

ELEN0040 – Digital Electronics 2023 - Introduction to VHDL and project



LIÈGE université
Sciences Appliquées

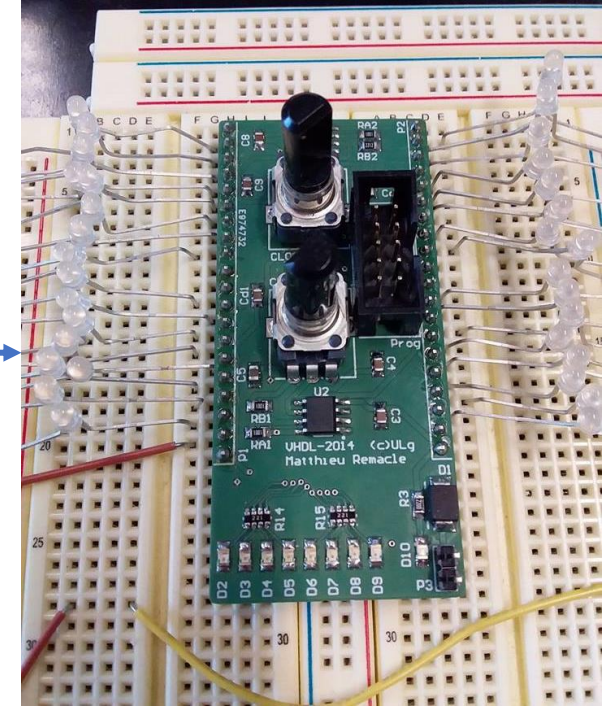
Arthur Fyon
afyon@uliege.be

Contact & information

- Teaching assistants:
 - Anaïs Halin (anaïs.halin@uliege.be) → Tutorials
 - Arthur Fyon (afyon@uliege.be) → Labs & Project
Office: 1.15 @ B28 Montefiore
- Course materials:
 - ~~Website~~ ([link](#))
 - Everything on myUliege

A few words about the project

The goal: to realize a real time system on a CPLD programmed in VHDL



VHDL

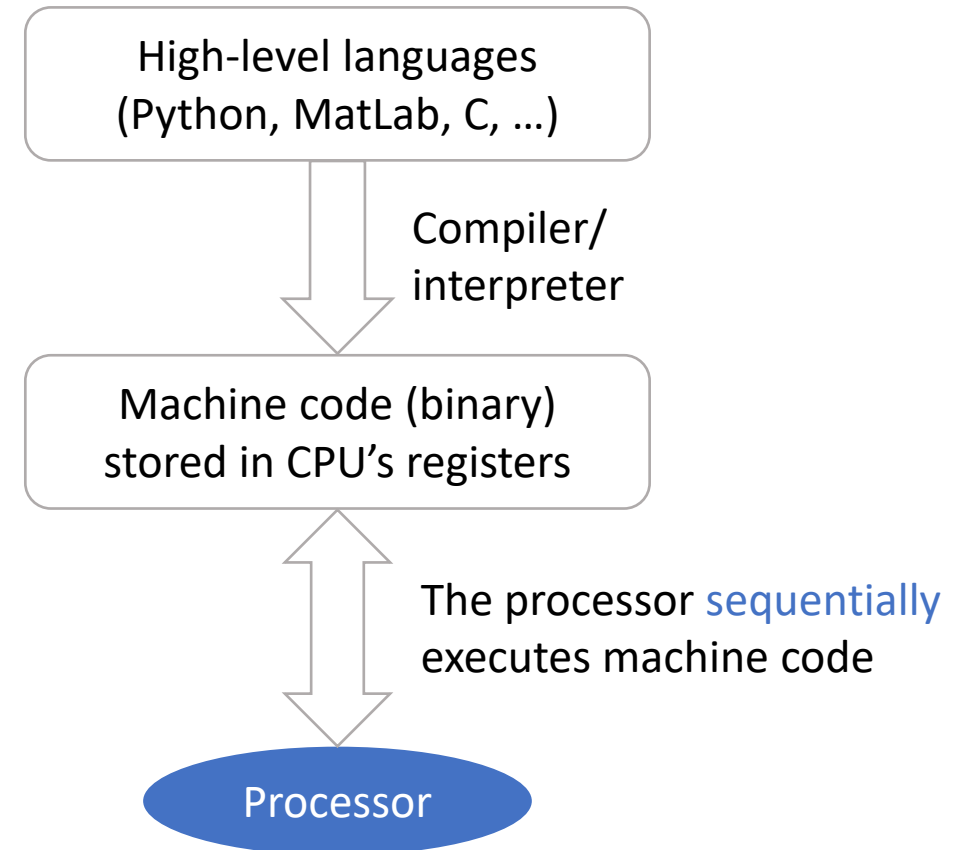
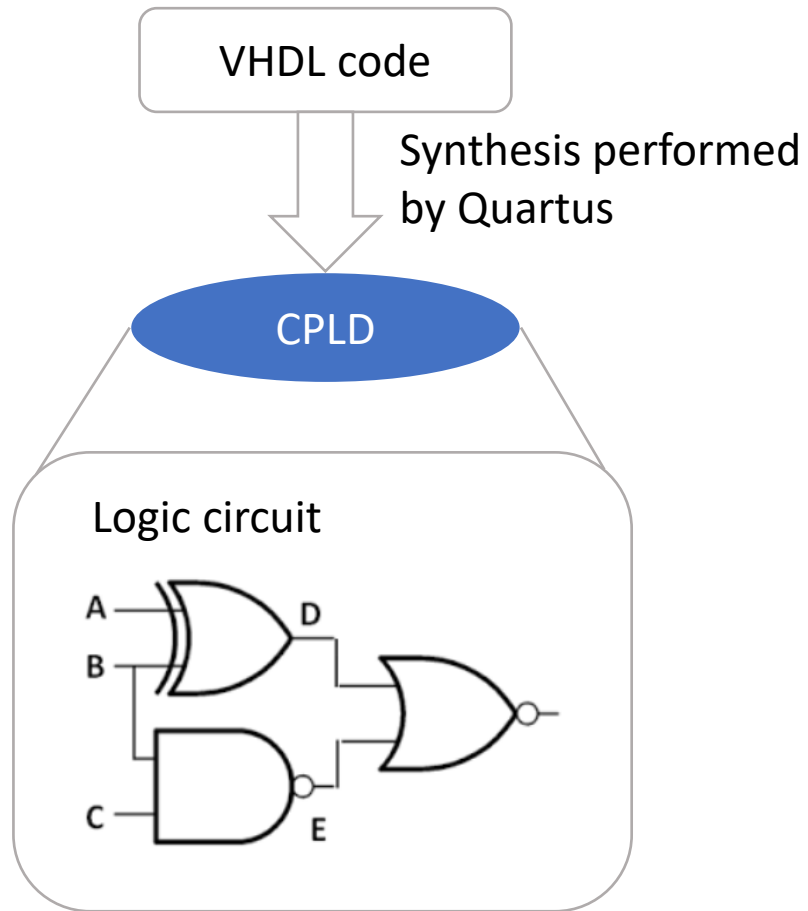
Hardware Description Language

Section 1: What is a CPLD?

A CPLD is a **programmable logic circuit**, i.e., an integrated logic circuit that can be reprogrammed as many time as you want!



A processor is a **central processing unit** that executes the machine instructions of computer programs.



How do programmable logic devices work?

Let's suppose that we would like to build a logic circuit that make a Boolean function:

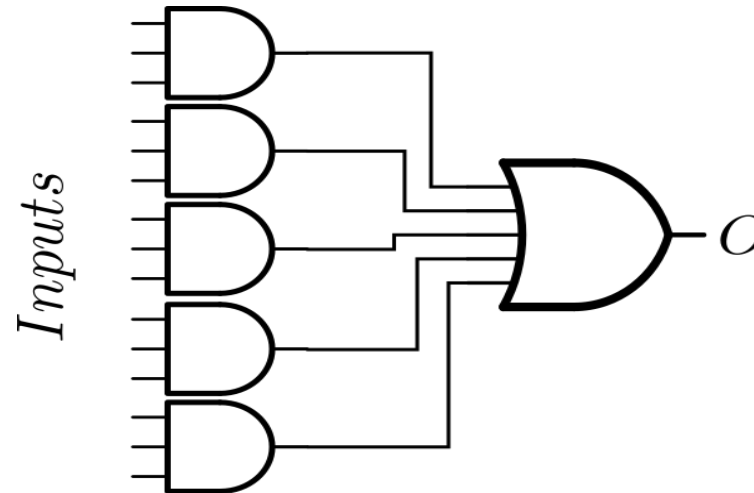
$$O = f(I_1, I_2, \dots)$$

This function can be written as a sum of midterms and can be simplified to a smaller sum of products equation (using Karnaugh map for instance)

$$O = \text{product}_1 + \text{product}_2 + \dots$$

In this Boolean function, the products are the **AND** operation between some inputs

The resulting logic circuit is therefore:

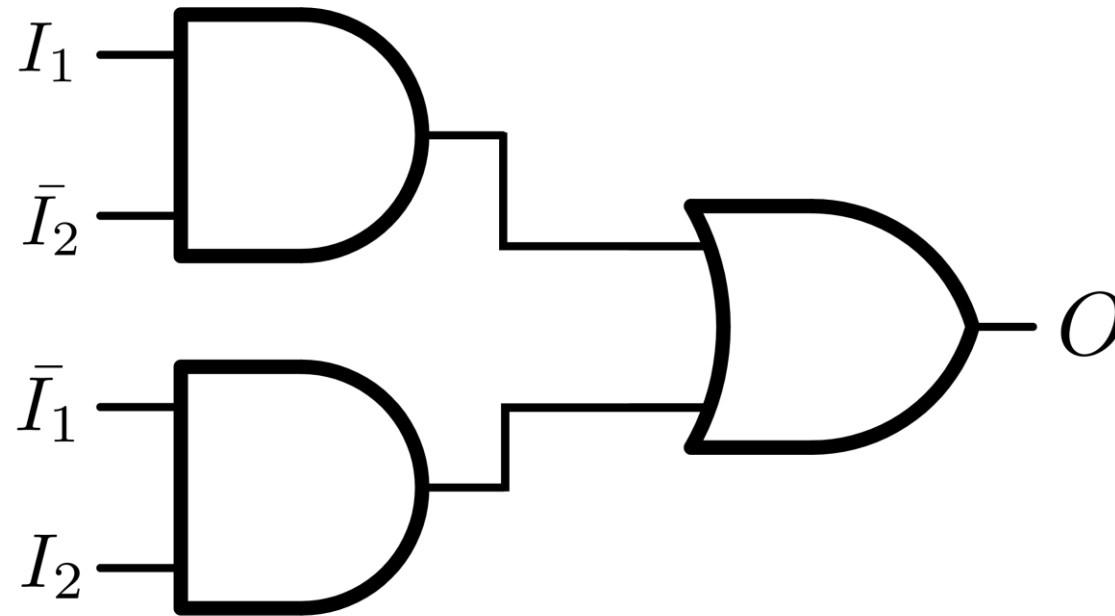


Example: XOR function

The **XOR** function can be written as:

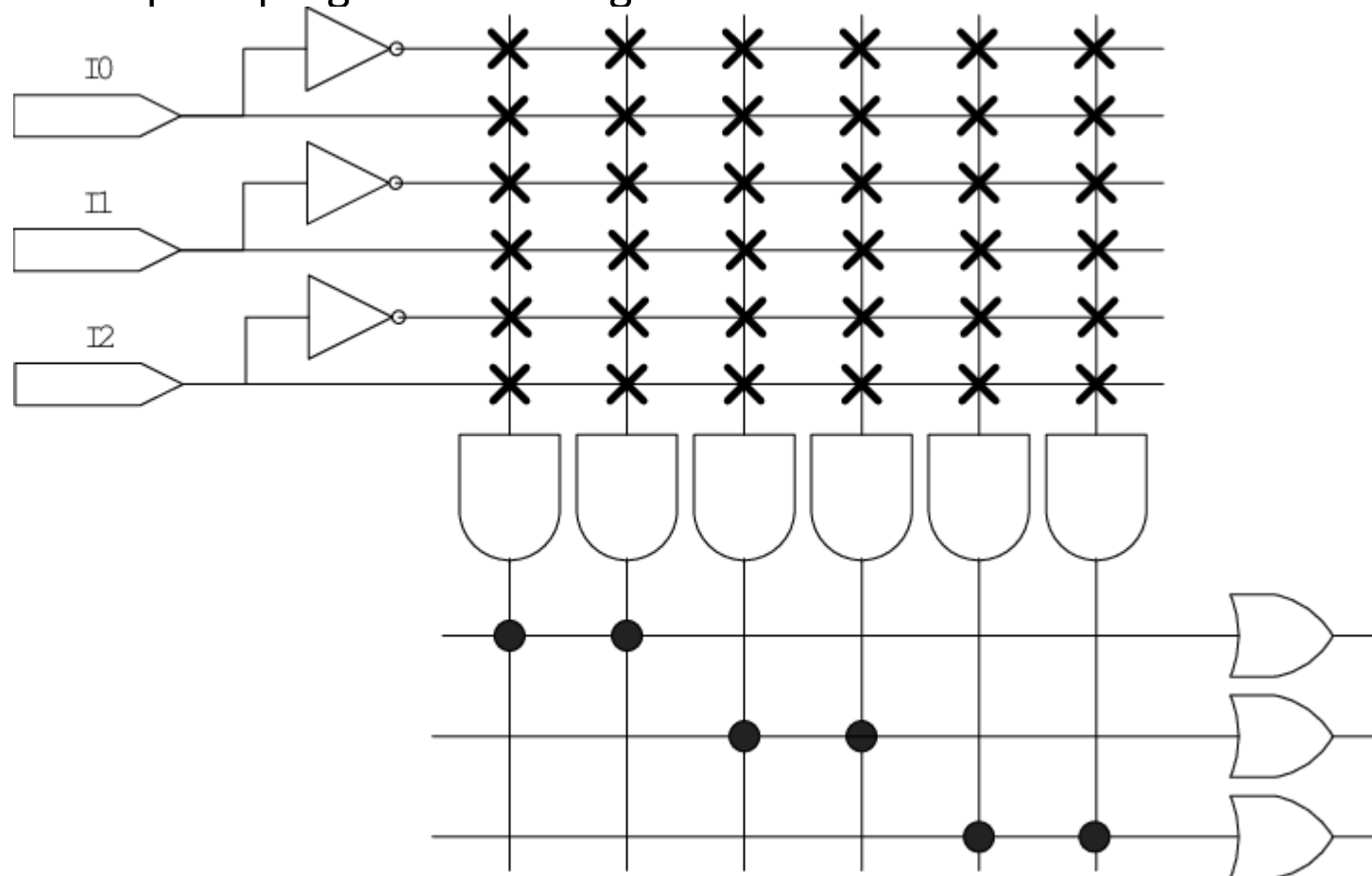
$$O = I_1\bar{I}_2 + \bar{I}_1I_2$$

The resulting logic circuit is therefore:



First basic block: Programmable Array Logic (PAL)

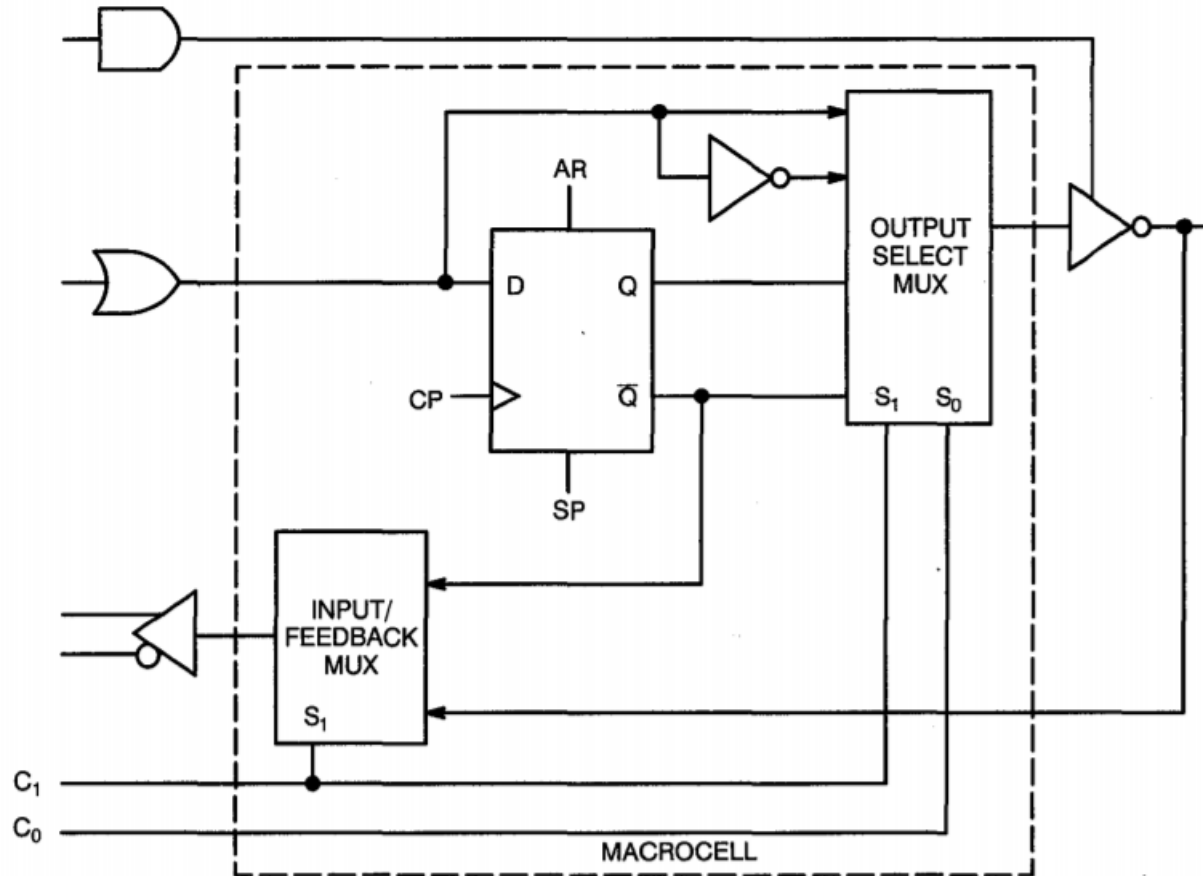
PALs are the simplest programmable logic devices



PALs can be reconfigured to create a very large number of [combinational logic circuits](#)

Second basic block: the macrocell

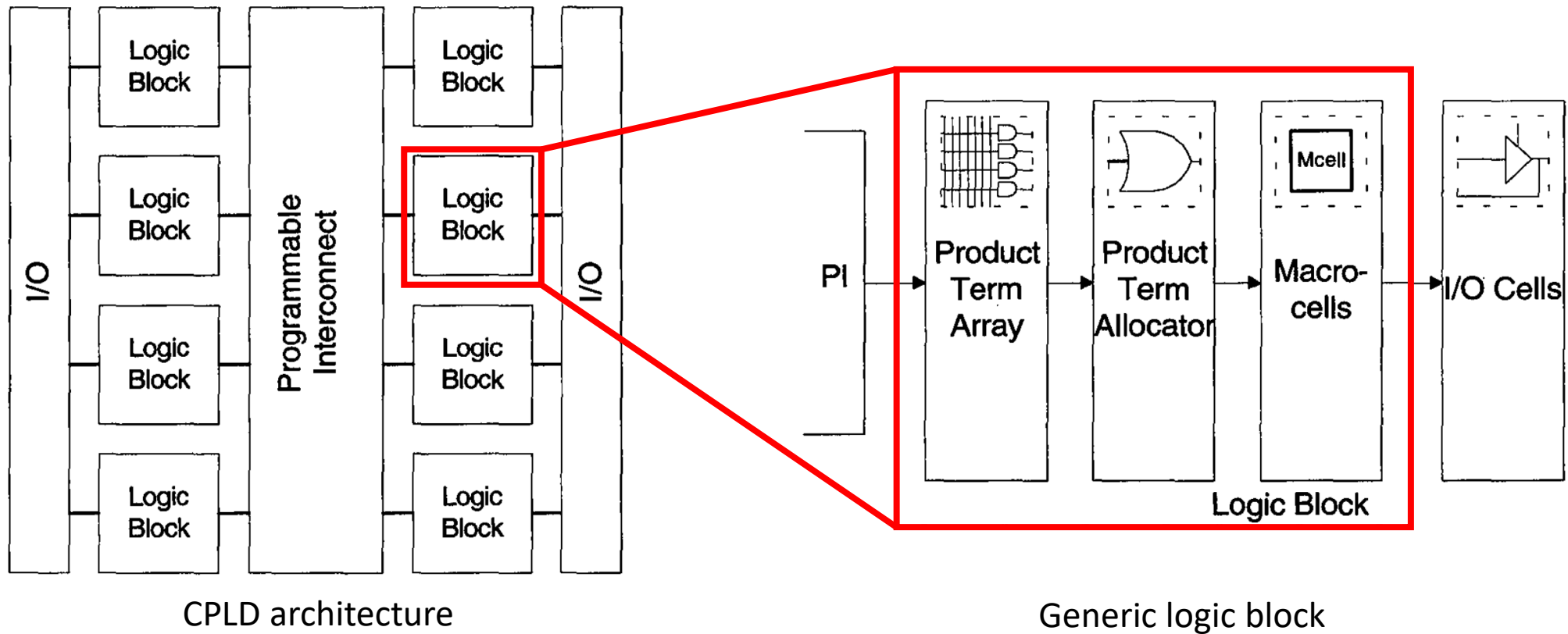
A macrocell can be added to the outputs of a PAL to configure [sequential logic circuits](#)



C_1 and C_0 determine the active mode of the macrocell

Coming to the CPLD

- CPLD = **Complex Programmable Logic Device**
- Composed of a set of **PLDs (PALs + macrocells)** connected to each other through a programmable network



Pros and cons of programmable logic circuits



Pros

- ☐ Parallel processing: several tasks can be performed at the same time (useful for data conversion, image filtering, ...)
- ☐ High frequency for critical control application
- ☐ Low size and power consumption

Cons

- ☐ Limitation of the application complexity due to the limited number of logic blocks



Section 2: VHDL introduction and bases

Since the hardware is completely different from a processor, an adapted programming language is required to program the CPLD

VHDL is a **hardware description language** designed to represent the **behavior** and the **architecture** of a digital electronic system (Verilog is another language mostly used in USA)

VHDL stands for **V**ery-high speed integrated circuit **H**ardware **D**escription **L**anguage

A specific **software** is needed to code in VHDL/Verilog and to program the CPLD:

- **Altera Quartus** (only for Windows and Linux)
- Xilinx ISE
- Lattice ISP Lever
- Altium Designer
- And many others

Software tools

These specific software contain different useful tools:

- Text Editor (to code)
- Simulator (to verify your design)
- Analyzer (responsible for transforming the VHDL code into basic logic elements)
- Place & Route (responsible for the placement and connection of logical elements in the component)
- Timing analyzer (to check if all timing requirements are met)
- A programmer (to program your circuit into the CPLD)
- And many others

Important note: during the simulations, the code will be executed sequentially by the processor of your computer, i.e., instruction by instruction. This is not what is happening when programming the code into the CPLD!

Entity/architecture pair and libraries

Any VHDL program have to be composed of at least one inseparable **entity/architecture** pair

- The **entity** describes the **Inputs/Outputs** of the component
- The **architecture** describes the **internal operation** of the component

Some **libraries** are available to provide a collection of related data types, functions, ...

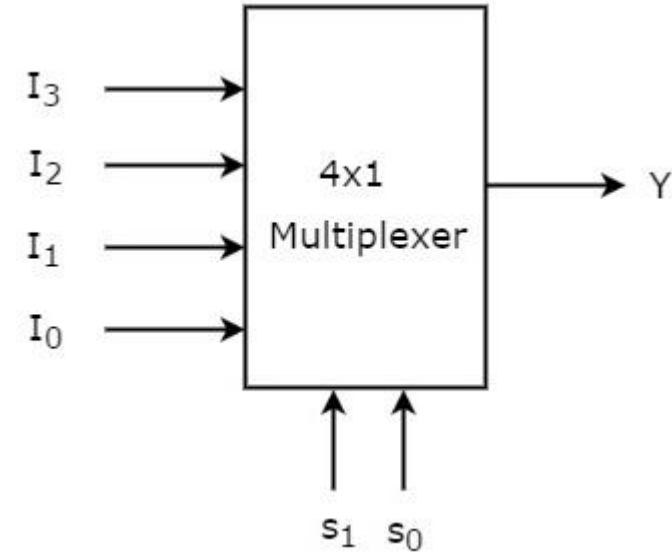
```
library ieee;  
use ieee.std_logic_1164.all;           -- type definition: bit, std_logic, ...  
use ieee.numeric_std.all;             -- signed and unsigned operations
```

The entity

An entity is a specification of the **design's external interface**

```
entity mux41 is port(  
    I0, I1, I2, I3: in std_logic;  
    S0, S1: in std_logic;  
    Y: out std_logic);  
end entity mux41;
```

- Declaration of the **entity mux41**
- Declaration of the **Inputs/Outputs**
- Declaration of the **I/O types** (in, out, inout, buffer)



The architecture

An architecture is a specification of the **design's internal implementation**

VHDL describes a logic circuit. Therefore, all statements are performed in parallel (“at the same time”). The order of the instructions has no influence on the result

3 different types of statements to build an architecture:

- Assignment statement
- Structural description
- Process statement

Assignment statements

First example: Boolean equations

- Architecture of the multiplexer 4 to 1

```
architecture mux41_arch of mux_41 is
begin
    Y <= ((not S0) and (not S1) and I0) or
          (S0 and (not S1) and I1) or
          ((not S0) and S1 and I2) or
          (S0 and S1 and I3);
end architecture mux41_arch;
```

The assignment statement defines directly the signal's value thanks to a logic or mathematical expression

Assignment statements

Second example: when-else condition

- Architecture of the multiplexer 4 to 1

```
architecture mux41_arch of mux_41 is
begin
    Y <= I0 when (S0='0' and S1='0') else
          I1 when (S0='1' and S1='0') else
          I2 when (S0='0' and S1='1') else
          I3
end architecture mux41_arch;
```

A lot of other possible declarations:

- Case-when condition
- With-select-then condition
- ...

Structural description

This type of architecture uses existing elements (in another file or library) and connects them together. This is the most equivalent to a main function in classical programming.

Example: using 2 bits adder to create a 3 bits adder

*1st: Create the 2 bits adder
entity/architecture pair*

```
entity add_2bits is port(  
    I1, I2 : in std_logic;  
    O1, O2 : out std_logic);  
end add_2bits;  
  
architecture arch2 of add_2bits is  
begin  
    O2 <= I1 xor I2;  
    O1 <= I1 and I2;  
end arch2;
```

*2nd: Use this 2 bits adder to create the
3 bits adder entity/architecture pair*

```
entity add_3bits is port(  
    A, B, C : in std_logic;  
    R, S : out std_logic);  
end add_3bits;  
  
architecture arch3 of add_3bits is  
    component add_2bits port(  
        I1, I2 : in std_logic;  
        O1, O2 : out std_logic);  
    end component add_2bits;  
  
    signal S1, S2, S3 : std_logic;  
begin  
    add1 : add_2bits port map(A, B, S1, S2);  
    add2 : add_2bits port map(S2, C, S3, S);  
    R <= S3 + S1;  
end arch3;
```

Process statement

A process statement describes the process functioning via a set of **sequentially interpreted instructions**

```
architecture mux41_arch of mux_41 is
begin
  main: process(I0, I1, I2, I3, S0, S1)
  begin
    O <= I0;
    if (S0 = '1' and S1 = '0') then
      O <= I1;
    elsif (S0 = '0' and S1 = '1') then
      O <= I2;
    elsif (S0 = '1' and S1 = '1') then
      O <= I3;
    end if
  end process main;
end architecture mux41_arch;
```

- Declaration of the **process main**
- The **sensitivity list** is a compact way of specifying the set of signal events on which may resume a process: it is a list of signals for which a change in value on one or more of these signals causes the process to be activated
- The process declaration allows to implement more complex systems in an easier way
- Although the instructions are interpreted sequentially, the program will analyze the process to transform it into a set of instructions that will run in parallel

The signals

Before describing other architecture types, let's define a way to store intermediate values: [the signals](#)

- For non-trivial applications, signals are often indispensable. They are used as intermediate values in order to implement more complex circuits
- It's the closest equivalent to variables in classical computer programming
- Modeling of an interconnection (or a group of interconnections)

Variables vs Signals

Another way to store intermediate values: [the variables](#)

```
entity binary_to_dec is port(
  I : in bit_vector( 7 downto 0 );
  O : out integer range 0 to 255);
end binary_to_dec;

architecture arch of binary_to_dec is
begin
  main: process( I )
    variable tmp1, tmp2 : integer range 0 to 255
  begin
    tmp1 := 1;
    tmp2 := 0;
    for i in 0 to 7 loop
      if I(i) = '1' then tmp2 := tmp2 + tmp1;
      tmp1 := tmp1*2;
    end if;
    end loop;
    O <= tmp2;
  end process;
end architecture arch;
```

- In a process, the value assignment of a value to a [signal](#) “occurs at the end of the process”. In other words, the assignment has no effect on further execution of the process in progress
- On the other hand, the assignment of a value to a [variable](#) is directly effective and accessible in the rest of the process

Summary: global architecture syntax

```
architecture architecture_name of entity_name is
    --Type declarations
    --Signal declarations
    --Constant declarations
    --Component declarations
begin
    --Assignment Statements and/or
    --Structural Descriptions and/or
    --Process Statements
end architecture architecture_name;
```

Important note: the statements order has no influence. All statements operate in parallel (called concurrent statements)

Syntax bases: identifier

Identifier: name of signals, variables, processes, entities, ...

- Alphabetic characters, numbers or underscores
- Must start with an alphabetical character
- Cannot end with an underscore
- Cannot contain two successive underscores
- Case insensitive

```
entity binary_to_dec is port(  
  I : in bit_vector( 7 downto 0 );  
  O : out integer range 0 to 255);  
end binary_to_dec;  
  
architecture arch of binary_to_dec is  
begin  
  main: process( I )  
    variable tmp1, tmp2 : integer range 0 to 255  
  begin  
    tmp1 := 1;  
    tmp2 := 0;  
    for i in 0 to 7 loop  
      if I(i) = '1' then tmp2 := tmp2 + tmp1;  
      tmp1 := tmp1*2;  
    end if;  
  end loop;  
  O <= tmp2;  
end process;  
end architecture arch;
```

Syntax bases: classes

- 3 different classes: `signals`, `variables` and `constants`
- Used in architecture implementation

General syntax:

```
class identifier: type [:= initial value];
```

Examples:

```
constant number_players: integer := 4;  
signal done: bit := '0';  
variable index_loop: integer range 0 to 127 := 0;
```


Syntax bases: predefined types

Basic types that are already defined:

- `type boolean is (false, true);`
- `type bit is ('0', '1');`
- `type std_logic is ('0', --logic 0
'1', --logic 1
'L', --weak signal that should probably go to 0
'H', --weak signal that should probably go to 1
'Z', --High Impedance
'U', --Uninitialized, this signal hasn't been set yet
'X', --Unknown, impossible to determine the value
'W') --weak signal, can't tell if it should be 0 or 1`
- `type integer, signed, unsigned, ...`

Don't forget the libraries that must be included to use specific predefined types

```
library ieee;  
use ieee.std_logic_1164.all;           -- type definition: bit, std_logic, ...
```

Syntax bases: types and subtypes

- Define a type or subtype to use it in the architecture implementation for any classes

General syntax:

```
type/subtype identifier is type/subtype_definition [range_definition];
```

Examples:

```
type numbers is range 0 to 4;  
type voltage is -3.3 to 3.3;  
subtype bit_index is integer range 0 to 31;
```

Pros: limit constants, variables or signals to useful values (better resource utilization) and improve readability of your code

Syntax bases: enumerations

- Very useful to design signals/variables with finite discrete states (memory optimization)

General syntax:

```
type identifier is (name1, name2, name3, ...);
```

Examples:

```
type states is (idle, preamble, data, jam, nosfd, error);  
signal current_state : states := idle;  
type days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);  
signal today : days;
```

Syntax bases: array type

- Useful to store several values into one identifier

General syntax:

```
type identifier is array(first_index to last_index) of elements;
```

Examples:

```
type code_lock is array (0 to 3) of integer range 0 to 9;  
constant my_code : code_lock := (4, 1, 8, 3);
```

`bit_vector` and `std_logic_vector` are already available in the basic libraries:

```
signal vector : std_logic_vector (2 downto 0) := "011";
```

- In vector, `vector(2) = '0'` and `vector(1) = vector(0) = '1'`

Syntax bases: multidimensional array type

Examples:

```
type table_16x8 is array (0 to 15, 0 to 7) of bit;  
signal my_table : table_16x8 := ("01100110", "10101010", ...);
```

Other way, array of vectors:

```
type square_matrix is array (0 to 7) of std_logic_vector(0 to 7);
```

Syntax bases: loop

```
[loop_label: ] loop
  --sequence of statements;
  --exit when condition;
  --[when when condition;]
end loop [loop_label];
```

```
[loop_label: ] while (condition) loop
  --sequence of statements;
  --[exit when condition;]
  --[when when condition;]
end loop [loop_label];
```

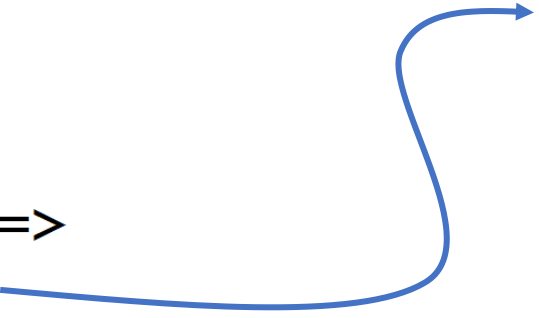
```
[loop_label: ] for identifier in discrete_range loop
  --sequence of statements;
  --[exit when condition;]
  --[when when condition;]
end loop [loop_label];
```

- The 3 different loops can be equivalent, but a judicious choice of the type of loop allows more concise and readable code
- **exit** allows to exit the loop
- **next** allows to go to the next iteration without executing the following instructions

Syntax bases: case-when statement

```
case S is
  when "00" =>
    O <= I0;
  when "01" =>
    O <= I1;
  when "10" =>
    O <= I2;
  when "11" =>
    O <= I3;
  when others =>
    O <= '-';
end case;
```

This just means that
we don't care about
the value of **O**



Section 3: Sequential and synchronous logic

- **Sequential logic:** logic circuit whose output depends not only on the present value of its input signals but on the **sequence** of past inputs
- **Synchronous logic:** logic circuit in which the changes in the state of memory elements are **synchronized** by a clock signal

Both requires memorization in the logic circuit. The only statement type that is able to implement memory is the **process statement**

D Flip-Flop example

```
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is port(
    d, clk : in std_logic;
    q : out std_logic);
end entity d_ff;

architecture d_ff_arch of d_ff is
begin
    process( clk )
    begin
        if rising_edge( clk ) then
            q <= d;
        end if;
    end process;
end architecture d_ff_arch;
```

- Synchronized circuit
- The **sensitivity list** is a list of signals. A change in value on one or more of these signals causes the process to be activated
- One process execution by clock cycle: **rising edge detection** to the clock signal
- No **else** in the condition statement as the Flip Flop must keep its value in memory even when the clock goes down
- Moreover, most simulators do not support an **else** after an **if rising_edge**. Indeed, this construction would be ambiguous (the event lasts only a short moment)

Latch transparent

```
library ieee;
use ieee.std_logic_1164.all;

entity latch is port(
    d, clk : in std_logic;
    q : out std_logic);
end entity latch;

architecture latch_arch of latch is
begin
    process( clk )
    begin
        if clk = '1' then
            q <= d;
        end if;
    end process;
end architecture latch_arch;
```

- Condition on **clk** directly, no more rising edge detection
- The circuit is fully transparent when the clock is high

FF with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;

entity ff_reset is port(
    d, clk, rst : in std_logic;
    q : out std_logic);
end entity ff_reset;

architecture ff_reset_arch of ff_reset is
begin
    process( clk, rst )
    begin
        if rst = '1' then
            q <= '0';
        elsif rising_edge( clk ) then
            q <= d;
        end if;
    end process;
end architecture ff_reset_arch;
```

- **rst** in the sensitivity list
- For a synchronous reset, another **if** condition in the rising edge and remove **rst** from the sensitivity list

Section 4: State machine and simulation

Until now, the design of a state machine required the following steps:

- Defining a state diagram
- Transformation of states into binary codes
- Karnaugh maps
- Determination of Boolean equations

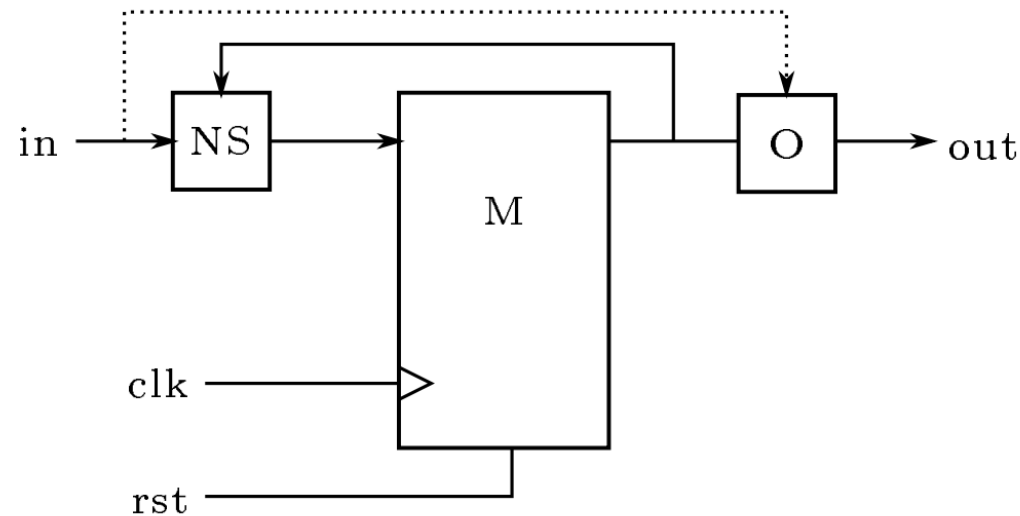
In VHDL, things are much simpler since some of the previous steps are done by Quartus

- Defining a state diagram
- Implementation

Implementation of a state machine

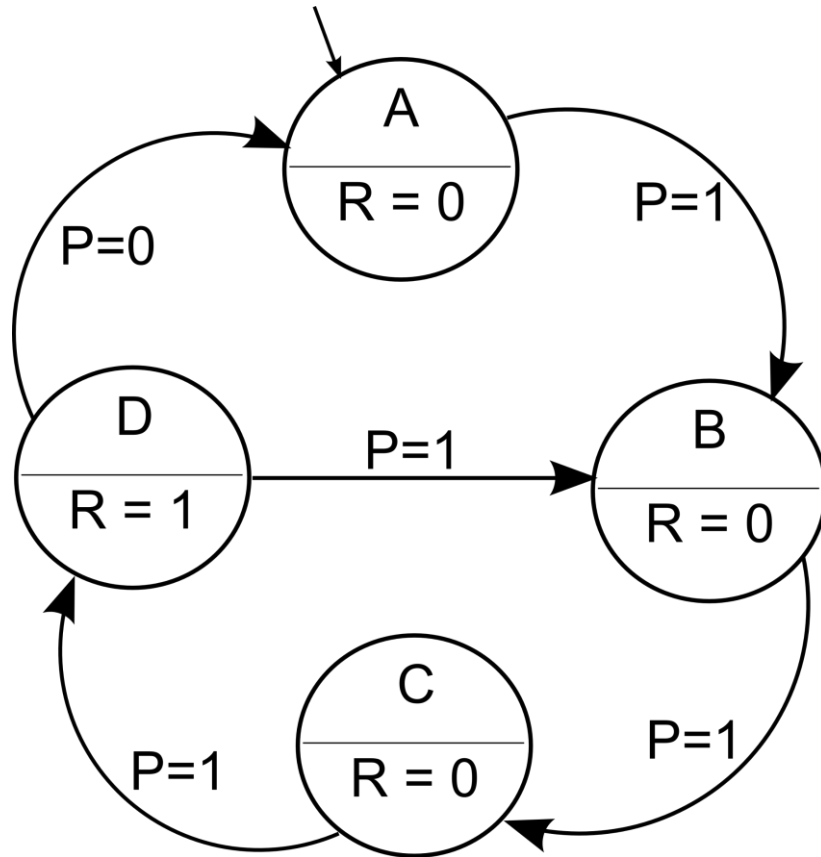
The implementation of a state machine is composed in 3 parts:

- A function to compute the next state
- A function to compute the outputs
- A function to store the current state



These 3 functions can be implemented in the same process or in different statements, in order to increase readability of the code.

Implementation of a state machine: example



- States: **A, B, C, D**
- Input: **P**
- Output: **R**

```
entity state_machine is port(  
    P : in std_logic;  
    R : out std_logic);  
end entity state_machine;
```

```

architecture arch of state_machine is
  type states is (A, B, C, D);
  signal state : state := A;
begin

```

```

  -- Update state function
  update_state: process( clk )
  begin
    if rising_edge( clk ) then
      case state is
        when A =>
          if P = '1' then
            state <= B;
          end if;

        when B =>
          if P = '1' then
            state <= C;
          end if;

        when C =>
          if P = '1' then
            state <= D;
          end if;

        when D =>
          if P = '1' then
            state <= B;
          else
            state <= A;
          end if;
      end case;
    end if;
  end process update_state;

```

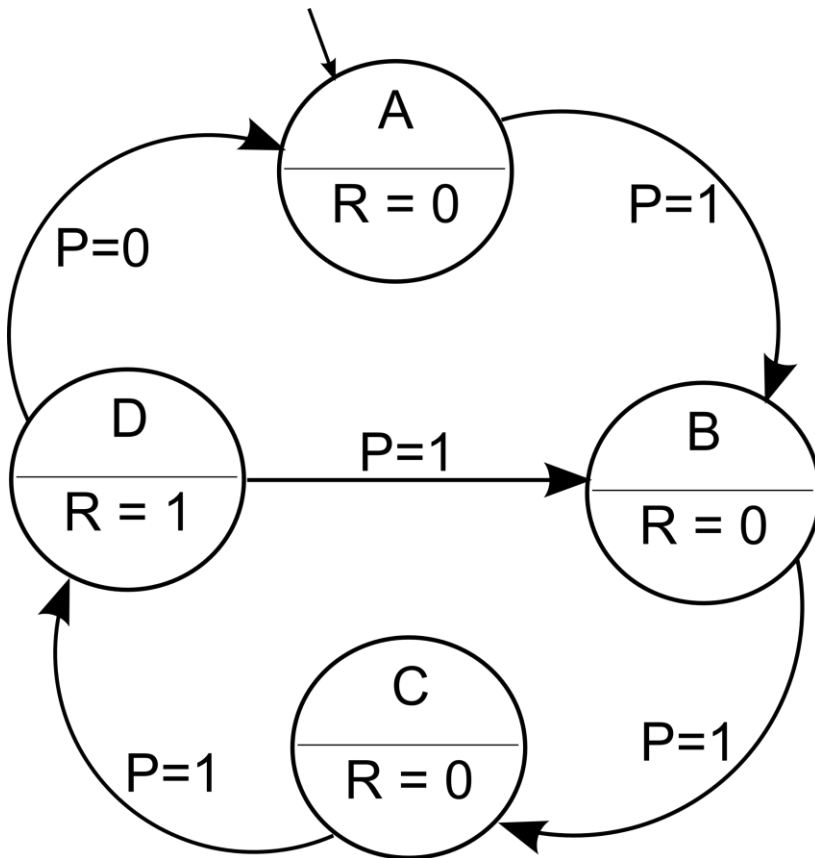
Function to compute
the next state

```

  -- Output function
  R <= '1' when (state = D) else '0';
end architecture arch;

```

Function to compute
the output



Simulation in Quartus

A simulator of code is available with Quartus called [Modelsim](#). As the simulator runs on a processor, the code is analyzed and executed sequentially. The simulation works as follows:

- Defining the simulation duration
- Defining the evolution of inputs (testbench)
- When an input changes at time t^* , the values of the signals and outputs are evaluated
- The modification of the signals and outputs takes place at time $t^* + \delta$ (to take into account the gate delay)

→ modifications are never executed immediately, but are executed after the complete code analysis

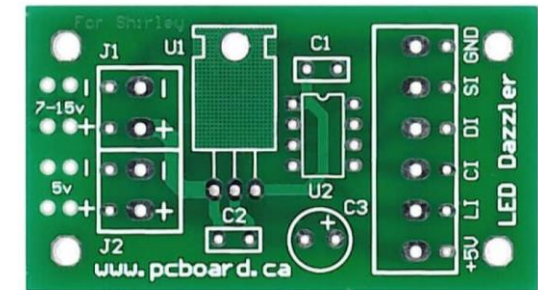
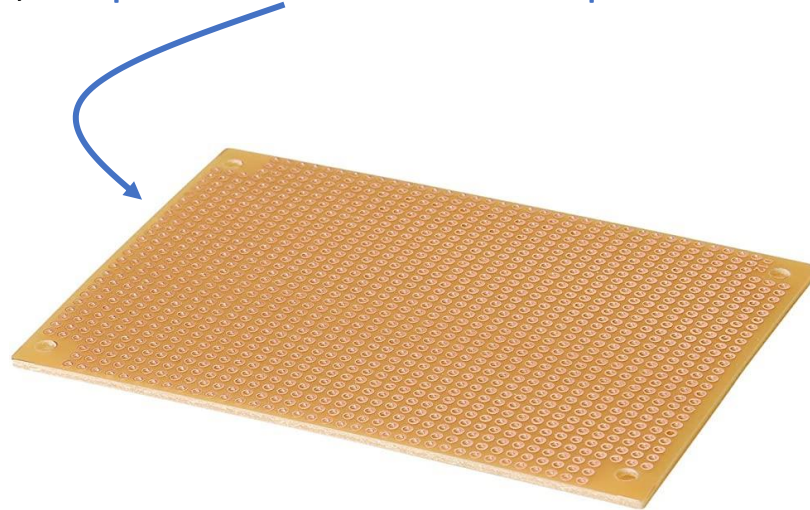
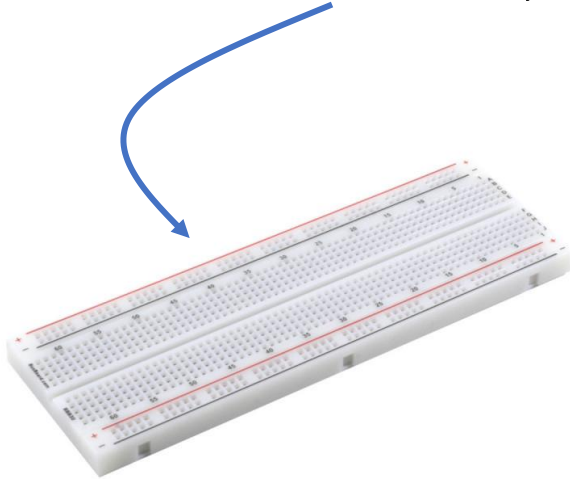
During synthesis, on the other hand, everything is executed in parallel. Therefore, there can be a difference between synthesis and simulation

Section 5: Project introduction

The objectives of this project is to familiarize you with:

- A [programmable logic device](#) (CPLD) and a [hardware description language](#) (VHDL)
- The creation of a simple [electronic](#) circuit

→ The project consist in the realization of a functional system, either on a [breadboard](#) or, better, on [pre-drilled board](#) or a [printed circuit board](#) (PCB)



Deadlines

The final deadline will be on the [submission platform](#)! Please, subscribe to the course and create your group of 4 ( the same as the one for the labs!) as soon as possible (first thing to do)!

March 12th: brief description in **ENGLISH** of the project to assess feasibility in a *pdf* report (refer your group number) sent by email to me ([Arthur Fyon](#)) with:

- Project description (5 lines is enough)
- Description of the code structure
- Hardware required in addition to the development board

May 1st: final project report in **ENGLISH** with:

- Description of your code structure (block diagram, pseudo-code, ...)
- Description of inputs, outputs, important signals, ...
- Electronic schematic
- Any other information you deem necessary (simulations, hardware, ...)

One *zip* file per group with the report and VHDL codes with the name:

ELEN0040_group_number.zip

Live demonstration

Your project will also be assessed through a 20-minutes presentation (at maximum) that will take place before the exam session (the exact schedule will come later)

You will be asked questions about:

- The code (structure, signals, processes, ...)
- The hardware and the electrical circuit
- The report (explanations, clarifications, ...)

No questions about anything else than your project!

How to start the project?

- Think about the **required hardware** and the **number of input/output** needed (e.g., Four-in-a-Row: a lot of LEDs and therefore a lot of outputs)
- Think about the **different tasks** you need to implement
- Think about the **structure of your code**:
 - Concurrent statements/processes?
 - State machine?
- **Start with something simple** that you can test and gradually increase the complexity of your code:
 - More difficult to debug than a classical language running in a processor
 - *Advice*: simple test with a button and a LED

Keep in mind that you are not programming, you are describing a logic circuit!

Example: Simon game

Hardware:

- 4 color LEDs
 - 4 buttons (or more? Reset?)
 - 2 7-segment displays?
- Number of pins required?



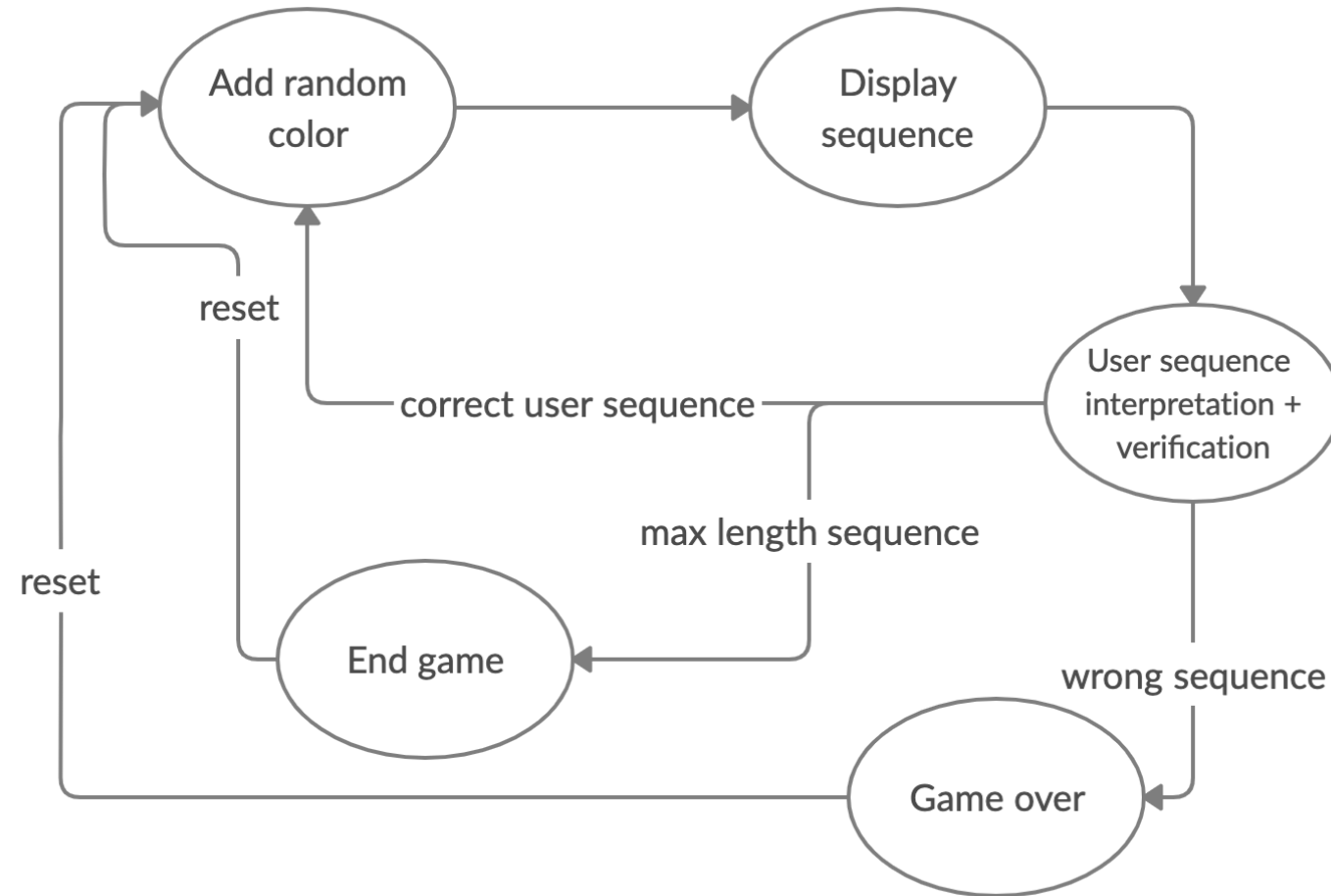
Tasks:

- Random number generator (RNG)
- Display the color sequence
- Interpret user commands
- Check if the user sequence is correct
- Display the score?

Code structure:

- As the game takes place in different phases, a state machine is well adapted
- State machine diagram !

Example: Simon game



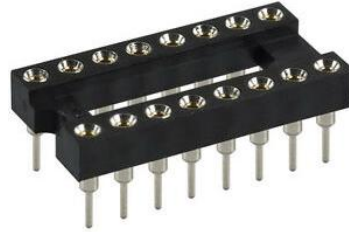
→ Complete design and implementation in the pdf file *ELEN0040_Simon.pdf*

Hardware: what can you have?

To build you circuit:

- Breadboard: simplest
- Stripboard: +1
- PCB: +3

Do no directly solder the development board with the CPLD on the stripboard or the PCB, use tulip support!



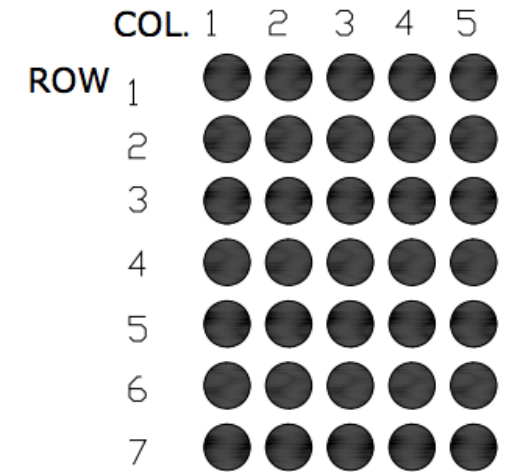
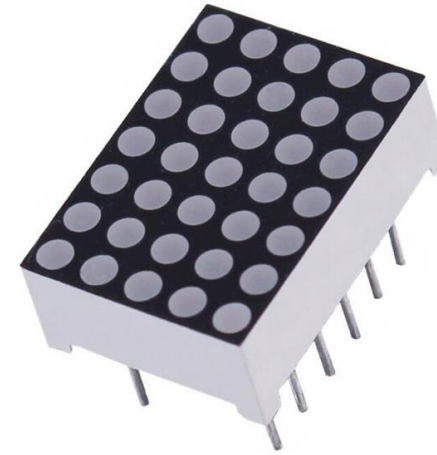
Other useful components:

- LEDs
- 7-segment displays
- Buttons
- Potentiometer (variable resistance)
- Decoder
- LED matrix



LED matrix

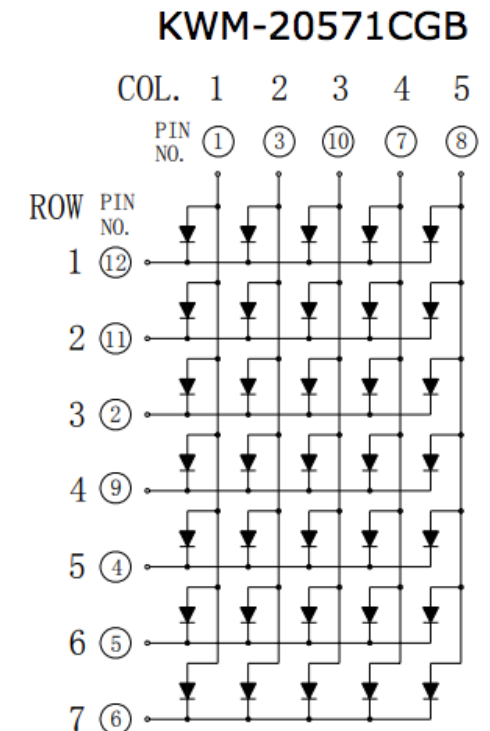
- $5 \times 7 = 35$ LEDs (exists also in 8×8)
- Cannot drive them independently \rightarrow too much pins required



LED matrix schematic:

- The anode of the same line are connected to each other
- The cathode of the same column are connected to each other
- 13 pins required ($\lll 35$)

The **trick** is to display pixels **row by row** (or **column by column**) at a speed that is too fast for the human eye \rightarrow perfect illusion



RNG: how to do it?

2 main options:

- Make a **LSFR**: generate a pseudo random sequence thanks to a binary equation
- Use an **unpredictable external signal** to generate a random number

For your project, the second option is often easier. For example, the **unpredictable time** when the user presses a button can be used as follows:

- A signal is changed on each rising edge of a clock signal (preferably fast). The signal takes all the possible random values
- When the user presses a button, the value of the signal is read and used as a random variable

General advice for the project

Keep calm and enjoy! This might be
your coolest project of Bachelor!

Contact me in case of questions (office hours will be organized
during 2 half days each week → schedule to come)

Reference

Nice book concerning VHDL for programmable logic ([link](#))

