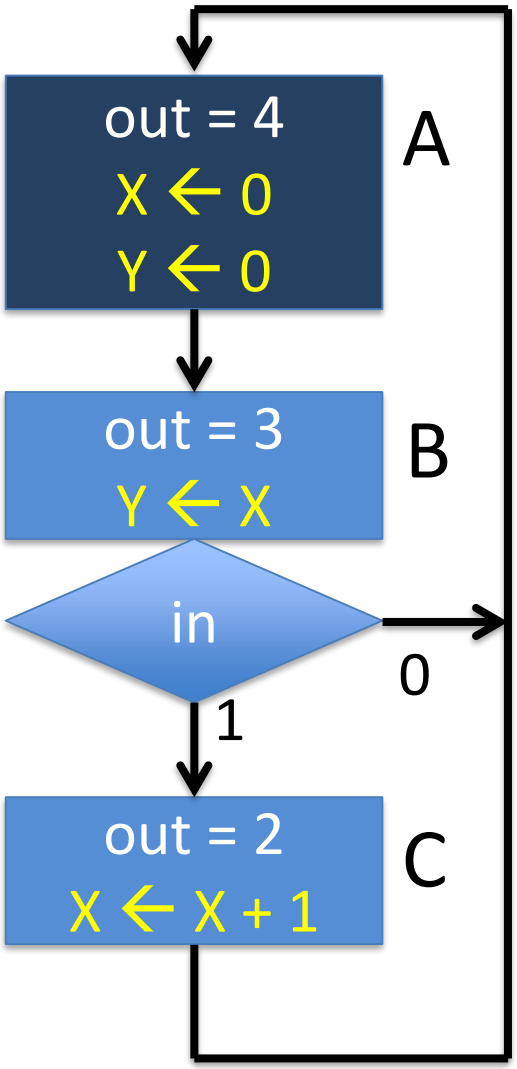


"=" versus "←"

- 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

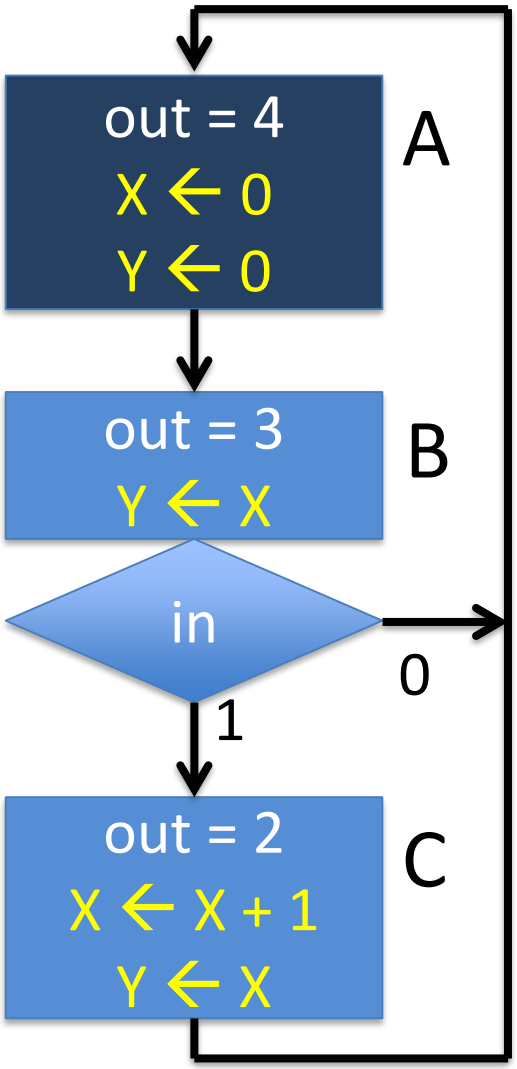


The values stored in register X and register Y become 0 at the state transition from state A to state B.

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.

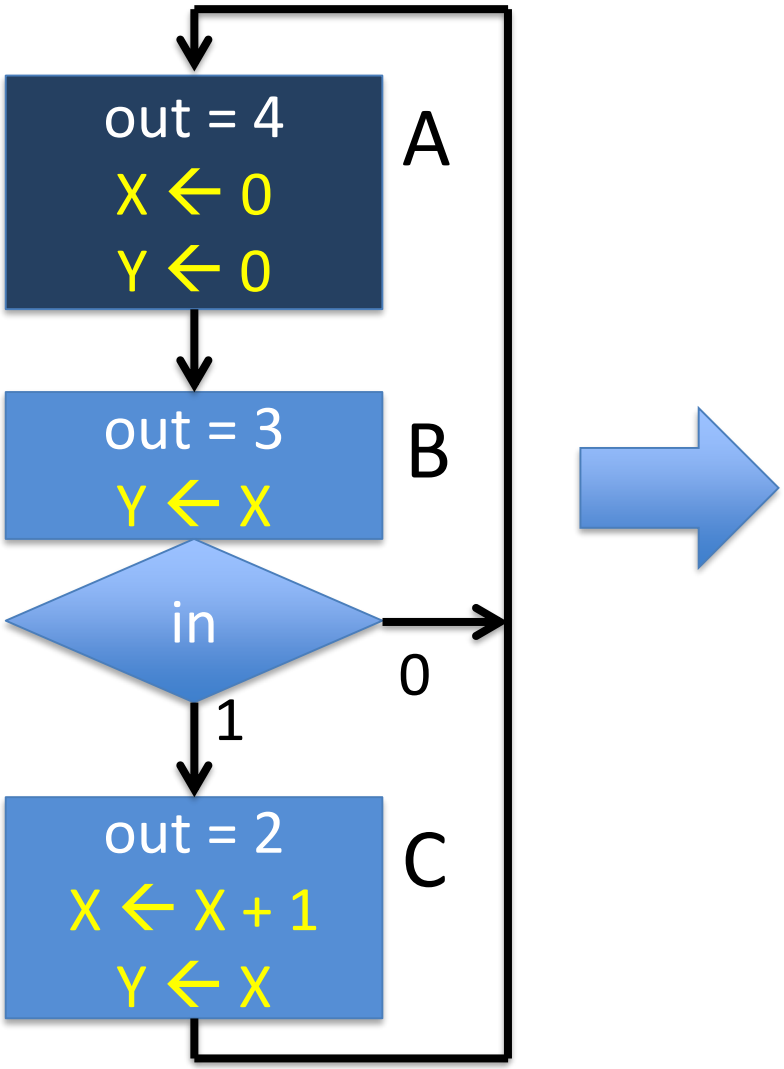
The value in register X gets incremented at the state transition following state C. Register Y gets the "old" value from X; i.e the value before X gets incremented.

For a Verilog example of this ASM graph see

chapter_03/example_00/05_asm.v

Manual Synthesis of an ASM Graph in Logisim

Register-transfer statements define the data path



The "neighborhood" of register X:

$X \leftarrow 0$

$X \leftarrow X$

$X \leftarrow X+ 1$

The "neighborhood" of register Y:

$Y \leftarrow 0$

$Y \leftarrow X$

The neighborhood of register X

Case 0: $X \leftarrow X$

Case 1: $X \leftarrow X+ 1$

Case 2: $X \leftarrow 0$

We need to distinguish between 3 cases.

The neighborhood of register X

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

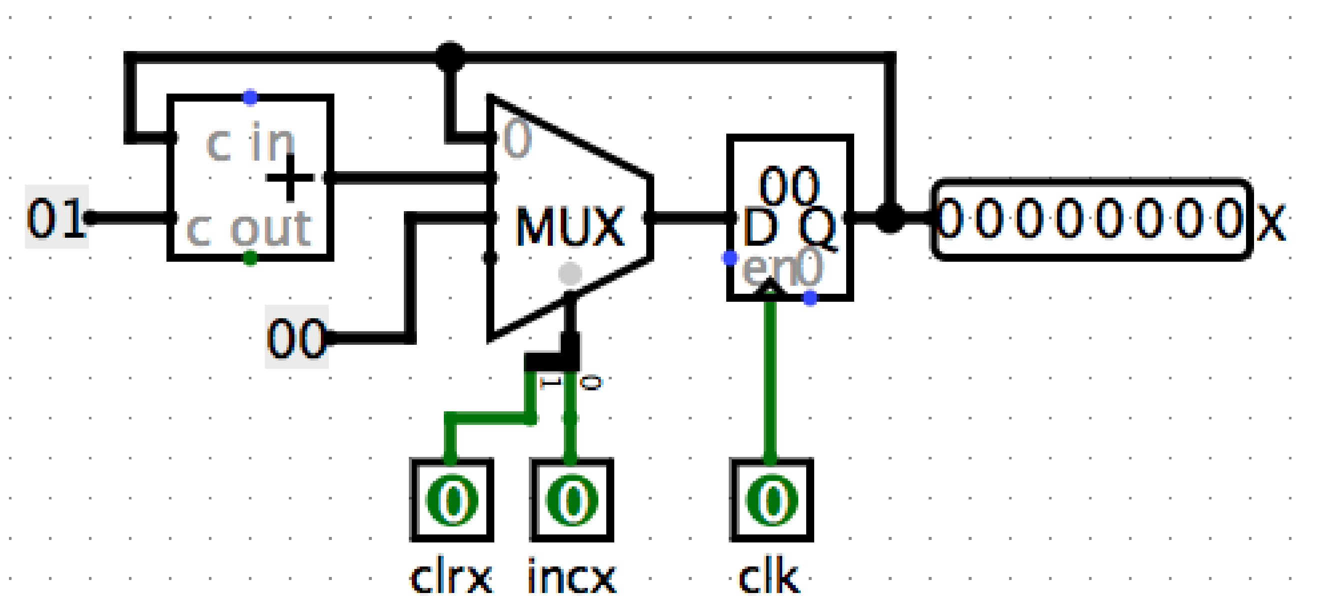
We use binary notation and name the two binary select variables.

The neighborhood of register X

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

We model the truth table with a multiplexer. For incrementing X we use an adder.

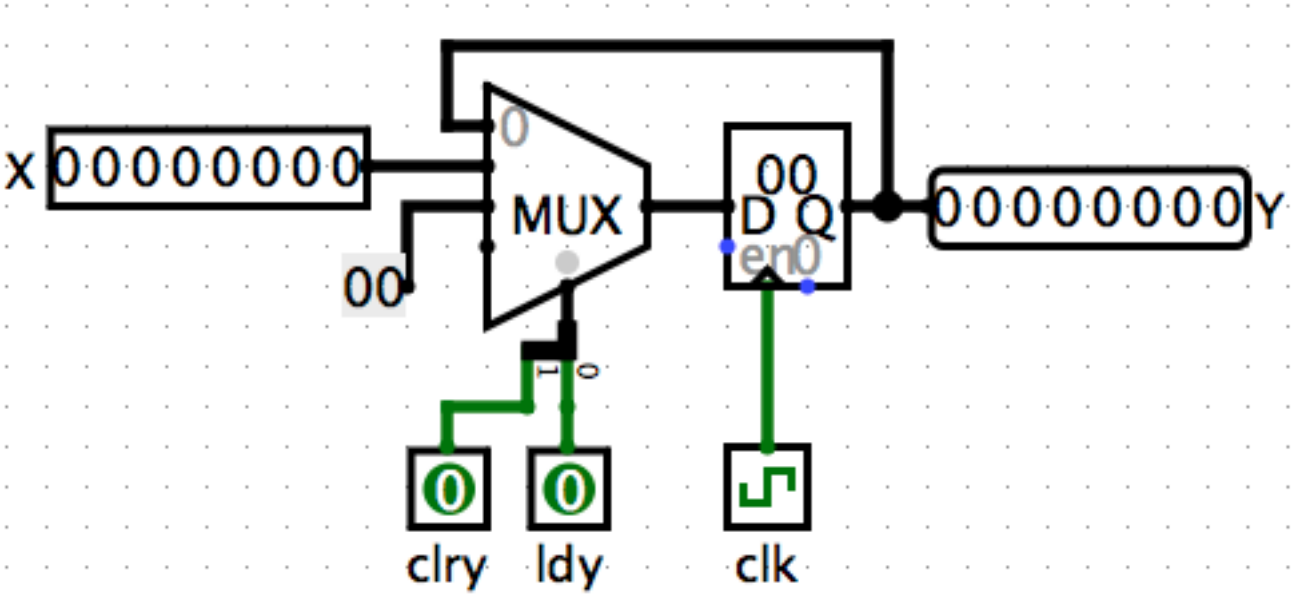
With Logisim we can model the neighborhood of Register and also simulate.



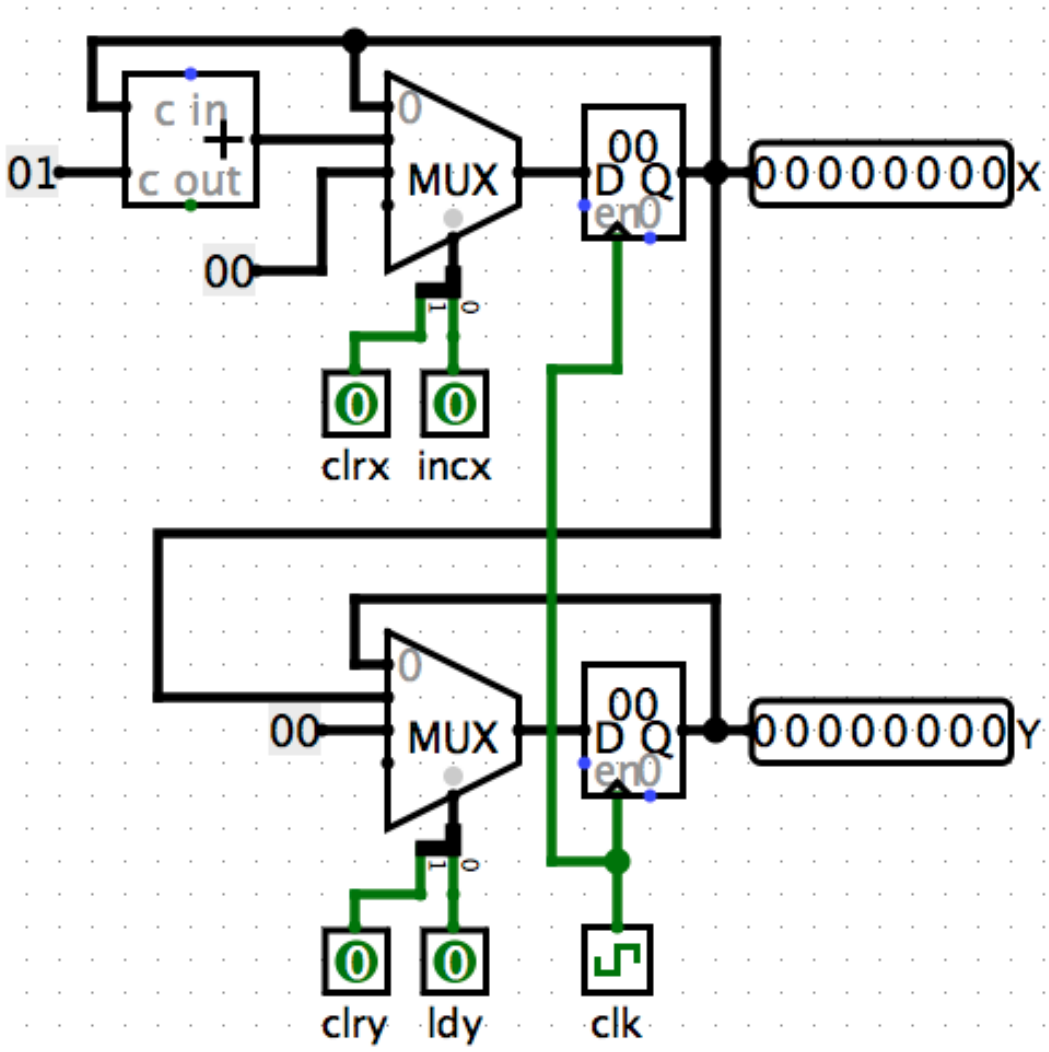
The neighborhood of register Y

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



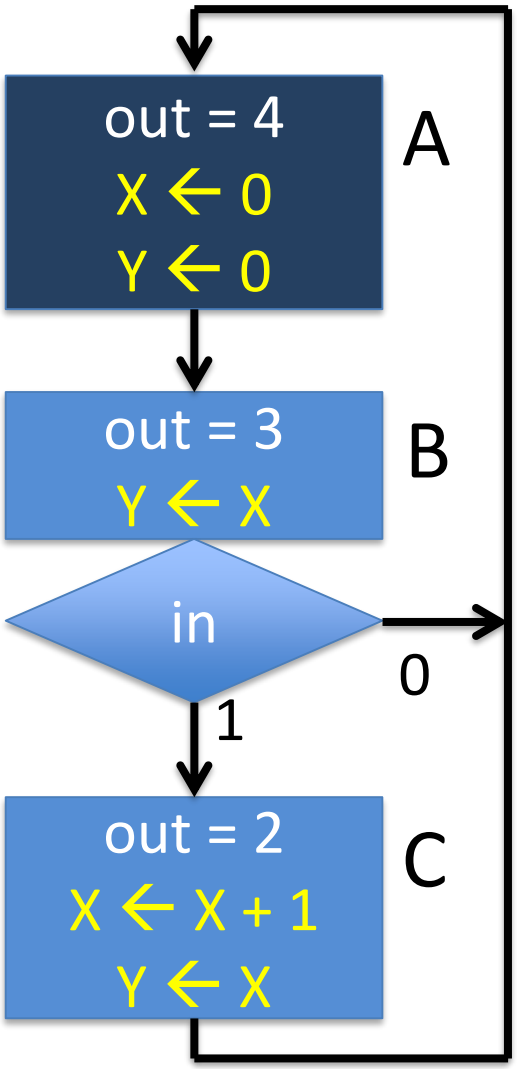
We combine the two neighborhoods.

Note that both neighborhoods are Moore machines.

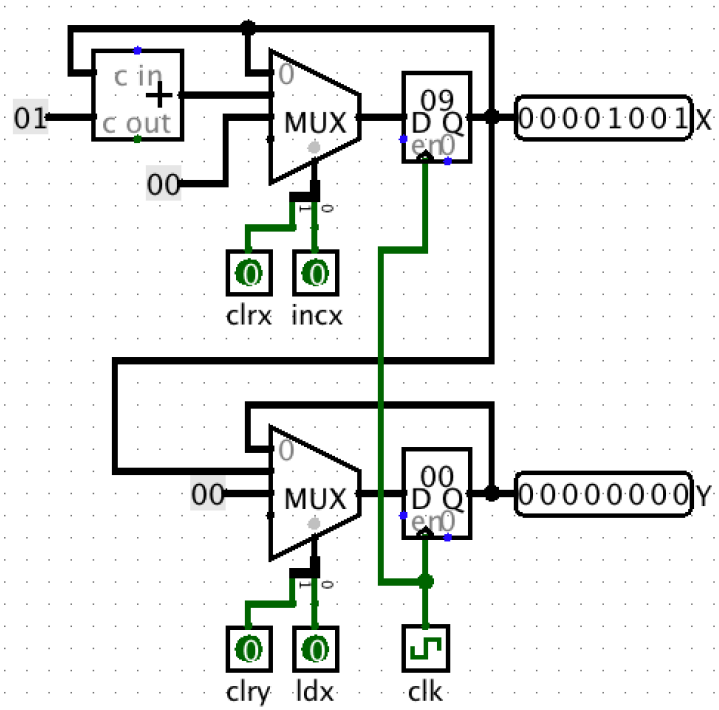
The Moore machine for X has 2 inputs: clrx and incx. Since we have chosen an 8-bit register for X, we have 256 possible states. The output function is the identity function.

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

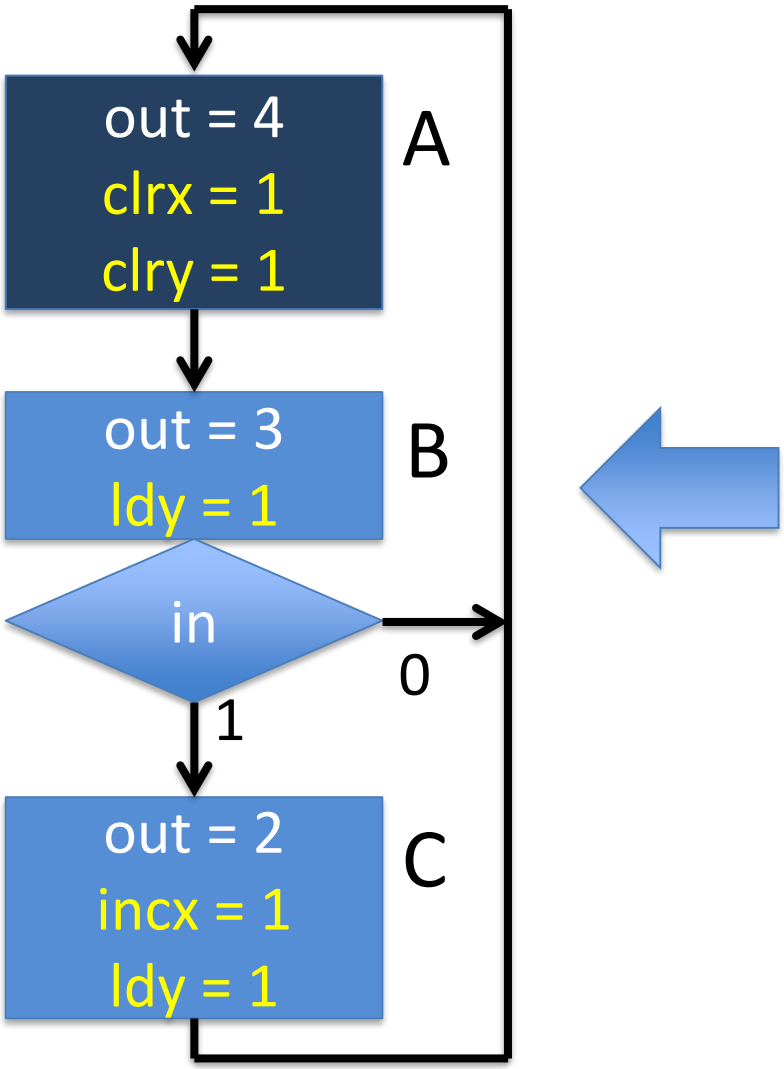
Register-transfer statements define the data path

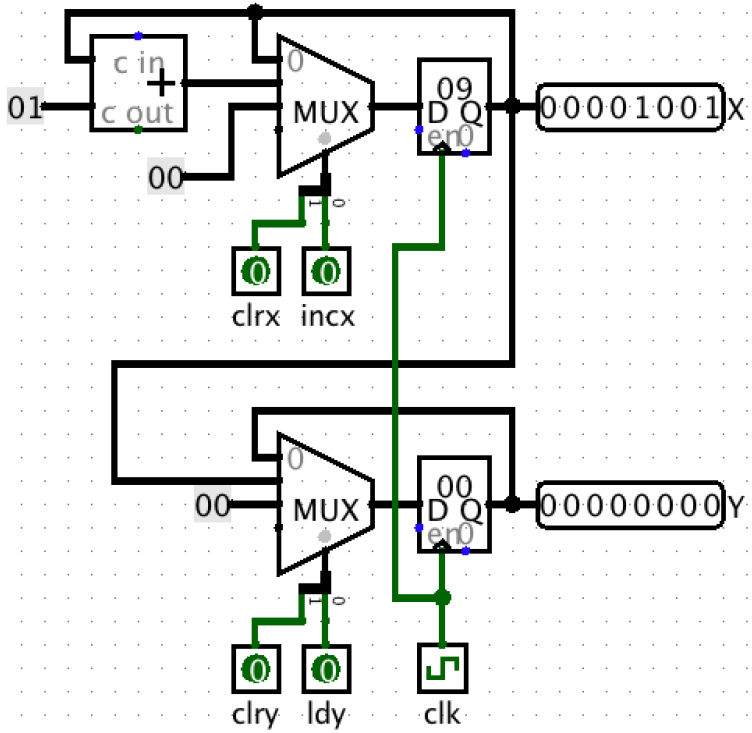


dp:

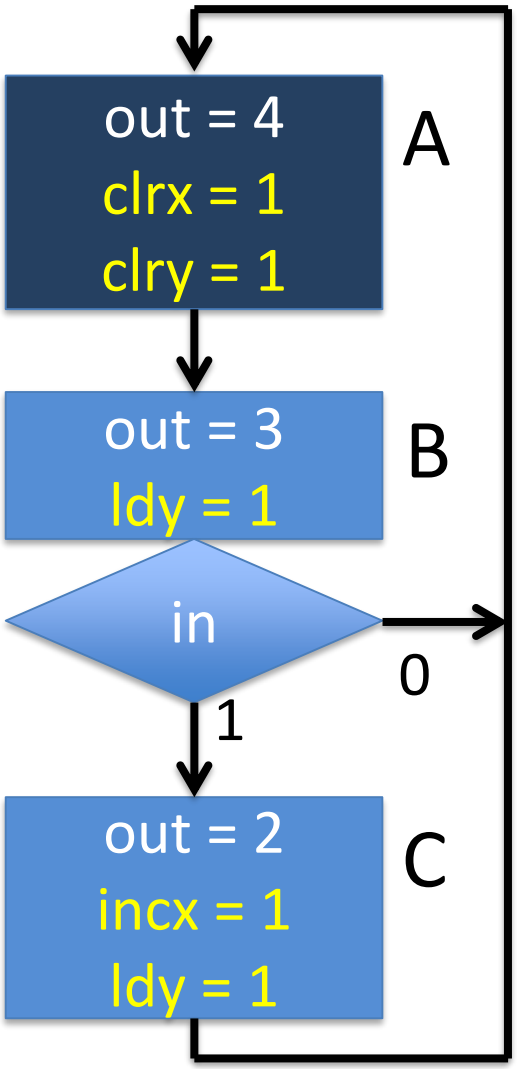


The control logic needs to provide the control signals



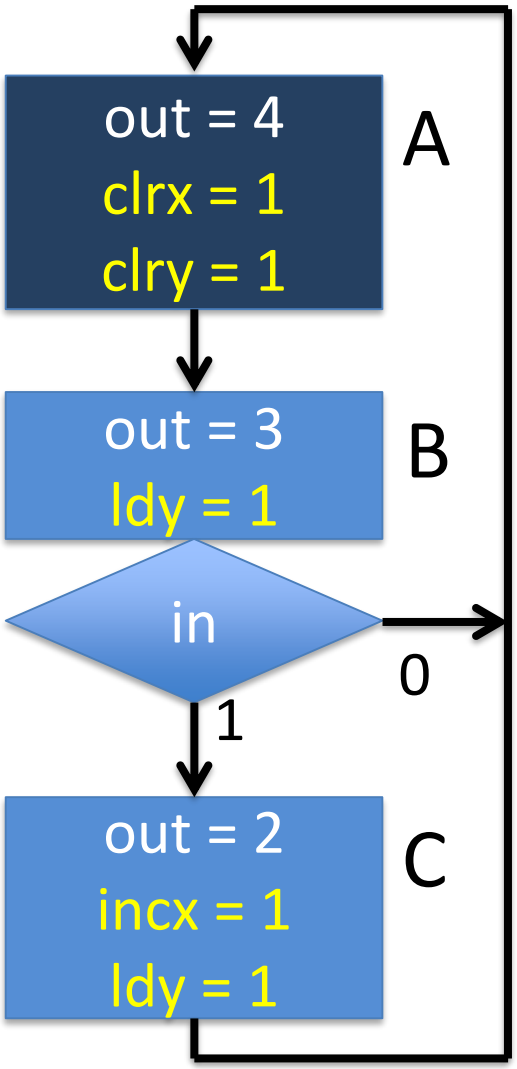


Truth table of output logic of controller



state	clrx	incx	clry	ldy	out
A	1	0	1	0	4
B	0	0	0	1	3
C	0	1	0	1	2

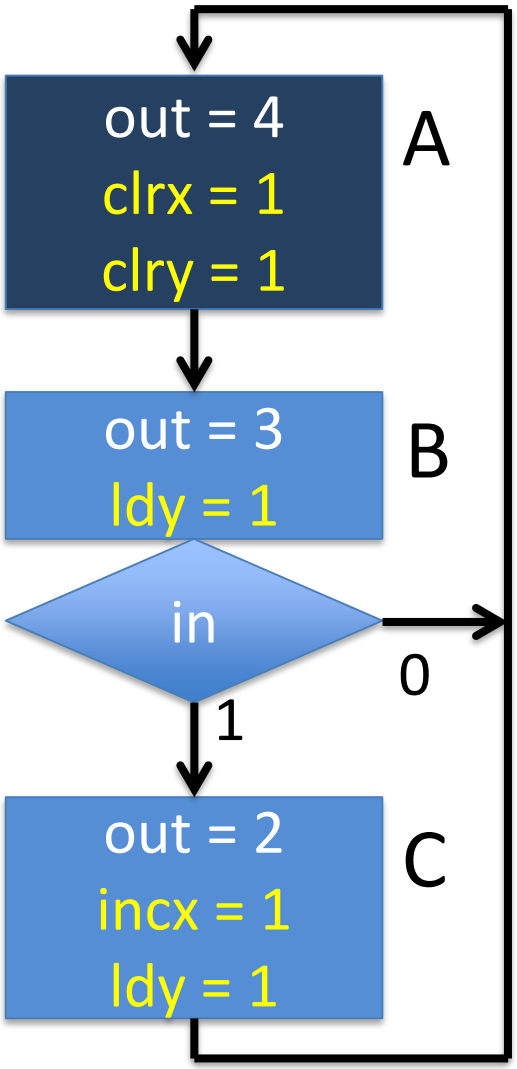
Truth table of output logic of controller



A 00
B 01
C 10

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

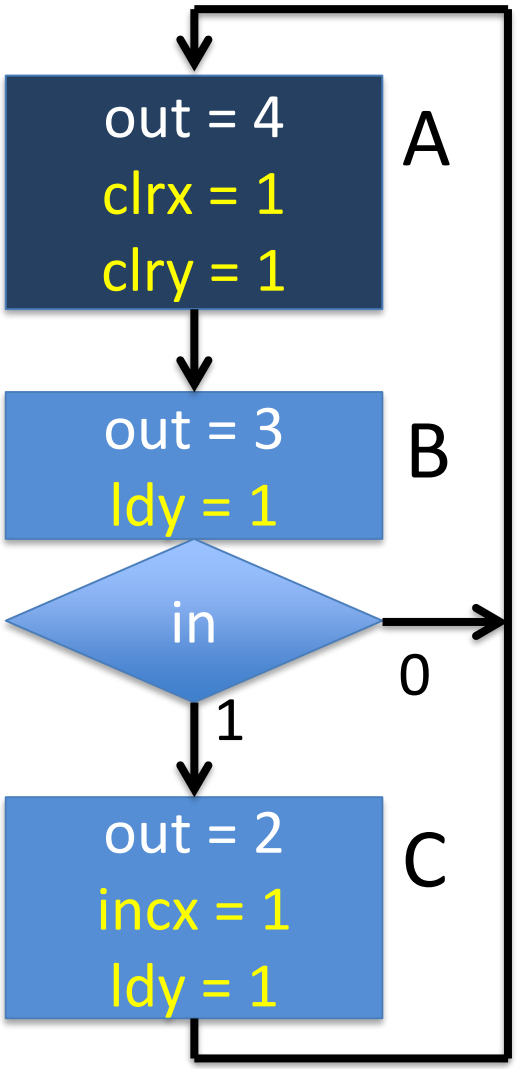
Truth table of next-state logic of controller



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

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

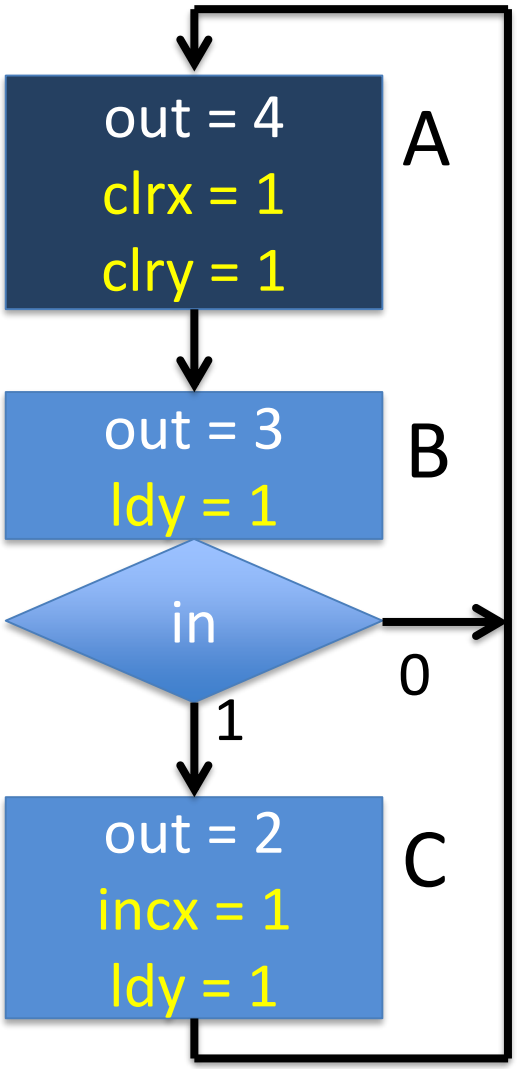


state	clrx	incx	clry	ldy	out
0 0	1	0	1	0	100
0 1	0	0	0	1	011
1 0	0	1	0	1	010

A 00
B 01
C 10

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

Truth table of next-state logic of controller



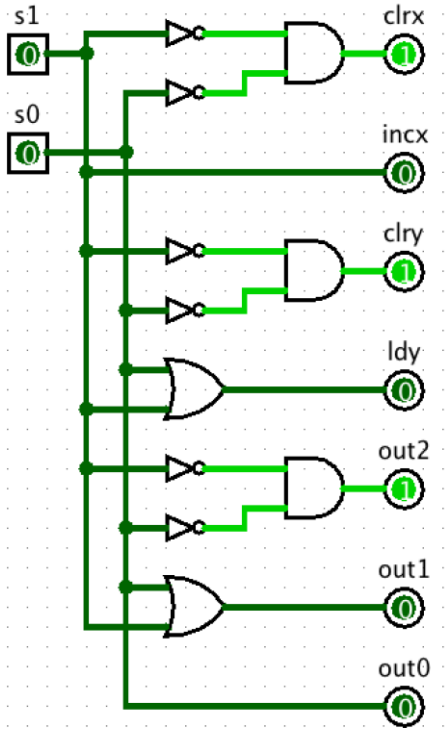
s1	s0	clrx	incx	clry	ldy	out
0	0	1	0	1	0	100
0	1	0	0	0	1	011
1	0	0	1	0	1	010

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

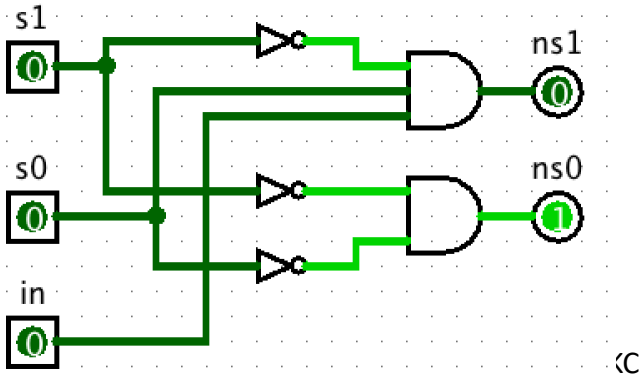
From truth table to implementation

outl:



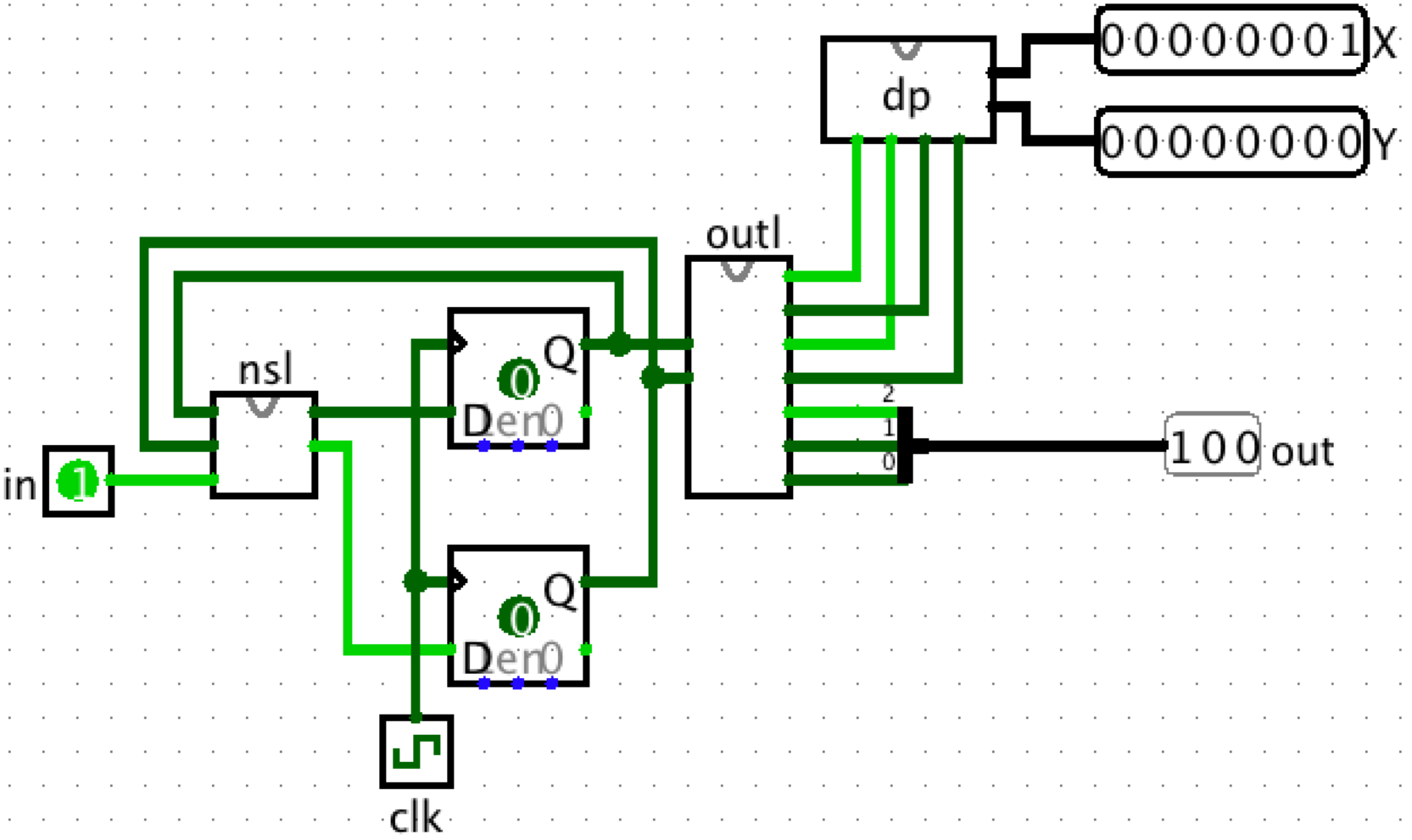
s1	s0	clrx	incx	clry	ldy	out
0	0	1	0	1	0	100
0	1	0	0	0	1	011
1	0	0	1	0	1	010
1	1	x	x	x	x	xxx

nsl:



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

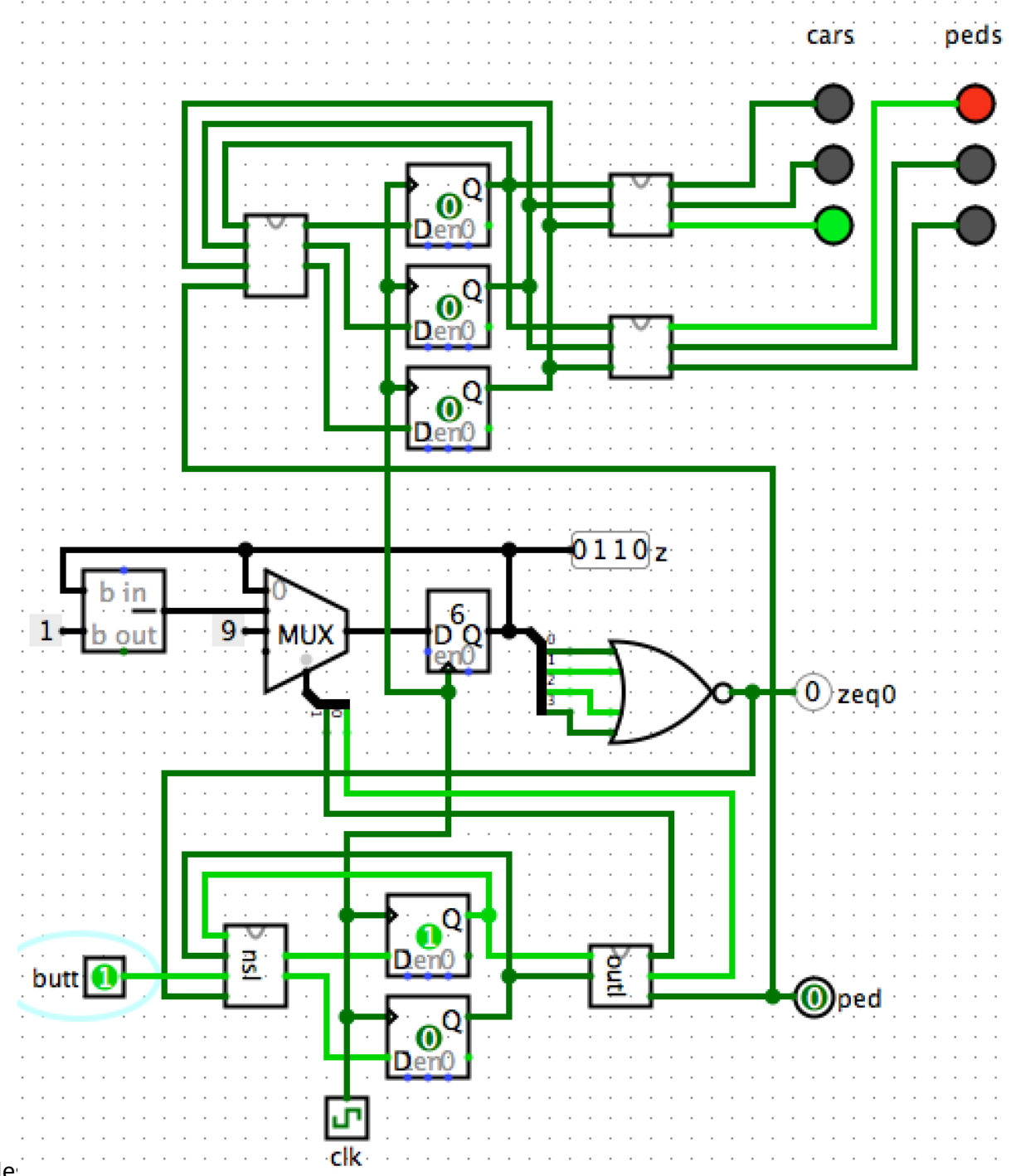


That's it.

- In principle, we can describe any synchronous automaton with an ASM diagram.
- In principle, every synchronous digital system can be described by a collection of ASM diagrams.
- The transformation from an ASM diagram into a logic circuit is defined by an algorithm; we call this algorithm "synthesis".

Hands-On Example

City of Tiny Lights



Overview

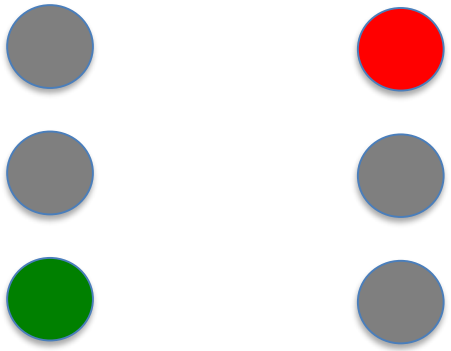
- Design a finite state machine from a functional description to logic layer.
- Decompose an ASM-diagram into data path and control logic.
- Connect several finite state machines into a larger finite state machine.

A traffic light

- Usually green for cars and red for pedestrians.
- A pedestrian wants to cross.
- The pedestrian hits a button and it gets green for her.
- Shortly afterwards, the cars get green light again.

An ASM-diagram of a simple traffic light

Pedestrians have access to an input button to ask for green light.

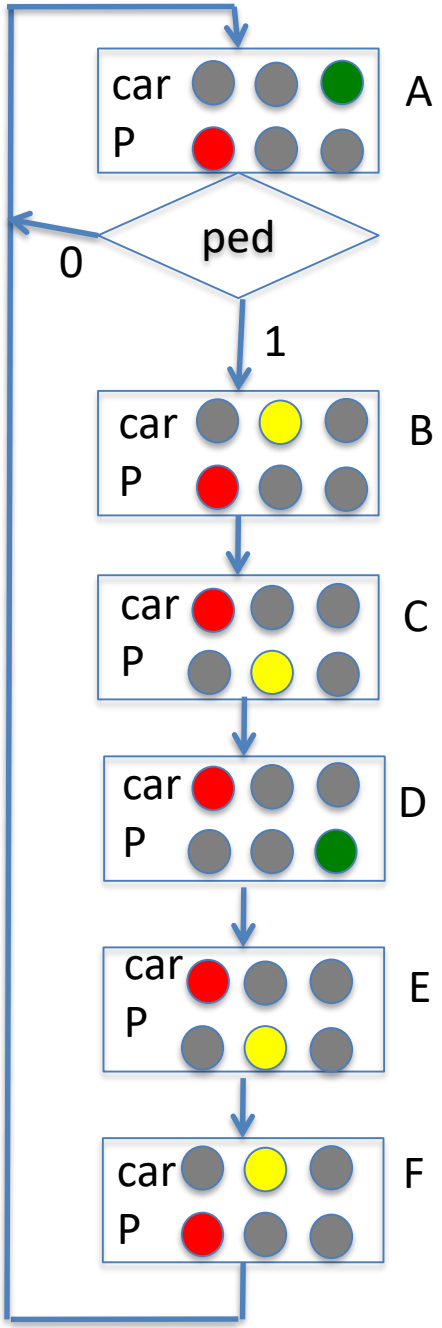


cars pedestrians

An ASM-diagram of a simple traffic light

Pedestrians have access to an input button to ask for green light.

Input button is called “ped”.



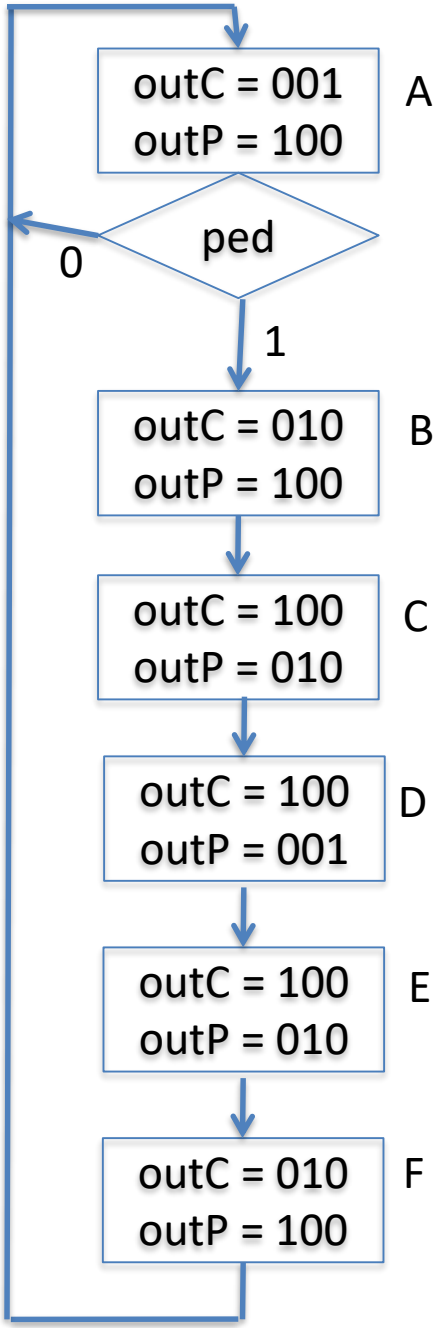
An ASM-diagram of a simple traffic light

outC = 001
outP = 100

outC is output visible to cars

outP is output visible to pedestrians

“ped” is an input button for pedestrians to request green light

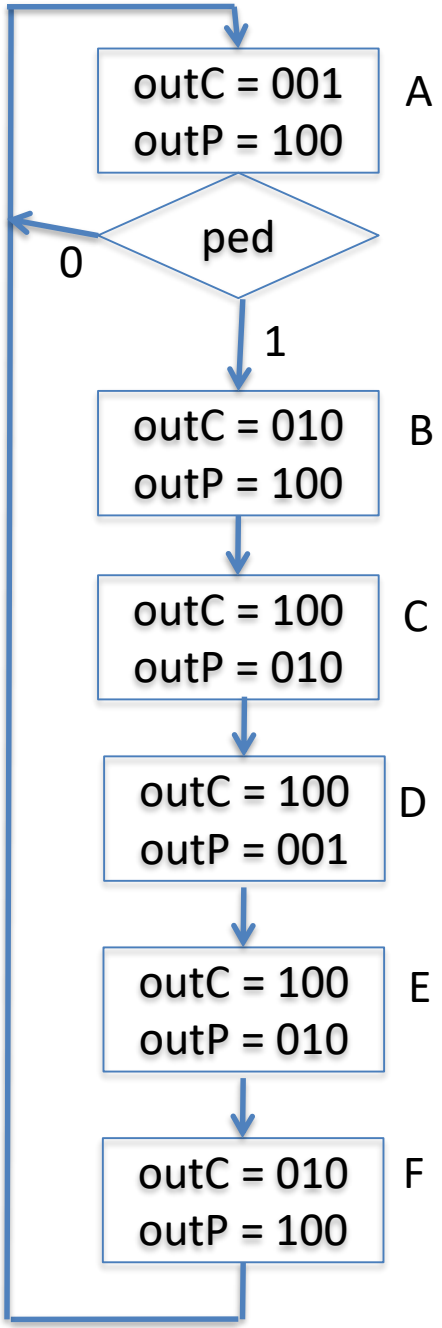


Next-state table

state	ped	next_state
A	0	A
A	1	B
B	x	C
C	x	D
D	x	E
E	x	F
F	x	A

"x" stands for "don't care"

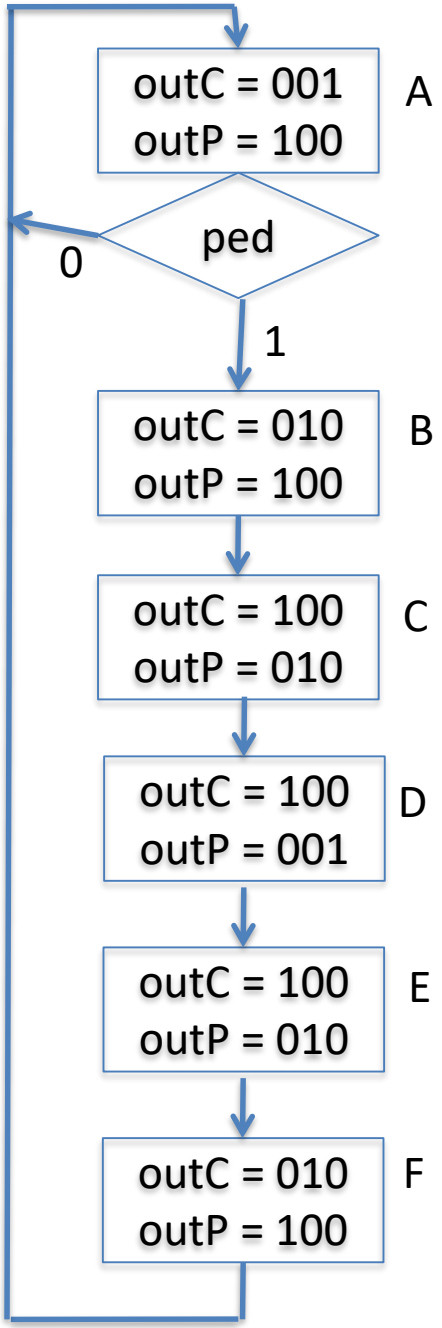
6 states: Thus, we need at least 3 bits for storing the state.



Next-state table

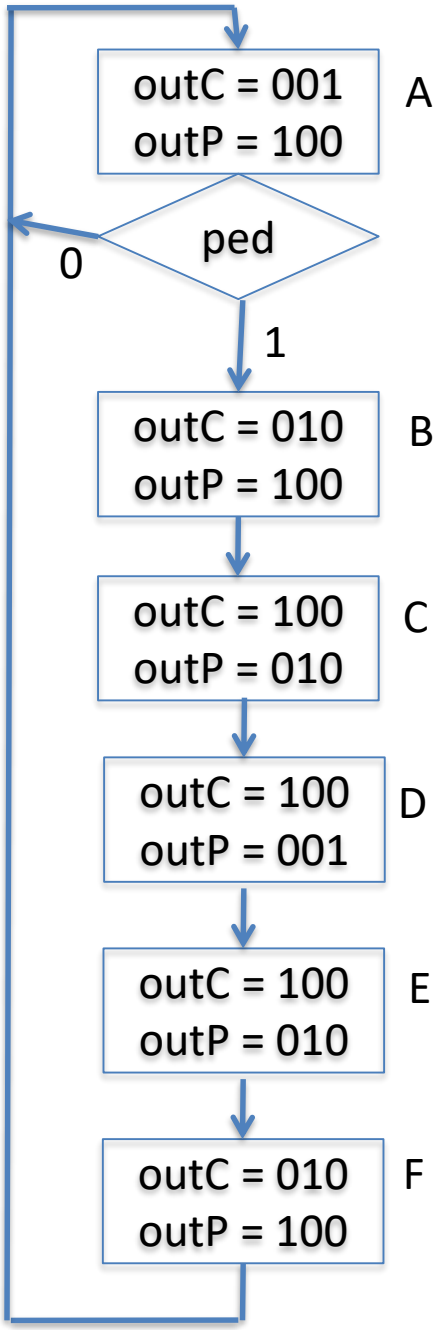
s2	s1	s0	ped	ns2	ns1	ns0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	0
0	1	0	x	0	1	1
0	1	1	x	1	0	0
1	0	0	x	1	0	1
1	0	1	x	0	0	0
1	1	0	x	x	x	x
1	1	1	x	x	x	x

"x" stands for "don't care"

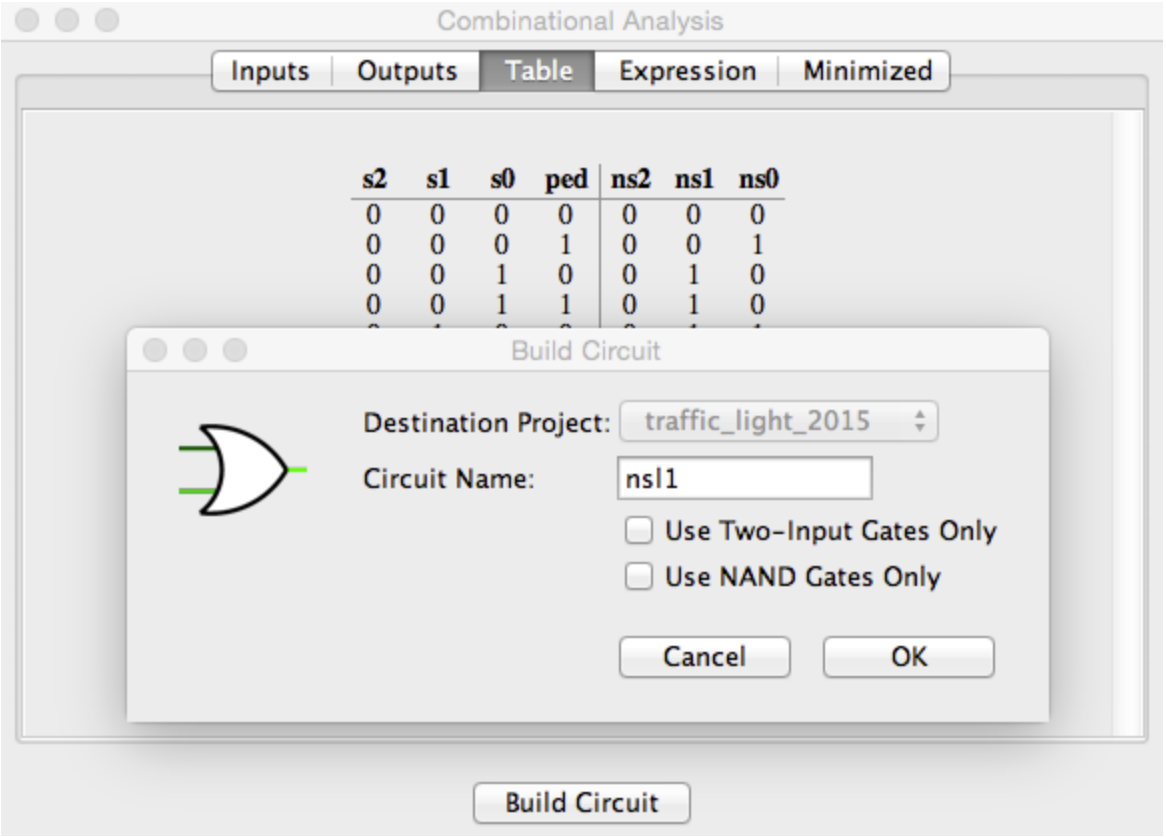


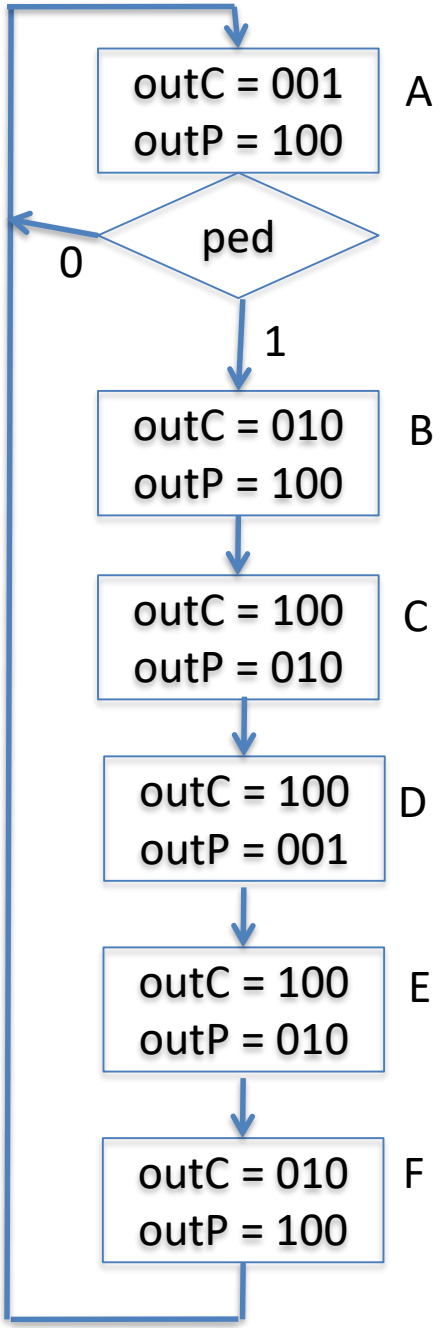
Next-state table (Logisim)

s2	s1	s0	ped	ns2	ns1	ns0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	1
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	x	x	x
1	1	0	1	x	x	x
1	1	1	0	x	x	x
1	1	1	1	x	x	x

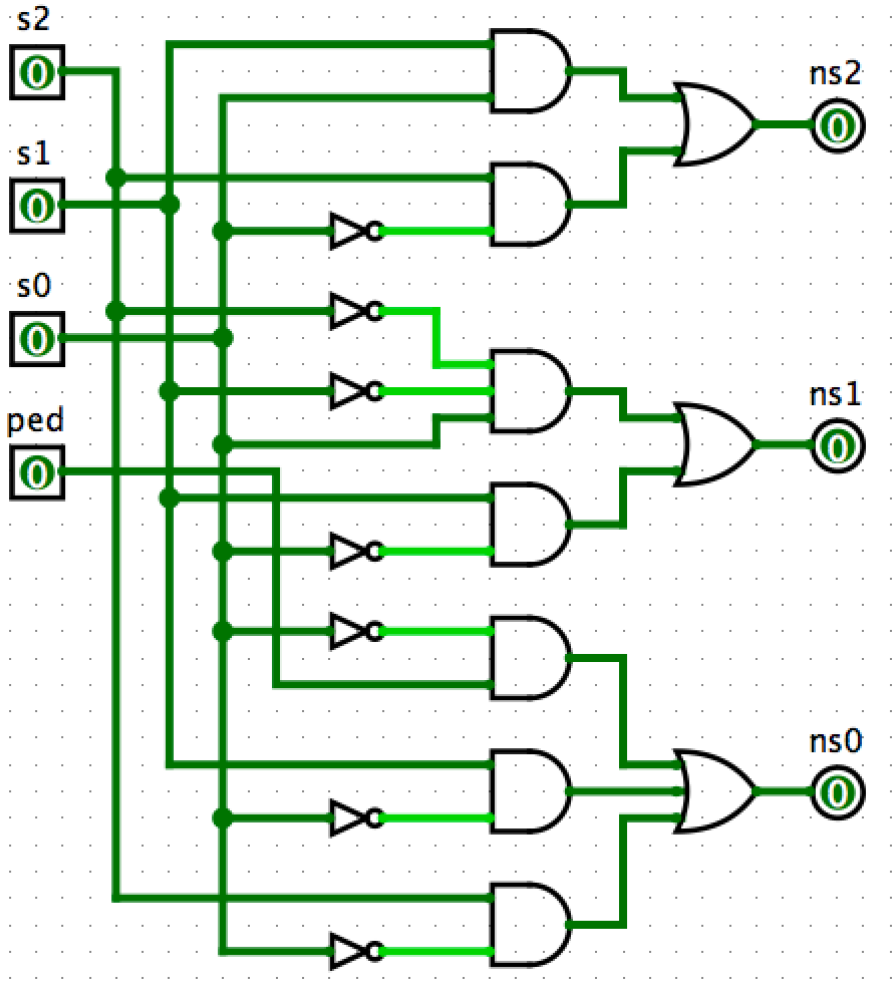


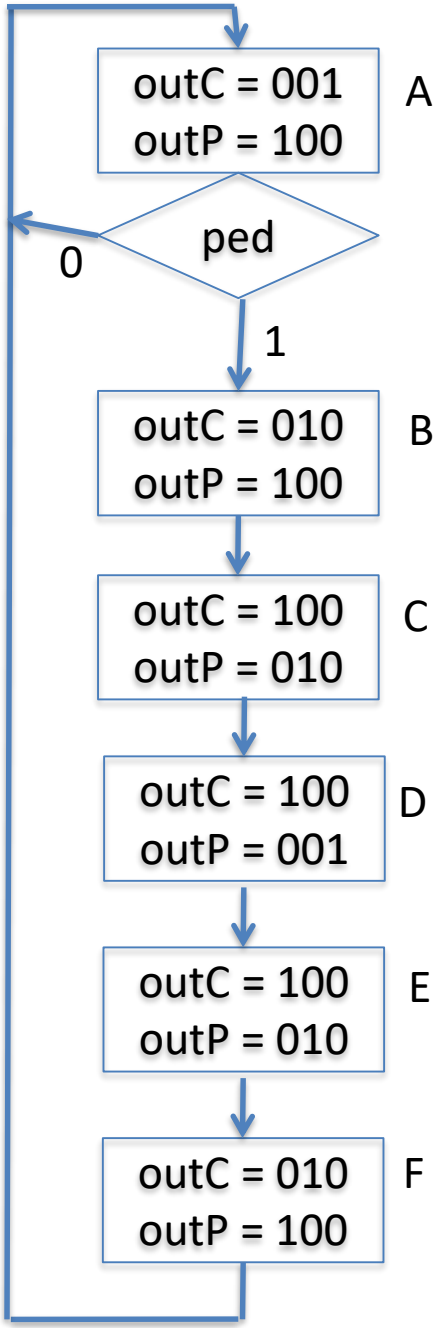
Synthesis mit Logisim





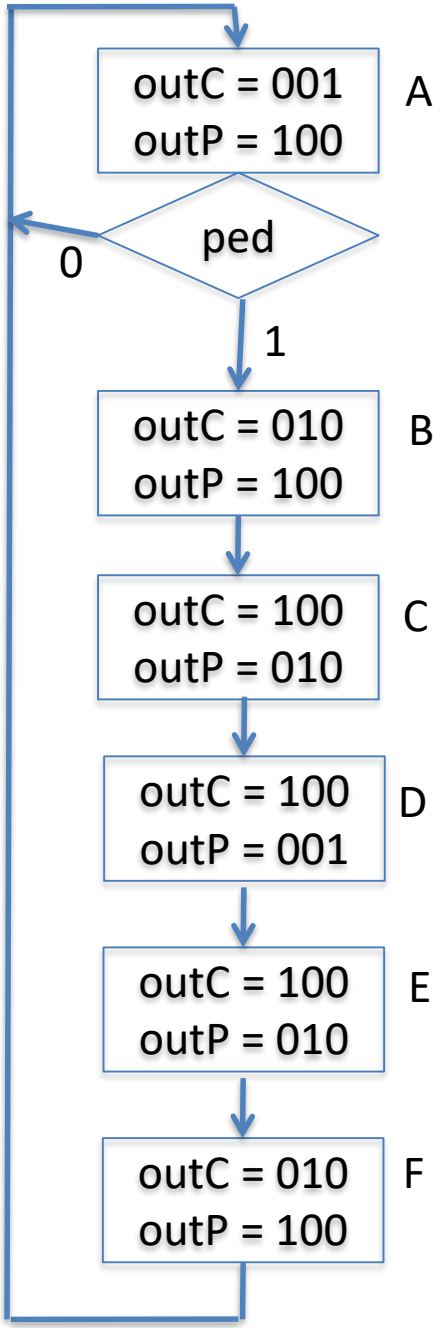
Next-state function synthesized in Logisim



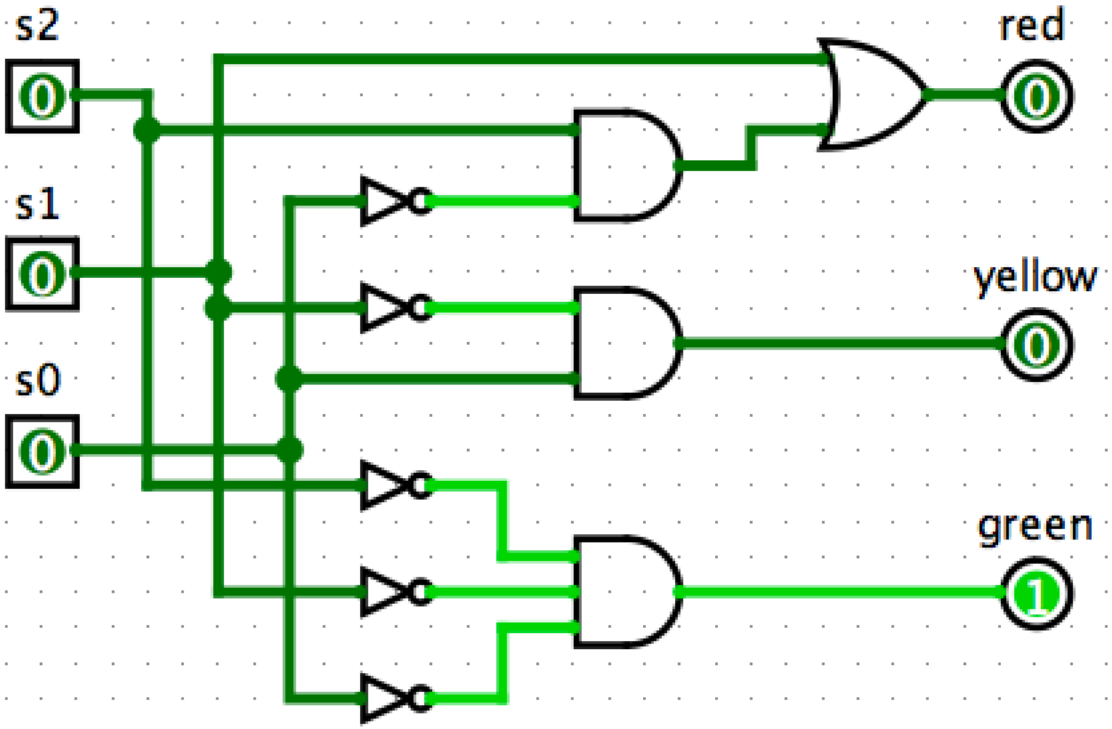


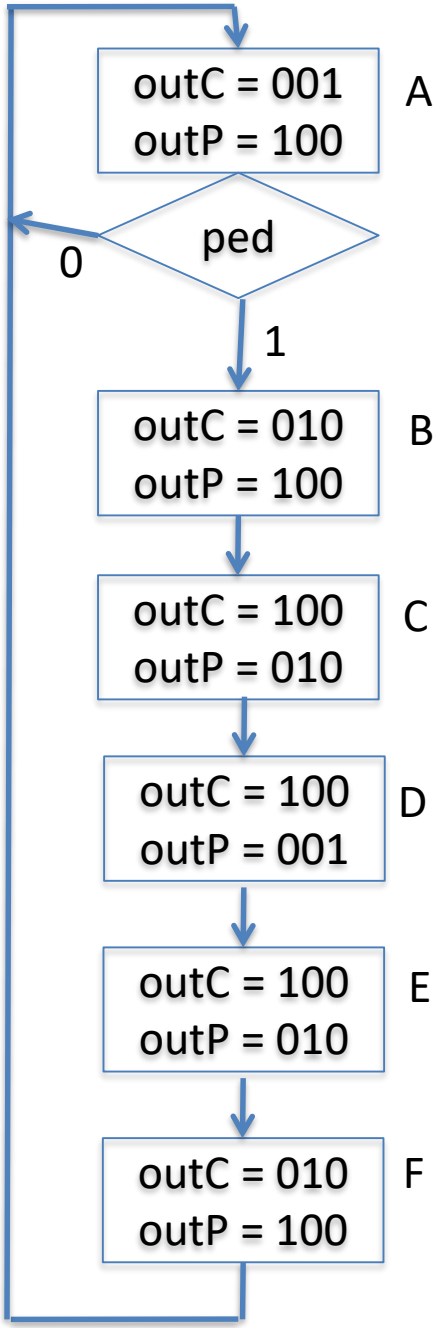
Output table for cars

s2	s1	s0	red	yellow	green
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	x	x	x
1	1	1	x	x	x



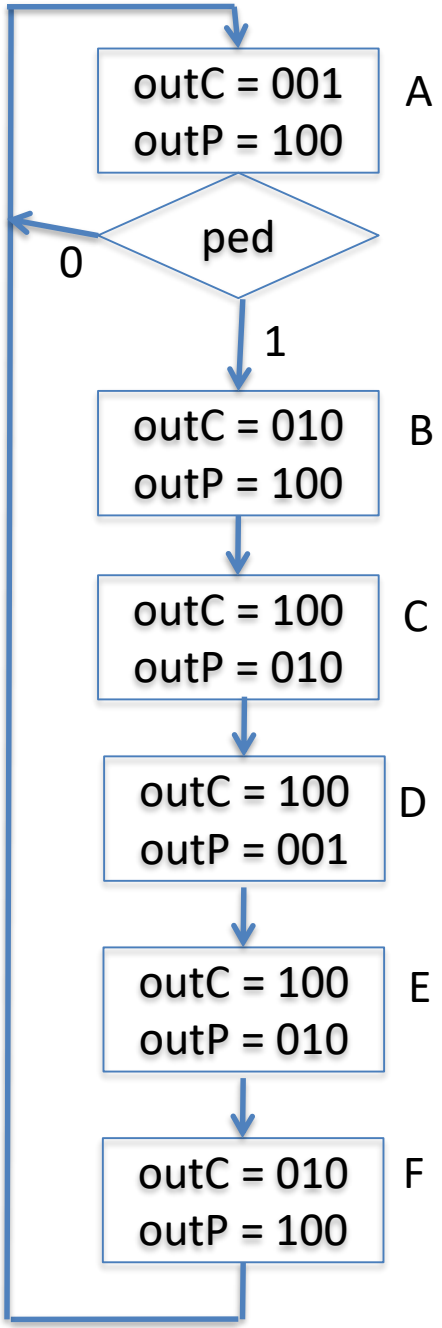
Output logic for cars



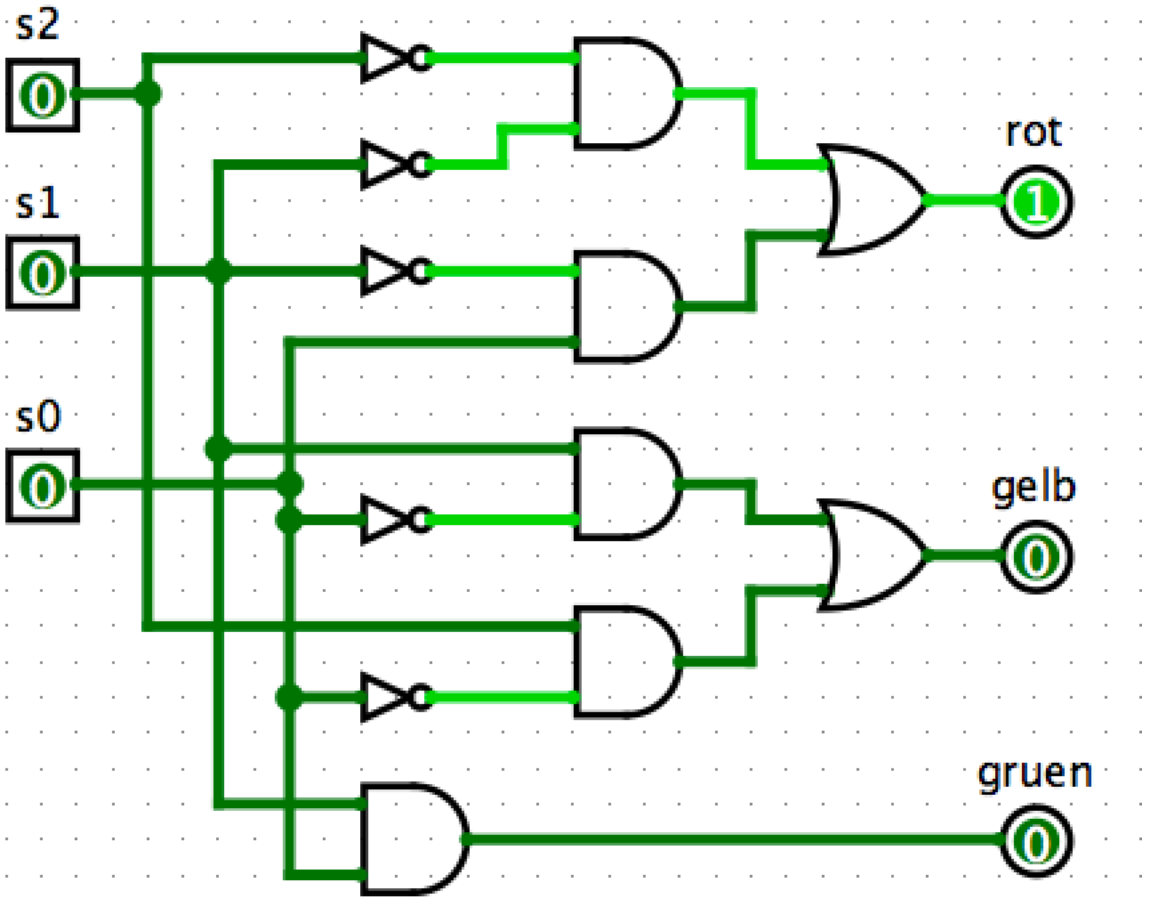


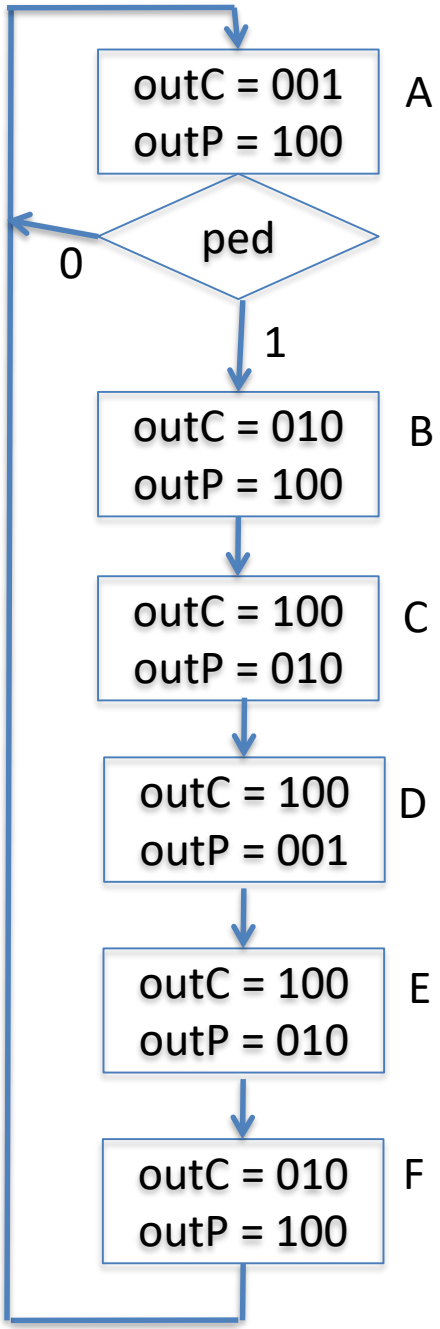
Output table for ped's

s2	s1	s0	red	yellow	green
0	0	0	1	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	x	x	x
1	1	1	x	x	x

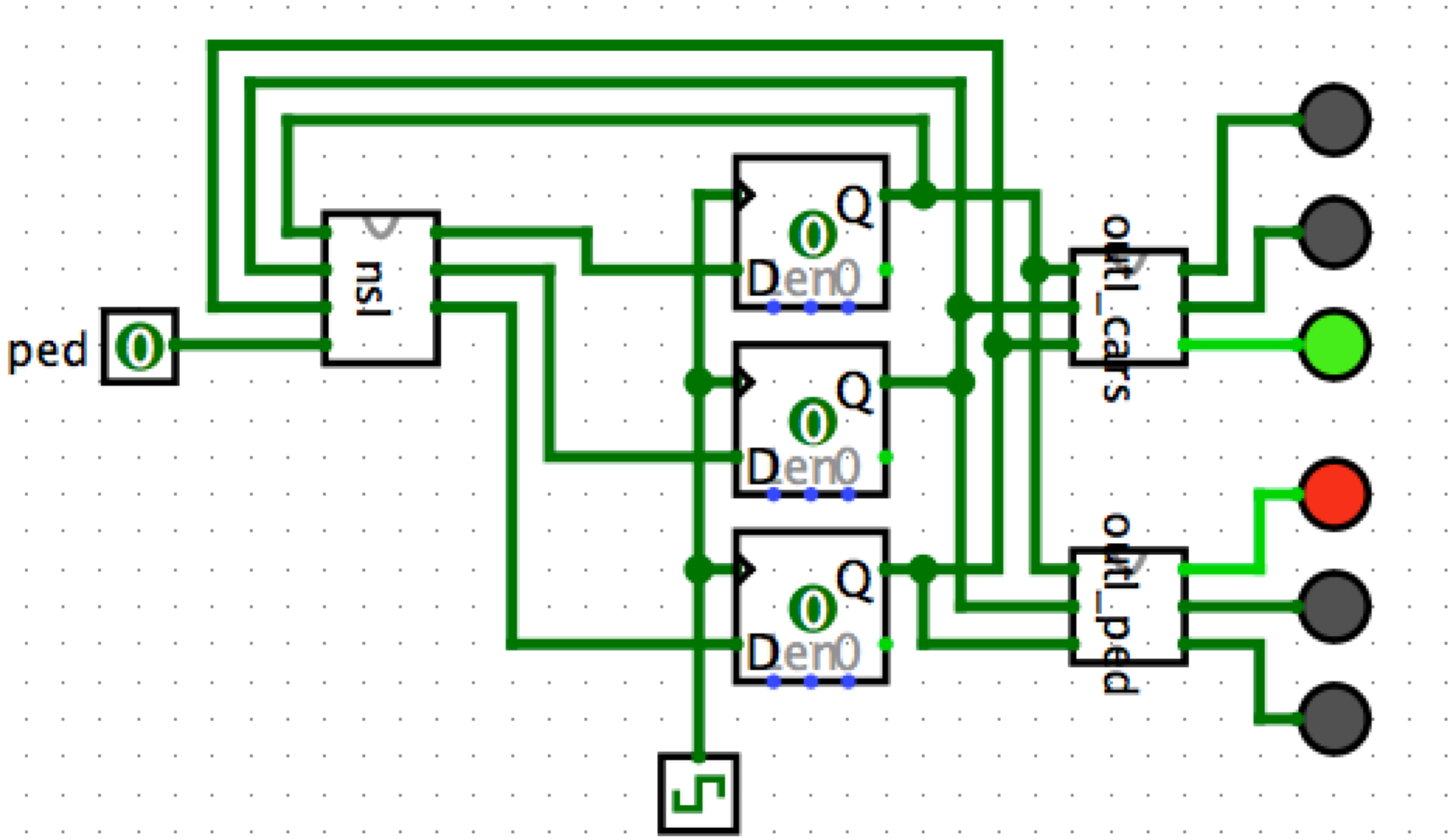


Output logic for ped's



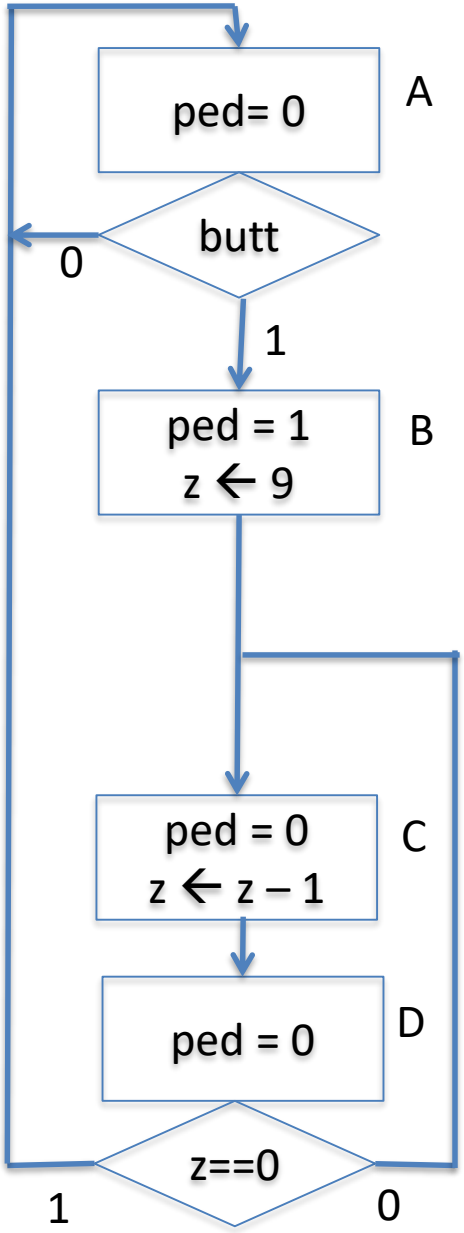


Traffic light (part 1)

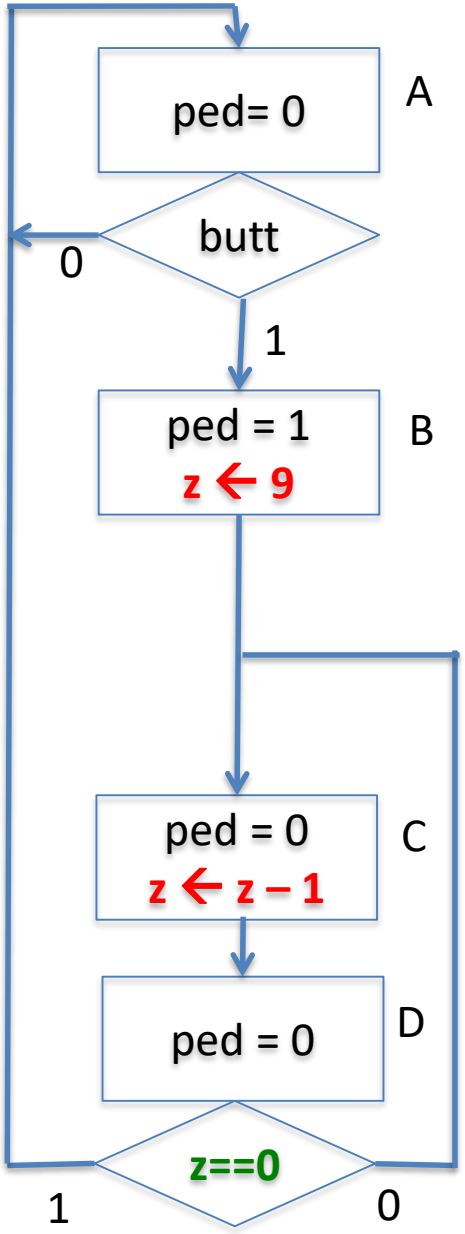


The problem

- A nasty pedestrian sticks a stick into the “ped”-button and leaves.
- Thus, the cars are always held by red, although there is no pedestrian.
- We want to avoid this “denial-of-service” attack.
- We add a second algorithmic state machine which tries to take care of this problem: After a pedestrian has hit the button and got green, there is a time delay before it can get green again.



Pedestrians hit “button”



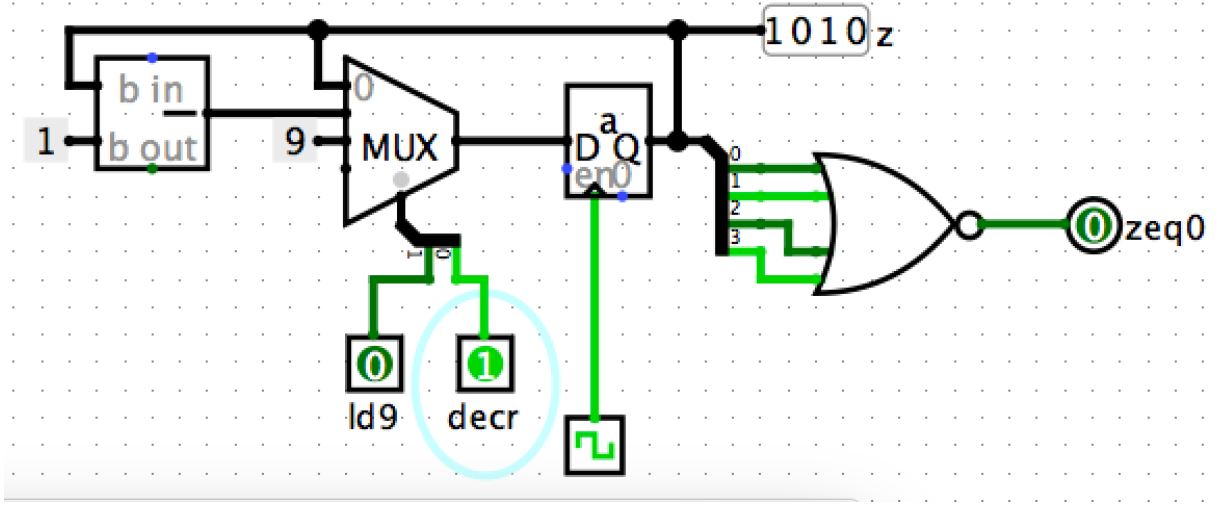
z ← 9

z ← z – 1

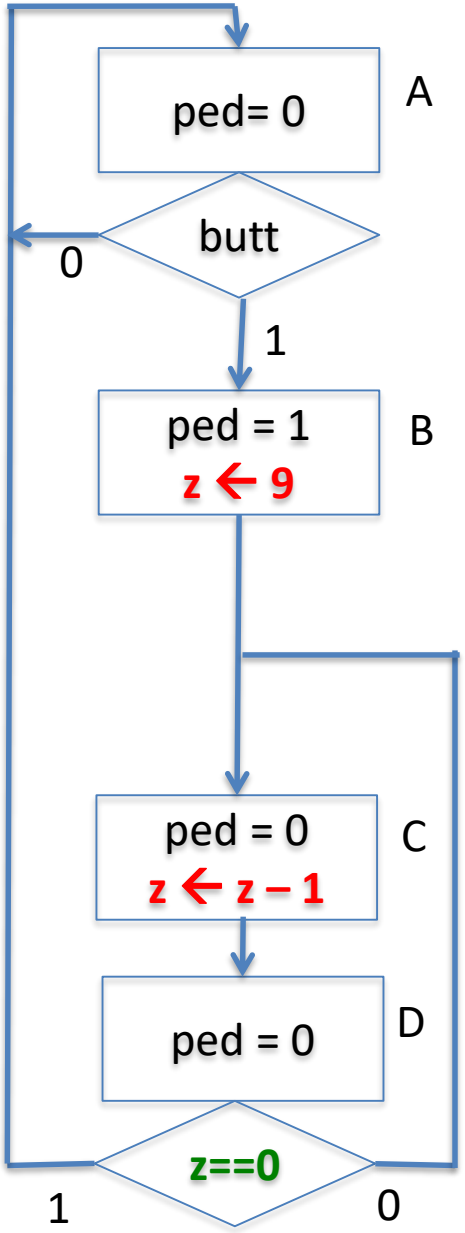
z ← z

z == 0

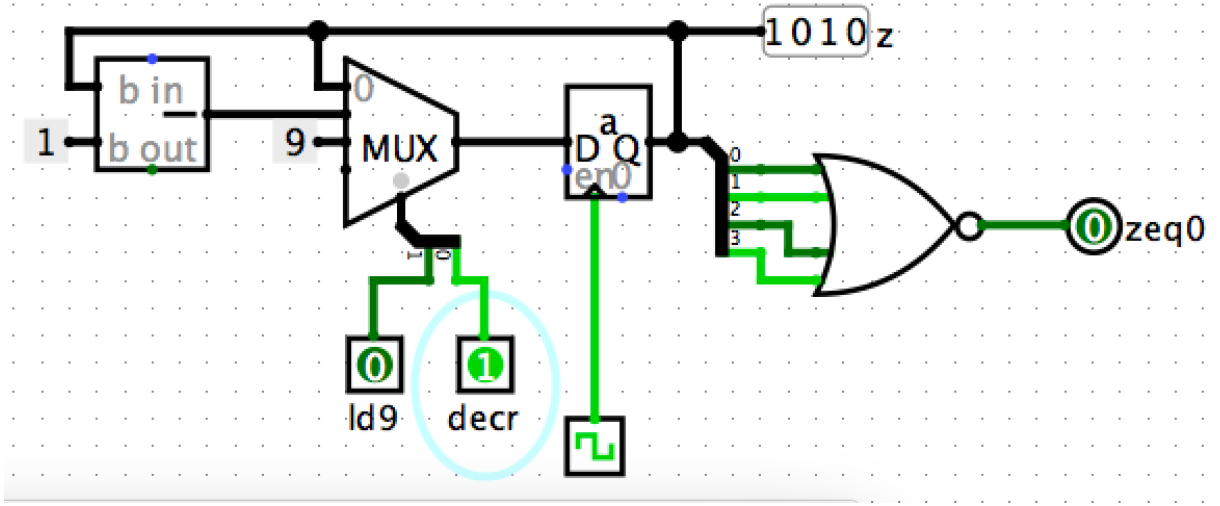
Data path

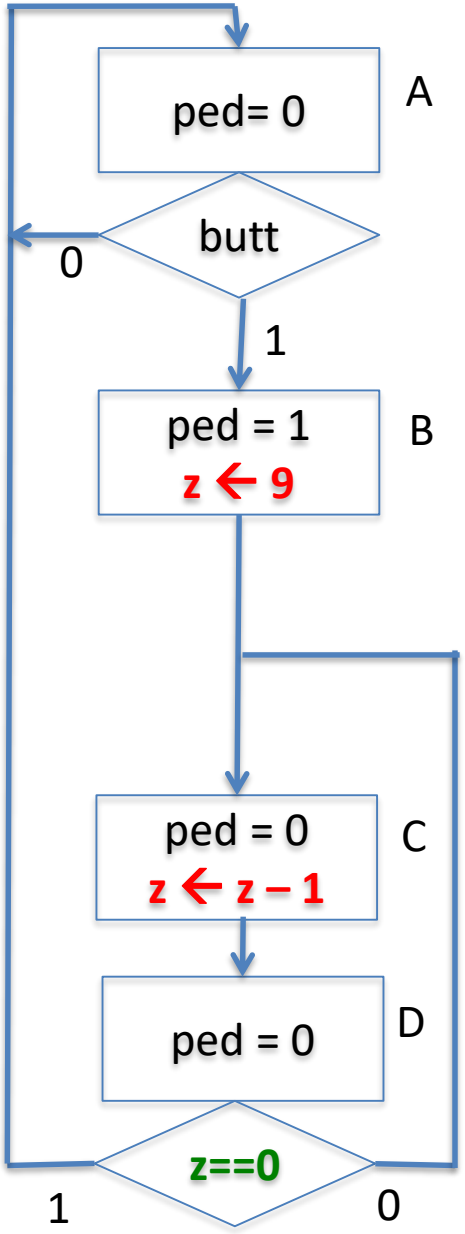


Multiplexer



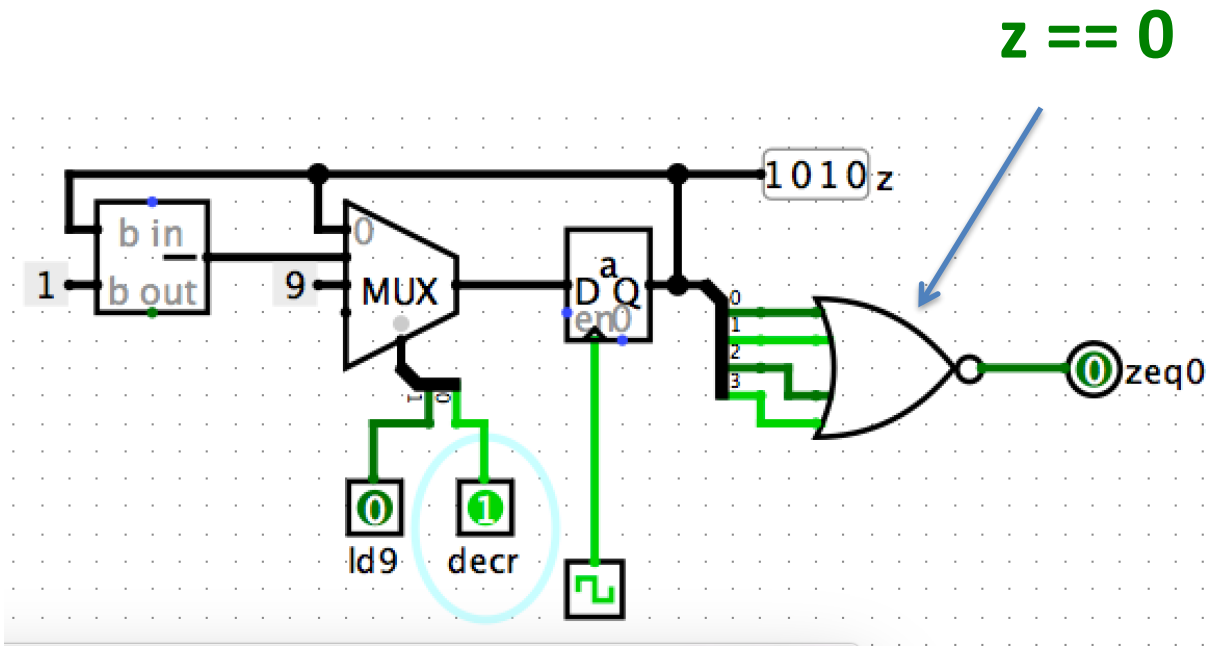
ld9	decr	
0	0	z ← z
0	1	z ← z – 1
1	0	z ← 9



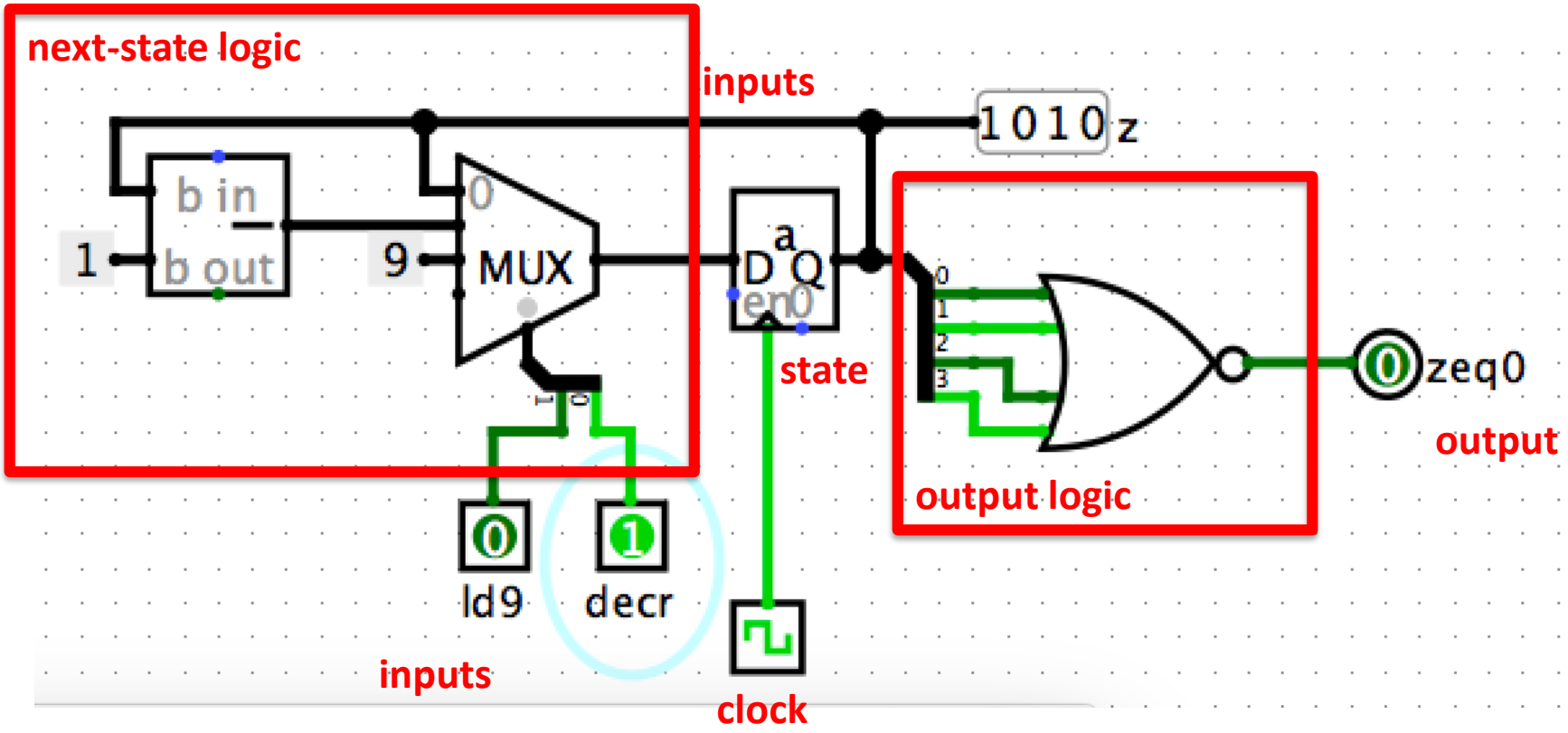


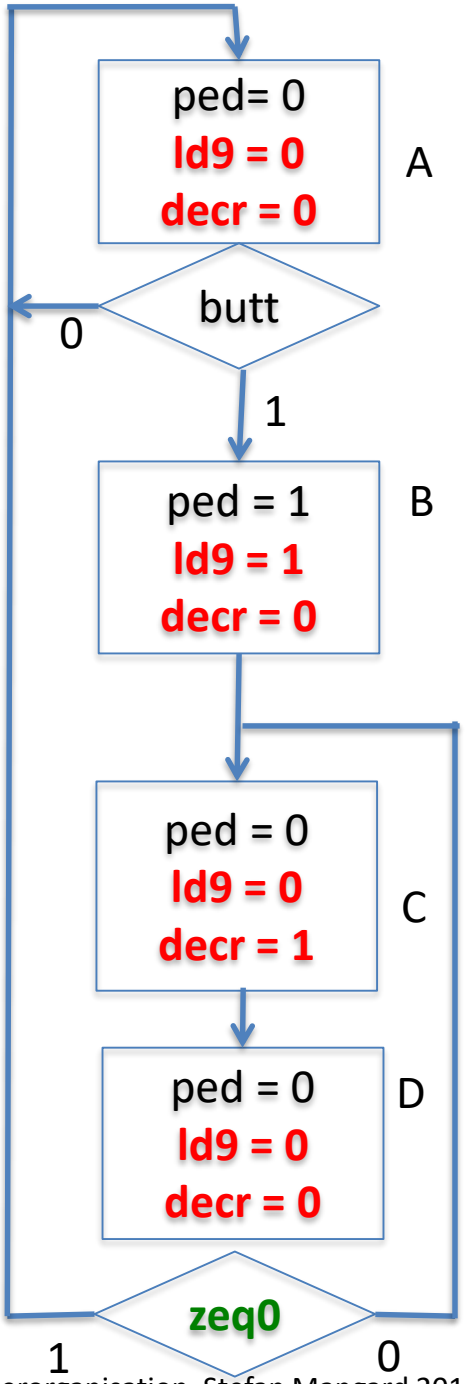
Multiplexer

ld9	decr	action at posedge of clk
0	0	z ← z
0	1	z ← z – 1
1	0	z ← 9



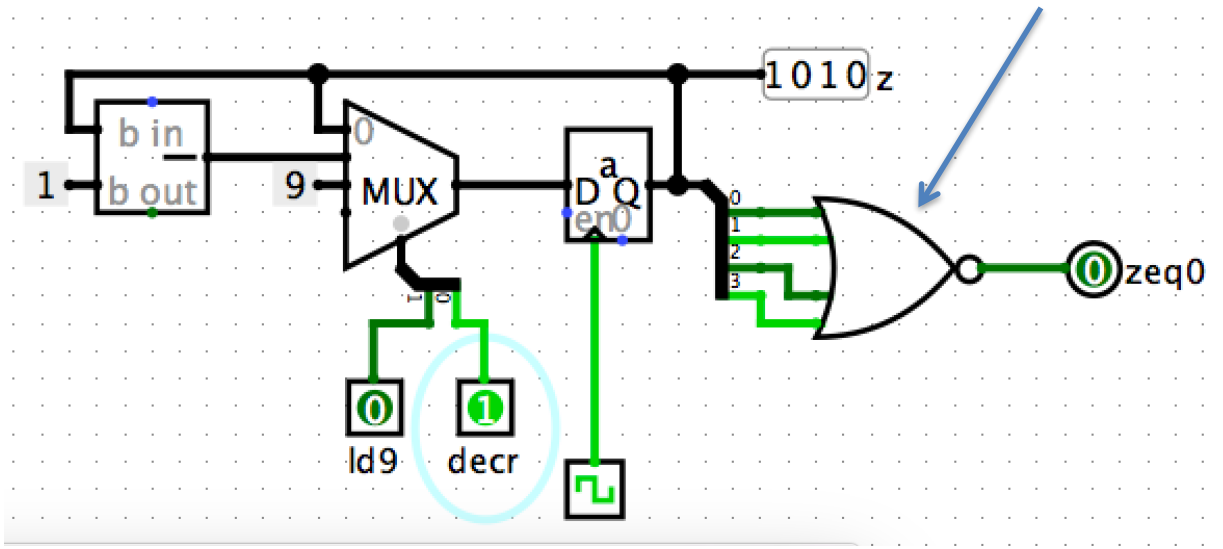
Data path is a Moore machine

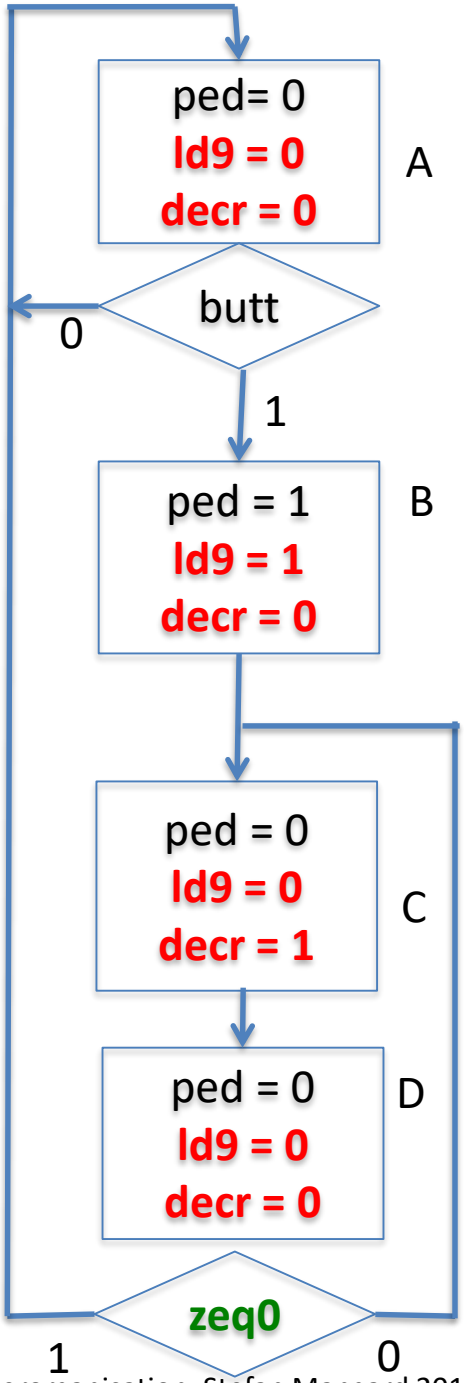




Changing the ASM

ld9	decr	action at posedge of clk
0	0	z ← z
0	1	z ← z – 1
1	0	z ← 9
1	1	not defined

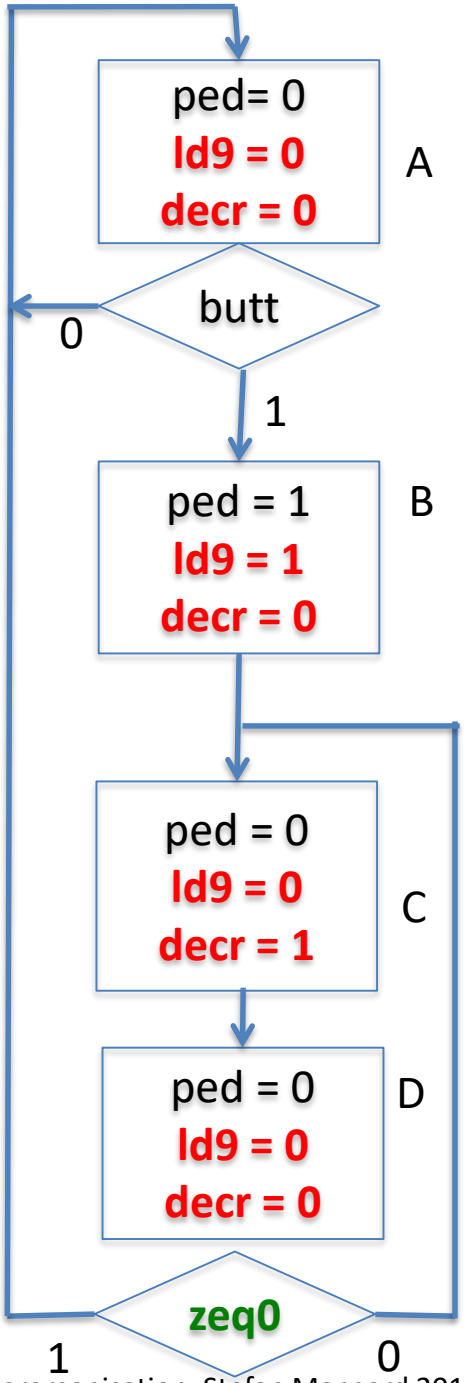




Next-state logic

state	butt	zeq0	next_state
A	0	x	A
A	1	x	B
B	x	x	C
C	x	x	D
D	x	0	C
D	x	1	A

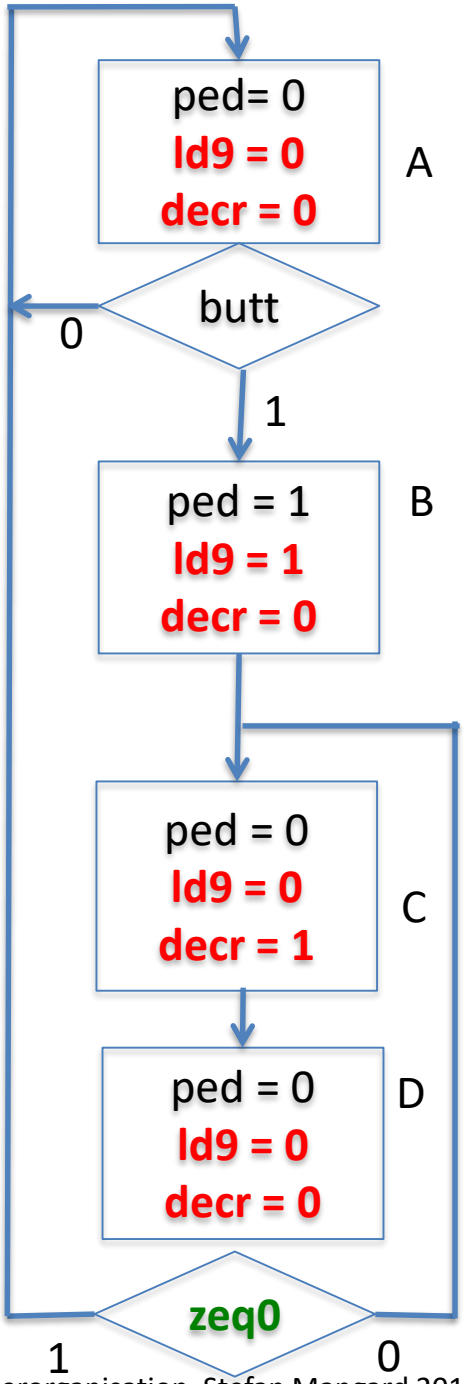
"x" stands for "don't care"



Next-state logic

t1	t0	butt	zeq0	nt1	nt0
0	0	0	x	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	0	x	x	1	1
1	1	x	0	1	0
1	1	x	1	0	0

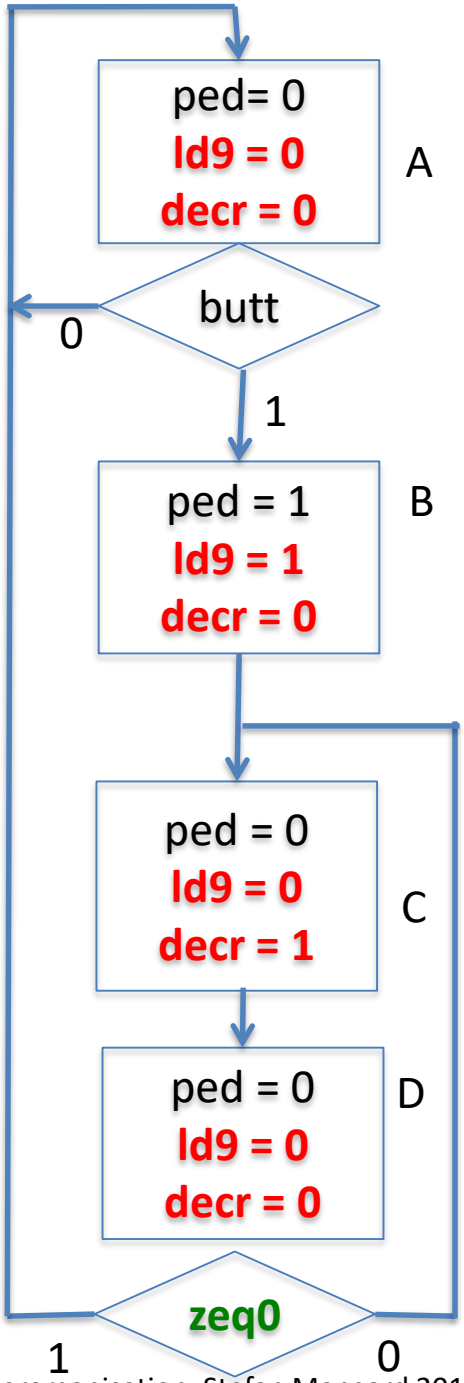
"x" stands for "don't care"

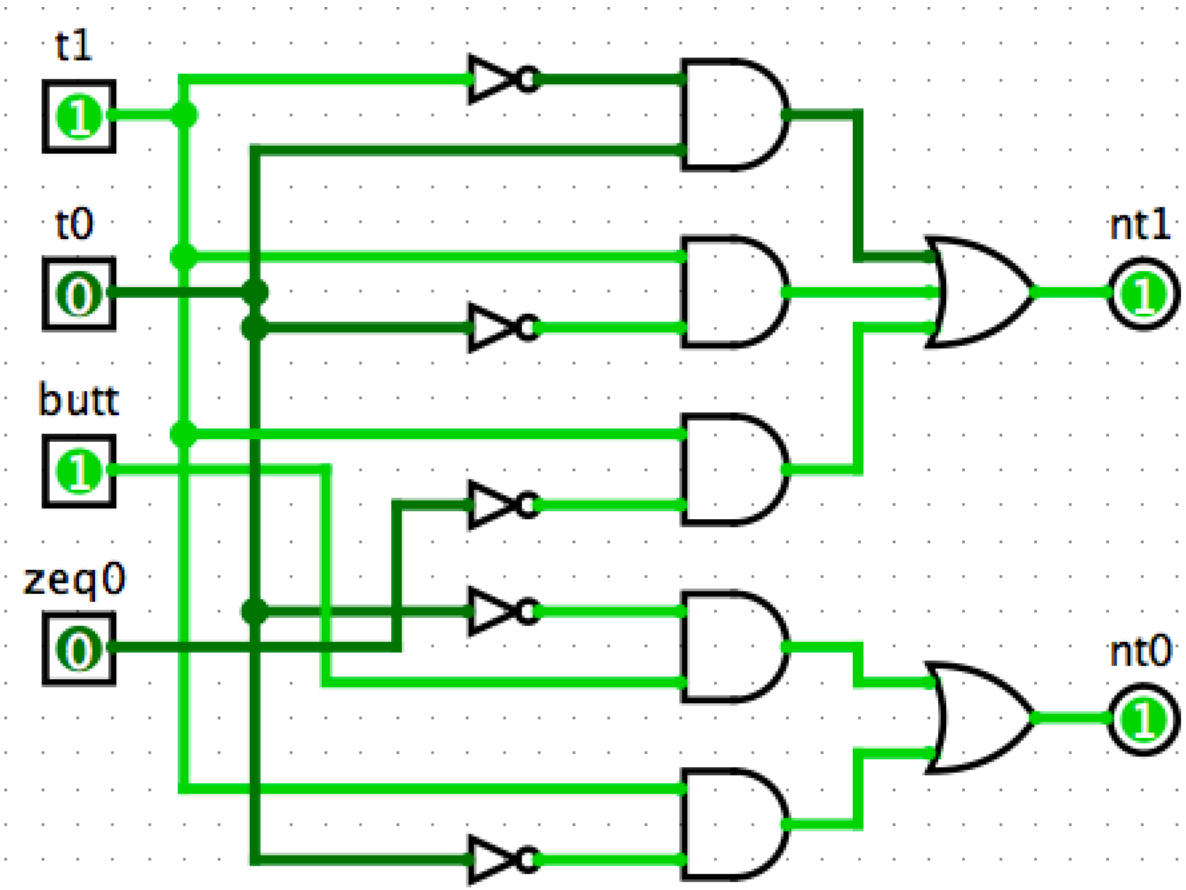


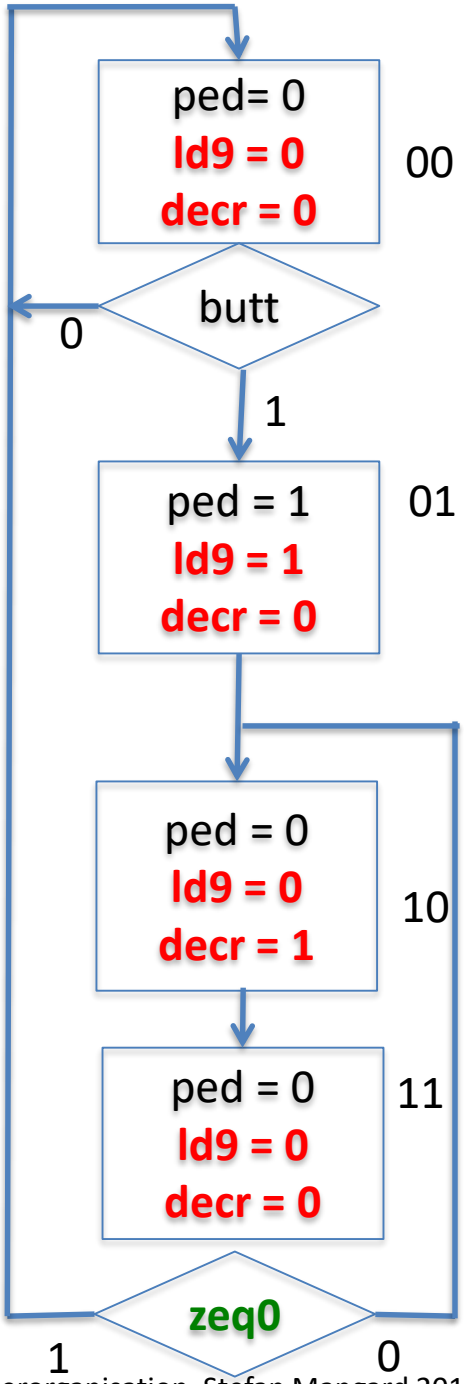
Next-state logic

t1	t0	butt	zeq0	nt1	nt0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	0

Next-state logic

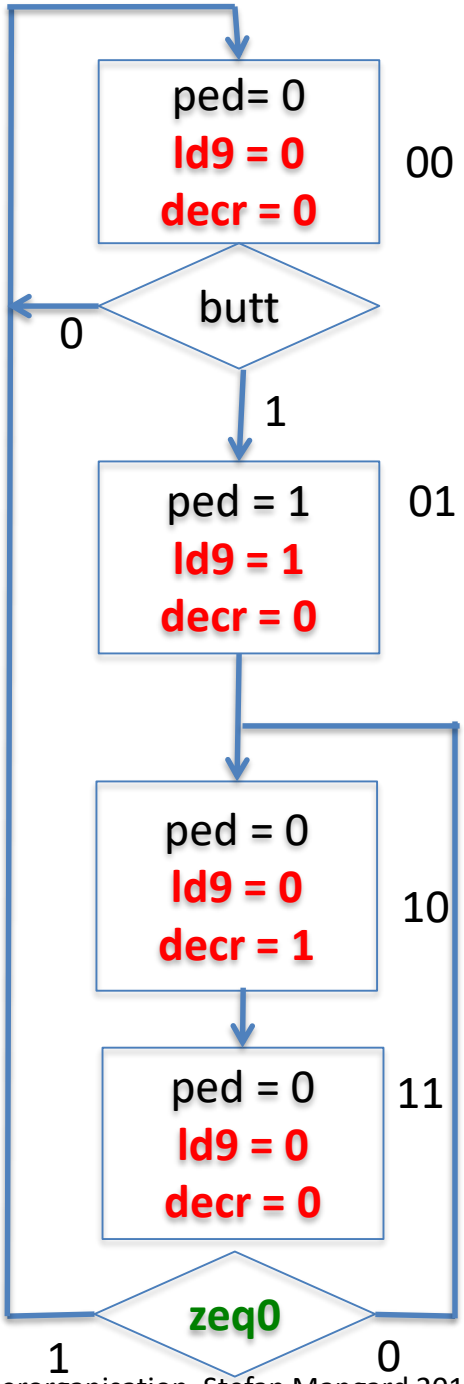






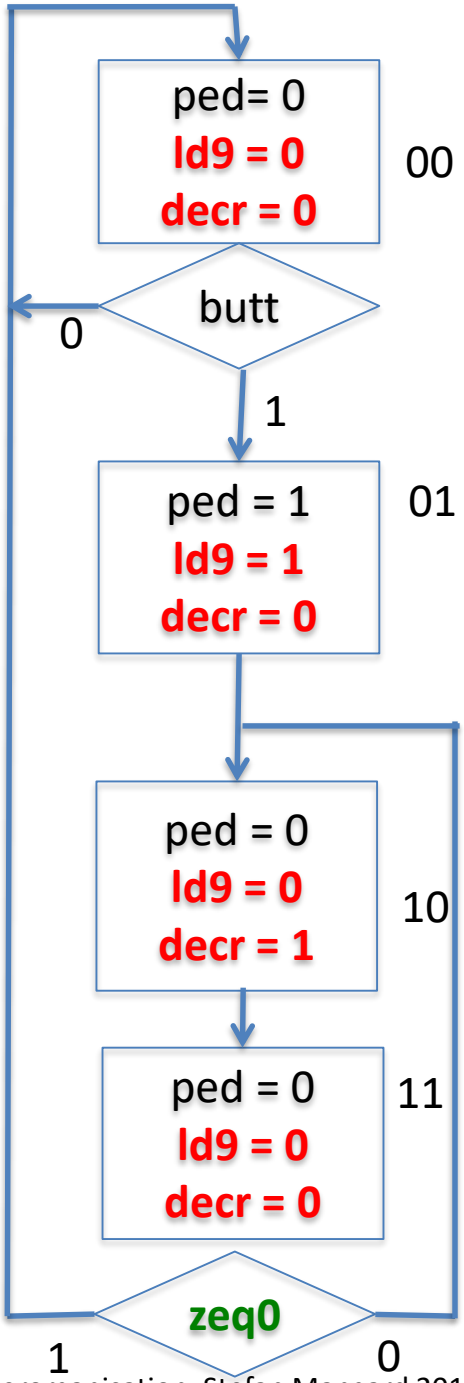
Output logic

t1	t0	ld9	decr	ped
0	0	0	0	0
0	1	1	0	1
1	0	0	1	0
1	1	0	0	0



Output logic

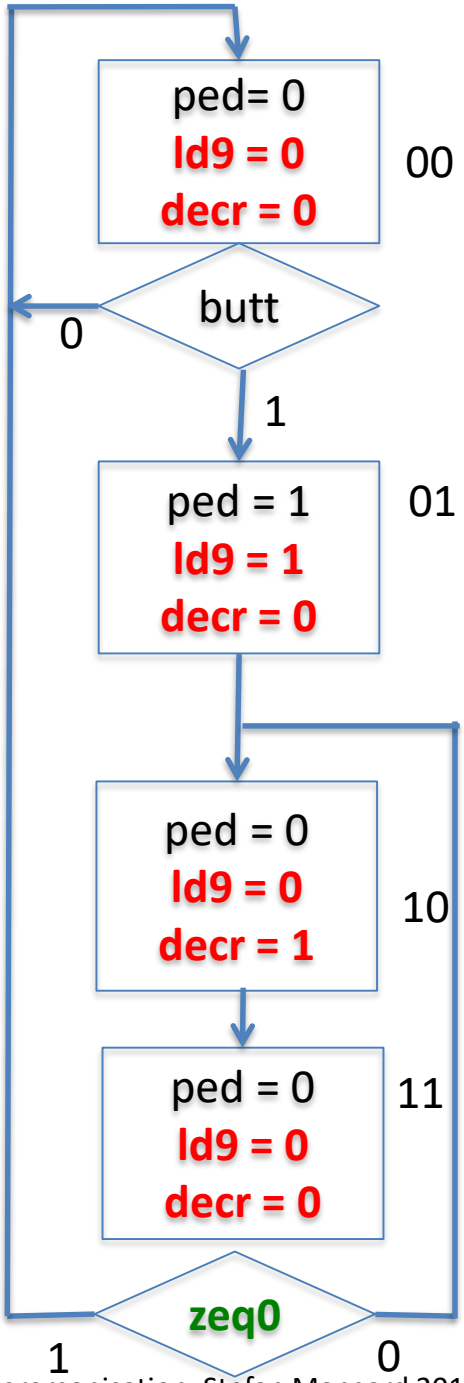
t1	t0	ld9	decr	ped
0	0	0	0	0
0	1	1	0	1
1	0	0	1	0
1	1	0	0	0

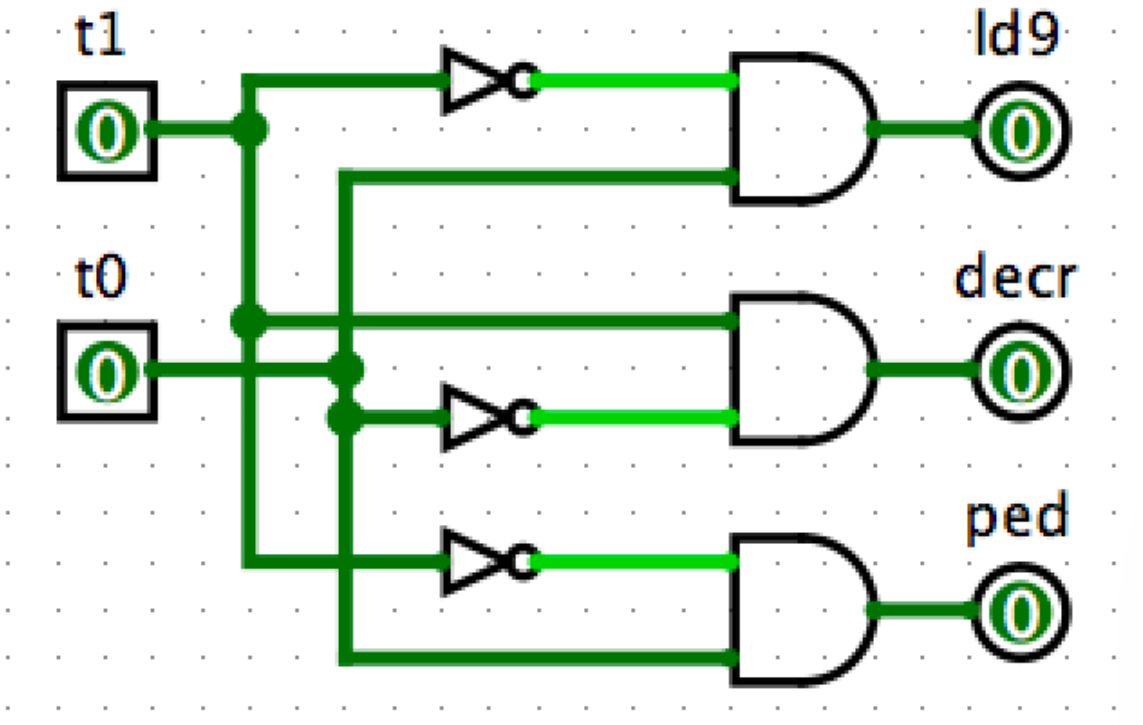


Output logic

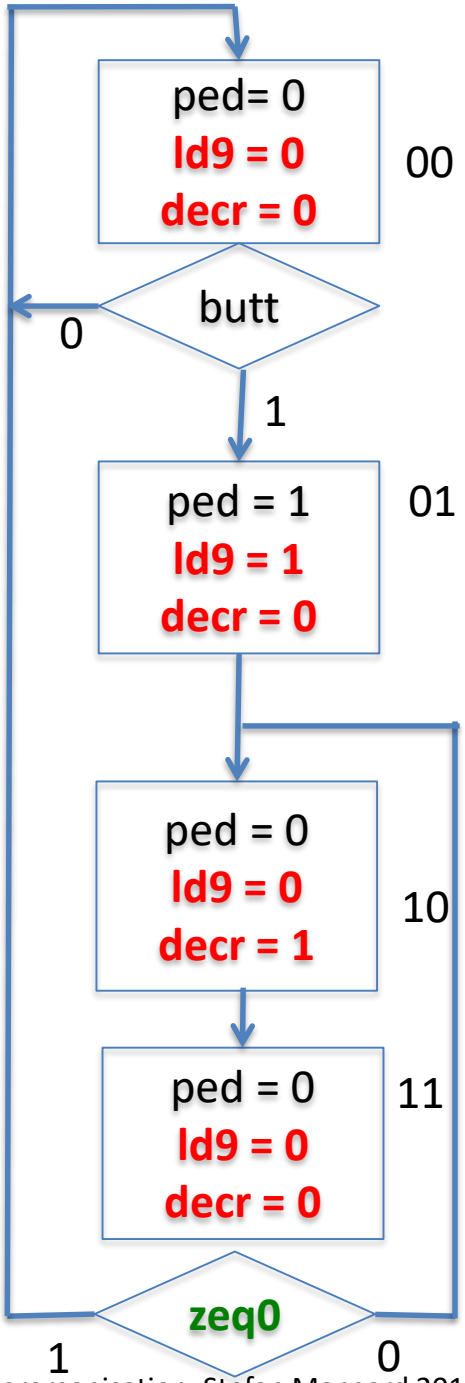
t1	t0	ld9	decr	ped
0	0	0	0	0
0	1	1	0	1
1	0	0	1	0
1	1	0	0	0

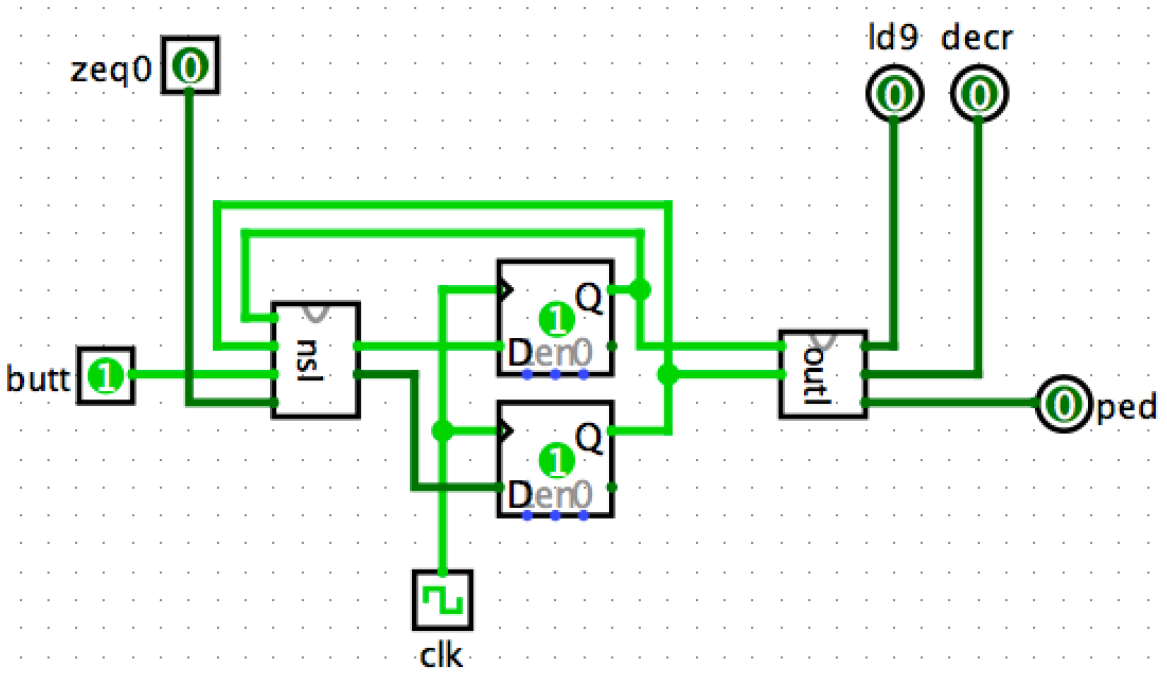
Output logic





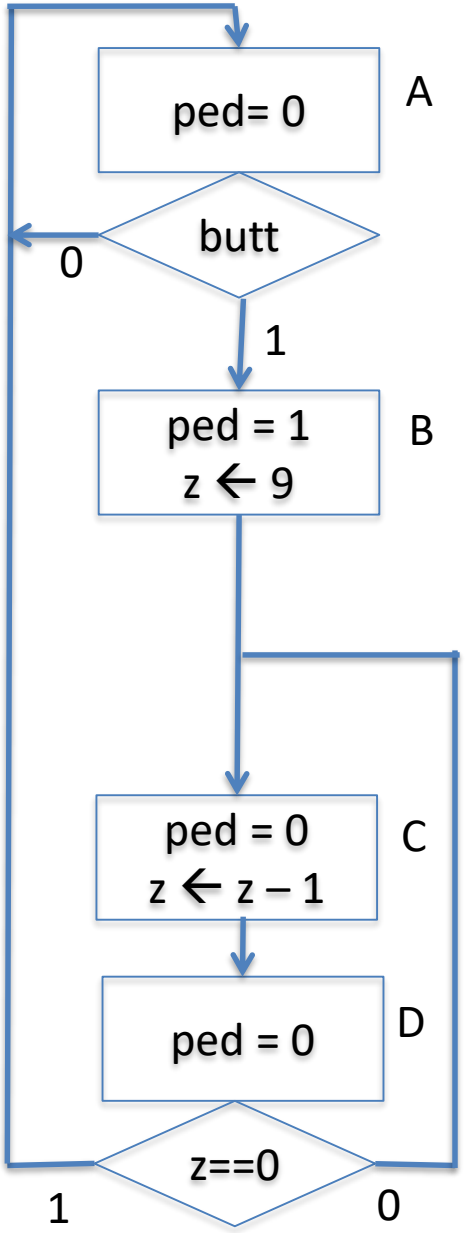
The controller

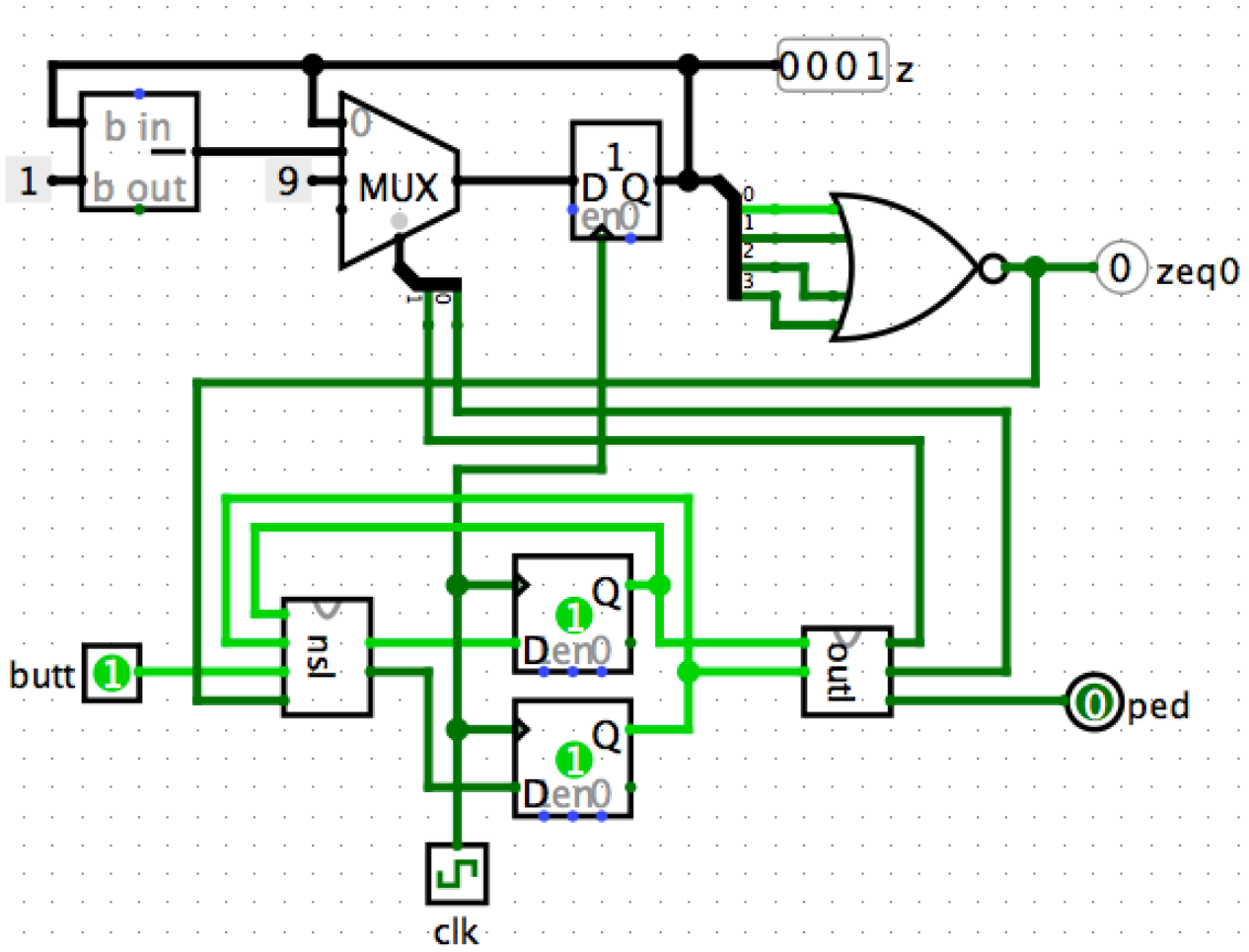




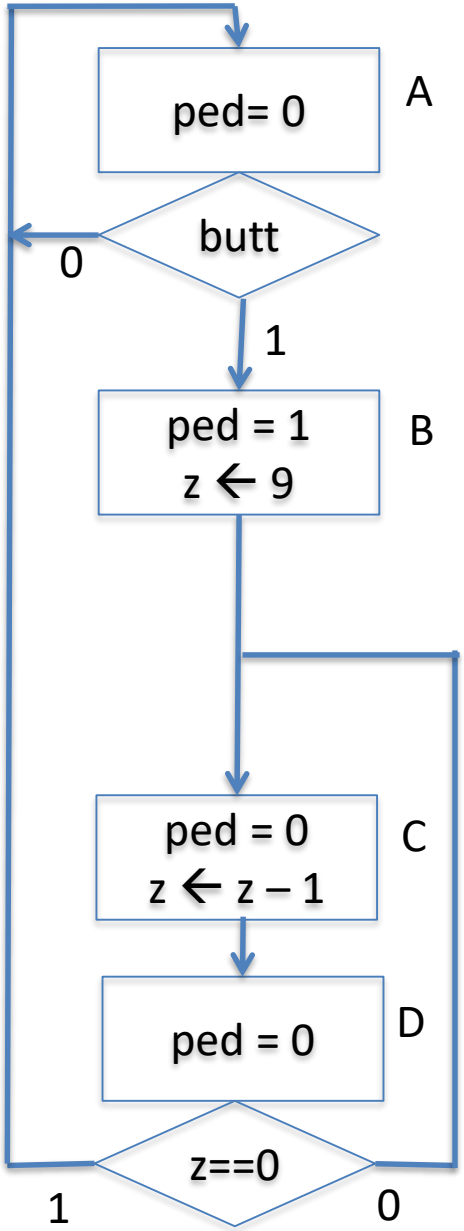
The controller is a Moore machine.

Data path + controller

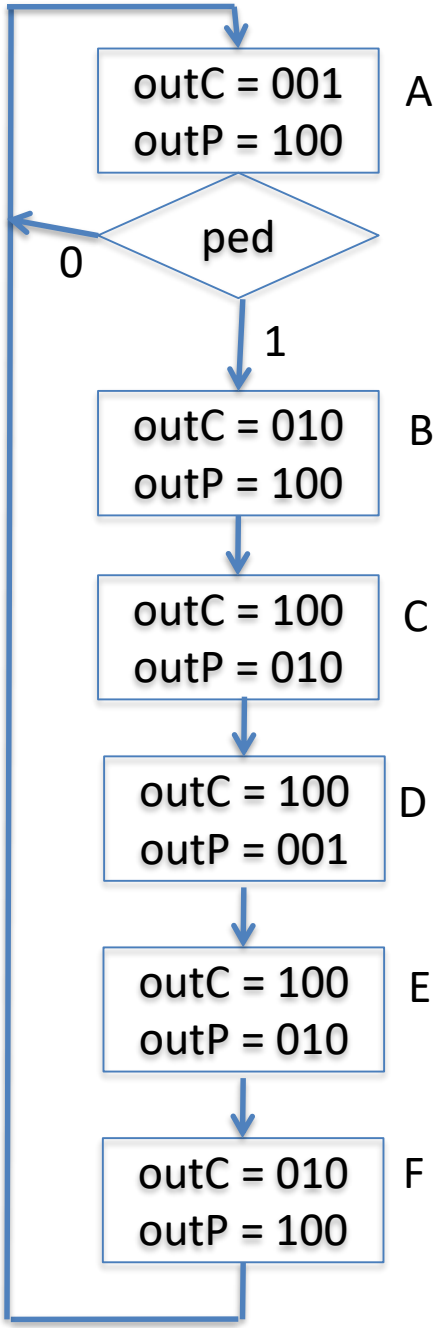




Both ASMs together



ped



City of tiny lights