# Advanced Electronics Laboratory / Part I

**Dott. Luca Pacher**

*luca.pacher@cern.ch*

*University of Torino*

A.Y. 2020/2021, Spring 2021

## Course introduction and overview

## Introduction

**Part I - Dott. Luca Pacher**

- introduction to FPGA programming using Xilinx Vivado and Verilog HDL
- course delivered in form of **remote video-lectures** (Webex) + **practical labs**
- 40 hours (4 CFU)

**Part II - Prof. Michela Greco**

- introduction to micro-controllers programming using Arduino
- introduction to LabView programming for Data Acquisition (DAQ) systems
- 20 hours (2 CFU)

*For the second part, details from Prof. Greco*

## Remote lectures

- 2 hours per lecture
- always connect to my personal Webex room:

*https://unito.webex.com/meet/luca.pacher*

- **all lectures will be recorded !**
- links to recorded sessions in the main course page on CampusNet
- I will also put the link in the main `README` file on the GitHub repository (more details about Git later on)

## Exam

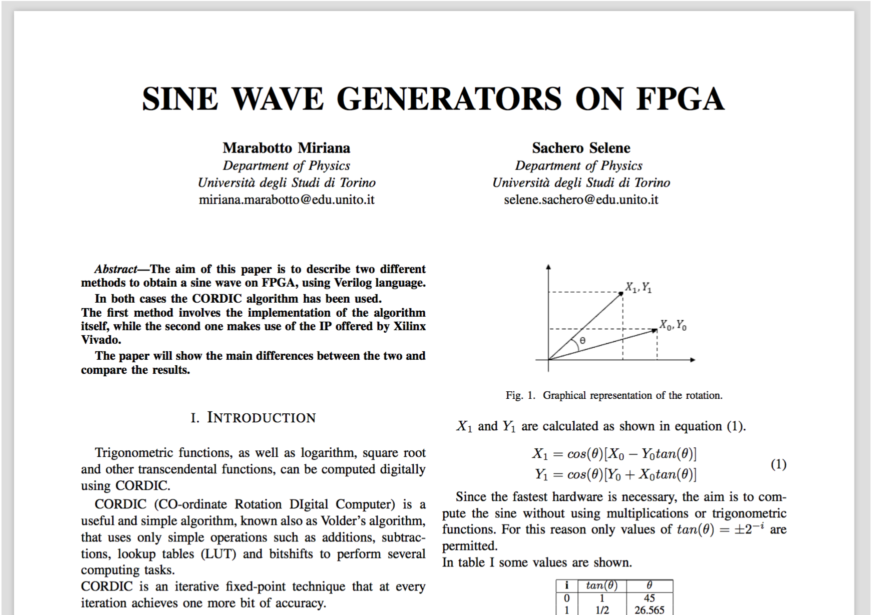
Also the exam is split into two parts :

– one grade for each part
– the final grade will be the **weighted mean** between the two
– you are not requested to take the exam in the same session
– "official" exam days, but you can arrange with teachers according to your needs

For the first part :

– **design and simulation** of a small digital system using Verilog HDL and Vivado (max. 2 students per project)
– **description of your project** in form of a **short report in English** (max. 8-10 pages) using LaTex, *"paper style"* (as it happens in the real research life)
– **oral presentation of your project** with **slides in English**, *"conference style"* (as it happens in the real research life) followed by a few questions

## To make you feel like a "pro" ...



# SINE WAVE GENERATORS ON FPGA

**Marabotto Miriana**
*Department of Physics*
*Università degli Studi di Torino*
miriana.marabotto@edu.unito.it

**Sachero Selene**
*Department of Physics*
*Università degli Studi di Torino*
selene.sachero@edu.unito.it

***Abstract*—The aim of this paper is to describe two different methods to obtain a sine wave on FPGA, using Verilog language.**

**In both cases the CORDIC algorithm has been used.**
**The first method involves the implementation of the algorithm itself, while the second one makes use of the IP offered by Xilinx Vivado.**

**The paper will show the main differences between the two and compare the results.**

## I. Introduction

Trigonometric functions, as well as logarithm, square root and other transcendental functions, can be computed digitally using CORDIC.

CORDIC (CO-ordinate Rotation DIgital Computer) is a useful and simple algorithm, known also as Volder's algorithm, that uses only simple operations such as additions, subtractions, lookup tables (LUT) and bitshifts to perform several computing tasks.
CORDIC is an iterative fixed-point technique that at every iteration achieves one more bit of accuracy.

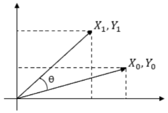


Fig. 1. Graphical representation of the rotation.

$X_1$ and $Y_1$ are calculated as shown in equation (1).

$$\begin{aligned} X_1 &= cos(\theta)[X_0 - Y_0 tan(\theta)] \\ Y_1 &= cos(\theta)[Y_0 + X_0 tan(\theta)] \end{aligned} \tag{1}$$

Since the fastest hardware is necessary, the aim is to compute the sine without using multiplications or trigonometric functions. For this reason only values of $tan(\theta) = \pm 2^{-i}$ are permitted.
In table I some values are shown.

| i | $tan(\theta)$ | $\theta$ |
|---|---|---|
| 0 | 1 | 45 |
| 1 | 1/2 | 26.565 |

## Final project

*Feel free to implement whatever you want ! ...*

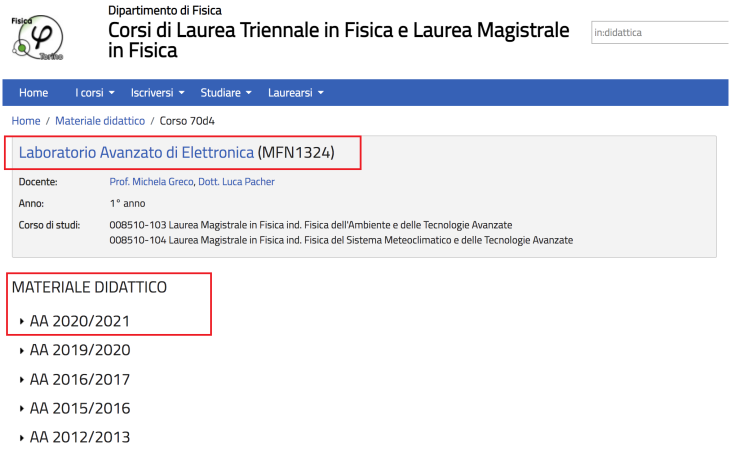
... or simply google for "FPGA project", "Verilog project" or "VHDL project" etc.

In case you really run out of ideas :

- a waveform generator
- a FIFO in Verilog (we will compile a soft IP for lectures instead...)
- a master/slave system connected through UART protocol
- a master/slave system connected through SPI protocol
- a master/slave system connected through JTAG protocol
  as from IEEE Std. 1149.1-2001
- a clock-domain-crossing (CDC) FIFO
- floating-point arithmetic (e.g. compute $\sqrt{x}$ or $\sin x$ in real hardware)
- etc.

## Main CampusNet course page

- *https://fisica.campusnet.unito.it/do/corsi.pl/Show?_id=70d4*
- slides and links to video-recorded lectures



Dipartimento di Fisica

Corsi di Laurea Triennale in Fisica e Laurea Magistrale in Fisica

Home / Materiale didattico / Corso 70d4

Laboratorio Avanzato di Elettronica (MFN1324)

| | |
|---|---|
| Docente: | Prof. Michela Greco, Dott. Luca Pacher |
| Anno: | 1° anno |
| Corso di studi: | 008510-103 Laurea Magistrale in Fisica ind. Fisica dell'Ambiente e delle Tecnologie Avanzate<br>008510-104 Laurea Magistrale in Fisica ind. Fisica del Sistema Meteoclimatico e delle Tecnologie Avanzate |

MATERIALE DIDATTICO

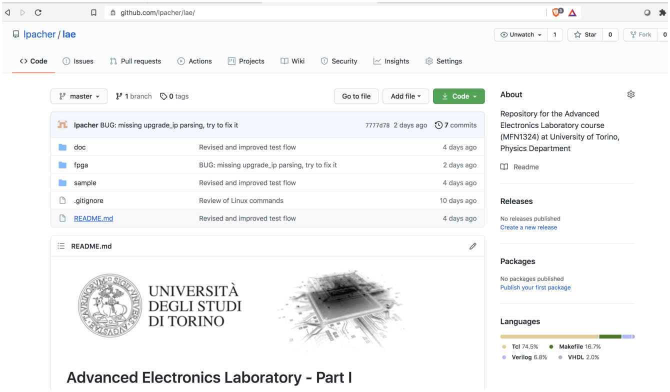
- AA 2020/2021
- AA 2019/2020
- AA 2016/2017
- AA 2015/2016
- AA 2012/2013

## GitHub repository /1

**GitHub repository :**

– *https://github.com/lpacher/lae*
– setup scripts, RTL sources, Tcl scripts, Makefiles, XDCs (more details later on ...)

*Register and create a GitHub account !*


github.com/lpacher/lae/

lpacher / lae

Unwatch 1 Star 0 Fork 0

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master 1 branch 0 tags Go to file Add file Code

lpacher BUG: missing upgrade_ip parsing, try to fix it 7777d78 2 days ago 7 commits

| | | |
|---|---|---|
| doc | Revised and improved test flow | 4 days ago |
| fpga | BUG: missing upgrade_ip parsing, try to fix it | 2 days ago |
| sample | Revised and improved test flow | 4 days ago |
| .gitignore | Review of Linux commands | 10 days ago |
| README.md | Revised and improved test flow | 4 days ago |

README.md

UNIVERSITÀ DEGLI STUDI DI TORINO

Advanced Electronics Laboratory - Part I

About

Repository for the Advanced Electronics Laboratory course (MFN1324) at University of Torino, Physics Department

Readme

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● Tcl 74.5% ● Makefile 16.7% ● Verilog 6.8% ● VHDL 2.0%

## GitHub repository /2

In this course we will **write a lot of code** !

- Verilog RTL and testbech sources
- simulation and implementation scripts in Tcl, GNU Makefiles
- timing and physical constraints (XDC)
- XML configuration files for IPs (XCI) etc.

Final "working solutions" for all examples proposed during lectures will be uploaded to our GitHub repository before each lecture. To **get all updates** just use `git` from the command line :

```
cd /path/to/lae
git pull origin master
```

Detailed information about Git installation and configuration can be found in the main `README` file *https://github.com/lpacher/lae/blob/master/README.md*

I would also recommend to use Git for your final project !

## Course program

- Verilog HDL fundamentals, HDL design flow
- Logic values, resolved vs. unresolved logic values, 3-state logic, buses and endianess
- review of boolean algebra
- introduction to Xilinx Vivado simulation and implementation flows
- design and simulation of combinational circuits with Verilog examples (multiplexers, decoders, encoders etc.)
- FPGA architectures overview and basic building blocks (fabric, BEL, LUT, CLB, CLA, slices, IOBs, hard-macros)
- introduction to Xilinx Design Constraints (XDCs)
- sequential circuits, latches and FlipFlops
- counters, registers, PWM, shift-registers, FSM, FIFOs, RAM/ROM
- advanced Xilinx Design Constraints (XDCs): timing fundamentals
- synchronous design good and bad practices, example Vivado IP flows (clock wizard, FIFO compiler)
- gate-level simulations with back-annotated delays (SDF)
- practical implementation and test of small digital systems targeting a Xilinx Artix-7 FPGA device

## Questions ?

# Introduction to digital systems design using Verilog HDL

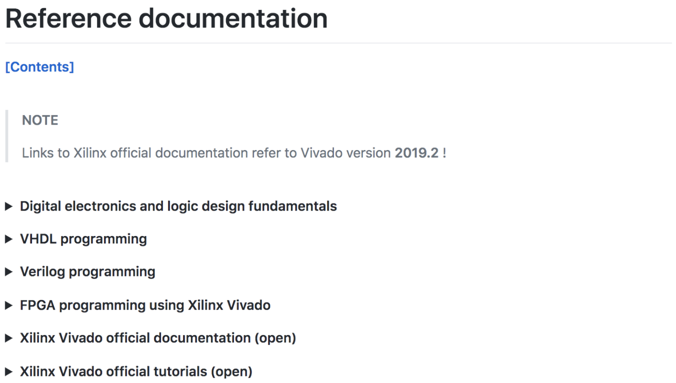
## References

No need to buy a book for the course, the provided course material is enough.
In case you want to deepen yourself your knowledge about FPGA programming:

- *Digital Design, Principles and Practice*
  J.F. Wakerly, Prentice Hall, 2005
- *Digital Electronics, Principles, Devices and Applications*
  A.K. Maini, Wiley&Sans, 2007
- *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
- *FPGA Prototyping By Verilog Examples*
  P.P. Chu, Wiley, 2008

## Other resources

- a lot of material in form of **tutorials**, **open documentation** and **YouTube videos** is available on the web
- complete list of reference material available in the main `README` file on GitHub



Reference documentation

[Contents]

NOTE

Links to Xilinx official documentation refer to Vivado version **2019.2** !

▶ Digital electronics and logic design fundamentals

▶ VHDL programming

▶ Verilog programming

▶ FPGA programming using Xilinx Vivado

▶ Xilinx Vivado official documentation (open)

▶ Xilinx Vivado official tutorials (open)

## 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** :

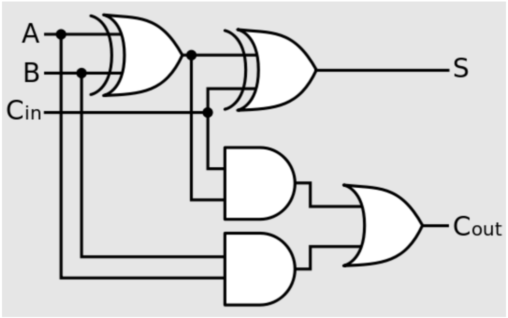
- usually continuous in time but **discrete in amplitude**
- only **two possible values** e.g. high/low voltage levels, true/false, on/off, closed/open etc.
- less information, but **more robust against noise**
- can be either **asynchronous signals** or **synchronous signals**

## Classification of digital circuits

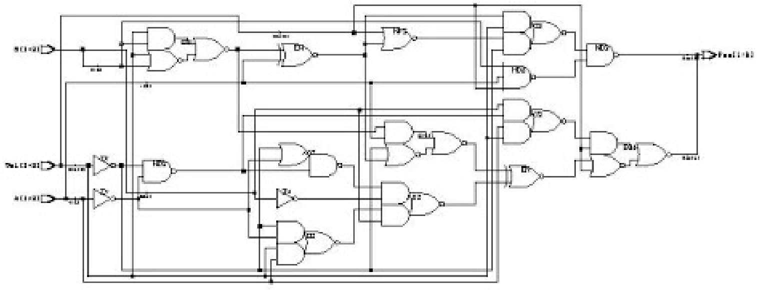
Digital circuits are classified as :

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

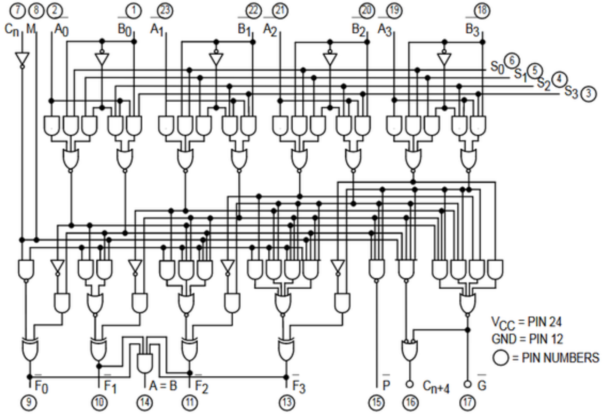
## Hardware Description Languages (HDL) /1



## Hardware Description Languages (HDL) /2

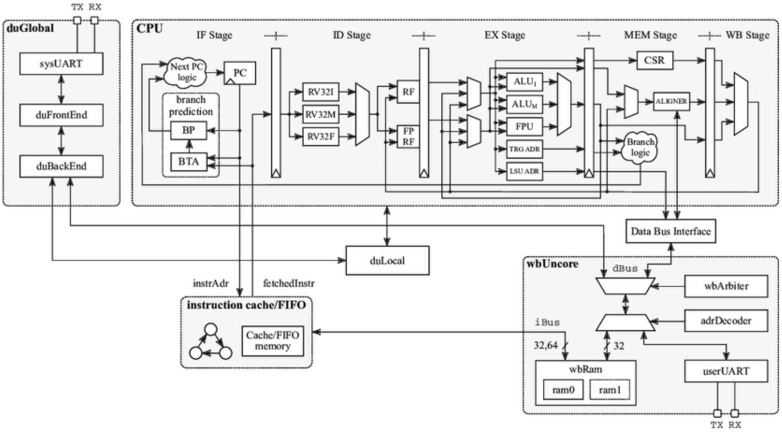


# Hardware Description Languages (HDL) /3



7 8 2 1 23 22 21 20 19 18
$C_n$ M $\overline{A_0}$ $\overline{B_0}$ $\overline{A_1}$ $\overline{B_1}$ $\overline{A_2}$ $\overline{B_2}$ $\overline{A_3}$ $\overline{B_3}$
$S_0$ 6 $S_1$ 5 $S_2$ 4 $S_3$ 3
$V_{CC}$ = PIN 24
GND = PIN 12
= PIN NUMBERS
$\overline{F_0}$ $\overline{F_1}$ A = B $\overline{F_2}$ $\overline{F_3}$ $\overline{P}$ $C_{n+4}$ $\overline{G}$
9 10 14 11 13 15 16 17

# Hardware Description Languages (HDL) /4

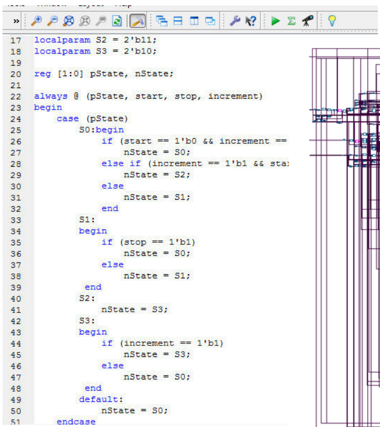


## Hardware Description Languages (HDL) /5

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

⇒ **HDL description and simulation** of the system, that is... write **CODE !**

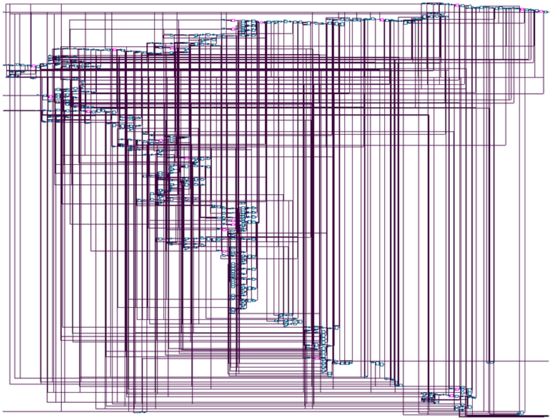
⇒ a **SYNTHESIS tool** generates the **real HARDWARE** for you !



```
localparam S2 = 2'b11;
localparam S3 = 2'b10;

reg [1:0] pState, nState;

always @ (pState, start, stop, increment)
begin
    case (pState)
        S0:begin
            if (start == 1'b0 && increment ==
                nState = S0;
            else if (increment == 1'b1 && sta
                nState = S2;
            else
                nState = S1;
            end
        S1:
        begin
            if (stop == 1'b1)
                nState = S0;
            else
                nState = S1;
         end
        S2:
            nState = S3;
        S3:
        begin
            if (increment == 1'b1)
                nState = S3;
            else
                nState = S0;
         end
        default:
            nState = S0;
    endcase
```



## Hardware Description Languages (HDL) /6

- Hardware Description Languages (HDLs) :
  - they are NOT "programming" languages !
  - provide constructs for both **implementation** (describe the functionality of digital circuits) and **verification** (simulations, assertions etc.)
  - synthesis (first step of modern digital design flow)
  - "synthesizable" constructs are only a minimal subset of the full language
- two main options on the market :
  - VHDL
  - Verilog
- advanced options :
  - high-level synthesis (HLS) using C/SystemC
  - SystemVerilog
  - analog and mixed-signal simulations (Verilog-A, Verilog-AMS, VHDL-AMS)

## VHDL

- VHDL = VHSIC-HDL (Hardware Description Language)
- VHSIC = Very High Speed Integrated Circuit
- born in 1983 as a project from US Department of Defense (DoD)
- syntax derived from the **ADA programming language** (similar to PASCAL) as requested by DoD
- initially foreseen as a language used to **simulate** digital integrated circuits
- later used also for **digital synthesis**
- 1987: first standardization as **IEEE Std. 1076-1987** (aka "VHDL 87")
- 1993: **first major revision** of the language as **IEEE Std. 1076-1993** (aka "VHDL 93")
- 2008: **second major revision** of the language as **IEEE Std. 1076-2008** (aka "VHDL 2008")
- provides constructs for both **physical implementation** (synthesizable) and **simulation**
- very reach syntax
- extremely **verbose** and **strongly typed** !

## Verilog

- created by P. Goel, P. Moorby, C.L. Huang and D. Warmke between late 1983 and early 1984
- syntax derived from the **C programming language**
- introduced by *Gateway Design Automation*, later purchased by *Cadence Design Systems* in 1990
- born as a **verification** language for **logic** designs, intended to describe and simulate digital integrated circuits similar to VHDL
- 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 (already standardized by IEEE in 1987)
- 1995: first standardization by IEEE as **IEEE Std. 1364-1995** (aka "Verilog 95")
- 2001: **first major revision** of the language with extensions and new language features known as **IEEE Std. 1364-2001** (aka "Verilog 2001")
- 2006: **second major revision** with minor changes as **IEEE Std. 1364-2006** (aka "Verilog 2006")
- finally merged as a sub-set of the **SystemVerilog** HDL as **IEEE Std. 1800-2009**

## Verilog vs. VHDL

Verilog and VHDL are the two most widespread HDLs in the world :

– 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 (ASICs)**

– traditionally VHDL more used for **FPGA programming** instead

– both **equally and well supported by Xilinx**

– exception: gate-level timing simulations only supports Verilog netlists

– individual preference, usually mostly historical (that is, first language learned...)

– Verilog is **easier to learn** (C-like syntax), but also potentially **more error-prone** due to its **relaxed data typing**

– that is... you can make **real disasters on silicon** with Verilog, while using VHDL is pretty impossible

– 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** from scratch... **learn VHDL first**, then move to Verilog once really annoyed with VHDL "verbose" coding

## Digital simulators

Many different digital simulators available on the market.

From the "big threes" :

- *QuestaSim* (the professional version of *ModelSim*) by Mentor (now Siemens)
- *Xcelium* (legacy *Incisive*) by Cadence
- *VCS* by Synopsys

Integrated with FPGA programming suites :

- *XSim* as part of the *Vivado* design suite by Xilinx
- *ISim* as part of the legacy *ISE* design suite by Xilinx
- *ModelSim Altera* as part of the *Quartus* design suite by Altera (now Intel)

Other solutions :

- *Aldec HDL* by Aldec
- open source simulators ( *FreeHDL, Icarus, GHDL etc,)*

## Verilog HDL syntax fundamentals

## Lab 1 - A simple inverter in Verilog

## 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++

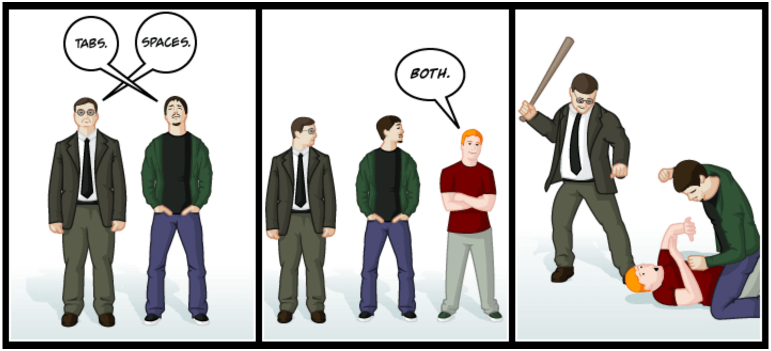
```
// this is a single-line C-style comment

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

## Code indentation

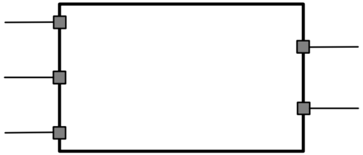
Always **indent your code** to improve readability but ...

**DO NOT USE TABs !!!**



## 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 explicitly declared as `reg` (more later on net data types).

## Testbench module

In order to **simulate**[1] 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**.

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

## Verilog data types

## Logic constants

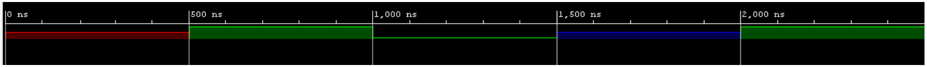
The Verilog HDL natively[2] provides **four logic values** `1`, `0`, `X` and `Z` :

- `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

[2]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.

## Timing diagrams (aka "waveforms")

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



X 1 0 Z 1

## 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
```

## Binary codes (most important... )

- "straight" binary (the usual "power of 2")
- Gray code
- thermometer
- one-hot / one-cold

Many many others : e.g. Hamming (for SEU protection)

## Straight binary code

Usual binary strings interpreted as usual as "power of 2" integer numbers :

```
3'b000  // 0
3'b001  // 1
3'b010  // 2
3'b011  // 3
3'b100  // 4
3'b101  // 5
3'b110  // 6
3'b111  // 7
```

$$x = b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + ... + b_{N-1} 2^{N-1} = \sum_{i=0}^{N-1} b_i \, 2^i$$

## Gray code

Only one bit change from a code to the next one :

```
        3'b000  // 0
        3'b001  // 1
        3'b011  // 2
        3'b010  // 3
        3'b110  // 4
        3'b111  // 5
        3'b101  // 6
        3'b100  // 7
```

## Thermometer code

Continue to "padd" the code with a 1 on the right :

```
    7'b0000000  // 0
    7'b0000001  // 1
    7'b0000011  // 2
    7'b0000111  // 3
    7'b0001111  // 4
    7'b0011111  // 5
    7'b0111111  // 6
    7'b1111111  // 7
```

## One-hot code

Only one bit set to one, then moving to the left :

```
        7'b0000000  // 0
        7'b0000001  // 1
        7'b0000010  // 2
        7'b0000100  // 3
        7'b0001000  // 4
        7'b0010000  // 5
        7'b0100000  // 6
        7'b1000000  // 7
```

The complementary is sometimes called "one cold"

## 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 ;
```

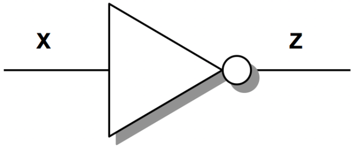
**time** and **realtime** numbers :

- $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

## Boolean algebra, combinational circuits

## Lab 2 - Fundamental logic gates in Verilog

## NOT gate



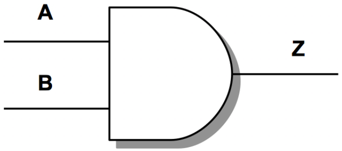
$$Z = \overline{X}$$

Verilog syntax :

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

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

## AND gate



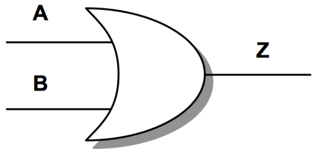
$$Z = A \cdot B$$

Verilog syntax :

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

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

## OR gate



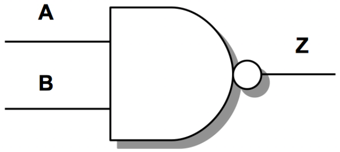
$$Z = A + B$$

Verilog syntax :

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

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

## NAND gate



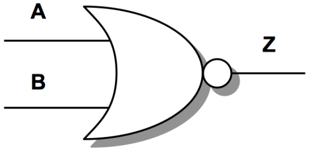
$$Z = \overline{A \cdot B}$$

Verilog syntax :

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

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

## NOR gate



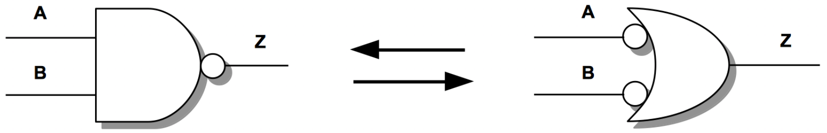
$$Z = \overline{A + B}$$

Verilog syntax :

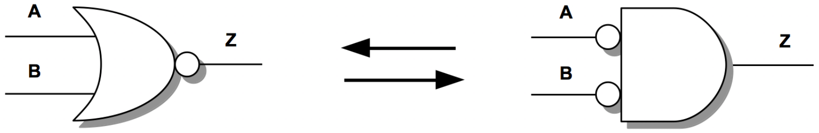
```
// continuous assignment
assign Z = ~(A | B) ;

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

## De Morgan's theorem



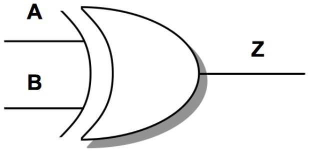
$$\overline{A \cdot B} = \overline{A} + \overline{B}$$



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

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

## XOR gate



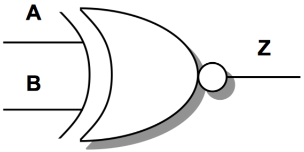
$$Z = A \oplus B = (A \cdot \overline{B}) + (\overline{A} \cdot B)$$

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