ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[1 - Introduction & contexte]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

Introduction to the sequence

Main objectives

- Understanding the HDL concepts
- Mastering the VHDL language
- Understanding relation between description and real hardware

What we have

- 13 course sessions (2 hours)
- No fixed plan (CM, TD, TP)
- VHDL project with E. Casseau (20h)
- Next semester: RISC-V design

Module evaluation

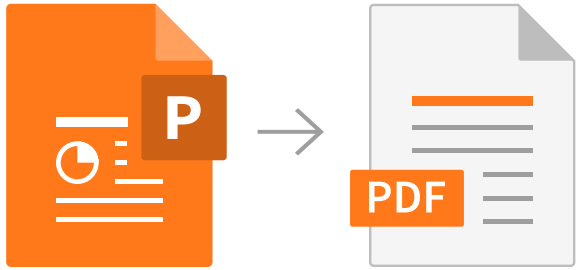
- The TP report will be evaluated
- The project report will be evaluated too (EC)

Cloning the GIT repository

GitHub

Clone the following repository

github.com/blegal/
ENSSAT-SNum2-VHDL-design-student

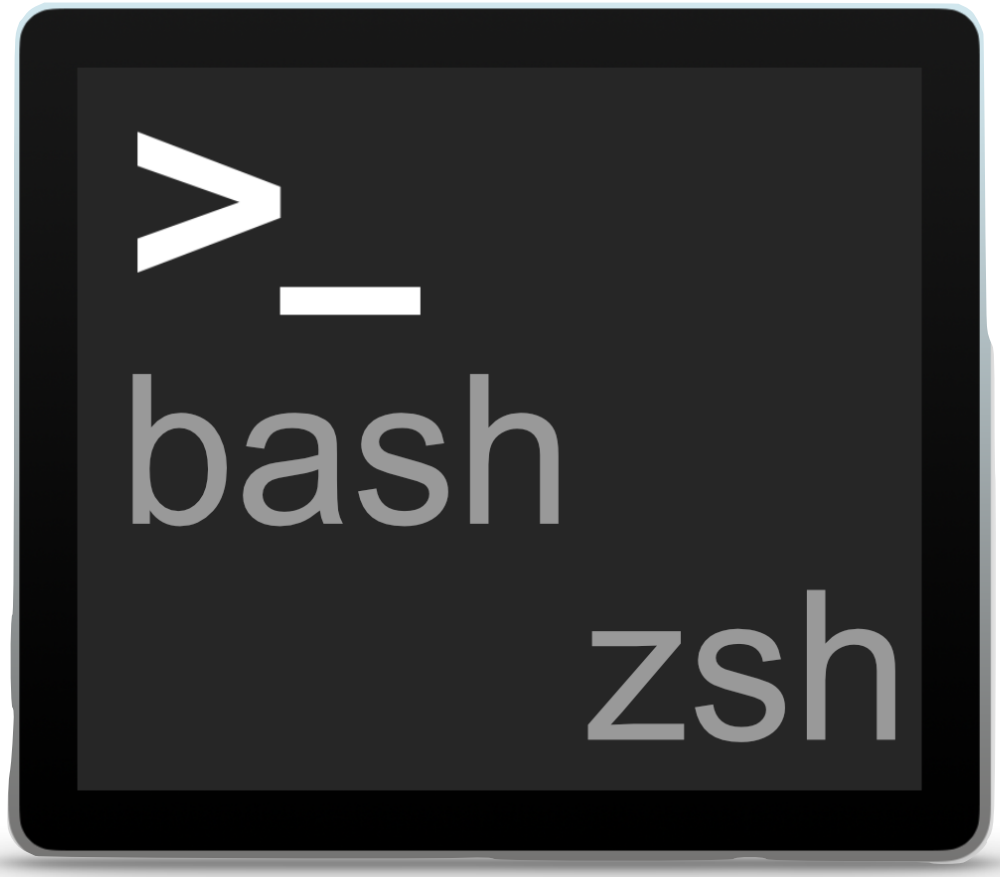


A PDF version of the slides

VHDL code exemples + scripts

Bash environment variables

- We need to update your env. variables
  - Add tool PATHs to ease their launching
  - Specifying tool license locations
  - Fixing some execution bugs
- This can be done in .bashrc file
  - cd ENSSAT-SNum2-VHDL-design-student
  - gedit ~/.bashrc
  - gedit bashrc
  - copy and paste the lines
- Close and restart your terminal

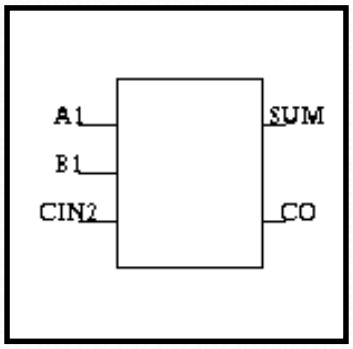


From algorithm to circuit

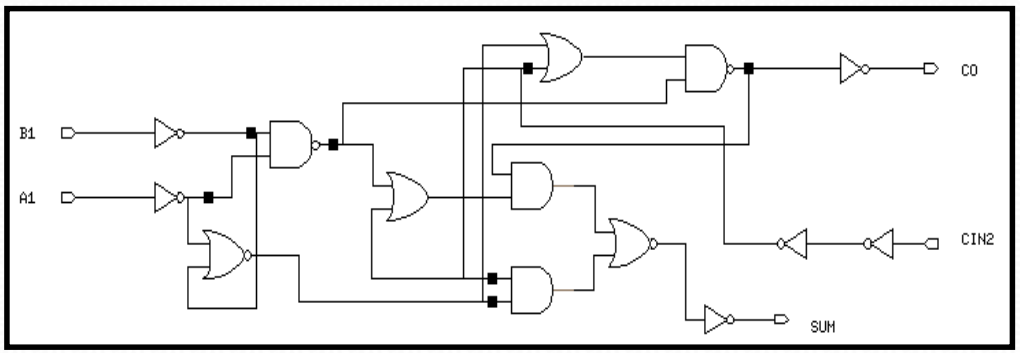
Register Transfer Level (RTL)

SUM :=

A1+B1

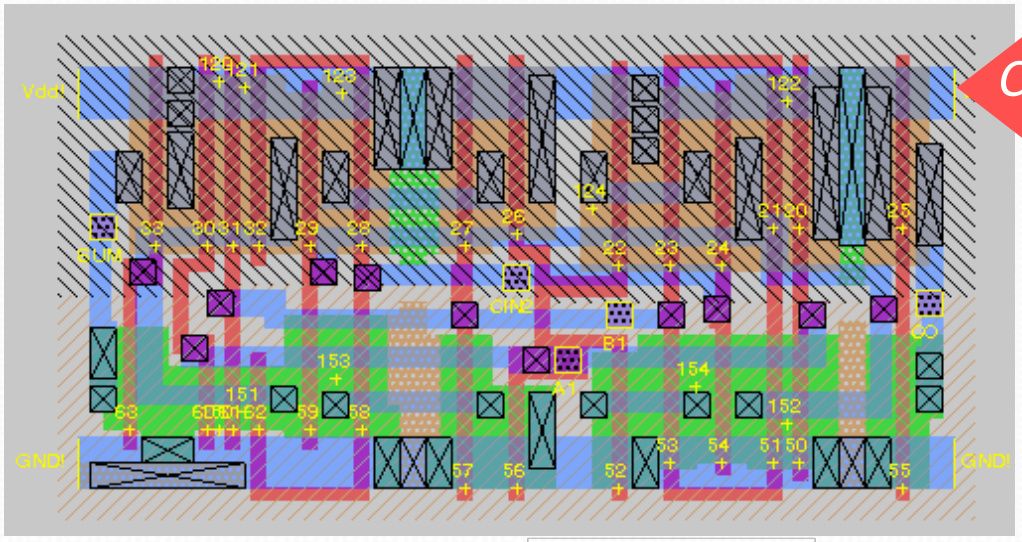


Gate Level

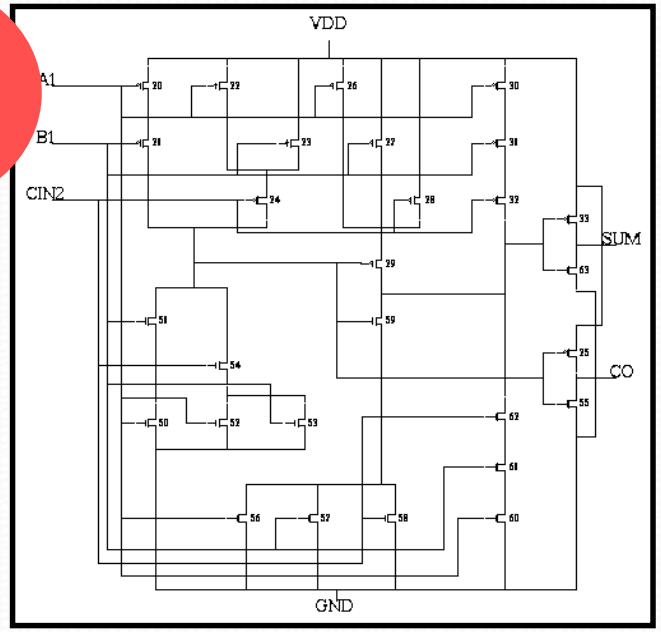


Algorithm

Circuit



Layout Level



Transistor or Circuit Level

Hardware versus Software flows

- Hardware
  - Layout-Level
  - Gate-Level (VHDL, Verilog)
  - Register-Transfer Level (VHDL, Verilog)
  - Algorithmic Level (C/C++, System Verilog)
  - System Level (e.g. UML, Matlab/Simulink)

- Software
  - Binary Code
  - Assembly Code
  - Machine Dependent Languages (e.g. C)
  - Virtual Machine (e.g. Java)
  - System Specifications (e.g. UML)

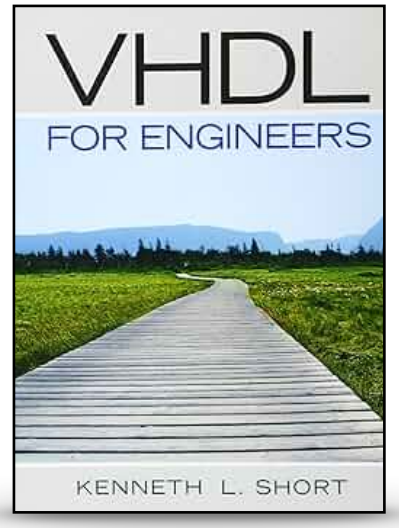
VHDL language

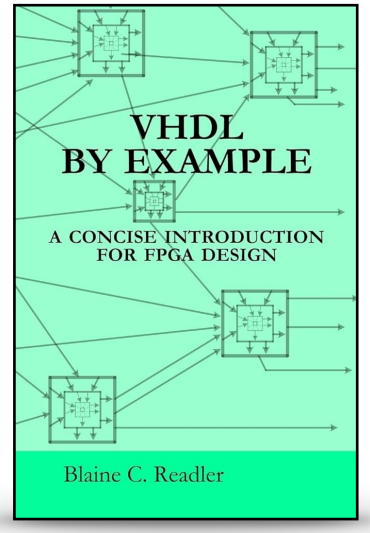
What is VHDL?

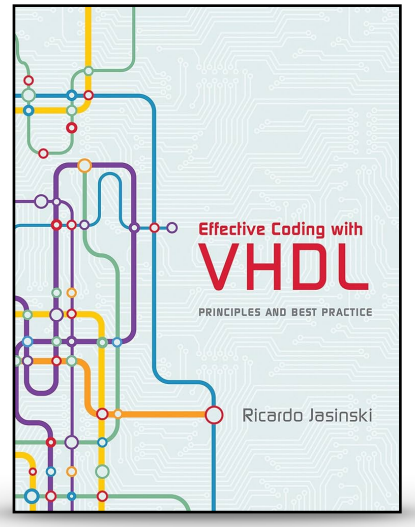
- VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

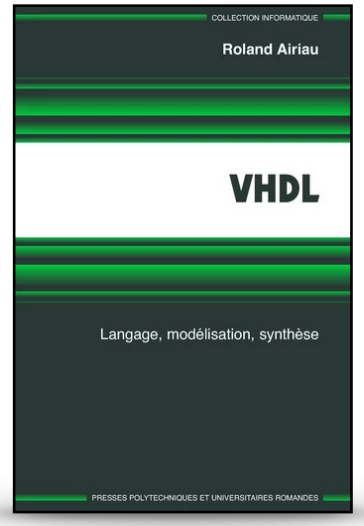
History

- Designed by IBM, Texas Instruments, and Intermetrics as part of the DoD funded VHSIC program
- Standardized by the IEEE in 1987: IEEE 1076-1987
- Enhanced version of the language: IEEE 1076-1993
- Additional standardized packages: IEEE 1164 (data types), IEEE 1076.3 (numeric), IEEE 1076.4 (timing)





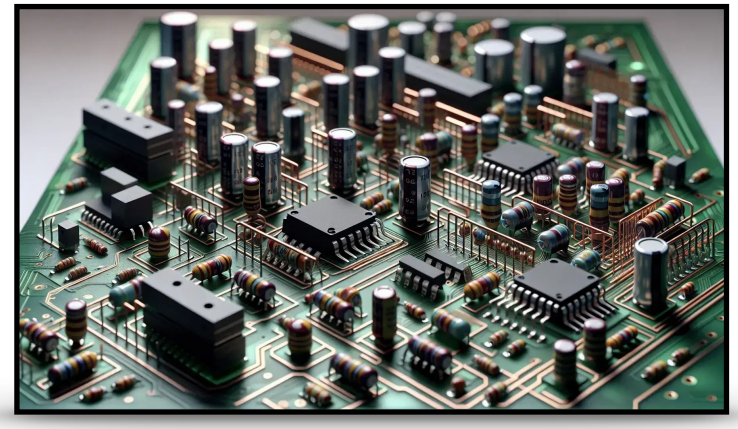




Prog. vs description language

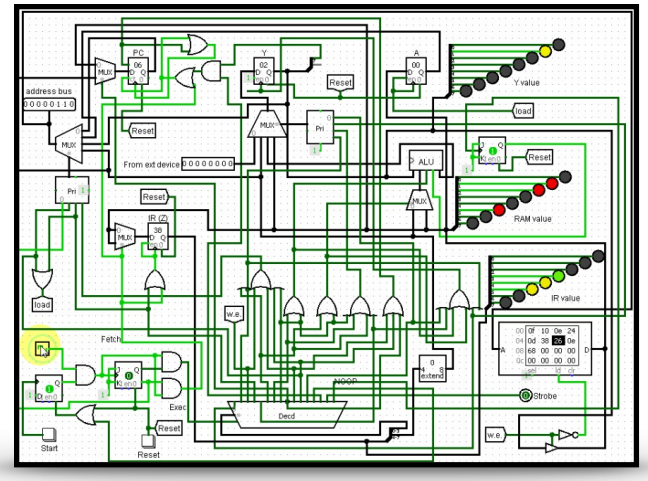
- Programming language provide the « *how* »
  - For computations
  - For data manipulation
  - For execution on a specific HW model

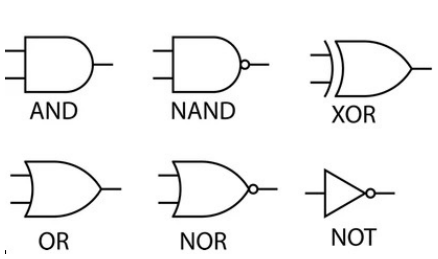
- HDL language is used to describe systems
  - System can be described from different point of views
  - Behavior (what does it do ?)
  - Structure (what is it composed of ?)
  - Functional properties (how to interface it ?
  - Physical properties (how fast is it ?)

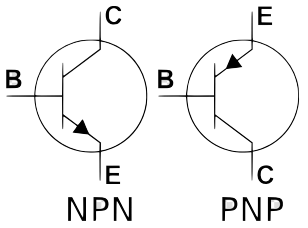


Interests of HDLs

- Descriptions can be done at different abstraction levels
  - Switch level (transistors)
  - Gate level (logical gates)
  - Register transfer level (RTL)
    - *combinational and sequential logic components*
  - Functional Instruction Set Architecture level (microprocessor)
- Behavioral descriptions are used for
  - System specification
  - Simulation
  - Hardware synthesis







ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[2 - Design Flow and Tools]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

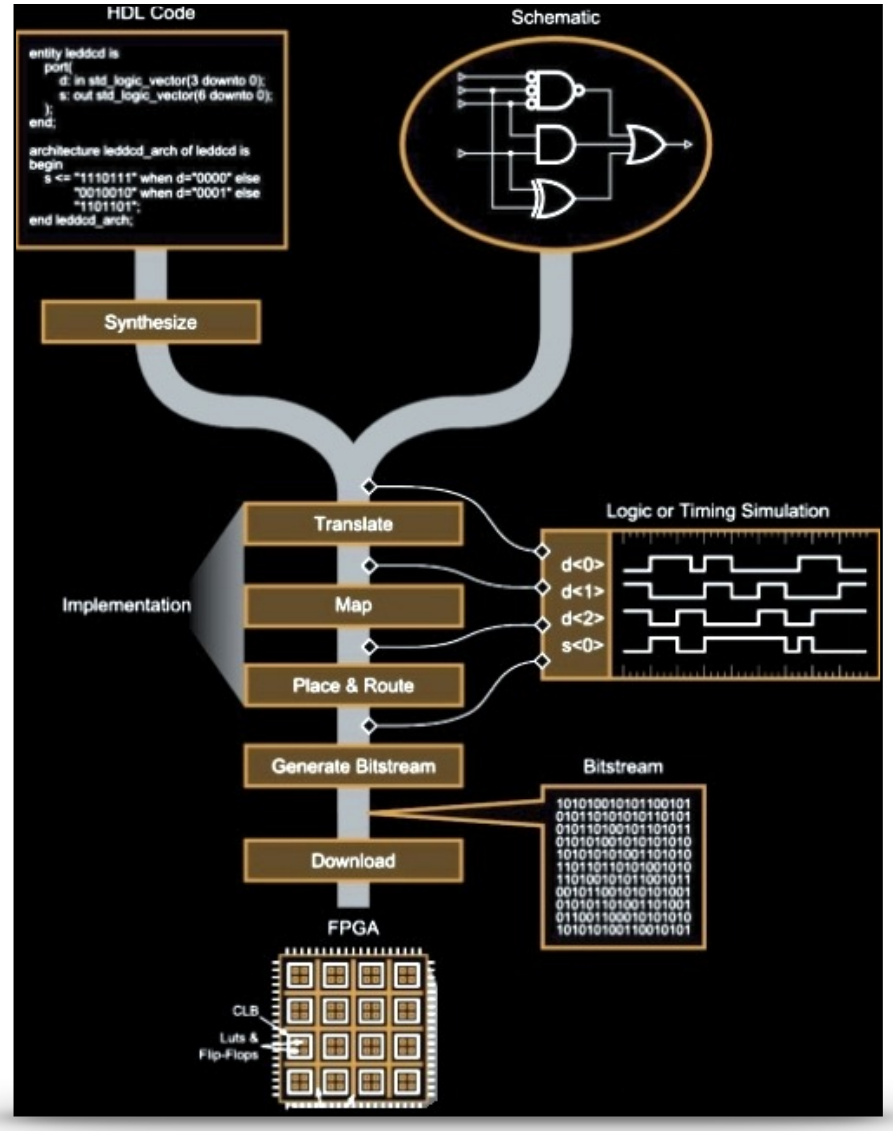
From RTL design to real hardware

Multiple stages

- Logical synthesis - transform the RTL hardware description to gate description
  - Karnaugh table optimization
- Physical synthesis - select the primitive in the technological library
  - Technological optimizations + cell timings
- Place & Route - map primitive to circuit and route signals between them
  - Physical optimization (retiming) + wire timings

Each step must be validated

- Tools are not perfect
- Timing issues (inaccurate models)



Logical synthesis flow (FPGA)

AMD Xilinx Vivado

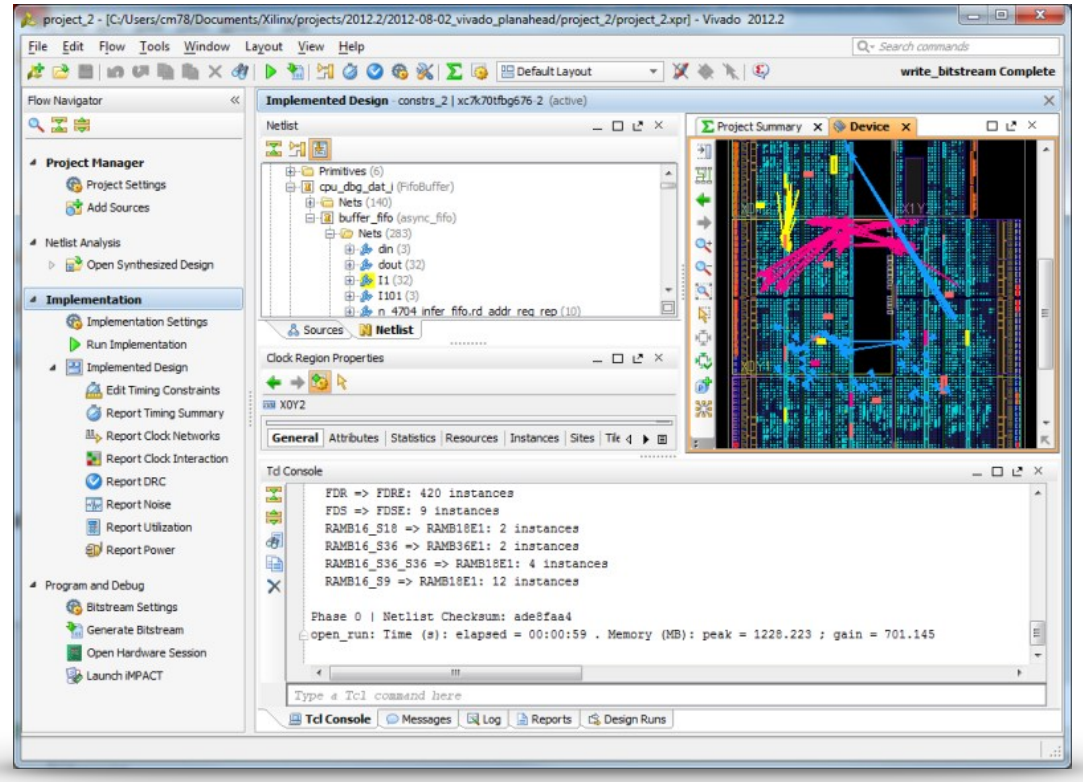
- VHDL & Verilog editor
- Logical synthesis
- Place & Route (Xilinx FPGA only)
- Bitstream generation

Simulation (xsim)

- Graphical User Interface

Getting Xilinx Vitis

- Available from internet (up to 140 Go)
- Free limited version for students (no limitation for small FPGAs)
- Ask your teacher for installation help



Logical synthesis flow (FPGA)

INTEL Altera Quartus

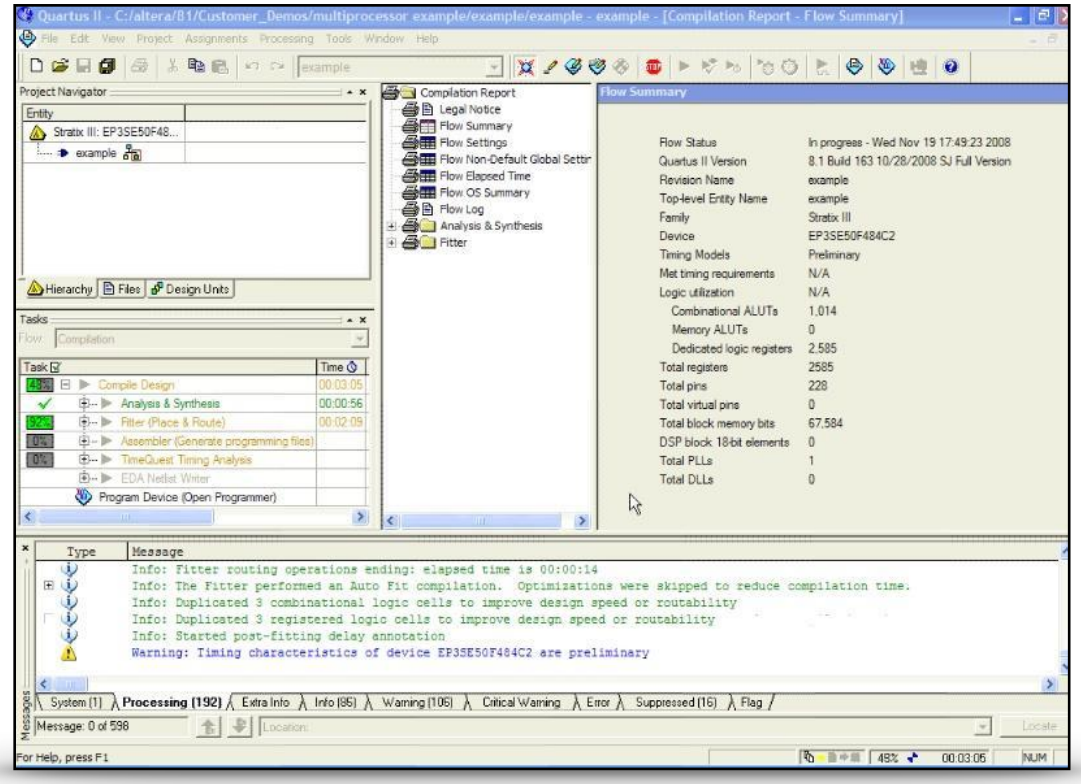
- VHDL & Verilog editor
- Logical synthesis
- Place & Route (Altera FPGA only)
- Bitstream generation

Simulation (modelsim)

- External simulation tool

Getting INTEL Quartus

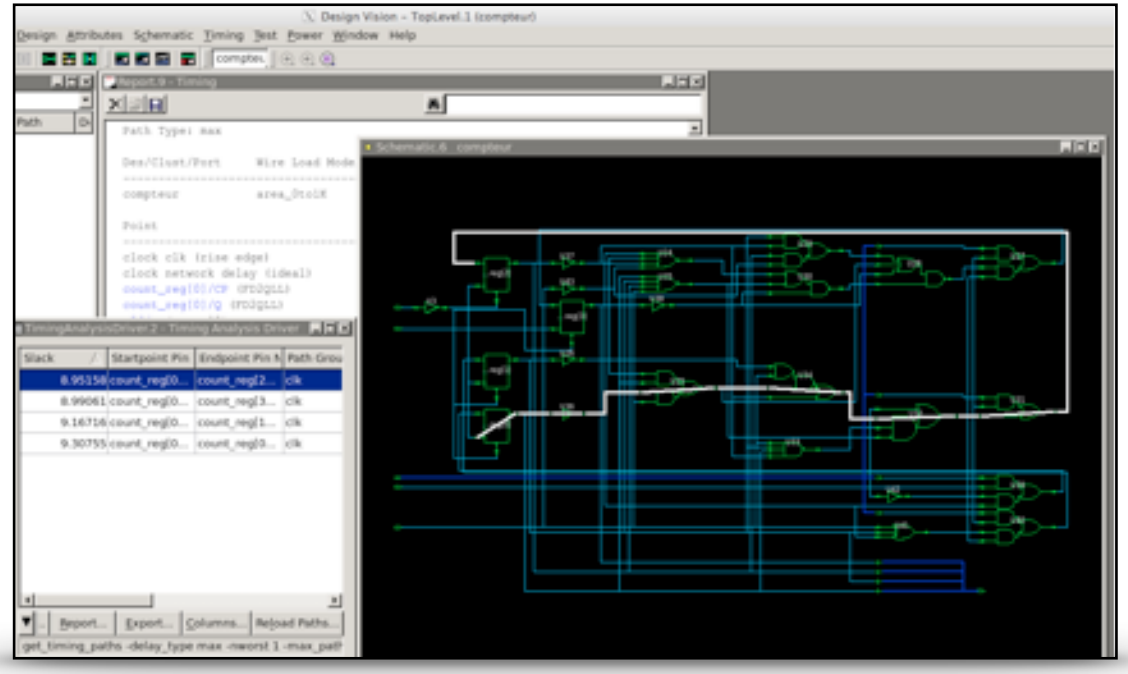
- Available from internet (up to ~30 Go)
- Free limited version for students (no limitation for small FPGAs)
- Ask your teacher for installation help



Logical synthesis flow (ASIC)

- Design Compiler (Synopsys)
  - VHDL & Verilog editor
  - Logical synthesis
  - Place & Route : rely on external tools
  - Constrained by ASIC PDKs
- Simulation (modelsim)
  - External simulation tool
- Getting Design Complier
  - No free version (industrial tool)
- Alternative & OpenRoad
  - Yosys + NextPnR (open-source)

SYNOPSYS®



Simulation tools (1)

Modelsim

- VHDL & verilog simulator
- Performs data visualization
- Developed by Mentor

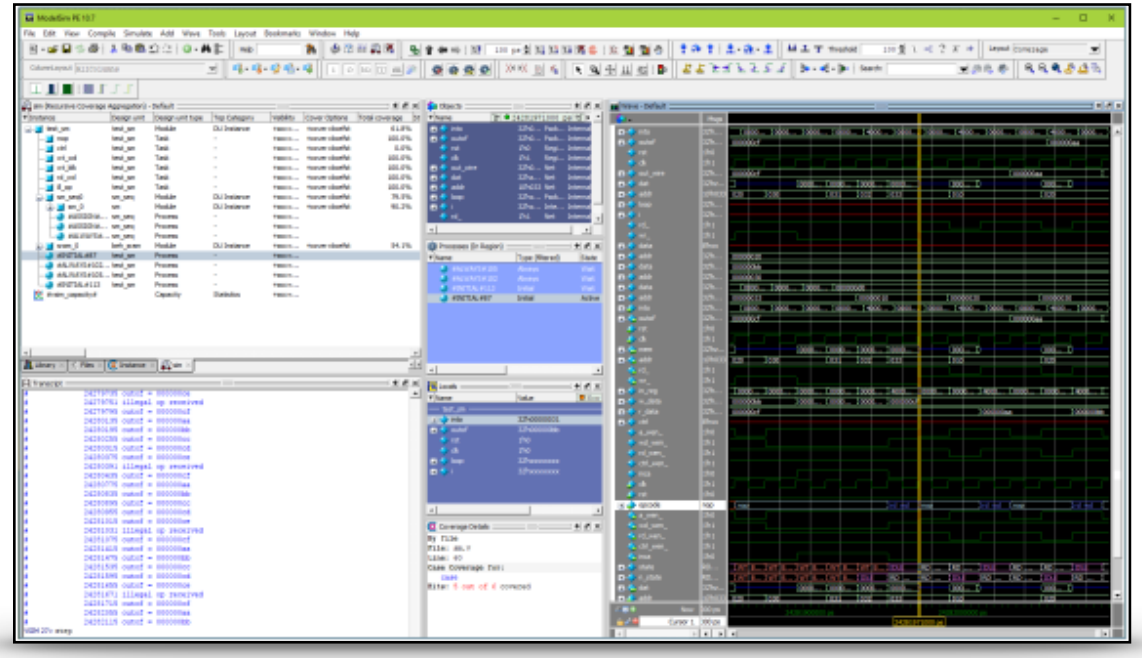
Mentor®
A Siemens Business

Using modelsim

- Graphical User Interface
- Command line interface
- Environnement script (school)

Getting Modelsim

- Available on Windows & Linux
- Free limited version for students



Simulation tools (2)

GHDL tool

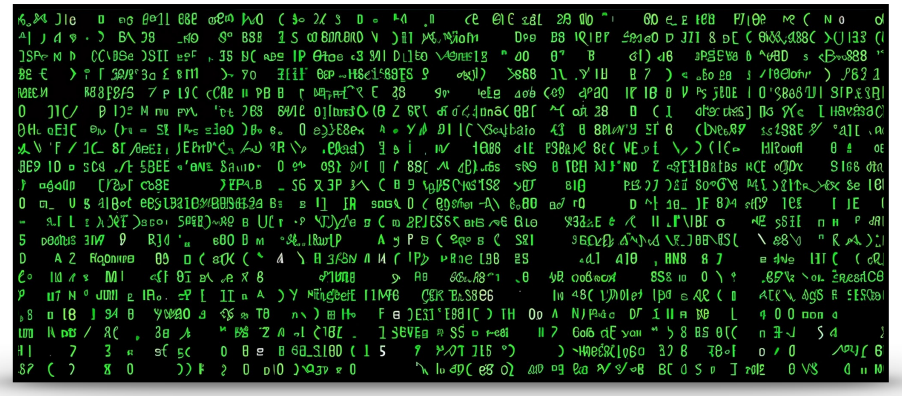
- VHDL (only) simulator
- Developed first by Tristan Gingold (Fr)

Using GHDL

- Digital simulator event based sim
- Command line interface only
- Scripts are provided in the lab

Getting GHDL

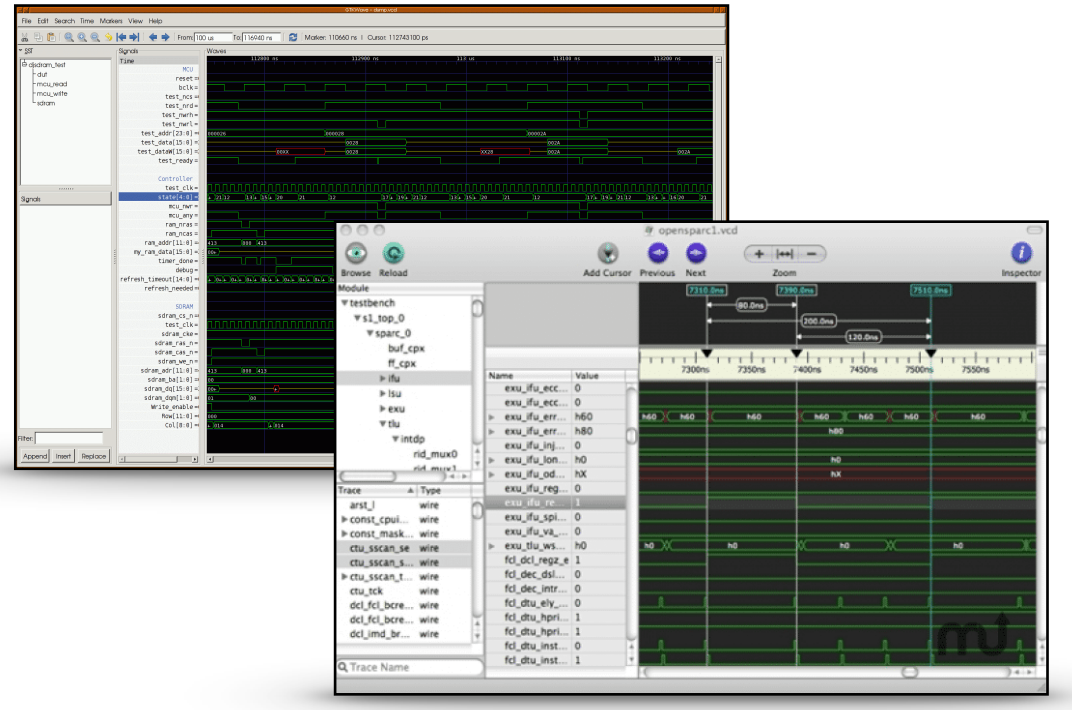
- Available on Windows & Linux & MacOS
- Ask your teacher for installation



External visualization tools

Gtkwave

- Trace visualizer
- Open source project
- Works on Windows & Linux & MacOS

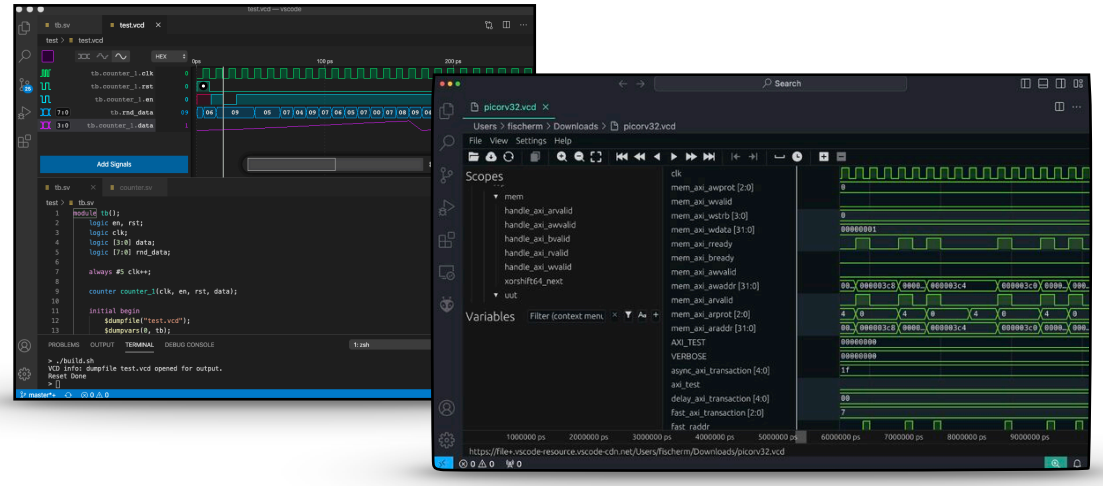


Scansion

- Trace visualizer
- Industrial tool (free now)
- Works on MacOS

Visual code plugins

- Surfer (my choice)
- WaveTrace
- VaporView
- others...



ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[3 - Basic Language Concepts]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

VHDL language

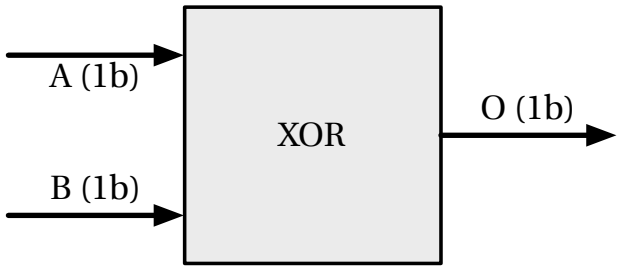
- VHDL is a verbose language
- It forces you to structure your descriptions
  - Remove all description ambiguities
  - Debugging hardware isn't fun!!!
- 5 basic concepts to master
  - Entities
  - Architectures
  - Processes
  - Signals
  - Variables

The « entity » concept

- Entity declaration
  - Module name
  - List of I/O names
  - List of I/O data types
- For each module I/O
  - Port direction (input and/or output)
  - Type of data exchanged
- I/O order has no impact
  - However, pay attention to « port map »
- Entity defines a component in your component library

```
ENTITY nom_entite IS
     PORT (
          -- LISTE DES ENTREES
          entree_1 : IN  TYPE_ENTREE;
          -- ... ... ... ...
          entree_n : IN  TYPE_ENTREE;

          -- LISTE DES SORTIES
          sortie_1 : OUT TYPE_SORTIE;
          -- ... ... ... ...
          sortie_n : OUT TYPE_SORTIE
     );
END nom_entite;
```

Describing a component (1)



Entity name corresponds to component name in library

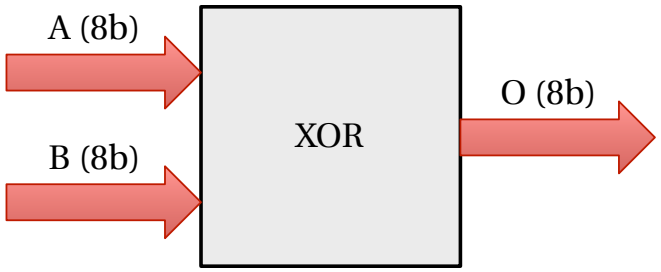
```
ENTITY xor_v1 IS
     PORT (
          A : in  STD_LOGIC;
          B : in  STD_LOGIC;
          O : out STD_LOGIC
     );
END xor_v1;
```

The name given to the input and output ports

Data type STD_LOGIC is used for one bit data

Port direction is specified using IN & OUT keywords

Describing a component (2)



Entity name corresponds to component name in library

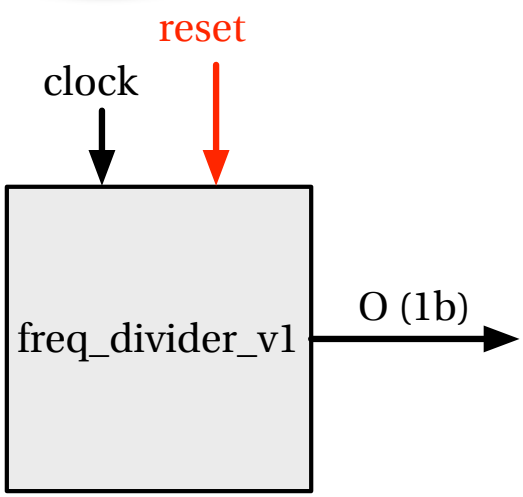
```
ENTITY xor_v2 IS
     PORT (
          A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          O : out STD_LOGIC_VECTOR(7 DOWNTO 0)
     );
END xor_v2;
```

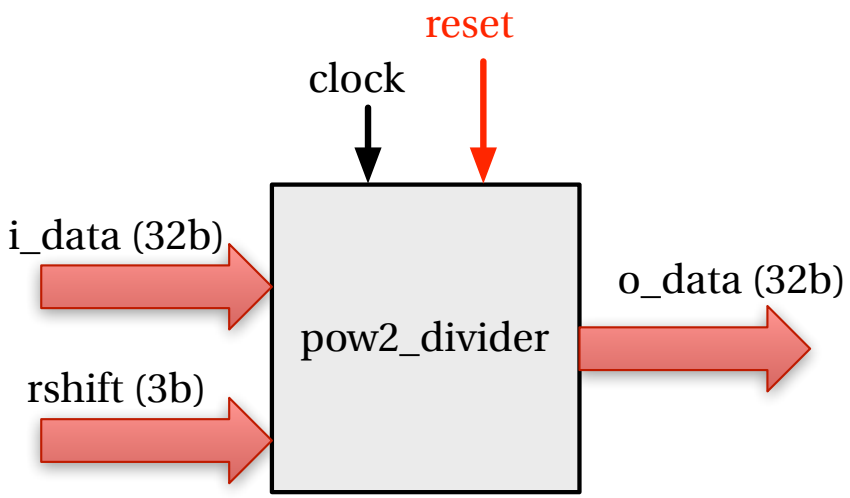
The name given to the input and output ports

Port direction is specified using IN & OUT keywords

STD_LOGIC_VECTOR type used here manage multiple-bit data

Describing a component (3)






```
ENTITY freq_divider_v1 IS
     PORT (
          clk    : in  STD_LOGIC;
          reset  : in  STD_LOGIC;
       O      : out STD_LOGIC
     );
END freq_divider_v1;
```

```
ENTITY pow2_divider_v1 IS
     PORT (
          clk     : in  STD_LOGIC;
          reset   : in  STD_LOGIC;
          i_data  : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
          rShift  : in  STD_LOGIC_VECTOR( 2 DOWNTO 0);
          o_data  : out STD_LOGIC_VECTOR(31 DOWNTO 0)
     );
END pow2_divider_v1;
```

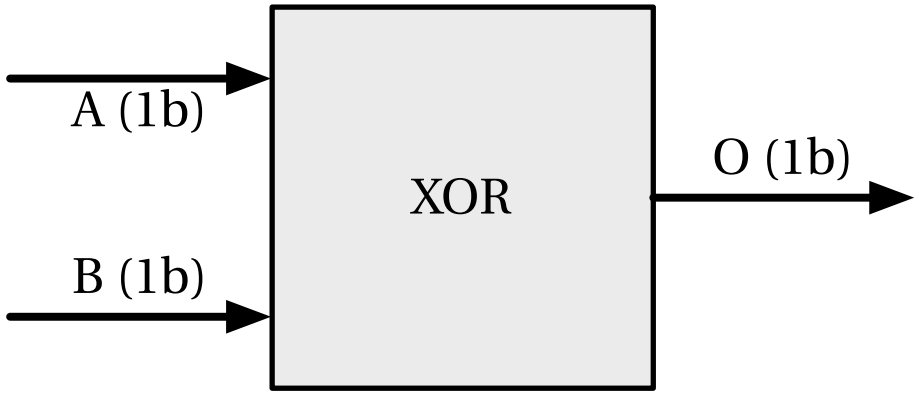
Clock and reset signals are binary signals like others

The « architecture » concept (1)

- Entities fixed the module name as its IOs
- Architecture section specifies the module internal behavior
- The architecture is divided into 2 distinct parts:
  - Declaration of signals and sub-modules used in the architecture
  - Declaration of the behavior itself
- Behavior can be described in different ways...

```
ARCHITECTURE nom_arch OF nom_entite IS
     --
     -- DECLARATION DES SIGNAUX INTERNES
     --
     SIGNAL signal_1 : TYPE_SIGNAL;
     -- ... ... ... ... ... ... ...
     -- ... ... ... ... ... ... ...
     SIGNAL signal_n : TYPE_SIGNAL;

     --
     -- DECLARATION DES "COMPONENTS"
     --

BEGIN

     --
     -- DU COMPORTEMENT DE L'ARCHITECTURE
     -- + PROCESSUS IMPLICITES,
     -- + PROCESSUS EXPLICITES,
     -- + INSTANCIATION DE SOUS MODULES,
     --

END nom_arch;
```

Using implicit process




```
ENTITY xor_gate_comb IS
PORT (
        A : IN  std_logic;
        B : IN  std_logic;
        C : OUT std_logic
    );
END xor_gate_comb;
```

The module output is calculated using an implicit process

Each time A or B is modified, the result of (A **xor** B) is re-evaluated

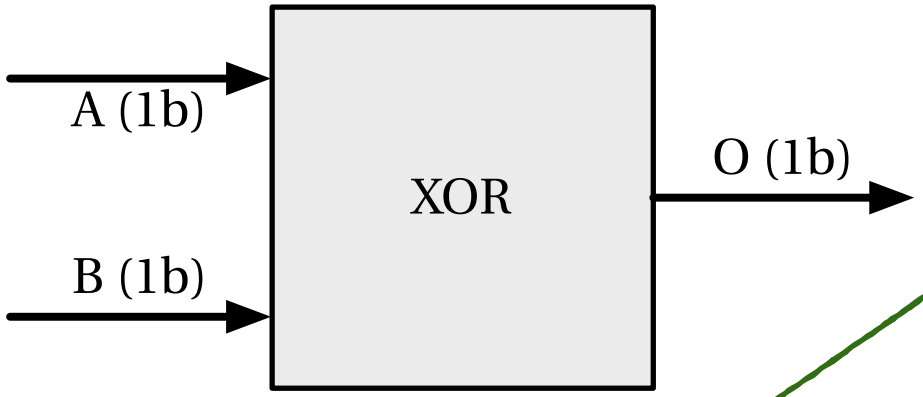
```
ARCHITECTURE arch OF xor_gate_comb IS
BEGIN

    C <= A XOR B;

END arch;
```

The description of behavior is concise, but limited to simple cases

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 1-xor-gate-comb
  - Analyse the VHDL description of the XOR module
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Using explicit process (1)




```
ARCHITECTURE arch OF xor_gate_process IS
BEGIN

    PROCESS(A, B)
    BEGIN
         C <= A XOR B;
    END PROCESS;

END arch;
```

The output calculation process is explicitly written in this example

When A or B changes, the behavior described in the process must be re-evaluated

Many processes can be described in a single architecture when it is needed

Beware of the sensitivity list !

It can generate simulation issues

It's time for me to take a break!

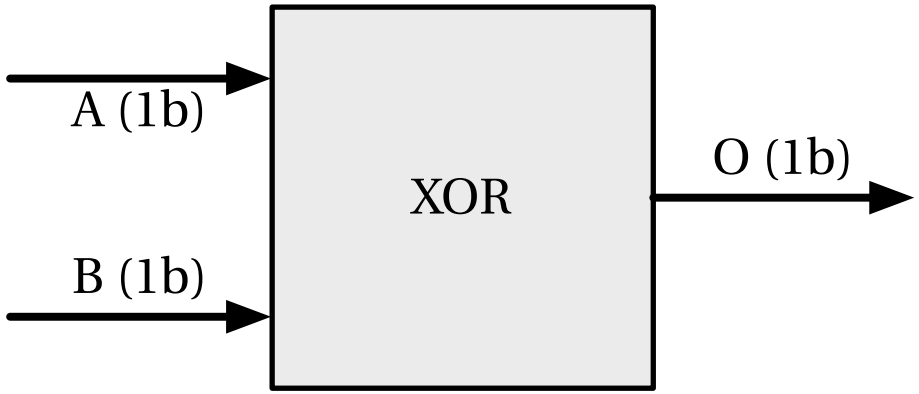
Analyze and simulate this module description

- Move to the example directory
  > cd 2-xor-gate-process
- Analyse the VHDL description of the XOR module
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 2

Modify also the sensitivity list to produce errors !

Using explicit process (2)




```
ARCHITECTURE arch OF xor_gate_split IS
BEGIN

    PROCESS(A, B)
    BEGIN
        C <= ((NOT A) AND B) OR (A AND (NOT B));
    END PROCESS;

END arch;
```

The module output is calculated using an explicit process

Beware of the sensitivity list !

It can generate simulation issues

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 3-xor-gate-split
- Analyse the VHDL description of the XOR module
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 3

The signal keyword

- I/O elements can be access naturally (read or write)
  - To decompose computations it is necessary to declare and use temporary values (wires)
- Signals are used to interconnect elements
  - They are (for now) equivalent to wires
- Signals are declared before architecture begin
  - Each signal has a name
  - Each signal has a data type

```
ENTITY xor_gate_signals IS
PORT (
        A : IN  std_logic;
        B : IN  std_logic;
        C : OUT std_logic
    );
END xor_gate_signals;

ARCHITECTURE arch OF xor_gate_signals IS
    SIGNAL T1 : STD_LOGIC;
    SIGNAL T2 : STD_LOGIC;
    SIGNAL T3 : STD_LOGIC;
    SIGNAL T4 : STD_LOGIC;
BEGIN

    T1 <= NOT A;
    T2 <= NOT B;
    T3 <= T1 AND  B;
    T4 <=  A AND T2;
    C  <= T3  OR T4;

END arch;
```

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 4-xor-gate-signals
  - Open the VHDL description of the XOR module and complete it
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Bit vectors

- **std_logic** type
  - Used to handle bit values
- **std_logic_vector** type
  - Used to handle bit vectors
  - Range (N-1 **downto** 0) should be fixed
  - **downto** syntax is preferred to **upto**
- Logical operation can be applied to logic vectors
  - Vector length should be the same
- One element bit vector
  - **std_logic_vector**(0 **downto** 0) exists but avoid it (equals **std_logic**)

4-bit I/O vectors

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_gate_vector IS
PORT (
       A : IN  std_logic_vector(3 downto 0);
       B : IN  std_logic_vector(3 downto 0);
       C : OUT std_logic_vector(3 downto 0)
   );
END xor_gate_vector;

ARCHITECTURE arch OF xor_gate_vector IS
BEGIN

       C <= A XOR B;

END arch;
```

4-bit **xor** operation

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 5-xor-gate-vector
  - Analyse the VHDL description of the XOR module
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Bit Split and Bit concatenate

- All bits from a vector are not always useful
  - Let' consider a signal A:
    A : **std_logic_vector**(15 downto 0)
- Bit selection
  - B <= A(15); // select 1 bit
  - C <= A(14 **downto** 0); // select 15 bits
  - A(12) <= B;
- Bit concatenation
  - D <= B & B // generate 2 bits
  - E <= B & C // generate 16 bits
  - D <= "00" & D // generate 4 bits
  - D <= '0' & B & '1' // generate 3 bits

Bit Selection

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_v1 IS
PORT (
       A : IN  std_logic_vector(3 downto 0);
       B : IN  std_logic_vector(3 downto 0);
       C : OUT std_logic_vector(3 downto 0)
   );
END xor_v1;

ARCHITECTURE arch OF xor_v1 IS
BEGIN

   C(0) <= A(0) XOR B(0);
   C(1) <= A(1) XOR B(1);
   C(2) <= A(2) XOR B(2);
   C(3) <= A(3) XOR B(3);

END arch;
```

A single bit of a vector can be read or written with the parenthesis operator

Selection of Bit Vectors

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_v2 IS
PORT (
       A : IN  std_logic_vector(3 downto 0);
       B : IN  std_logic_vector(3 downto 0);
       C : OUT std_logic_vector(3 downto 0)
   );
END xor_v2;

ARCHITECTURE arch OF xor_v2 IS
BEGIN

   C(1 DOWNTO 0) <= A(1 DOWNTO 0) XOR B(1 DOWNTO 0);
   C(3 DOWNTO 2) <= A(3 DOWNTO 2) XOR B(3 DOWNTO 2);

END arch;
```

It is also possible with the parenthesis operator to read from or write to sub-vectors of arbitrary size

Bit Concatenation

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_v3 IS
PORT (
        A : IN  std_logic_vector(3 downto 0);
        B : IN  std_logic_vector(3 downto 0);
        C : OUT std_logic_vector(3 downto 0)
    );
END xor_v3;

ARCHITECTURE arch OF xor_v3 IS
    SIGNAL D0, D1, D2, D3 : STD_LOGIC;
BEGIN

    D3 <= A(3) XOR B(3);
    D2 <= A(2) XOR B(2);
    D1 <= A(1) XOR B(1);
    D0 <= A(0) XOR B(0);
    C  <= (D3 & D2 & D1 & D0);

END arch;
```

An example of the use of bit aggregation after information processing

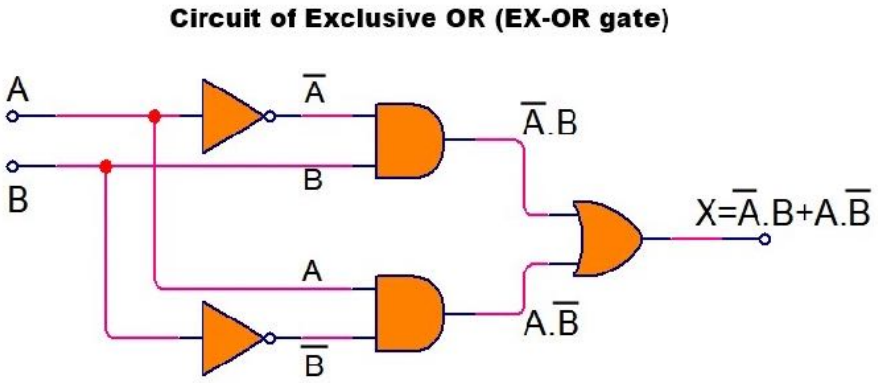
It's time for me to take a break!

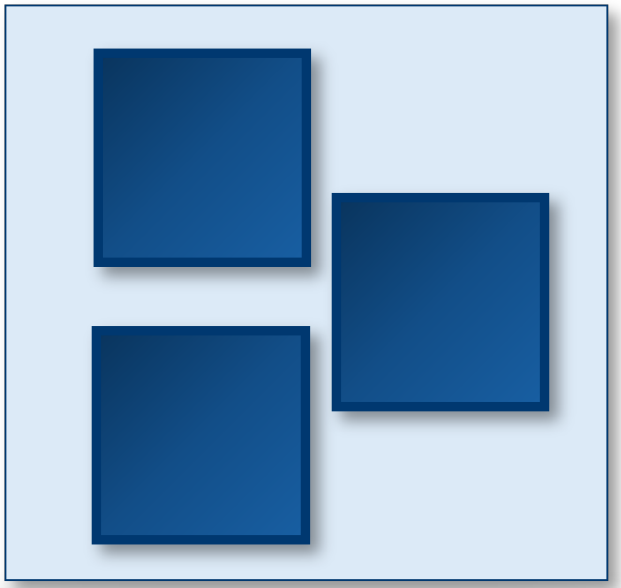
Analyze and simulate this module description

- Move to the example directory
  > cd 6-xor-gate-others/{v1,v2,v3}
- Analyse the VHDL descriptions of the XOR modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronograms
  > make gtkwave
- Observe the RTL designs
  > make dc-rtl

Hierarchical descriptions

- Architecture can quickly become complex !
- VHDL modules can be composed of sub-modules
  - Reduces the number of VHDL lines
  - Improve reusability (like C libraries)
  - Hierarchical description enables complex designs e.g. processors
- Instantiated modules should have
  - A name
  - A type (a defined entity)
  - An interconnection mapping
  - Configuration parameters (generic)





Component declarations

Current module declaration

Declaration of an existing component

Same declaration that the « entity » with the COMPONENT keyword

Wire declarations required to interconnect the components

```
ENTITY xor_portmap IS
PORT (
        A : IN  std_logic;
        B : IN  std_logic;
        C : OUT std_logic
    );
END xor_portmap;

ARCHITECTURE arch OF xor_portmap IS

    COMPONENT or_gate IS
    PORT (
        A : IN  std_logic;
        B : IN  std_logic;
        C : OUT std_logic
    );
    END COMPONENT;
    --
    -- other components...
    --

    SIGNAL T1, T2, T3, T4 : STD_LOGIC;

BEGIN
    --
    -- code lines...
    --
END arch;
```

Component instantiation (1)

```
ARCHITECTURE arch OF xor_portmap IS
    --
    -- component declarations...
    --
    SIGNAL T1, T2, T3, T4 : STD_LOGIC;
BEGIN

    c1 : not_gate port map(
                            A =>  A,
                            B => T1);

    c2 : not_gate port map(A =>  B, B => T2          );
    c3 : and_gate port map(A => T1, B => B,  C => T3);
    c4 : and_gate port map(A =>  A, B => T2, C => T4);

    c5 :  or_gate port map(
                            A => T3,
                            B => T4,
                            C =>  C);

END arch;
```

Name of the component

Mapping of I/O signals

Type of the component

Instantiation primitive

Component instantiation (2)

```
ARCHITECTURE arch OF xor_portmap IS

    --
    -- component declarations...
    -
    SIGNAL T1, T2, T3, T4 : STD_LOGIC;

BEGIN
    c1 : not_gate port map(A,T1);
    c2 : not_gate port map(B,T2);
    c3 : and_gate port map(T1, B,  T3);
    c4 : and_gate port map( A, T2, T4);
    c5 :  or_gate port map(T3, T4,  C);
END arch;
```

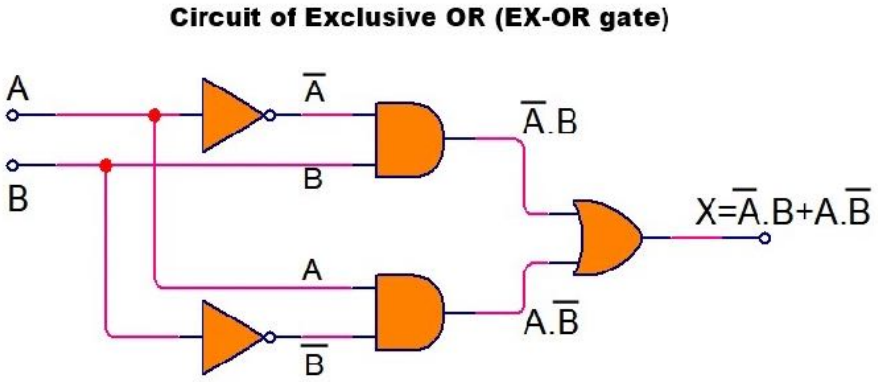
A more concise way to interconnect the instantiated modules

Interconnection is done according to the signal order in the COMPONENT declaration !

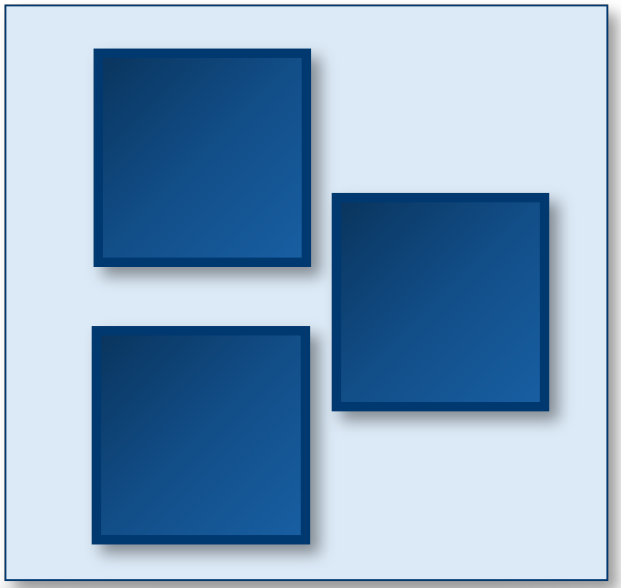
During this teaching sequence, you **should** use first approach !

XOR hierarchical description

- Analyze and simulate this module description
  - Move to the example directory
    > cd 7-xor-gate-portmap
  - Open the descriptions and complete the XOR description
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl



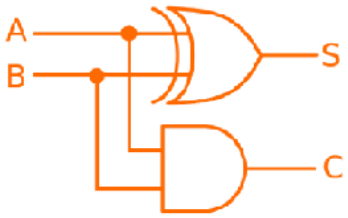
Hierarchical description



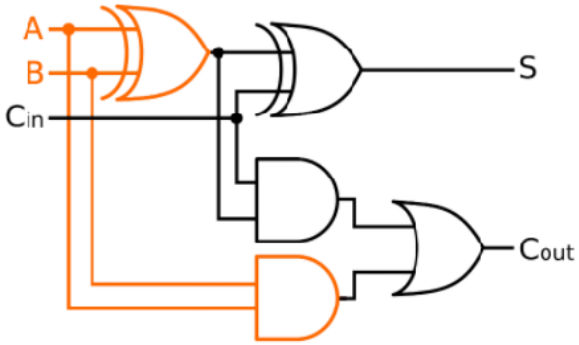
Let's build a 4 bit adder (1)

- Write a module that describe a 4-bit adder
  - Move to the example directory
    > cd 8-adder_comb
  - Write the VHDL description of the module (combinatory)
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

$$\begin{array}{rr} & A_0 \\ + & B_0 \\ \hline R_0 & S_0 \end{array}$$



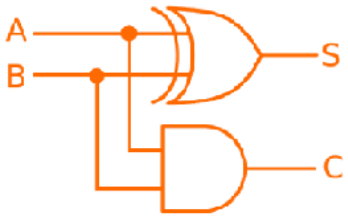
$$\begin{array}{rr} & A_1 \\ + & B_1 \\ + & R_0 \\ \hline R_1 & S_1 \end{array}$$



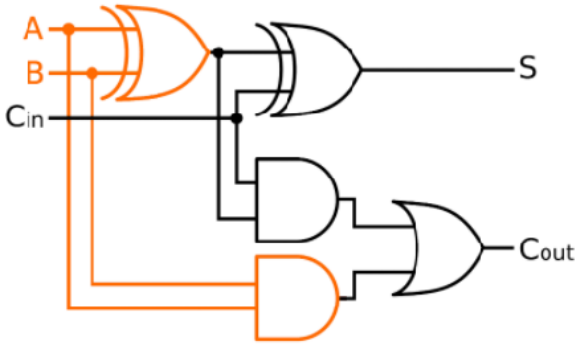
Let's build a 4 bit adder (2)

- Write a module that describe a 4-bit adder
  - Move to the example directory
    > cd 9-adder_portmap
  - Write the VHDL description of the module (port map)
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

$$\begin{array}{rr} & A_0 \\ + & B_0 \\ \hline R_0 & S_0 \end{array}$$

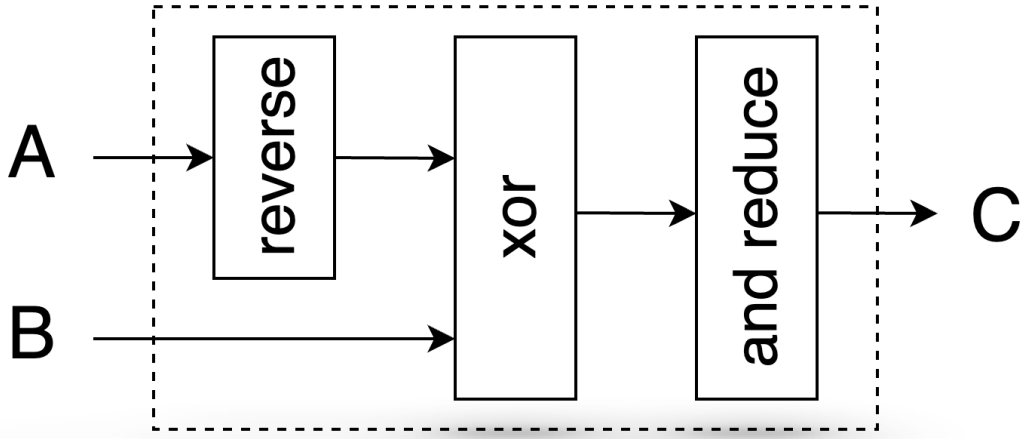


$$\begin{array}{rr} & A_1 \\ + & B_1 \\ + & R_0 \\ \hline R_1 & S_1 \end{array}$$



Let's design a custom function

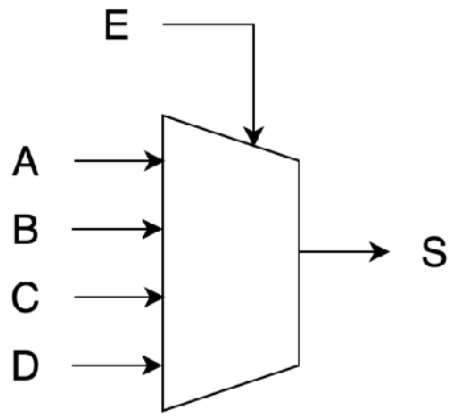
- Write a module that describes a custom func.
  - Move to the example directory
    > cd 10-fonc-logique
  - Write the VHDL description of the proposed module
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

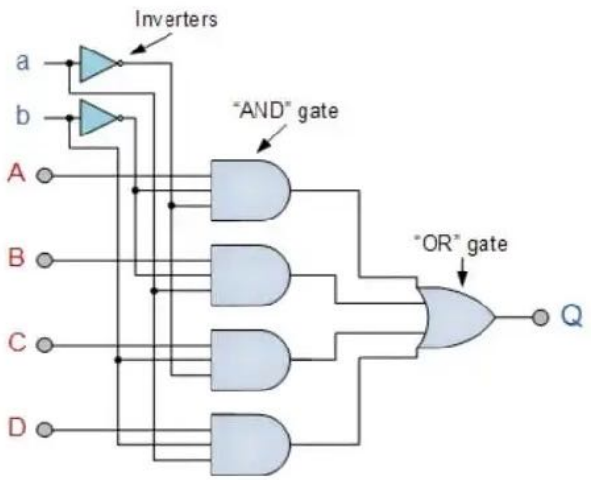


Reverse : *bit reversal*
And reduce : *and reduction*

Conditional constructs

- VHDL is a rich language with many constructs
- One behavior, multiple descriptions
  - Process or not
  - Hierarchical design
- Many constructs for conditional structures
  - With or without process (confusing)
  - Adapted to certain use cases
- Let's describe a simple multiplexor (4 → 1)





When-Else conditional structure

```
ENTITY mux_4_when_else IS
PORT (
        A : IN  std_logic_vector(3 downto 0);
        B : IN  std_logic_vector(3 downto 0);
        C : IN  std_logic_vector(3 downto 0);
        D : IN  std_logic_vector(3 downto 0);
        E : IN  std_logic_vector(1 downto 0);
        S : OUT std_logic_vector(3 downto 0)
    );
END mux_4_when_else;

ARCHITECTURE arch OF mux_4_when_else IS
BEGIN

    S <= A WHEN E = "00" ELSE
         B WHEN E = "01" ELSE
         C WHEN E = "10" ELSE
         D;

END arch;
```

4-input to 1-output multiplexer

The **ELSE** ... **WHEN** structure is used to implicitly describe multiplexers

Implicit processes can express “simple” conditional structures.

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 11-mux-when-else
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Case-When conditional structure

The **CASE** ... **WHEN** structure used in explicit processes provides the same behavior

The approach is more verbose, but it is possible to add conditions to **WHEN**s (see FSMs)

The **WHEN others** is mandatory, and is often used to manage the case that is not described (here "11")

4-input to 1-output multiplexer

```
ARCHITECTURE arch OF mux_4_case_is IS
BEGIN

    PROCESS(A, B, C, D, E)
    BEGIN
        CASE E IS
            WHEN "00"   => S <= A;
            WHEN "01"   => S <= B;
            WHEN "10"   => S <= C;
            WHEN OTHERS => S <= D;
        END CASE;
    END PROCESS;

END arch;
```

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 12-mux-case-is
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

If-Elsif-Else conditional structure

The **IF … ELSIF … ELSE** structure used in explicit processes provides the same behavior

The approach is more verbose, but it is possible to add specific conditions to some statements (see FSMs)

Pay attention: the **ELSIF** clause is has a different meaning than **ELSE IF** from a hardware point of view

The test sequence is ordered here

4-input to 1-output multiplexer

```
ARCHITECTURE arch OF mux_4_if_else IS
BEGIN

    PROCESS(A, B, C, D, E)
    BEGIN
        IF    ( E = "00" ) THEN S <= A;
        ELSIF( E = "01" ) THEN S <= B;
        ELSIF( E = "10" ) THEN S <= C;
        ELSE                   S <= D;
        END IF;
    END PROCESS;

END arch;
```

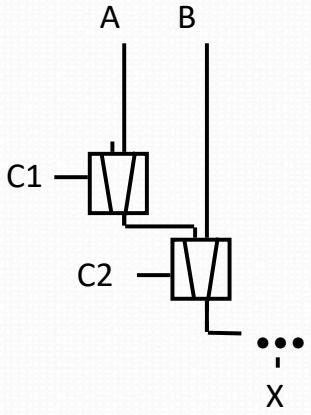
It's time for me to take a break!

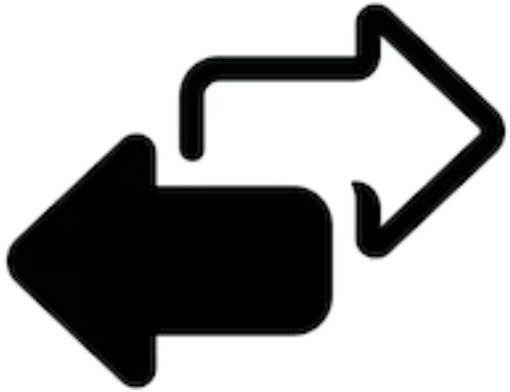
- Analyze and simulate this module description
  - Move to the example directory
    > cd 13-mux-if-else
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

What you have is what you described

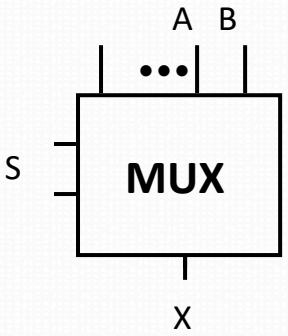
With priority

```
process (A,B,...C1,...)
begin
   if C1  then
      X  <= A;
   elsif C2  then
      X  <= B;
   •••
   else
   end if;
end process;
```





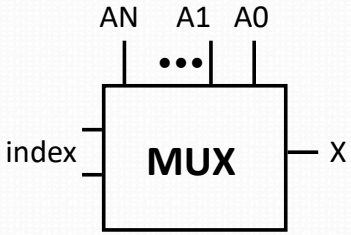
Without priority




```
process (A,B,...,S)
begin
   case (S) is
    when C1 =>
      X  <= A;
   when C2 =>
      X  <= B;
   •••
end process;
```

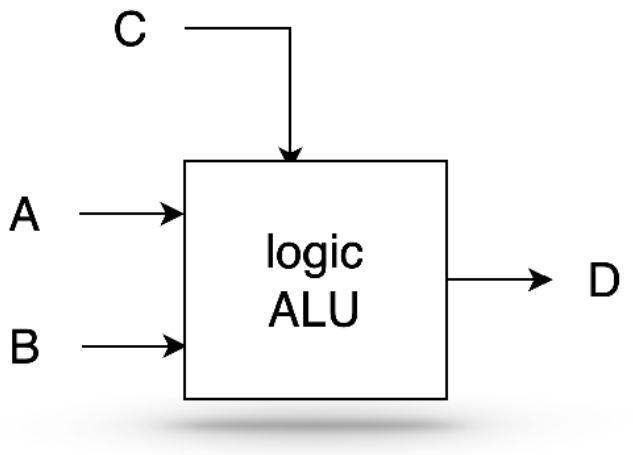
Array Index

```
X <= A(index);
```



Make your own VHDL description

- Describe a simple logic ALU with the following behavior
- Module interface
  - 2 logic inputs (**A**, **B**)
  - A selection signal (**C**)
  - 1 output (**D**)
- 7 logic features
  - **NOT** A, **NOT** B,
  - A **AND** B, A **OR** B, A **NAND** B, A **NOR** B, A **XOR** B
- Choose your coding style (easiest for you)



It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 14-alu-logique
- Open the description of the VHDL module and complete it
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[4 - Simulating VHDL modules]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

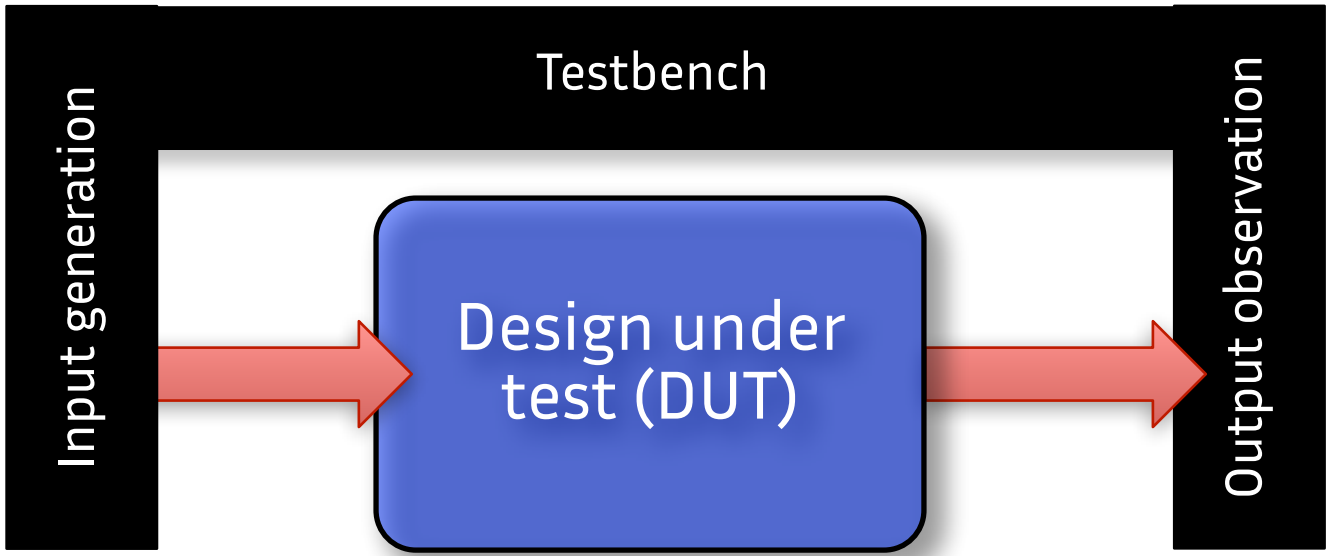
bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

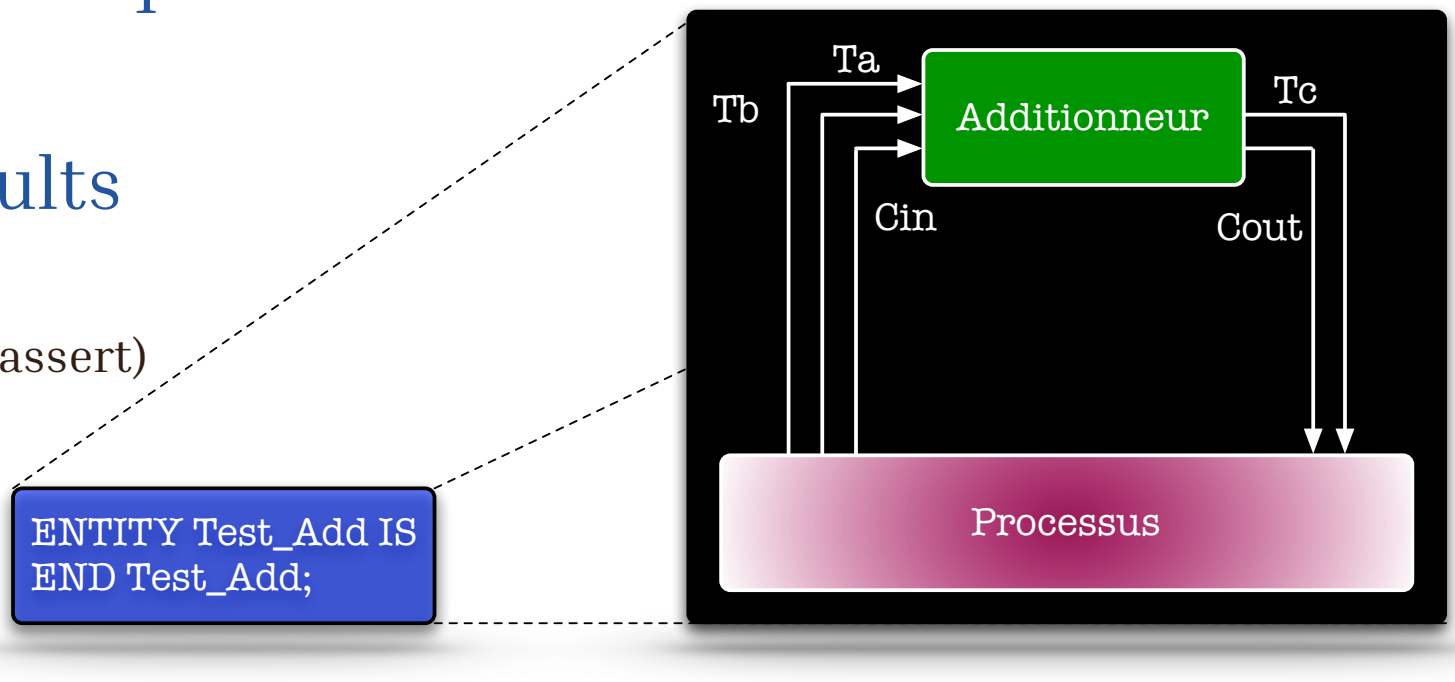
The DUT philosophy

- A **testbench** is generally understood as the set of code pieces used to apply a predetermined set of inputs to a **component** in order to observe its response
  - A testbench cannot be synthesized neither on ASIC or FPGA!



Testbench structure

- Instantiation of the module to be checked
- Initialization of input signals
- Generation of a sequence of stimuli
- Analysis of results
  - Visual analysis
  - Automatic analysis (assert)



VHDL source code

Empty entity declaration

Declaration of the design under test

Definition of the interconnection wires

Instantiation of tested module with I/O interconnects

```
entity testbench is
end;

architecture bench of testbench is

  component xor_gate_comb
  PORT (
    A : IN  std_logic;
    B : IN  std_logic;
    C : OUT std_logic
  );
  end component;

  signal A : std_logic;
  signal B : std_logic;
  signal C : std_logic;

begin

  uut: xor_gate_comb port map ( A => A,
                                B => B,
                                C => C );

    -- ............
    -- I/O generation
    -- process (next slide)
    -- ............

end;
```

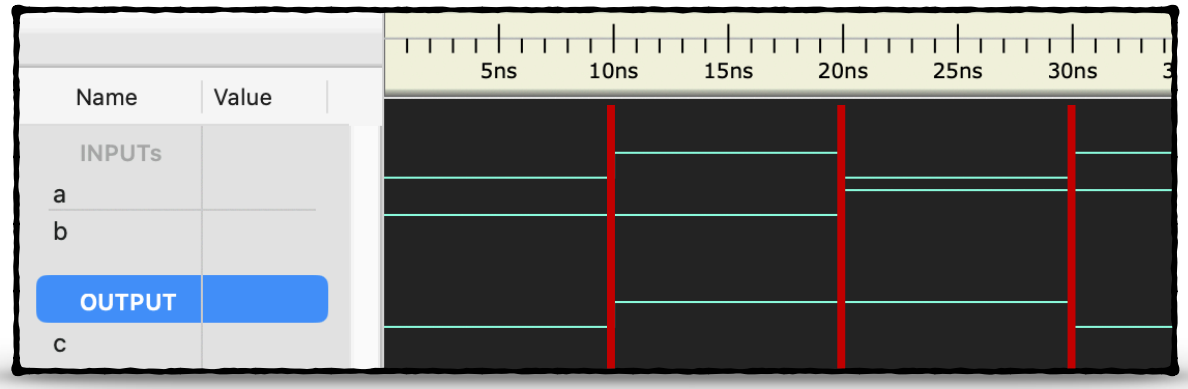
From chronogram to test bench

Processus declaration (no sensitivity)

Initialization of A&B inputs

Wait statement - makes simulation time progress

```
stimulus: process
 begin
   REPORT "Simulation start...";
   A <= '0';
   B <= '0';
   wait for 10 ns;
   A <= '1';
   B <= '0';
   wait for 10 ns;
   A <= '0';
   B <= '1';
   wait for 10 ns;
   A <= '1';
   B <= '1';
   wait for 10 ns;
   -- ............
   REPORT "Simulation end...";
   wait;
 end process;
```



It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 15-write-testbench
  - Analyse the description of the **tzcnt** VHDL module
    > make vscode
  - Write a complete testbench for the module
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave

Understanding the simulation proc.

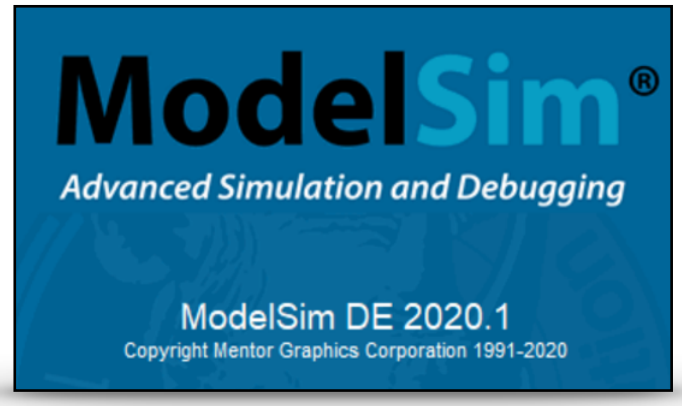
- Since the beginning you launch make scripts
  - They call simulation tools
  - They call visualization tools
- Avoid you to learn
  - GUI tool usage
  - Command lines
- The GNU make tool
  - Launch command sequences described in the Makefile
- Let's have a look to what they hide

GNU Make

Introduction to Modelsim

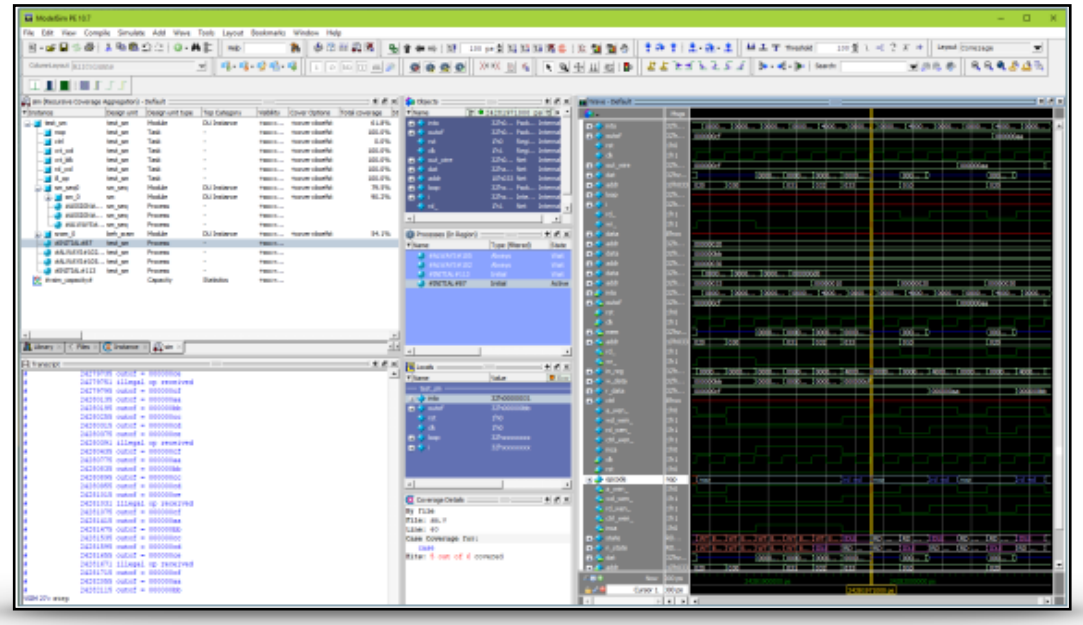
Modelsim simulation

- Industrial tool (expensive)
- GUI or CLI simulation modes



Simulation procedure

- vsim (in **modelsim** directory)
- File - New project - Give a name
- Add Files - Your VHDL files
- Projet - Right click - Compile all
- Library - work - right click on testbench - Simulate
- Select your component & Add Waves



Introduction to GHDL & GtkWave

GHDL tool

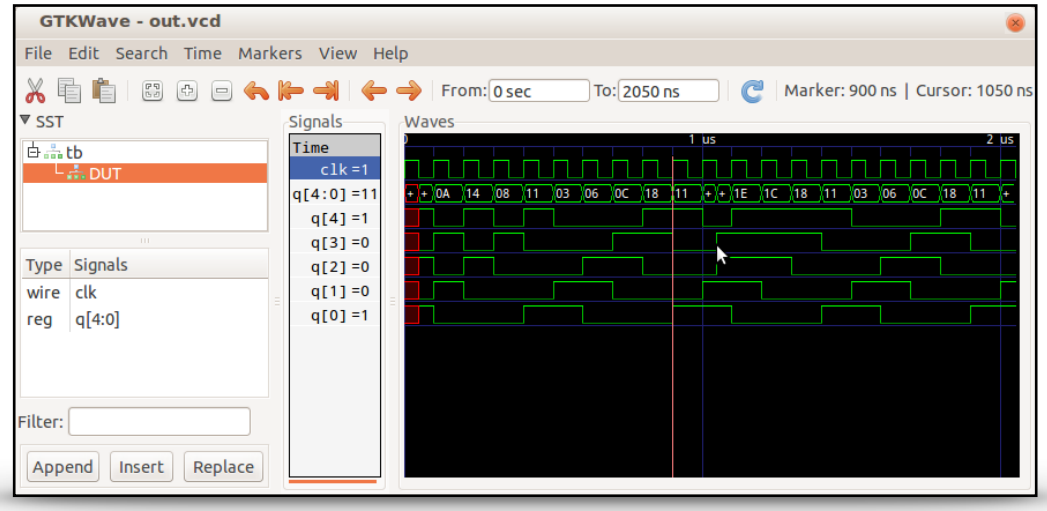
- Open-source VHDL simulator
- Generate a executable model

GTKWave tool

- Signal trace viewer (vcd files)

Simulation procedure

- ghdl -a tzcnt.vhd
- ghdl -a testbench.vhd
- ghdl -e testbench
- ghdl -r testbench --stop-time=400ns
- ghdl -r testbench --stop-time=400ns —vcd=trace.vcd
- gtkwave trace.vcd



ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[5 - Modeling Digital Systems]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

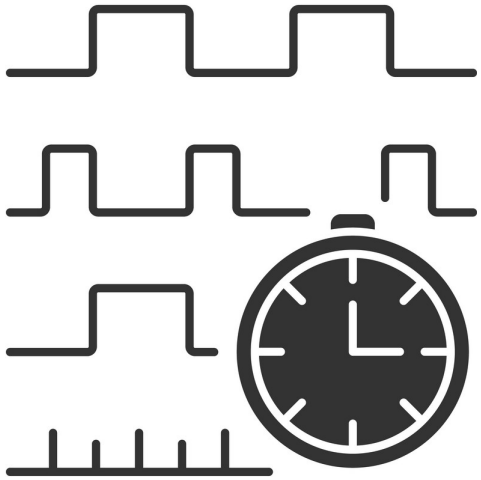
Université de Rennes

École affiliée IMT

Synchronous behavior (1)

Combinatorial circuits

- The value of the inputs is enough to compute the output value
- A combination of inputs corresponds to a single output state

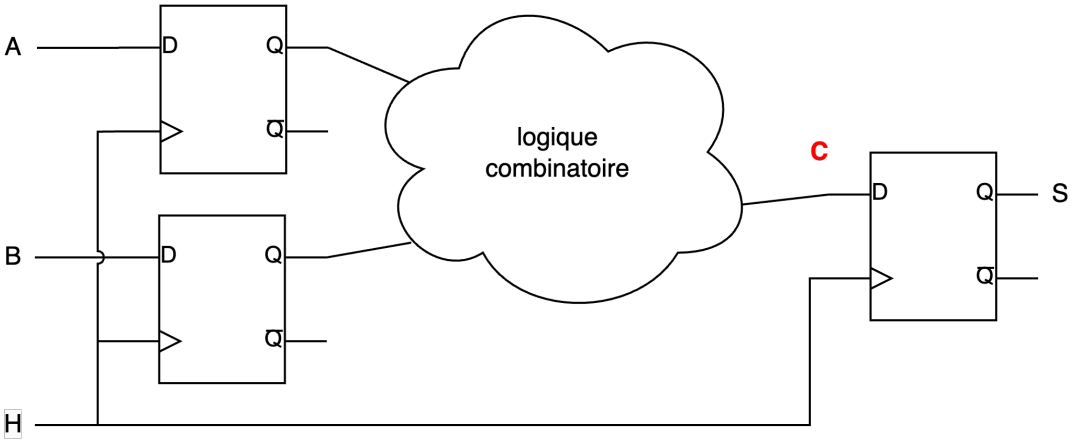


Sequential circuits

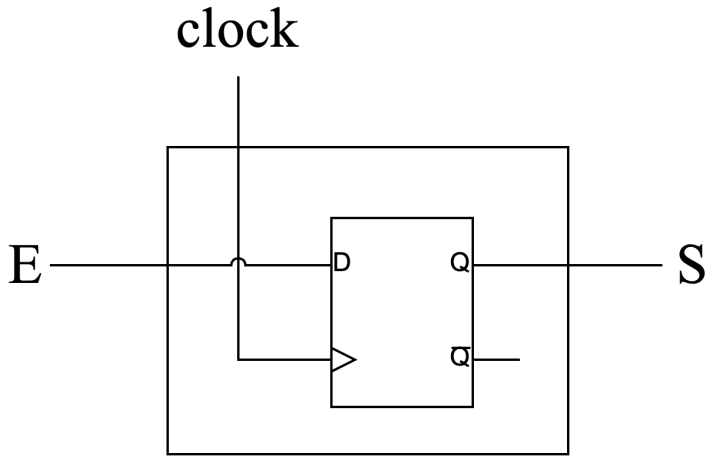
- A set of inputs can give ≠ output states depending on the system's history

Storing system states (time)

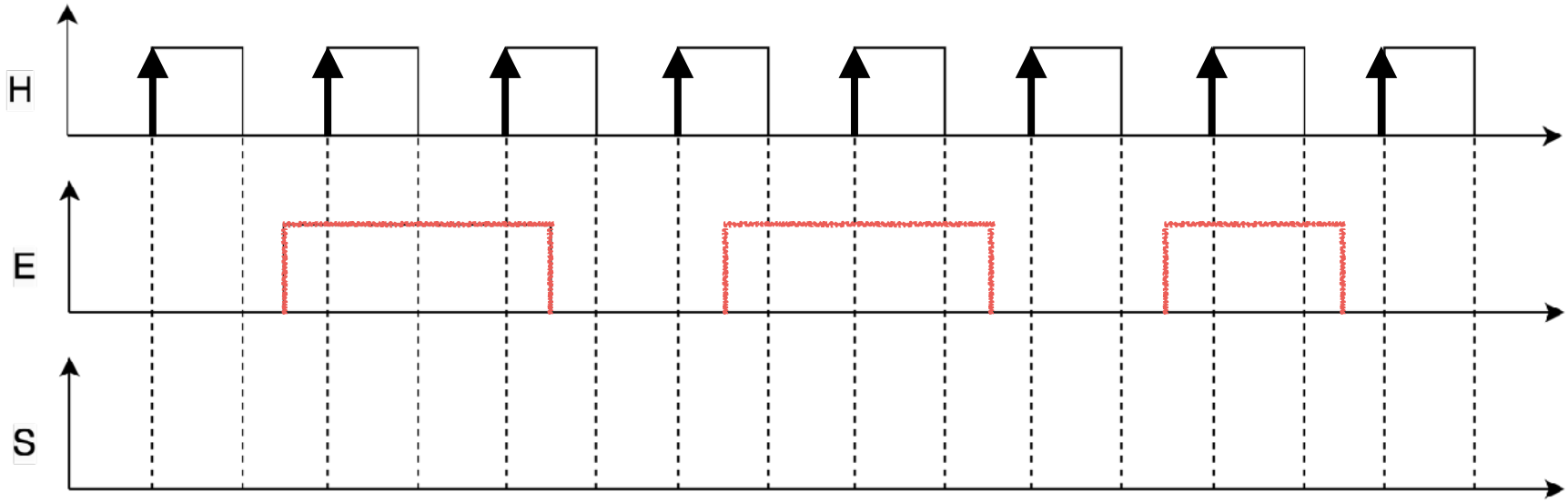
- The hardware blocks that store the data are flip-flops
- These elements are (generally) synchronized via a clock



Synchronous behavior (2)



E	H	S_{t+1}
x	0	Q_t
x	1	Q_t
0	↑	0
1	↑	1



Creating a synchronous element

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY reg_no_reset IS
    PORT (
        clock : IN  STD_LOGIC;
        e     : IN  STD_LOGIC;
        s     : OUT STD_LOGIC
    );
END reg_no_reset;

ARCHITECTURE behavior OF reg_no_reset IS
   SIGNAL mem : STD_LOGIC;
BEGIN
    PROCESS( clock )
    BEGIN
        IF clock'event and clock = '1' THEN
            mem <= e;
        END IF;
    END PROCESS;

    s <= mem;

END;
```

Synchronous elements are always declared in processes sensitive to clock signal

We specify that we expect a change in the signal state **AND** that the signal changes to '1'

This equals to a rising edge.

Be careful, we are **ONLY** synchronous on the clock !!!

Creating a synchronous element

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY reg_no_reset IS
    PORT (
        clock : IN  STD_LOGIC;
        e     : IN  STD_LOGIC;
        s     : OUT STD_LOGIC
    );
END reg_no_reset;

ARCHITECTURE behavior OF reg_no_reset IS
   SIGNAL mem : STD_LOGIC;
BEGIN
    PROCESS( clock )
    BEGIN
        IF clock'event and clock = '1' THEN
            mem <= e;
        END IF;
    END PROCESS;

    s <= mem;

END;
```

Two ways to declare a synchronous element i.e. a flip-flop

Equivalence - give the same hardware

```vhdl
ARCHITECTURE behavior OF reg_no_reset IS
   SIGNAL mem : STD_LOGIC;
BEGIN

    PROCESS( clock )
    BEGIN
        IF rising_edge(clock) THEN
            mem <= e;
        END IF;
    END PROCESS;

    s <= mem;
END;
```

Register with a synchronous reset

```
ENTITY reg_sync IS
    PORT (
         clock : IN  STD_LOGIC;
         reset : IN  STD_LOGIC;
         e     : IN  STD_LOGIC;
         s     : OUT STD_LOGIC
    );
END reg_sync;

ARCHITECTURE behavior OF reg_sync IS
   SIGNAL mem : STD_LOGIC;
BEGIN

    PROCESS( clock )
    BEGIN
       IF clock'event and clock = '1' THEN
          IF reset = '1' THEN
             mem <= '0';
          ELSE
             mem <= e;
          END IF;
       END IF;
    END PROCESS;

    s <= mem;
END;
```

Synchronous modules always use processes !

Register value only changes on rising edge of the clock

If reset signal equal '1' and a clock signal happens, then the register value is cleared

Be careful, we are **ONLY** synchronous on the clock !!!

Register with an asynchronous reset

```
ENTITY reg_async IS
    PORT (
        clock : IN  STD_LOGIC;
        reset : IN  STD_LOGIC;
        e     : IN  STD_LOGIC;
        s     : OUT STD_LOGIC
    );
END reg_async;

ARCHITECTURE behavior OF reg_async IS
  SIGNAL mem : STD_LOGIC;
BEGIN

    PROCESS( reset, clock )
    BEGIN
        IF reset = '1' THEN
            mem <= '0';
        ELSIF clock'event and clock = '1' THEN
            mem <= e;
        END IF;
    END PROCESS;

    s <= mem;
END;
```

Synchronous modules always use processes !

If reset signal equal '1' and even if no clock signal happens, the the register value is cleared

Register value changes on rising edge and reset equals '0'

Be careful, we are **ONLY** synchronous on the clock !!!

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 16-register-reset
  - Analyse the descriptions of the VHDL module
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Register with load signal

```
ENTITY reg_sync_load_8b IS
    PORT (
         clock : IN  STD_LOGIC;
         reset : IN  STD_LOGIC;
         load  : IN  STD_LOGIC;
         e     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         s     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END reg_sync_load_8b;

ARCHITECTURE behavior OF reg_sync_load_8b IS
  SIGNAL mem : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

    PROCESS( clock )
    BEGIN
         IF rising_edge( clock ) THEN
              IF reset = '1' THEN
                   mem <= (OTHERS => '0');
              ELSIF load = '1' THEN
                mem <= e;
              END IF;
         END IF;
    END PROCESS;

    s <= mem;
END;
```

Register value only changes on rising edge of the clock

If reset signal equal '1', clear the register value

If load signal equals '1' then the input value copied on output.

Otherwise, it keeps the value

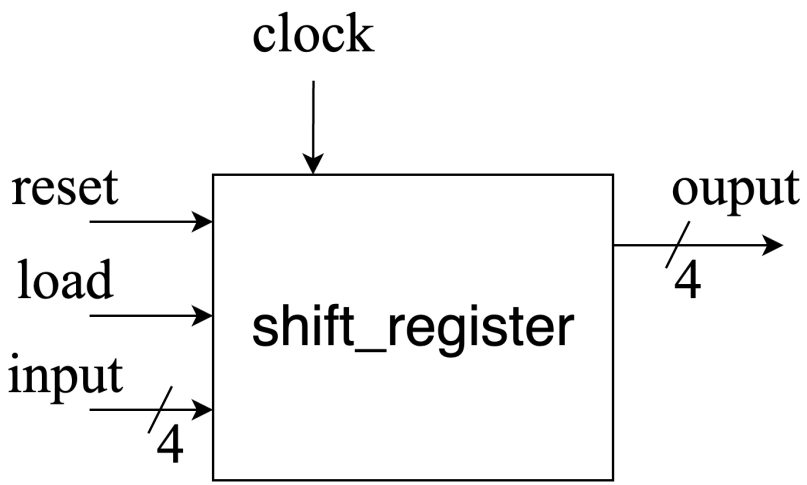
It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 17-register-reset-load
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

A 4-bit left shift register

- Let's design a simple sequential circuit
  - A 4-bit synchronous shift register
- Three possible states
  - **reset** = 1: register is cleared
  - **load** = 1: the 4-bit input is loaded in the register
  - **otherwise**: the internal value is left shifted by one bit by clock event
- The design is composed of at least 4 flip-flops
- Sounds like your 1st TP of « base du numérique » ;-)




```
ENTITY shift_register IS
  PORT (
    clock : IN  STD_LOGIC;
    reset : IN  STD_LOGIC;
    load  : IN  STD_LOGIC;
    e     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    s     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
  );
END shift_register;
```

It's time for me to take a break!

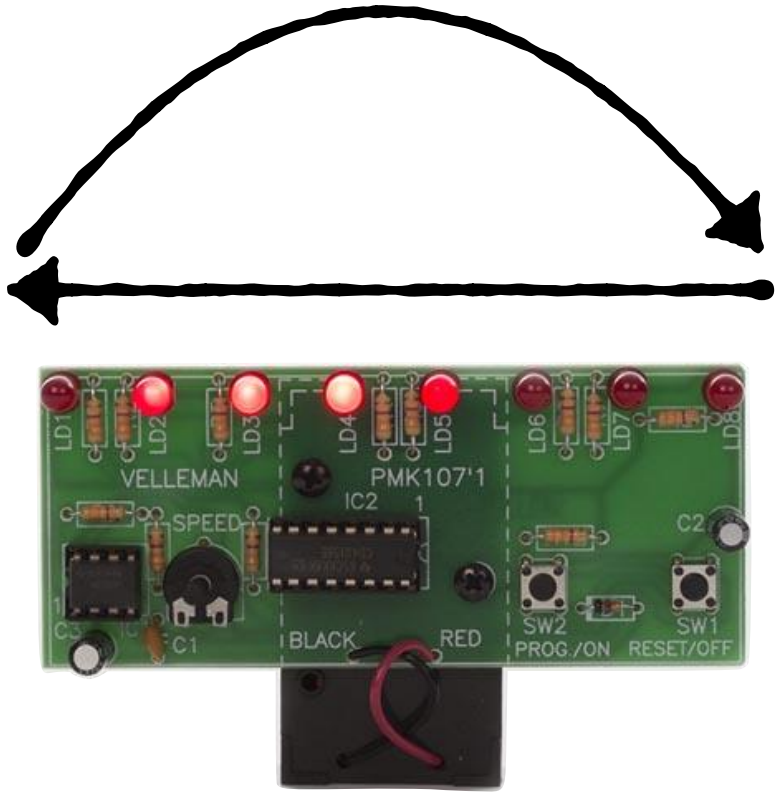
Analyze and simulate this module description

- Move to the example directory
  > cd 18-shift-register
- Write the VHDL description in the module
  > make vscode
- Simulate the module behavior
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

Let's create a sequential circuit (2)

- Update your design to realize a bit rotation
  - Let's consider 8-bit values
  - The 00000001 will be set on reset (no more load signal)
  - Left bit rotation
- For fun we will perform a FPGA implementation
  - DE-10 board (you master them)
  - So should be using in Windows software

rotation behavior



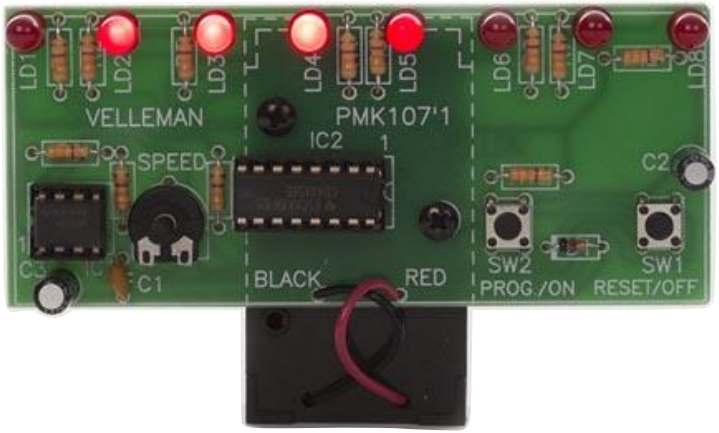
It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 19-chenillard
- Write the VHDL description in the module
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

Let's create a sequential circuit (3)

- Let's design a more complex circuit
  - K2000 based led animation
  - What is K2000 ?
- Design constraints
  - Let's consider 8-bit values
  - The 00000001 will be set on reset (no more load signal)
  - Once the led has reached an edge, the rotation direction changes
- For fun we will perform a FPGA implementation too



It's time for me to take a break!

Analyze and simulate this module description

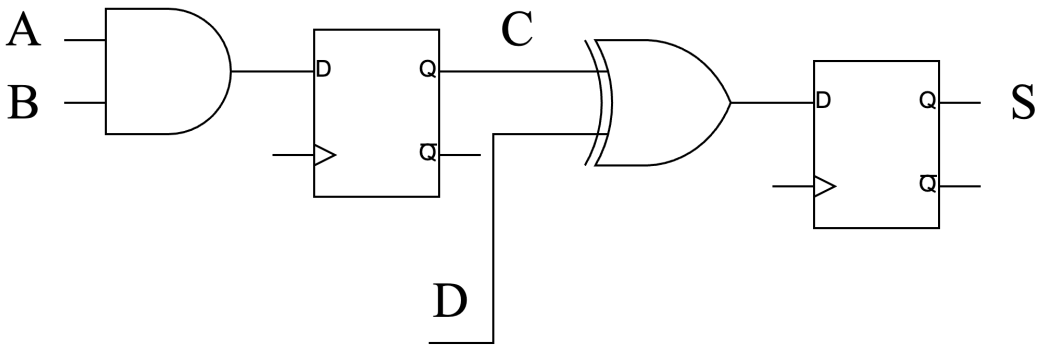
- Move to the example directory
  > cd 20-chenillard-k2000
- Write the VHDL description in the module
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

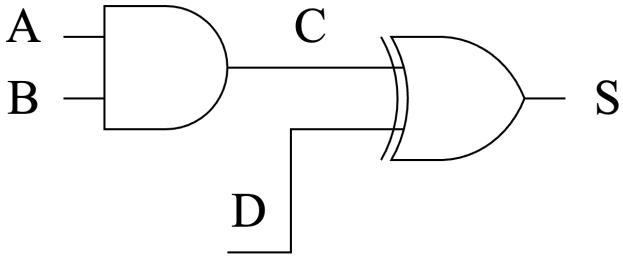
Signal implementations

Signal affectation in synchronous process produces flip-flop whereas in non synchronous process it generates wires

```
PROCESS( clock )
BEGIN
    IF rising_edge(clock) THEN
        C <= A AND B;
        S <= C XOR D;
    END IF;
END PROCESS;
```

```
PROCESS( A, B, C, D )
BEGIN
    C <= A AND B;
    S <= C XOR D;
END PROCESS;
```

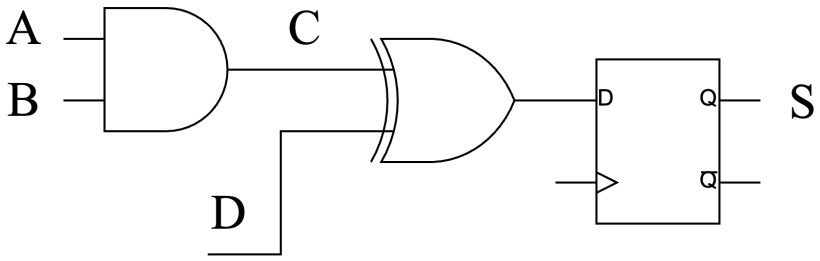




The process variables

- Variable keyword is available in processes
  - Used to declare local data
- Managed differently to signals
  - Only available in the process
  - Does not appears in sensibility list
  - Affectation is done with := operator
  - Immediate assignment
- Always implemented as a wire in the design

```
PROCESS( clock )
    VARIABLE C : STD_LOGIC;
BEGIN
    IF rising_edge(clock) THEN
        C := A AND B;
        S <= C XOR D;
    END IF;
END PROCESS;
```



ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[6 - RTL and Logic Synthesis]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

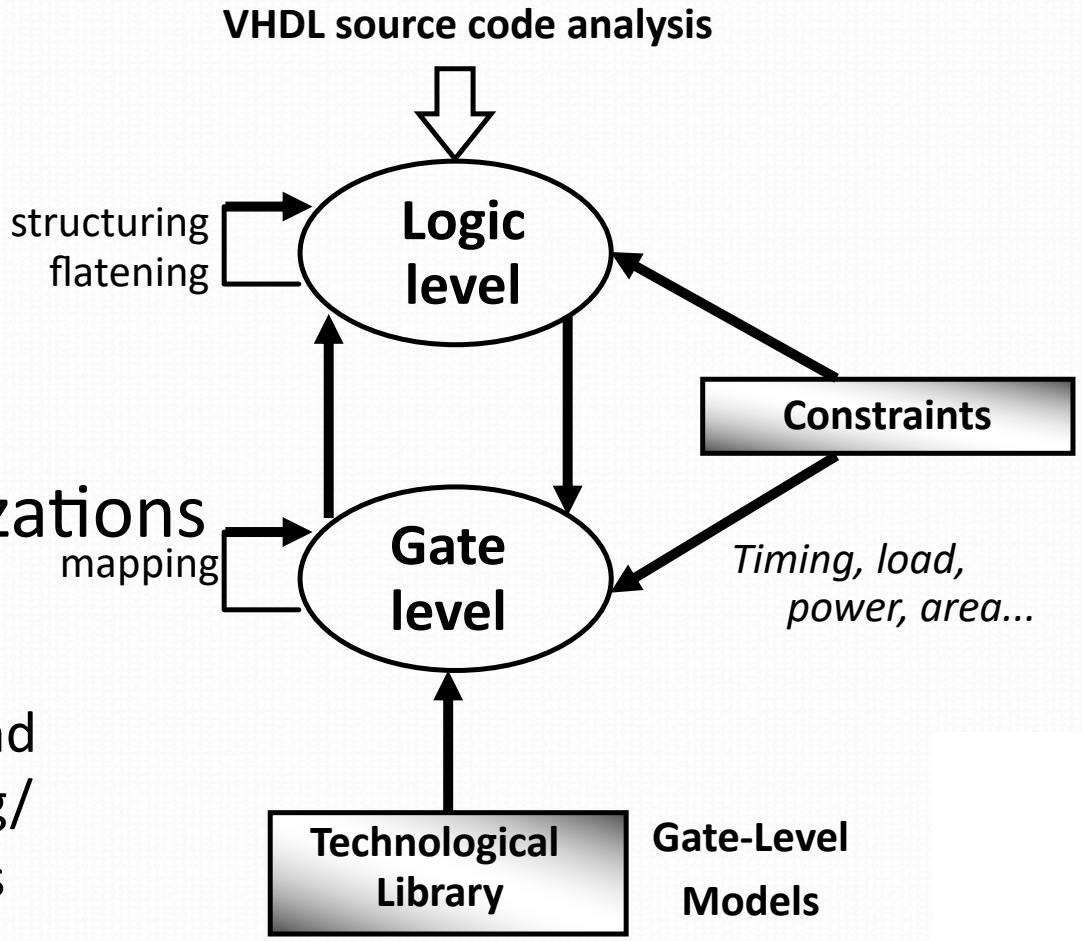
bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

Logical synthesis (ASIC)

- Logic Synthesis = Translation + Optimizations
- Translation
  - VHDL source code is analyzed and transformed into a generic logic representation
- Constrained optimizations
  - Logic is mapped onto gates from the technological library and optimized under timing/ power/area constraints

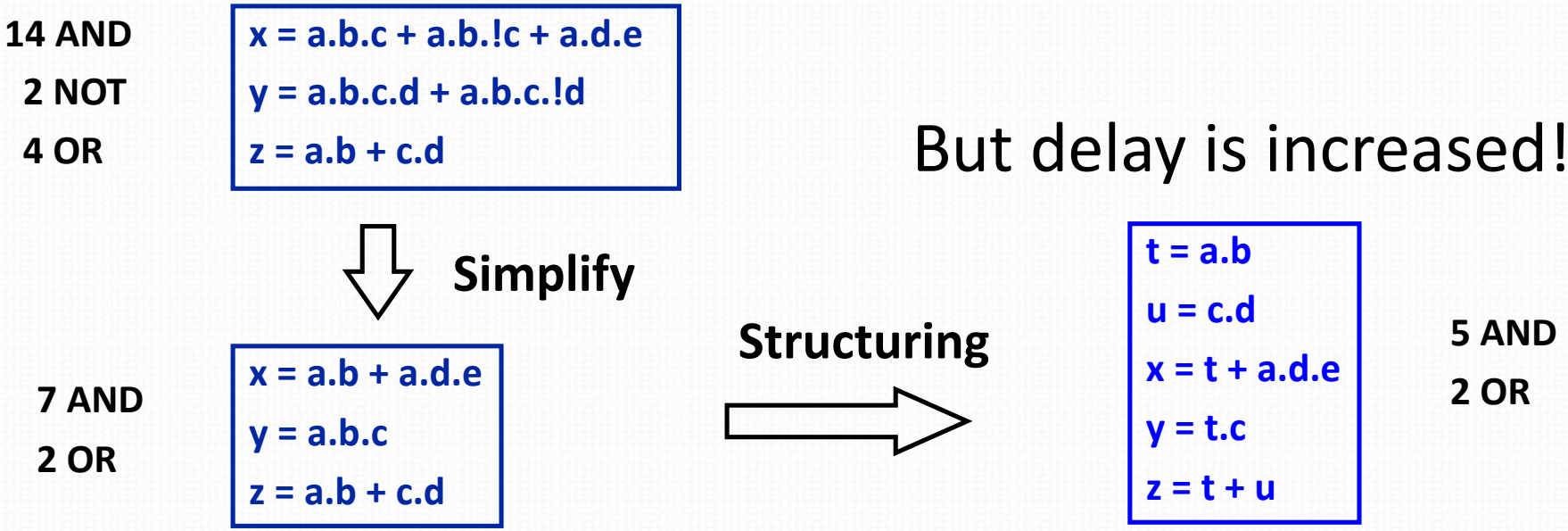


Optimizations

- Structuring: area oriented
  - Simplify and factorize Boolean equations
- Flattening: speed oriented
  - Distribute factors in Boolean equations
- Mapping
  - Selection of gates in the technological library
  - Specification of constraints (power/delay/load/area)
  - Generate several solutions

Structuring

- Structuring: area oriented
  - Simplify and factorize Boolean equations
  - Share common factors

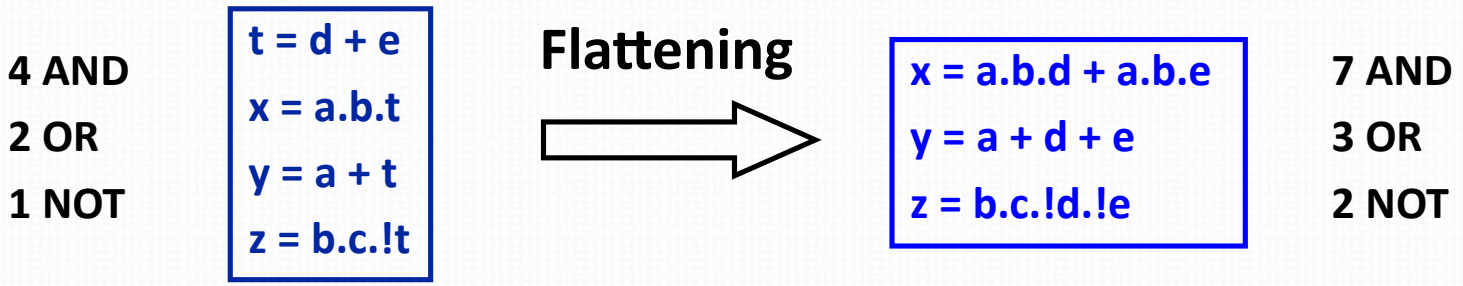


But delay is increased!

Flattening

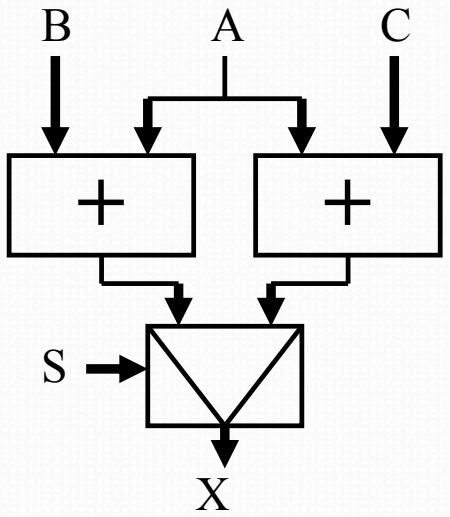
- Flattening: speed oriented
  - Distribute factors in Boolean equations
  - Reduces delay

Delay is decreased!

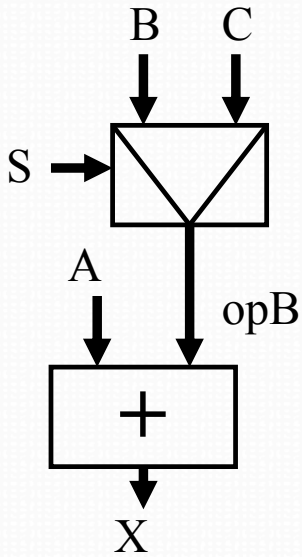


Ressource sharing

```
process (A,B,C,S)
begin
    if (S = ' 1 ') then
        X <= A + B ;
    else
        X <= A + C ;
    end if;
end process;
```




```
process (A,B,C,S)
    variable opB : integer;
begin
    if (S = ' 1 ') then
        opB :=  B ;
    else
        opB := C ;
    end if;
    X <=  A + opB;
end process;
```



ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[7 - Logic and Arithmetic]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

VHDL data types

- In VHDL, there are many types of data, especially in VHDL'14 standard release
- Predefined types in the language
  - BIT, BOOLEAN, BIT_VECTOR, INTEGER,
- Types defined in the package IEEE.STD_LOGIC_1164
  - STD_LOGIC, STD_LOGIC_VECTOR,
- Types defined in the package IEEE.NUMERIC_STD
  - SIGNED, UNSIGNED
- Typically, you will only manipulate the following types
  - STD_LOGIC, STD_LOGIC_VECTOR, SIGNED, UNSIGNED, INTEGER

The STD_LOGIC type

- The BIT type defined in the language is used to represent {0, 1}.
  - Enough to describe digital architectures
  - Not enough to simulate
- The STD_LOGIC type meets this need
  - Uninitialized value (U)
  - Error value (X)
  - Don't care value (-)
    - Interesting way of telling the synthesizer that you don't care

Value	Interpretation
U	Uninitialized
X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don't Care

The STD_LOGIC_VECTOR type

- It intended for modeling binary data groups
  - SIGNAL A : STD_LOGIC_VECTOR(31 DOWNTO 0) := "0101.....1101";
  - SIGNAL B : STD_LOGIC_VECTOR(31 DOWNTO 0) := x"89ABCDEF";
- It is possible to extract subsets of bits from variables
  - SIGNAL C : STD_LOGIC_VECTOR(15 DOWNTO 0) := A(23 DOWNTO 8);
  - SIGNAL D : STD_LOGIC := A(16);
- And to combine binary information to create vectors
  - SIGNAL E : STD_LOGIC_VECTOR(31 DOWNTO 0) := C & C;
  - SIGNAL F : STD_LOGIC_VECTOR(4 DOWNTO 0) := D & '0' & D & '1';
- Logic operations are usable
  - SIGNAL G : STD_LOGIC_VECTOR(15 DOWNTO 0) := A(31 DOWNTO 15) OR (NOT C);
  - SIGNAL H : STD_LOGIC := A(7 DOWNTO 6) /= (D & D);

UNISIGNED and SIGNED types

- STD_LOGIC_VECTOR represents bit-vector without numerical properties
- Data types are dedicated to arithmetic calculations
  - The **IEEE.NUMERIC_STD** library must be included in your VHDL modules
- For unsigned data
  - **UNSIGNED**(msb **DOWNTO** 0) => manage values in range [0, $+2^{MSB}-1$]
  - **SIGNAL** CPT : **UNSIGNED**(10 **DOWNTO** 0) := **TO_UNSIGNED**(0, 11);
- For signed data (two's complement)
  - **SIGNED**(msb **DOWNTO** 0) => manage values in range [-2^{MSB-1}, $+2^{MSB-1}-1$]
  - **SIGNAL** CPT : **SIGNED**(10 **DOWNTO** 0) := **TO_SIGNED**(0, 11);

Conversions between data types

- Consider a STD_LOGIC_VECTOR
  - VARIABLE STDL : STD_LOGIC_VECTOR(7 DOWNTO 0) := "1000 0001";
- it is necessary to convert it to SIGNED or UNSIGNED in order to do arithmetic computations
  - VARIABLE UNS : UNSIGNED(7 DOWNTO 0) := UNSIGNED(STDL);
    - The unsigned variable UNS contains the numerical value 128.
  - VARIABLE SIG : SIGNED(7 DOWNTO 0) := SIGNED(STDL);
    - The signed variable (2's complement) SIG contains the numerical value -63.
- The opposite conversion is just as trivial
  - VARIABLE DEST1 : STD_LOGIC_VECTOR(7 DOWNTO 0) := STD_LOGIC_VECTOR(UNS);
  - VARIABLE DEST2 : STD_LOGIC_VECTOR(7 DOWNTO 0) := STD_LOGIC_VECTOR(SIG);
    - DEST1 & DEST2 equal "1000 0001" binary value

Computing an addition with cast (1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_8b_v1 IS
   PORT (
       A :  IN std_logic_vector(7 downto 0);
       B :  IN std_logic_vector(7 downto 0);
       C : OUT std_logic_vector(7 downto 0)
   );
END add_8b_v1;

ARCHITECTURE arch OF add_8b_v1 IS
BEGIN

   PROCESS (A, B)
   BEGIN
       C <= STD_LOGIC_VECTOR( UNSIGNED(A) + UNSIGNED(B) );
   END PROCESS;

END arch;
```

Inputs are casted, then summed and finally casted again

Computing an addition with cast (2)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_8b_v2 IS
    PORT (
        A : IN std_logic_vector(7 downto 0);
        B : IN std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(7 downto 0)
    );
END add_8b_v2;

ARCHITECTURE arch OF add_8b_v2 IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : UNSIGNED(7 downto 0);
        VARIABLE E : UNSIGNED(7 downto 0);
    BEGIN
        D := UNSIGNED(A);
        E := UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( D + E );
    END PROCESS;

END arch;
```

The addition process can be broken down to make it easier to understand

Declaration of intermediate variables

Typing unsigned input signals

Adding up data

Addition with carry extension (1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_9b_v1 IS
    PORT (
        A :  IN std_logic_vector(7 downto 0);
        B :  IN std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(8 downto 0)
    );
    END add_9b_v1;
ARCHITECTURE arch OF add_9b_v1 IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : UNSIGNED(8 downto 0);
        VARIABLE E : UNSIGNED(8 downto 0);
    BEGIN
        D := '0' & UNSIGNED(A);
        E := '0' & UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( D + E );
    END PROCESS;

END arch;
```

To obtain the carry bit, it is necessary to manually extend the operands

Intermediate data has one more bit than inputs

Data are unsigned, so add a zero bit to the signal

The addition operation now takes place on 9 bits (MSB = carry / overflow)

Addition with carry extension (2)

```
PROCESS (A, B)
   VARIABLE D : UNSIGNED(7 downto 0);
   VARIABLE E : UNSIGNED(7 downto 0);
BEGIN
   D := UNSIGNED(A);
   E := UNSIGNED(B);
   C <= STD_LOGIC_VECTOR( RESIZE(D, 9) + RESIZE(E, 9) );
END PROCESS;
```

The RESIZE function in the NUMERIC_STD library is used to resize data.

```
PROCESS (A, B)
   VARIABLE D : SIGNED(7 downto 0);
   VARIABLE E : SIGNED(7 downto 0);
 BEGIN
   D := SIGNED(A);
   E := SIGNED(B);
   C <= STD_LOGIC_VECTOR( RESIZE(D, 9) + RESIZE(E, 9) );
END PROCESS;
```

The same applies to SIGNED data, but what does it actually do?

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 24-adders
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 24

The multiplication operation

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY mul_16b_8b_8b IS
     PORT (
          A : IN  std_logic_vector( 7 downto 0);
          B : IN  std_logic_vector( 7 downto 0);
          C : OUT std_logic_vector(15 downto 0)
     );
END mul_16b_8b_8b;

ARCHITECTURE arch OF mul_16b_8b_8b IS
BEGIN

     PROCESS (A, B)
          VARIABLE D : unsigned(7 downto 0);
          VARIABLE E : unsigned(7 downto 0);
     BEGIN
          D := UNSIGNED(A);
          E := UNSIGNED(B);
          C <= STD_LOGIC_VECTOR( D * E );
     END PROCESS;

END arch;
```

The result is coded on $P \times Q$ bits if its operands are on P and Q bits

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY mul_8b_8b_8b IS
     PORT (
          A : IN  std_logic_vector(7 downto 0);
          B : IN  std_logic_vector(7 downto 0);
          C : OUT std_logic_vector(7 downto 0)
     );
END mul_8b_8b_8b;

ARCHITECTURE arch OF mul_8b_8b_8b IS
BEGIN

     PROCESS (A, B)
          VARIABLE D : unsigned(15 downto 0);
          VARIABLE E : unsigned( 7 downto 0);
     BEGIN
          D := UNSIGNED(A) * UNSIGNED(B);
          E := D(7 downto 0);
          C <= STD_LOGIC_VECTOR( E );
     END PROCESS;

END arch;
```

Truncating the result of the operation is required to limit dynamic expansion

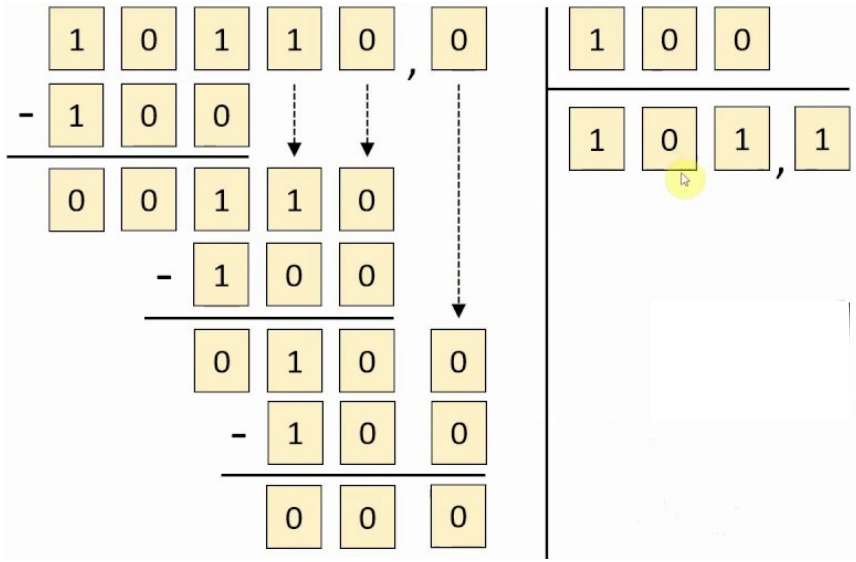
It's time for me to take a break!

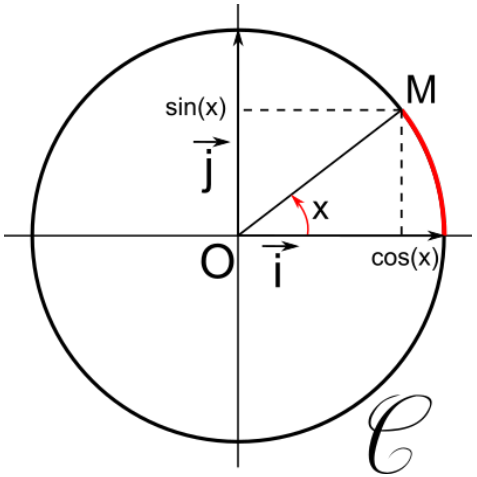
Analyze and simulate this module description

- Move to the example directory
  > cd 25-multipliers
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

The others mathematical operations

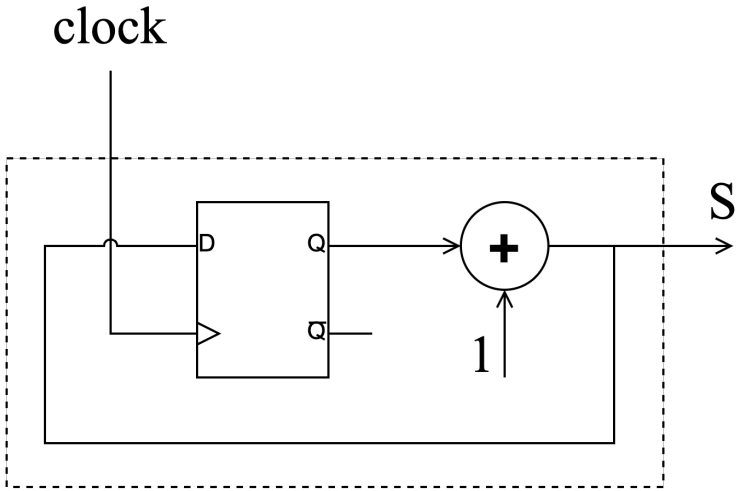
- Many other arithmetic operations
  - Division & modulus
  - Square root
  - Trigonometric computations
  - IEEE 754 operations
- Their are not basic circuits
  - Many algorithms
  - Various way to design them (combinatory, clocked, etc.)
- Some library provide simulation support





Describing counters

- With arithmetic operators we can build other modules
- Counter structure is useful in many systems
  - Counting events
  - Generating timers
  - etc.
- Hardware design
  - An adder
  - A register
  - A comparator



The simplest counter architecture (power 2)

Your first modulo counter

```
ENTITY counter IS
  PORT (
    CLK : in  STD_LOGIC;
    RST : in  STD_LOGIC;
    O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
  );
END counter;

ARCHITECTURE Behavioral OF counter IS
  SIGNAL COUNTER : UNSIGNED(7 DOWNTO 0);
BEGIN

  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      COUNTER <= TO_UNSIGNED( 0, 8 );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      IF COUNTER = TO_UNSIGNED( 255, 8 ) THEN
        COUNTER <= TO_UNSIGNED( 0, 8 );
      ELSE
        COUNTER <= COUNTER + TO_UNSIGNED( 1, 8 );
      END IF;
    END IF;
  END PROCESS;

  O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

The internal signal is UNSIGNED

The signal is initialized to zero (coded on 8 bits)

The value is compared with the maximum limit

Incrementing the counter value

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 26-counter-unsigned-if
- Open the VHDL description of the VHDL counter and complete it
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 26

Your second modulo counter

```
ENTITY counter IS
  PORT (
    CLK : in  STD_LOGIC;
    RST : in  STD_LOGIC;
    O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
  );
END counter;

ARCHITECTURE Behavioral OF counter IS
  SIGNAL COUNTER : UNSIGNED(7 DOWNTO 0);
BEGIN

  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      COUNTER <= TO_UNSIGNED( 0, 8 );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      COUNTER <= COUNTER + TO_UNSIGNED( 1, 8 );
    END IF;
  END PROCESS;

  O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Coded on 8 bits, the internal counter value automatically resets to zero

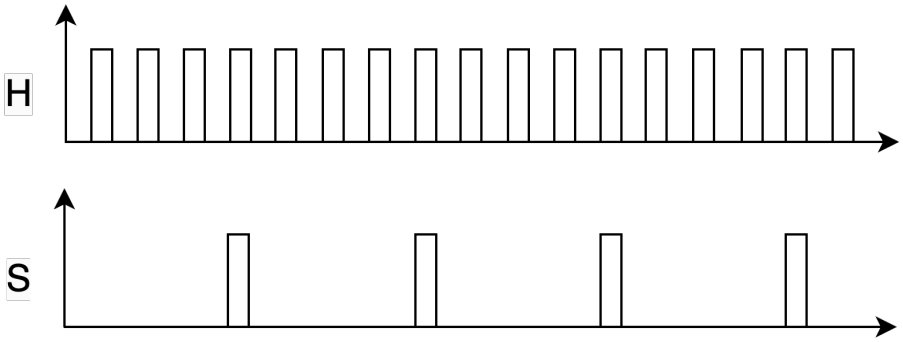
It's time for me to take a break!

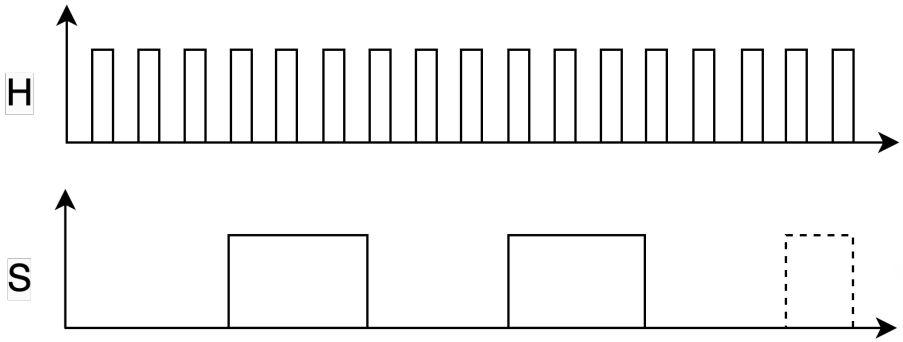
- Analyze and simulate this module description
  - Move to the example directory
    > cd 27-counter-unsigned
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

HOMEWORK 27

Signal generators

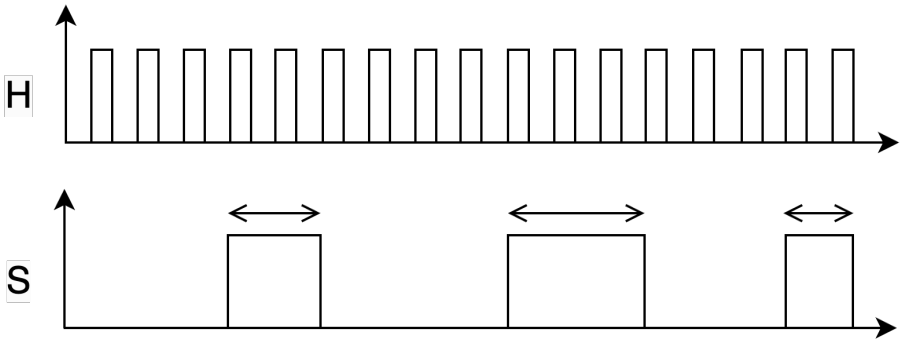
Chip Enable (CE) generation





Clock frequency divider

Pulse Width Modulation (PWM) generation



Clock frequency divider

```
ARCHITECTURE Behavioral OF clock_divider IS
  SIGNAL COUNTER : UNSIGNED(2 DOWNTO 0);
  SIGNAL OUT_V   : STD_LOGIC;
BEGIN

  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      OUT_V   <= '0';
      COUNTER <= TO_UNSIGNED( 0, 3 );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      IF COUNTER = TO_UNSIGNED( 7, 3 ) THEN
        COUNTER <= TO_UNSIGNED( 0, 3 );
        OUT_V   <= NOT OUT_V;
      ELSE
        COUNTER <= COUNTER + TO_UNSIGNED( 1, 3 );
      END IF;
    END IF;
  END PROCESS;

  O <= OUT_V;

END Behavioral;
```

The module divide the clock frequency by 8

The state of the flip-flop is inverted when the counter loops back

The internal flip-flop connected to output

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 28-clock-divider
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

HOMEWORK 28

Chip Enable generation

```
ARCHITECTURE Behavioral OF chip_enable IS
  SIGNAL COUNTER : UNSIGNED(2 DOWNTO 0);
  SIGNAL OUT_V   : STD_LOGIC;
BEGIN

  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      OUT_V   <= '0';
      COUNTER <= TO_UNSIGNED( 0, 3 );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      IF COUNTER = TO_UNSIGNED( 7, 3 ) THEN
        COUNTER <= TO_UNSIGNED( 0, 3 );
        OUT_V   <= '1';
      ELSE
        COUNTER <= COUNTER + TO_UNSIGNED( 1, 3 );
        OUT_V   <= '0';
      END IF;
    END IF;
  END PROCESS;

  O <= OUT_V;

END Behavioral;
```

The module generates a pulse equal to one clock cycle

The state of the flip-flop is set to 1 when the counter is reset to zero

The rest of the time, output is zero

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 29-chip-enable-genration
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

PWM signal generator

```
ENTITY pwm_generation IS
PORT (
  CLK : in  STD_LOGIC;
  RST : in  STD_LOGIC;
  LMT : in  STD_LOGIC_VECTOR(2 DOWNTO 0);
  O   : out STD_LOGIC
);
END pwm_generation;

ARCHITECTURE Behavioral OF pwm_generation IS
  SIGNAL COUNTER : UNSIGNED(2 DOWNTO 0);
BEGIN

  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      COUNTER <= TO_UNSIGNED( 0, 3 );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      COUNTER <= COUNTER + TO_UNSIGNED( 1, 3 );
    END IF;
  END PROCESS;

  O <= '1' WHEN COUNTER < UNSIGNED(LMT) ELSE '0';

END Behavioral;
```

The module generates a pulse whose width is input controlled

A simple modulo 8 counter

The output of the component is thresholded

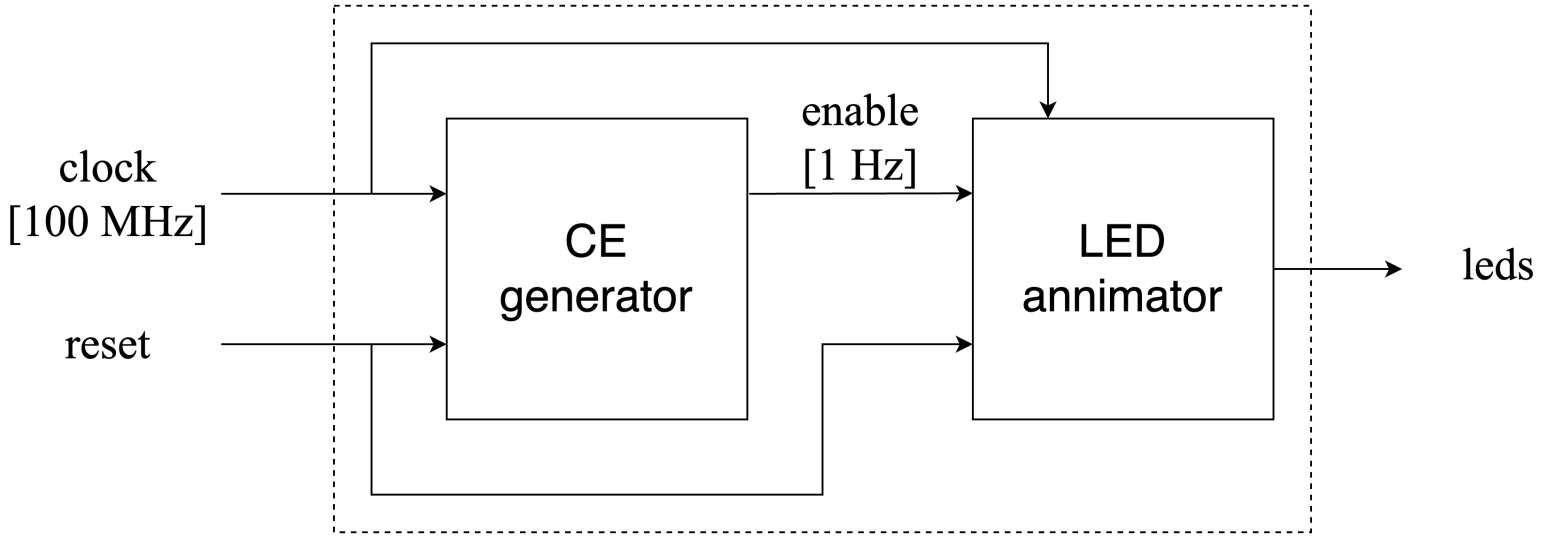
It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 30-pwm-genration
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

Clean k2000 system design

Chip Enable generation

- Authorize to have a single clock in designs
- Easier to design & to debug
- However, solutions exist to manage different clock domain



It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 31-chenillard-k2000-clean
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

The INTEGER data type

- INTEGER type simplify the writing of arithmetic computations
  - It's more designer-friendly, however, less precise for implementation
  - An INTEGER data type (without range) can represent a number in the **[-2^{31}, +2^{31}-1]** interval
  - **SIGNAL** the_value : **INTEGER** = 73;
- The INTEGER type is often declared with a specification of its variation interval
  - **SIGNAL** the_value_p : **INTEGER RANGE** 0 **TO** 124 := 24;
  - **SIGNAL** the_value_n : **INTEGER RANGE** -24 **TO** 13 := -4;
- Conversion from other/to SIGNED/UNSIGNED
  - **SIGNAL** the_value_u : **UNSIGNED**(7 DOWNTO 0) := **TO_UNSIGNED**(the_value_p, 8);
  - **SIGNAL** the_value_i : **INTEGER** := **TO_INTEGER**(the_value_u);

Your third modulo counter

```
ENTITY counter IS
     PORT (
          CLK : in  STD_LOGIC;
          RST : in  STD_LOGIC;
          O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
     );
END counter;

ARCHITECTURE Behavioral OF counter IS
     SIGNAL COUNTER : INTEGER RANGE 0 TO 255;
BEGIN

     PROCESS(RST, CLK)
     BEGIN
          IF RST = '1' THEN
               COUNTER <= 0;
          ELSIF CLK = '1' AND CLK'EVENT THEN
               IF COUNTER = 255 THEN
                    COUNTER <= 0;
               ELSE
                    COUNTER <= COUNTER + 1;
               END IF;
          END IF;
     END PROCESS;

     O <= STD_LOGIC_VECTOR( TO_UNSIGNED(COUNTER, 8) );

END Behavioral;
```

Easier handling of digital data

Testing with the maximum value is mandatory, otherwise it will generate simulation errors

The integer data must be converted to 8 bits for output (double conversion)

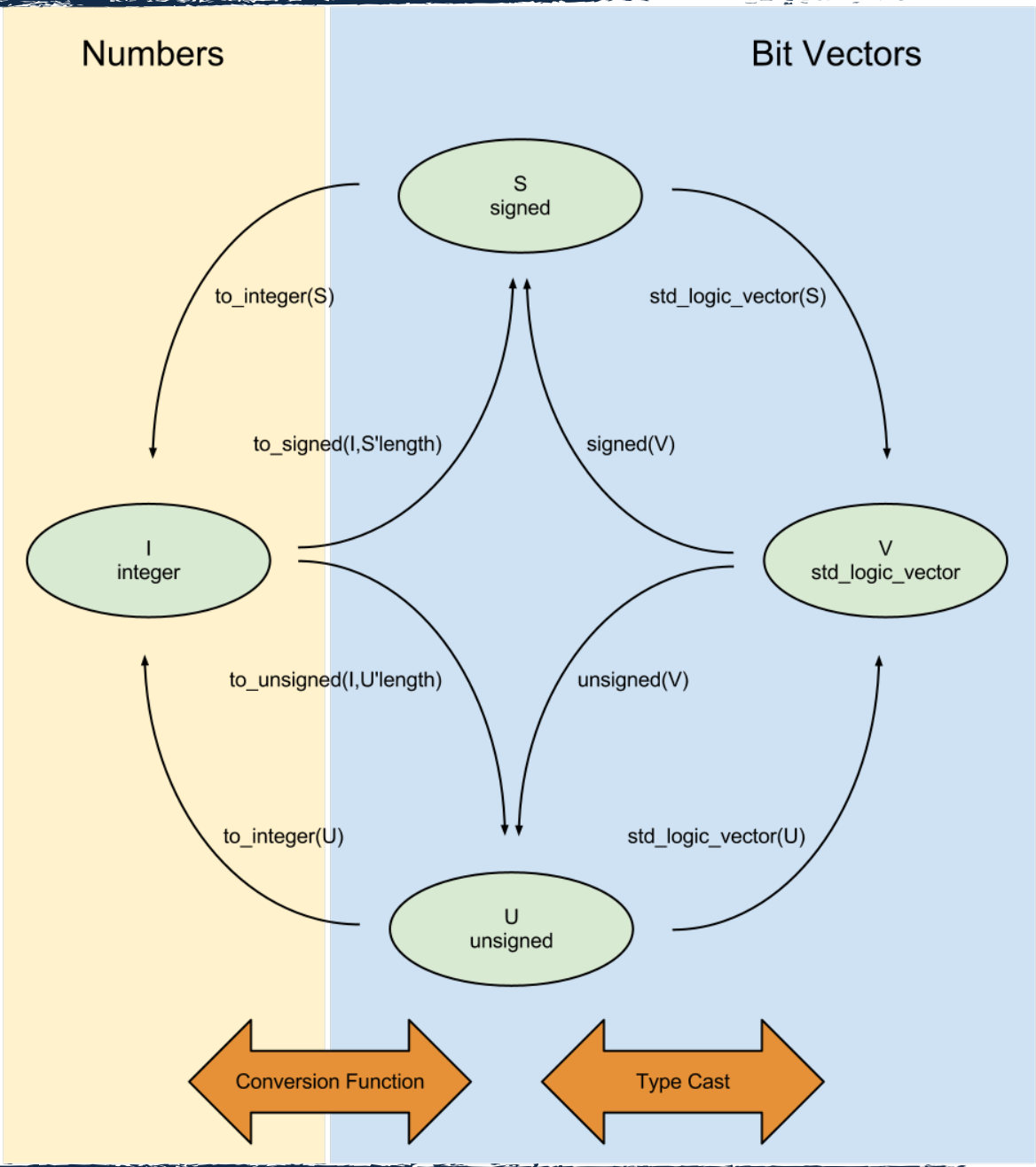
It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 32-counter-integer
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

The conversion functions

- Converting data types is quite easy
  - You just need to remember the function names ;-)
- Integers are different
  - UNSIGNED( )
  - SIGNED( )
  - STD_LOGIC_VECTOR( )
  - TO_INTEGER( )



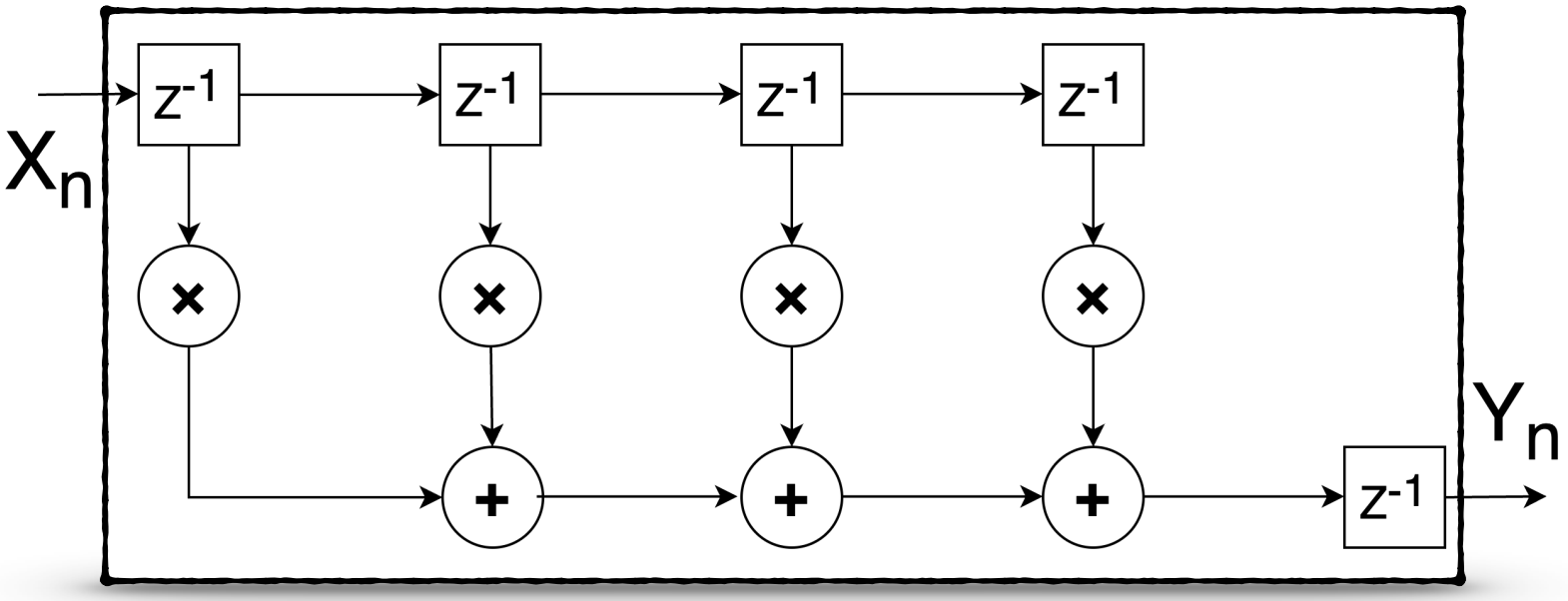
FIR filters

- Arithmetic operators can be used for parallel computing
  - Not limited to address counters ;-)
- FIR filtering
  - Massively used in audio applications
  - Made FPGA essential during 90's
  - Requires additions and multiplications
  - Various architectural solutions (trade-off: complexity & throughput)
- Fixed-point processing
  - Do you have the basis ?

$$y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k)$$

FIR filter structure

$$y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k)$$



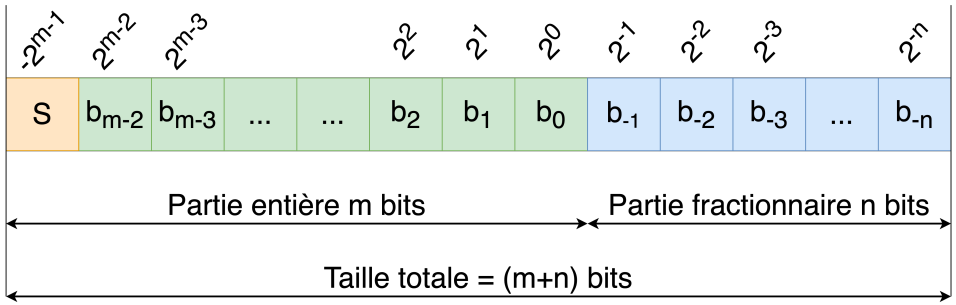
FIR filter coefficients

Coefficient	Fractional value
1	-0028094547010435022
2	+0.000000000000000007
3	+0.042141820515652538
4	+0.090915863922831200
5	+0.136373795884246835
6	+0.168567282062610097
7	+0.180191569250188838
8	+0.168567282062610097
9	+0.136373795884246835
10	+0.090915863922831200
11	+0.042141820515652538
12	+0.000000000000000007
13	-0.028094547010435022

Fixed-point coding

- Negative powers are used to manage fractional parts
- You need to know where the decimal point is (mental view)
- $Q_{n.m}$ format
  - (m+n) bits in total
  - m bits for the integer part
  - n bits for the fractional part
  - the resolution is $q = 2^{-n}$
- Arithmetic operations are unchanged

Power	Value
2^0	1
2^{-1}	0,5
2^{-2}	0,25
2^{-3}	0,125
2^{-4}	0,0625
2^{-5}	0,03125
2^{-6}	0,015625
2^{-7}	0,0078125
2^{-8}	0,00390625



Fixed-point coefficients

Coefficient	Fractional value	Fixed point (8 bits)
1	-0.028094547010435022	-3
2	+0.000000000000000007	0
3	+0.042141820515652538	5
4	+0.090915863922831200	11
5	+0.136373795884246835	17
6	+0.168567282062610097	21
7	+0.180191569250188838	22
8	+0.168567282062610097	21
9	+0.136373795884246835	17
10	+0.090915863922831200	11
11	+0.042141820515652538	5
12	+0.000000000000000007	0
13	-0.028094547010435022	-3

8 bits means here one sign bit and 7 fractional bits

Don't forget to divide the FIR result by 128 !

floor(data x127)

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 33-fir-comb
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

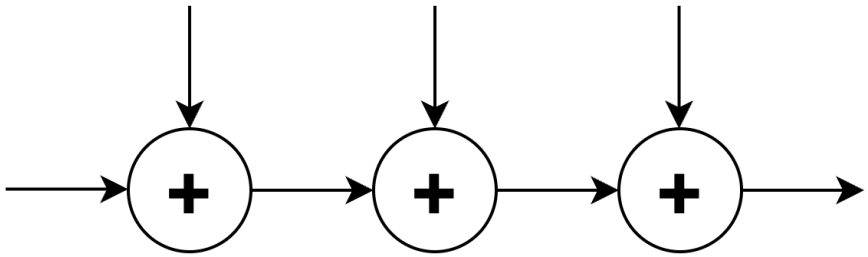
HOMEWORK 33

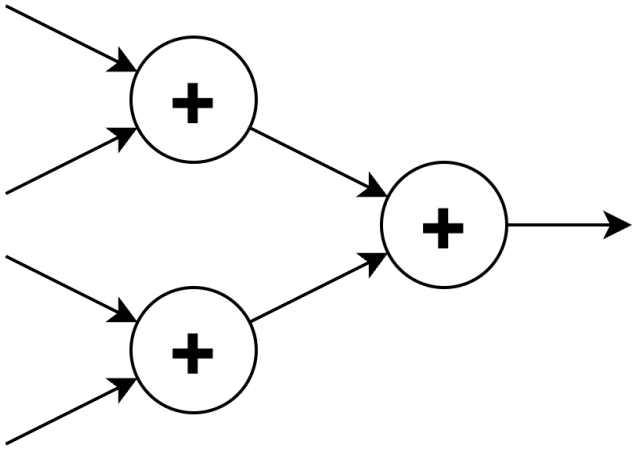
Simulation results

Design compiler result analysis

What you describe is what you get

$$S = T_1 + T_2 + T_3 + T_4$$

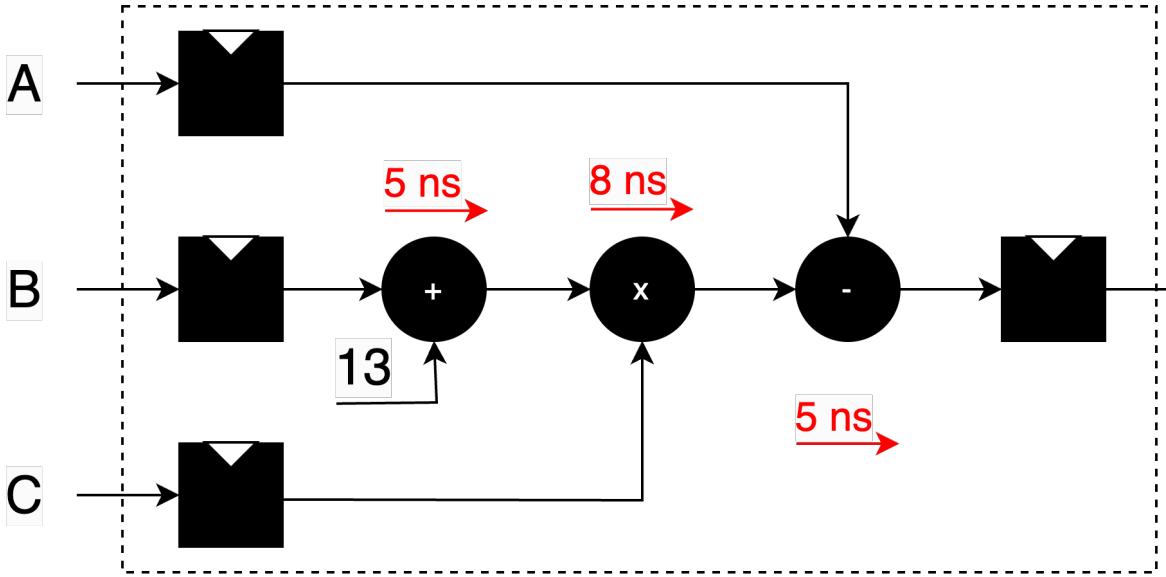




$$S = (T_1 + T_2) + (T_3 + T_4)$$

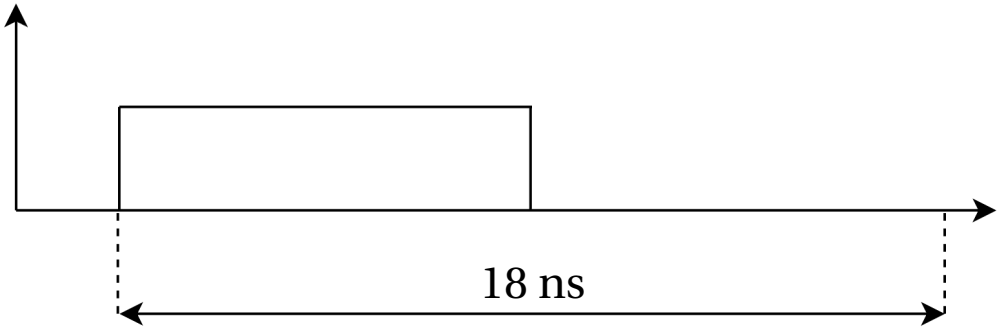
Design compiler result analysis

Introduction to pipelining (1)

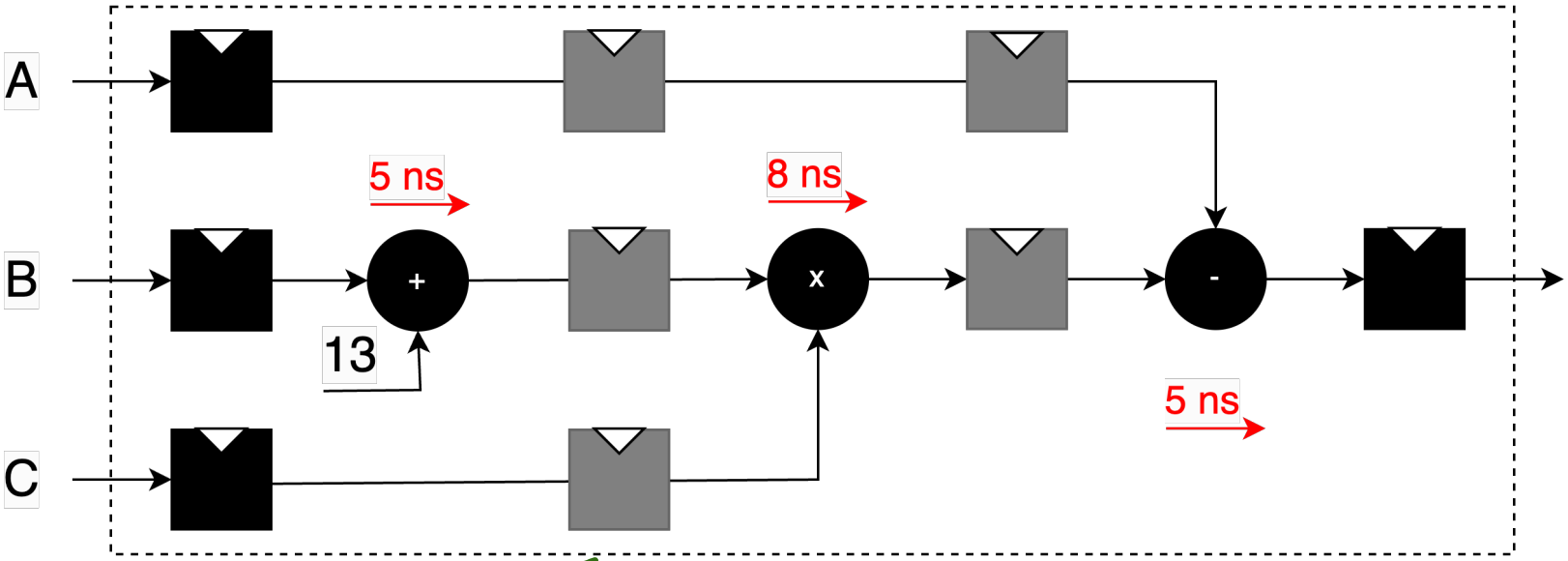


The computation result (D) need two clock cycles to be produced

The circuit critical path is equal to 18 ns meaning a working frequency = 55 MHz

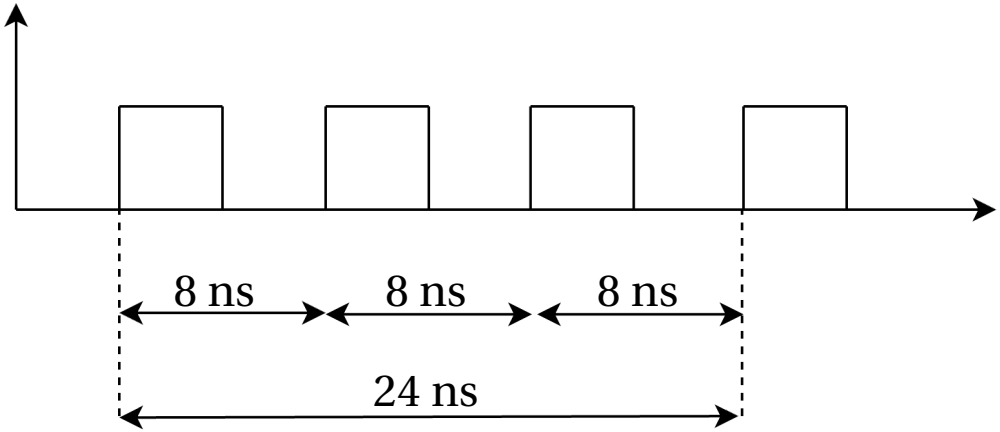


Introduction to pipelining (2)



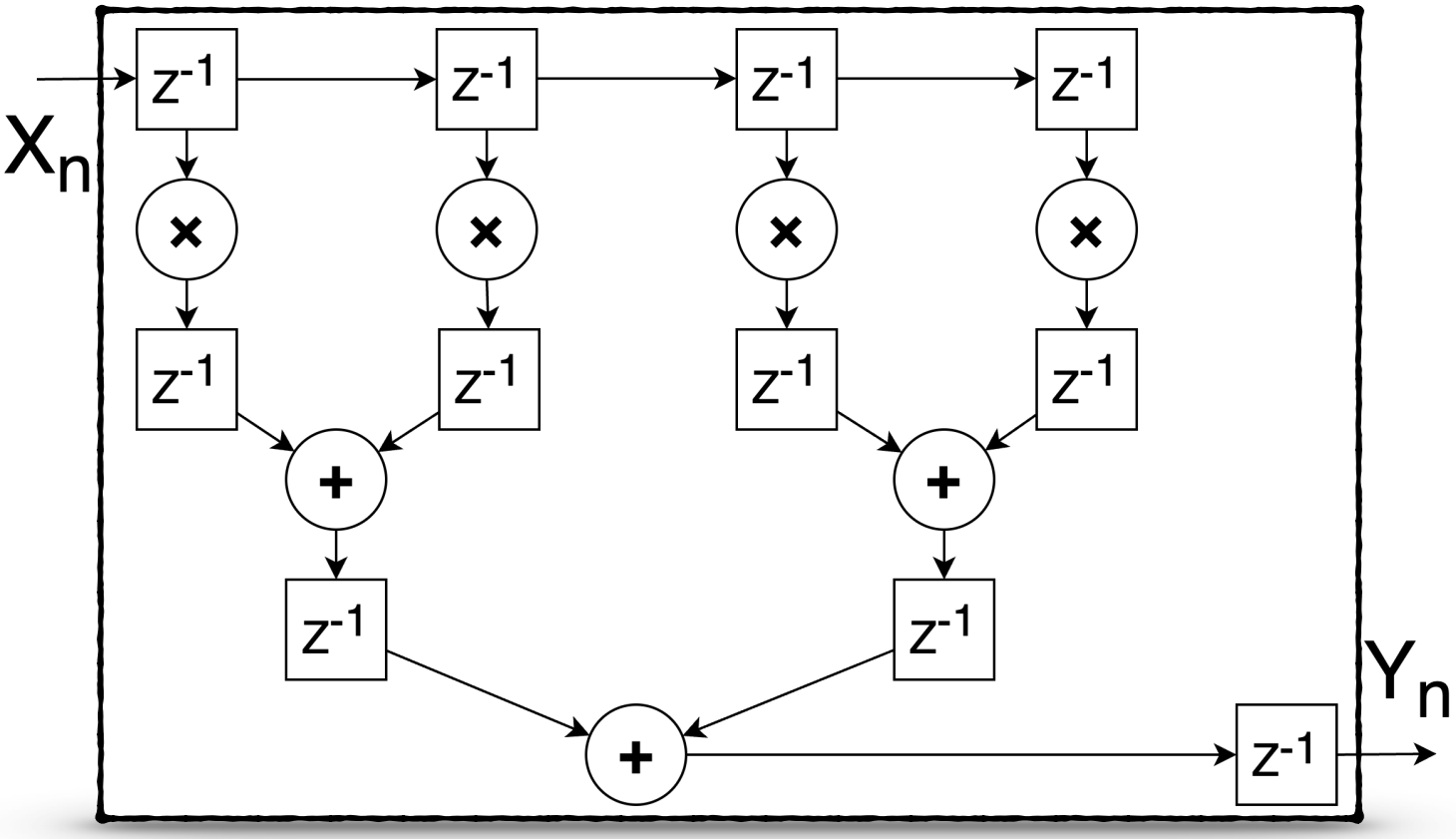
D computation needs three clock cycles

The circuit critical path is equal to MAX(8, 5, 5) = 8 ns meaning a working frequency = 125 MHz



Pipelined FIR filter structure

- To improve working frequency we should break the critical paths
  - Add some registers between operators
  - Make sure you balance the registers in the different paths



It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 34-fir-pipeline
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

Design compiler result analysis

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[8 - VHDL for Flexibility]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

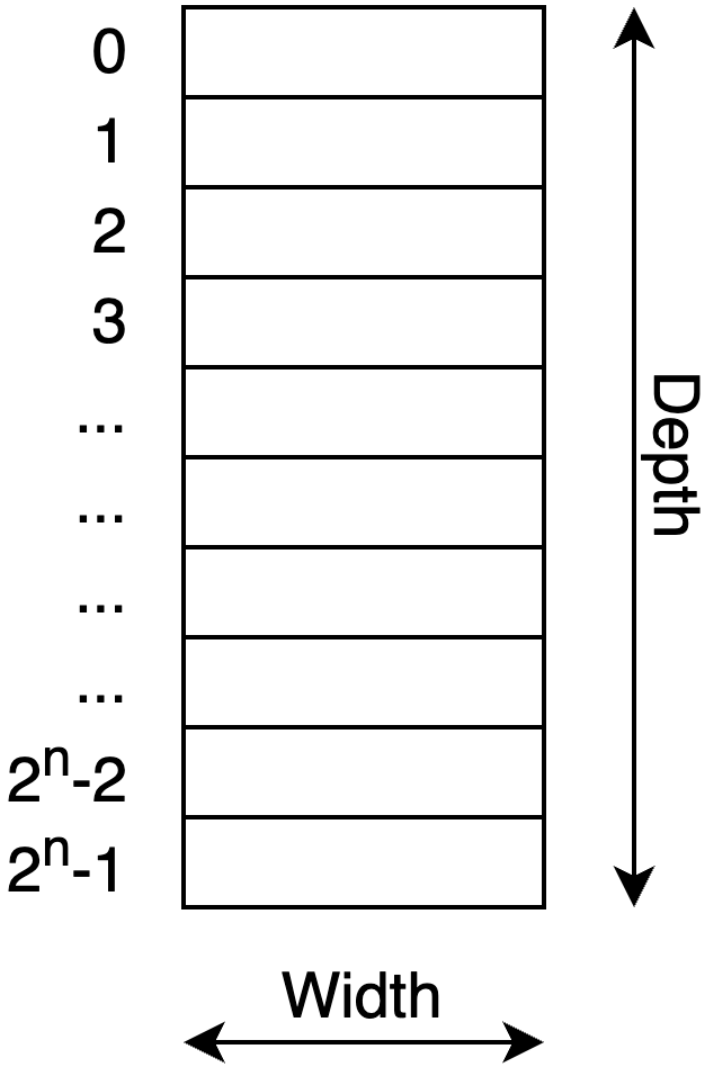
École affiliée IMT

Describing memory elements

- Applications needs to store large set of data
  - flip-flop based solution is not enough
  - Costly in ASIC technology
  - In modern FPGA ~200 000 bits
- Memory primitives
  - Small memory banks in ASIC
  - 18k/36k memory banks in FPGA
- FPGA logical synthesis tools need to understand memory description
- External memory (CTRL)

Explication au tableau

- A memory is like an array in C language
  - Memory word length (width)
  - Depth in #elements (depth)
- A memory is (normally) a synchronous element
  - One data is read per clock cycle
  - One data is written per clock cycle
- No reset signal!
  - We reset the elements one after the other
  - It can took time depending on the memory depth

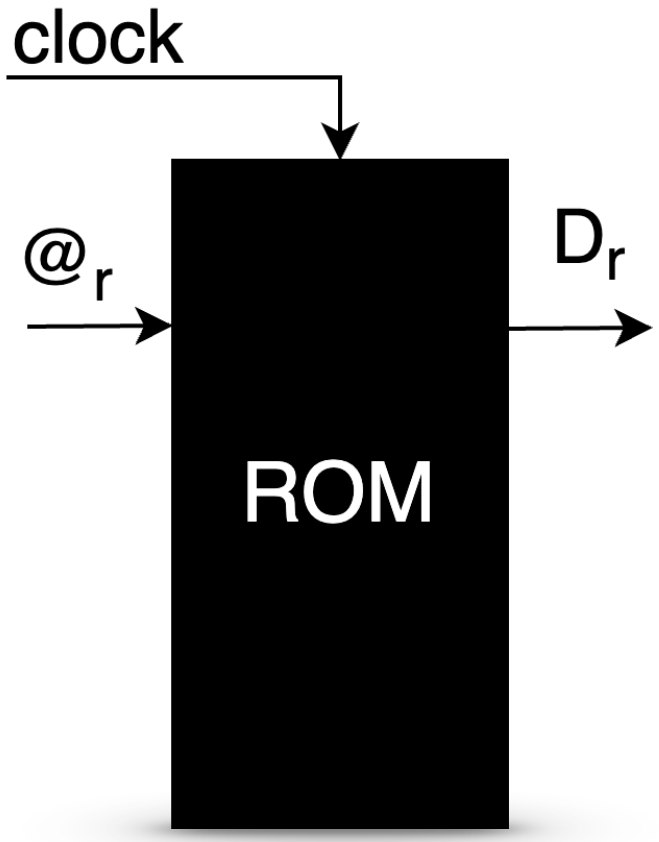


Declaring arrays in VHDL

- An array declaration is done in three times
  - Declaration of the element type (element)
  - Declaration of the size of the array (tab_type)
  - Declaration of the array (table)
- First two steps can be combined

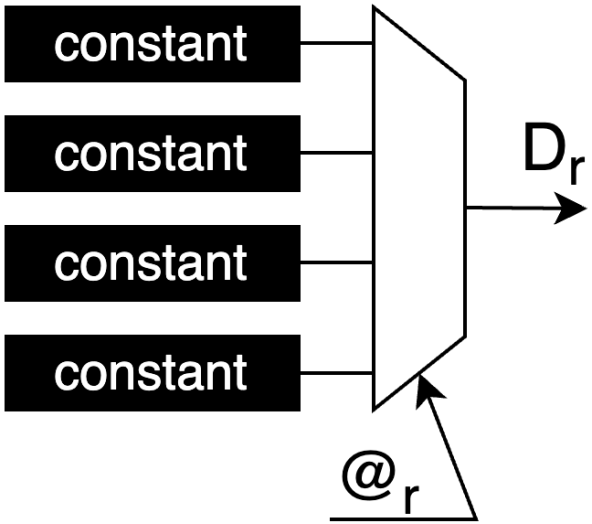
```
SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
TYPE tab_type    IS ARRAY (0 TO 7) OF element;

CONSTANT table : rom_type := (
  "1010", "1101", "1001", "0001",
  "1111", "0101", "1011", "1001"
 );

SUBTYPE element  IS UNSIGNED(3 DOWNTO 0);
TYPE tab_type IS ARRAY (0 TO 15) OF element;
SIGNAL table : tab_type;
```

Describing a Read-Only Memory




```
ENTITY ROM IS
PORT (
     CLOCK     : IN  STD_LOGIC;
     ADDR      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
     DATA_OUT  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
     );
END ROM;

ARCHITECTURE Behavioral OF ROM IS
 SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
 TYPE rom_type IS ARRAY (0 TO 7) OF element;

 CONSTANT memory : rom_type := (
   "1010", "1101", "1001", "0001",
   "1111", "0101", "1011", "1001"
  );

BEGIN

 PROCESS (CLOCK)
   VARIABLE V : INTEGER;
 BEGIN
   IF (CLOCK'event AND CLOCK = '1') THEN
     V := to_integer(UNSIGNED(ADDR));
     DATA_OUT <= memory(V);
   END IF;
 END PROCESS;

END Behavioral;
```

Describing a Read-Only Memory



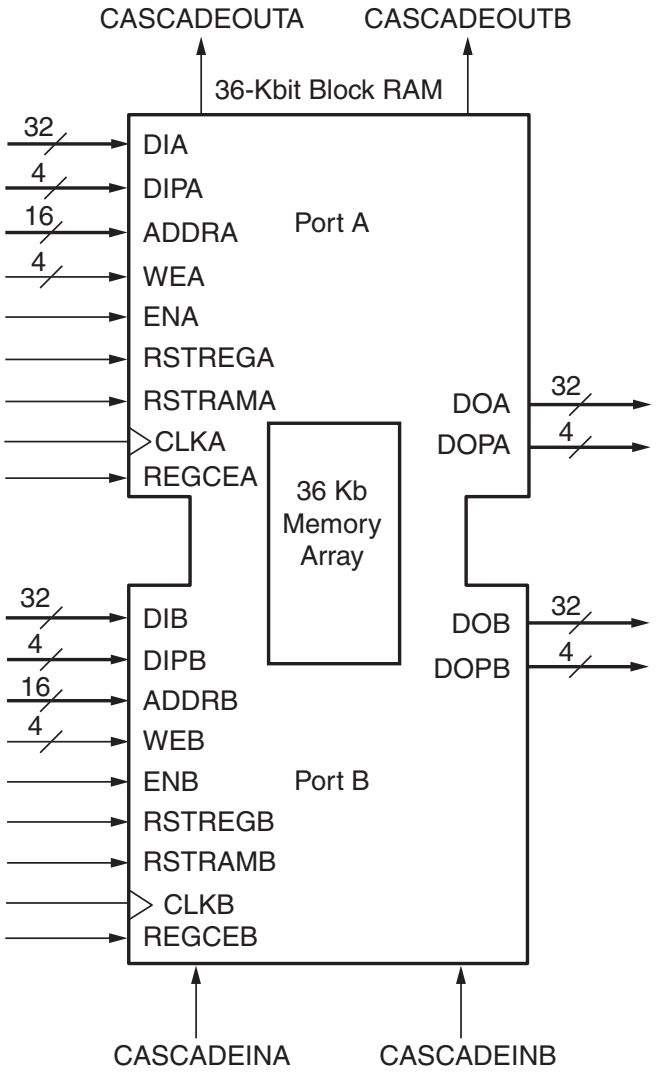

```
ENTITY ROM IS
PORT (
      CLOCK     : IN  STD_LOGIC;
      ADDR      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
      DATA_OUT  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
      );
END ROM;

ARCHITECTURE Behavioral OF ROM IS
  SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
  TYPE rom_type IS ARRAY (0 TO 7) OF element;

  CONSTANT memory : rom_type := (
    "1010", "1101", "1001", "0001",
    "1111", "0101", "1011", "1001"
   );

BEGIN

  PROCESS ( ADDR )
    VARIABLE V : INTEGER;
  BEGIN
      V := to_integer(UNSIGNED(ADDR));
      DATA_OUT <= memory(V);
  END PROCESS;

END Behavioral;
```

It's time for me to take a break!

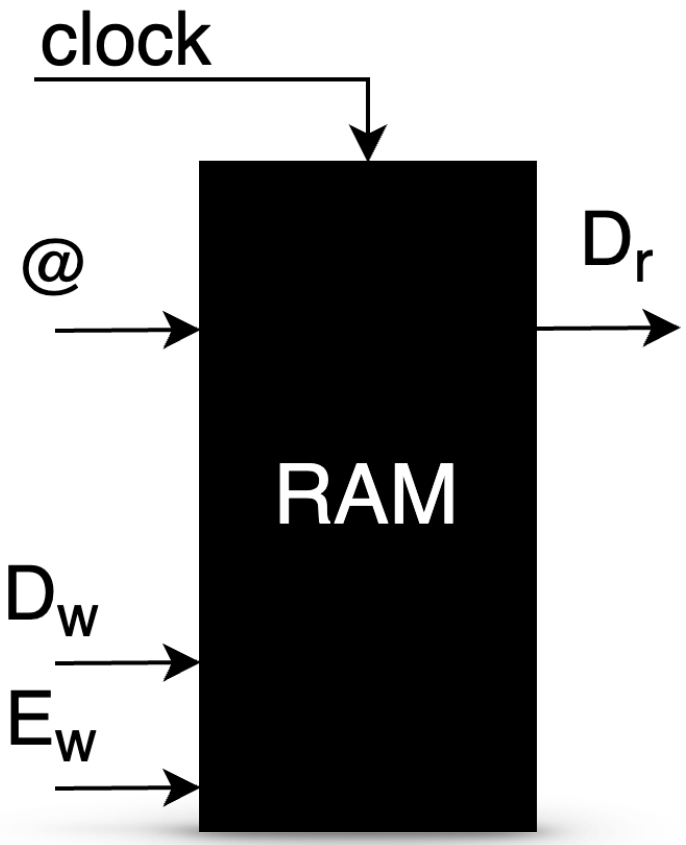
- Analyze and simulate this module description
  - Move to the example directory
    > cd 35-ROM-memories
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave

Random-Access Memories

- Different types of RAMs
  - Single-Port
  - Dual-Ports
- Reading & writing data
  - Writing data in memory need 1 cycle
  - Reading data in memory need 1 cycle
  - 2 cycles are required to see written value at output side
- VHDL description of RAM depends on FPGA vendor
  - All FPGA RAMs does not offers the same features



Describing a Single-Port RAM

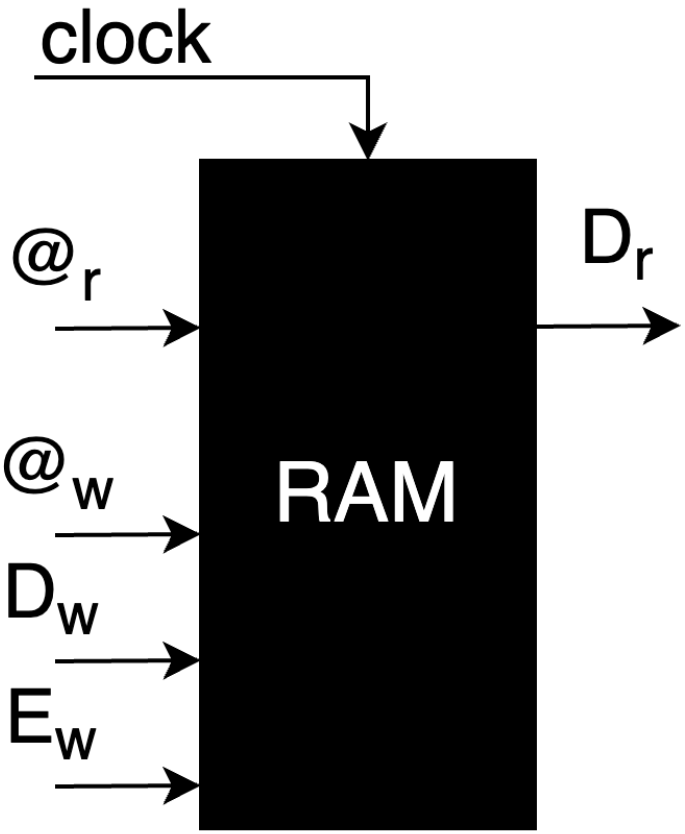



```
ARCHITECTURE Behavioral OF ROM IS
 SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
 TYPE ram_type IS ARRAY (0 TO 15) OF element;

 SIGNAL memory : ram_type;
BEGIN

 PROCESS (CLOCK)
   VARIABLE V : INTEGER;
 BEGIN
   IF (CLOCK'event AND CLOCK = '1') THEN
     V := to_integer(UNSIGNED(ADDR));

     IF DATA_We = '1' THEN
       memory(V) <= DATA_W;
     ELSE
       DATA_R <= memory(V);
     END IF;

   END IF;
 END PROCESS;

END Behavioral;
```

It's time for me to take a break!

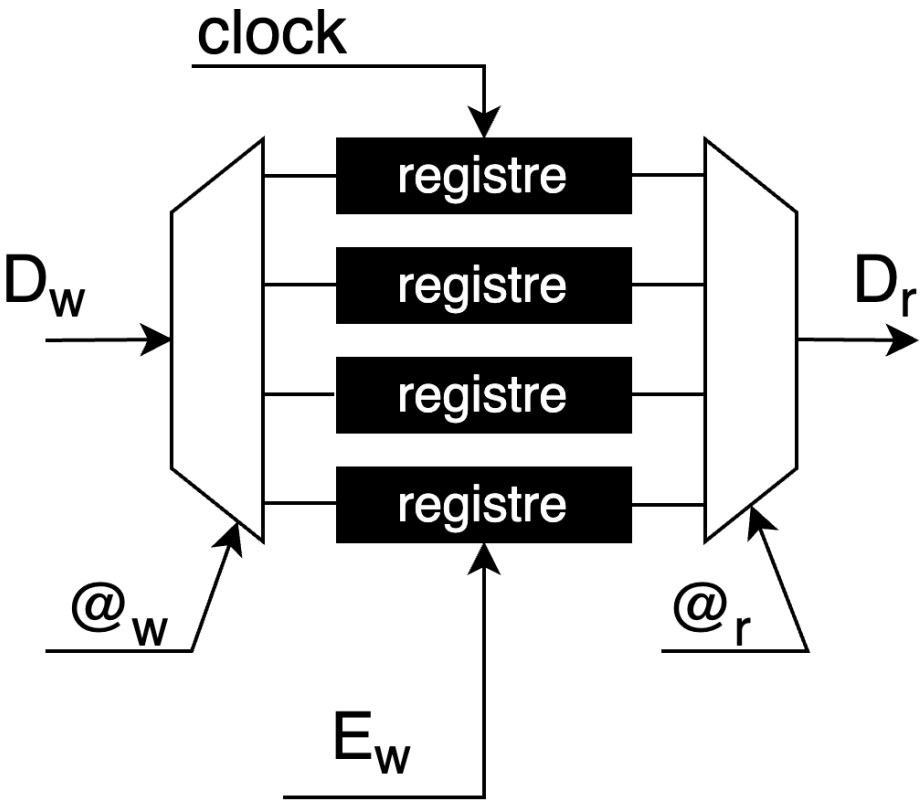
- Analyze and simulate this module description
  - Move to the example directory
    > cd 36-RAM-memory-sp
  - Analyse the description of the VHDL module
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave

Describing a Dual-Port RAM




```
ARCHITECTURE Behavioral OF ROM IS
 SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
 TYPE ram_type IS ARRAY (0 TO 15) OF element;

 SIGNAL memory : ram_type;
BEGIN

 PROCESS (CLOCK)
   VARIABLE V : INTEGER;
 BEGIN
   IF (CLOCK'event AND CLOCK = '1') THEN
     V := to_integer(UNSIGNED(ADDR));

     IF DATA_We = '1' THEN    -- if write is
       memory(V) <= DATA_W;   -- enable then
     END IF;                  -- do it

     DATA_R <= memory(V);
   END IF;
 END PROCESS;

END Behavioral;
```

Describing a LUT-based RAM




```
ARCHITECTURE Behavioral OF ROM IS
 SUBTYPE element  IS STD_LOGIC_VECTOR(3 DOWNTO 0);
 TYPE ram_type IS ARRAY (0 TO 15) OF element;

 SIGNAL memory : ram_type;
BEGIN

 PROCESS (CLOCK)
   VARIABLE V : INTEGER;
 BEGIN
   IF (CLOCK'event AND CLOCK = '1') THEN
     V := to_integer(UNSIGNED(ADDR));
     IF DATA_We = '1' THEN    -- if write is
       memory(V) <= DATA_W;   -- enable then
     END IF;                  -- do it
   END IF;
 END PROCESS;

 PROCESS (CLOCK)
   VARIABLE V : INTEGER;
 BEGIN
   V := to_integer(UNSIGNED(ADDR));
   DATA_R <= memory(V);
 END PROCESS;

END Behavioral;
```

It's time for me to take a break!

- Analyze and simulate this module description
  - Move to the example directory
    > cd 37-RAM-memories-dp
  - Analyse the descriptions of the VHDL modules
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[8 - VHDL for Flexibility]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

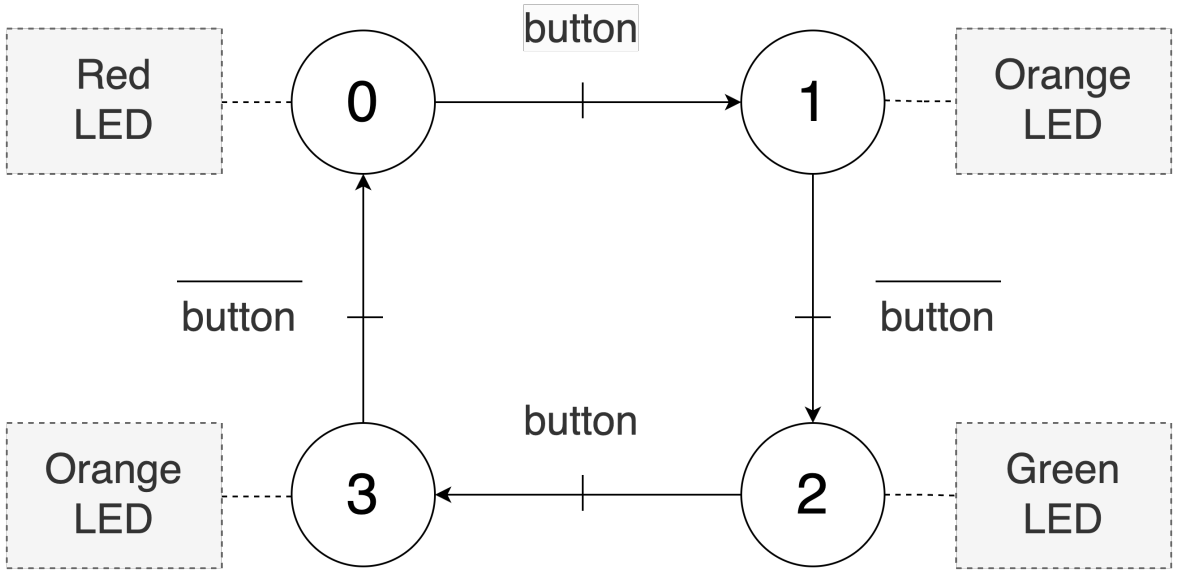
Université de Rennes

École affiliée IMT

Finite State Machines

Finite State Machine describe temporal behavior

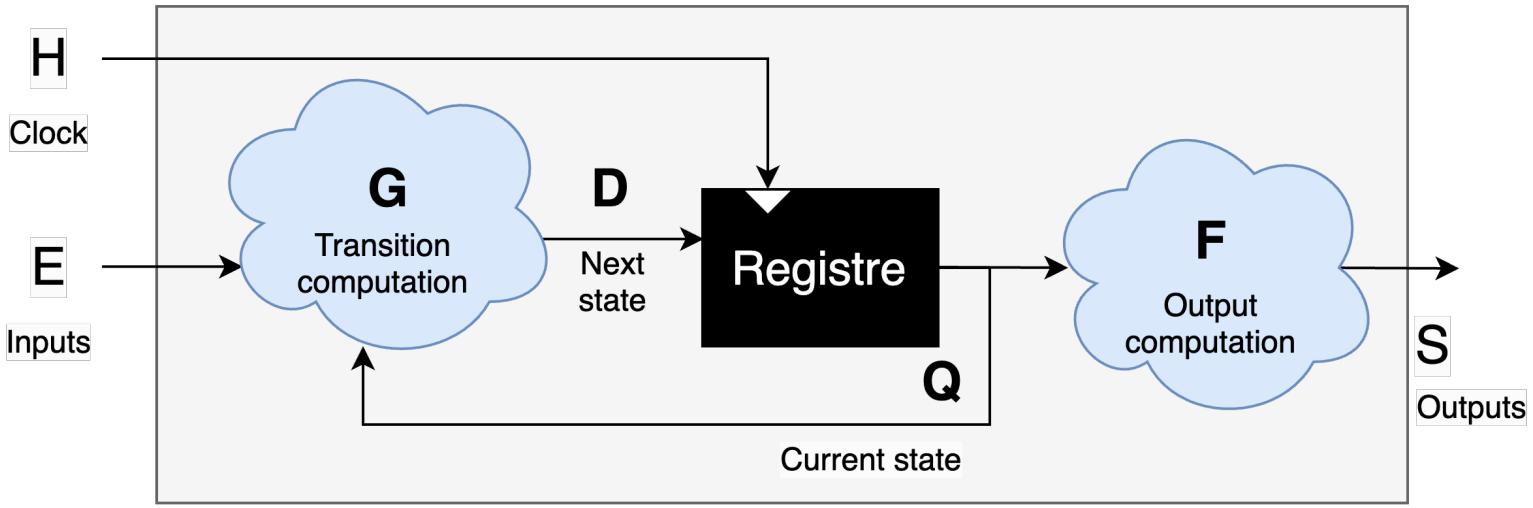
- Most systems need a controller (synchronization, computation scheduling, etc.)



Moore's FSM

Moore's FSM

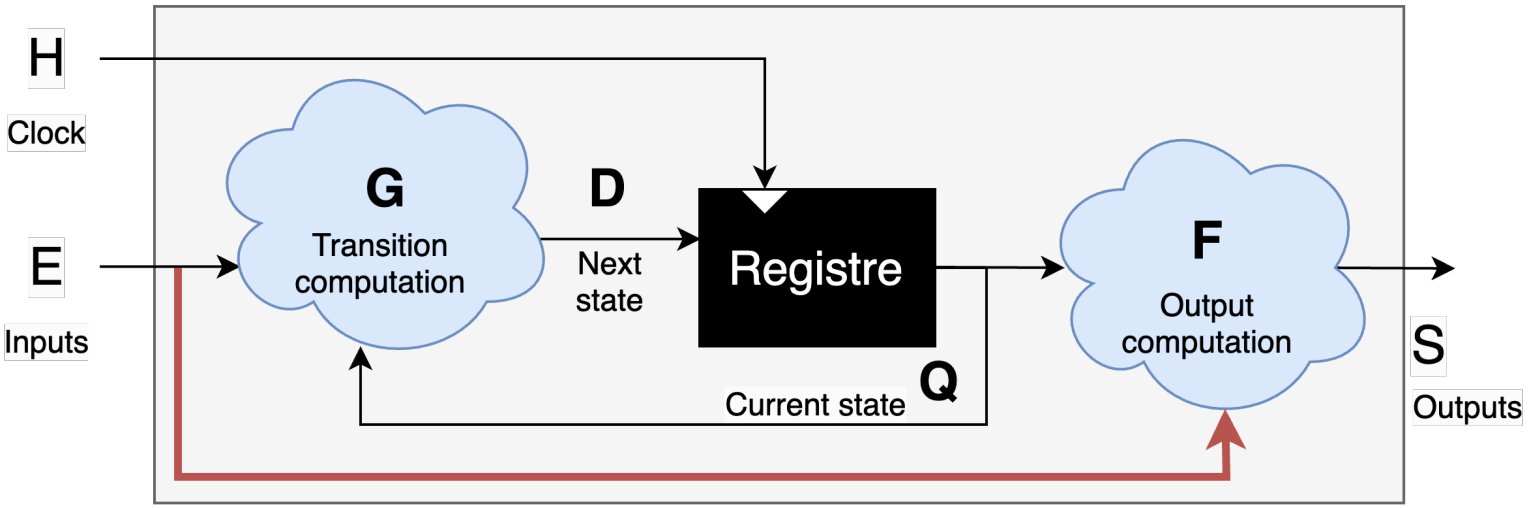
- A register of at least **log2(#states) = n bits** store the current state Q of the automata
- The F logic function computes the output values S depending on current state Q
- The G logic function computes the next FSM state D according to current state Q and the input values E



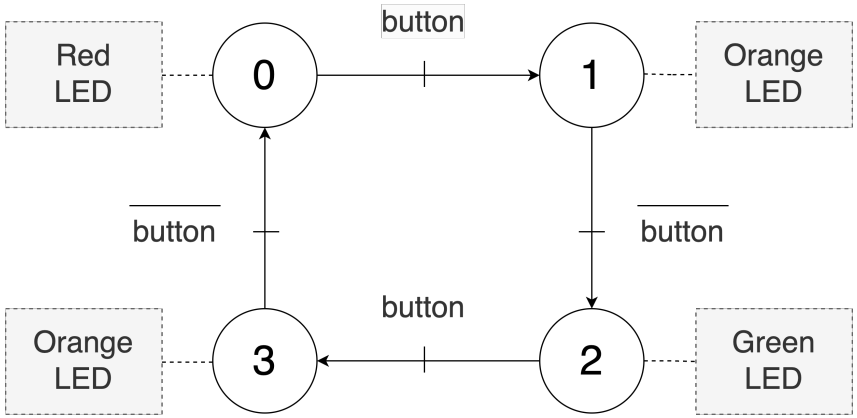
Mealy's FSM

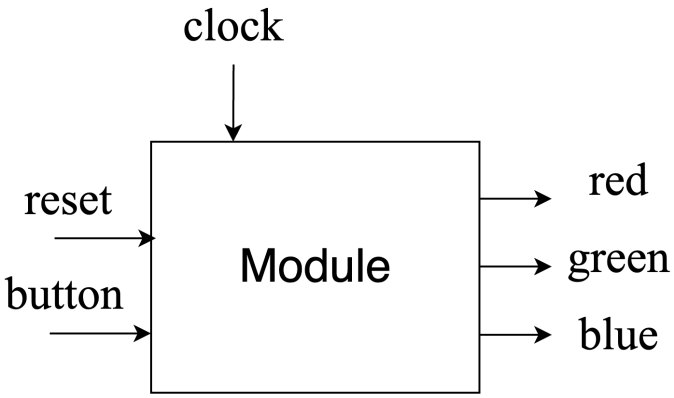
Mealy's FSM

- A register of at least **log2(#states) = n bits** store the current state Q of the automata
- The F logic function computes the output values S according to the current FSM state Q and the input values E
- The G logic function computes the next FSM state D according to current state Q and the input values E



VHDL description of a FSM (1)






```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY led_controler IS
    PORT (
        clock  : IN  STD_LOGIC;
        reset  : IN  STD_LOGIC;
        button : IN  STD_LOGIC;
        red    : OUT STD_LOGIC;
        orange : OUT STD_LOGIC;
        green  : OUT STD_LOGIC
    );
END led_controler;
```

A very classic entity ;-)

VHDL description of a FSM (2)

```
ARCHITECTURE arch OF led_controler IS
    type t_State is (S0, S1, S2, S3);

    signal c_state : t_State;
    signal n_state : t_State;
BEGIN

  --
  -- The processes
  --

END arch;
```

VHDL allows the description of enumerated types

We define a type that can have 4 distinct values

We define two signals of the type we have created

The logic synthesis tool performs binary mapping

VHDL description of a FSM (3)

```
ARCHITECTURE arch OF led_controler IS
    type t_State is (S0, S1, S2, S3);

    signal c_state : t_State;
    signal n_state : t_State;
BEGIN

    PROCESS( clock )
    BEGIN
        IF rising_edge( clock ) THEN
            IF reset = '1' THEN
                c_state <= S0;
            ELSE
                c_state <= n_state;
            END IF;
        END IF;
    END PROCESS;

    _
    -- The other processes
    --

END arch;
```

The first synchronous process takes care of the evolution of the current state

With each clock cycle, the system state evolves

The next state is calculated in another process

VHDL description of a FSM (4)

```
ARCHITECTURE arch OF led_controler IS
    -- ... ... ... …
BEGIN

    PROCESS( button, c_state )
    BEGIN
       CASE c_state IS
        WHEN S0 =>
                IF button = '1' THEN n_state <= S1;
                ELSE                 n_state <= S0;
                END IF;
        WHEN S1 =>
                IF button = '0' THEN n_state <= S2;
                ELSE                 n_state <= S1;
                END IF;
            WHEN S2 =>
                IF button = '1' THEN n_state <= S3;
                ELSE                 n_state <= S2;
                END IF;
            WHEN S3 =>
                IF button = '0' THEN n_state <= S0;
                ELSE                 n_state <= S3;
                END IF;
       END CASE;
    END PROCESS;

END arch;
```

Combinatorial process that describes the next state value as a function of the cur state & inputs

The switch-case structure is well suited to describing this type of process

Be careful to describe all signals, otherwise latches will appear.

VHDL description of a FSM (4)

```
ARCHITECTURE arch OF led_controler IS
    -- ... ... ... ...
BEGIN

    PROCESS( c_state )
    BEGIN
       CASE c_state IS
           WHEN S0 =>     red    <= '1';
                          orange <= '0';
                          green  <= '0';
           WHEN S1 =>     red    <= '0';
                          orange <= '1';
                          green  <= '0';
           WHEN S2 =>     red    <= '0';
                          orange <= '0';
                          green  <= '1';
           WHEN S3 =>     red    <= '0';
                          orange <= '1';
                          green  <= '0';
       END CASE;
    END PROCESS;

    -
    -- The other processes
    --

END arch;
```

Combinatorial process calculates output value as a function of current state only

The switch-case structure is well suited to describing this type of process

We could merge the 2 processes, but that wouldn't bring any advantages

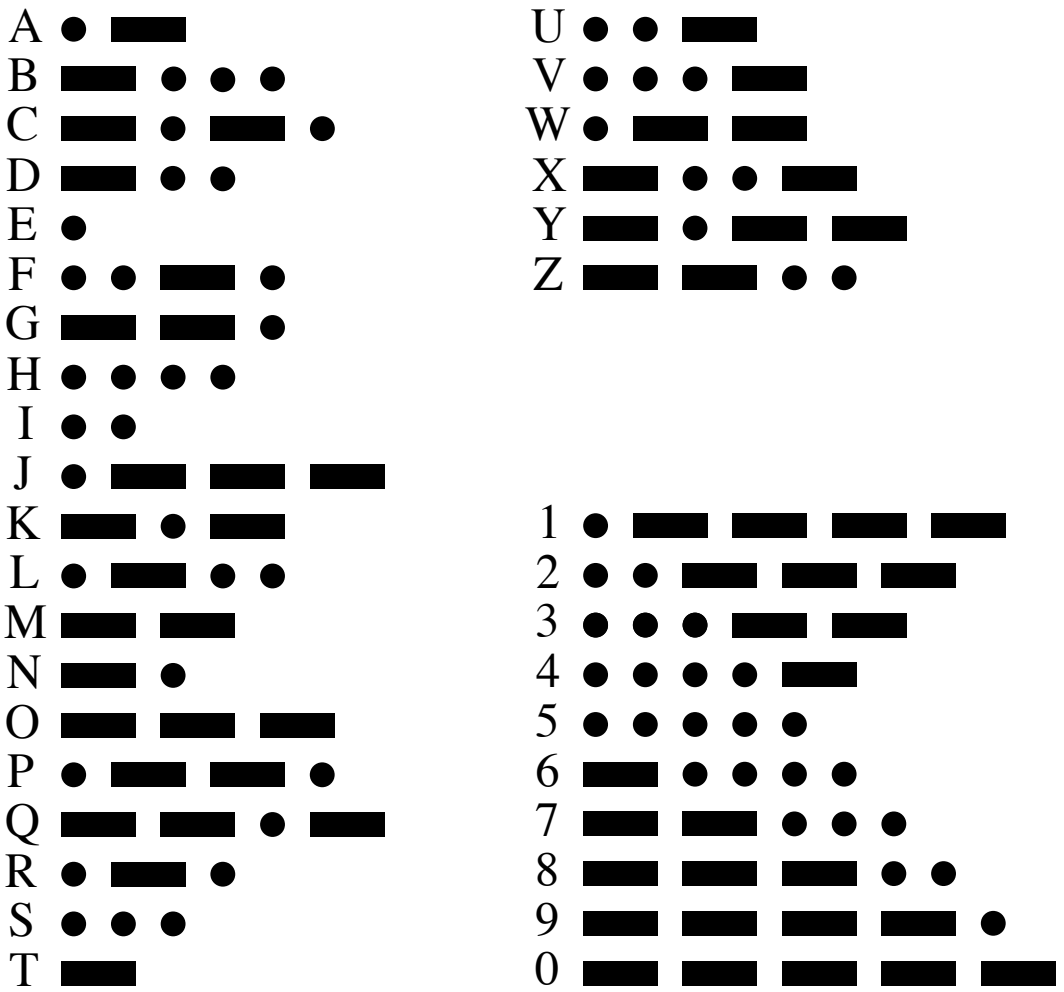
Be careful to describe all signals, otherwise latches will appear

It's time for me to take a break!

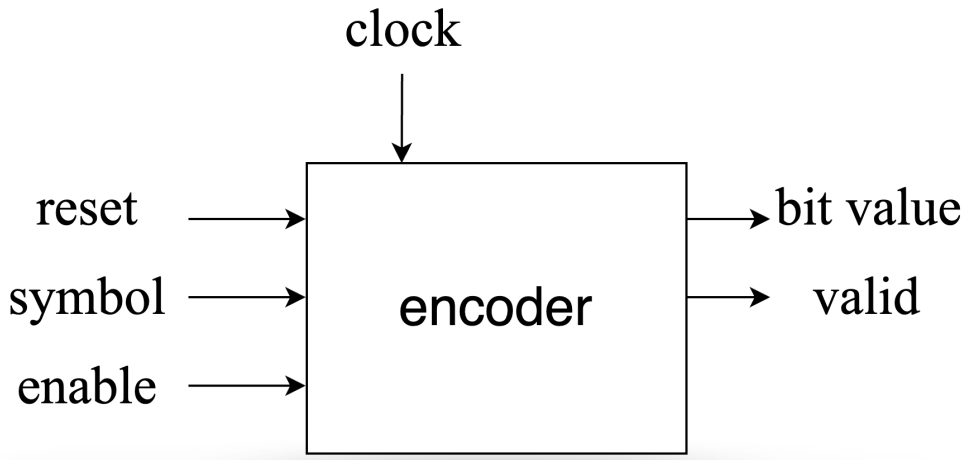
- Analyze and simulate this module description
  - Move to the example directory
    > cd 38-fsm-led-controler
  - Open the VHDL description of the FSM and complete it
    > make vscode
  - Simulate the module behaviors
    > make trace
  - Analyze the chronogram
    > make gtkwave
  - Observe the RTL design
    > make dc-rtl

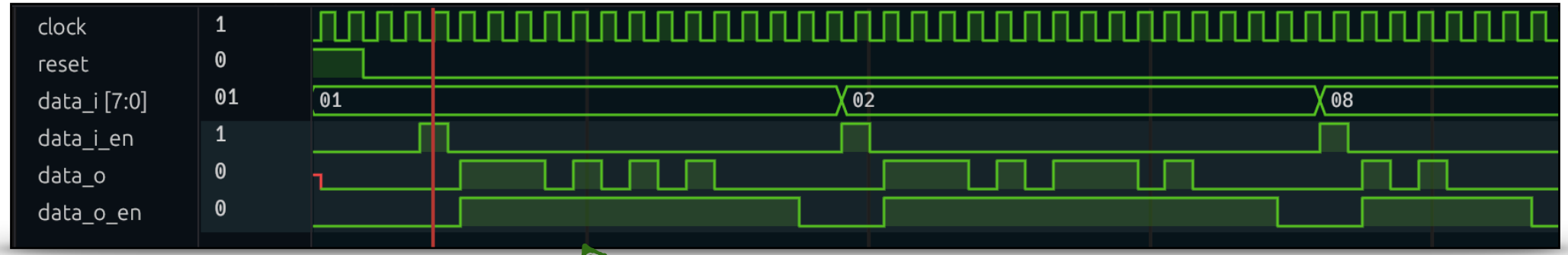
Transmitting information in Morse

- We're going to design a circuit for fun
  - FPGA implementation
- The system will consist of
  - An morse code emitter
  - A morse code receiver
- The main issue for you is not the VHDL ;-)



Morse encoding process





Generation of the « B » symbol

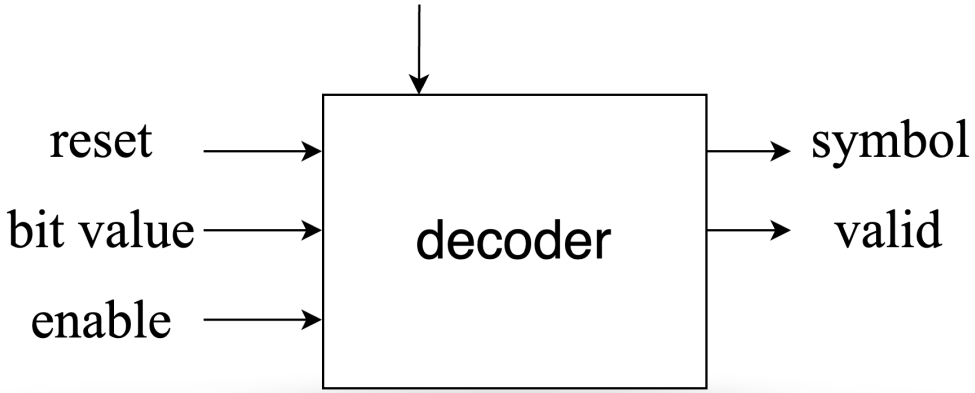
Morse encoder architecture

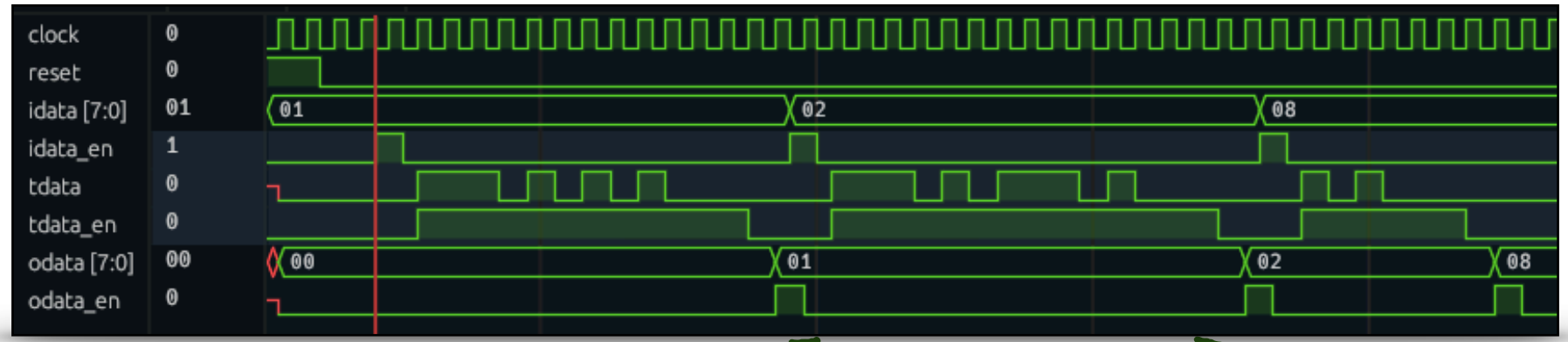
It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 39-code-morse-encoder
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

Morse decoding process





Regeneration of transmitted symbols

Morse decoder architecture

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 40-code-morse
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

Let's go on FPGA device

- This time, it is not so easy !
- Your computer (gtkterm)
  - Produces the 8b characters to transmit
  - Show the decoded 8b characters
  - Modules are required to communicate with your computer (UART)
- The morse system should be slow down
  - Clock frequency is 50~100 MHz
  - To see LED signals, the on/off frequency should be ~1 Hz
- Frequency domain converters are required

Top module architecture

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 41-code-morse-fpga
- Analyse and complete the module descriptions
  > make vscode
- Simulate the module behaviors without the UART interfaces
  > make trace
- Analyze the chronogram
  > make gtkwave
- Implement the design on FPGA
  > make fpga-build pga-load

HOMEWORK 41

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[8 - VHDL for Flexibility]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

The CONSTANT keyword

- In order to make VHDL code more generic and therefore more reusable, various solutions exist
- The constant data type can be used instead of the signal or variable types, to define data whose value cannot be modified
- This constant data type can be used to define the type of other data

```
ARCHITECTURE arch OF module IS
     CONSTANT N : INTEGER := 8;
     SIGNAL   B : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
BEGIN
```

VHDL for reusability

```
ENTITY counter_v1 IS
     PORT (
          CLK : in  STD_LOGIC;
          RST : in  STD_LOGIC;
          O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
     );
END counter_v1;

ARCHITECTURE Behavioral OF counter_v1 IS
     CONSTANT N      : INTEGER                  := 8;
     CONSTANT ZERO   : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
     CONSTANT ONE    : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
     SIGNAL COUNTER : UNSIGNED(N-1 DOWNTO 0);
BEGIN

     PROCESS(RST, CLK)
     BEGIN
          IF RST = '1' THEN
               COUNTER <= ZERO;
          ELSIF CLK = '1' AND CLK'EVENT THEN
               COUNTER <= COUNTER + ONE;
          END IF;
     END PROCESS;

     O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Description of a semi-generic counter

Modifying the value of N modifies the entire internal behavior of the counter

Quick and easy to adapt for new projects

All instances of this module will have the same characteristics

The module's generic concept

- It is possible to make a module totally generic by specifying its own parameters when instantiating it
- In addition to the I/O declaration in the entity, generic module parameters must be specified
- Be careful not to abuse this possibility, which can make things more complex:
  - VHDL writing in the module,
  - Debugging of written VHDL code
  - Its use in straightforward cases
- Default values can be set

Generic component description (1)

```
ENTITY counter_v3 IS
     GENERIC(
          N : INTEGER := 8
     );
     PORT (
          CLK : in  STD_LOGIC;
          RST : in  STD_LOGIC;
          O   : out STD_LOGIC_VECTOR(N-1 DOWNTO 0)
     );
END counter_v3;

ARCHITECTURE Behavioral OF counter_v1 IS
     CONSTANT ZERO  : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
     CONSTANT ONE   : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
     SIGNAL COUNTER : UNSIGNED(N-1 DOWNTO 0);
BEGIN

     PROCESS(RST, CLK)
     BEGIN
          IF RST = '1' THEN
               COUNTER <= ZERO;
          ELSIF CLK = '1' AND CLK'EVENT THEN
               COUNTER <= COUNTER + ONE;
          END IF;
     END PROCESS;

     O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Definition of a counter with a generic number of bits

Definition of the N value, which defaults to 8

The N parameter is re-used when defining signals

Generic component description (2)

```
ENTITY counter_v4 IS
     GENERIC(
          N : INTEGER := 8
     );
     PORT (
          CLK : in  STD_LOGIC;
          RST : in  STD_LOGIC;
          O   : out STD_LOGIC_VECTOR(N-1 DOWNTO 0)
     );
END counter_v4;

ARCHITECTURE Behavioral OF counter_v4 IS
     SIGNAL COUNTER : INTEGER RANGE 0 TO (2**N)-1;
BEGIN

     PROCESS(RST, CLK)
     BEGIN
          IF RST = '1' THEN
               COUNTER <= 0;
          ELSIF CLK = '1' AND CLK'EVENT THEN
               IF COUNTER = (2**N)-1 THEN
                    COUNTER <= 0;
               ELSE
                    COUNTER <= COUNTER + 1;
               END IF;
          END IF;
     END PROCESS;

     O <= STD_LOGIC_VECTOR( TO_UNISGNED(COUNTER, N) );

END Behavioral;
```

Definition of a counter with a generic number of bits

Definition of the N value, which defaults to 8

The N parameter is re-used when defining signals

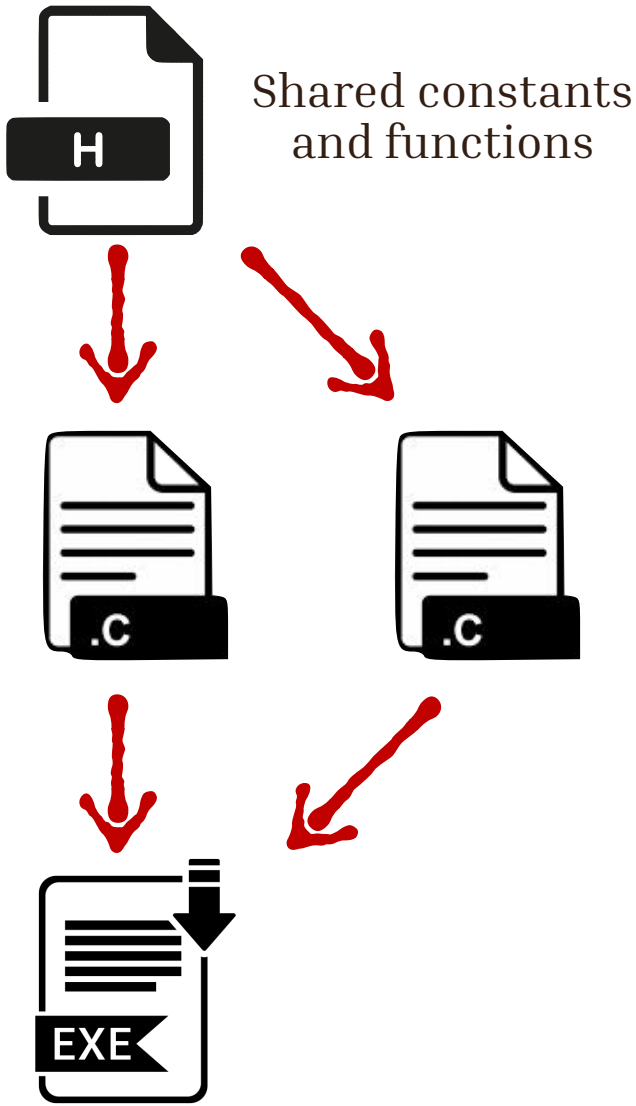
Generic component instantiation

- Modules are configured using instantiation parameters
- Two possibilities available
  - Use default settings, i.e. no GENERIC MAP declaration
  - Specify your own parameter values i.e. GENERIC MAP declaration
- Using generics makes module development more complex, but increases reusability

```
-- COMPTEUR DE 0...255
c1 : ENTITY work.counter_v4
    PORT MAP(
          CLK => CLK,
          RST => RST,
          O   => VALUE
    );

-- COMPTEUR DE 0...1023
c2 : ENTITY work.counter_v4
    GENERIC MAP(
          N => 10
    )
    PORT MAP(
          CLK => CLK,
          RST => RST,
          O   => VALUE
    );
```

Using configuration packages

- The module's genericity is interesting
  - Can be difficult to manage in a complex project
- An another approach consist in using packages
  - They are equivalent to #define and ".h" files in C language
- A package is defined in a separate VHDL file
  - In VHDL module header we specified in included packages



Package usage example

Here we specify in our module that we wish to use the package

We create a VHDL file (package) named “conf_counter.vhd” that contains the shared information

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package conf_counter is
     CONSTANT N    : INTEGER :=   8;
     CONSTANT MINV : INTEGER :=  13;
     CONSTANT MAXV : INTEGER := 222;
end conf_counter;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.conf_counter.all;

ENTITY counter IS
  PORT (
    CLK : in  STD_LOGIC;
    RST : in  STD_LOGIC;
    O   : out STD_LOGIC_VECTOR(N-1 DOWNTO 0)
  );
END counter

ARCHITECTURE Behavioral OF counter IS
  SIGNAL COUNTER : UNSIGNED(N-1 DOWNTO 0);
BEGIN
  PROCESS(RST, CLK)
  BEGIN
    IF RST = '1' THEN
      COUNTER <= TO_UNSIGNED( MINV, N );
    ELSIF CLK = '1' AND CLK'EVENT THEN
      IF COUNTER = TO_UNSIGNED( MAXV, N ) THEN
        COUNTER <= TO_UNSIGNED( MINV, N );
      ELSE
        COUNTER <= COUNTER + TO_UNSIGNED( 1, N );
      END IF;
    END IF;
  END PROCESS;

  O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 42-counter-package
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 42

The generate statements

- Generate statements are resolved at synthesis time
- Two main goals:
  - Turning on/off blocks of logic
  - Replicating logic or modules
- label : for ... generate
  - Copy. paste N times a VHDL code
- label : if ... generate
  - Include or remove a VHDL code from the module behavior

Generate statements can be seen as « preprocessor commands » in C, but they does not offer same features

Pre-Processor	Description
#include	It is used to include a file in the C program
#define	It is used to define a macro or constant
#ifdef	It checks if a macro is defined or not
#ifndef	It checks if a macro is not defined
#if	It checks for a condition at compile time
#else	It specifies the alternative statement if the condition is false
#elif	It specifies the next condition if the previous #if statement is false
#endif	It denotes the end of the conditional statements
#pragma	It is used to provide additional information to the compiler

For-generate loops in VHDL (1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_gate_generate_for IS
PORT (
        A : IN  std_logic_vector(3 downto 0);
        B : IN  std_logic_vector(3 downto 0);
        C : OUT std_logic_vector(3 downto 0)
    );
END xor_gate_generate_for;

ARCHITECTURE arch OF xor_gate_generate_for IS
BEGIN

        LOOP_ADD : for I in 0 to 3 generate
                C(i) <= A(i) XOR B(i);
        end generate;

END arch;
```

The for loop will be unrolled 4 times at synthesis time

The loop counter could come from VHDL package or could be a generic parameter

For-generate loops in VHDL (2)

```
ENTITY adder_8b IS
    GENERIC( N : INTEGER := 8 );
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        S : OUT std_logic_vector(8 downto 0)
    );
END adder_8b;

ARCHITECTURE arch OF adder_8b IS
    SIGNAL C : std_logic_vector(0 to N+1);
BEGIN

    C(0) <= '0';

    LOOP_ADD : for I in 0 to N generate
        S(I)     <=  A(I) xor B(I) xor C(I);
        C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;

    S(8) <= C(8);

END arch;
```

The loop content can be more complex and if needed it can include processes

For-generate loops in VHDL (2)

```
ARCHITECTURE arch OF adder_8b IS

    COMPONENT full_adder IS
    PORT ( ..... );
    END COMPONENT;

    COMPONENT half_adder IS
    PORT ( ..... );
    END COMPONENT;

    SIGNAL C : STD_LOGIC_VECTOR(7 DOWNTO 1);

BEGIN

    i1 : half_adder port map(A(0), B(0), S(0), C(1) );

    LOOP_ADD : for I in 1 to 6 generate
        inst : full_adder port map(A(i), B(i), C(i),   S(i), C(i+1));
    end generate;

    inst : full_adder port map(A(7), B(7), C(7), S(7), S(8));

END arch;
```

For-generate loops are also used to instantiate modules

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 43-adder-*
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 43 44 45

If-generate structure in VHDL (1)

```
ENTITY adder_8b IS
    GENERIC( NATIVE : INTEGER := 1 );
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        S : OUT std_logic_vector(8 downto 0)
    );
END adder_8b;

ARCHITECTURE arch OF adder_8b IS
    SIGNAL C : std_logic_vector(0 to 8);
BEGIN
    cnd : if NATIVE = 0 generate

        C(0) <= '0';
        LOOP_ADD : for I in 0 to 7 generate
            S(I)     <=  A(I) xor B(I) xor C(I);
            C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
        end generate;
        S(8) <= C(8);

    else generate

        S <= STD_LOGIC_VECTOR( RESIZE(UNSIGNED(A), 9) +  RESIZE(UNSIGNED(B), 9));

    end generate;

END arch;
```

Depending on GENERIC parameter, the internal behavior specification is not the same

If-generate structure in VHDL (2)

```
ARCHITECTURE arch OF adder_8b IS

    COMPONENT adder_8b_comb IS
    PORT (
        -- ... I/O list
    );
    END COMPONENT;

    COMPONENT adder_8b_native IS
    PORT (
        -- ... I/O list
    );
    END COMPONENT;

BEGIN

    cnd : if NATIVE = 0 generate

        add : adder_8b_comb. ( A => A, B => B, C => C);

    else generate

        add : adder_8b_native( A => A, B => B, C => C);

    end generate;

END arch;
```

The same structure can be applied to instantiate specific modules depending on user choice

It's time for me to take a break!

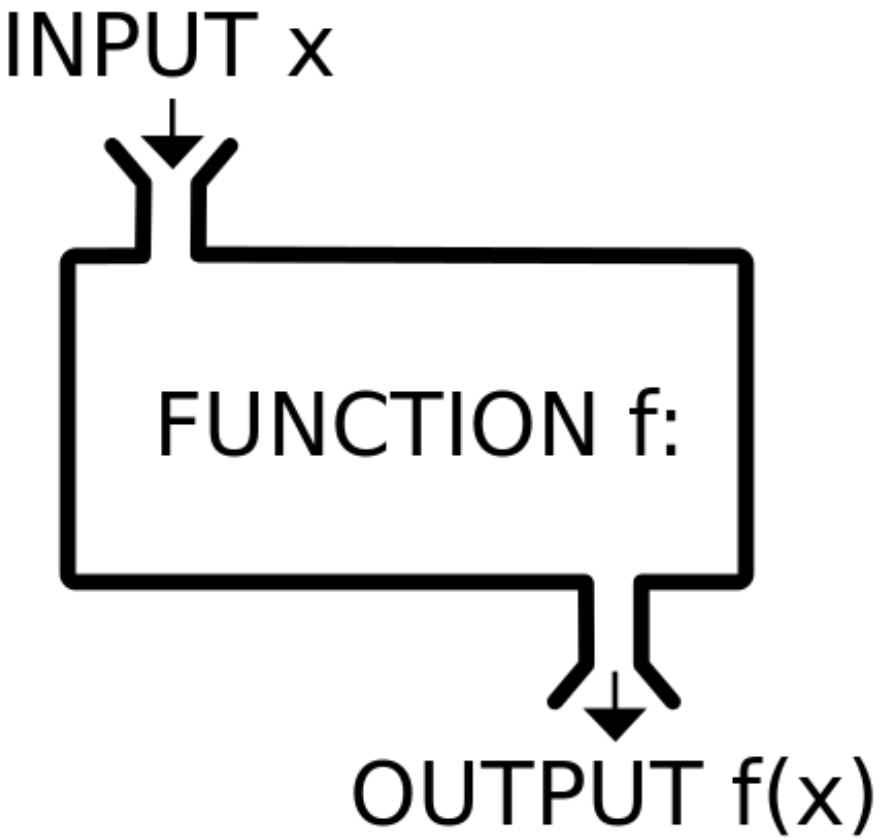
Analyze and simulate this module description

- Move to the example directory
  > cd 46-adder-generate-if
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

HOMEWORK 46 47

Using functions in VHDL models

- You can define functions in VHDL
  - This improves the readability of the source code
  - This enhances code reusability
  - This simplifies debugging
- The functions are rolled out during logical synthesis
  - A function called N times implies N distinct module instantiations in the post-synthesis circuit
- Different to C functions !



VHDL is a awful language

```
ENTITY min_v1 IS
     PORT (
          A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          C : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          D : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          S : out STD_LOGIC_VECTOR(7 DOWNTO 0)
     );
END min_v1;
```

The **min_v1** VHDL module must compute the minimum value between 4 unsigned inputs

```
ARCHITECTURE Behavioral OF min_v1 IS
BEGIN

     PROCESS(A, B, C, D)
          VARIABLE T1, T2, T3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
     BEGIN
          IF UNSIGNED(A) < UNSIGNED(B) THEN
               T1 := A;
          ELSE
               T1 := B;
          END IF;
          IF UNSIGNED(C) < UNSIGNED(D) THEN
               T2 := C;
          ELSE
               T2 := D;
          END IF;
          IF UNSIGNED(T1) < UNSIGNED(T2) THEN
               S <= T1;
          ELSE
               S <= T2;
          END IF;
     END PROCESS;

END Behavioral;
```

The description of the module's internal behavior is repetitive...

Functions help factoring the code

```
ENTITY min_v1 IS
     PORT (
          A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          C : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          D : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
          S : out STD_LOGIC_VECTOR(7 DOWNTO 0)
     );
END min_v1;
```

The **min_v1** VHDL module must compute the minimum value between 4 unsigned inputs

Declaration of a generic function that computes the mini value of two STD_LOGIC_VECTOR data

The module's source code is clearer and more concise

```
ARCHITECTURE Behavioral OF min_v2 IS

     FUNCTION mini(A, B: STD_LOGIC_VECTOR )
               return STD_LOGIC_VECTOR IS
     BEGIN
          IF UNSIGNED(A) < UNSIGNED(B) THEN
               RETURN A;
          ELSE
               RETURN B;
          END IF;
     END mini;

BEGIN

     PROCESS(A, B, C, D)
          VARIABLE T1, T2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
     BEGIN
          T1 := mini( A,  B);
          T2 := mini( C,  D);
          S  <= mini(T1, T2);
     END PROCESS;

END Behavioral;
```

It's time for me to take a break!

Analyze and simulate this module description

- Move to the example directory
  > cd 47-min4-function
- Analyse the descriptions of the VHDL modules
  > make vscode
- Simulate the module behaviors
  > make trace
- Analyze the chronogram
  > make gtkwave
- Observe the RTL design
  > make dc-rtl

VHDL Signal Properties

- Signals have properties that can simplify code writing
- You've already used these properties
  - IF CLOCK'EVENT AND CLOCK = '1' THEN
- Other properties include
  - Size of data vectors
  - MSB and LSB positions, etc.
- All types do not offer the same properties

A small list of signal methods

The most valuable signal's properties

S'EVENT	is true if signal S has had an event this simulation cycle.
T'LEFT	is the leftmost value of type T. (Largest if downto)
T'RIGHT	is the rightmost value of type T. (Smallest if downto)
T'IMAGE(X)	is a string representation of X that is of type T.
T'VALUE(X)	is a value of type T converted from the string X.
A'LEFT	is the leftmost subscript of array A or constrained array type.
A'LEFT(N)	is the leftmost subscript of dimension N of array A.
A'RIGHT	is the rightmost subscript of array A.
A'RIGHT(N)	is the rightmost subscript of dimension N of array A.
A'RANGE	is the range A'LEFT to A'RIGHT or A'LEFT downto A'RIGHT .
A'REVERSE_RANGE	is the range of A with to and downto reversed.
A'REVERSE_RANGE(N)	is the REVERSE_RANGE of dimension N of array
A.A'LENGTH	is the integer value of the number of elements in array A.

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[10 - Let's Design a Complete System]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

Other HDL languages

- During last decades, a lot of HDL languages were proposed
  - Academics or industrial
  - Offers different properties
  - Concepts are the SAMES
- Currently
  - VHDL (European)
  - Verilog / SystemVerilog (USA)
  - SystemC (Hard/Soft)
  - Python-based frameworks

https://en.wikipedia.org/wiki/Hardware_description_language

Verilog language

```
Entity and3 is
 Port(
  e1,e2,e3: in bit;
  s: out bit);
End and3;
Architecture RTL of and3 is
Begin
   s <= e1 and e2 and e3;
End RTL;
```

```
Module and3(e1,e2,e3);
  Input e1,e2,e3;
  Output s;

  Assign s= e1 & e2 & e3;
Endmodule
```

SystemC language

```
Entity and3 is
 Port(
  e1,e2,e3: in bit;
  s: out bit);
End and3;
Architecture RTL of and3 is
Begin
   s <= e1 and e2 and e3;
End RTL;
```

```
#include "systemc.h"
SC_MODULE(and3) {
        sc_in<bool> e1;
        sc_in<bool> e2;
        sc_in<bool> e3;
        sc_out<bool> s;
        void compute_and() {
                s = e1 & e2 & e3;
         };
        SC_CTOR(and3) {
                SC_METHOD(compute_and);
                sensitive << e1 << e2 << e3;
        }
 };
```

ENSSAT

LANNION

INFORMATIQUE
PHOTONIQUE
SYSTÈMES NUMÉRIQUES

VHDL language for Synthesis

[9 - Other HDL Languages]

Bertrand LE GAL

Laboratoire IRISA, UMR CNRS 5218
Département « D3 - Architecture »
Equipe projet INRIA « TARAN »

bertrand.le-gal@irisa.fr
https://blegal.github.io

Université de Rennes

École affiliée IMT

Let's design a real system

- Register-Transfer Level (RTL) Specifications
  - Specification of the logic/arithmetic behavior between clocked registers