Logic Gates & Flip-Flops

Paolo PRINETTO

Director
CINI Cybersecurity
National Laboratory

Paolo.Prinetto@polito.it

Mob. +39 335 227529

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity
National Lab

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

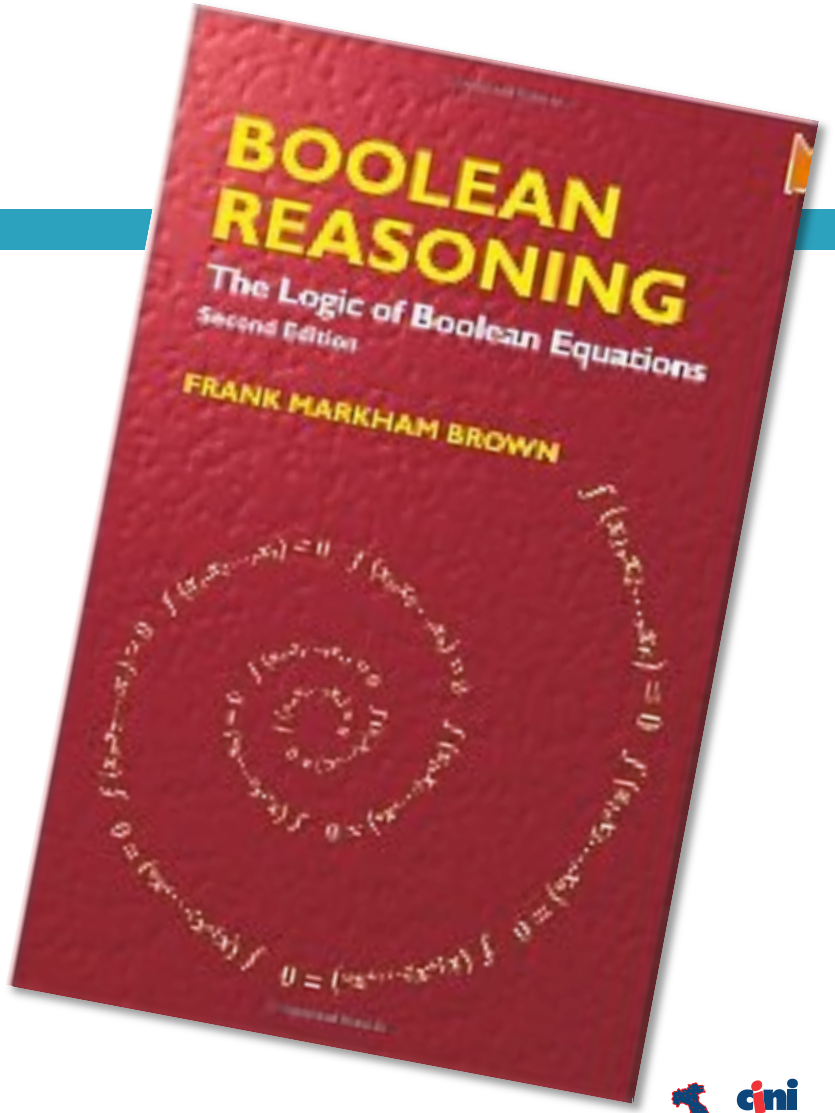
- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Prerequisites

- Students are assumed to be familiar with:
  - Lecture:
    - *HW_S_2 – Hardware Systems Representations*
  - The basic concepts, postulates, and theorems of Boolean Algebras.

Further readings

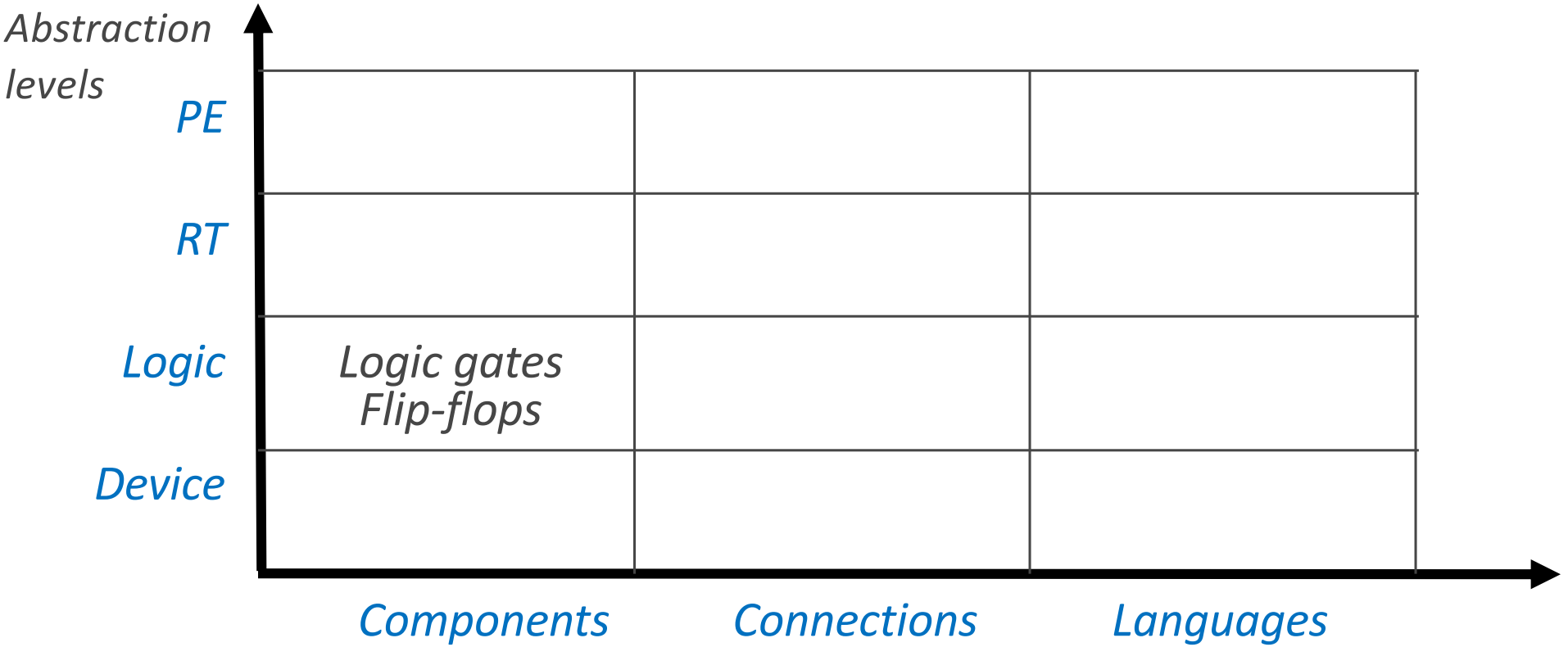
- Students interested in a deeper knowledge of the arguments covered in this lecture can refer, for instance, to:



Goal

- This lecture presents the set of devices usually considered as “elementary” at the Structural domain of the Logic level abstraction, i.e., *Logic Gates* and *Flip-Flops.*

Structural domain – Logic level



Outline

- Combinational devices:
  - Basic Boolean Functions & Logic Gates
- Sequential devices:
  - Flip-Flops
  - Scannable Flip-Flops

Outline

- Combinational devices:
  - Basic Boolean Functions & Logic Gates
- Sequential devices:
  - Flip-Flops
  - Scannable Flip-Flops

Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

Why are they so significant?

- Electronic devices (often referred to as *Logic Gates*) are available to implement each of these basic boolean functions
- Each device is usually given the same name of the corresponding boolean functions, e.g.:

AND function ⇔ AND gate

Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.

- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“Logic Complement” or “NOT” function

Informal definition

- The NOT function is a unary function, providing the logic complement of its input variable.

Representations

- x’
- $\bar{x}$
- not(x)

“Logic Complement” or “NOT” function

Axiomatic definition

0’ = 1

1’ = 0

Truth table

not (0) = 1

not (1) = 0

x	not(x)
0	1
1	0

“Logic Complement” or “NOT” function

Functional definition

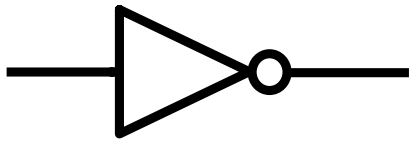
```
if x = 1
    then not(x)= 0
    else not(x)= 1
endif
```

Theorems

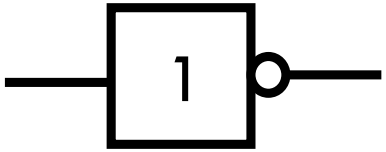
- (x’)’ = x
- not(not(x)) = x

NOT gate, or inverter

traditional symbol



IEEE symbol



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.

- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“Logic product” or “AND” function

Informal definition

- The AND function on 2 (or more) input variables provides the value 1 iff *all* its input variables get the value 1.

Representations

- $x \cdot y$
- x y
- and (x, y)

“Logic product” or “AND” function

Axiomatic definition

$0 \cdot 0 = 0$ and(0,0) = 0

$0 \cdot 1 = 0$ and(0,1) = 0

$1 \cdot 0 = 0$ and(1,0) = 0

$1 \cdot 1 = 1$ and(1,1) = 1

Truth table

x y	and(x,y)
0 0	0
0 1	0
1 0	0
1 1	1

“Logic product” or “AND” function

Functional definition

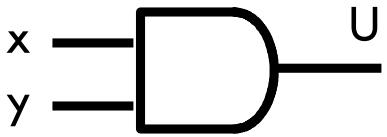
```
if x = 1
   then and(x,y) = y
   else and(x,y) = 0
endif
```

Theorems

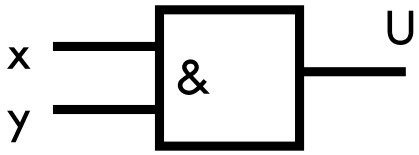
- $x \cdot 0 = 0$
- $x \cdot 1 = x$
- $x \cdot x = x$
- $x \cdot x' = 0$
- $x \cdot y = y \cdot x$
- $x \cdot y \cdot z = (x \cdot y) \cdot z = x \cdot (y \cdot z)$

“Logic product” or “AND” function

traditional symbol



IEEE symbol



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“Logic sum” or “OR” function

Informal definition

- The OR function on 2 (or more) input variables provides the value 1 iff *at least one* of its input variables gets the value 1.

Representations

- x + y
- or(x,y)

8. A logician's wife is having a baby. The doctor immediately hands the newborn to the dad.

His wife asks impatiently: "So, is it a boy or a girl" ?

The logician replies: "yes".

“Logic sum” or “OR” function

Axiomatic definition

0 + 0 = 0 or(0,0) = 0

0 + 1 = 1 or(0,1) = 1

1 + 0 = 1 or(1,0) = 1

1 + 1 = 1 or(1,1) = 1

Truth table

x	y	or(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

“Logic sum” or “OR” function

Functional definition

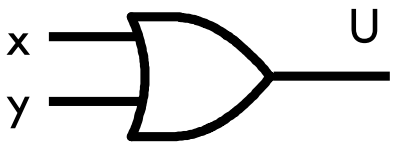
```
if x = 1
   then or(x,y)= 1
   else or(x,y)= y
endif
```

Theorems

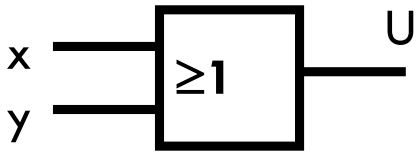
- $x + 0 = x$
- $x + 1 = 1$
- $x + x = x$
- $x + x' = 1$
- $x + y = y + x$
- $x + y + z = (x + y) + z = x + (y + z)$

“Logic sum” or “OR” function

traditional symbol



IEEE symbol



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“NAND” function

Informal definition

- The NAND function on 2 (or more) input variables provides the value 1 iff *at least one* of its input variables get the value 0.

Representations

- nand (x, y)

“NAND” function

Axiomatic definition

nand(0,0) = 1

nand(0,1) = 1

nand(1,0) = 1

nand(1,1) = 0

Truth table

x	y	nand(x,y)
0	0	1
0	1	1
1	0	1
1	1	0

“NAND” function

Functional definition

```
if x = 1
 then nand(x,y)=not(y)
 else nand(x,y) = 1
endif
```

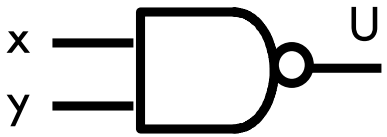
Theorems

- nand (x, 0) = 1
- nand (x, 1) = x’
- nand (x, x) = x’
- nand (x, x’) = 1
- nand (x, y) = nand (y, x)
- nand (x, y) = not (and (x, y))
- nand (x, y, z) = nand (x, and (y, z)) =

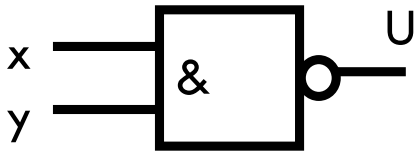
  = nand (and (x, y), z) = nand (x, and (y,z))

“NAND” function

traditional symbol



IEEE symbol



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“NOR” function

Informal definition

- The NOR function on 2 (or more) input variables provides the value 1 iff *none* of its input variables get the value 0.

Representations

- nor (x, y)

“NOR” function

Axiomatic definition

nor(0,0) = 1

nor(0,1) = 0

nor(1,0) = 0

nor(1,1) = 1

Truth table

x	y	nor(x,y)
0	0	1
0	1	0
1	0	0
1	1	0

“NOR” function

Functional definition

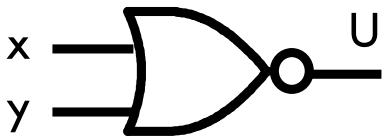
```
if x = 1
 then nor(x,y)=0
 else nor(x,y)=not(y)
endif
```

Theorems

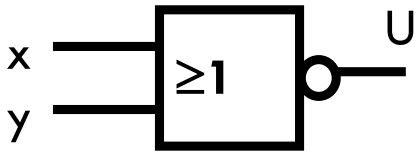
- nor (x, 0) = x’
- nor (x, 1) = 0
- nor (x, x) = x’
- nor (x, x’) = 0
- nor (x, y) = nor (y, x)
- nor (x, y) = not (or (x, y))
- nor (x, y, z) = nor (x, or (y, z)) =
  = nor (or (x, y), z) = nor (x, or (y,z))

“NOR” function

traditional symbol



IEEE symbol



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“EXOR” function

Informal definition

- The EXOR function on 2 (or more) input variables provides the value 1 iff *an odd #* of its input variables get the value 1.

Representations

- $x \oplus y$
- exor(x,y)
- xor(x,y)

“EXOR” function

Axiomatic definition

$0 \oplus 0 = 0$

$0 \oplus 1 = 1$

$1 \oplus 0 = 1$

$1 \oplus 1 = 0$

Truth table

x	y	exor(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

“EXOR” function

Functional definition #1

```
if x = 1
 then exor(x,y)= not(y)
 else exor(x,y)= y
endif
```

Remark

The exor function can be seen as a *controlled inverter* :

x	y	exor(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

“EXOR” function

Functional definition #2

```
if x = y
    then exor(x,y)=0
    else exor(x,y)=1
endif
```

Remark

The exor function can be seen as a *comparator*:

x	y	exor(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

“EXOR” function

Functional definition #3

```
if (x+y)mod 2)=0
    then exor(x,y)=0
    else exor(x,y)=1
endif
```

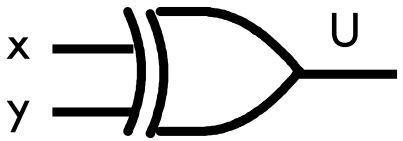
Remark

The exor function can be seen as a *modulo 2 adder:*

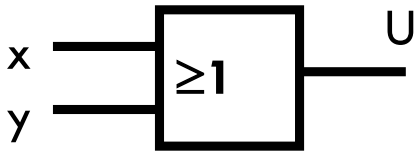
x	y	exor(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

“EXOR” function

traditional symbol



IEEE symbol



“EXOR” function

Theorems

- exor (x, 0) = x
- exor (x, 1) = x’
- exor (x, x) = 0
- exor (x, x’) = 1
- exor (x, y) = exor (y, x)
- exor (x, y) = exor (x’ , y’)
- exor (x, y’) = exor (x’, y) = (exor(x,y))’
- exor (x, y) = x y’ + x’ y = (x y)’ (x + y)

Composition

exor (x, y, z) = exor (x, exor (y, z)) =

= exor (exor (x,y), z)

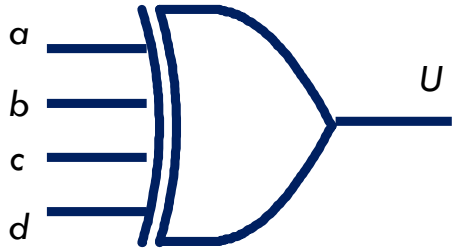
“EXOR” function

Composition

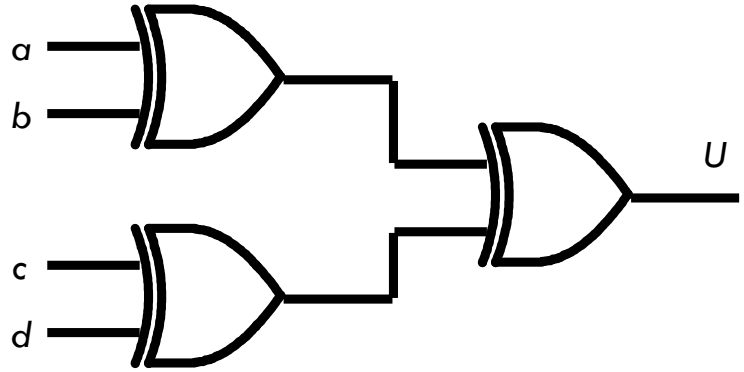
When using 2-input exor gates to implement N-input exor functions, 2 solutions are possible

“EXOR” function

Example



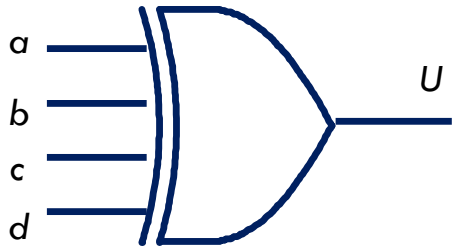
1st solution



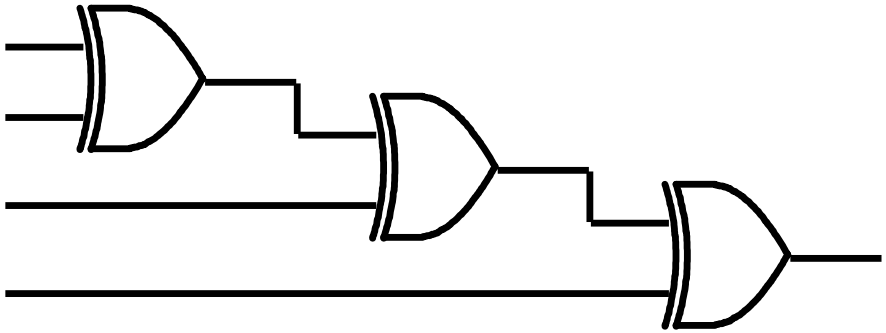
- Delay: O (log n)
- Cost: O (n)

“EXOR” function

Example

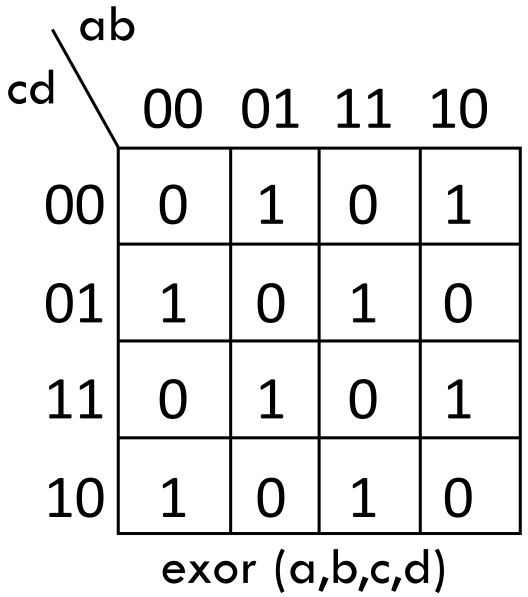


2nd solution



- Delay: O (n)
- Cost: O (n)

Karnaugh map for a 4-inputs exor



Basic Boolean Functions

- Several “Basic” Boolean Functions, often referred to as *Boolean Primitives*, have been defined, each identified by a unique name, become a world-wide de-facto standard.
- They include:
  - not
  - and
  - or
  - nand
  - nor
  - exor
  - exnor

“EXNOR” function

Informal definition

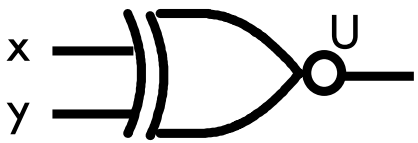
- The EXOR function on 2 (or more) input variables provides the value 1 iff *an even #* of its input variables get the value 1.

Representations

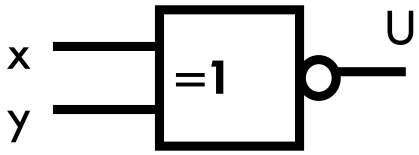
- $x \odot y$
- exnor(x,y)

“EXNOR” function

traditional symbol



IEEE symbol



“EXNOR” function

Remark

exnor(x,y) = (exor(x,y))’

Complete sets of logic primitives

Question

- Is it possible to represent any Boolean Function resorting to a whatever subset of the basic functions seen so far?

Answer

- No: you have to carefully select the subset!

Examples of Complete sets

- and, or, not
- and, not
- or, not
- nand
- nor
- …

Outline

- Combinational devices:
  - Basic Boolean Functions & Logic Gates
- Sequential devices:
  - Flip-Flops
  - Scannable Flip-Flops

Flip-Flop

- A flip-flop is a single-bit storage device implemented as a Moore machine.

Moore vs Mealy machines

Moore machine

- An FSM whose output values are determined only by its current state.

Mealy machine

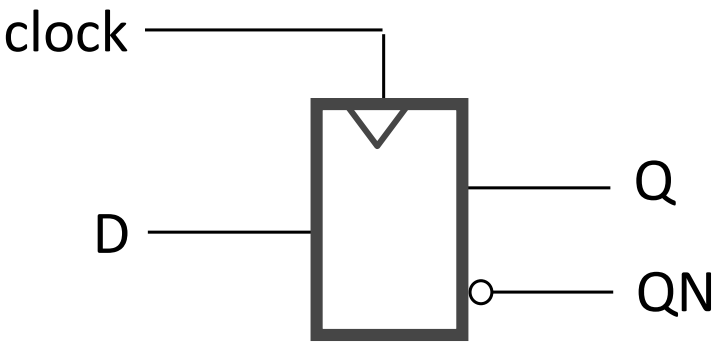
- An FSM whose (Mealy) output values are determined both by its current state and by the values of its inputs.

FF classification w.r.t. data inputs

- According to their behavior w.r.t. data inputs, flip-flops are usually classified as:
  - D
  - SR
  - JK
  - T

D-type Flip-Flop

- Is the most widely used FF



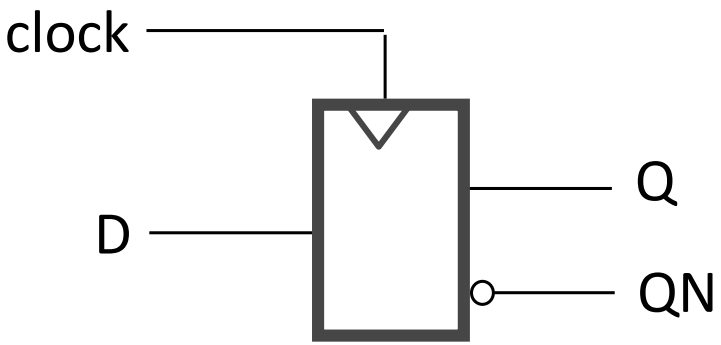
D	Q
0	0
1	1

Caveat

- Distinction between data inputs and control inputs is context and application dependent.

Example: (D = data input)

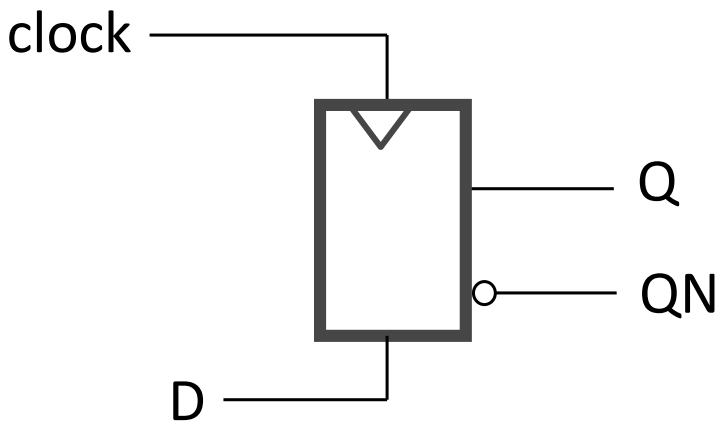
- At any clock, it stores the input data



Example: (D = control input)

- At any clock:

```
if (D = 0)
   then force Q=0
   else force Q=1
end if
```



Additional FF types

- In addition to the D-type FF, other ones have been proposed (namely: *JK*, *SR*, and *T*) but they are practically no longer used, due to testability reasons.

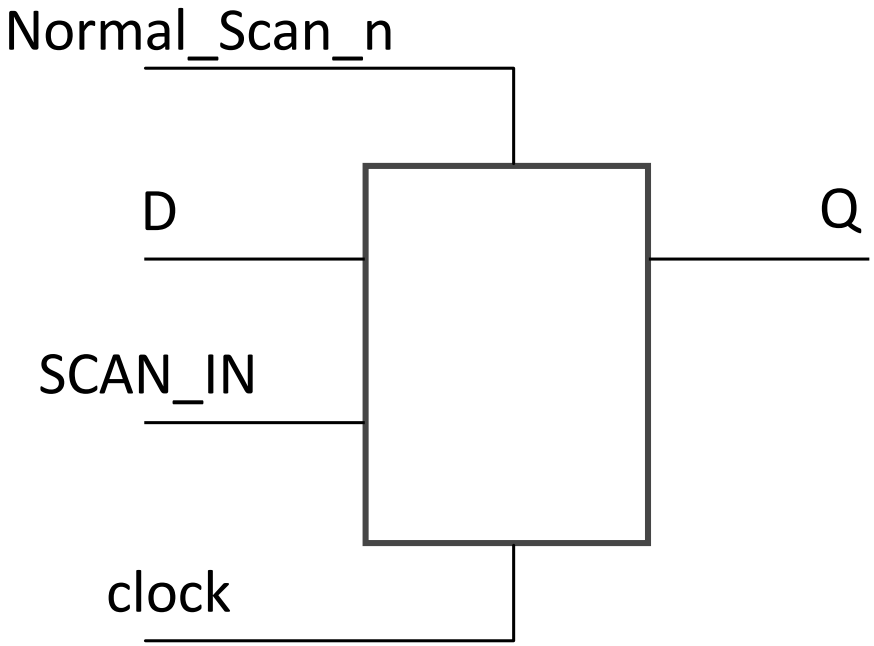
Outline

- Combinational devices:
  - Basic Boolean Functions & Logic Gates
- Sequential devices:
  - Flip-Flops
  - Scan Flip-Flops

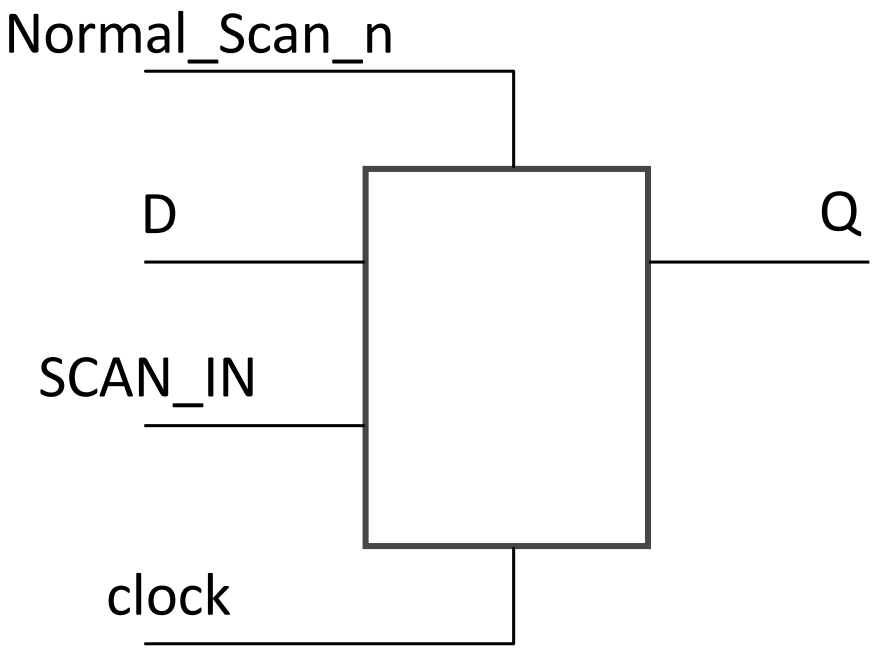
Scan Flip-Flops

- In order to improve testability, most of today cell libraries include *Scan Flip-Flops*, i.e., modified versions of the D-type FF designed to make easier their inclusion into the so-called *Scan chains*.

Scan Flip-Flop

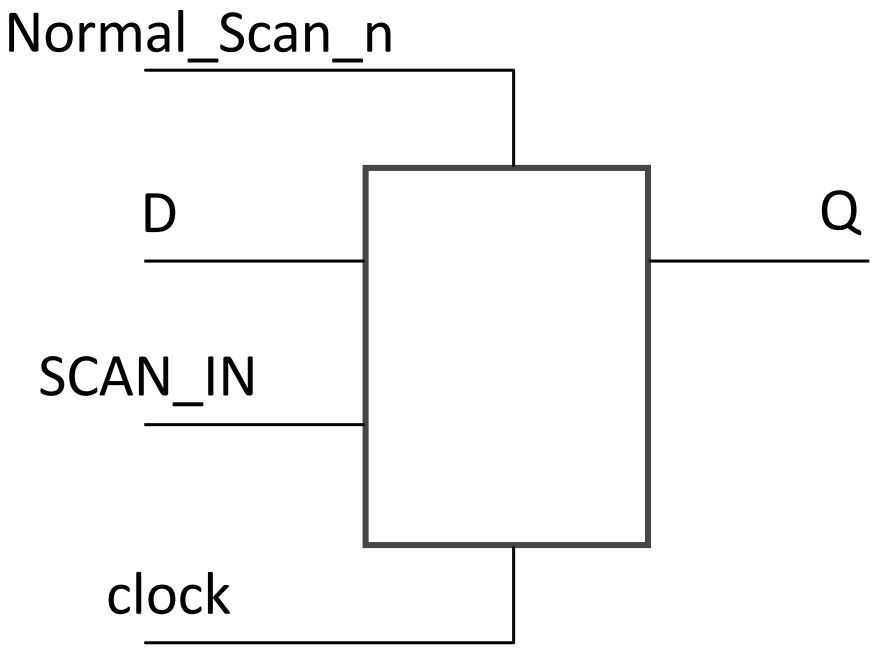


Scan Flip-Flop



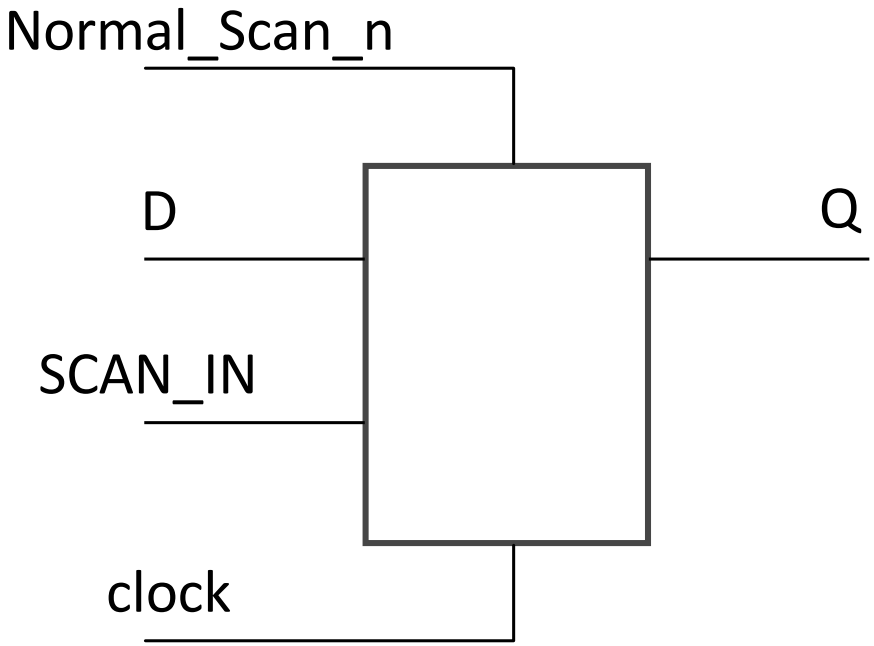

```
architecture scan_ff_arch of scan_ff is
  begin
    process(clk, rst)
      begin
        if rst='1' then
          Q <= '0';
        elsif rising_edge(clk) then
          if Normal_Scan_n = '1' then
            Q <= D;
          else
            Q <= SCAN_IN;
          end if;
          end if;
        end if;
    end process;
end scan_ff_arch;
```

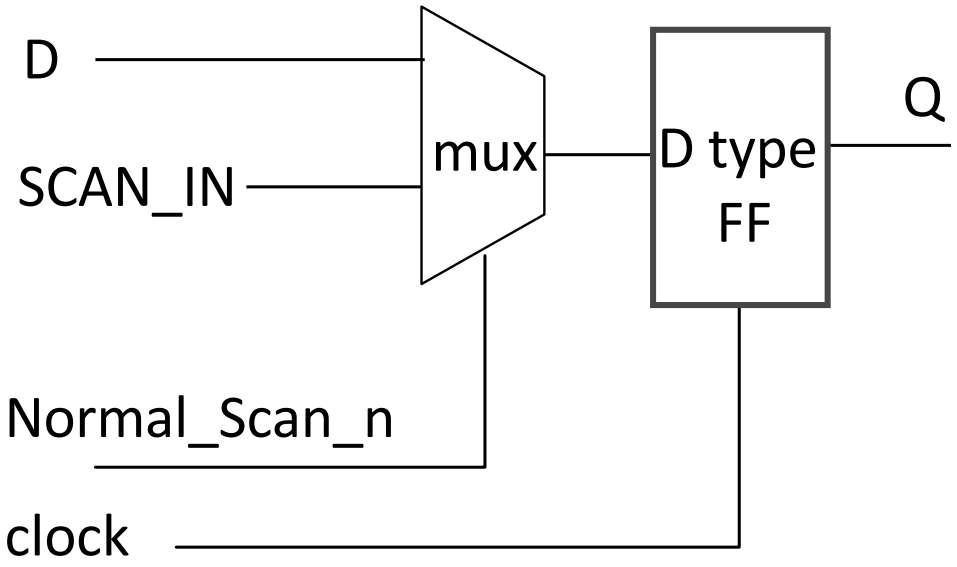
Scan Flip-Flop




```
architecture scan_ff_arch of scan_ff is
  begin
    process(clk, rst)
      begin
        if rst='1' then
          Q <= '0';
        elsif rising_edge(clk) then
          if Normal_Scan_n = '1' then
            Q <= D;
          else
            Q <= SCAN_IN;
          end if;
        end if;
    end process;
end scan_ff_arch;
```

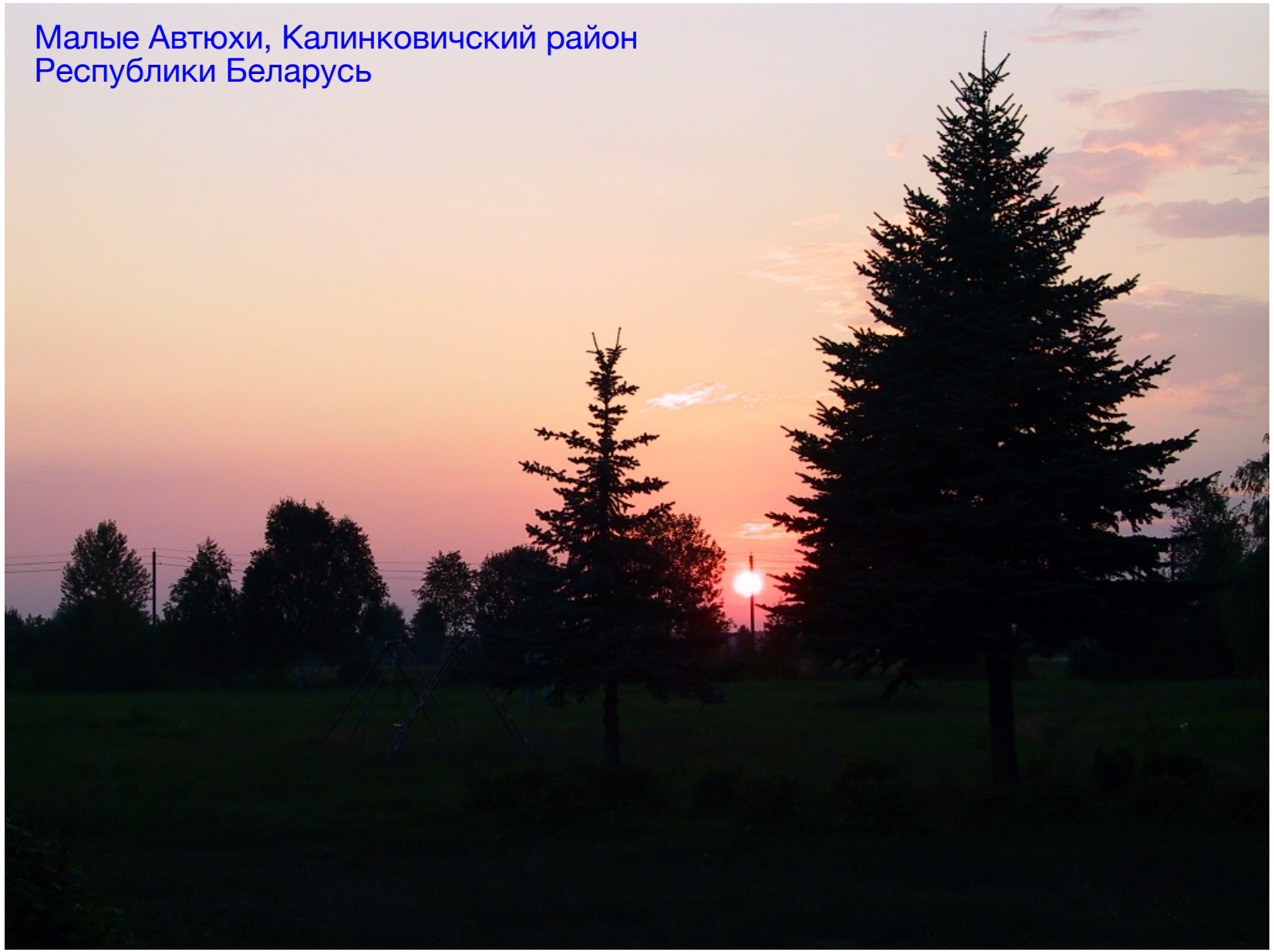
Scan Flip-Flop – Internal structure





Usage

- An analysis of the motivations of the adoption of Scan FFs and of their usage to improve testability is introduced in lecture:
  - *HT#1 – Hardware testing - Basic concepts*



Малые Автюхи, Калинковичский район
Республики Беларусь

Paolo PRINETTO

Director
CINI Cybersecurity
National Laboratory

Paolo.Prinetto@polito.it

Mob. +39 335 227529

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity
National Lab

https://cybersecnatlab.it