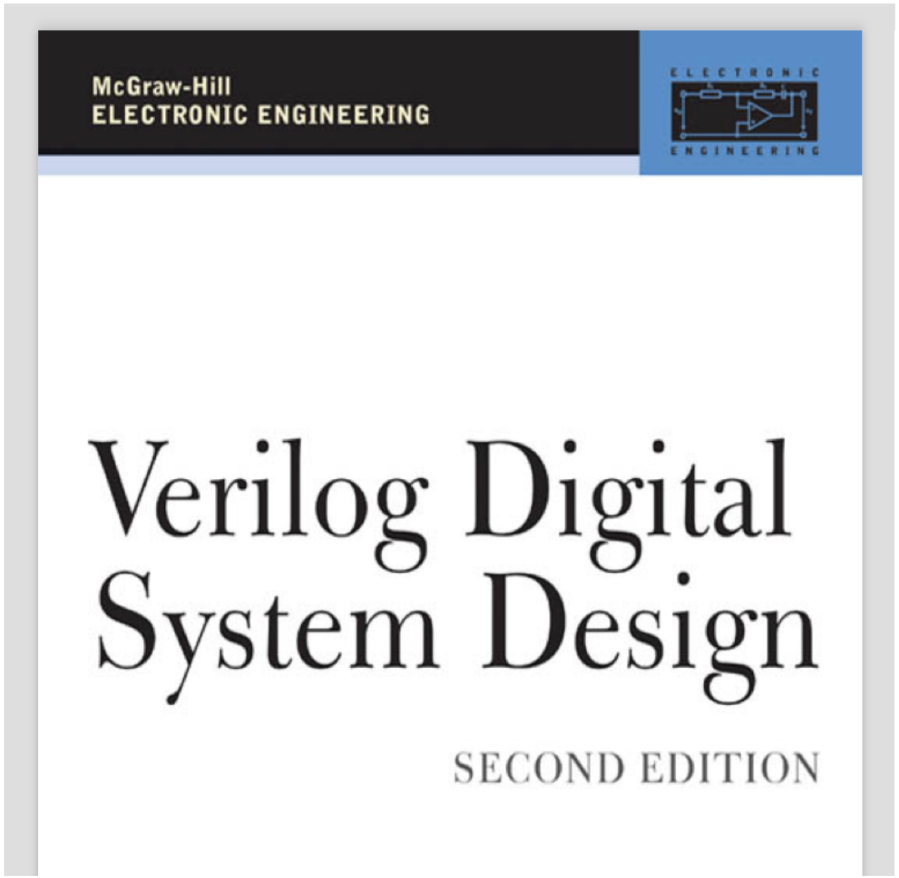
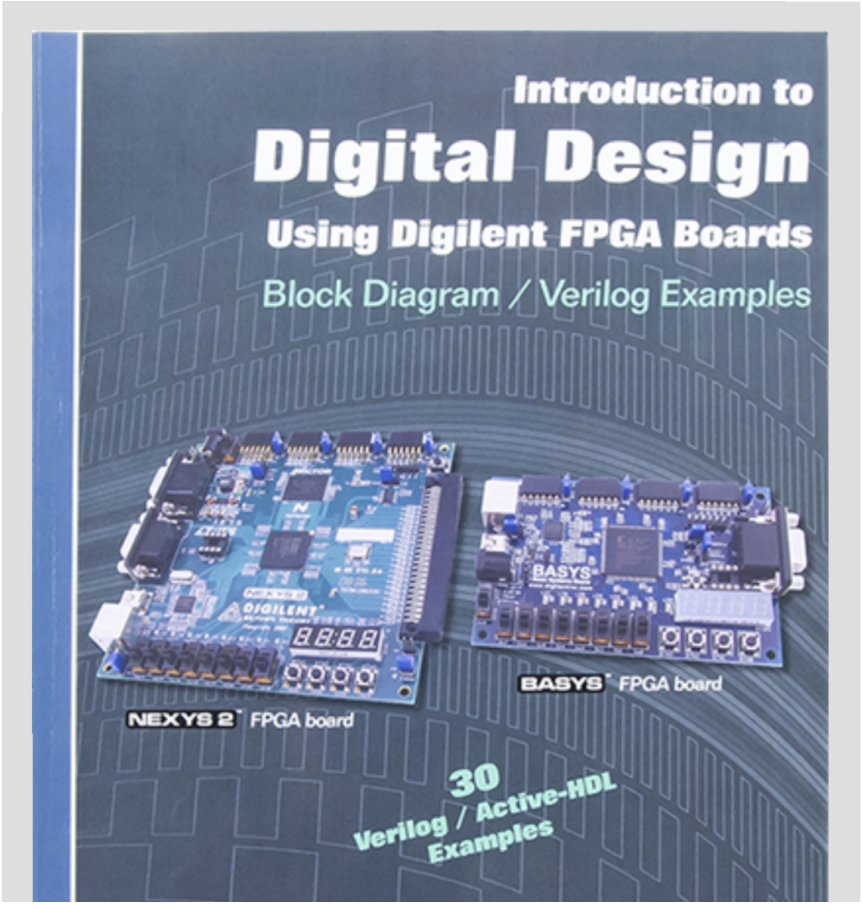
Introduction to digital systems design using Verilog HDL

References

- *Verilog Digital System Design*, Z. Navabi, McGraw-Hill, 2006
- *Introduction to Digital Design Using Digilent FPGA Boards - Block Diagram/Verilog Examples*, R.E. Haskell, D.M. Hanna, LBE Books, 2009





Additional references

A lot of material in form of **tutorials** and **YouTube videos** is available on the web :

- https://www.fpga4fun.com
- https://www.nandland.com/verilog/tutorials/index.html
- http://www.asic-world.com/verilog/index.html
- https://www.youtube.com/watch?v=PybxgAroozA&app=desktop
- https://www.youtube.com/playlist?list=PLB52B8F4E464CEEF7&app=desktop

Digital systems

analog signals :

- **continuous** in both **time** and **amplitude**
- usually a voltage $v(t)$ or a current $i(t)$ as a function of time
- reach of information (e.g. frequency spectrum, FFT)

digital signals :

- usually continuous in time but **discrete in amplitude**
- only **two possible values** e.g. high/low voltage levels, true/false, on/off, closed/open etc.
- less information, but **more robust against noise**
- can be either **asynchronous signals** or **synchronous signals**

Classification of digital circuits

Digital circuits are classified as :

- **combinational** circuits
- **sequential** circuits
  - asynchronous
  - synchronous

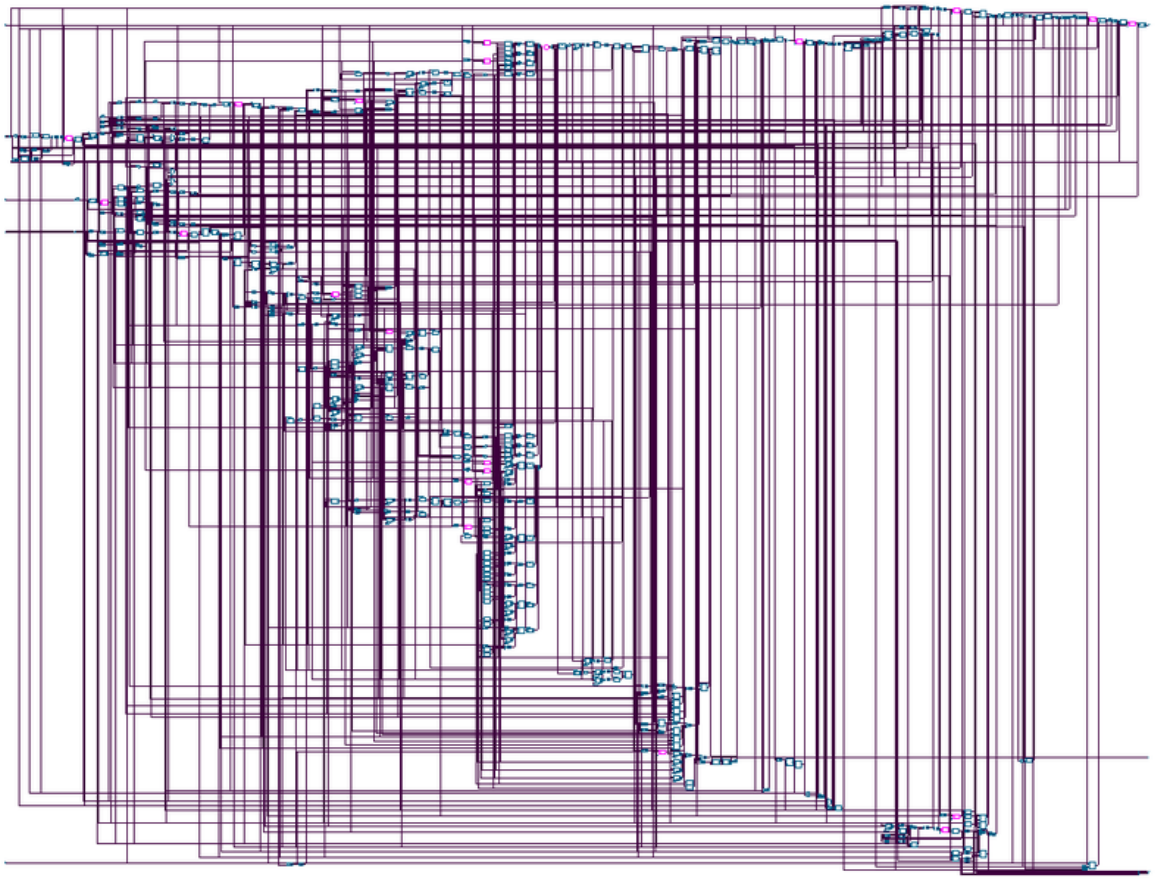
Hardware Description Languages (HDL)

COMPLEX digital system (e.g. micro-processor)

$\Rightarrow$ **HDL description and simulation** of the system, that is... write **CODE !**

$\Rightarrow$ a **SYNTHESIS tool** generates the **real HARDWARE** for you !

```
17  localparam S2 = 2'b11;
18  localparam S3 = 2'b10;
19
20  reg [1:0] pState, nState;
21
22  always @ (pState, start, stop, increment)
23  begin
24      case (pState)
25          S0:begin
26              if (start == 1'b0 && increment ==
27                  nState = S0;
28              else if (increment == 1'b1 && sta:
29                  nState = S2;
30              else
31                  nState = S1;
32              end
33          S1:
34          begin
35              if (stop == 1'b1)
36                  nState = S0;
37              else
38                  nState = S1;
39           end
40          S2:
41              nState = S3;
42          S3:
43          begin
44              if (increment == 1'b1)
45                  nState = S3;
46              else
47                  nState = S0;
48           end
49          default:
50              nState = S0;
51      endcase
```



Verilog vs. VHDL

Verilog and **VHDL** are the two most widespread HDLs in the world for this job :

- approx. 50% market each one
- Verilog more popular in US and Japan, VHDL in Europe
- Verilog more used (and integrated with professional CAD tools) to design Application-Specific Integrated Circuits (ASIC) design
- traditionally VHDL more used for FPGA programming instead
- both supported by Xilinx Vivado flows
- Verilog is **easier to learn**, but also potentially **more error-prone due to its relaxed data typing**
- if you will do my job at the end you will learn both, or at least you will be able to read both codes
- if you start **alone** (but I'm here ...) from scratch... learn VHDL first, then move to Verilog once really annoyed with VHDL "verbose" coding
- since VHDL is already used in the *Digital Electronics* course we will learn Verilog

Verilog HDL fundamentals

Verilog HDL

- created by P. Goel, P. Moorby, C.L. Huang and D. Warmke between late 1983 and early 1984
- introduced by *Gateway Design Automation*, later purchased by *Cadence Design Systems* in 1990
- born as a **verification** language for **logic** designs, at the beginning only intended to describe and simulate digital circuits
- initially a **proprietary** and **closed** language owned by *Cadence*
- later released by *Cadence* as an **open language** in order to cope with the increasing popularity of VHDL
- standardized in 1995 by IEEE (Institute for Electrical and Electronics Engineering) as **IEEE Std. 1364-1995**
- second revision of the language with major extensions and new language features in 2001, known as **IEEE Std. 1364-2001**
- third revision with minor changes in 2006, known as **IEEE Std. 1364-2006**
- finally merged as a sub-set of the **SystemVerilog** HDL as **IEEE Std. 1800-2009**

Syntax fundamentals

The Verilog HDL by purpose follows a syntax derived from the well-known **C programming language** :

- Verilog is **case sensitive** (on the contrary, VHDL is not case sensitive)
- **blanks** between statements and **empty lines** are ignored by the compiler
- **comments** can be **single-line** or **multiple-lines** as in C/C++

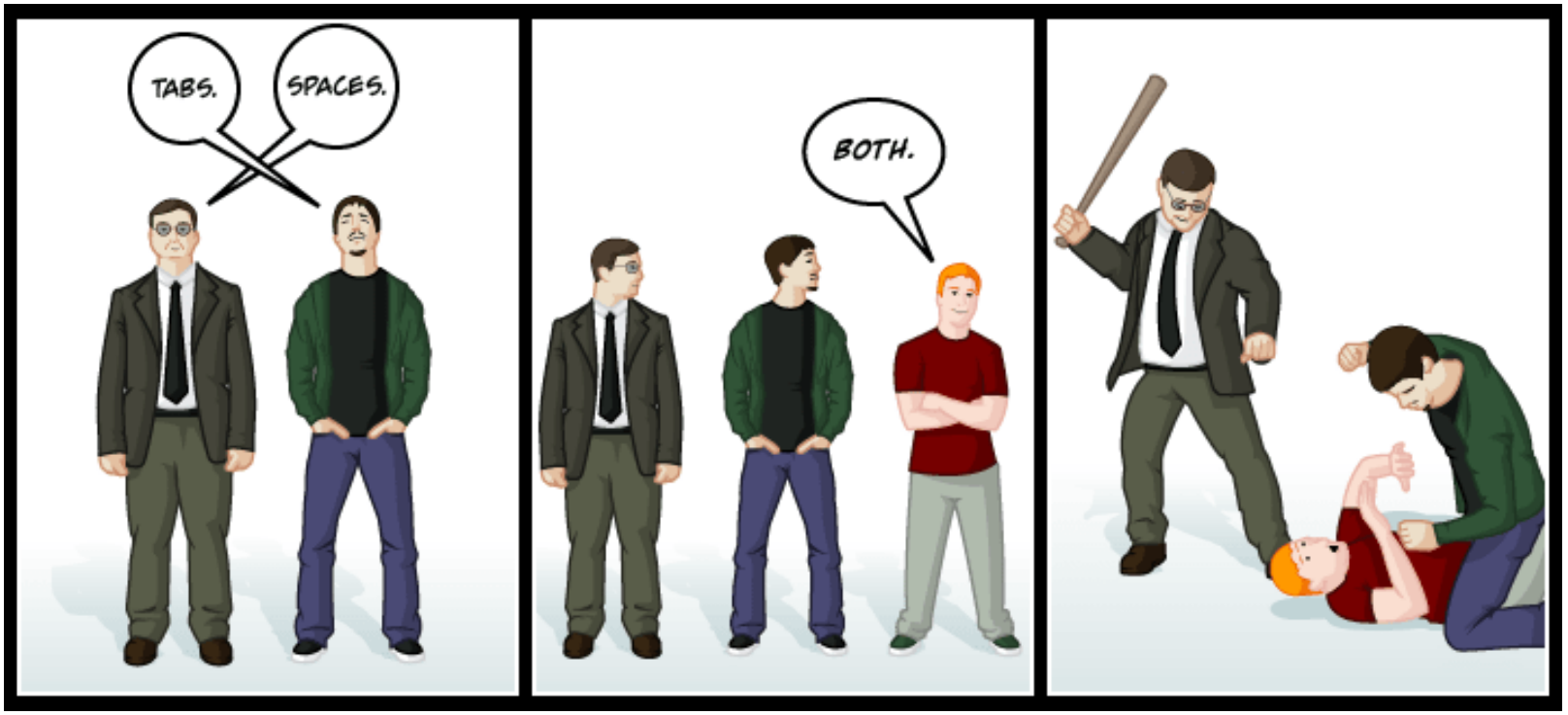
```
// this is a single-line C-style comment

/*
   this is a multiple-lines
   C-style comment
*/
```

Code indentation

Always **indent your code** to improve readability but ...

DO NOT USE TABs !!!

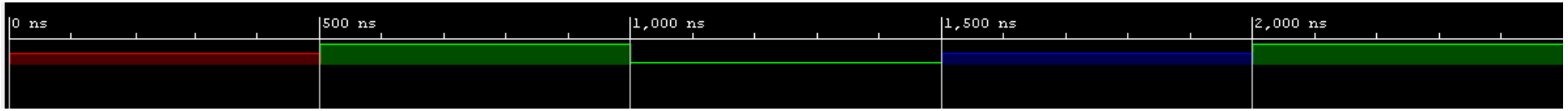


Logic values and timing diagrams

The Verilog HDL natively[1] provides **four logic values** `1`, `0`, `X` and `Z` :

- `1` for **logic-high** (true, closed, on, etc.)
- `0` for **logic-low** (false, open, off, etc.)
- `X` for **unknown**, can be either `0` or `1`
- `Z` for the **high-impedance** state

Logic **simulation results** are visualized in form of **timing diagrams**, also referred to as **waveforms** :



`X` `1` `0` `Z` `1`

[1] On the contrary, by default VHDL only provides the `bit` values `0/1` and you need to include `IEEE.std_logic_1164.all` to extend logic values.

Basic net data types: wires and registers

In Verilog **physical nets** can be either of type `wire` or `reg` :

- `wire` nets are physical signals assigned using **continuous assignments**
- `reg` nets are more like **variables** in programming languages and are assigned within **procedural blocks** such as `initial` and `always` (see later)

> *Declaring something as* `reg` *does not imply that a "register" (latch or FlipFlop) is inferred in real hardware !*

Additional net types exist for special purposes (not used in the course) :

`wand`, `wor`, `tri`, `triand`, `trior`, `supply0`, `supply1`

Scalars, buses and endianness

Single wires are called **scalar signals** :

```
wire sum, cout ;
reg  ZN ;
```

More wires sharing the same functionality can be **grouped together** to form a **bus** using **C-style arrays** :

```
wire [1:0] select ;
reg  [4:0] count = 5'b00000 ;
```

Any **binary word** always has a **Most Significant Bit (MSB)** and a **Least Significant Bit (LSB)** along with a well-defined **bit-ordering**, called **endiannes** :

```
wire [N-1:0] be_bus ;   // big-endian    => MSB ... LSB
wire [0:N-1] le_bus ;   // little-endian => LSB ... MSB
```

Slicing and concatenation

Since buses follow the syntax of C-style arrays :

- a single element of a bus can be accessed using the **index in the array** e.g. `wsb[3]`
- two or more **consecutive bits** of a bus are a **slice** and can be accessed using **two indices in the array** e.g. `wsb[7:3]`

Differently from C/C++ programming languages, **curly brackets** { and } are used to **concatenate** scalars or buses to build larger buses :

```
wire A,B ;

wire [1:0] select ;
assign select = {A,B} ;    // concatenation

wire [3:0] control ;

wire [5:0] LED ;
assign LED = {A,B,control[3:0]} ;    // concatenation
```

Numbers and radices

Buses are collections of bits that can be interpreted as **numbers** in some **radix (base)** :

```
busWidth'radix<value>
```

Available **base identifiers** in Verilog are :

- `'b` for base-2 i.e. **binary** numbers
- `'o` for base-8 i.e. **octal** numbers
- `'d` for base-10 i.e. **decimal** numbers
- `'h` for base-16 i.e. **hexadecimal** numbers

Example :

```
wire [11:0] number ;   // 12-bit bus

assign number = 12'b1010_1001_1011 ;   // straight binary
assign number = 12'hA9B  ;             // hexadecimal
assign number = 12'o5233 ;             // octal
assign number = 12'd2715 ;             // decimal
```

Additional useful data types: integer

integer numbers :

- **32-bit signed integer**
- same as `int` in C/C++
- **synthesizable**
- mainly used as **iterators** inside `for` and `while` loop statements

```
integer i ;

for(i=0 ; i<N ; i=i+1) begin
    ...
    ...
end
```

Implicit type casting

Verilog HDL is **extremely relaxed** with respect to VHDL in **handling different data types** and assignments !

Example :

```
// this syntax compiles, however...
reg [4:0] count = 0 ;
```

- `count` is declared as a 5-bit bus, but `0` is a 32-bit integer !
- a VHDL compiler will never accept such an assignment, while Verilog implicitly performs a **type cast** for you

As a good coding practice **always specify the right size** to avoid **unexpected results after synthesis** :

```
reg [4:0] count = 5'b00000 ;     // or simply 5'd0
```

Additional useful data types: real

real numbers :

- **64 bit IEEE double-precision floating point number**
- same as `double` in C/C++
- **NOT synthesizable !**
- mainly used for **simulation purposes** or to **model A/D and D/A converters**

```
parameter real PERIOD = 50.0 ;

reg clk = 1'b0 ;
always #(PERIOD/2.0) clk = ~ clk ;
```

time and **realtime** numbers :

- `$time` returns a **64-bit integer value** for the simulation time
- `$realtime` returns a **64-bit real value** for the simulation time
- only for simulation purposes

Modules

Any **digital block** implementing some functionality in Verilog is called `module` :

```
module ModuleName (
    ...
    ... ) ;

    ...
    ...
endmodule
```

A module has **I/O ports** that can be declared as :

- `input`
- `output`
- `inout`

By default, all ports are considered of net type `wire` unless expilicitely declared as `reg`.

Testbench module

In order to **simulate**[2] the functionality of the digital block we also need a **testbench module** that generates **stimuli** fed to input ports of our **Module Under Test (MUT)**, also referred to as **Device Under Test (DUT)** :

```
module tb_ModuleName ;
    ...
    ...

    // Device Under Test (DUT)
    ModuleName DUT (....) ;

    ...
endmodule
```

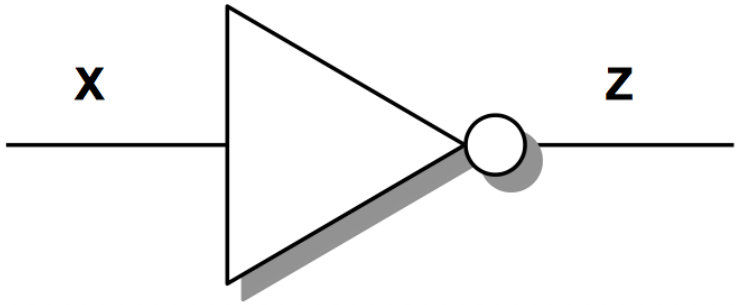
The module under test is always **instantiated** inside the testbench module, which contains **non-synthesizable code**.

[2] In semiconductor industry there is a ratio of about 10:1 between verification engineers and HDL designers.

Lab 1 - A simple inverter

Lab 2 - Fundamental logic gates in Verilog

NOT gate



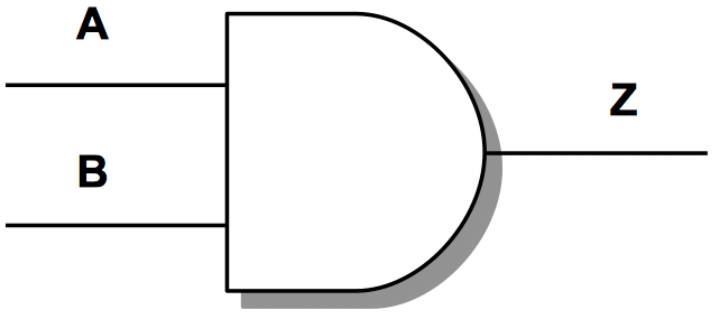
$$Z = \overline{X}$$

Verilog syntax :

```
// continuous assignment
assign Z = ~ X ;

// primitive instantiation
not u1 (Z,X) ;
```

AND gate



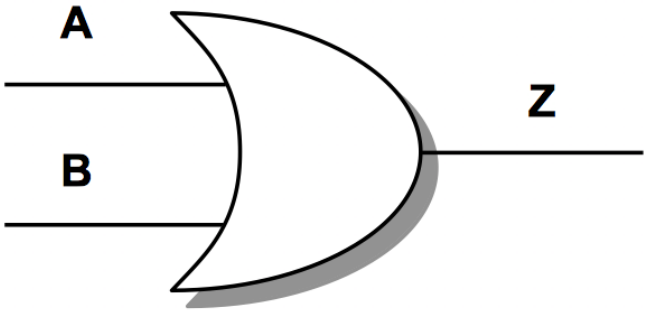
$$Z = A \cdot B$$

Verilog syntax :

```
// continuous assignment
assign Z = A & B ;

// primitive instantiation
and u2 (Z,A,B) ;
```

OR gate



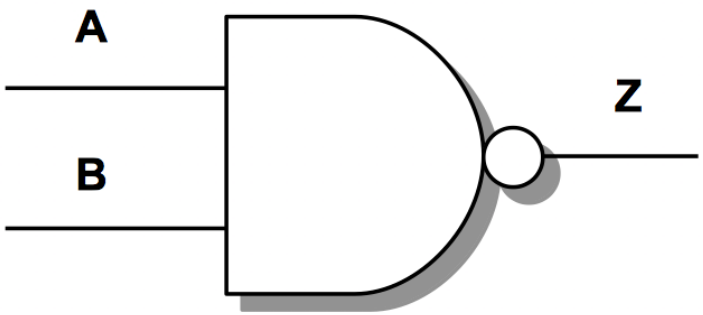
$$Z = A + B$$

Verilog syntax :

```
// continuous assignment
assign Z = A | B ;

// primitive instantiation
or u3 (Z,A,B) ;
```

NAND gate



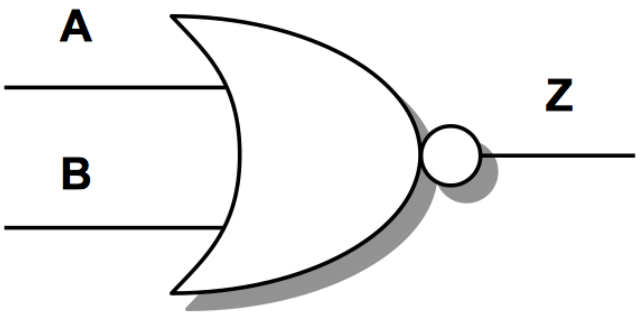
$$Z = \overline{A \cdot B}$$

Verilog syntax :

```
// continuous assignment
assign Z = ~(A & B) ;

// primitive instantiation
nand u4 (Z,A,B) ;
```

NOR gate



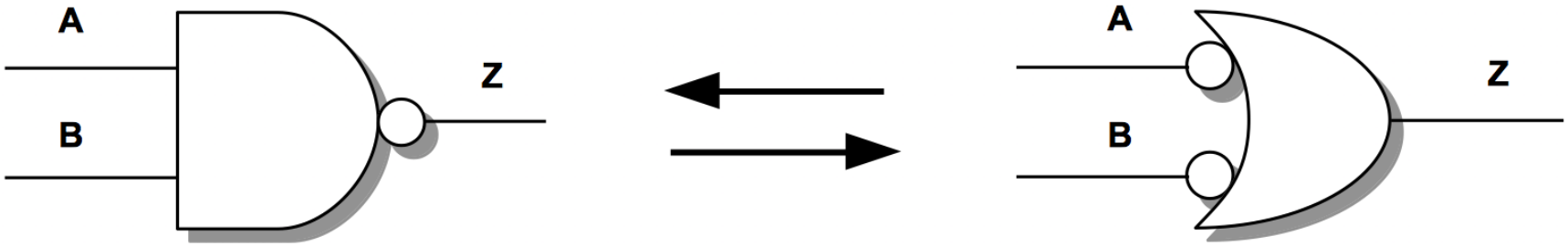
$$Z = \overline{A + B}$$

Verilog syntax :

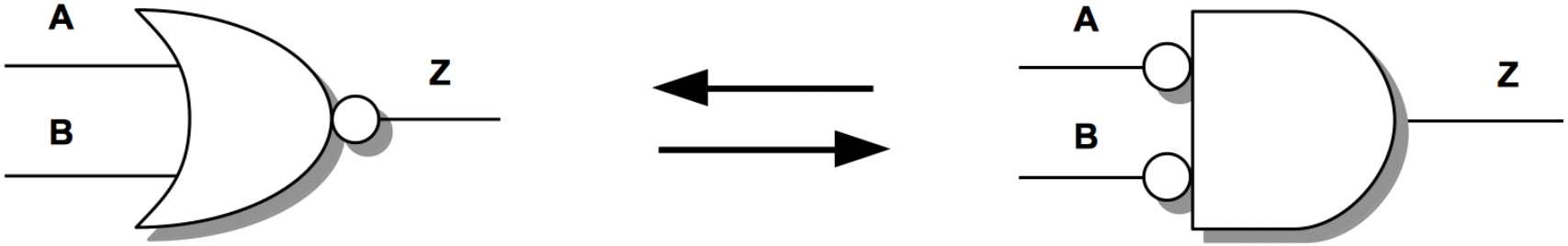
```
// continuous assignment
assign Z = ~(A | B) ;

// primitive instantiation
nor u5 (Z,A,B) ;
```

De Morgan's theorem



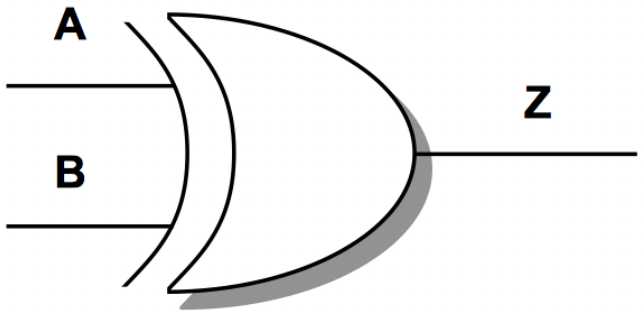
$$\overline{A \cdot B} = \overline{A} + \overline{B}$$



$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Thanks to the De Morgan's theorem NAND and NOR gates are promoted to **universal gates**.

XOR gate



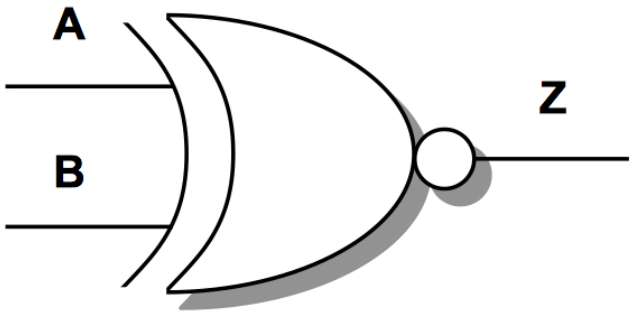
$$Z = A \oplus B = (A \cdot \overline{B}) + (\overline{A} \cdot B)$$

Verilog syntax :

```
// continuous assignment
assign Z = A ^ B ;

// primitive instantiation
xor u6 (Z,A,B) ;
```

XNOR gate



$$Z = \overline{A \oplus B} = (A \cdot B) + (\overline{A} \cdot \overline{B})$$

Verilog syntax :

```
// continuous assignment
assign Z = ~(A ^ B) ;

// primitive instantiation
xnor u7 (Z,A,B) ;
```

Lab 3 - Different coding styles for a simple 2:1 multiplexer

Lab 4 - Example combinational blocks