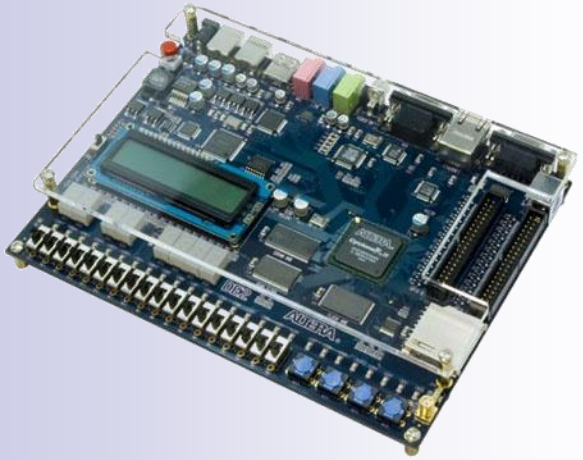
Logic Systems and Processors

cz: Logické systémy a procesory

Lecture 1: Introduction to Logic Circuit Design



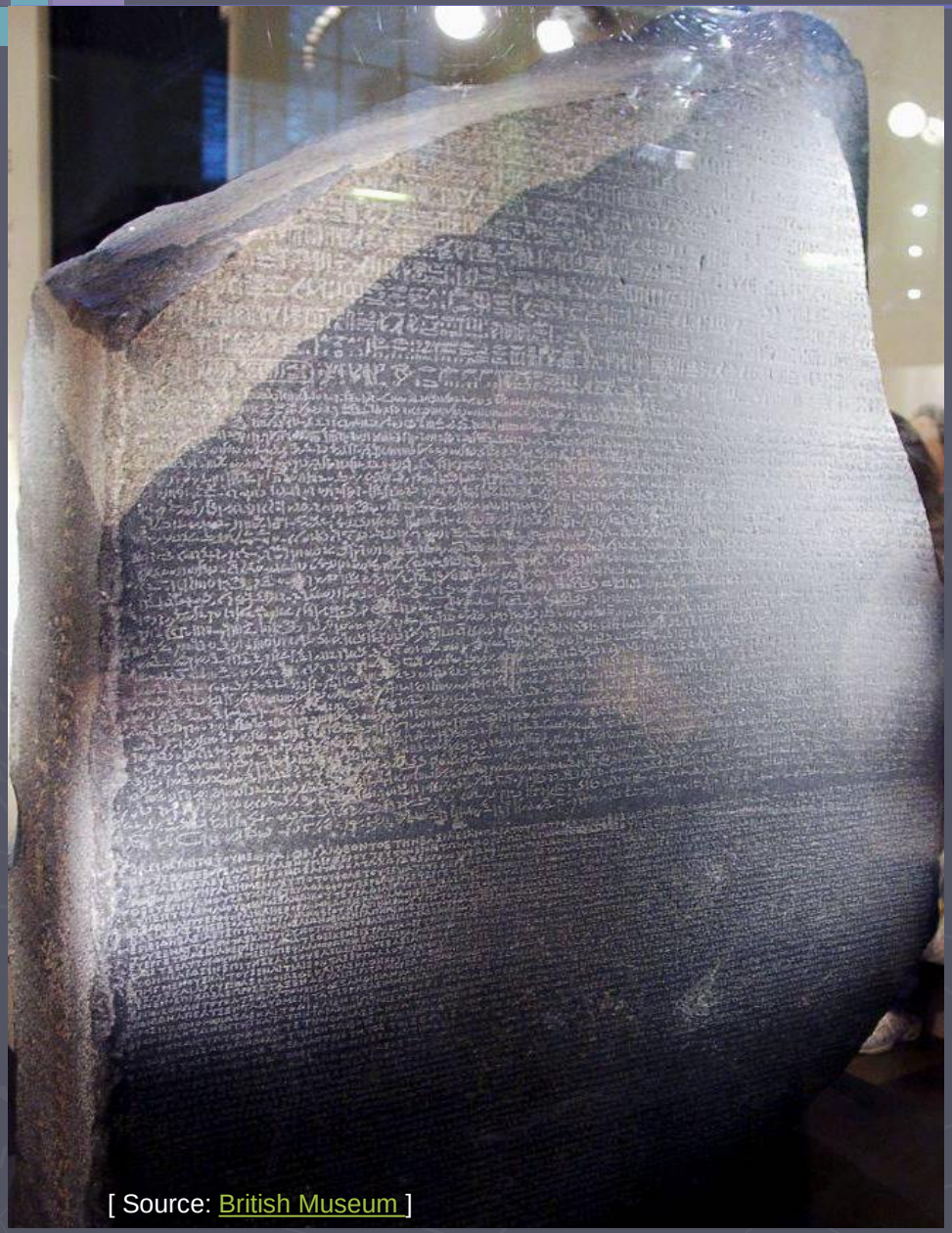
Lecturer: Richard Šusta

susta@fel.cvut.cz, richard@susta.cz

+420 2 2435 **7359**

Version: 1.1

Notes



[Source: British Museum]

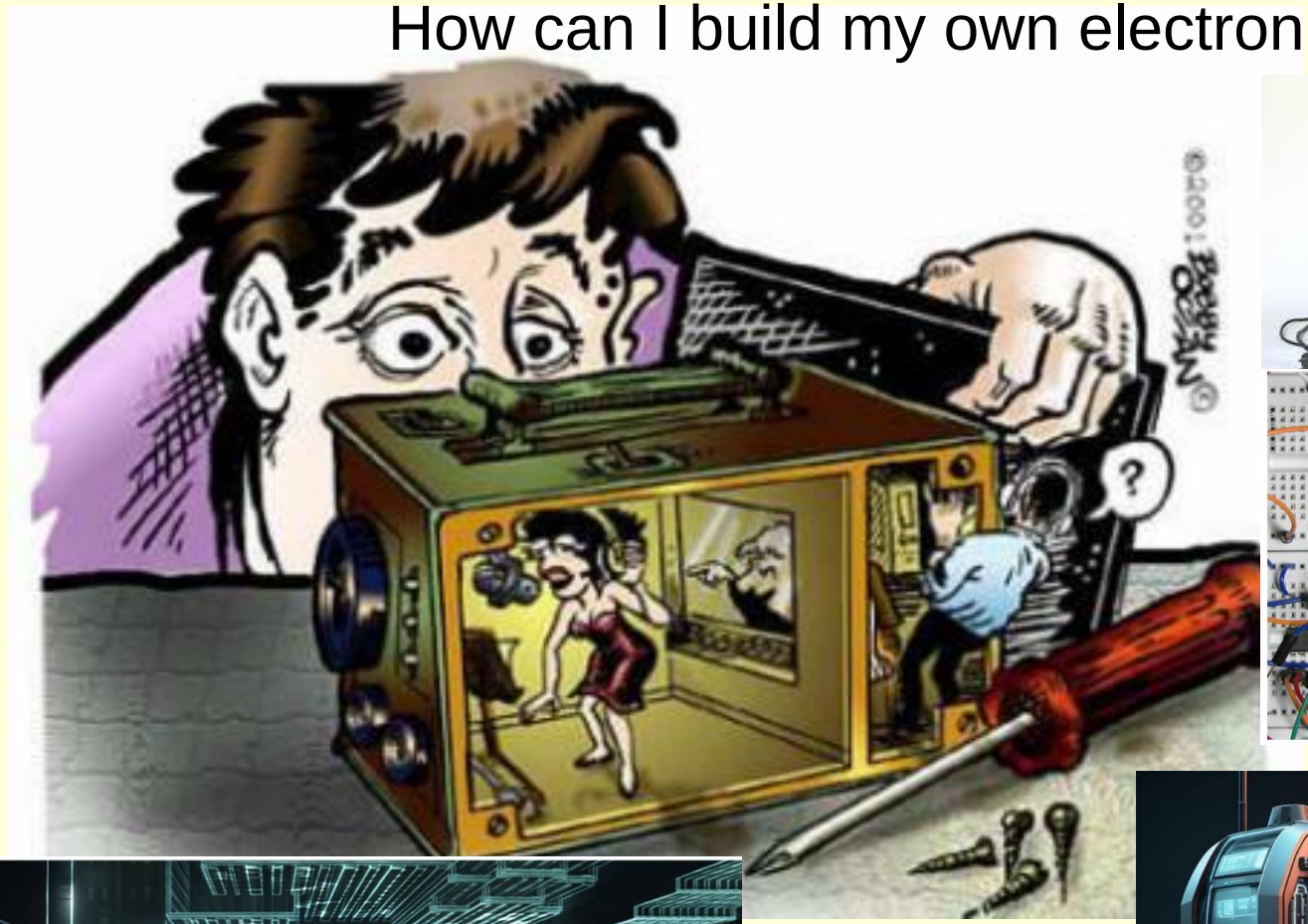
The 3S image specifies
a **'Self-Study Slide',**
which summarizes or expands
the previous explanations.
It was usually
skipped in lectures.

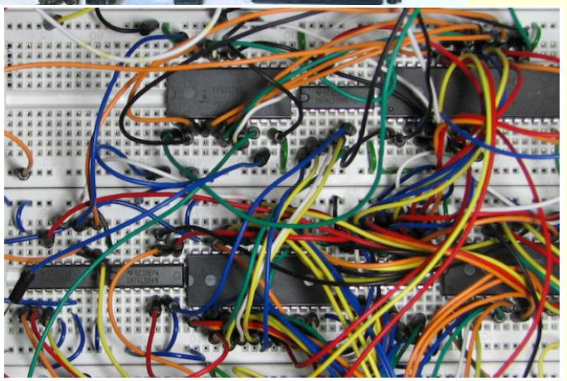
The arrow at the bottom edge
is merely a teacher's mark that
animations still
need to be finished.

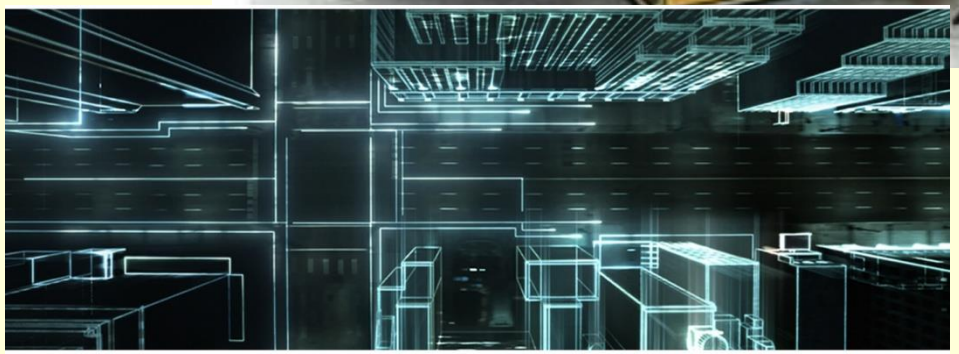
Questions: What lies inside of technical miracles?

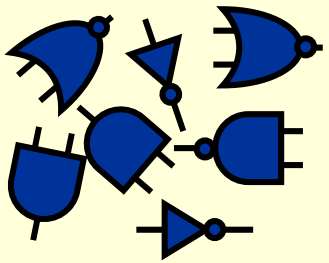
What are their most important parameters?

How can I build my own electronic miracle?



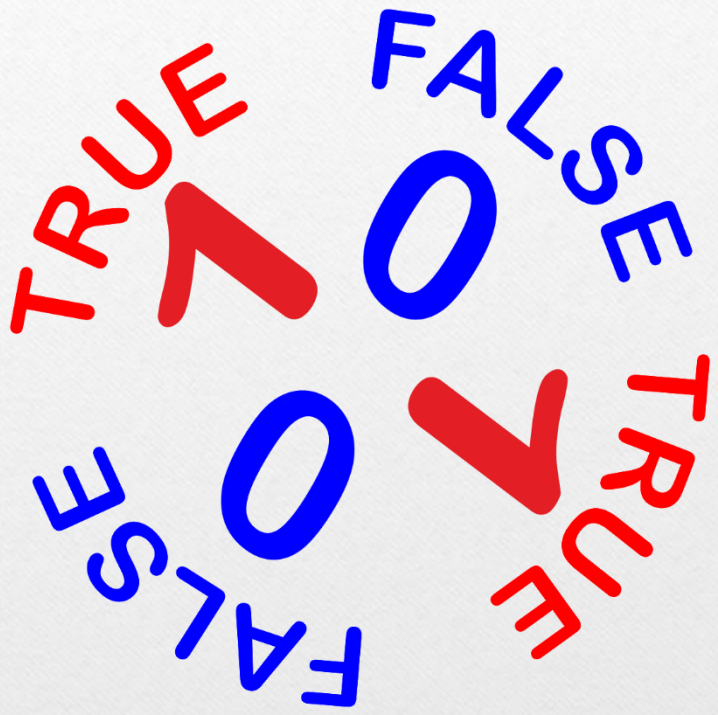




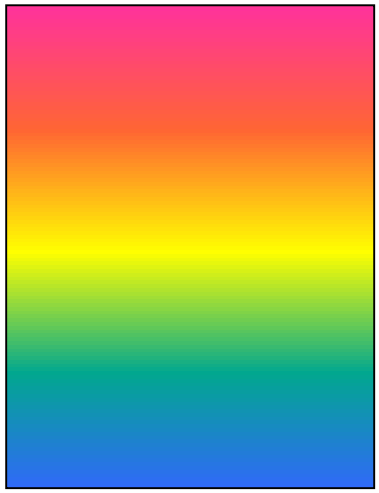


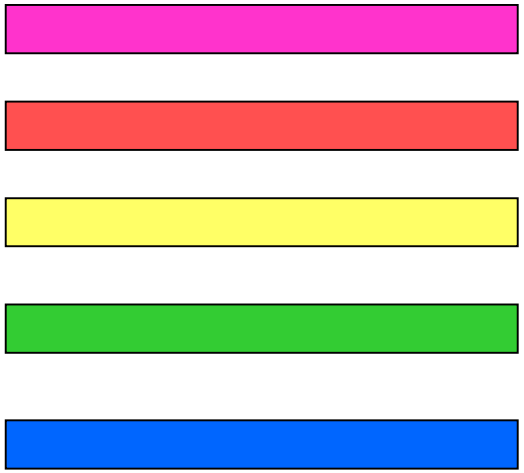
[Pictures sources: radio411.com, freepik.com, apple.com]

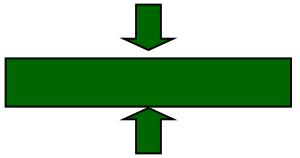
ABOUT VALUES



Data

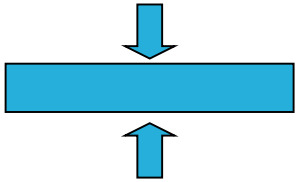




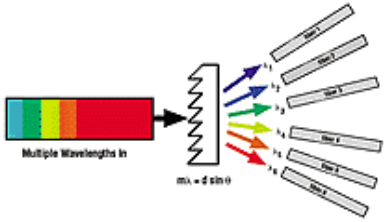


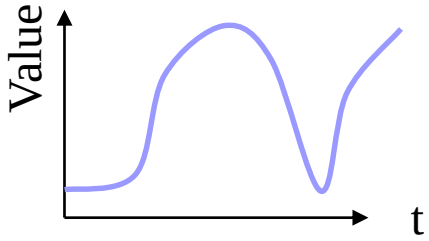
'1', true

?

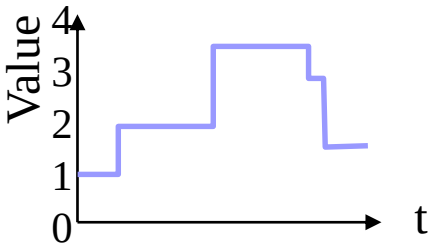


'0', false

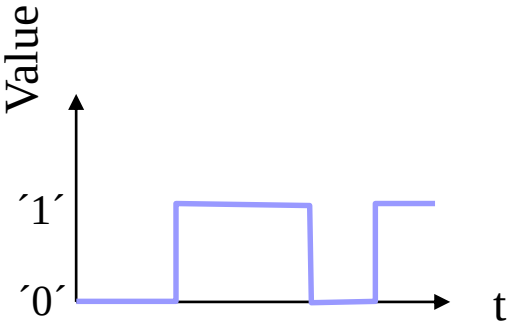




*** analog signal**
transmits theoretically infinitely many values



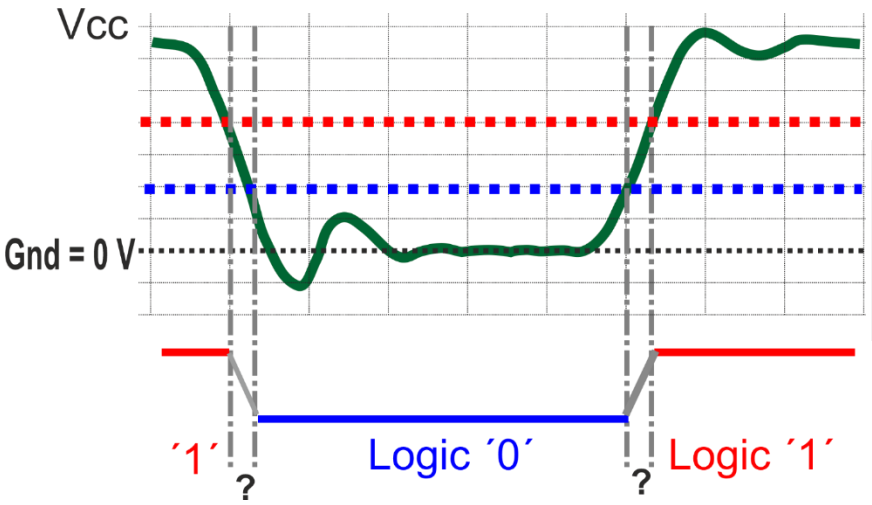
*** digital signal**
sends limited values



*** binary signal**
was reduced to two logical values

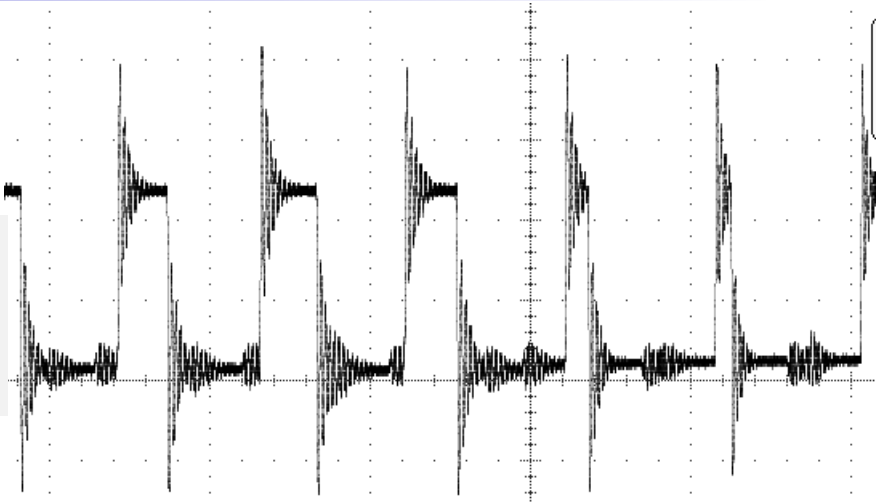
Oscillograph of a Logical Signal

Positive Voltage Logic



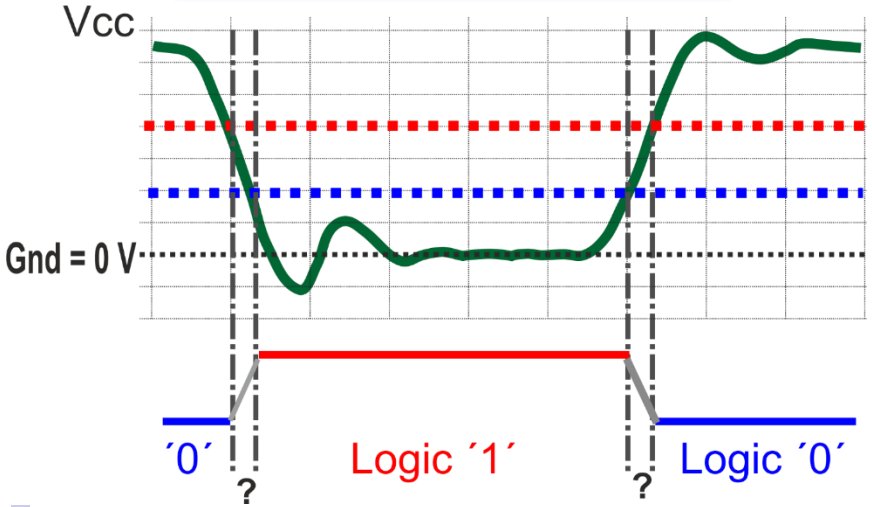
+ noise

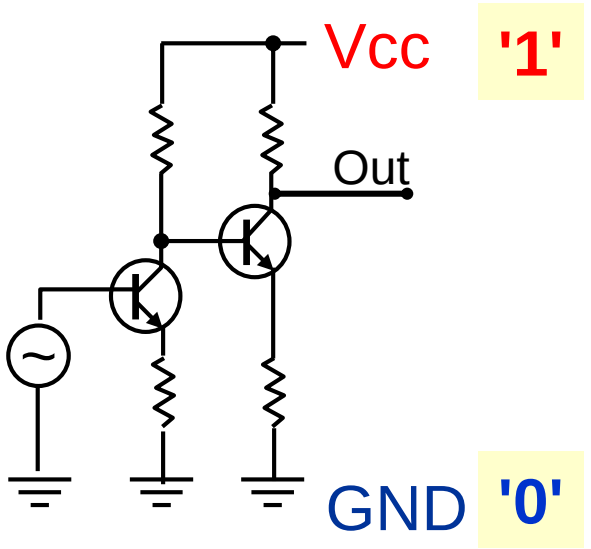
→



[Measured binary output of audio player, source Texas Instruments]

Negative Voltage Logic





[Acronyms are explained in Logic circuits on FPGA, page 22]

Logic '0' and '1'

- They are reducing the considered values to simplify designs.
- They do not refer to a physical quantity. They can be realized not only by voltage levels but also by current, signal phase, pulses, etc. Thus, Abstract '0' and '1' merely encode properties of real values;
- Their mapping to physical values is up to a designer.
- Switches can only realize a small subset of logic equations, but they do not create every logic function.
- Many different Vcc power supply voltages are used nowadays. The older TTL technology had **Vcc=5 V**. Today, we can find even **0.6 V** for 3 nm technology or **1.2 V** for 60 nm FPGA on our VEEK-MT2 board, but the industry frequently uses **+12 V** ('0') and **-12 V** ('1') logic.

Logic circuits

can include

Combinational circuits

their outputs are determined only by immediate values on their inputs

Sequential circuits

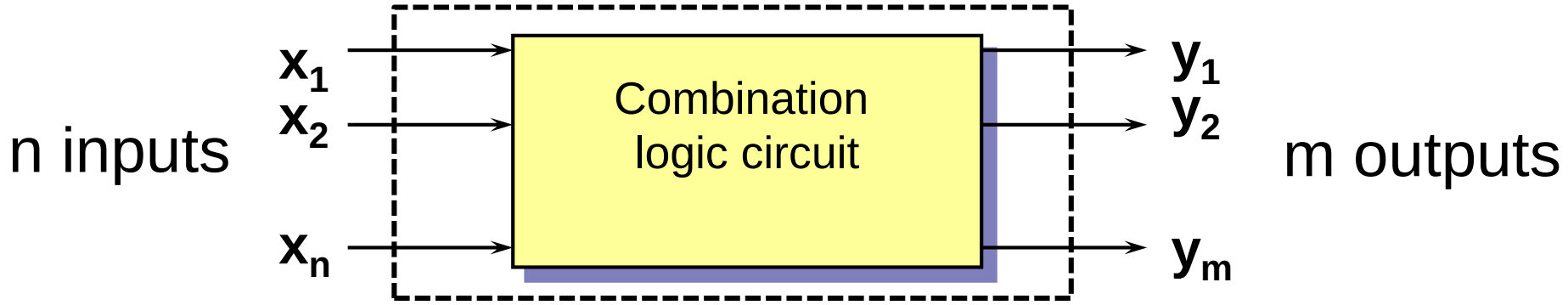
their output depends on a sequence of their input values

Combinational Circuit

aka Combinational Logic Circuit

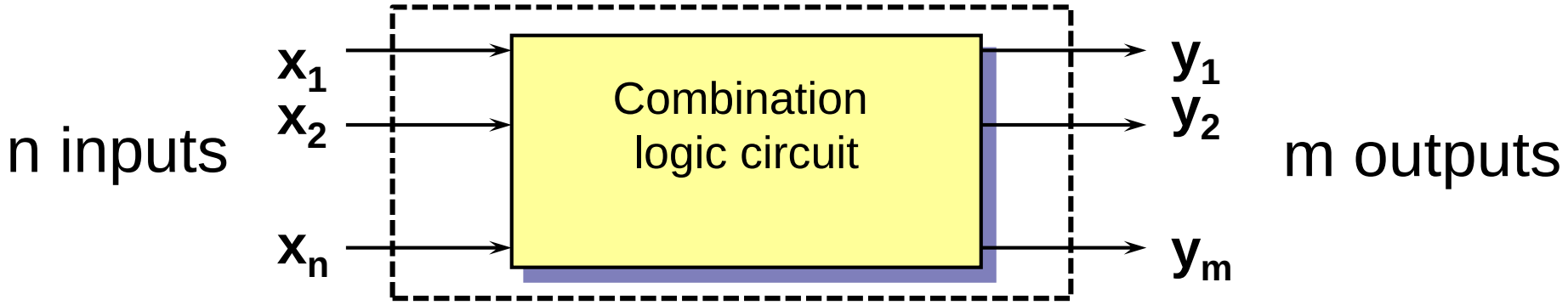
Necessary condition: **Its m>0 logic outputs depend only on the values of n>=0 inputs.** *In other words:*

After all circuit transients caused by input change are over, the same input values always result in the same output values.



Combinational Circuit

aka Combinational Logic Circuit



We can describe any combinational circuit as the collection of **m** logic functions:

$$\mathbf{y_1} = f_1(x_1, x_2, \ldots x_n) \quad \ldots \quad \mathbf{y_m} = f_m(x_1, x_2, \ldots x_n)$$

However, some its functions can certainly depend only on subsets of all possible inputs x_1 to x_n, e.g., constant logic function GND $(x_1, x_2, \ldots x_n)$ gives '0' value regardless of its input.

Example

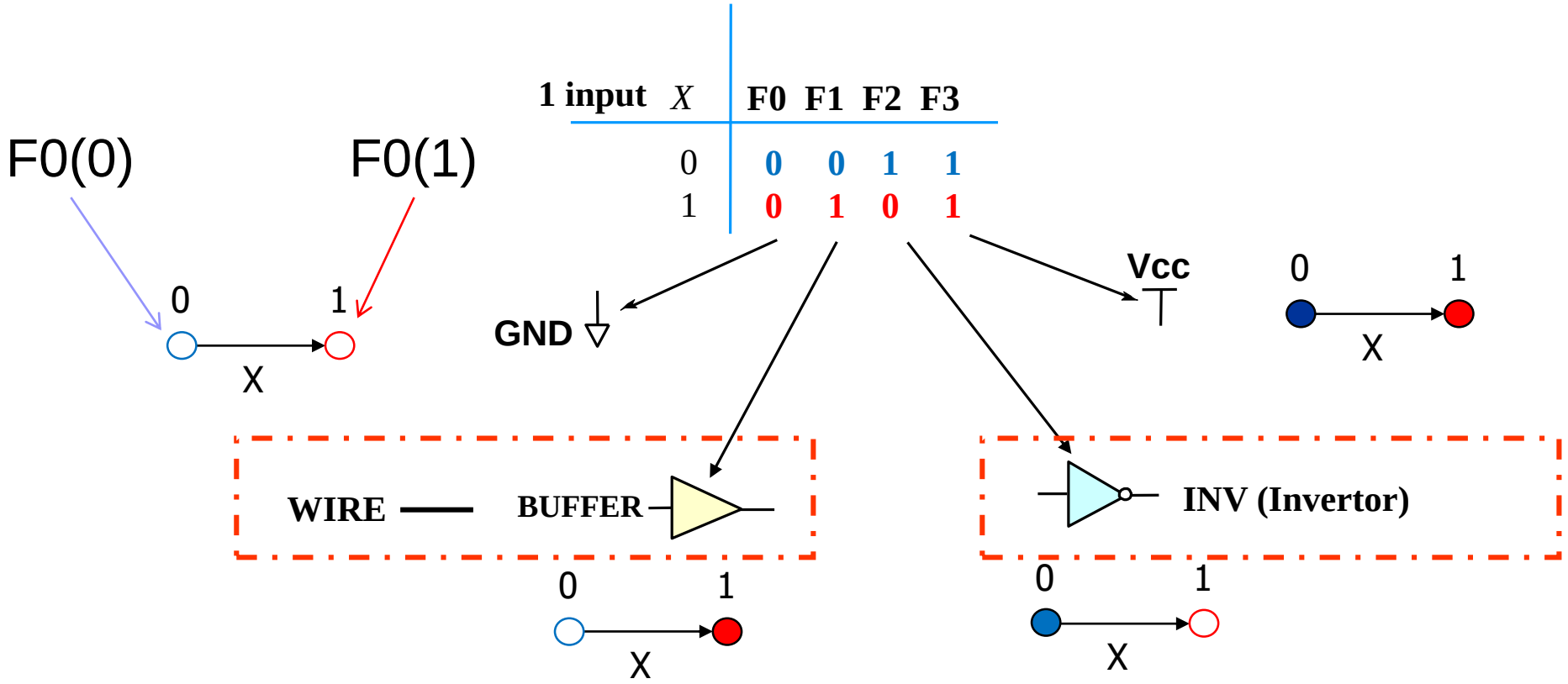
We write the function of the two inputs, n = 2, as $y = f(x_1 , x_2)$,
Here, $\mathbf{B^2}$ contains four pairs, i.e., $\mathbf{B^2}$ = { (0, 0), (0, 1), (1, 0), (1, 1) }
For n=2, there are $2^{2^2} = 2^4 = 16$ *possible assignments on defining the function. Let's choose one of them - perhaps the one called* **xor** (exclusive-or) *or non-equivalency - that has its output in logic '1' only when the inputs are mutually different.*

We define our function $y=f(x_1 , x_2)$ by the following prescription:

xor: $\mathbf{B^2 \rightarrow B}$ =

	simplified notation	
(0, 0) → **0**		0 0 → **0**
(0, 1) → **1**		0 1 → **1**
(1, 0) → **1**		1 0 → **1**
(1, 1) → **0**		1 1 → **0**

All functions with 1 input

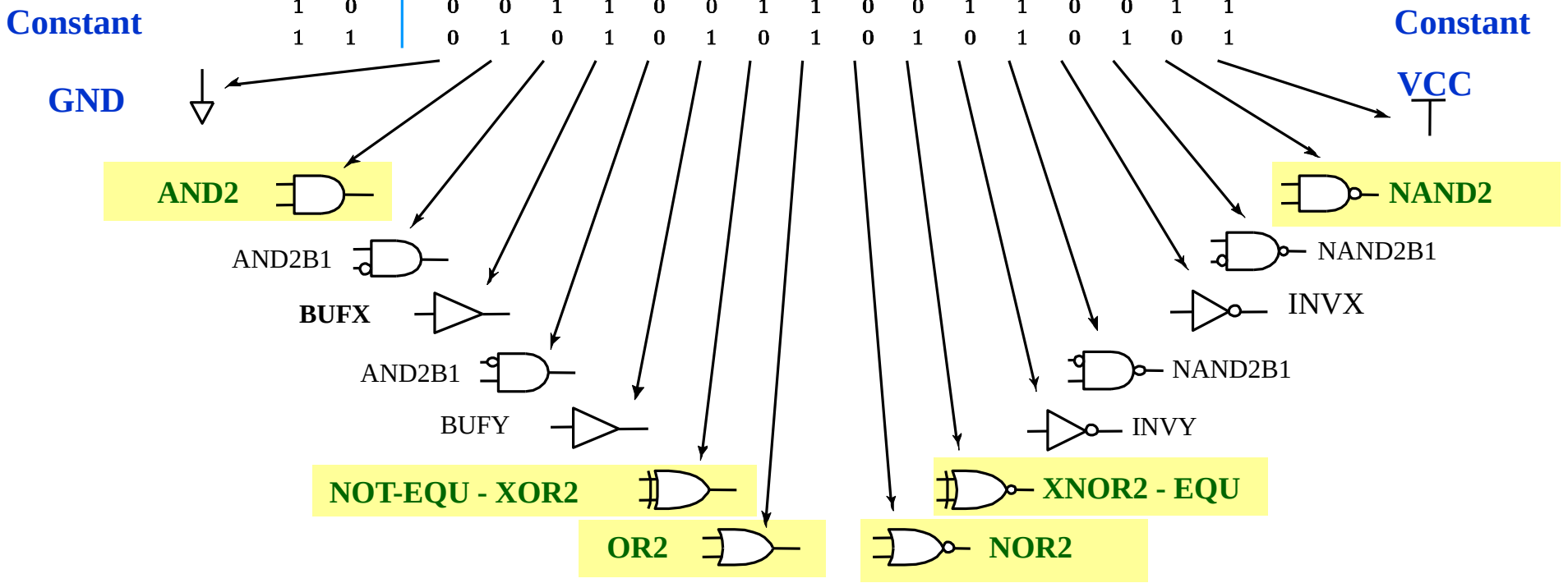


WIRE - a connection;
BUFFER - separator, driver;
INV = inverter, aka negation or complement

All functions with 2 inputs

- There are 2^2 pairs in B^2 , $B=\{0,1\}$
- There are $2^{2^2} = 16$ different logic functions

2 input variables X	Y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1



Only the highlighted functions are more frequently used because they are easy to remember.

Let the ordering of logical values be defined by the rule **'0' < '1'**, then the basic logical operations can be thought of as:

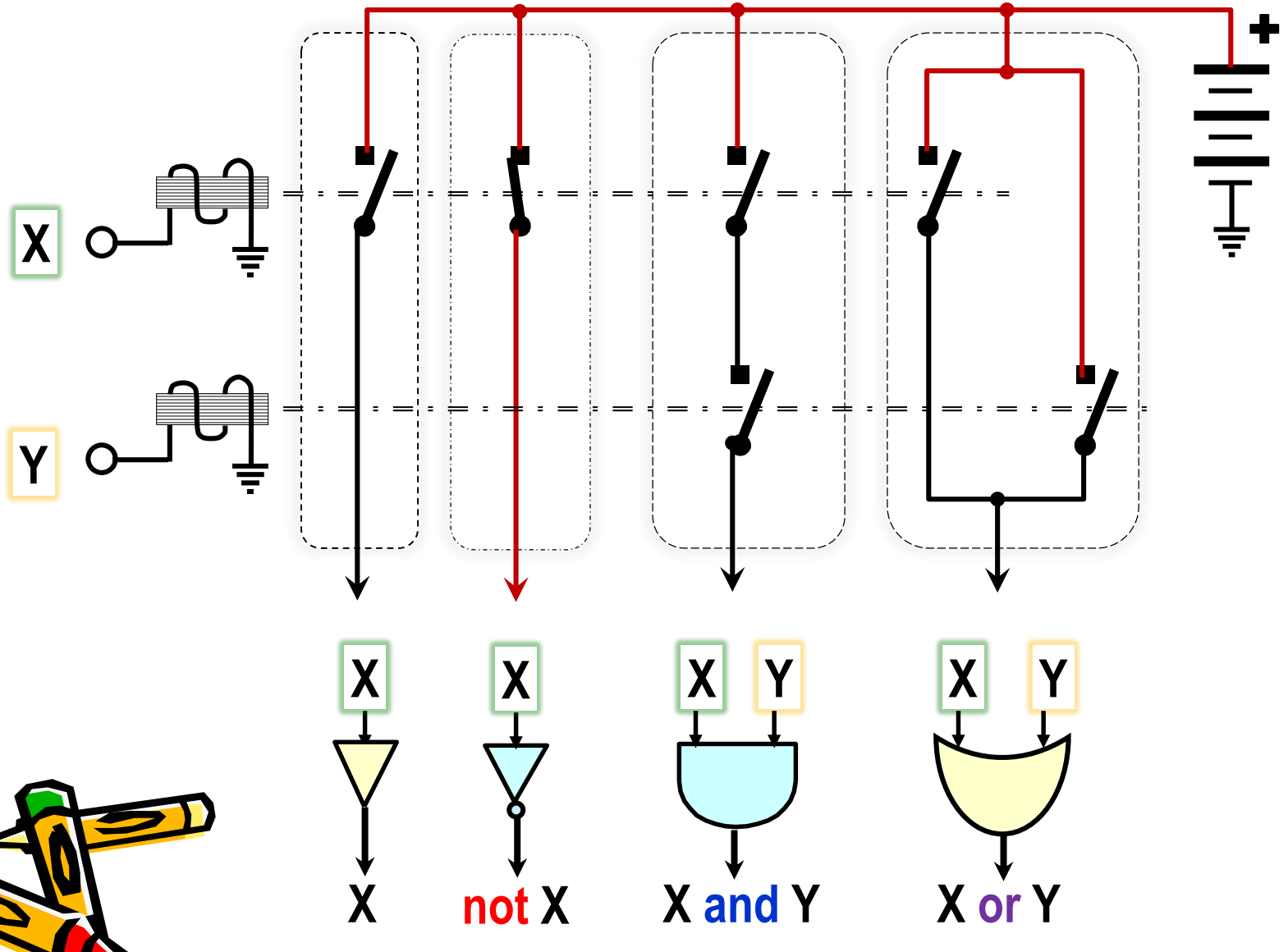
- **AND** performs the **selection of minimum**
  and thus, its output is '**1**' only for one input combination — when all inputs are at logical '**1**', i.e., at **maxima**.
- **The OR** performs the **selection of maximum**,
  and thus, its output equals '**0**' only for one combination of its inputs — when they're all at logic '**0**', i.e., at **minima**.
- **NOT negates** '**0**' to '**1**' and '**1**' to '**0**',
  thus performing their inversion → the inverter.

Grammar: minimum, pl. minima; maximum, pl. maxima; as datum, pl. data

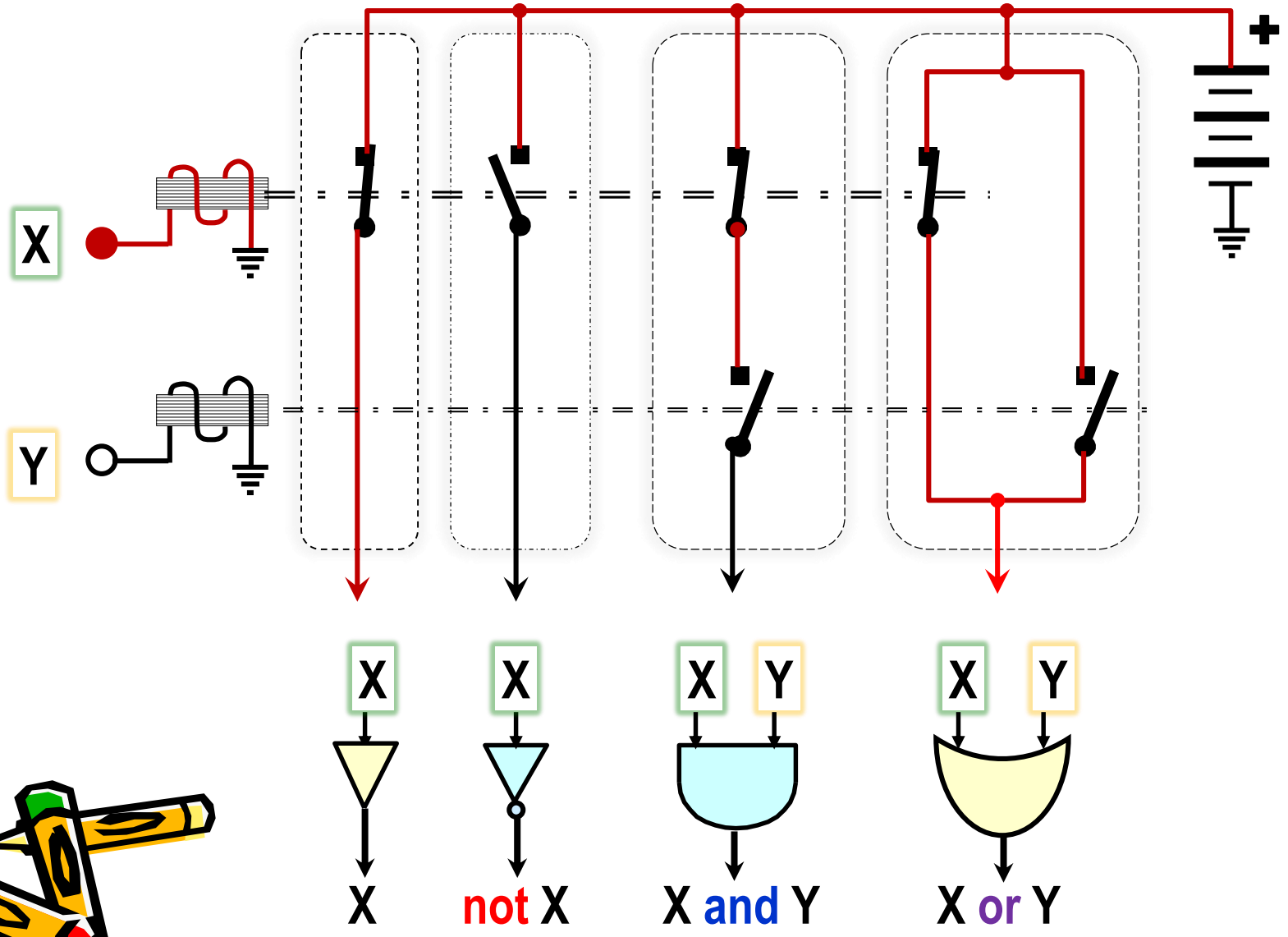
Gate

- **A gate is a circuit of arbitrary principle that implements a simple logic function.**
- The first mechanical gates appeared in 1837 in a computer built by Charles Babbage.
- Nicola Tesla patented an electric gate and switch in 1903.
- For a long time, the gates were implemented primarily by relay circuits. Industrial machines used to be controlled by them.
- The first replacement of the relay control appeared in 1968, the PLC = *Programmable Logic Controller.*
- Today, gates are mostly implemented with **CMOS transistors,** but they work similarly to relay circuits:-)

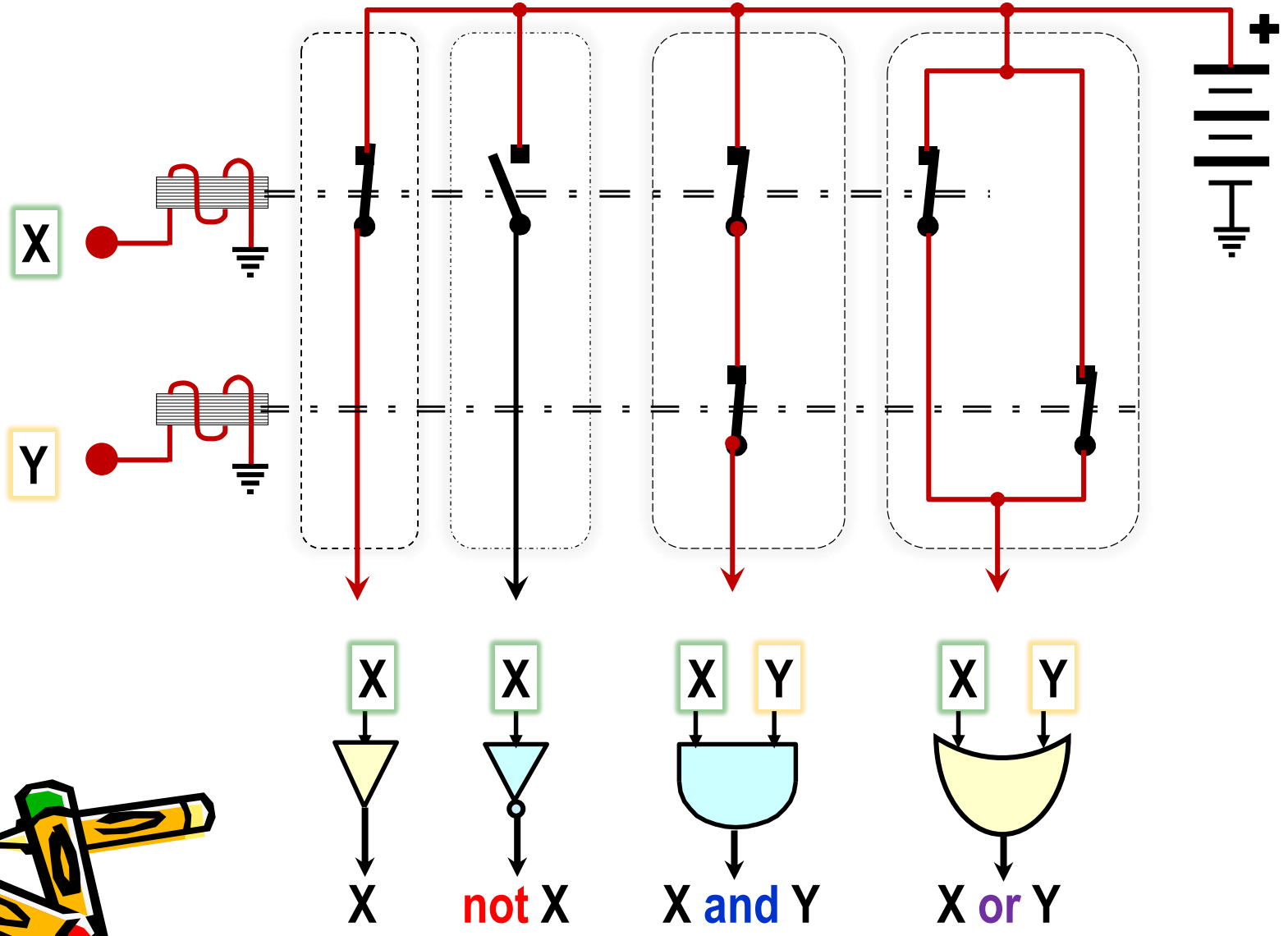
Relay Gates



Relay Gates



Relay gGates



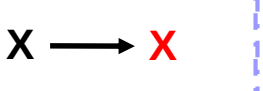
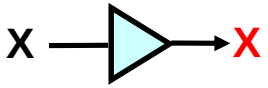
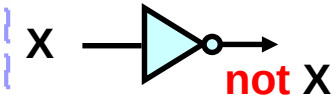
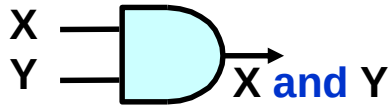
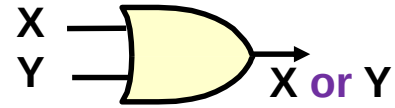
Different Notations for AND, OR, and NOT

Alternative	x'	$x \cdot y$	$x+y$
Mathematical Operators	$\neg x$ or $\overline{x}$	$x \wedge y$	$x \vee y$
	$-x$	$x \times y$, xy	$x + y$
bitwise operators			
C, C#, Java, Python	`~x`	`x & y`	`x | y`
logical operators			
C, C#, Java	`!x`	`x && y`	`x || y`
Pascal, Python	`not x`	`x and y`	`x or y`

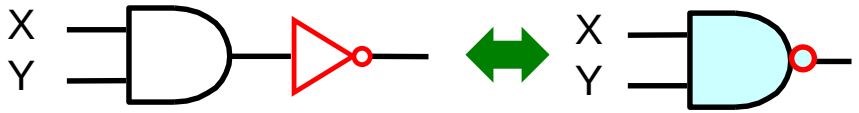
In VHDL, logical and bitwise operators

Gages AND, OR, and NOT

not x ***x and y*** ***x or y***

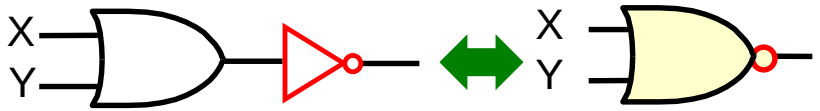
wire / signal	buffer	inverter / NOT	AND-gate	OR-gate
				

and + inverter **nand** gate



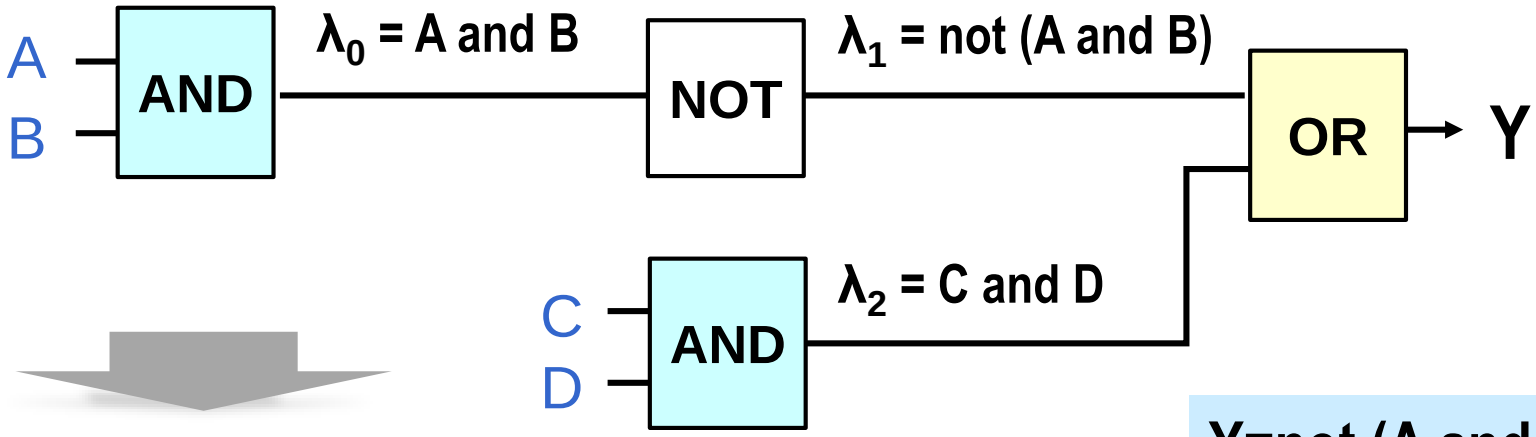
not (X and Y) **not (X and Y)**

or + inverter **Nor** gate

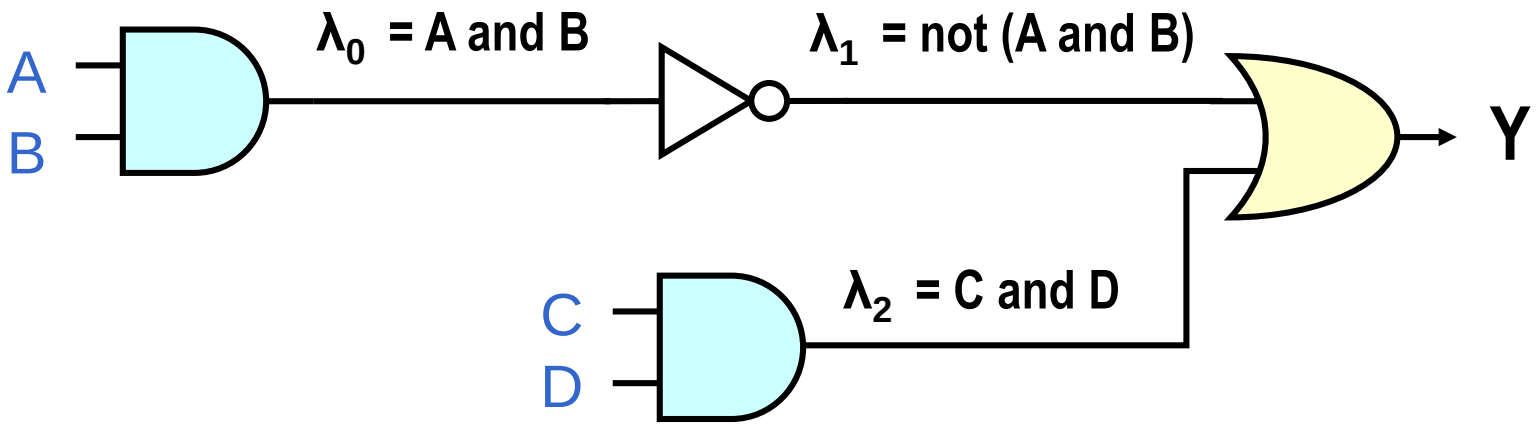


not (X or Y) **not (X or Y)**

Scheme = Evaluation Order



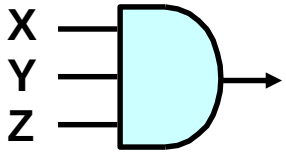
Y=not (A and B) or (C and D)



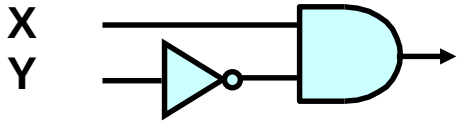
*Note: In logic, AND and OR operations have **same precedence**, so we must specify the calculation procedure with parentheses.*

Minterms and Maxterms

Minterms

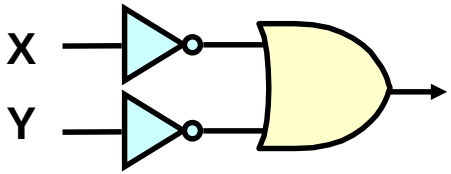


X and Y and Z

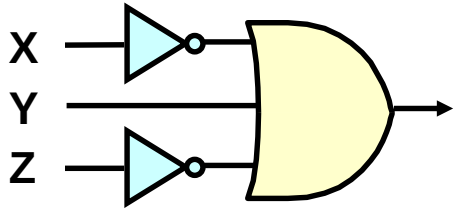


X and not Y

Maxterms



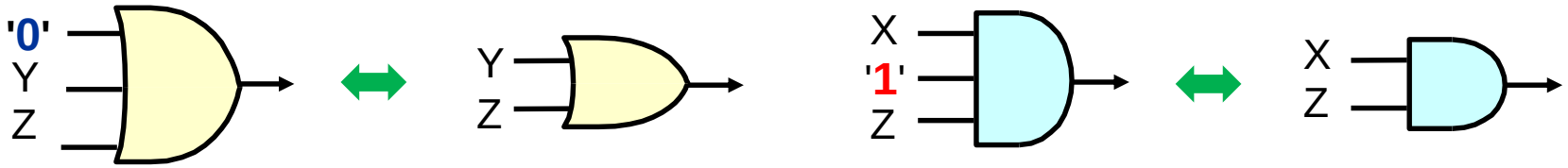
not X or not Y



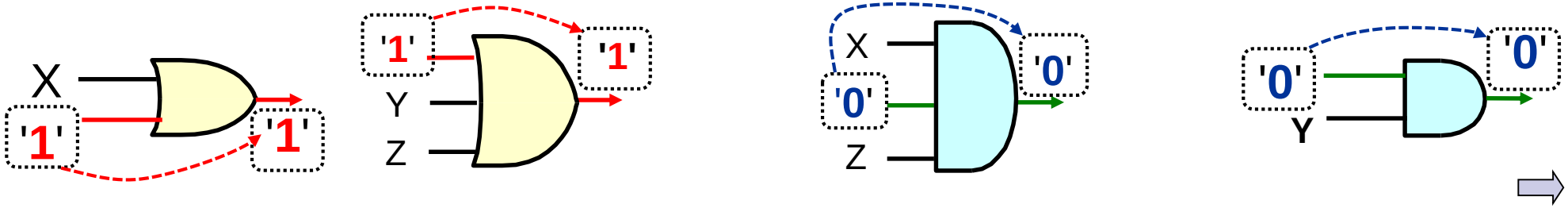
not X or Y or not Z

Identity Law and Annulment Law

Postulate	*OR version*	*And version*
Identity Law *cz: Neutrality*	$x + '0' = x$	$x \bullet '1' = x$



Theorem	*OR version*	*And version*
Annulment Law *cz: Aggression*	x + '1' = '1'	x ● '0' = '0'



Sum-of-Products = S-o-P, based on minterms

F = m(1,3,5,6,7) **not F = m(0,2,4)**

	C	B	A	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

(not C and not B and A)
or
(not C and B and A)
or
(C and not B and A)
or
(C and B and not A)
or
(C and B and A)

not F
1
0
1
0
1
0
0
0

(not C and not B and not A)
or
(not C and B and not A)
or
or (C and not B and not A)

Products-of-Sum = P-o-S, based on maxterms

F = M**(0,2,4)**

	C	B	A	F
0	0	0	0	**0**
1	0	0	1	1
2	0	**1**	0	**0**
3	0	1	1	1
4	**1**	0	0	**0**
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

(C or B or A)
and
(C or not B or A)
and
and (not C or B or A)

On-set and off-set

- *Note: Minterms describe elements of the on-set and are created according to binary values. E.g. F1=m(1)=A'+B'+C for index 1, which is binary 001, has negated variables A' and B' in place of '00' and variable C in place of '1' in the minterm.*
- *Note: Maxterms describe elements of off-set. They write variables negated relative to the binary value! For example, F4=M(4)=A'+B+C for index 4, which is binary 100, has the variable A' in the '1' place and the negated variables B' and C' in the '00' place in the maxterm.*
- *The term off-set is a well-established term in logic, but a bit misleading; outside of it, it is used in a different sense.*

Logic circuit

may include

Combinational circuits

their outputs are determined only by immediate values on their inputs

It can be described normal form with lower complexity

decoders, comparators, multiplexers, demultiplexers...

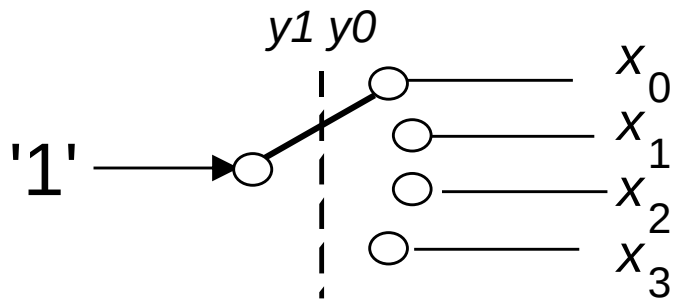
It must be implemented by decomposition for simpler blocks

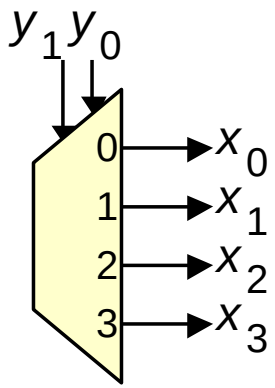
adders, multipliers, ...

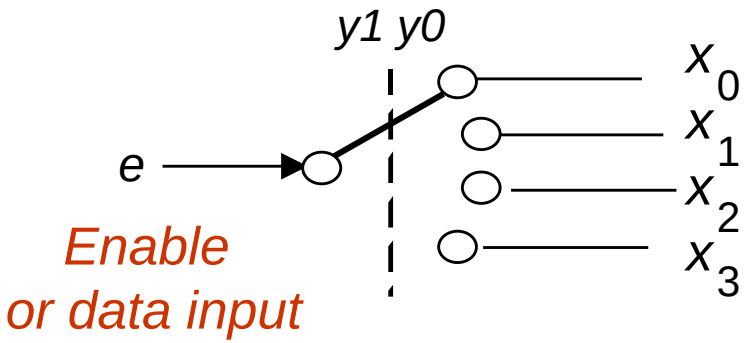
Sequential circuits

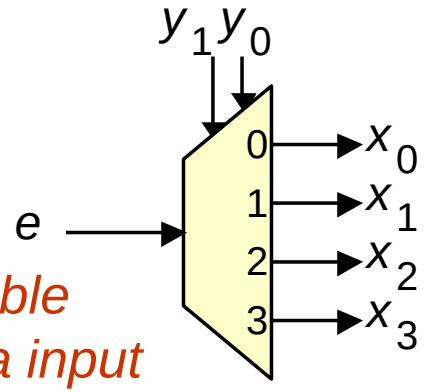
their output depends on a sequence of input values

Decoder One Hot and Demux









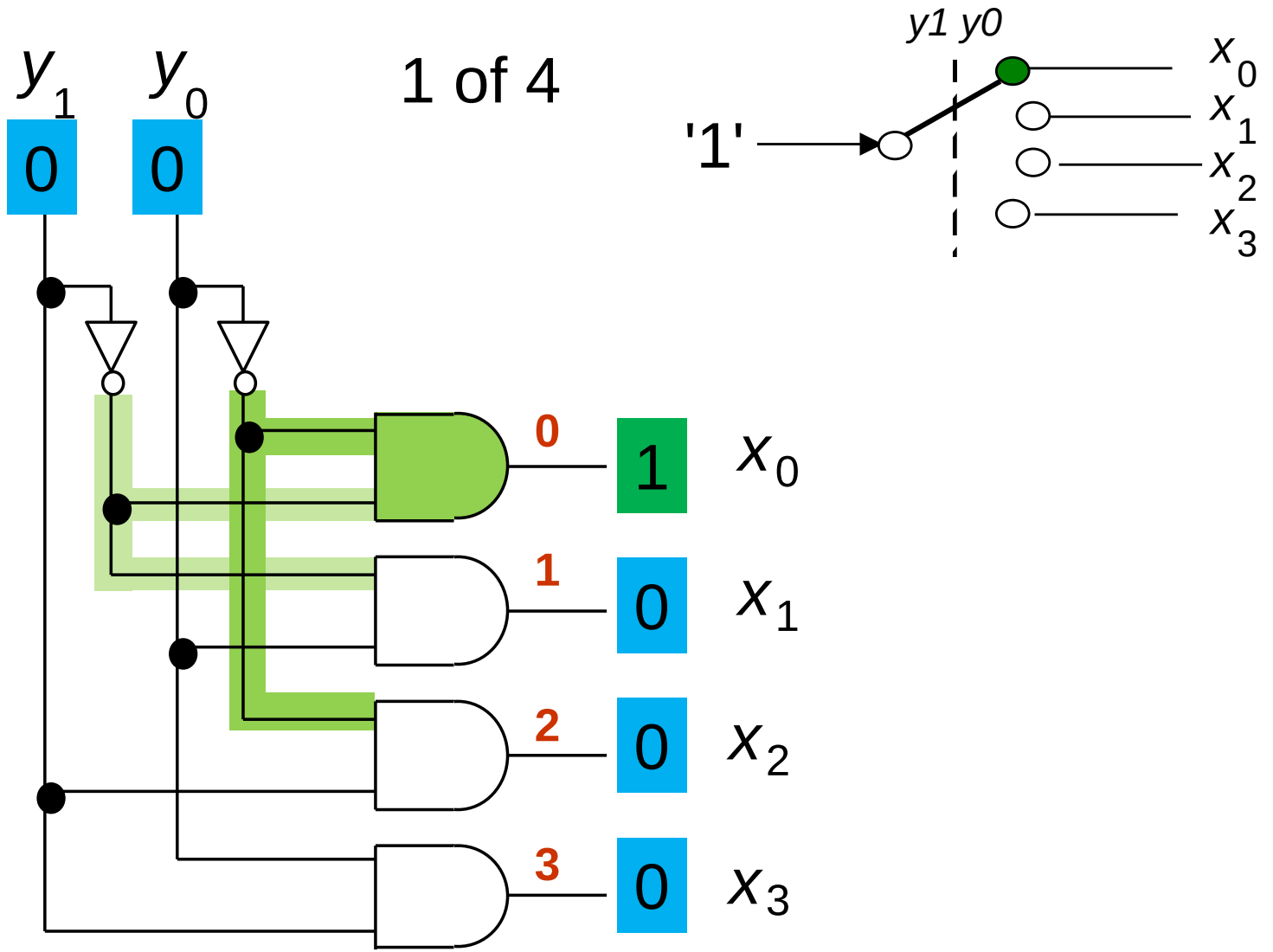
We constructed Demux by the direct usage of minterms

Enable

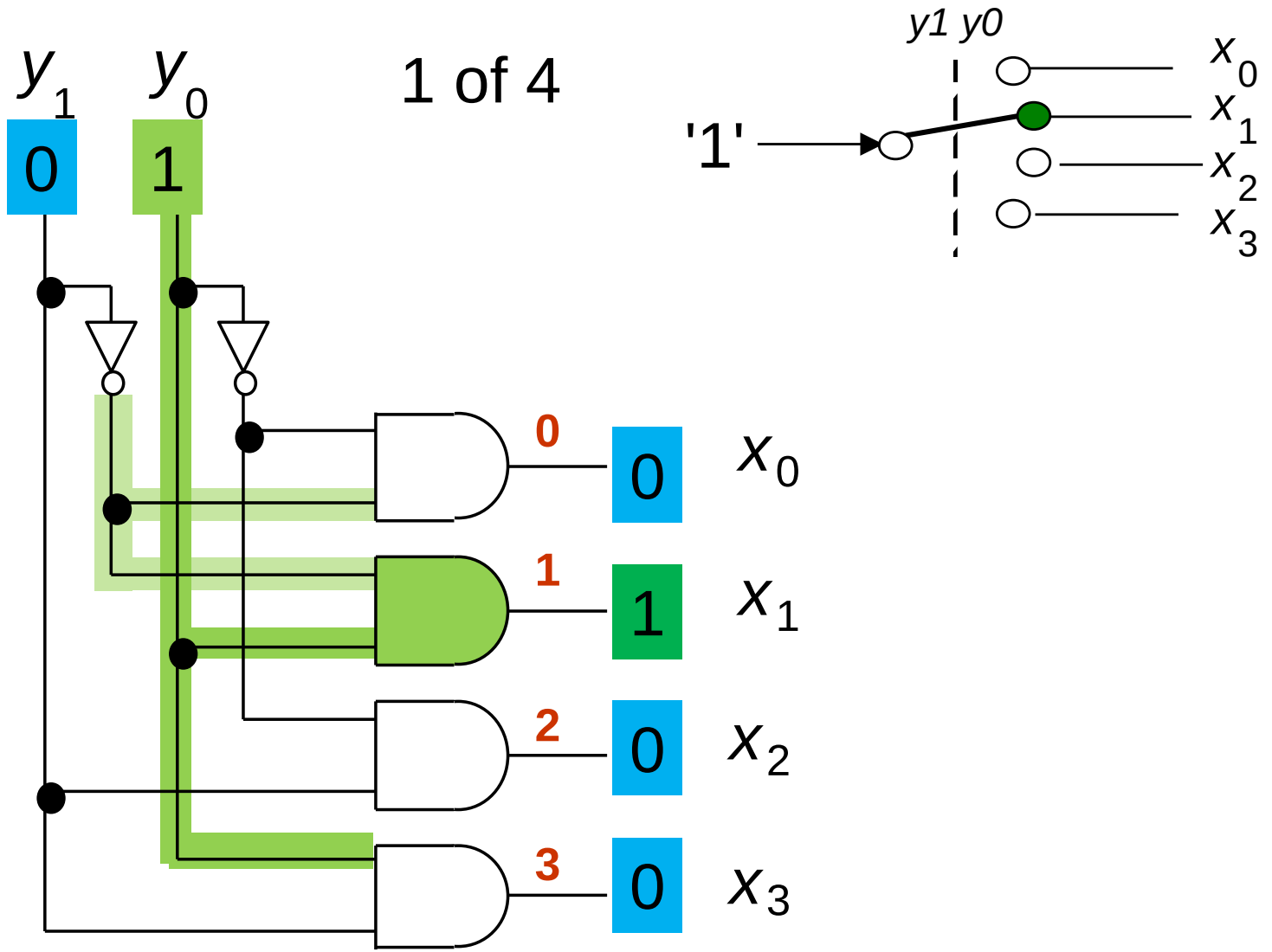
- Circuits often have "enable" inputs, i.e., output release or output activation, with "enable" active in ´1´ or ´0´ - the choice depends on the circuit designer.
- The "enable" input is often called EN or G (from gate), but some use other names.
- A circuit can have multiple auxiliary "enable" inputs, and the resulting "enable" is then given by some logical function of these inputs, usually AND.
- If "enable" is not active, the outputs are inactive, which can be either ´0´ or ´1´, or even a high impedance state, again depending on the designer.
- If the outputs do not go to a high impedance state when "enable" is inactive but to ´0´ or to ´1´, then the "enable" input can sometimes serve as a data input.

Example: a decoder with "enable" active in ´1´ and with outputs inactive in ´0´ has all its outputs in ´0´ when "enable"=´0´. In contrast, when "enable"=´1´, it sends the value of ´1´ to the output selected by the address bits. The circuit thus passes data from "enable" to the currently selected output.

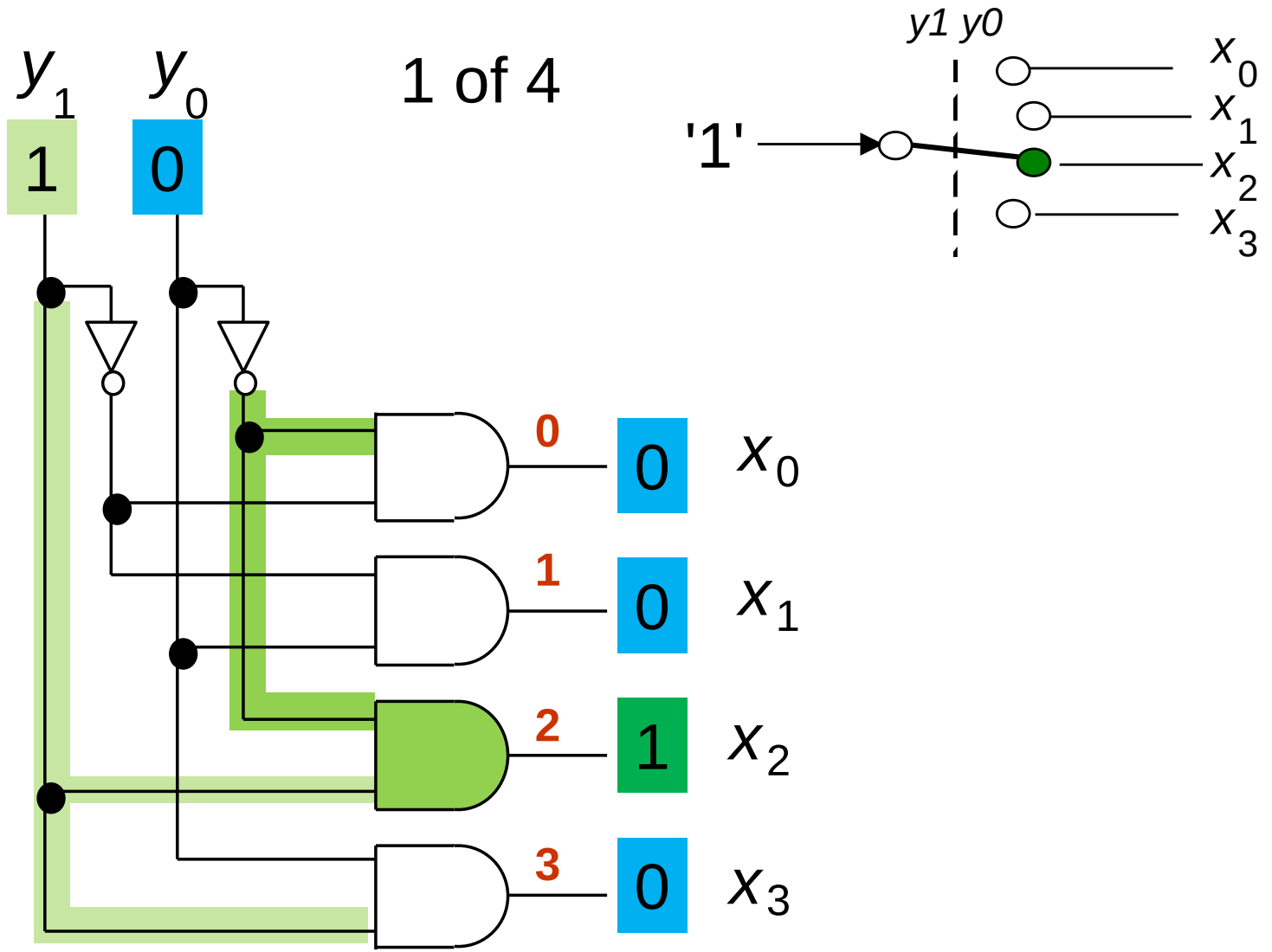
Decoder One Hot 1 of 4 [1/4]



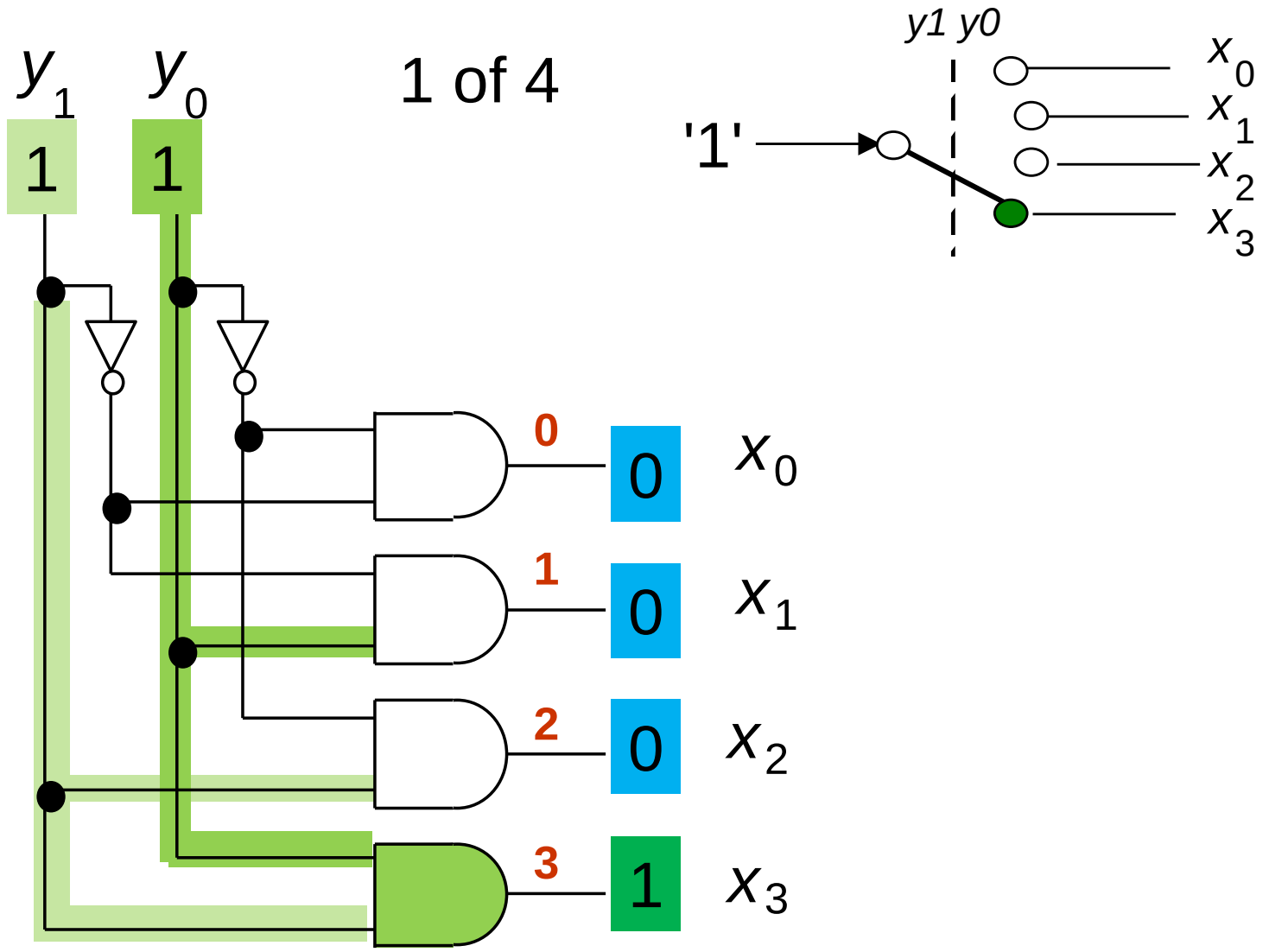
Decoder One Hot 1 of 4 [2/4]



Decoder One Hot 1 of 4 [3/4]



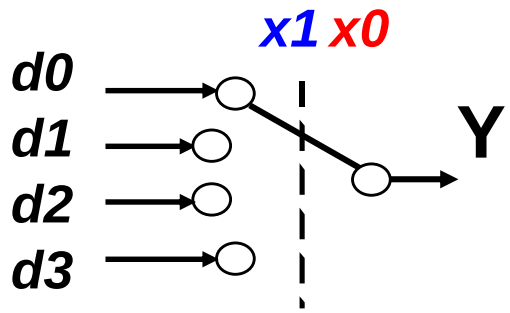
Decoder One Hot 1 of 4 [4/4]

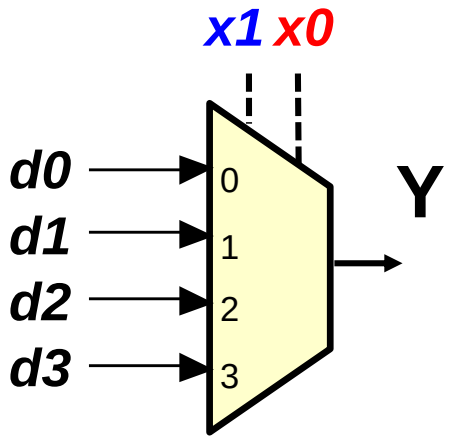


4 channel multiplexer / 4:1 multiplexer

Multiplexer *or MUX aka data selector* has

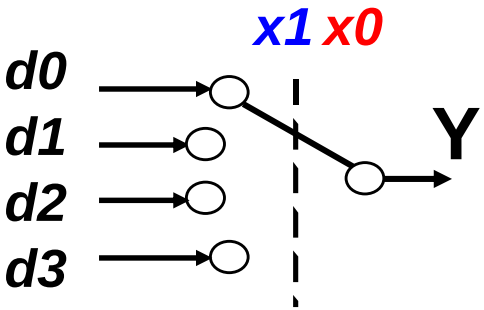
- n address inputs, 2^n data inputs
- one data output

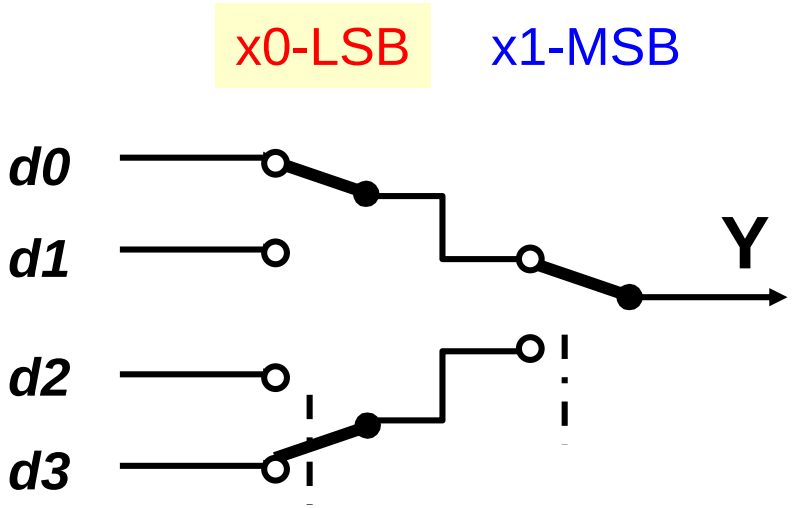


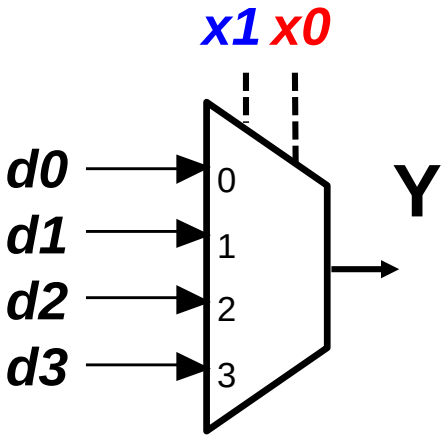


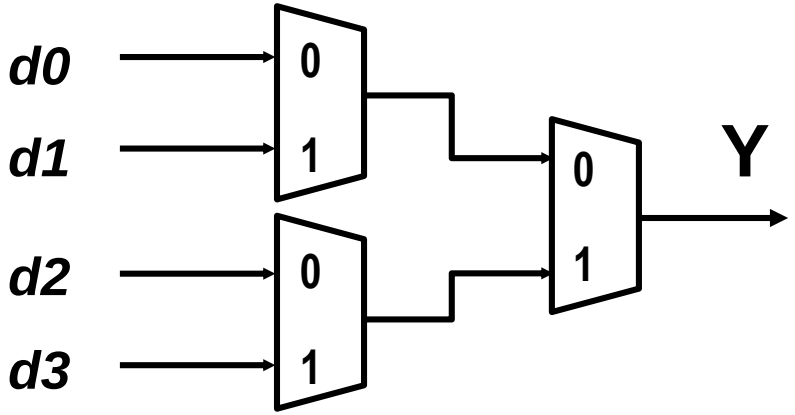
The multiplexor is a very important element, from which configurable FPGA circuits are built.

Multiplexer 4:1 Splitted to 2:1

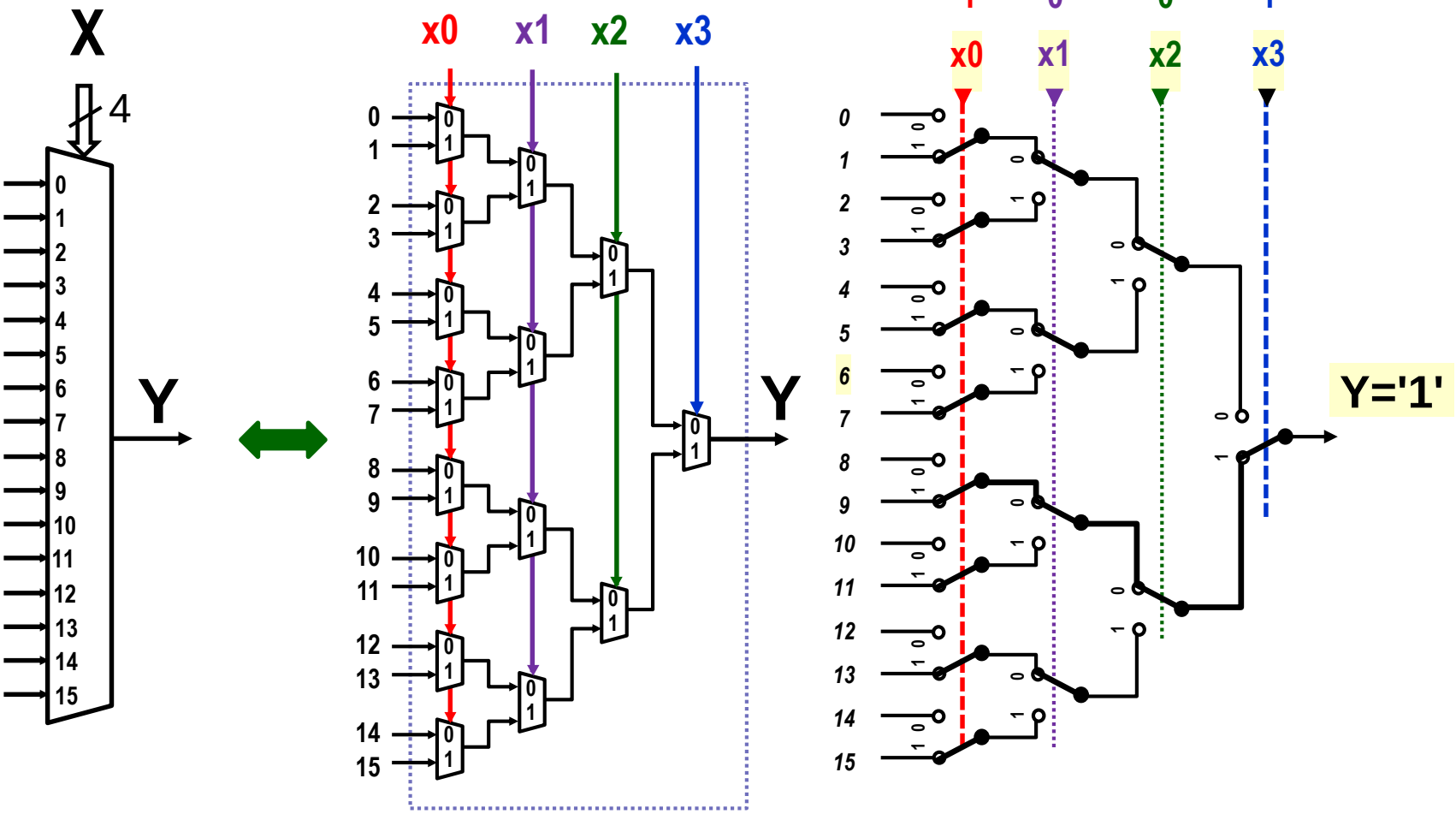








16:1 Multiplexer Splitted to 2:1

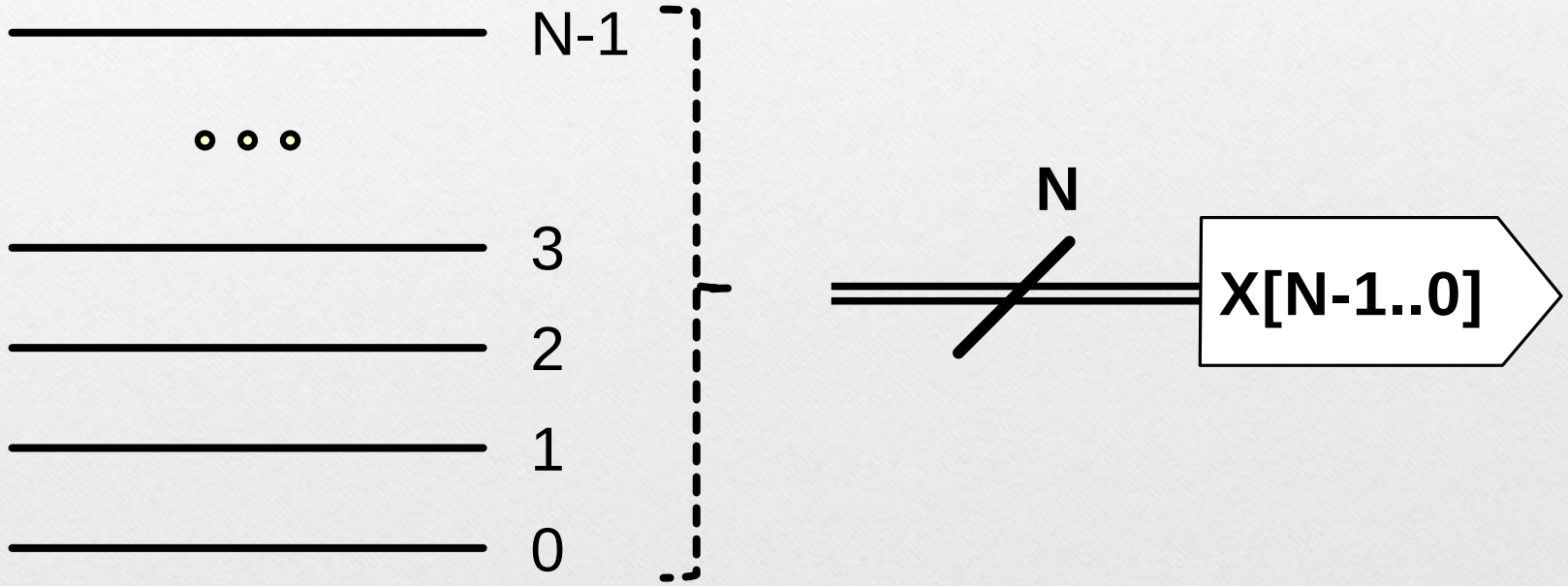


Mux 16:1

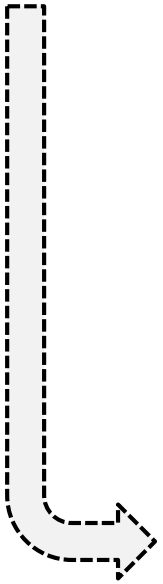
15 x Mux 2:1

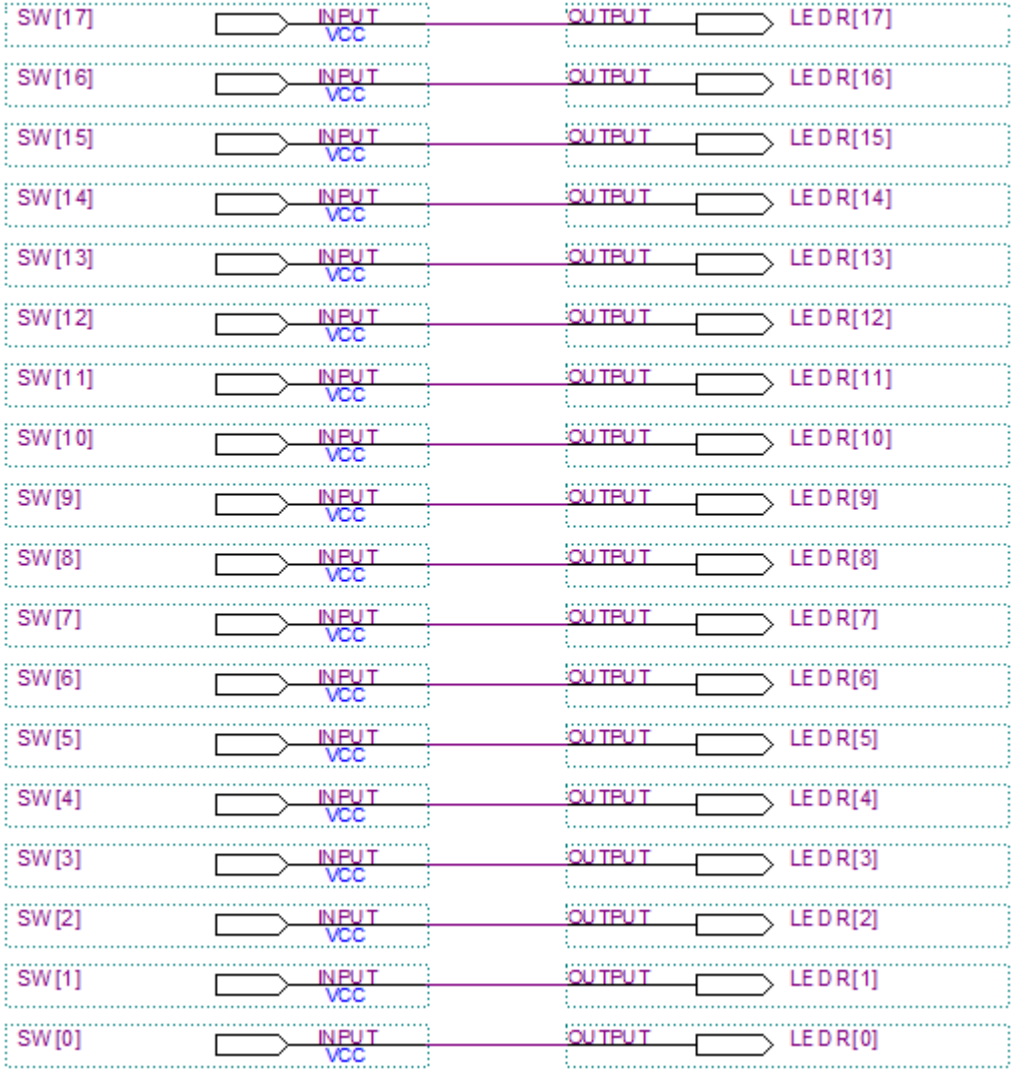
Switch analogy

Bus
(bus, bus signal)

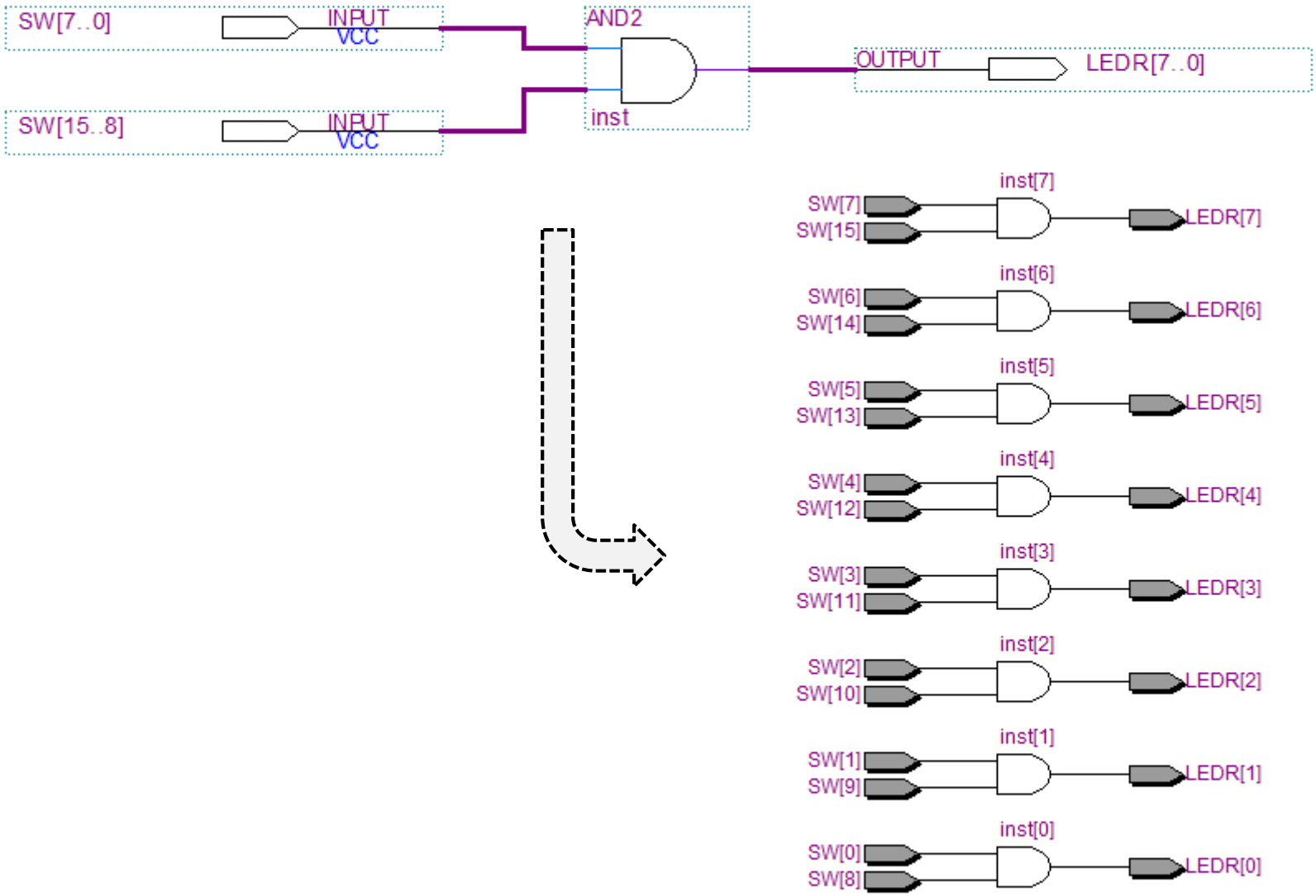


Buses allow easier wiring



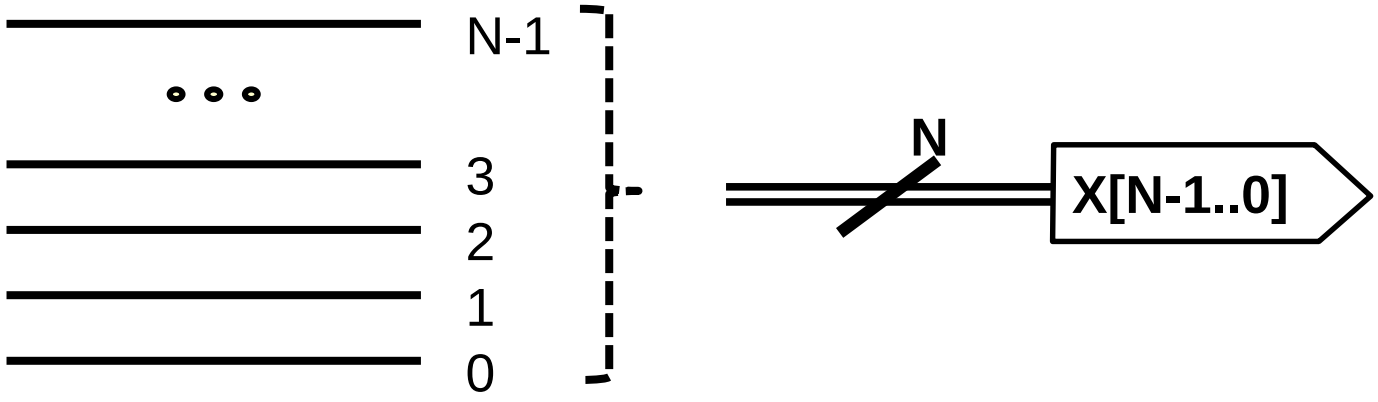


Logical Operations with Bus-signals

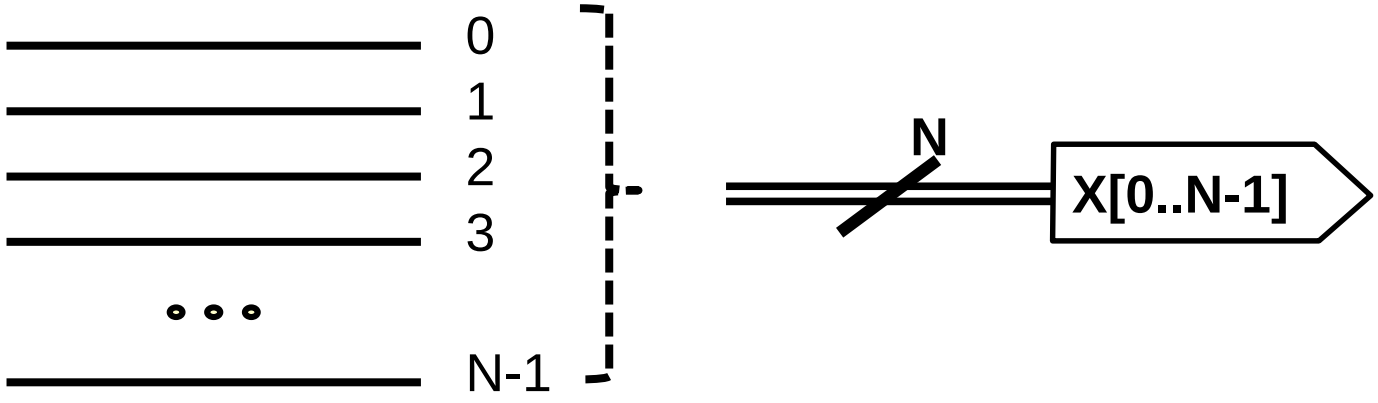


We utilize two bus numbering directions

More common is **downto** *related to the bit numberings in numbers.*



Direction **to** *is used for array/memory indexing*



Logic Circuit

can include

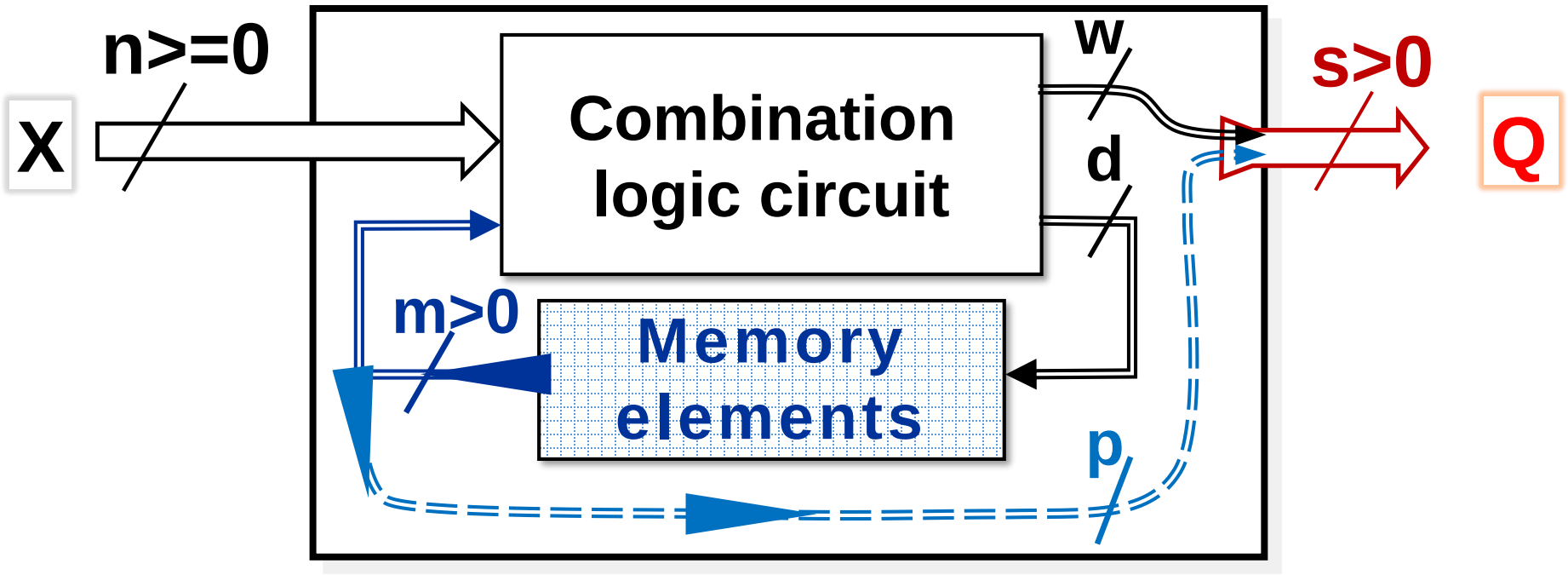
Combinational circuits

Sequential circuits

Synchronous - have internal memory elements and are controlled synchronously by the clock

Asynchronous - they do not use clocks, are faster, but have a very complex design that requires a solid state realization.

Sequential Logic Circuit

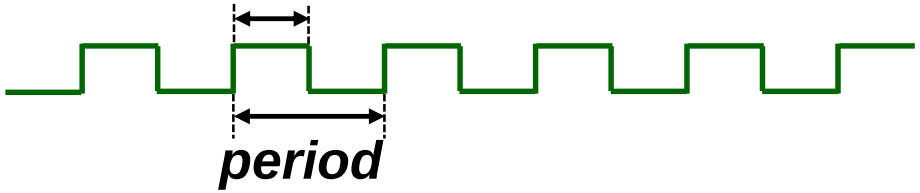


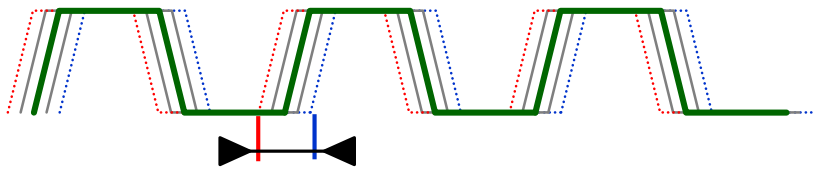
Combination circuit:
Its outputs depend only on the instantaneous inputs.

Sequential circuit:
Its outputs depend on the sequence of previous inputs.

Terminology

Duty cycle: 50% *or 1:1 alternation (cz only)*





jitter (clock variation)

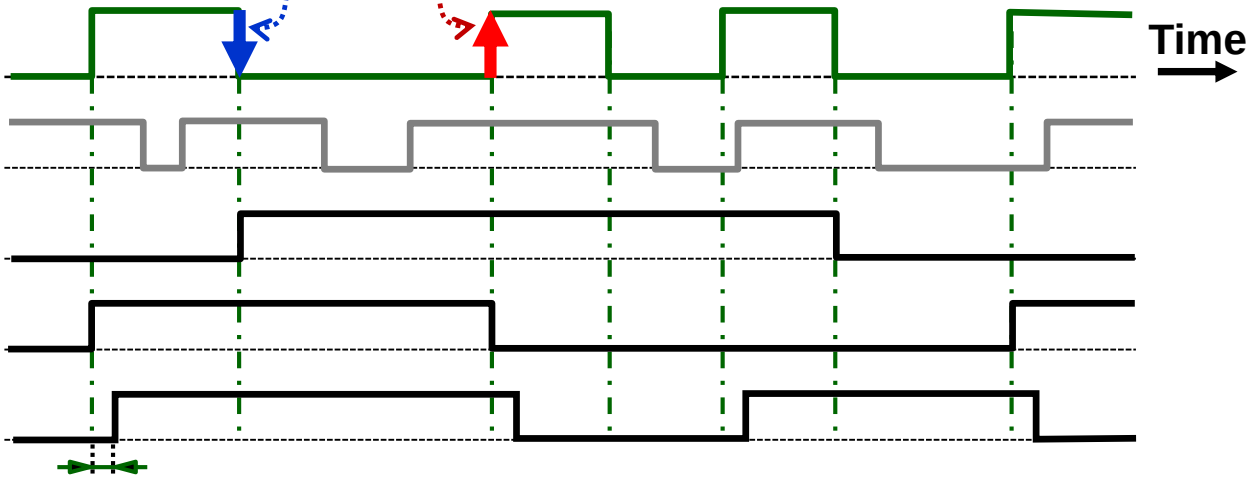
falling or descending edge

falling edge

leading edge

rising edge

Clock, clock



Asynchronous

x1

Synchronous with the clock

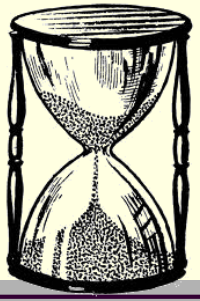
x2

x3

x4

x4 - clock skew *aka* **timing skew**

Terminology Synchronous/Asynchronous



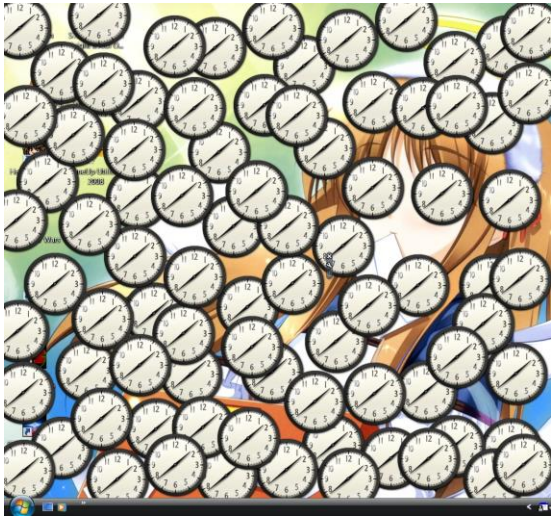
- *Synchronous - occurring, existing, or arising at the same time, from Greek* ***syn****=plus/compute,* ***chronos****=time*

συγχρονοσ

synchronos

- Two signals can be either synchronous or asynchronous to each other.
- An input is referred to as a **synchronous input relative to the clock** if it can only affect the output synchronously with the clock signal.
- In contrast, the "**asynchronous input**" of the circuit will affect the output **independently of the clock**.

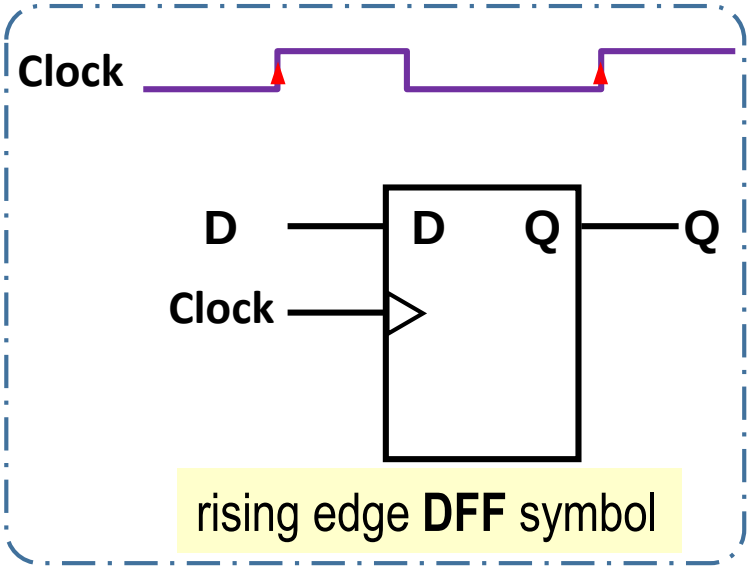
Sequential Synchronous Circuits

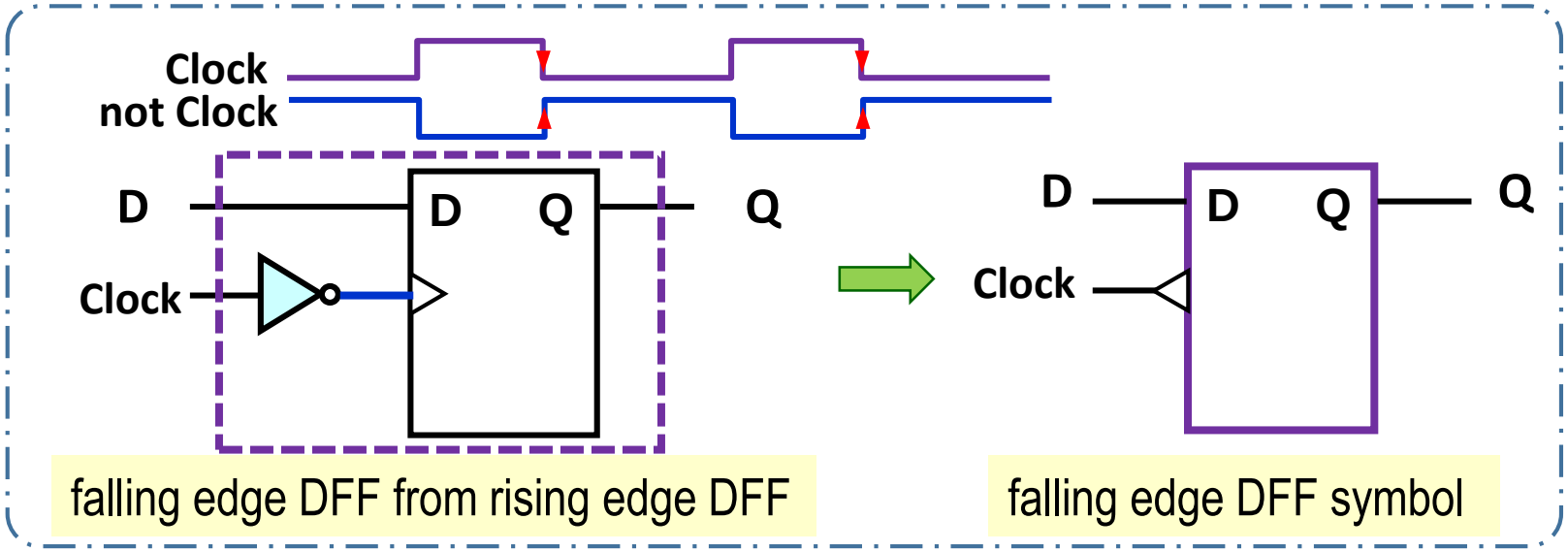


http://www.beige-box.com/

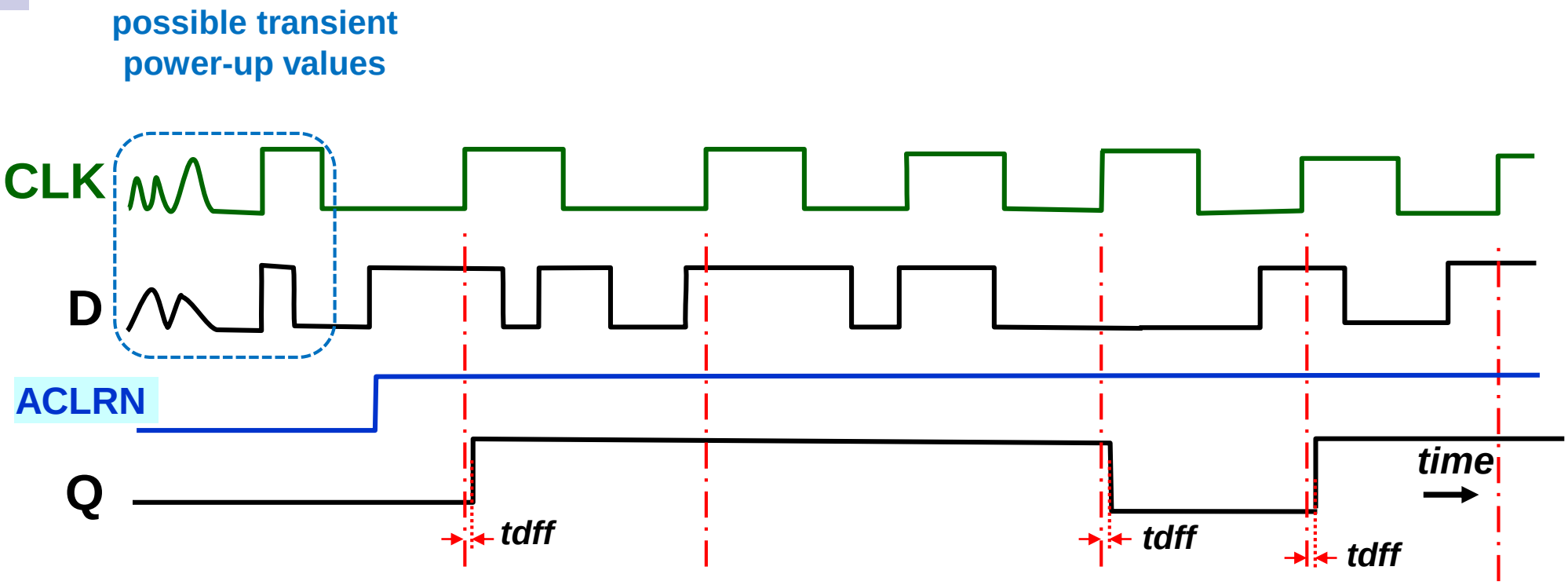
DFF - Data Flip Flop

The most frequent internal implementation of DFFs samples input D on rising edges of clock!





Asynchronous Clearing

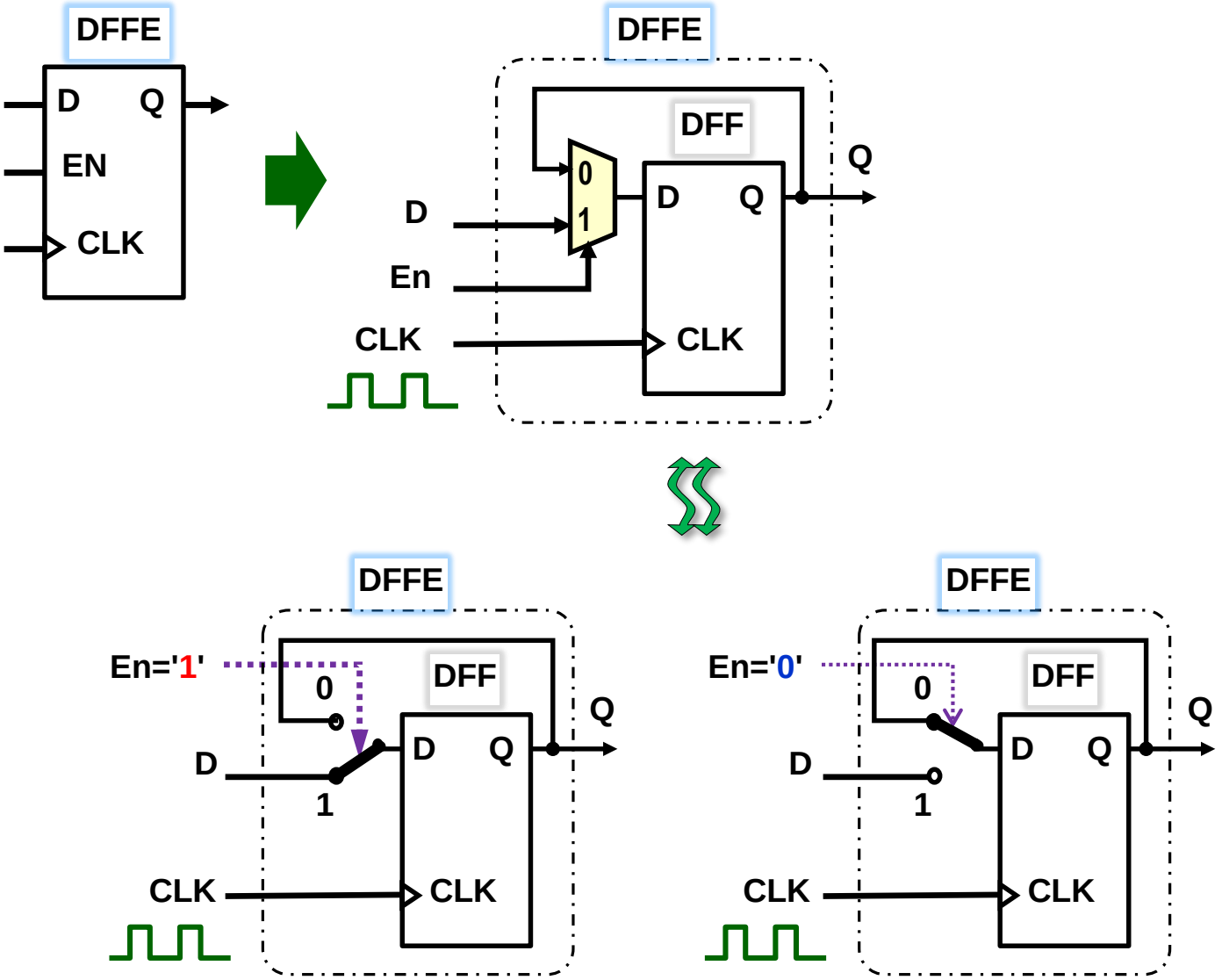


t_{dff} - propagation delay

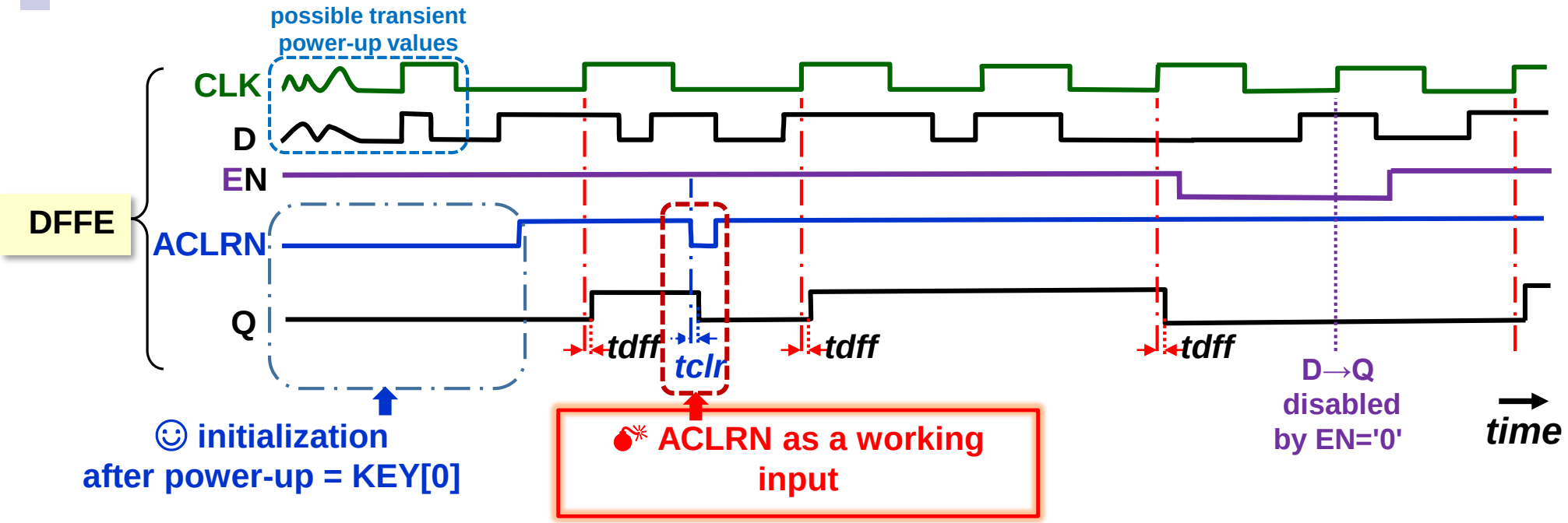
ACLRN - **A**synchronous **CL**ea**R** **N**egative-logic

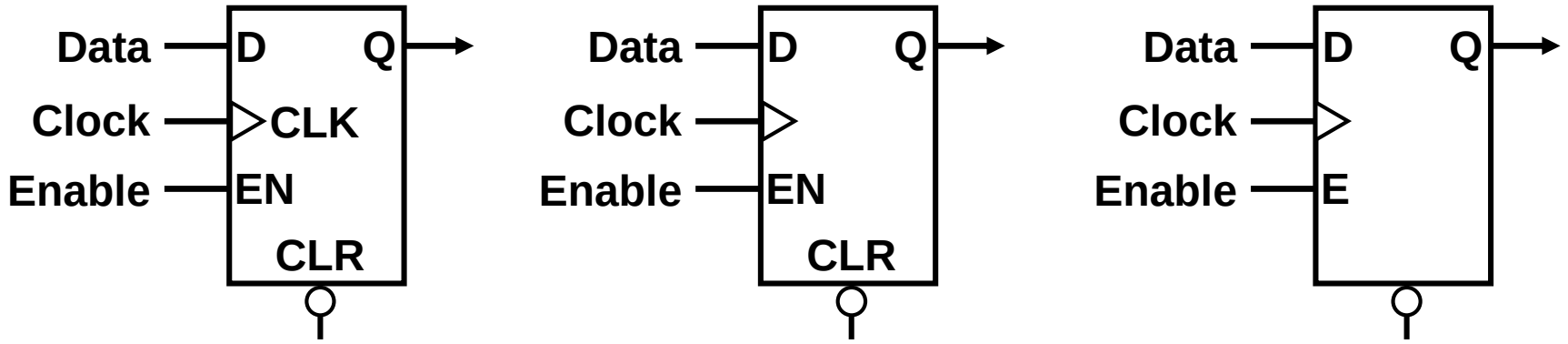
- It clears the output(s) with immediate effect.

DFFE - Data Flip-Flop with Enable

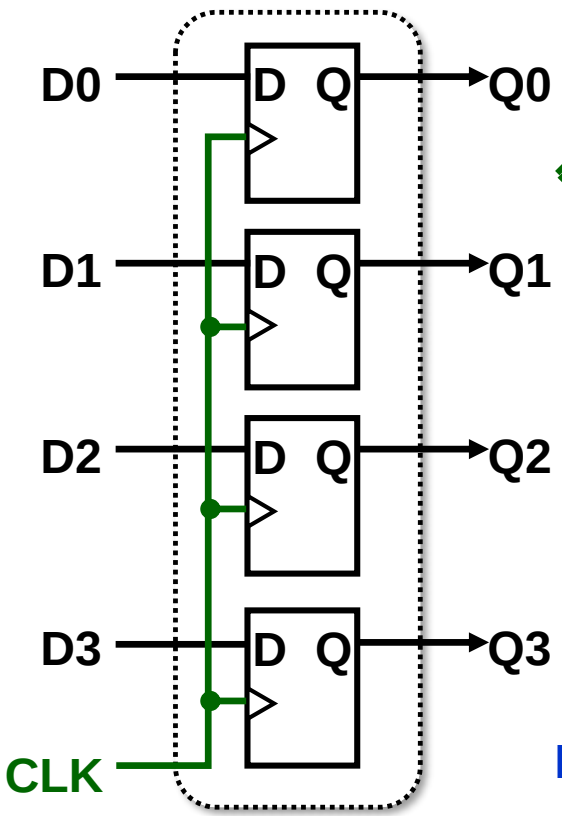


Asynchronous Clearing

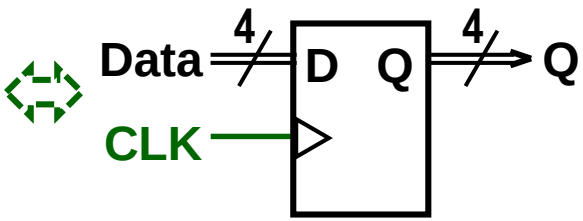




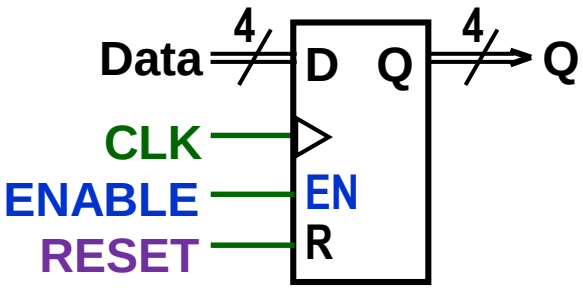
Register Consists of DFF group

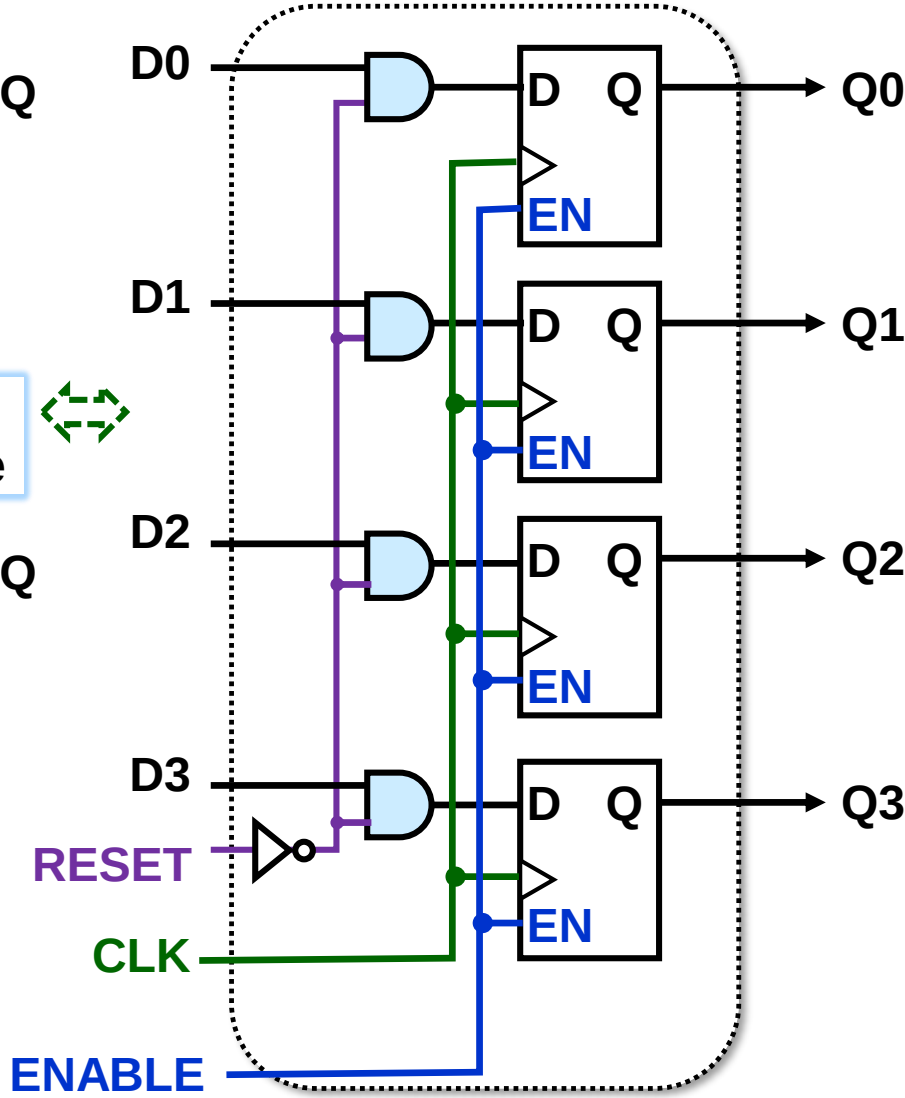


4-bit Register

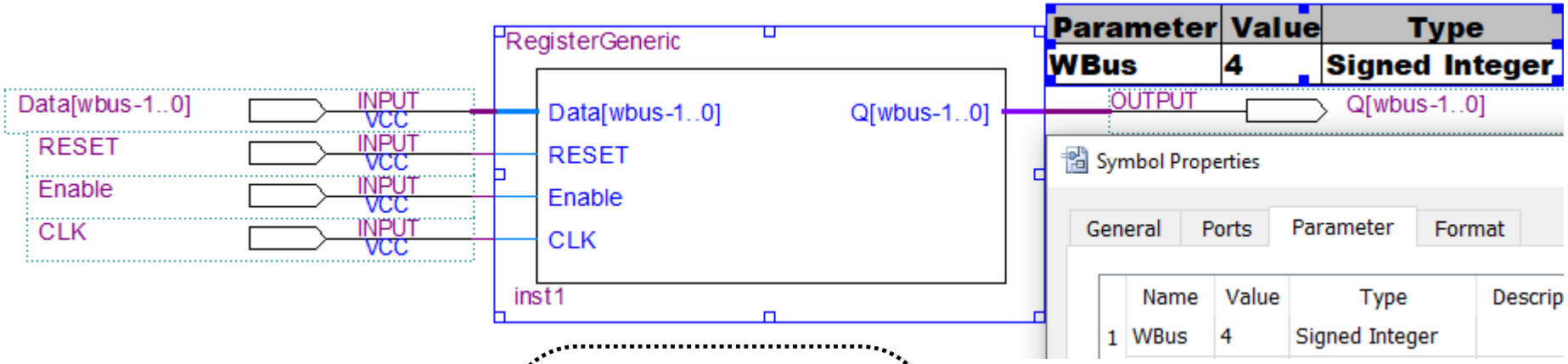


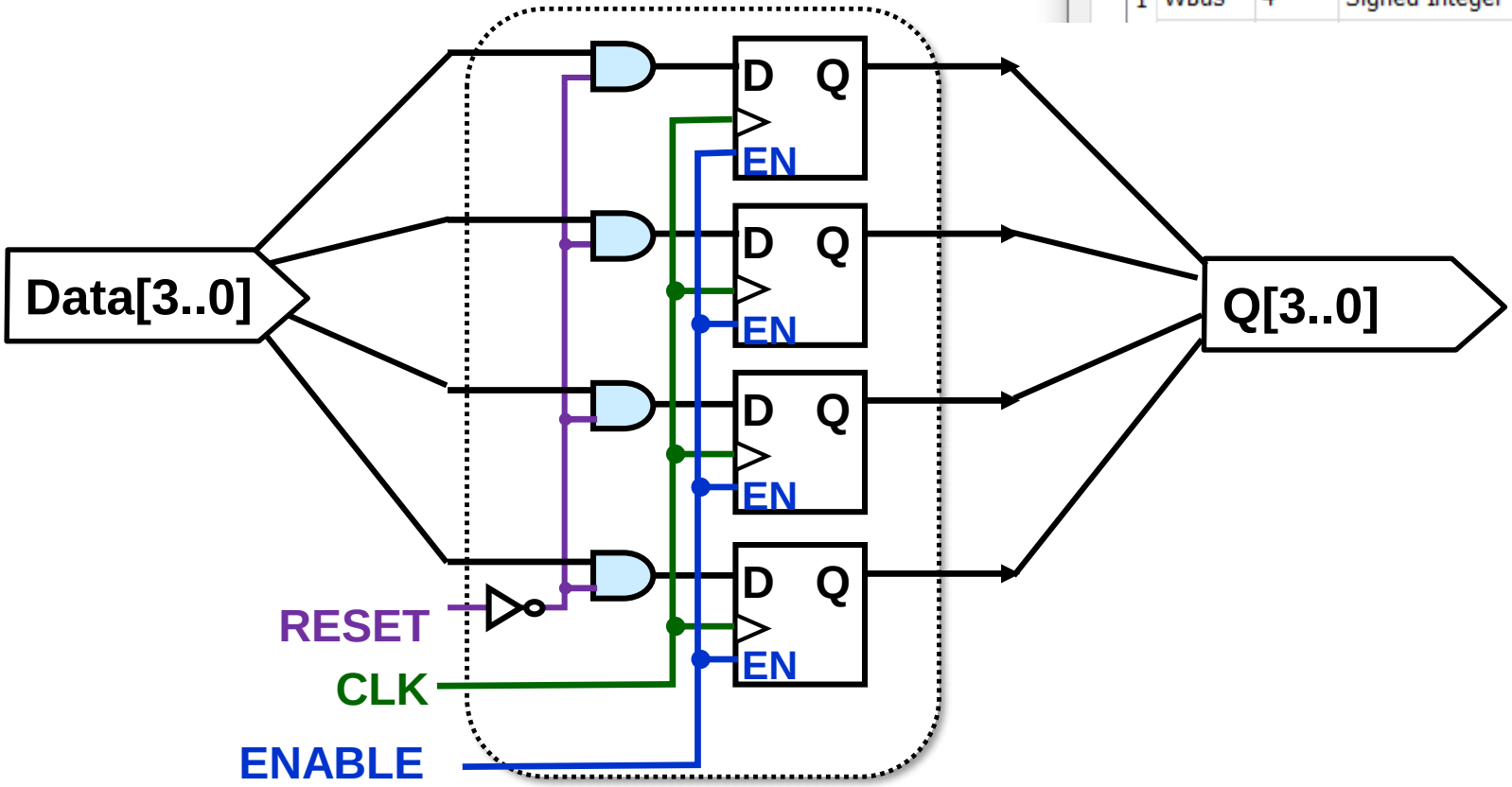
4-bit Register
with Reset+Enable





Register in DCE/library

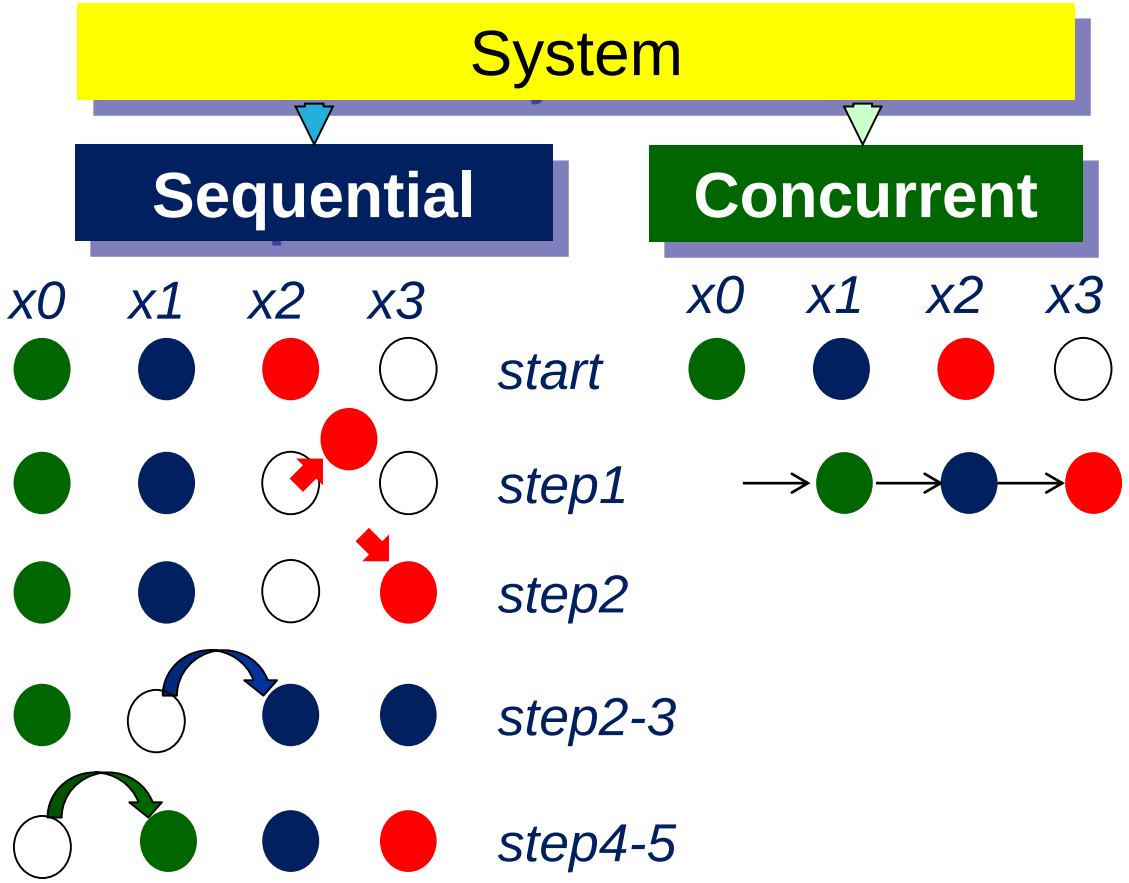




Shift / Delay

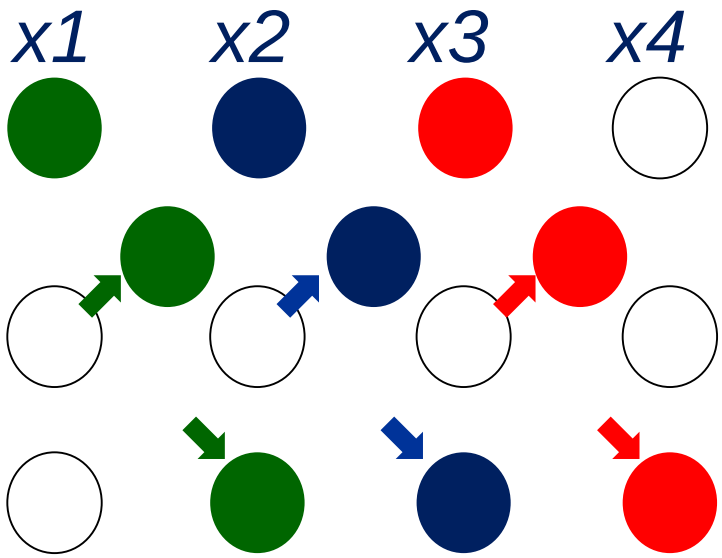
Shift Register

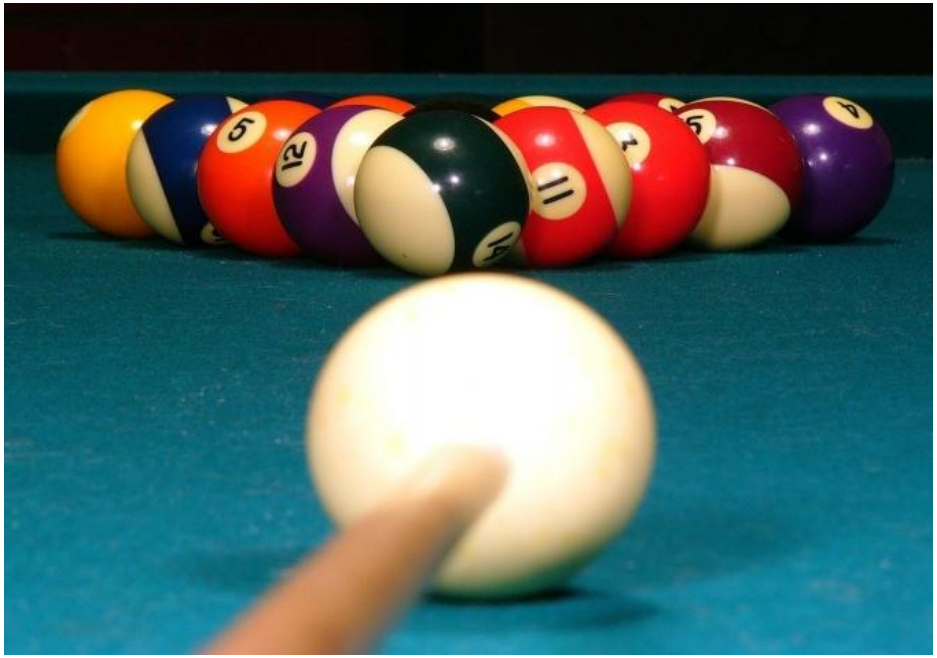
Task: Shift an Array



concurrent = *executed at the same time*

A. Emulated by parallel processors

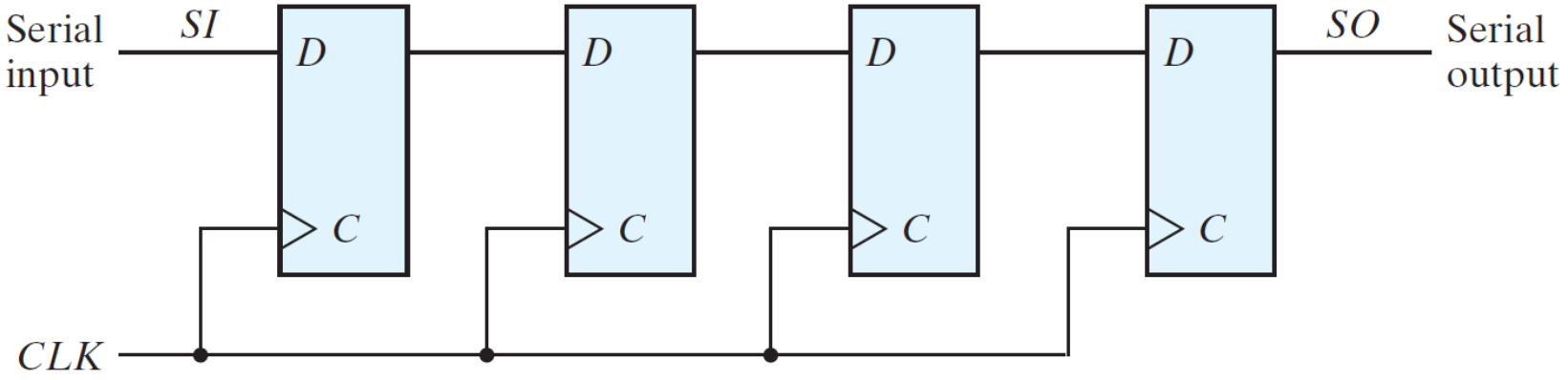




B. Feature of circuits

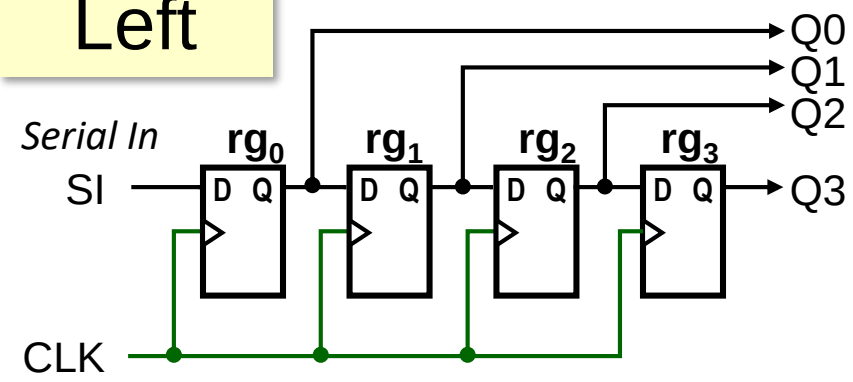
Shift register / Shift Register

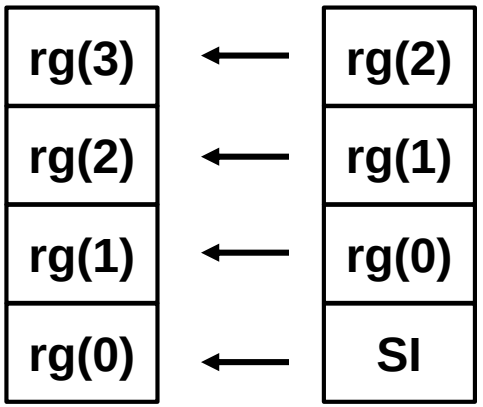
- *shifts its cells with each rising/falling edge of the clock*



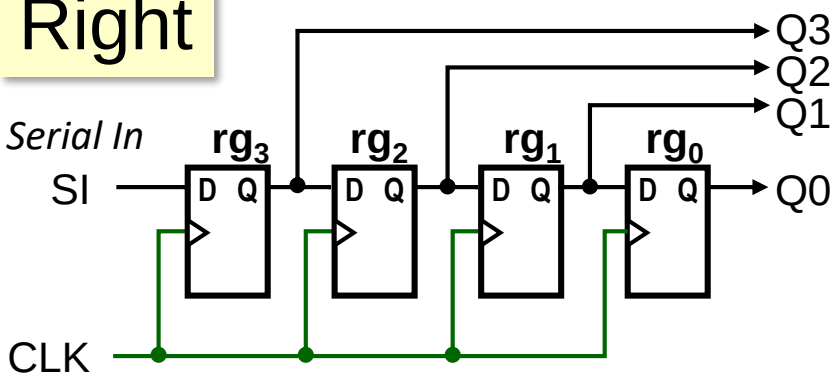
Direction of Shift

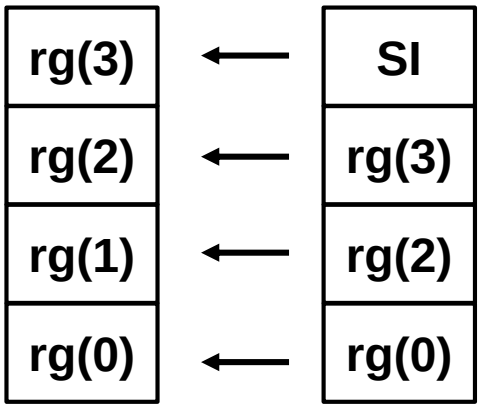
Left





Right



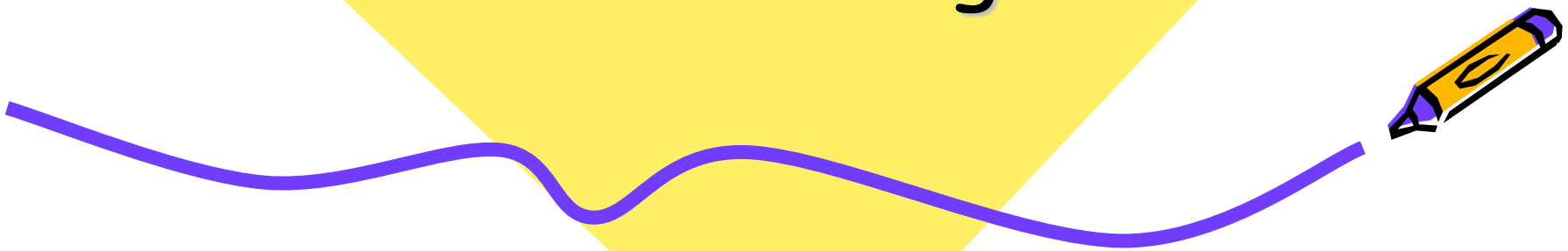


The left and right direction of shift registers are not determined by the positions of their flip-flop circuits in the schematics, but the left/right shift always depends on the weights of the output bits:

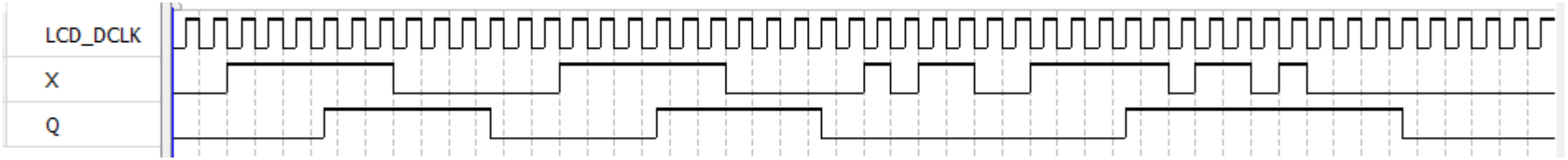
- *the right direction means dividing an unsigned binary number by two*
- *the left direction represents multiplication without the sign 2.*

Filtering

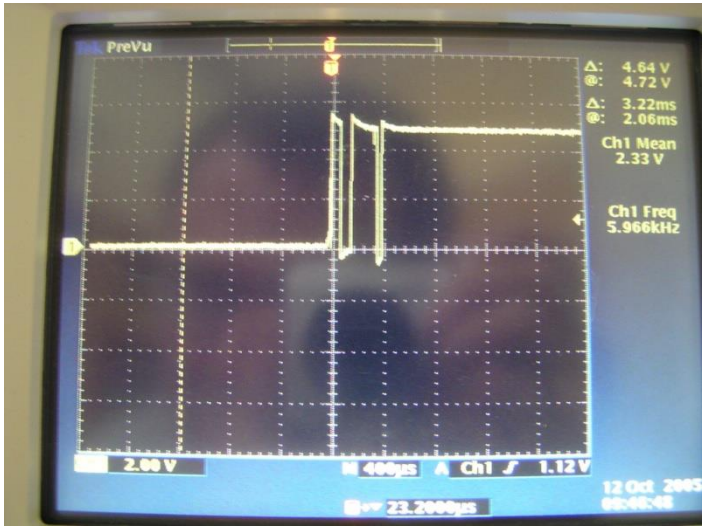
of Distorted Signals



Desired Output

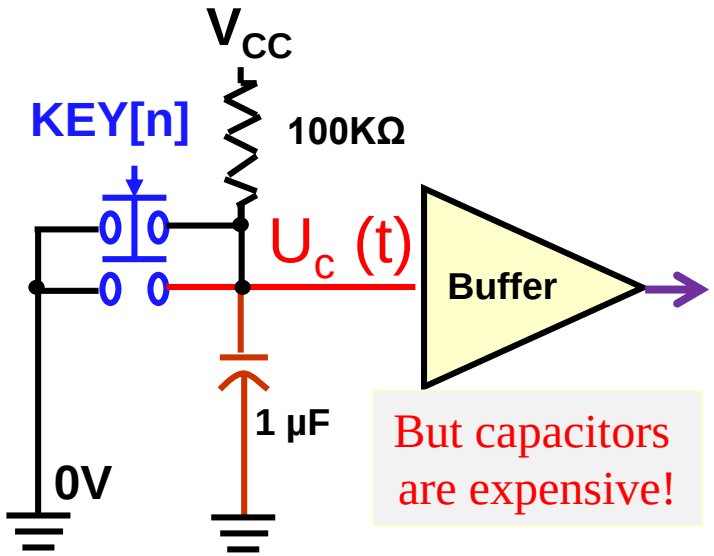


Actual input

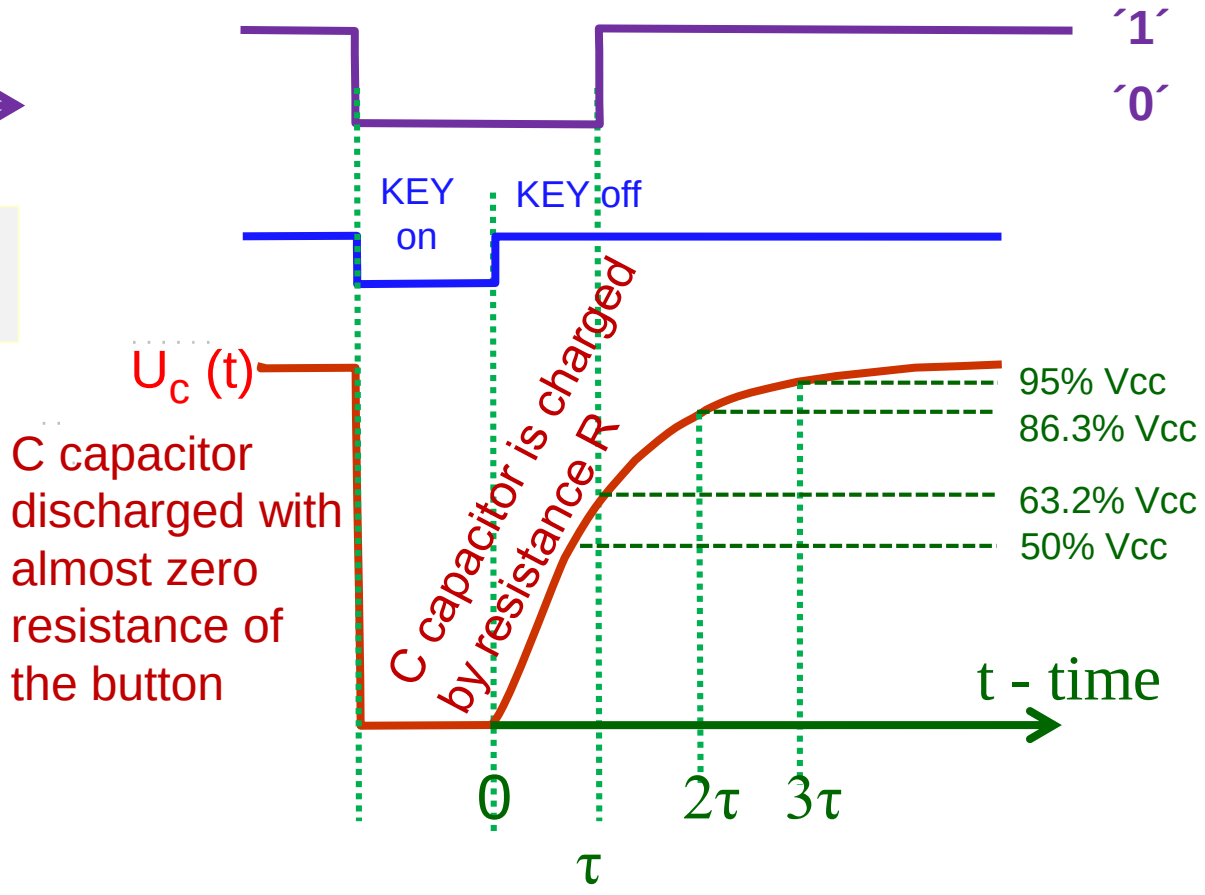


[Source: L. Traylor: Computer Techniques, Oregon 2009]

Analog Debouncing of a Switch



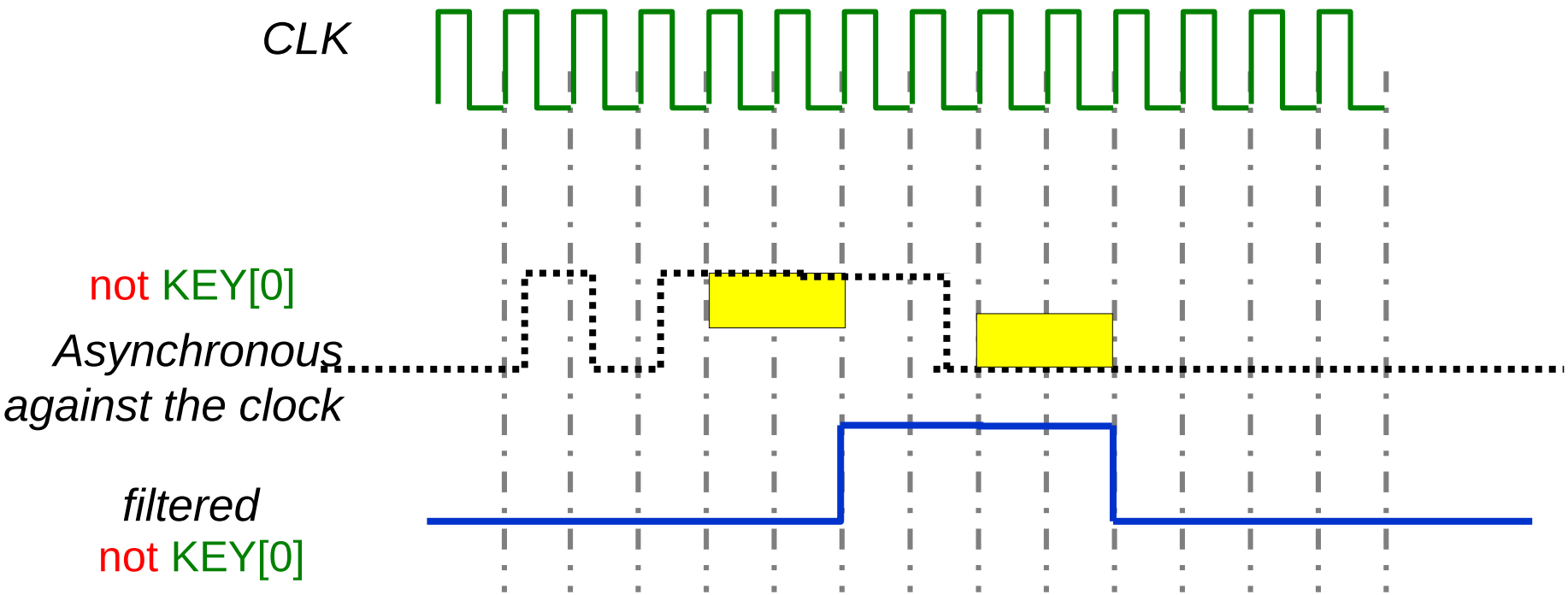
But capacitors are expensive!



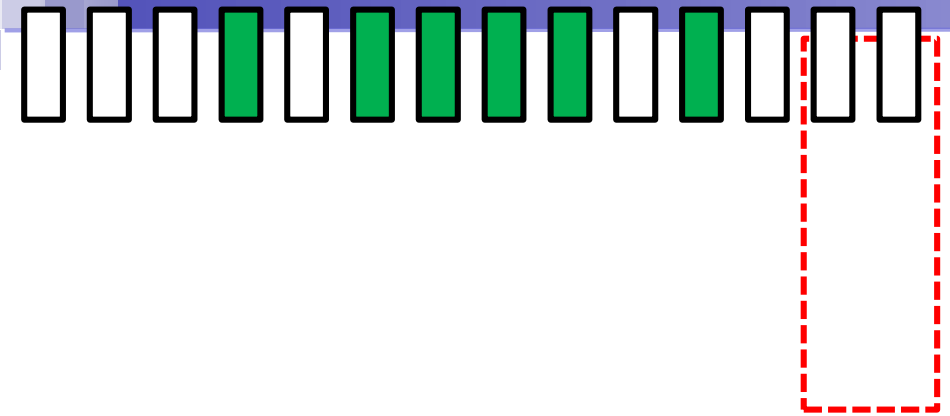
$U_c(t) = V_{CC}\ (1 - e^{-t/\tau})$,

$\tau = 0.69\ RC \approx \mathbf{0.7\ RC} = 0.7 * \mathbf{100K\Omega * 1\ \mu F} = 0.7 * 10^5 * 10^{-6} = 70$ ms

The switch can be debounced by many methods. For example, we can detect stable values over multiple clock cycles. In programs, keyboards are read in this way.

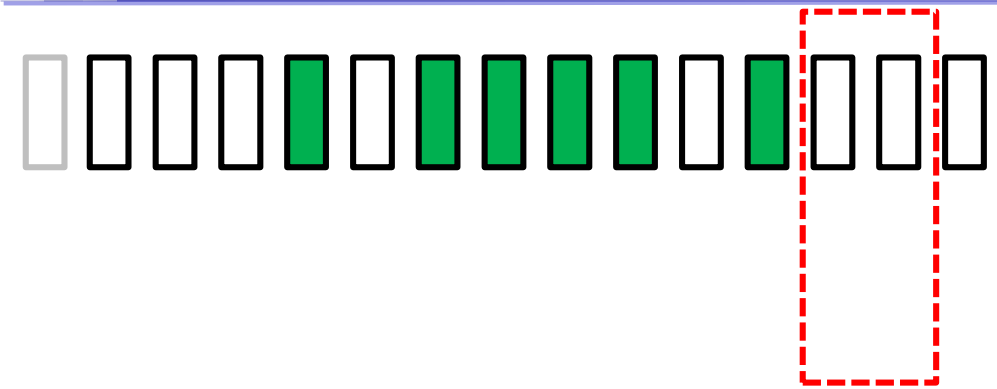


Filtration length 2



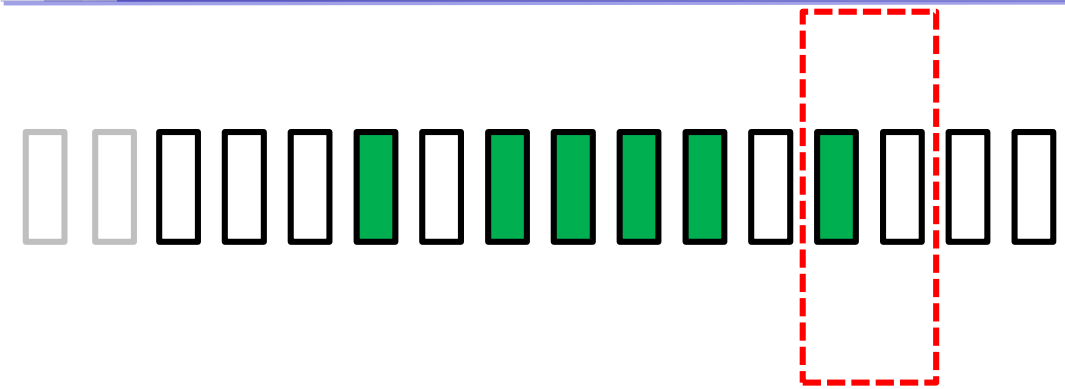
time

Filtration length 2



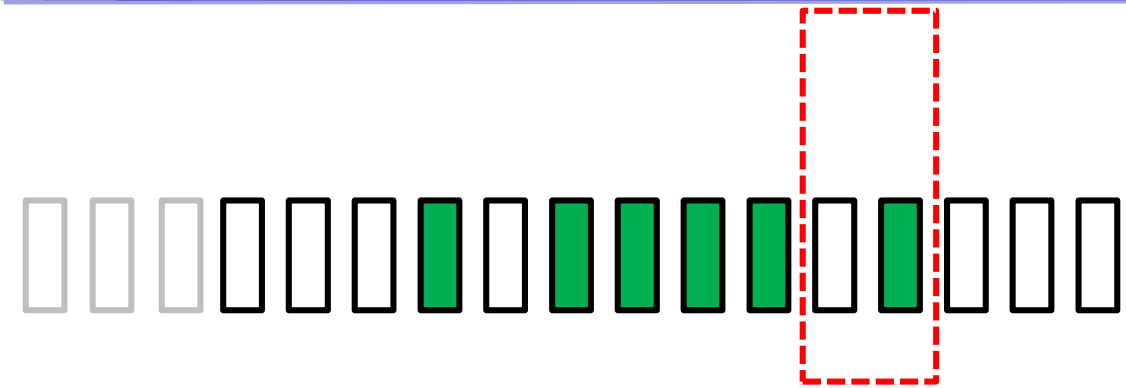
time

Filtration length 2



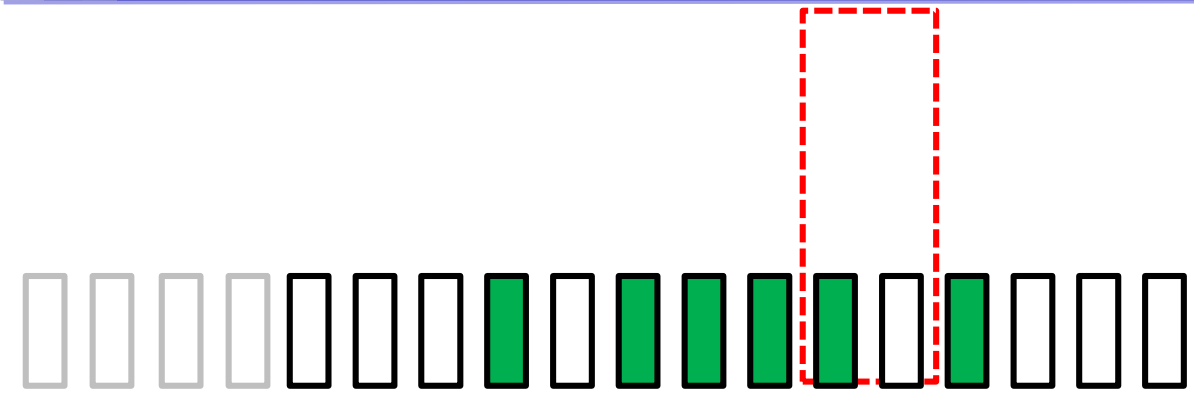
time

Filtration length 2



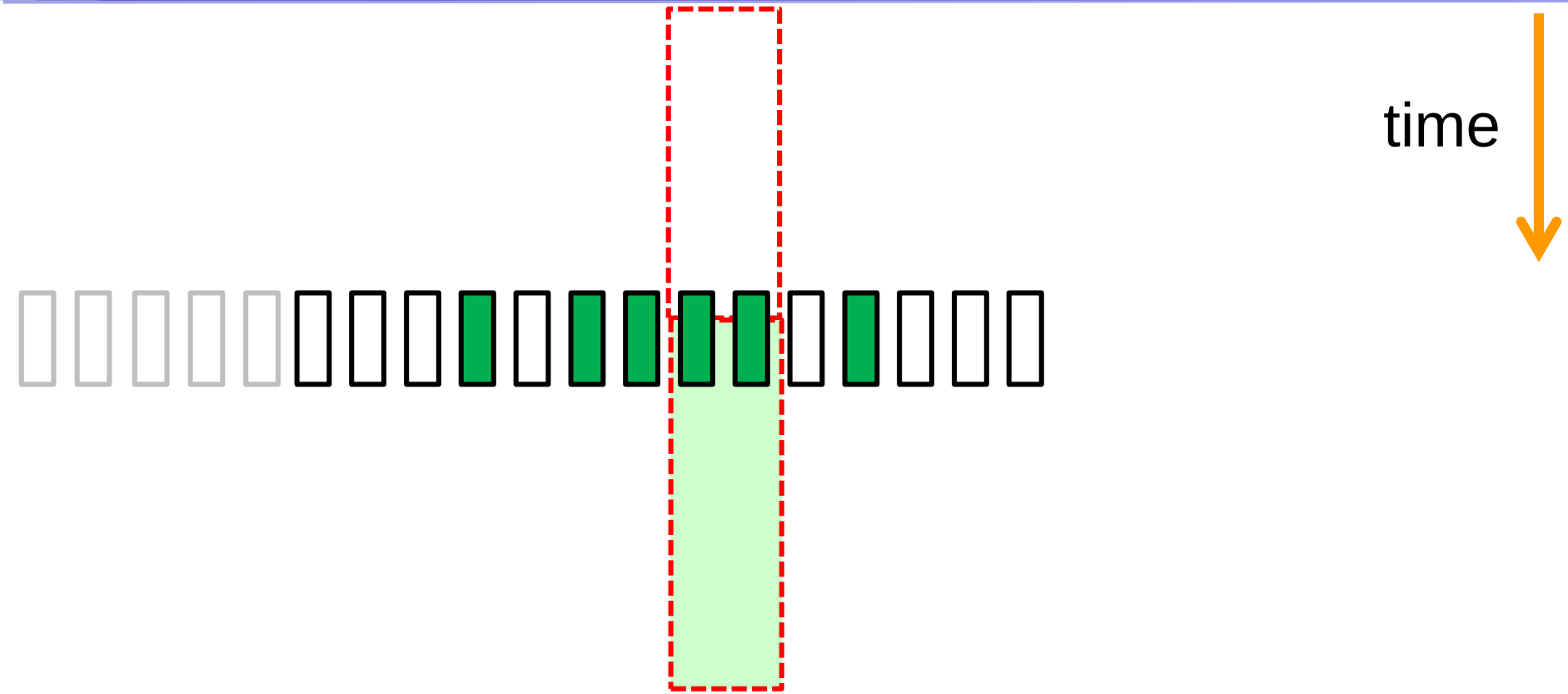
time

Filtration length 2

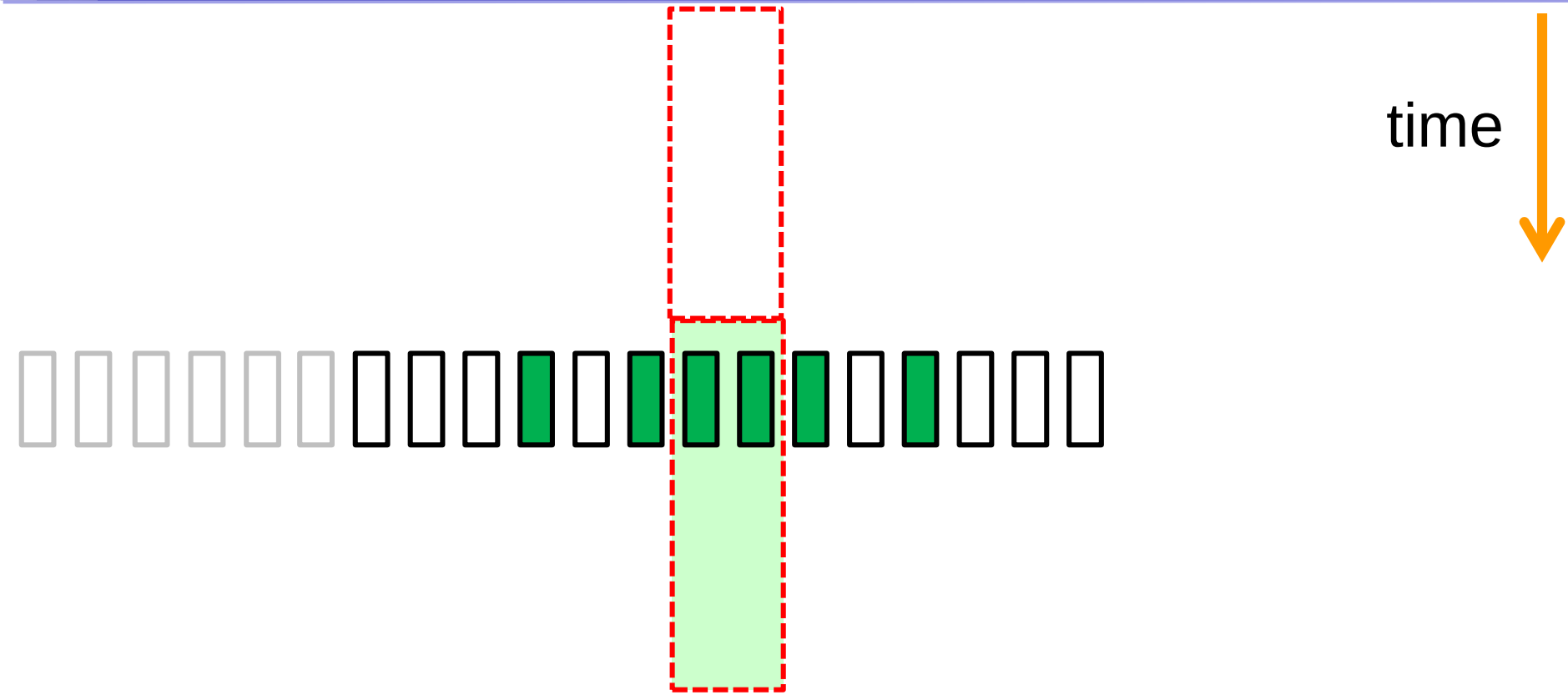


time

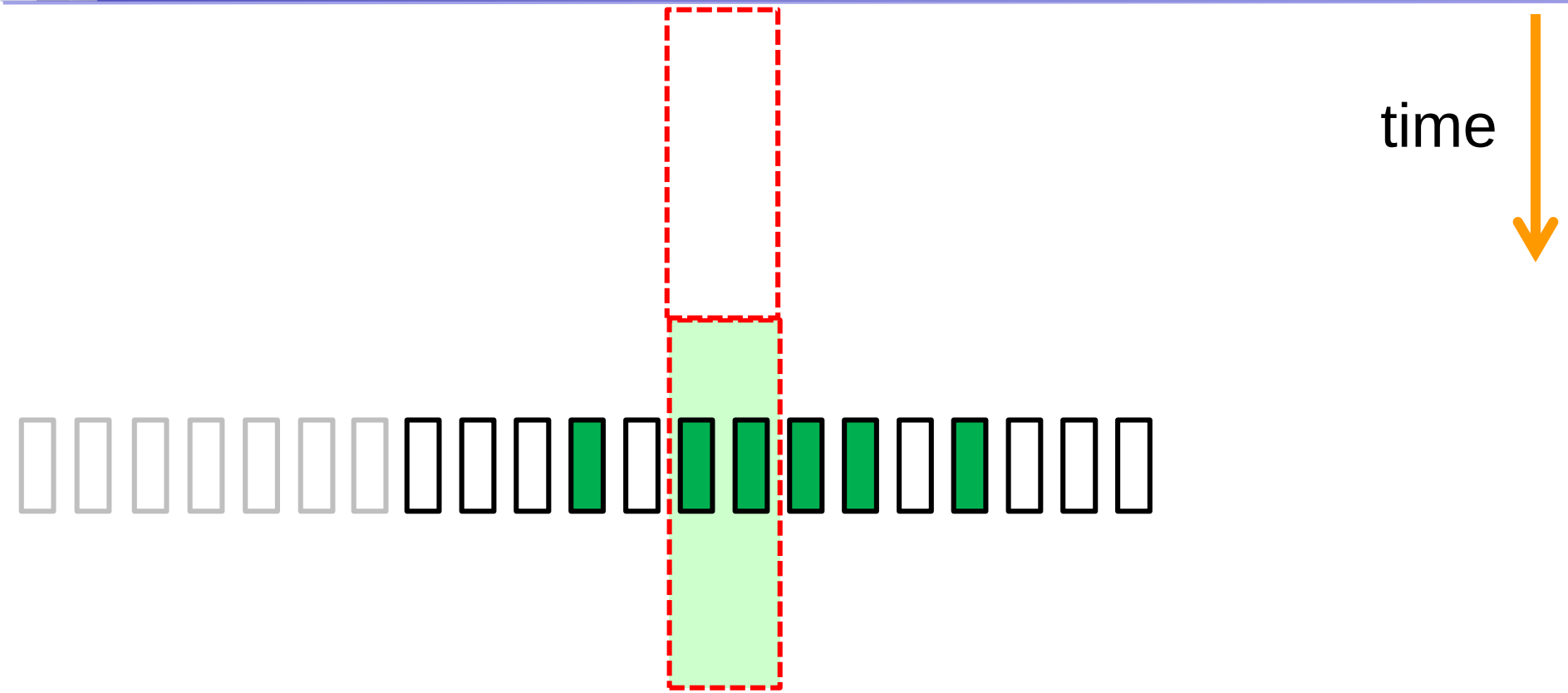
Filtration length 2



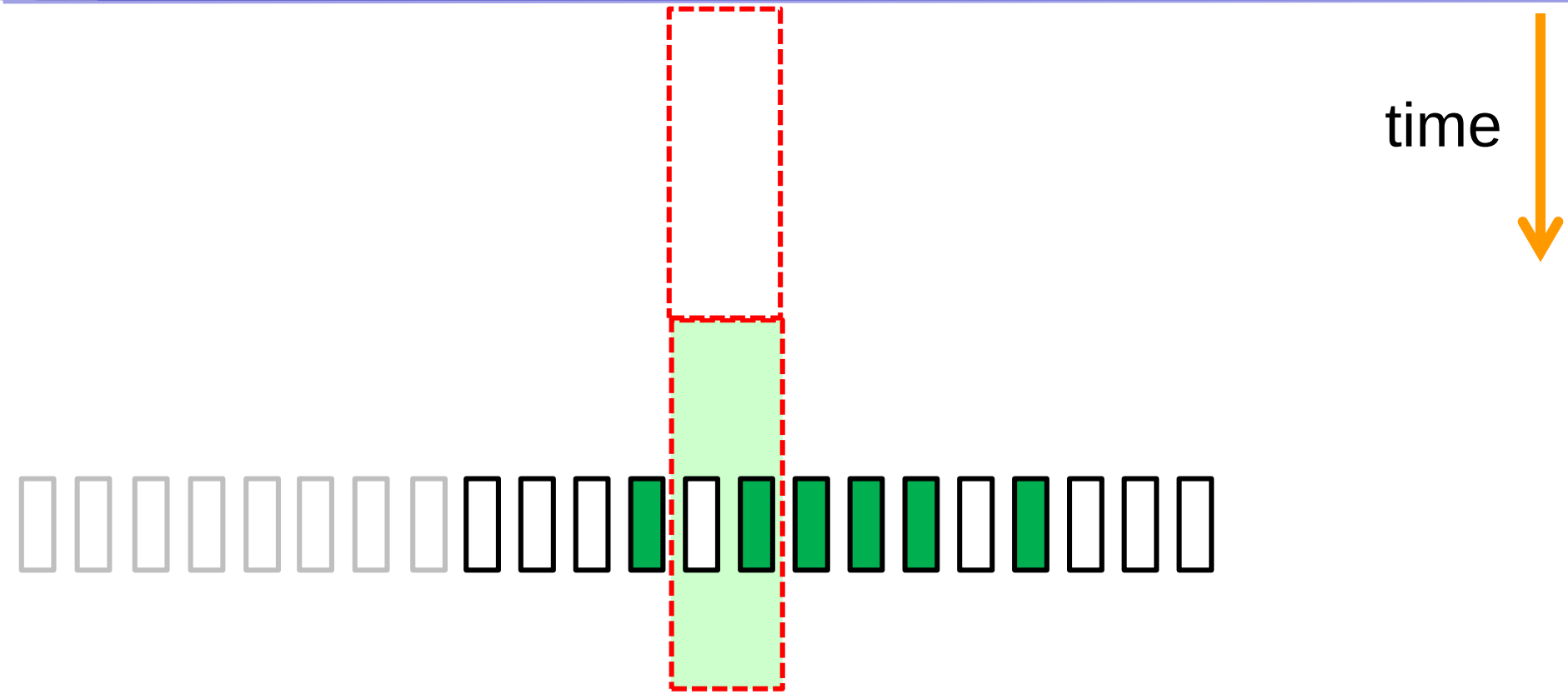
Filtration length 2



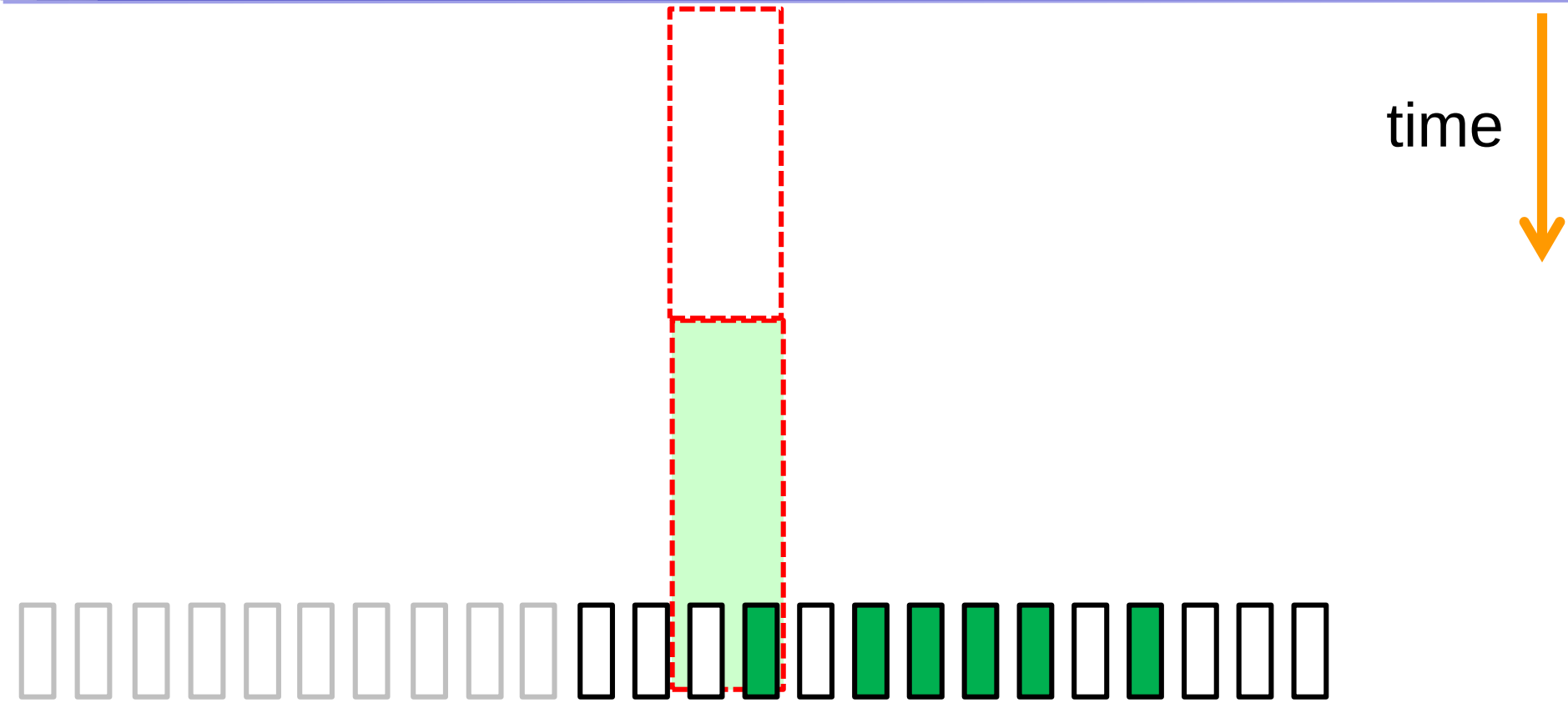
Filtration length 2



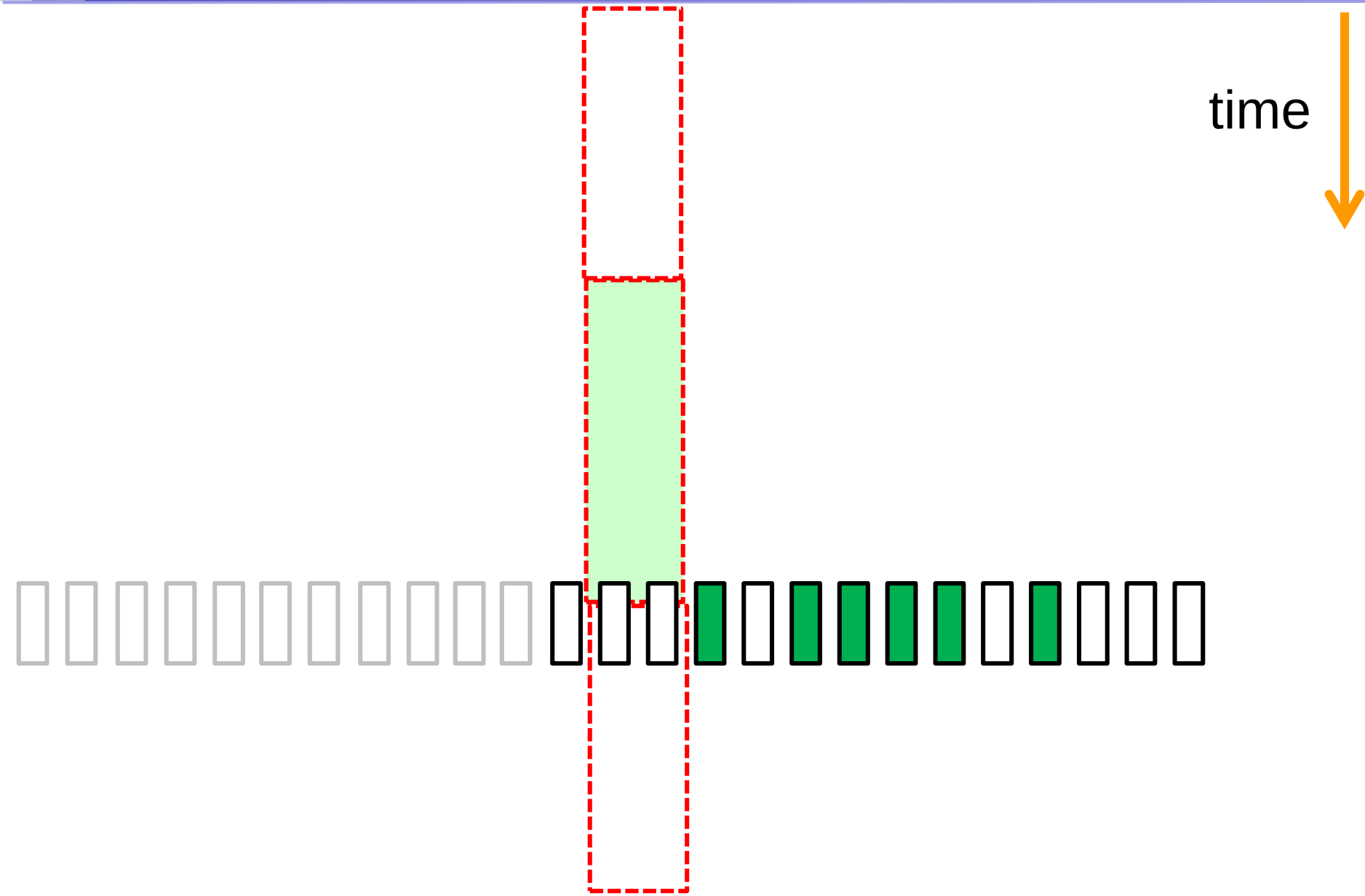
Filtration length 2



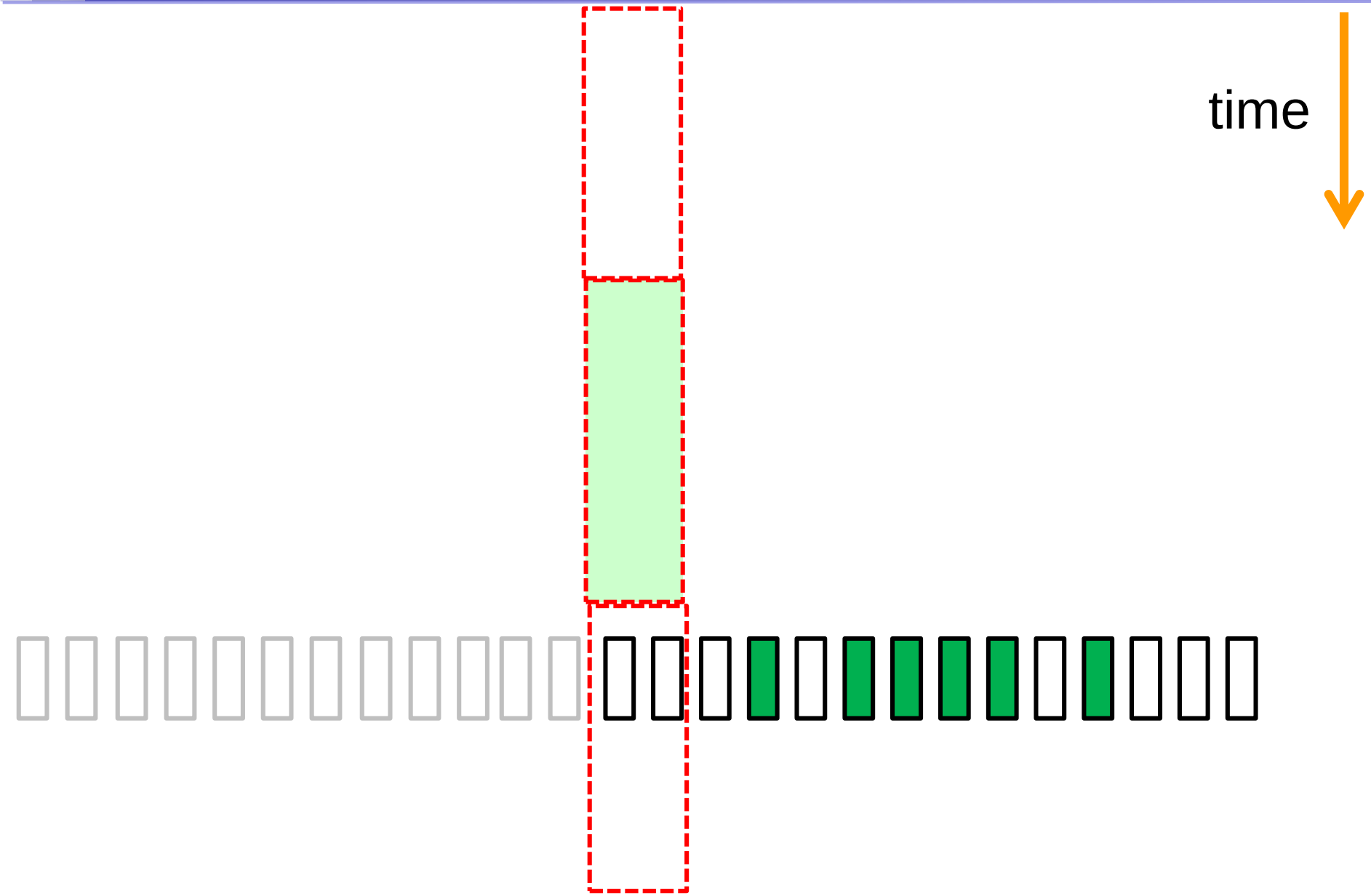
Filtration length 2



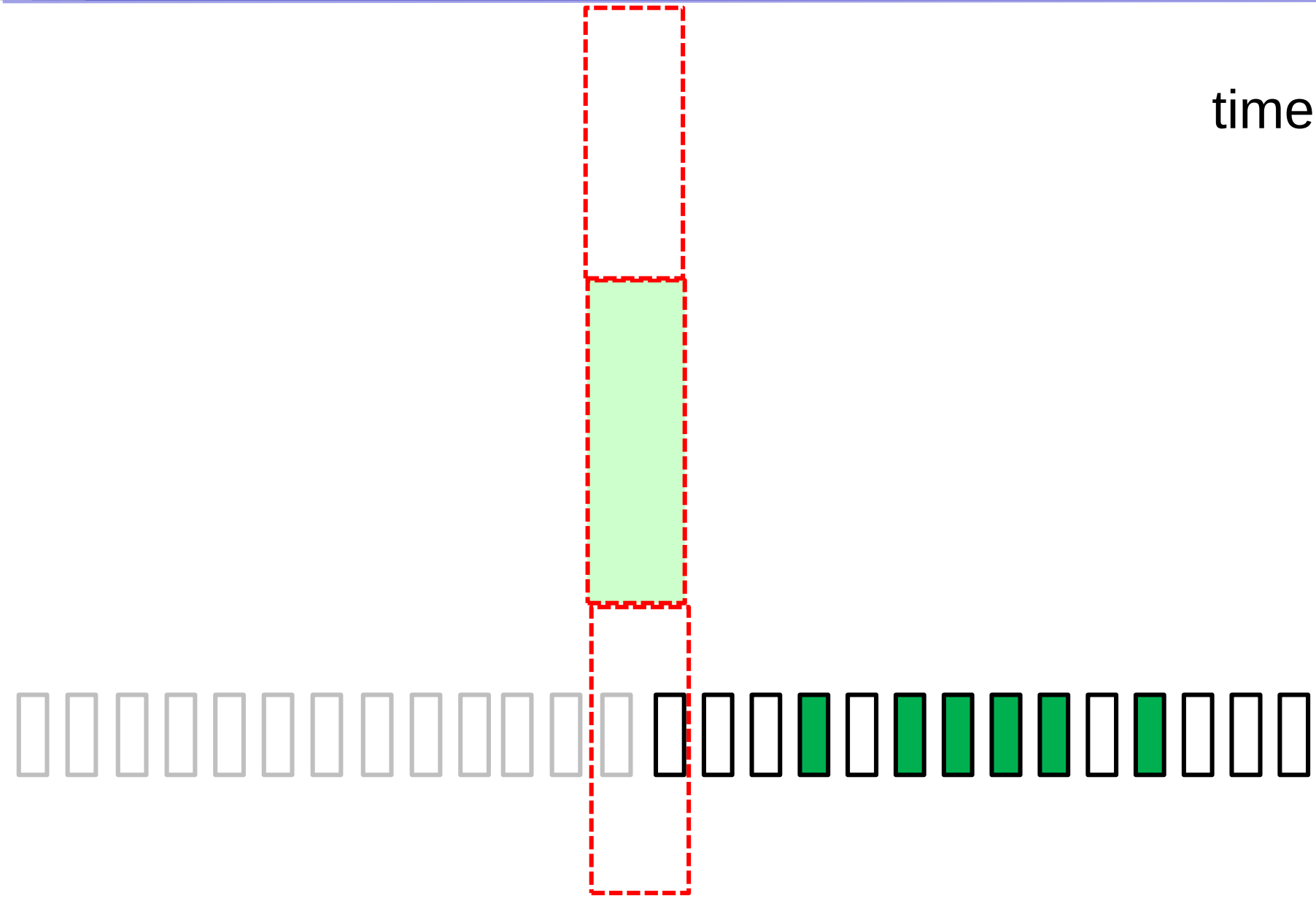
Filtration length 2



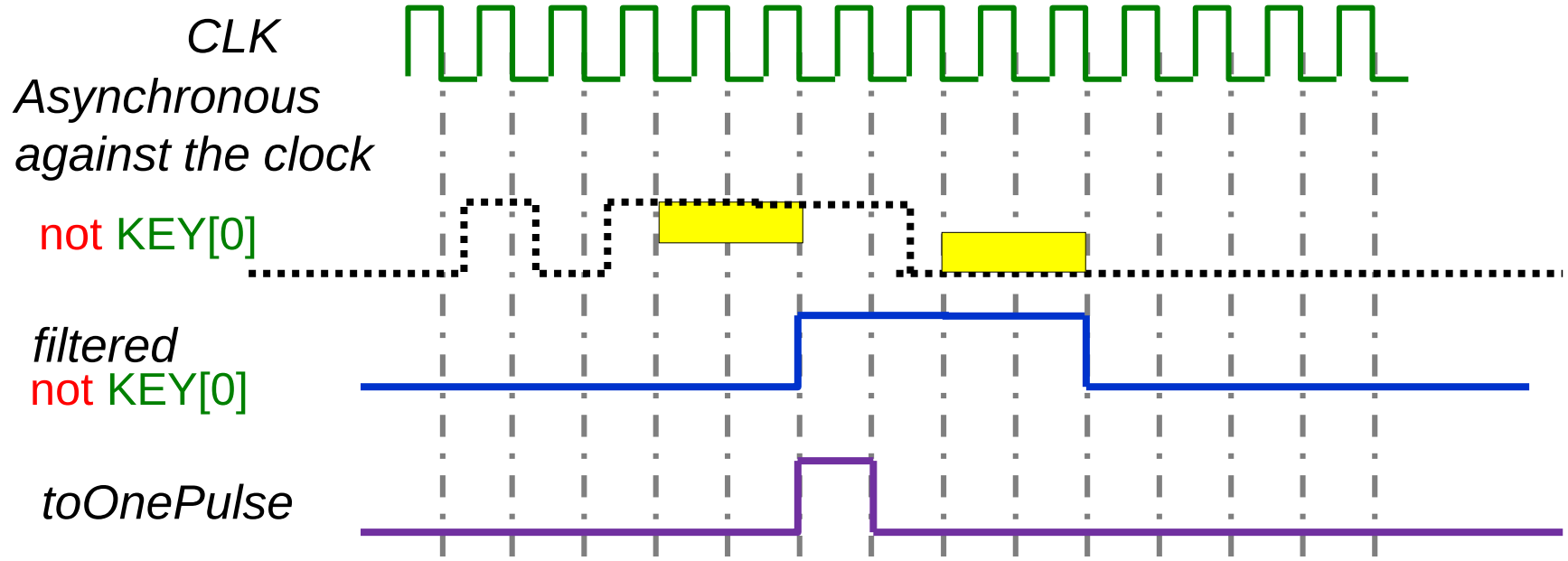
Filtration length 2



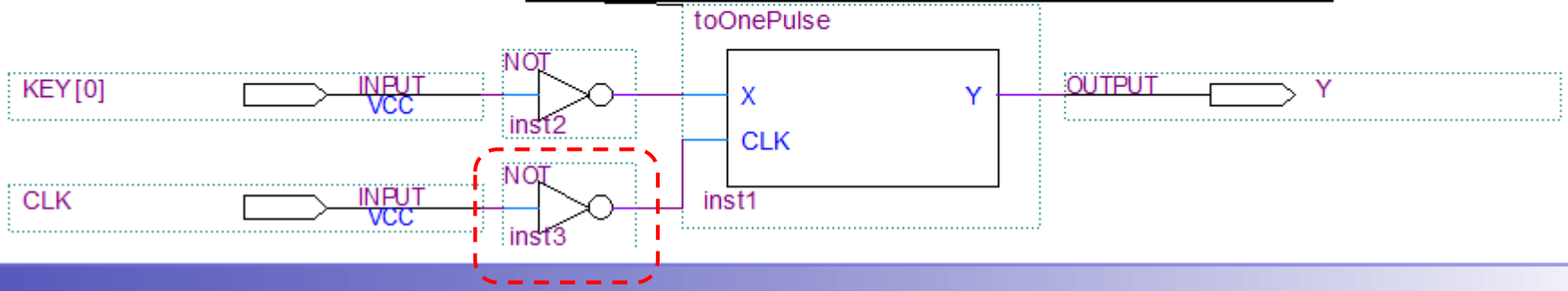
Filtration length 2



*It shifts inputs on the **falling** edges. Thus, its filtered output is **stable** for some time when the **rising** edges arrive.*

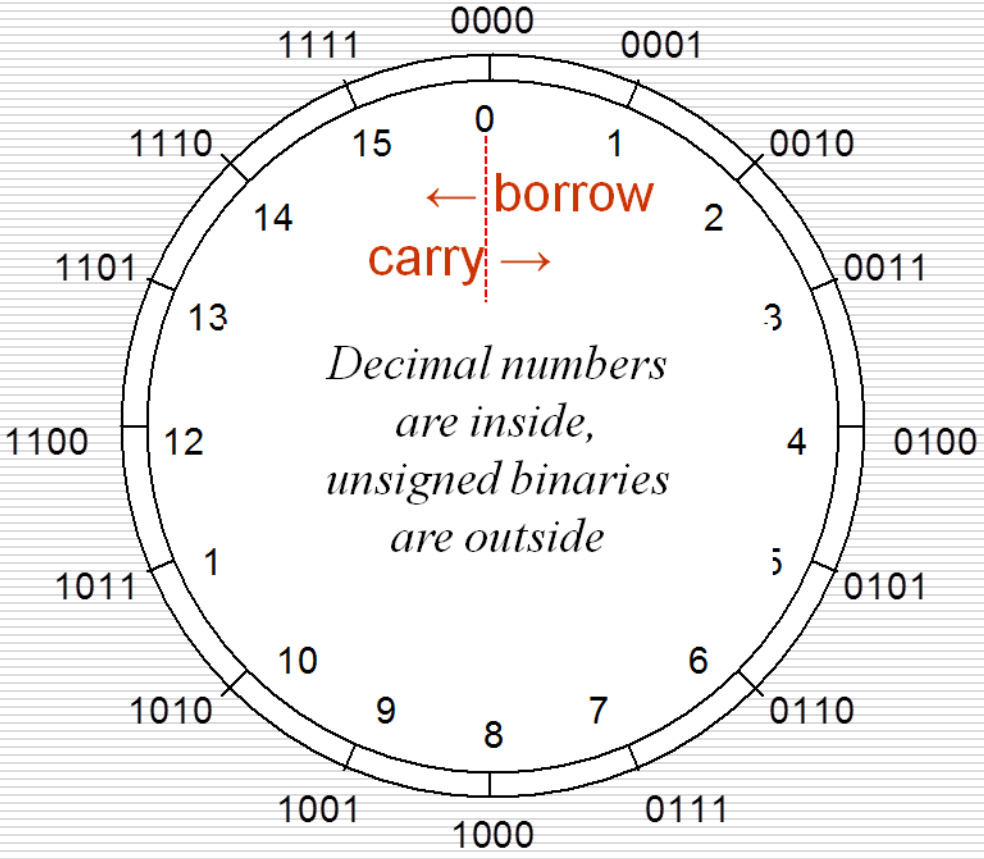


Parameter	Value	Type
DIGITAL_FILTER_LENGTH	2	Signed Integer

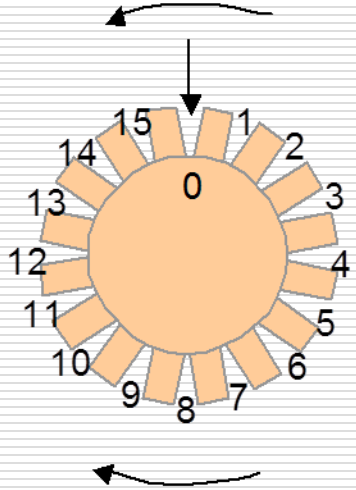


Counters
and **Frequency Dividers**

or Clock Dividers or Scallers



We add by turning counterclockwise



We subtract by turning clockwise

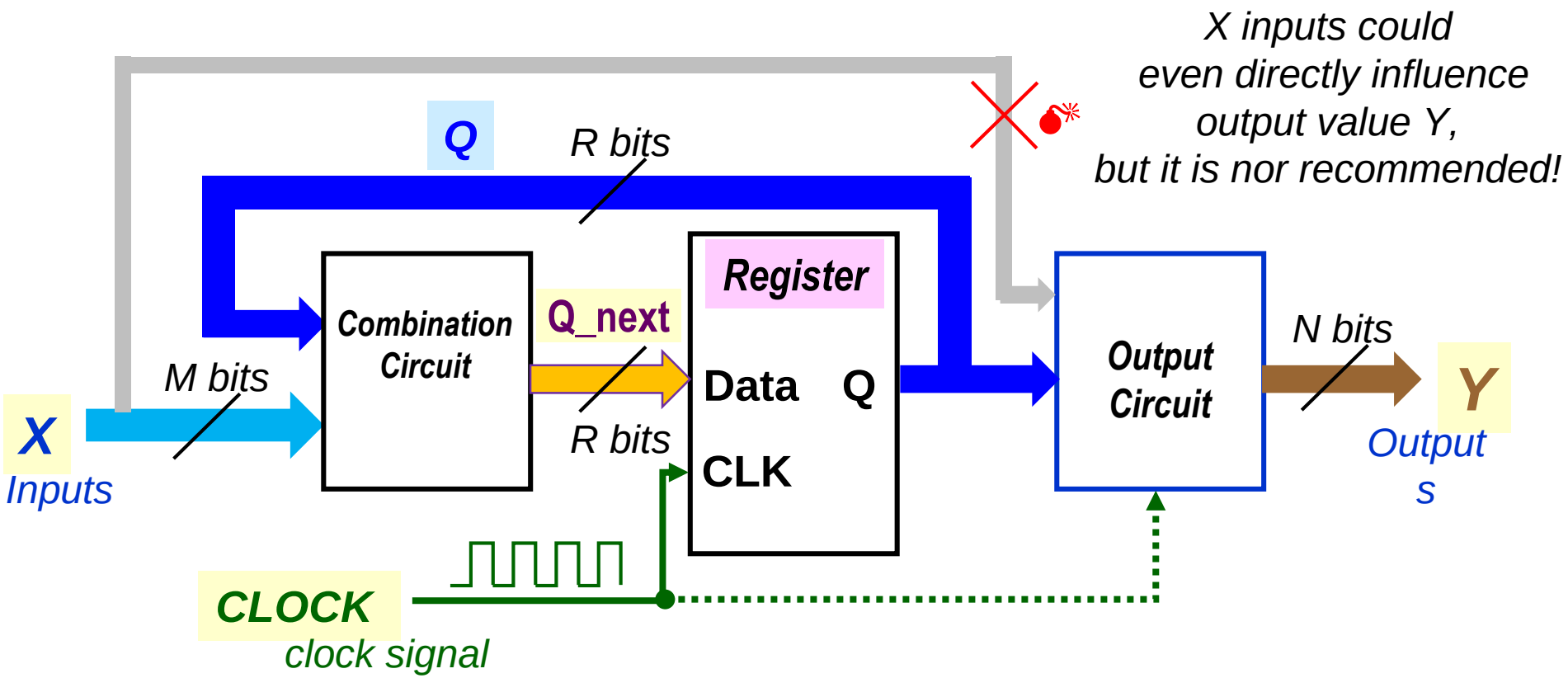
- Let the frequency of the clock signal at CLK input is $\mathbf{f_{clk}}$.
- If we use the highest active bit of the counter as an output signal, then it surely changes from ´0´ to ´1´ and from ´1´ to ´0´ after some count. Let us denote the frequency of this signal as $\mathbf{f_{out}}$.
- If the binary counter counts by 1's in the range 0 to M-1, where M is any natural number greater than 1, then the counter also divides the input frequency by M:

$$f_{out} = \frac{f_{clk}}{M}$$

Since the internal design is nearly identical, the word divider (frequency) is sometimes used as a synonym for a binary counter incrementing by +1.

They differ only in their possibilities. Dividers can be considered as simplified counter because they typically have only a 1-bit output but counters more bit binary. Counters allow clearing or presetting of their momentary count, changing direction up/down of counting, etc.

Synchronous Circuit with FSM Structure



FSM = Finite State Machine

Design a 2-bit counter

- 00 -> 01 -> 10 -> 11 -> 00 -> 01 ... etc.

Solution:

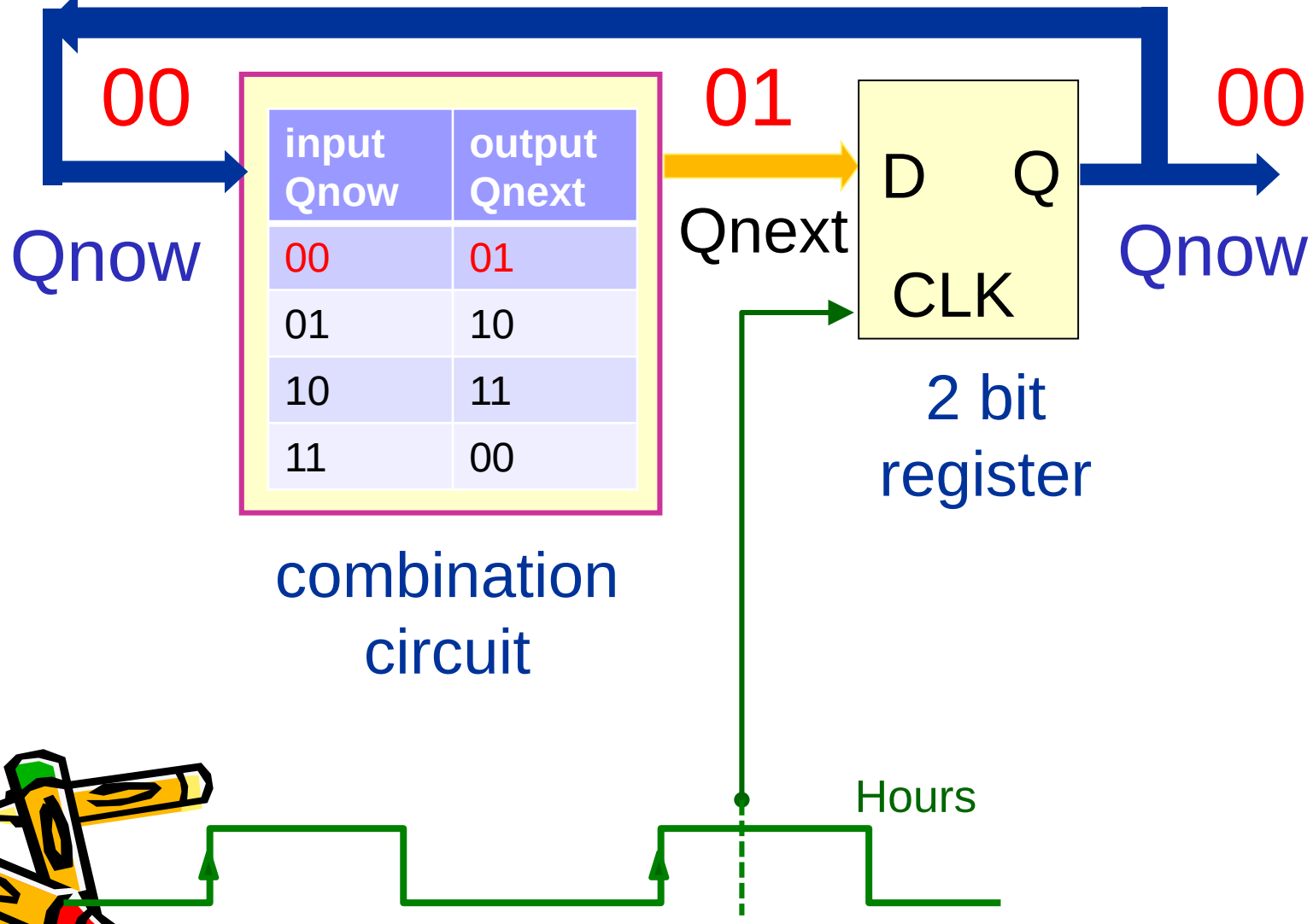
1. Build the combinational circuit of the +1 adder

input Qnow	output Qnext
00	01
01	10
10	11
11	00

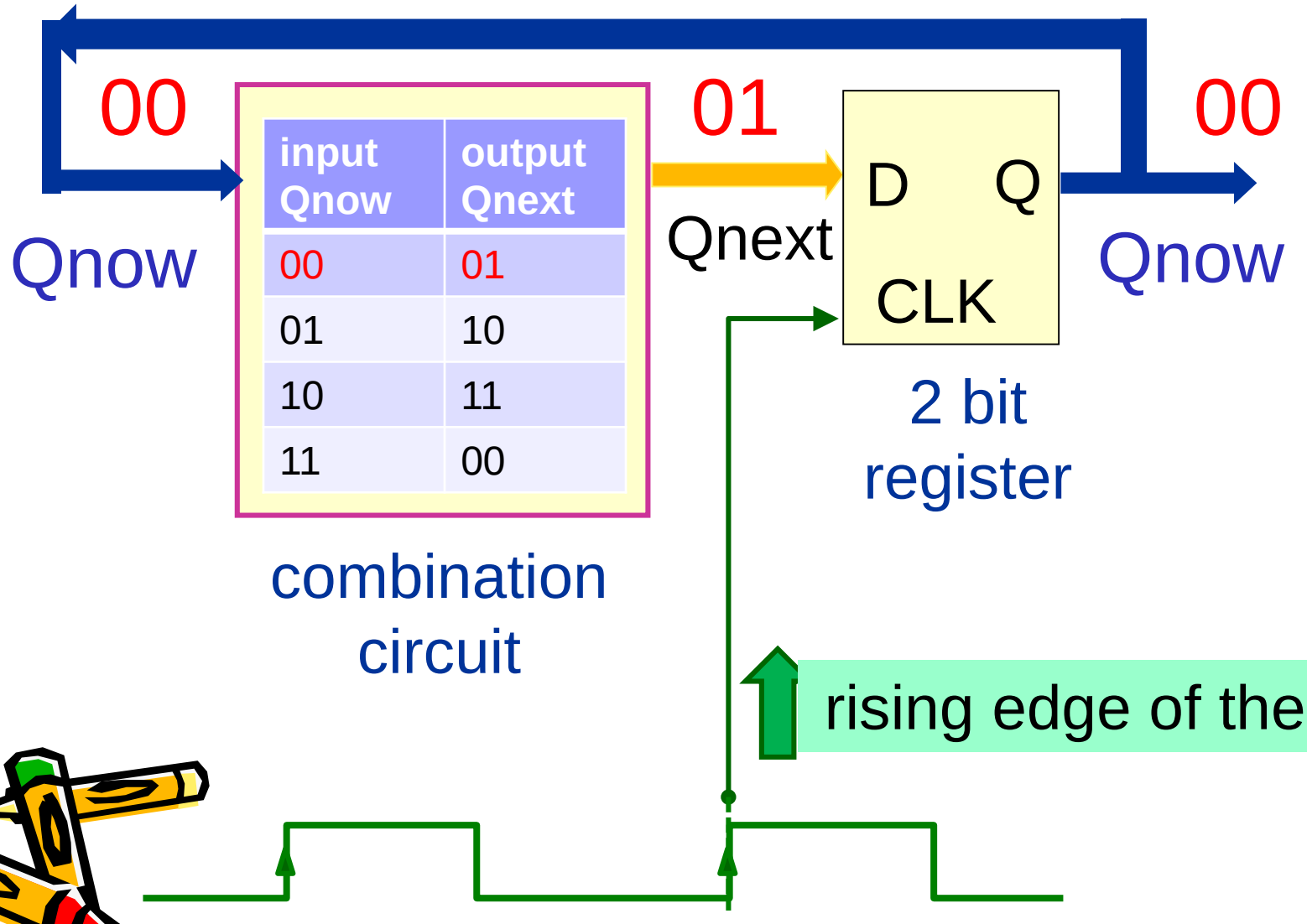
2. Use the adder as a combinational circuit of next Q.

2-bit 1/12 counter

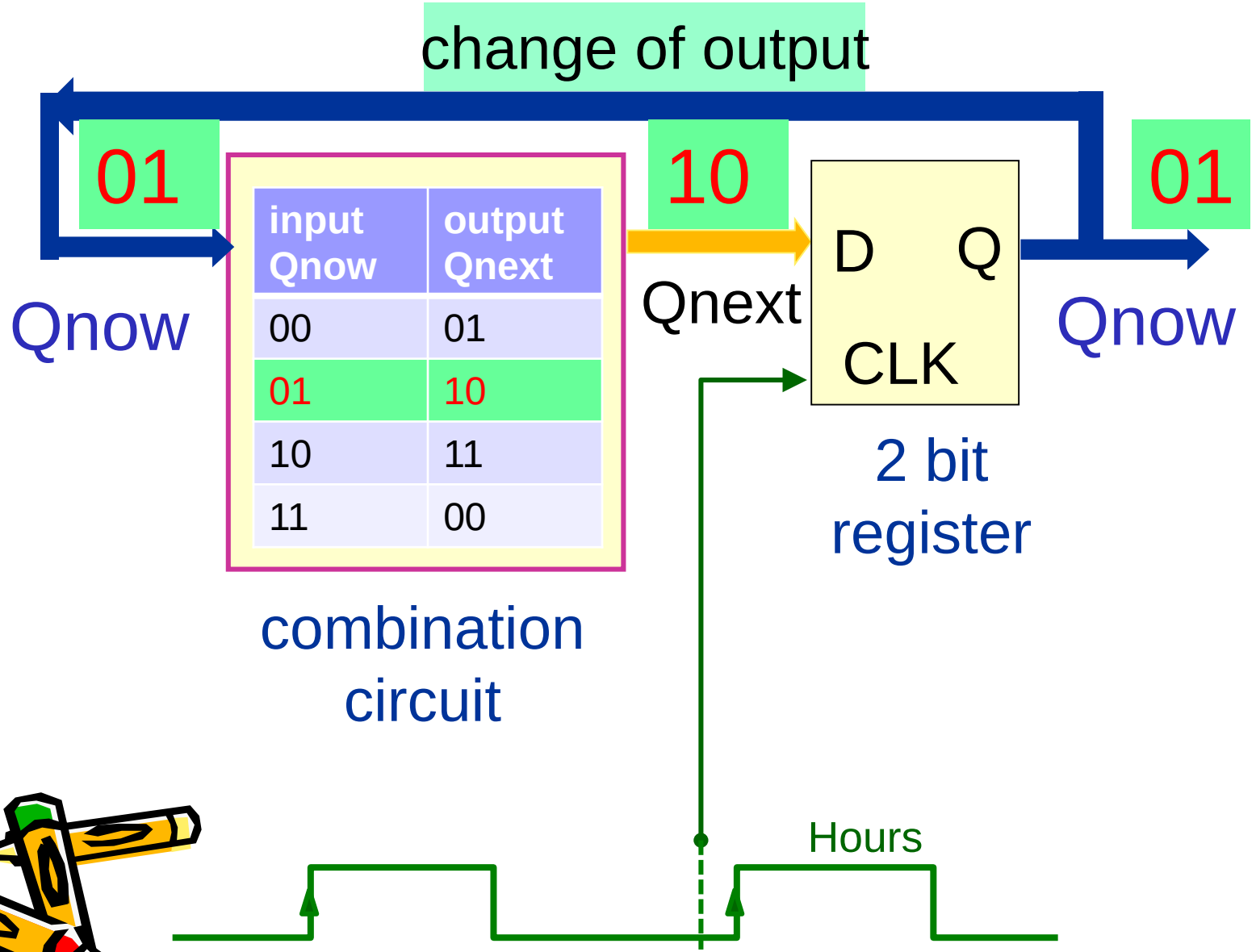
steady state



2-bit counter 2/12

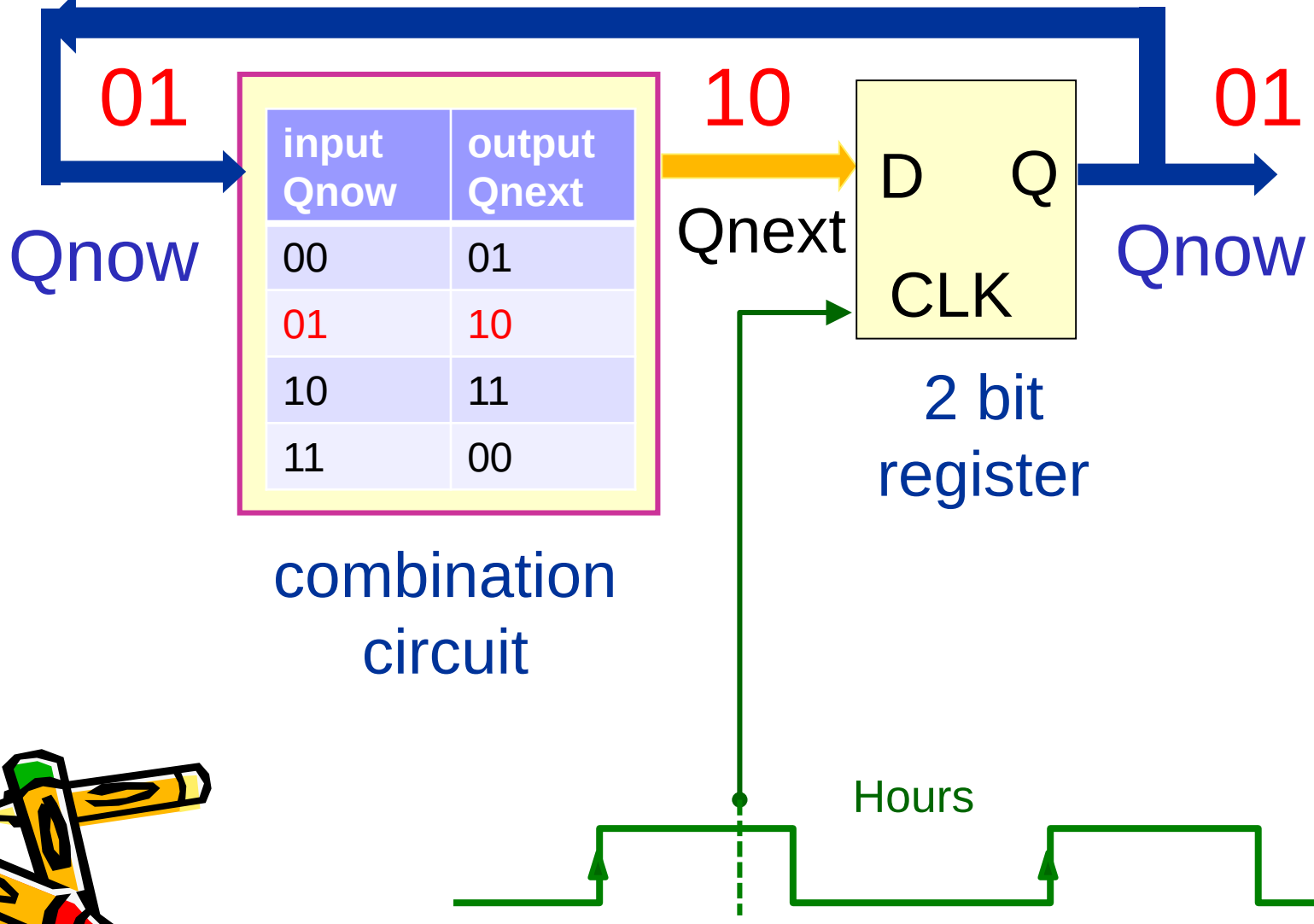


2-bit counter 3/12

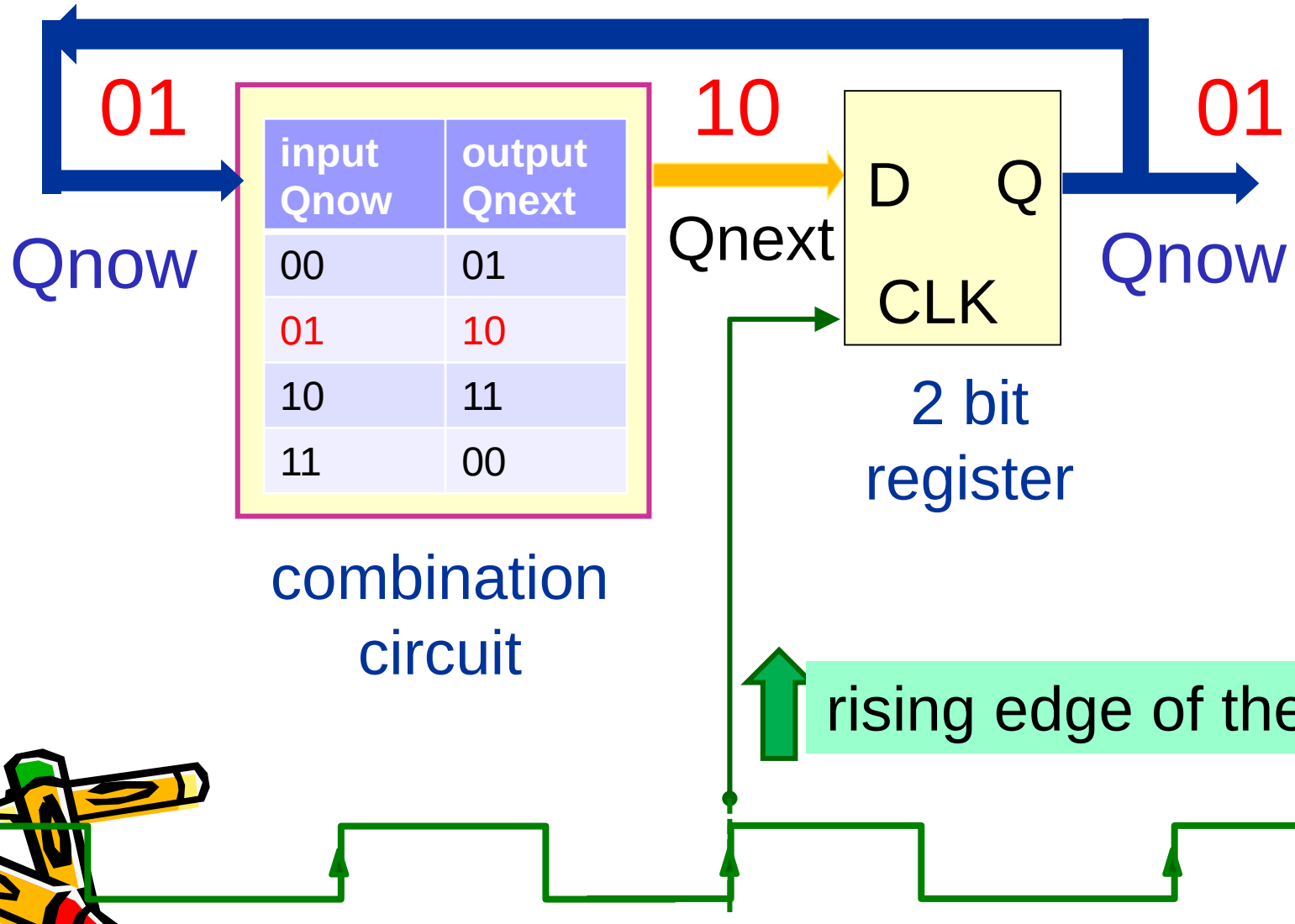


2-bit counter 4/12

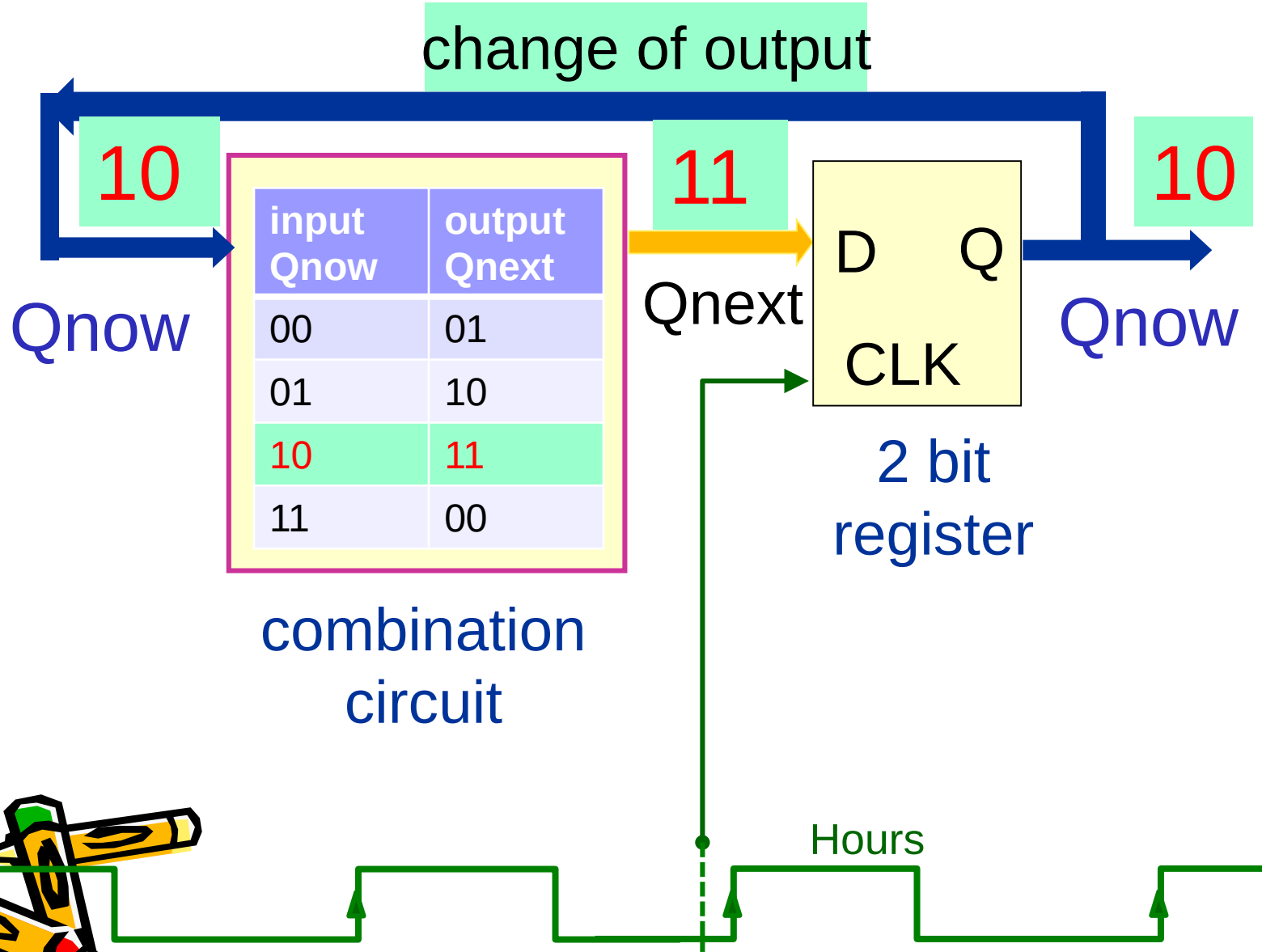
steady state



2-bit 5/12 counter



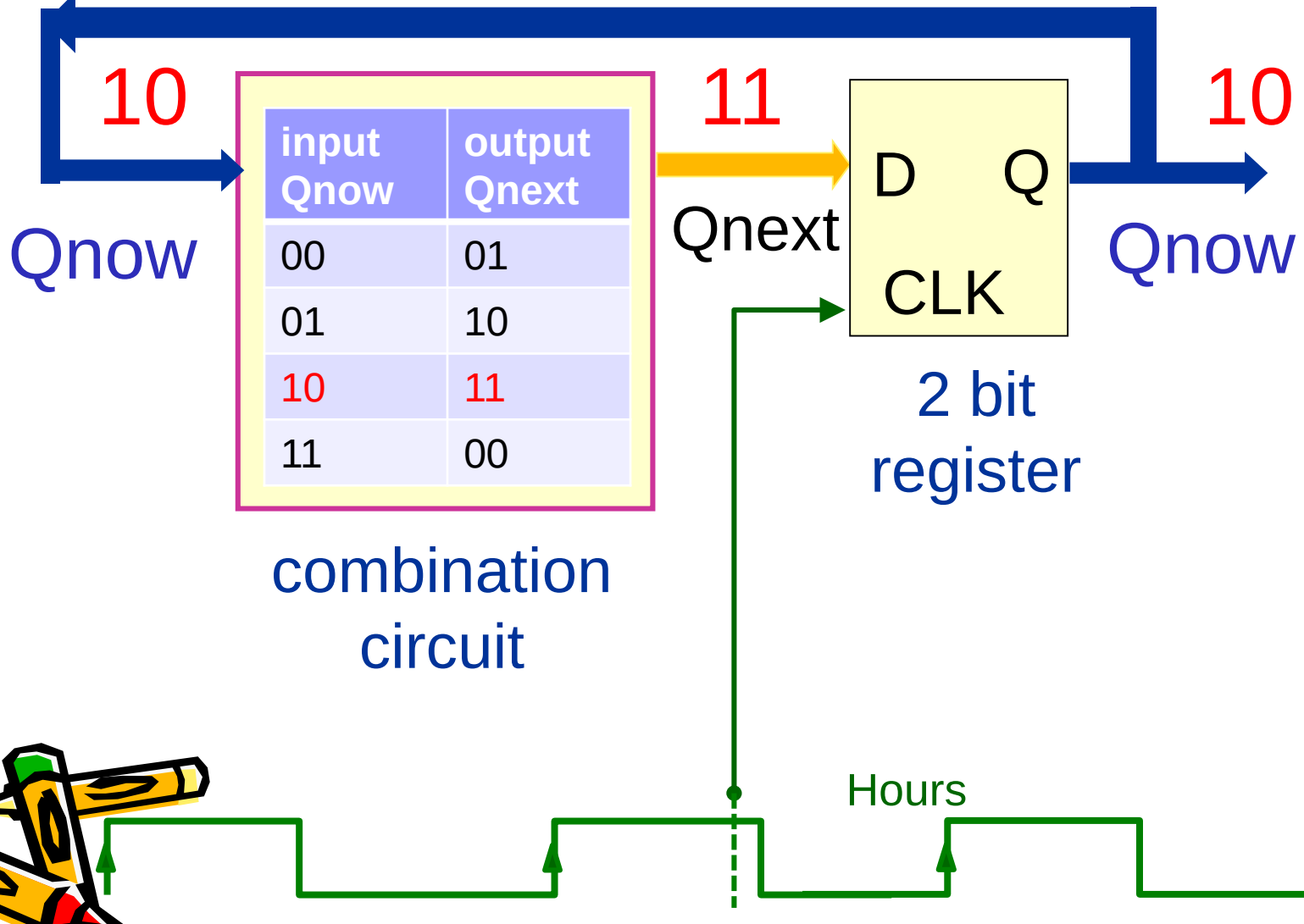
2-bit counter 6/12



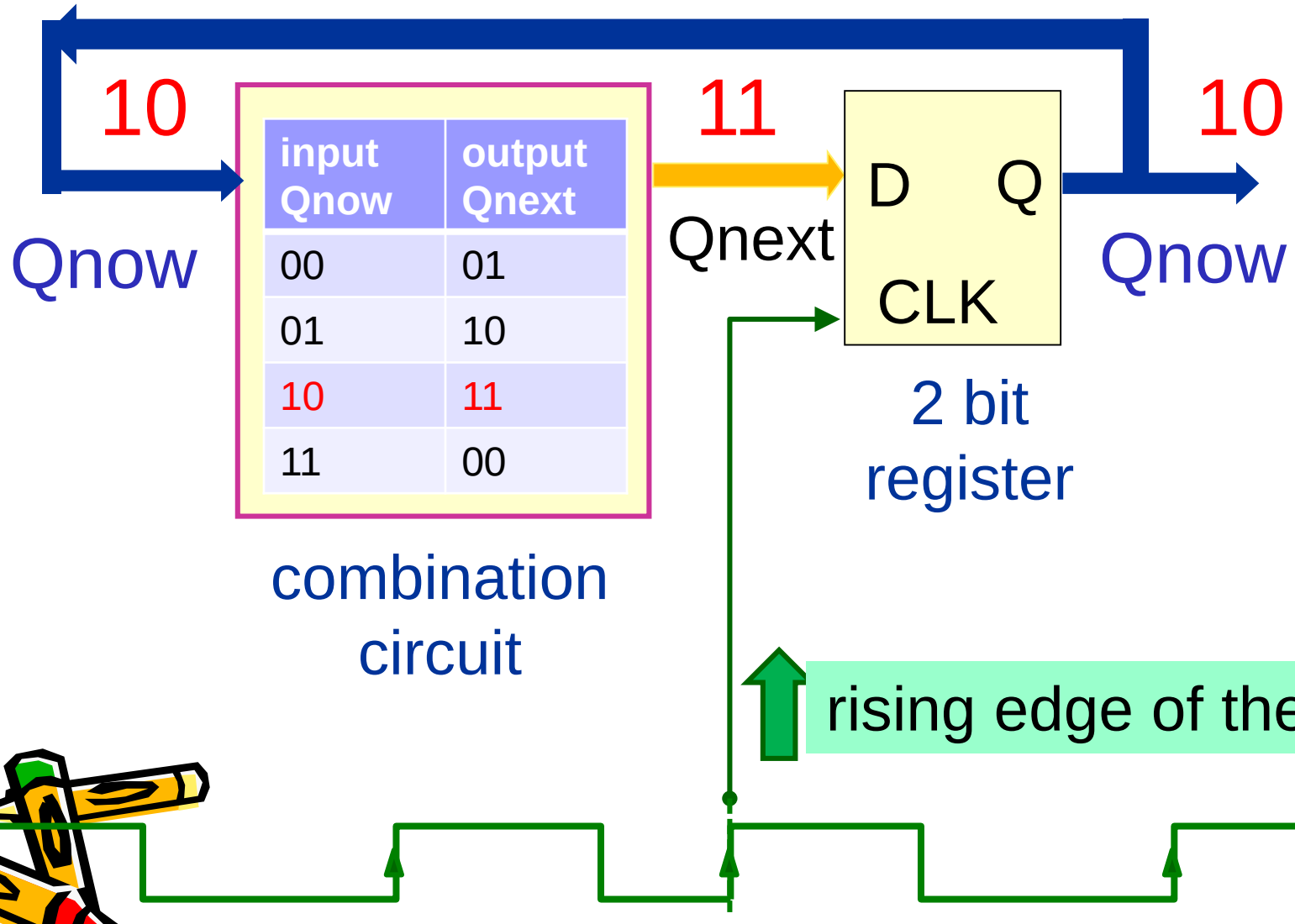
input Qnow	output Qnext
00	01
01	10
10	11
11	00

2-bit counter 7/12

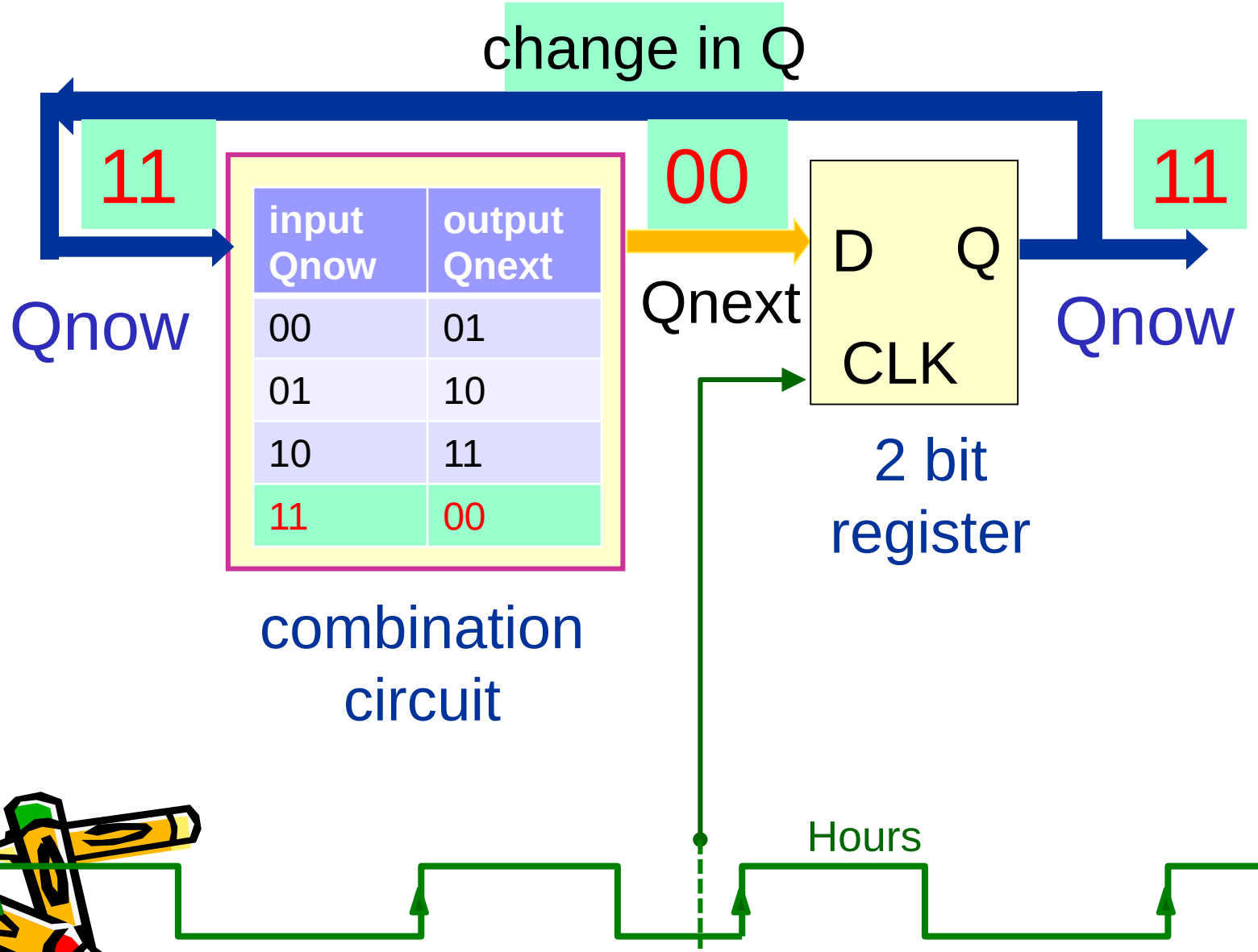
steady state



2-bit 8/12 counter

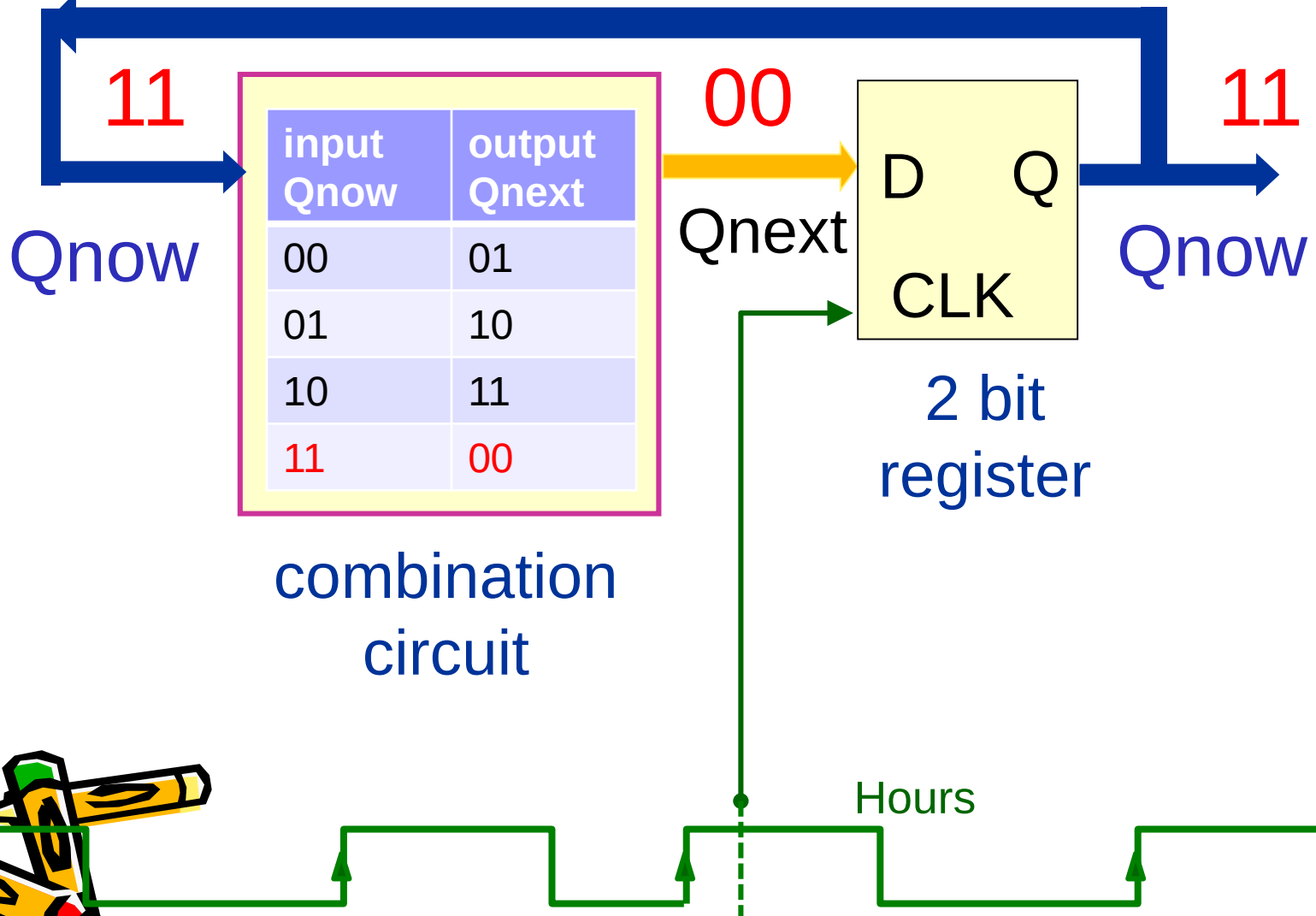


2-bit counter 9/12

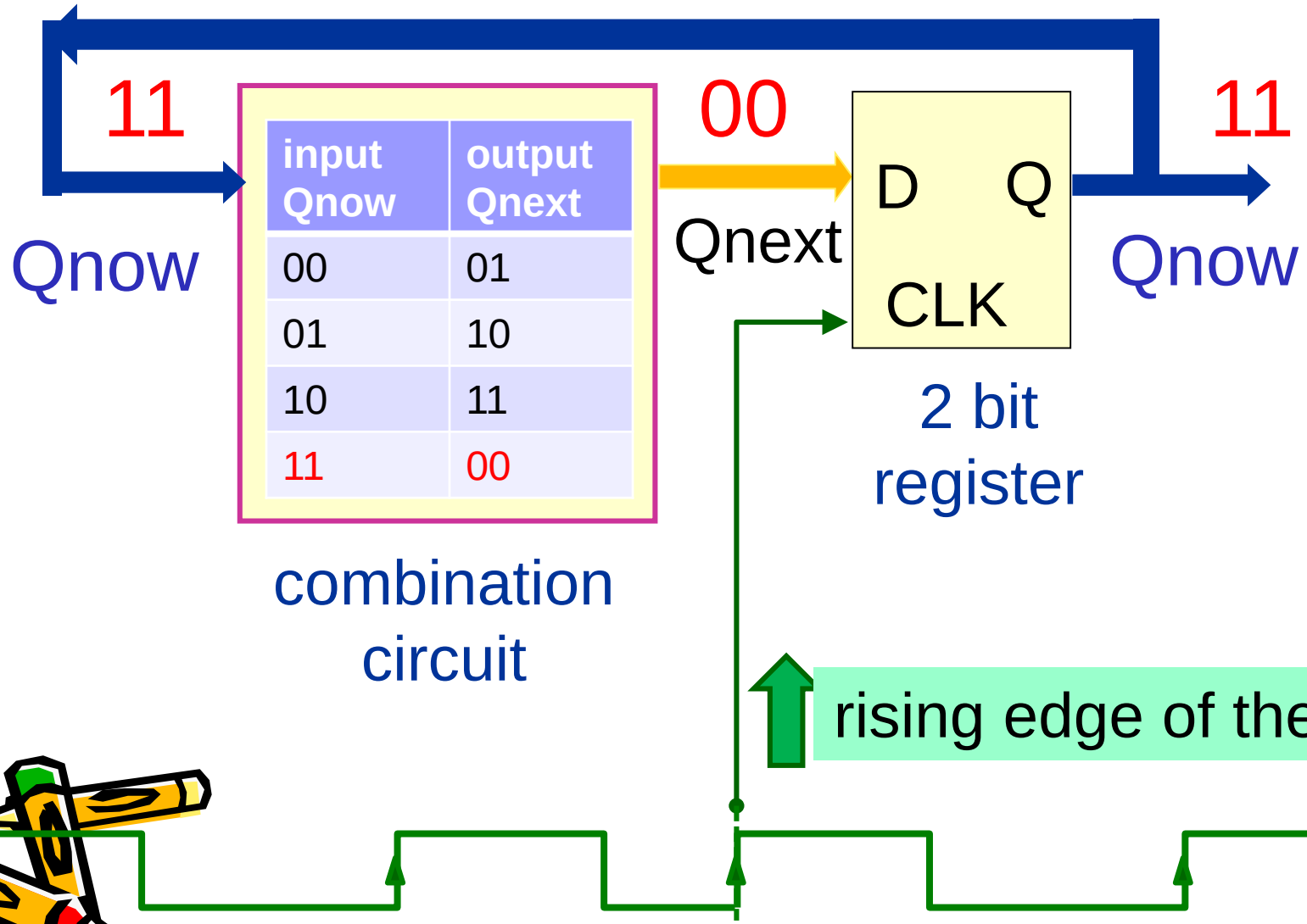


2-bit counter 10/12

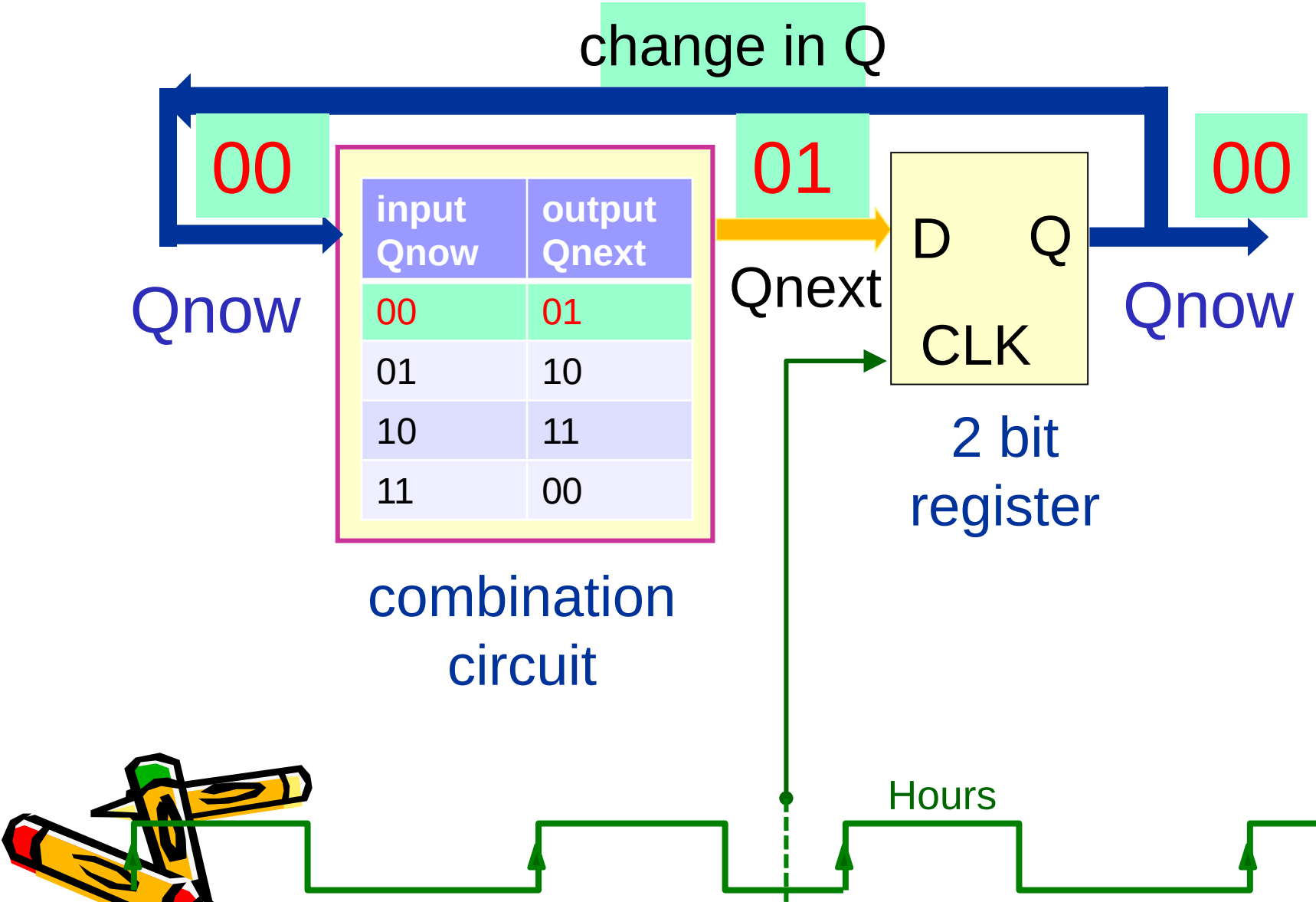
steady state



2-bit counter 11/12

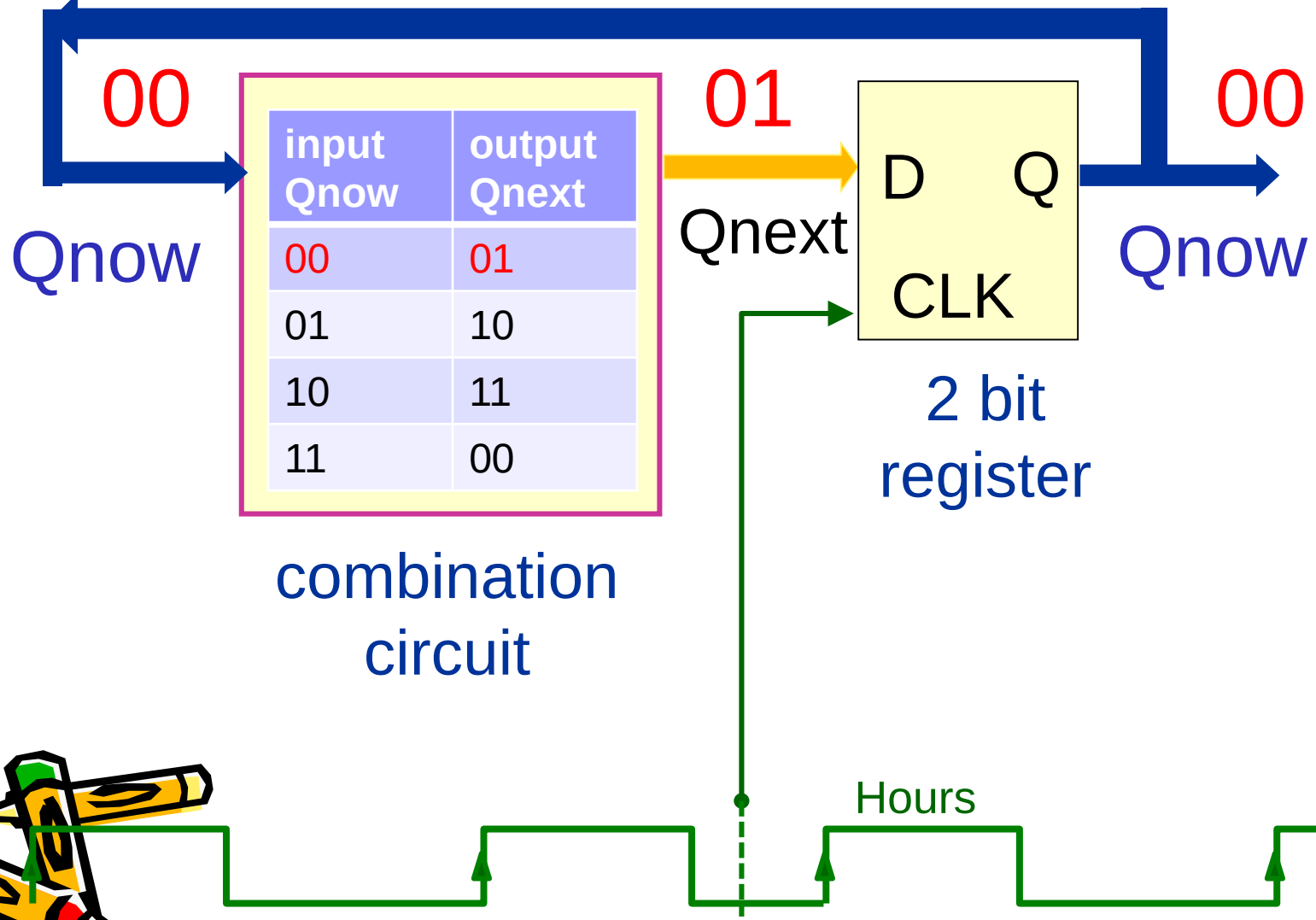


2-bit 12/12 counter



2-bit counter - again 1/12

steady state



6-bit Counter for Morse Beacon