Electronics Systems (938II)

Lecture 2.1

Building Blocks of Electronic Systems – MUX, DEMUX, XOR, Comparator

Building Blocks of Electronic Systems

- An electronic system is made of combinational and/or sequential circuits
  - **Combinational** circuit
    - Output depends only on the current input(s)
  - **Sequential** circuit
    - Output depends on both the current input(s) and the current state
    - Current state = previous output → It has memory

Building Blocks of Electronic Systems

- Combinational circuits
  - The best known and most widely used are the NOT, AND and OR gates
  - Any logic function can be implemented using a combination of these gates

  - Or, NOT, NOR and NAND gates
    - Remember advantages of active-low logic
    - Rember De Morgan's laws
      - $A \cdot B = \overline{\bar{A} + \bar{B}}$
      - $A + B = \overline{\bar{A} \cdot \bar{B}}$

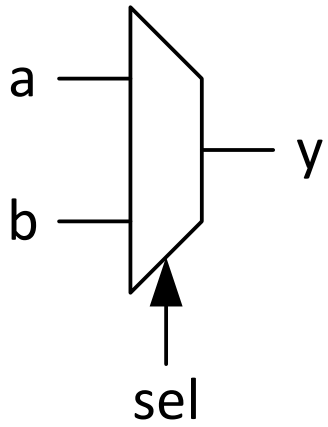
Not required

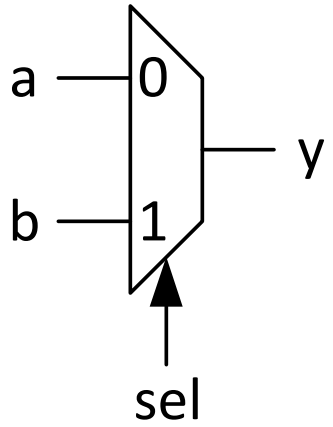
Building Blocks of Electronic Systems

- Combinational circuits
  - Other primitives exist
  - For example, the multiplexer (or MUX)

Multiplexer (or MUX)

- Logic symbol and functionality
  - 1 (input) control signal = sel
  - 1 output = y
  - N (data) inputs, e.g., N = 2: a, b





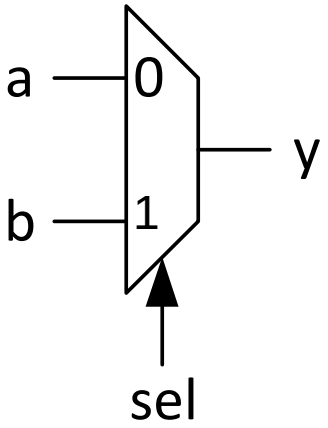
  - If sel = 0 → y = a, else (sel = 1) → y = b

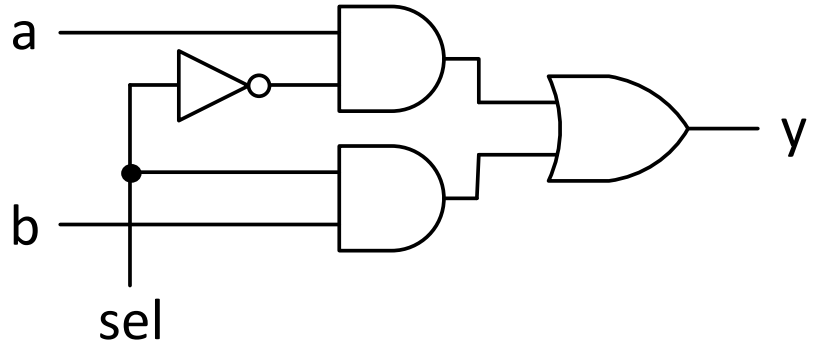
  - Bit width of sel must be $\geq ceil(\log_2(N))$
  - Output and (data) inputs can also be multi-bit

↳ bit width of input and output can differ

Multiplexer (or MUX)

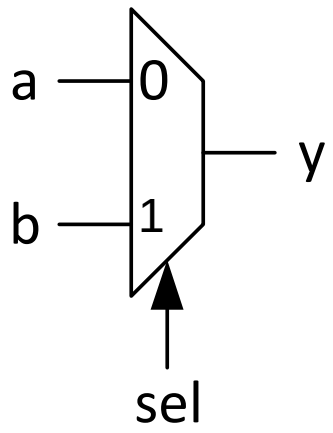
- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

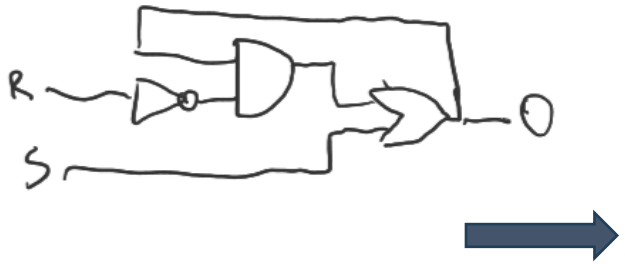


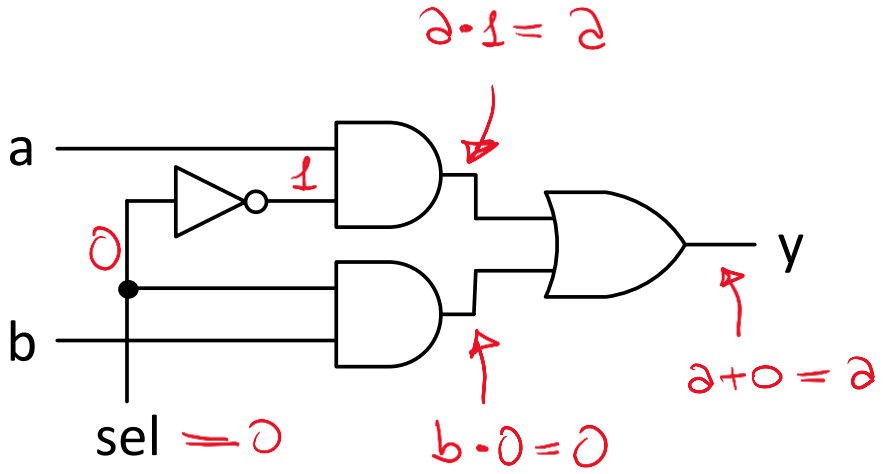


Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

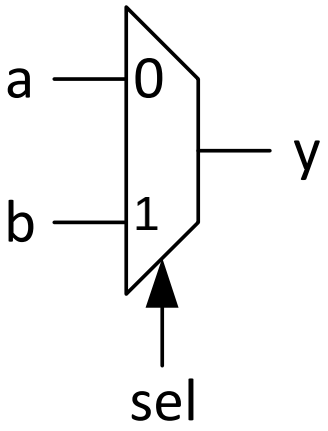


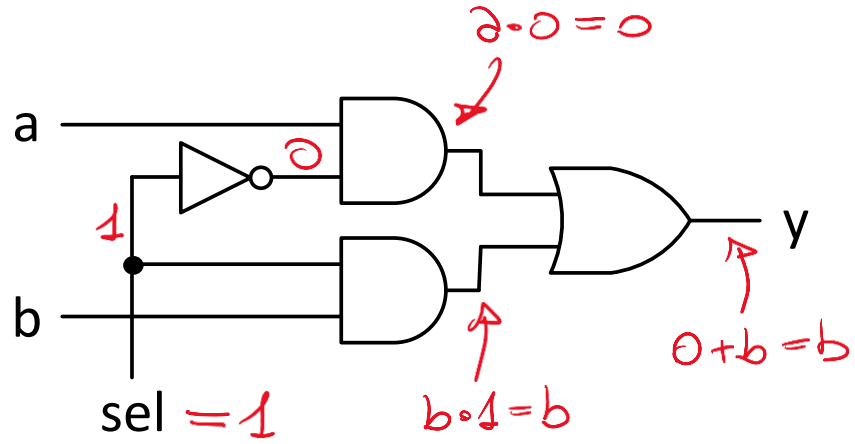




Multiplexer (or MUX)

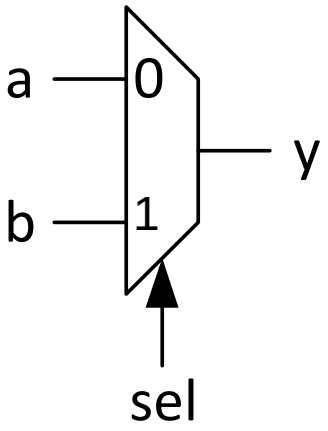
- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

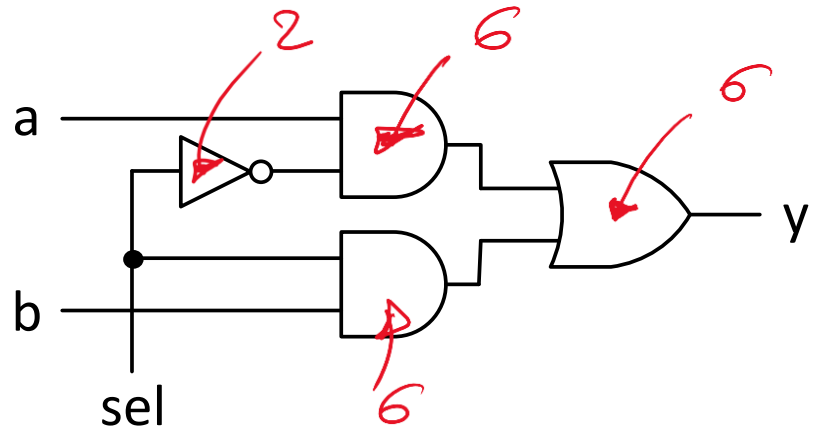




Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$





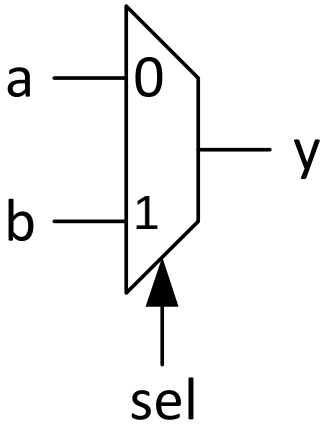
transistor count = 20!!!

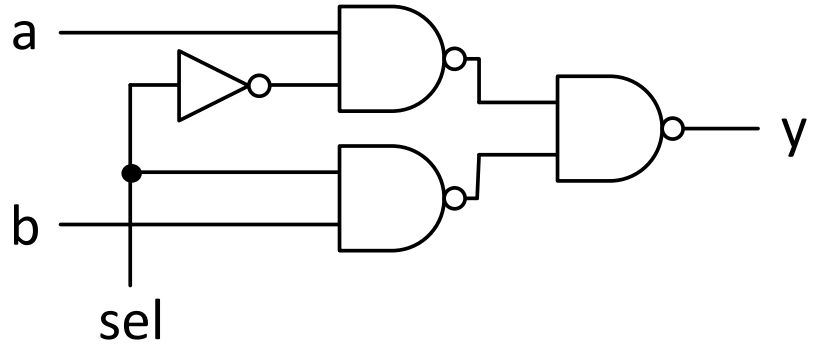
Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

equivalent!

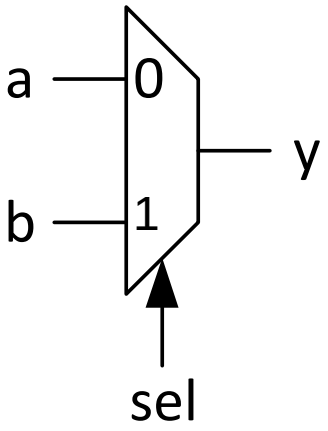
(NAND gate-based)



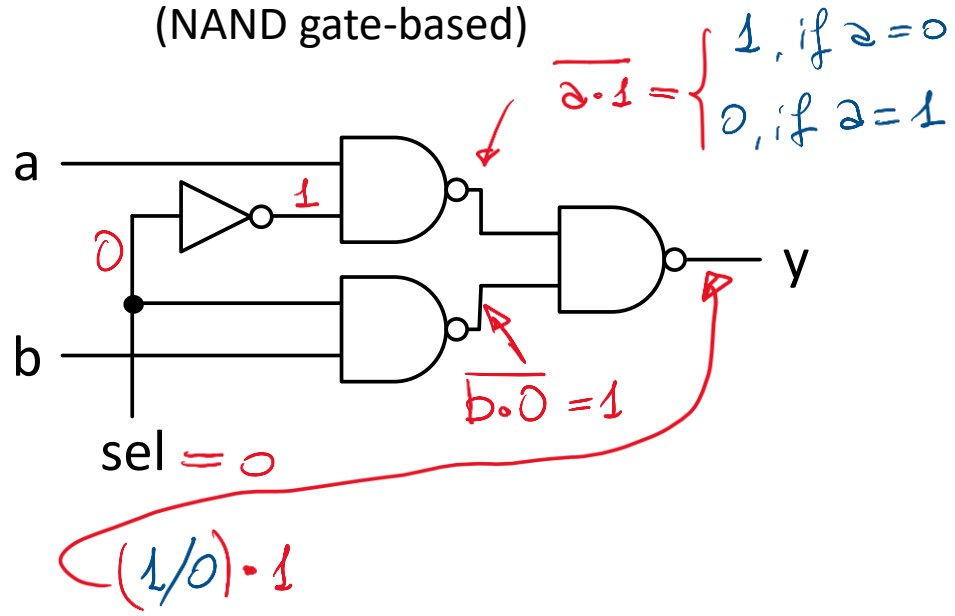


Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

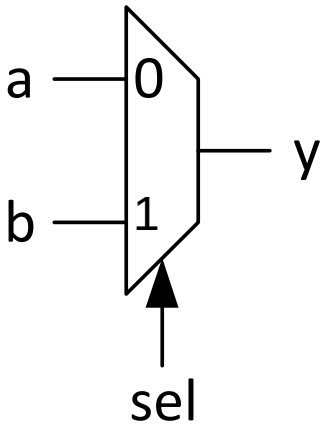


(NAND gate-based)

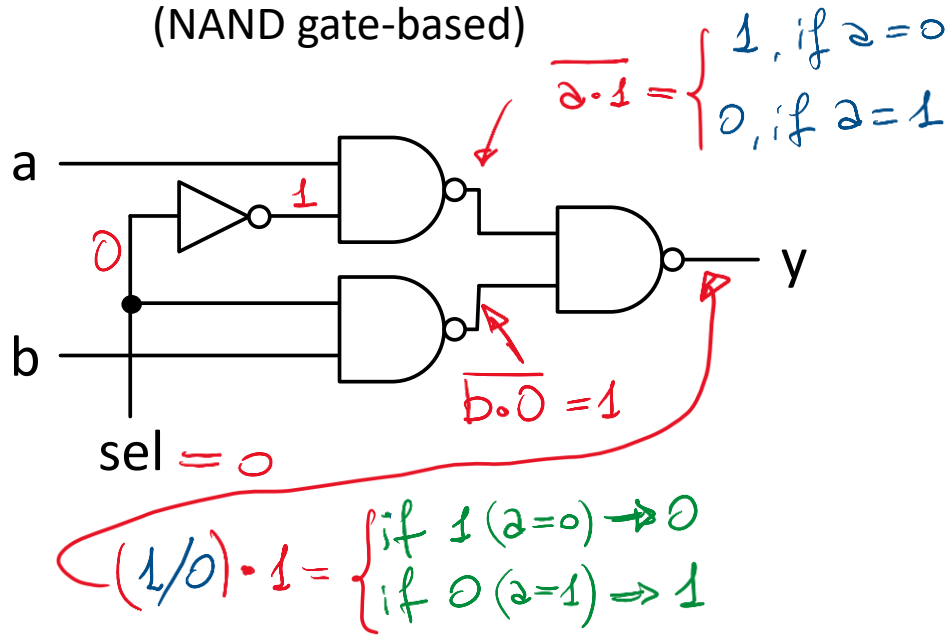


Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

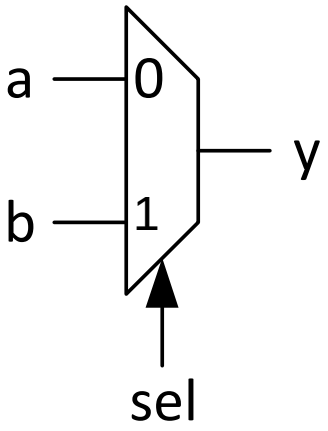


(NAND gate-based)

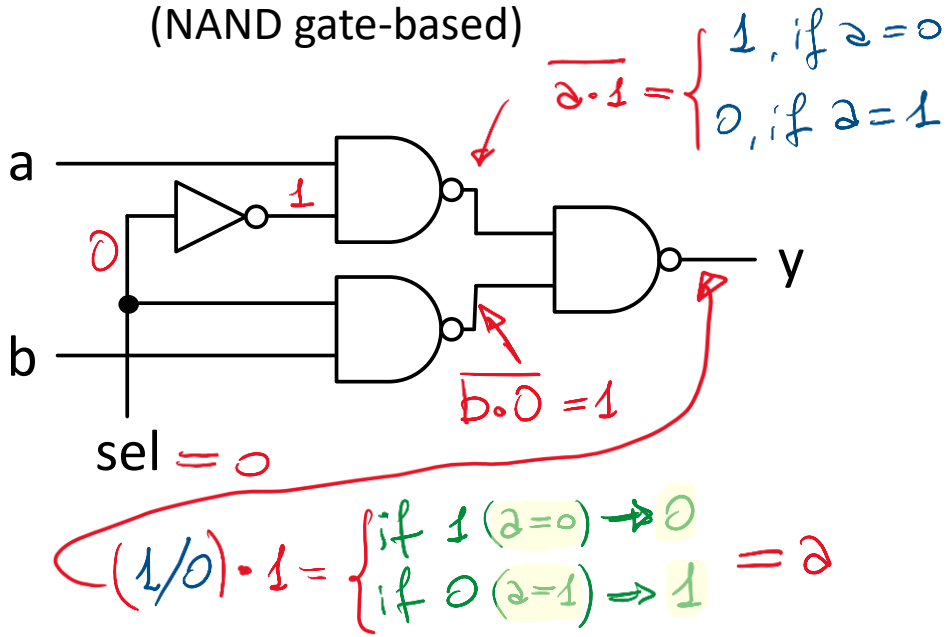


Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$

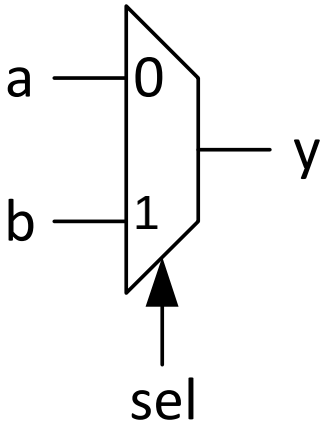


(NAND gate-based)

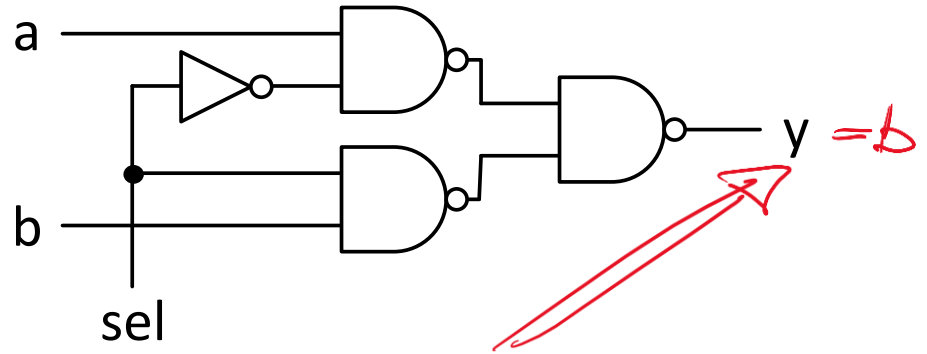


Multiplexer (or MUX)

- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b → $y = a \cdot \overline{sel} + b \cdot sel$



(NAND gate-based)

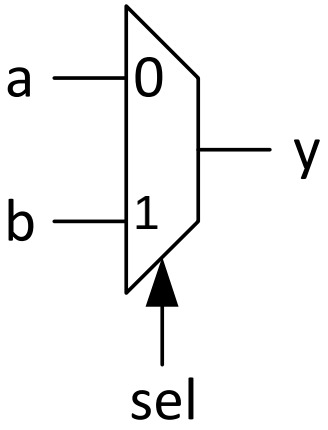


the opposite of before

Active low: when the signal is low the sg. is active. In practice this means using NAND or NOR instead of AND or OR or NOT

Multiplexer (or MUX)

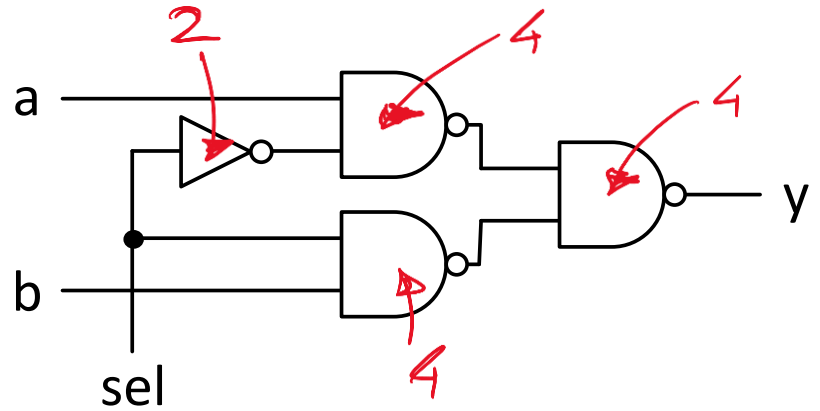
- Gate-level circuit
  - If sel = 0 → y = a, else (sel = 1) → y = b ⟹ $y = a \cdot \overline{sel} + b \cdot sel$



Better to use active low version

active low logic!

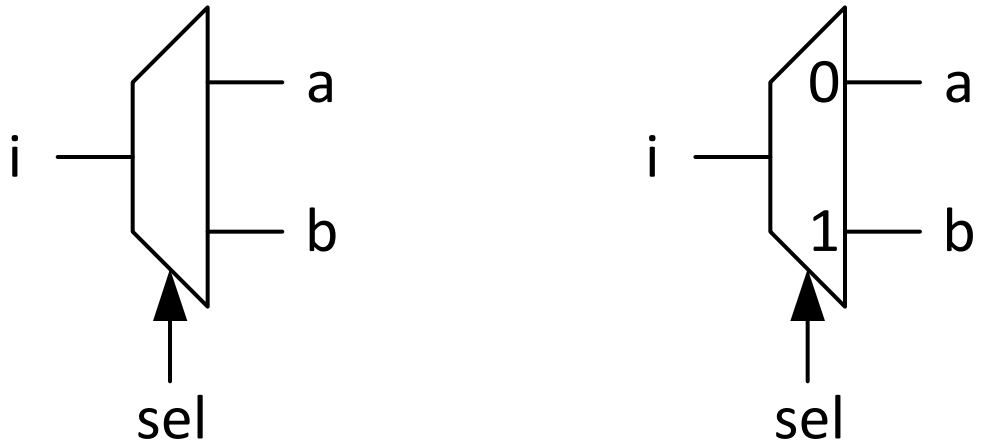
(NAND gate-based)



transistor count = 14: lower (before was 20)!

Demultiplexer (or DEMUX)

- Opposite of MUX
- Logic symbol and functionality
  - 1 (input) control signal = sel
  - N outputs, e.g., N = 2: a, b
  - 1 (data) input = i



  - If sel = 0 → a = i, else (sel = 1) → b = i

Bit width of the input is the same of the output

  - Bit width of sel must be $\geq ceil(\log_2(N))$
  - Outputs and (data) input can also be multi-bit

XOR (or exclusive-OR)

- Description
  - Two inputs
  - One output
  - Functionality
    - One input or the other one is equal to (logic) 1
    - **Exclusive OR** (one or the other)
  - Boolean symbol: ⊕

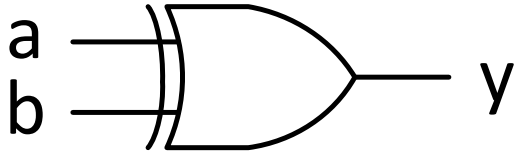
Truth table

a	b	$y = a \oplus b$
0	0	0
0	**1**	**1**
1	0	**1**
1	1	0

$\bar{a}b + a\bar{b}$

XOR (or exclusive-OR)

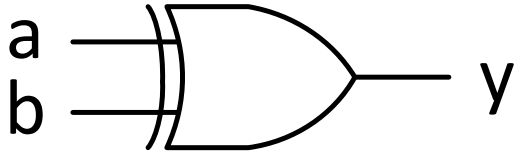
- Another single-gate logical function
  - Logic symbol = gate-level symbol



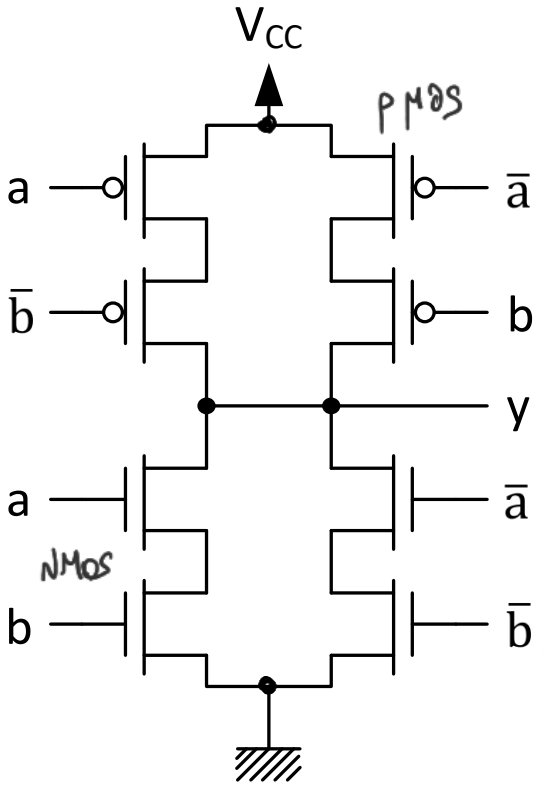
(gate-level = logic symbol)

XOR (or exclusive-OR)

- Another single-gate logical function
  - Logic symbol = gate-level symbol



(gate-level = logic symbol)



(transistor-level)

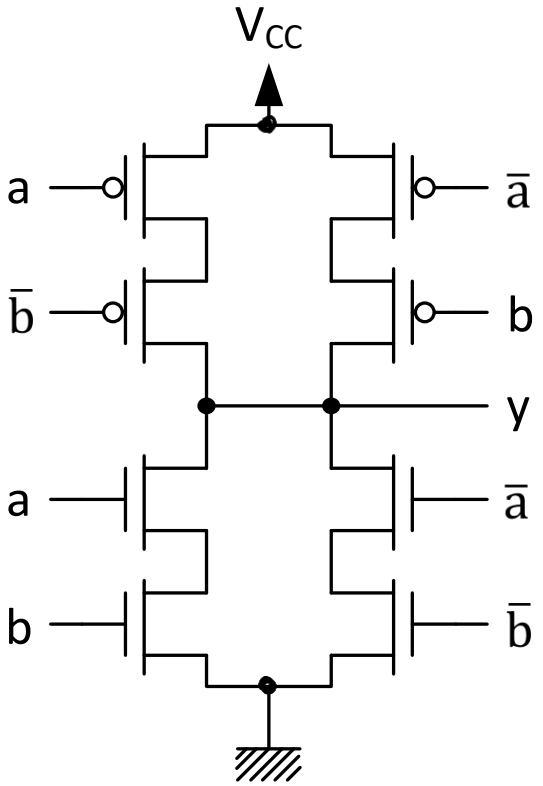
We need 4 more transistors for the NOT

XOR (or exclusive-OR)

- Let's verify it works

Truth table

a	b	$y = a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0



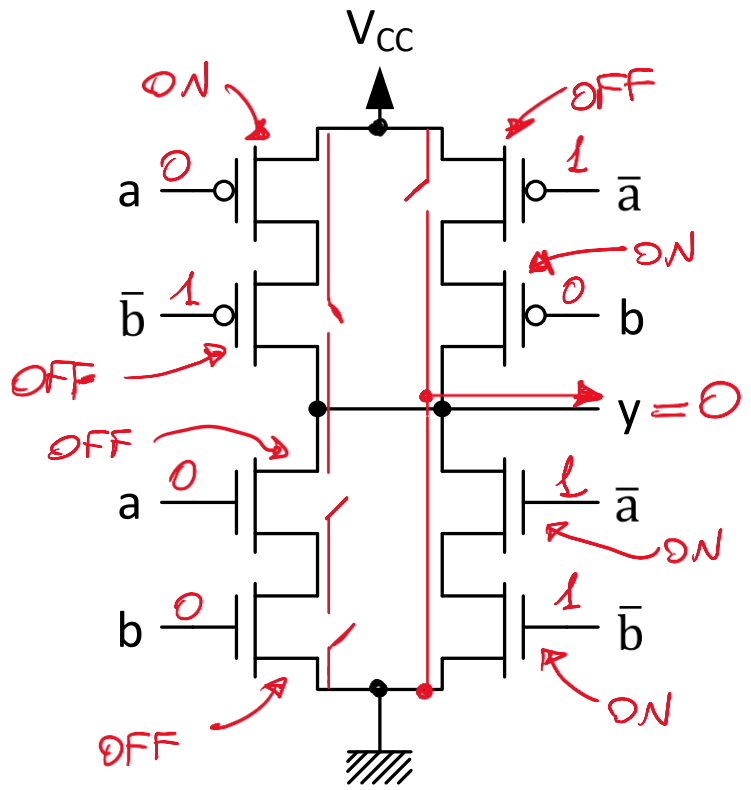
(transistor-level)

XOR (or exclusive-OR)

- Let's verify it works

Truth table

a	b	$y = a \oplus b$
0	**0**	**0**
0	1	1
1	0	1
1	1	0



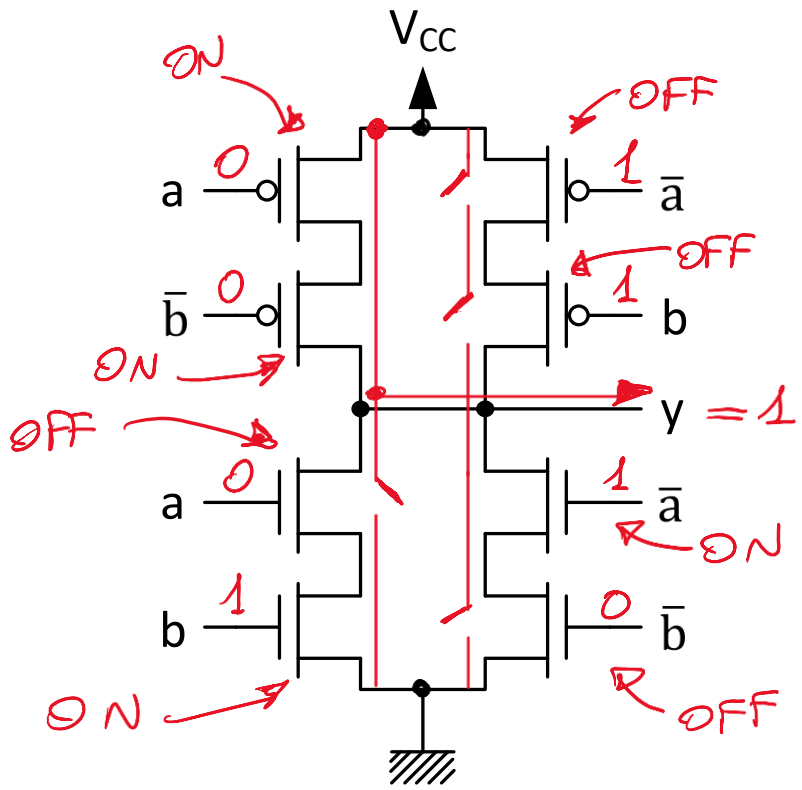
(transistor-level)

XOR (or exclusive-OR)

- Let's verify it works

Truth table

a	b	$y = a \oplus b$
0	0	0
0	**1**	**1**
1	0	1
1	1	0



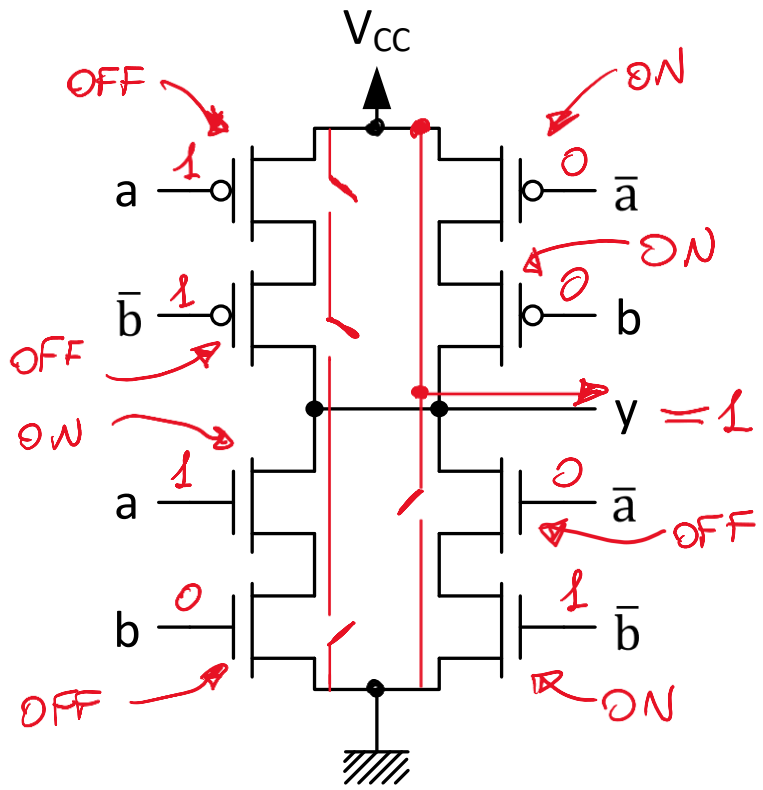
(transistor-level)

XOR (or exclusive-OR)

- Let's verify it works

Truth table

a	b	$y = a \oplus b$
0	0	0
0	1	1
1	**0**	**1**
1	1	0



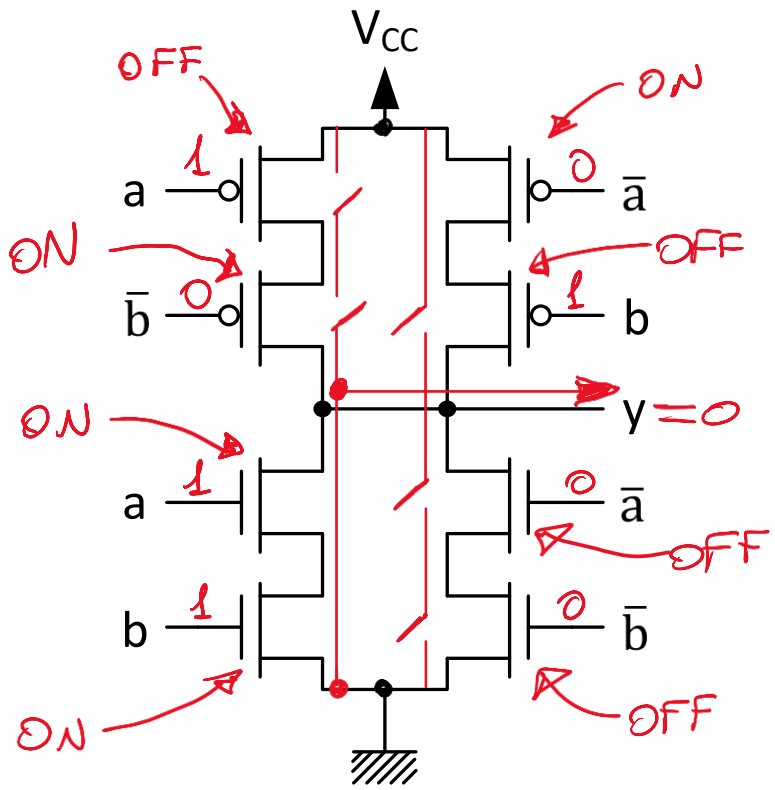
(transistor-level)

XOR (or exclusive-OR)

- Let's verify it works

Truth table

a	b	$y = a \oplus b$
0	0	0
0	1	1
1	0	1
1	**1**	**0**



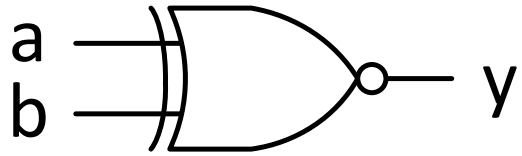
(transistor-level)

XNOR (or exclusive-NOR)

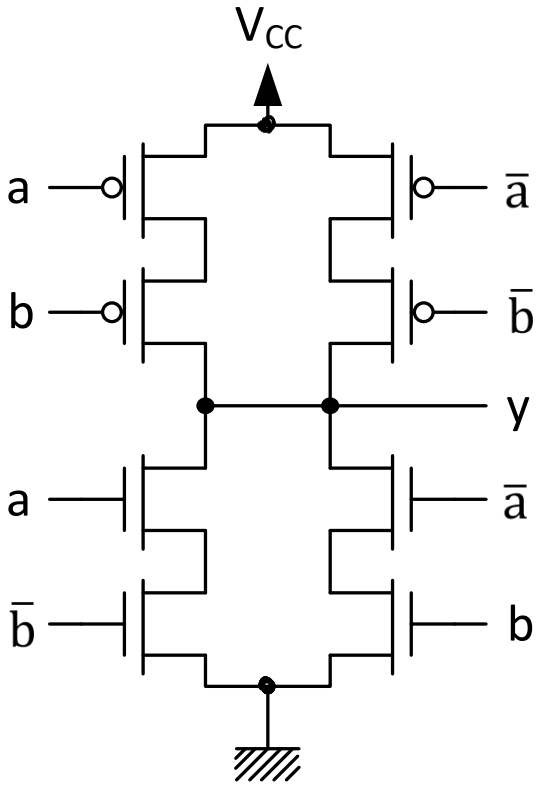
- Logical complement of XOR
  - Boolean symbol: $\overline{\oplus}$
  - Another single-gate logical function
    - Logic symbol = gate-level symbol

a	b	$y = a \overline{\oplus} b$
0	0	1
0	**1**	**0**
1	0	**0**
1	1	1

Truth table



(gate-level)



(transistor-level)

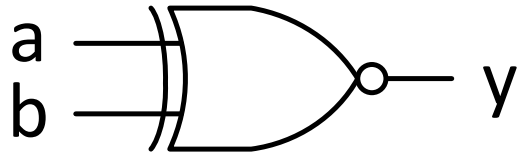
XNOR (or exclusive-NOR)

- Logical complement of XOR
  - Boolean symbol: $\overline{\oplus}$
  - Another single-gate logical function
    - Logic symbol = gate-level symbol

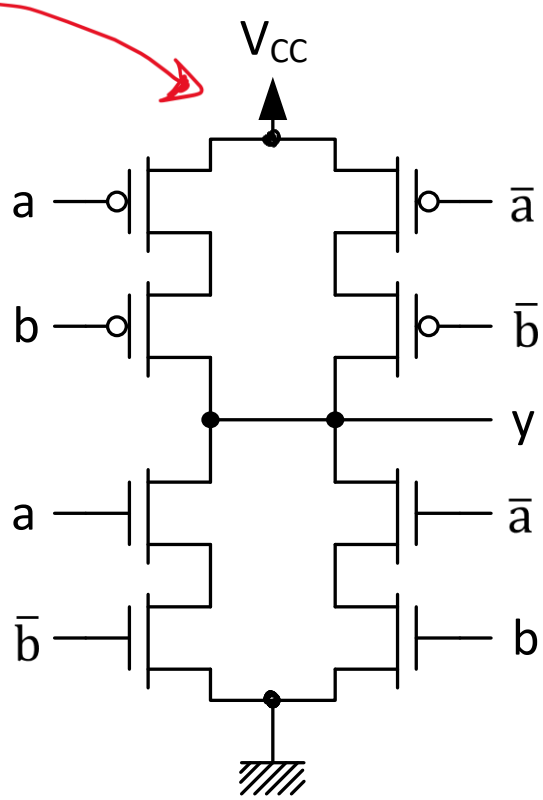
It works: trust me!
(or verify on your own)

a	b	$y = a \overline{\oplus} b$
0	0	1
0	**1**	**0**
1	0	**0**
1	1	1

Truth table



(gate-level)



(transistor-level)

Comparator

- Looking at the truth table of the XOR (or XNOR) function, essentially it indicates if inputs are different (or equal)

XOR truth table

a	b	$y = a \oplus b$
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

1 when **inputs** are **different**

XNOR truth table

a	b	$y = a \overline{\oplus} b$
0	**0**	**1**
0	1	0
1	0	0
1	**1**	**1**

1 when **inputs** are **equal**

Comparator

- Looking at the truth table of the XOR (or XNOR) function, essentially it indicates if inputs are different (or equal)

XOR truth table

a	b	$y = a \oplus b$
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

1 when **inputs** are **different**

XNOR truth table

a	b	$y = a \overline{\oplus} b$
0	**0**	**1**
0	1	0
1	0	0
1	**1**	**1**

1 when **inputs** are **equal**

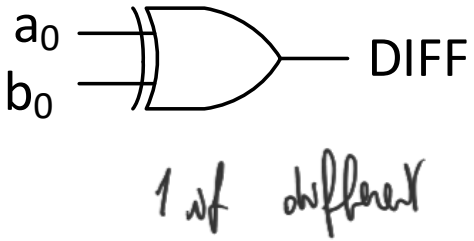
Can be used to build comparators

- check if two vectors (single-bit or multi-bit) are different (or equal)

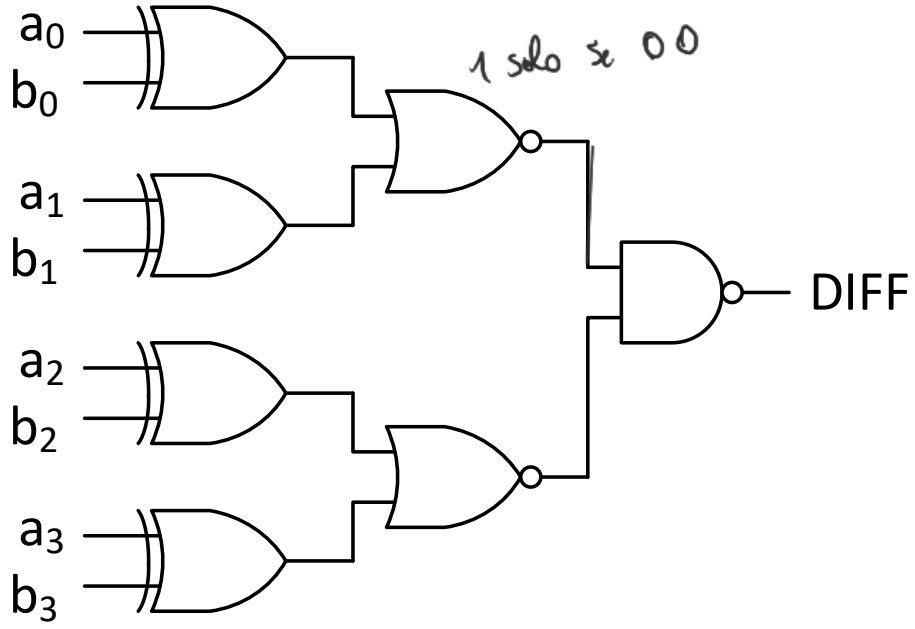
Comparator

- Parallel version: DIFF = 1, if inputs are different (otherwise 0)

1-bit comparator



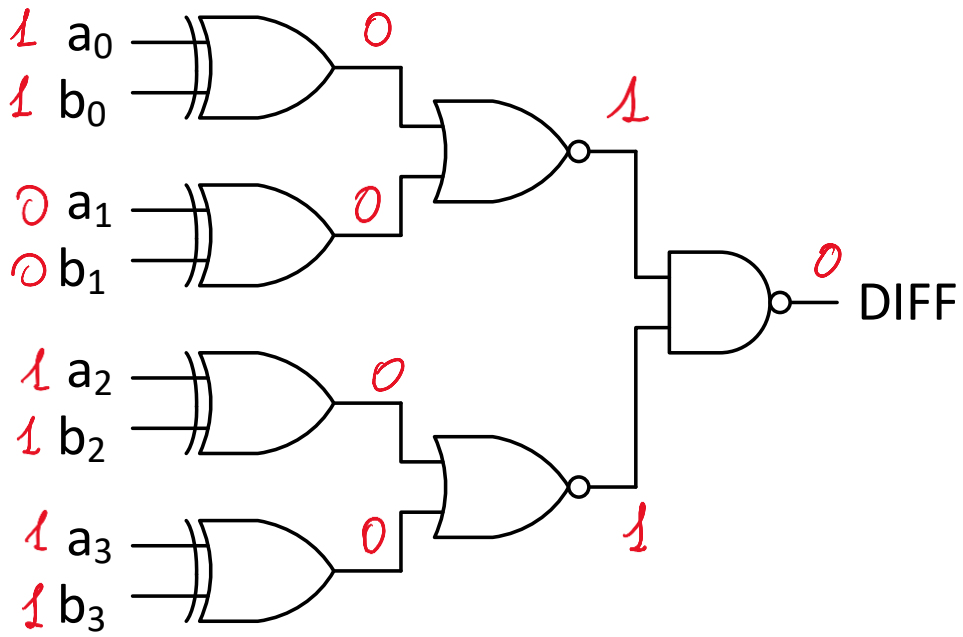
4-bit comparator



Comparator

- Parallel version: DIFF = 1, if inputs are different (otherwise 0)
  - Inputs
    - a = {a_3 a_2 a_1 a_0}
    - b = {b_3 b_2 b_1 b_0}
  - Example
    - a = {1101}
    - b = {1101}

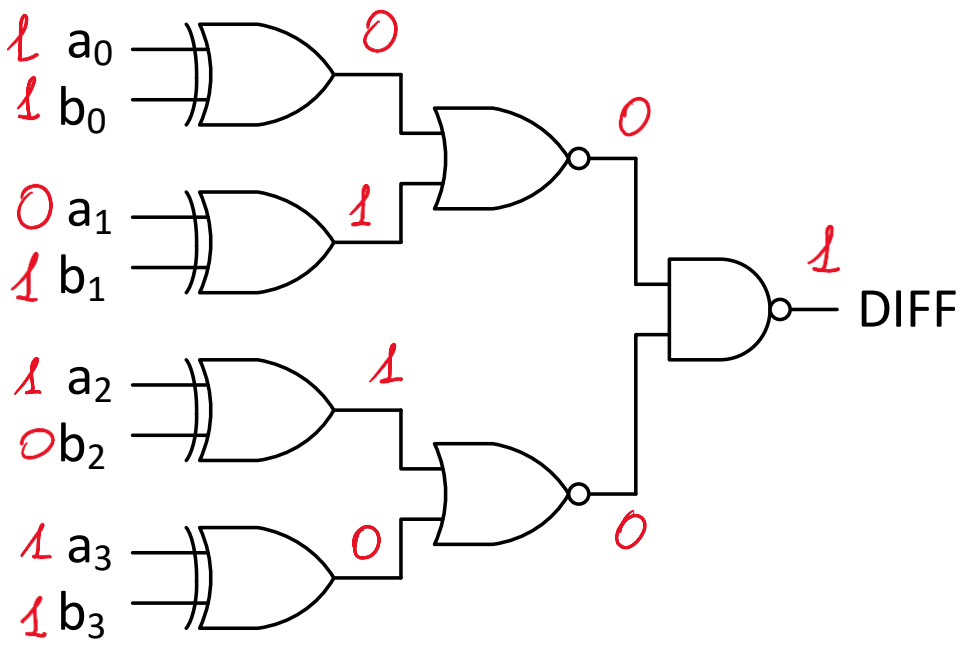
4-bit comparator



Comparator

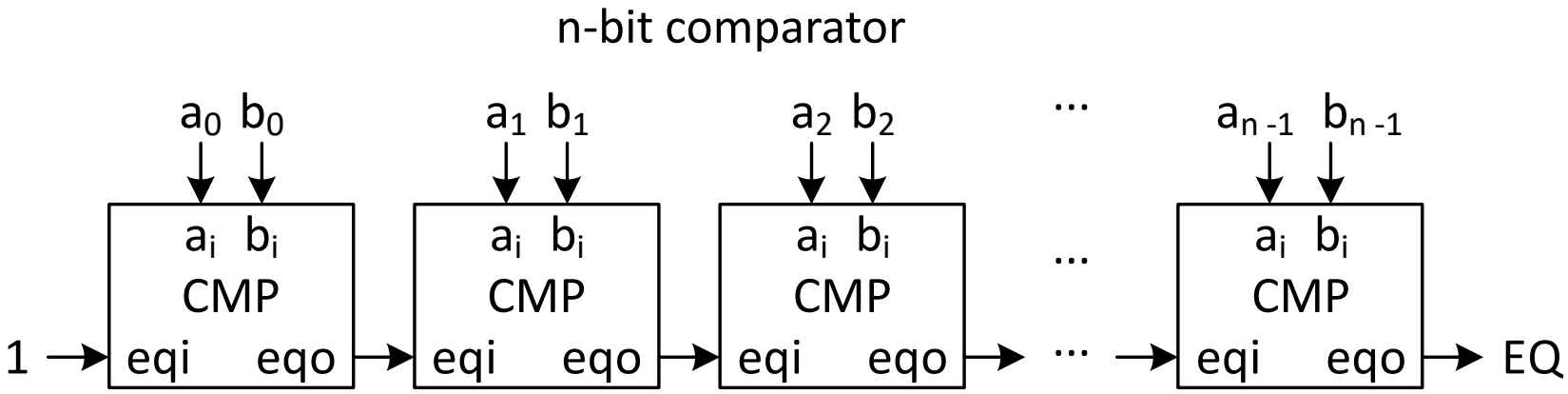
- Parallel version: DIFF = 1, if inputs are different (otherwise 0)
  - Inputs
    - $a = \{a_3\ a_2\ a_1\ a_0\}$
    - $b = \{b_3\ b_2\ b_1\ b_0\}$
  - Example
    - a = {1101}
    - b = {1011}

4-bit comparator



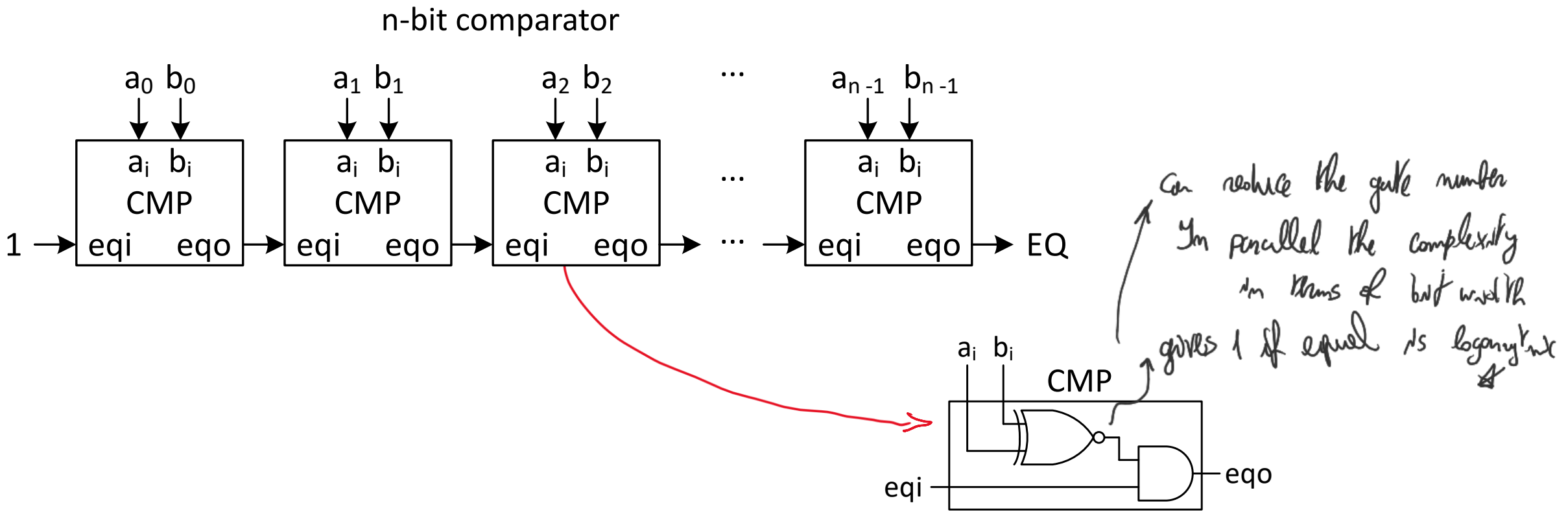
Comparator

- Iterative (or sequential) version: EQ = 1, if inputs are equal



Comparator

- Iterative (or sequential) version: EQ = 1, if inputs are equal

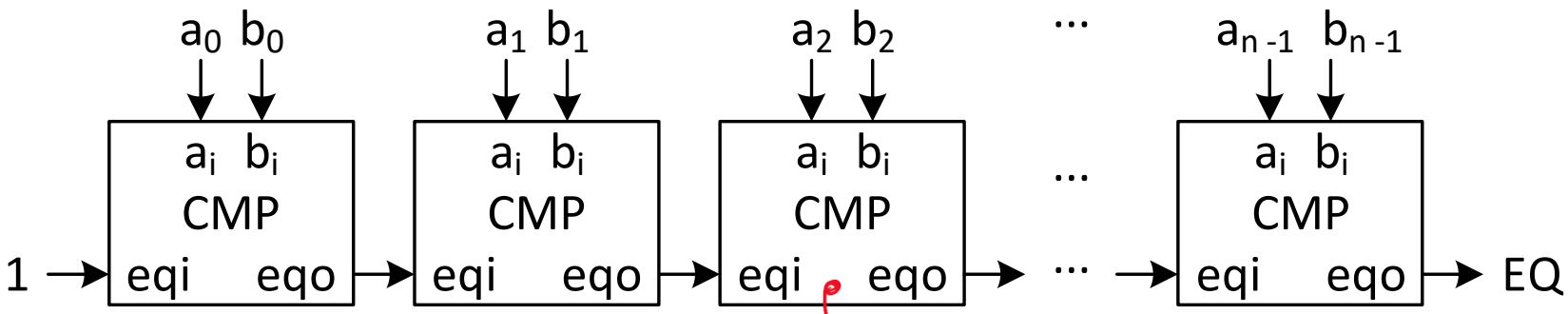


Number of levels is the $\log_2$ (number of inputs), while it is linear in series

Comparator

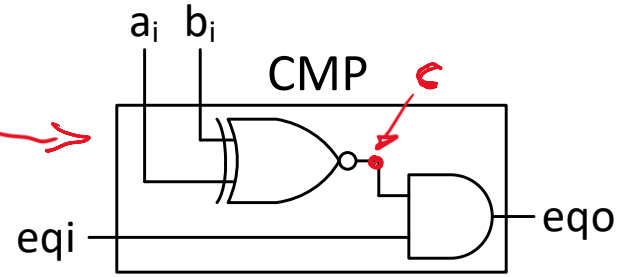
- Iterative (or sequential) version: EQ = 1, if inputs are equal

n-bit comparator



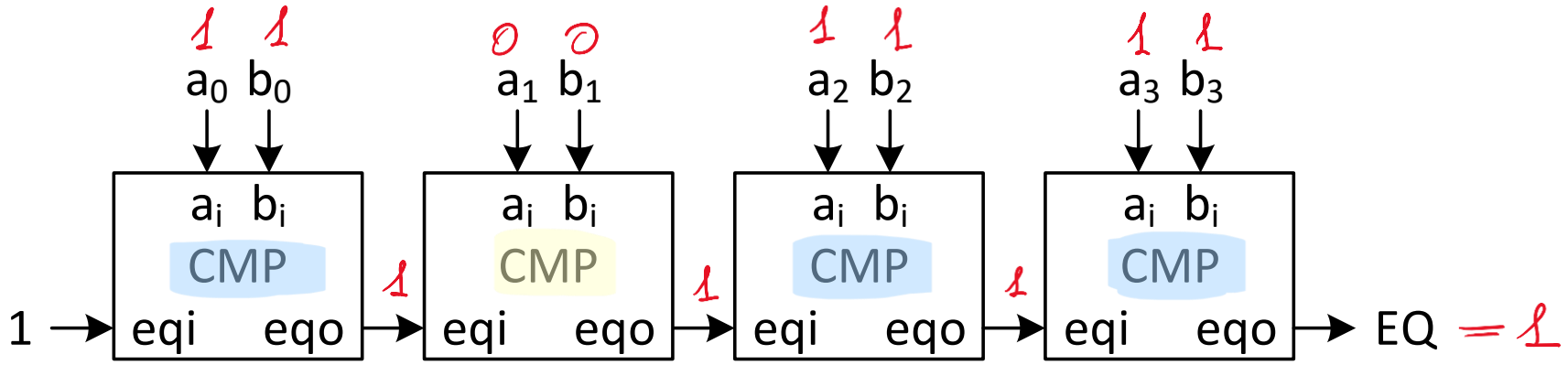
CMP truth table

eqi	a_i	b_i	c	eqo
0	x	x	x	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



Comparator

- Iterative (or sequential) version: EQ = 1, if inputs are equal
  - Example: 4-bit comparator
    - a = {1101}
    - b = {1101}

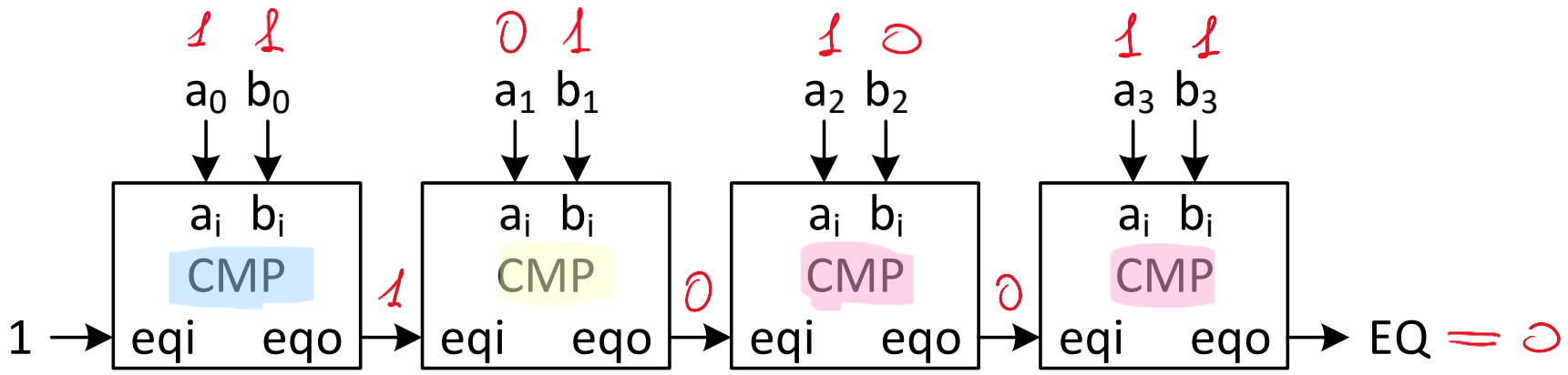


CMP truth table

eqi	a_i	b_i	c	eqo
0	x	x	x	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Comparator

- Iterative (or sequential) version: EQ = 1, if inputs are equal
  - Example: 4-bit comparator
    - a = {1101}
    - b = {1011}

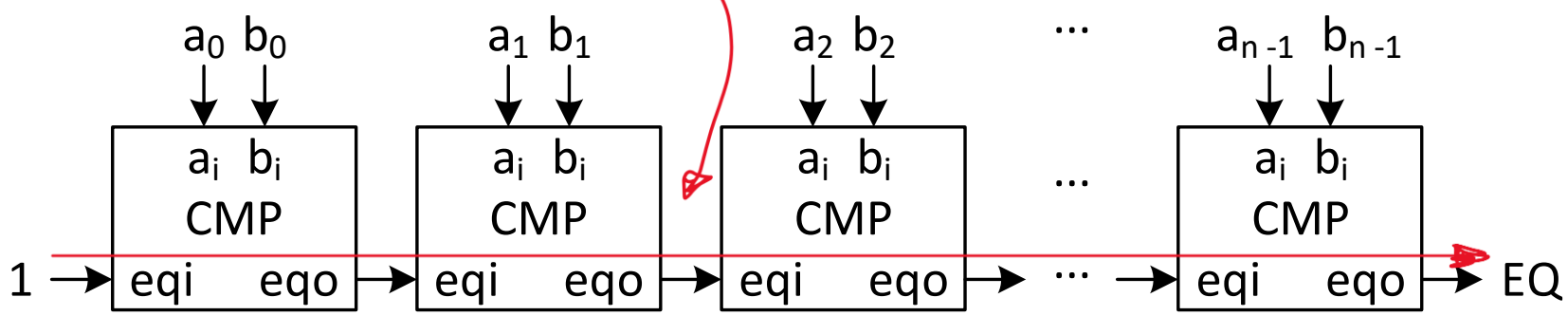


CMP truth table

eqi	a_i	b_i	c	eqo
0	x	x	x	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Comparator

- Iterative (or sequential) version: EQ = 1, if inputs are equal
  - For very long n-bit input vectors (i.e. n is very high), it may reduce resource consumption
  - But very long path/delay



Exercise with SystemVerilog

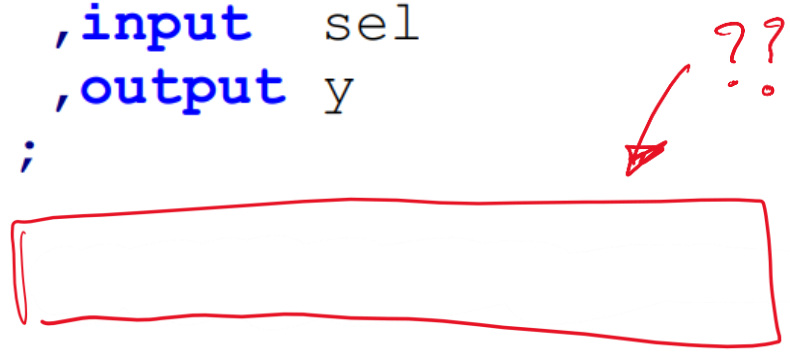
- Implementation of a MUX and simulation with Modelsim

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1 (
    input  a
   ,input  b
   ,input  sel
   ,output y
);

endmodule
```



Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1 (
    input  a
   ,input  b
   ,input  sel
   ,output y
);

  assign y = sel ? b & a;

endmodule
```

- **Continuous assignment**
  - **wire** signal
  - **syntax:** assign <signal> = <expression>;

One possibility to design combinatory networks.

has to be a wire

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1 (
    input  a
   ,input  b
   ,input  sel
   ,output y
);

  assign y = sel ? b & a;

endmodule
```

y = wire: ports are wire type by default

- **Continuous assignment**
  - **wire** signal
  - **syntax:** assign <signal> = <expression>;

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1 (
    input  a
   ,input  b
   ,input  sel
   ,output y
);

  assign y = sel ? b : a;

endmodule
```

- **Continuous assignment**
  - **wire** signal
  - **syntax:** assign <signal> = <expression>;
- Ternary operator
  - <condition> ? <if true> : <if false>
  - sel ? … → sel == 1 ? …
    - Non-zero value = true
    - Zero value = false

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1_v2 (
    input       a
   ,input       b
   ,input       sel
   ,output reg  y
);

  always_comb
    if(sel)
      y = b;
    else
      y = a;

endmodule
```

- **Blocking assignment**
  - **reg** signal
  - **always_comb** block
  - operator **=**

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1_v2 (
    input       a
   ,input       b
   ,input       sel
   ,output reg y
);

  always_comb
    if(sel)
      y = b;
    else
      y = a;

endmodule
```

- **Blocking assignment**
  - reg signal
  - always_comb block
  - operator =

To perform assignment on always_comb block, you need a reg signal

Always; means that the block is always working!

Comb = combinational

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_2_to_1_v2 (
    input       a
   ,input       b
   ,input       sel
   ,output reg  y
);

  always_comb
    if(sel)
      y = b;
    else
      y = a;

endmodule
```

- **Blocking assignment**
  - **reg** signal
  - **always_comb** block
  - operator **=**

- **if-else** statement:

  if (<condition>)

  <assignment>

  else

  <assignment>

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_4_to_1 (
    input             a
   ,input             b
   ,input             c
   ,input             d
   ,input       [1:0] sel
   ,output reg        y
);

  always_comb
    case(sel)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      2'b11: y = d;
    endcase

endmodule
```

at least 2 inputs in sel

- **case** statement:

```
case (<signal>)
    <case value 1> : <assignment>;
    <case value 2> : <assignment>;
    …
endcase
```

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_4_to_1 (
    input              a
   ,input              b
   ,input              c
   ,input              d
   ,input       [1:0] sel
   ,output reg         y
);

  always_comb
    case(sel)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      2'b11: y = d;
    endcase

endmodule
```

- **case** statement:

  case (<signal>)

  <case value 1> : <assignment>;

  <case value 2> : <assignment>;

  …

  endcase

- if not all the cases are specified, the **default** case must be included:

  …

  default: <assignment>;

  endcase

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_4_to_1 (
    input             a
   ,input             b
   ,input             c
   ,input             d
   ,input       [1:0] sel
   ,output reg        y
);

  always_comb
    case(sel)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      2'b11: y = d;
    endcase

endmodule
```

- Constants
  - **syntax**: <bit width>'<base><value>
  - examples:
    - 2'b00
    - 4'd0
    - 16'hFA3E
    - …

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim

```
module mux_4_to_1 (
    input             a
   ,input             b
   ,input             c
   ,input             d
   ,input       [1:0] sel
   ,output reg        y
);

  always_comb
    case(sel)
      2'b00: y = a;
      2'b01: y = b;
      2'b10: y = c;
      2'b11: y = d;
    endcase

endmodule
```

- Constants
  - syntax: <bit width>'**<base>**<value>
    - b = binary
    - d = decimal
    - h = hexadecimal

Exercise with SystemVerilog

- Additional note
  - If any of the previous assignment blocks (i.e., always_comb, if-else, case value) contains more than one assignment, the **begin – end** delimiters must be used

```
always_comb begin
  y1 = a;
  y2 = c;
end
```

```
always_comb
  if (sel) begin
    y1 = a;
    y2 = c;
  end
  else begin
    y1 = b;
    y2 = d;
  end
```

More than one statement too, equivalent of brackets in C

but two here

Just one for the always comb

```
always_comb begin

  if (sel1)
    y1 = a;
  else
    y1 = b;

  if(sel2)
    y2 = c;
  else
    y2 = d;

end
```

```
always_comb
  case(sel)
    2'b00: begin
      y1 = a;
      y2 = c;
    end

    2'b10: begin
    // ...
    end

    // ...

endcase
```

Exercise with SystemVerilog

- Implementation of a MUX and simulation with Modelsim
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.1
  - Try to simulate the different implementations of the MUX with Modelsim
    - Refer to Lecture 1.2

Add to wave selected WF

dopo aver selezionato i segnali.

Do file è un txt file con comandi

Separati da andata a capo e salvarli come file DO

Thank you for your attention

Luca Crocetti
(luca.crocetti@unipi.it)