Programming FPGA with VHDL
An introduction – Part 1

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

2024 - Sciences du Numérique
Parcours Architecture, Systèmes et Réseaux

Cours à l'ENSEEIHT

1. 5 Cours (2 cours VHDL + 3 cours Architecture Avancée)
2. 5 TD (VHDL)
3. 5 TP d'initiation (VHDL)
4. 5 TP encadrés mini-projet (VHDL)
5. 9 TP projet avec encadrement léger (VHDL)
6. évaluation : 1 Exam 1h30 + 1 mini projet (individuel) + 1 projet (binôme)

Outline

1. FPGA
2. VHDL language

Sources and Language References

- Wikipedia
  - FPGA (http://en.wikipedia.org/wiki/FPGA)
  - VHDL (http://en.wikipedia.org/wiki/VHDL)
- Xilinx /AMD (https://www.xilinx.com/products/silicon-devices/fpga.html)
- VHDL (http://www.csee.umbc.edu/portal/help/VHDL/summary.html)
- Free Range VHDL (https://www.isy.liu.se/edu/kurs/TSEA83/kursmaterial/vhdl/free_range_vhdl_2019.pdf)

Context

- design of an electronic device
- usually the device (circuit) is dedicated to perform a specific task (data manipulation, image treatments, signal processing, ...)
- once it is manufactured, you cannot modify it
  - improve/update it
  - change its functionalities
  - ...

What's a FPGA

A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured **after** manufacturing to the desired application or functionality requirements.

FPGAs:

- are based around a matrix of **Configurable Logic Blocks (CLBs)** connected through programmable interconnects,
- are opposed to **Application Specific Integrated Circuits (ASICs)**, where the device is custom built for a particular design,
- have large resources of **logic gates** and **RAM blocks** to implement complex digital computations,
- can be programmed using a **Hardware Description Language (HDL)** or ***schematic design***.

What's a FPGA

Figure: Altera StratixIVGX FPGA

Figure: Xilinx

FPGA Advantages

The main advantages of FPGAs

- ability to re-program in the field,
- ability to update the functionality after shipping,
- partial re-configuration of a portion of the design,

- shorter time to market,
- lower non-recurring engineering costs,
- cheaper for low-volume applications.

Now, FPGAs rival corresponding ASIC solutions (reduced power, increased speed, lower materials cost, minimal implementation real-estate, ...)

FPGA Advantages

Middle road:

- develop hardware on ordinary FPGAs,
- then manufacture final version as an ASIC

$=>$ The hardware can no longer be modified after the design has been committed

FPGA Applications

An FPGA can be used to solve any problem which is computable.

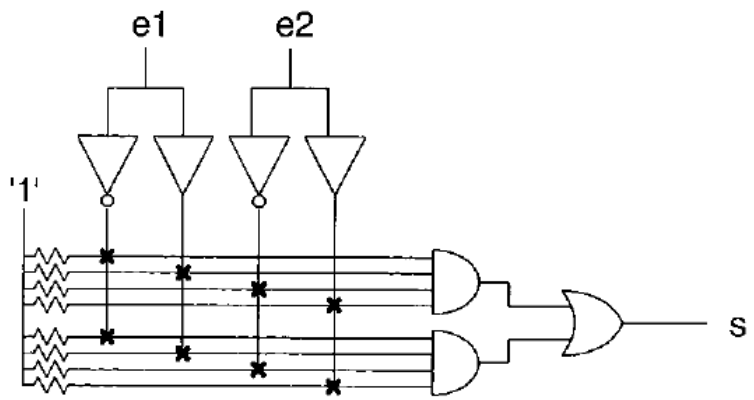
Specific applications of FPGAs:

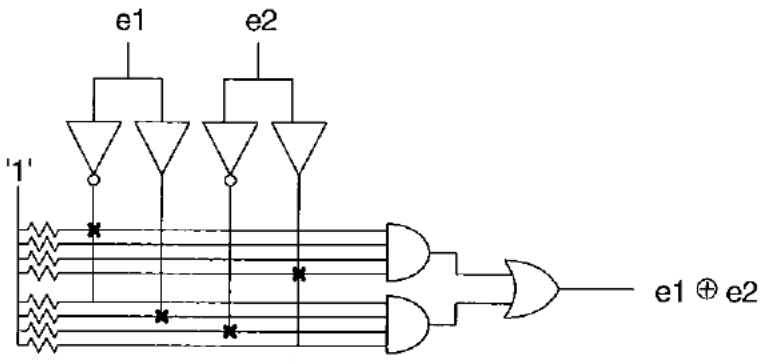
- Aerospace and Defense,
- Emulation & Prototyping,
- Automotive,
- Broadcast,
- Consumer Electronics,
- Data Center,
- High Performance Computing,
- Industrial,
- Medical,
- Test & Measurement,
- Wired & Wireless Communication

see more `https://www.amd.com/en/resources/case-studies.html`

FPGA Architecture

Example of PAL (Programmable Array Logic)

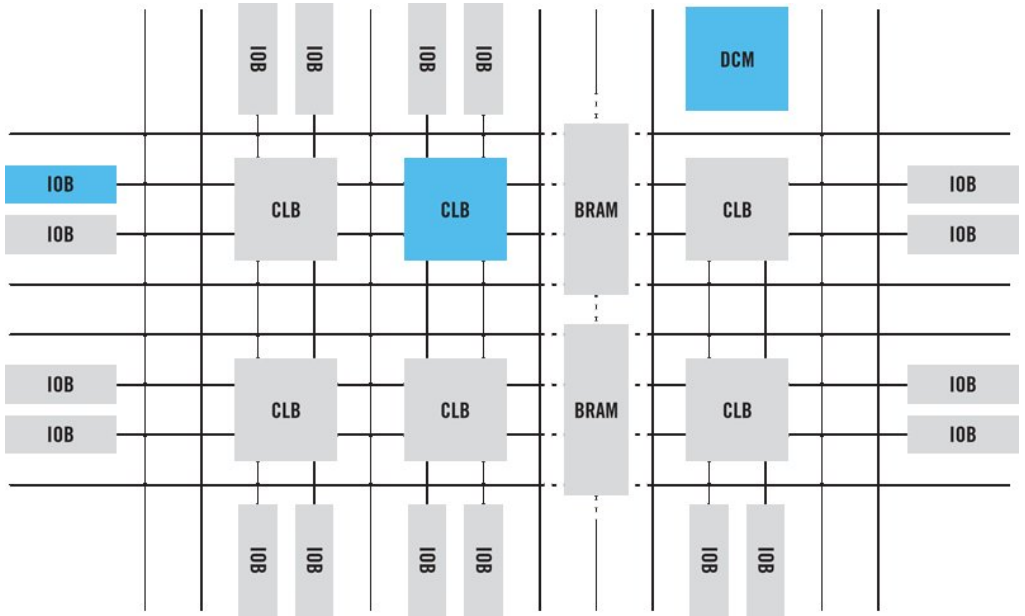




Source `Le Langage VHDL`, *Jacques Weber, Maurice Meaudre*

FPGA Architecture

The most common FPGA architecture consists of an array of logic blocks (Configurable Logic Block (CLB) or Logic Array Block (LAB)), I/O pads, and routing channels.



Source `Xilinx`

FPGA Architecture

- Each CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops.
- The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM.

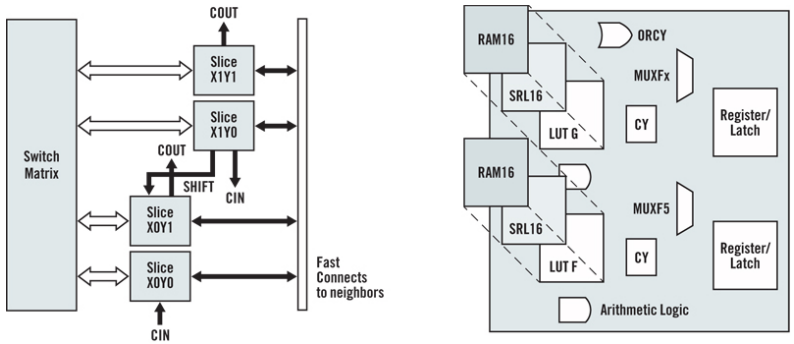


Figure: Basic Configurable Logic Block Structure (source `Xilinx`)

- Flexible interconnect routing routes the signals between CLBs and to and from I/Os.

FPGA Architecture

Modern FPGAs include higher level functionalities

- multipliers,
- generic DSP blocks,
- embedded processors,
- high speed I/O logic,
- embedded memories.

Major manufacturers

- The FPGA market is projected to grow from USD 12.1 billion in 2024 and is projected to reach USD 25.8 billion by 2029 (`source: marketsandmarkets`)
- **AMD (formerly Xilinx)** and **Intel (formerly Altera)** are the current FPGA market leaders
- FPGA board manufacturers: Digilent, Bittware, Trenz Electronic, Inrevium, ...
- Open-source

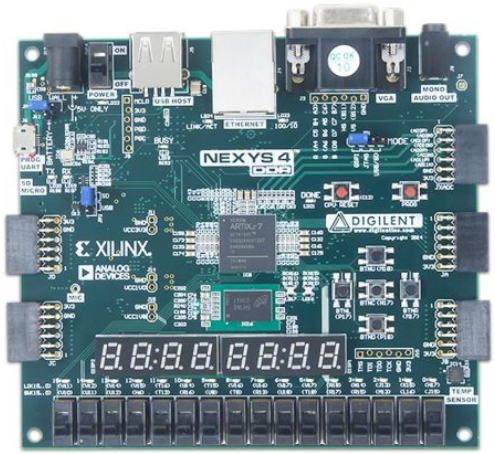


Figure: Digilent Nexys 4

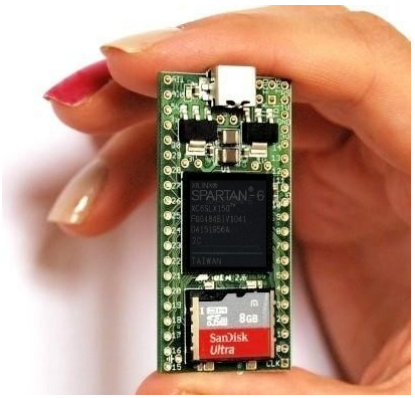
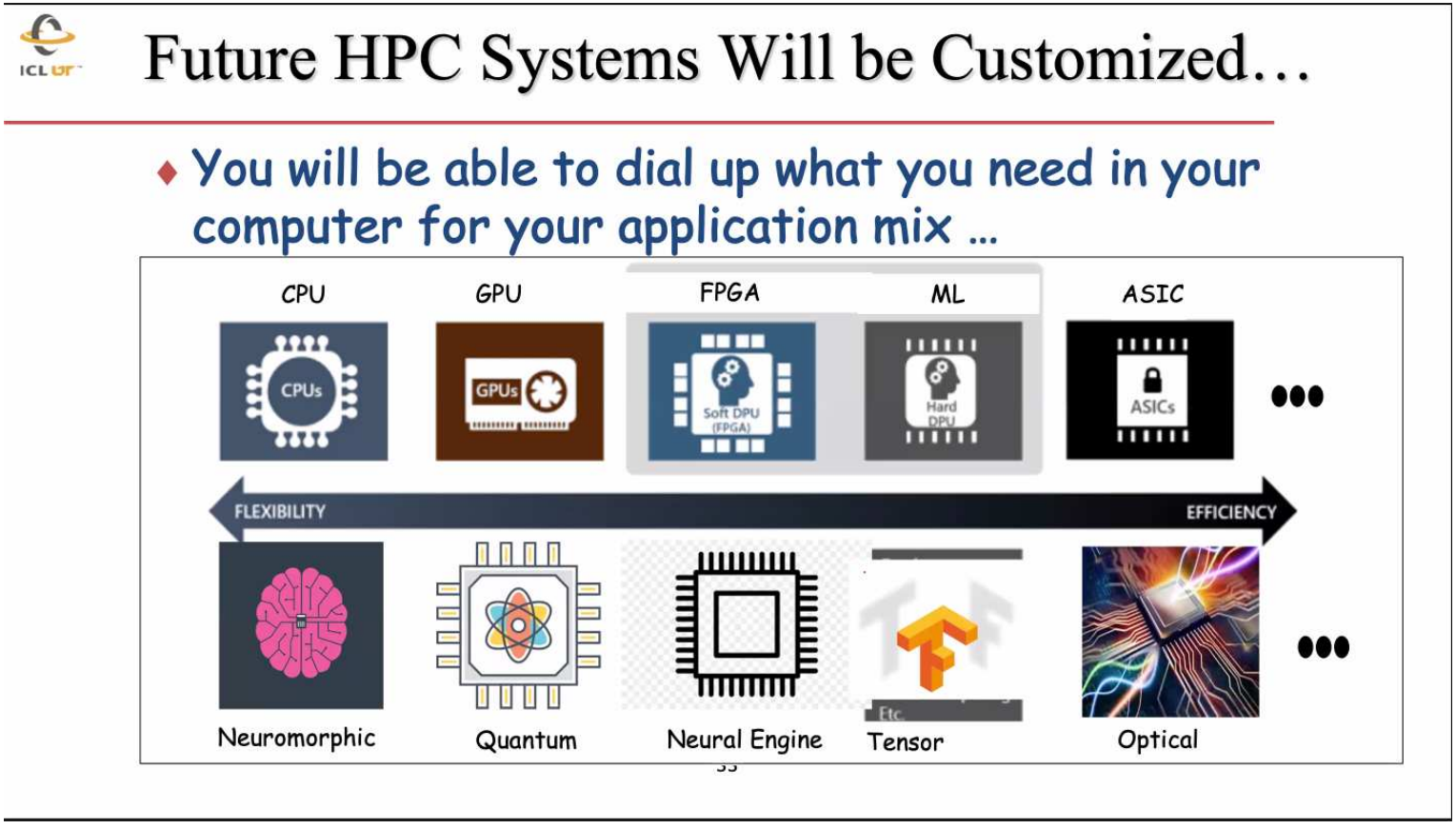


Figure: XuLA 2 experimental board (free range factory)

FPGA in future HPC Systems



(source Jack Dongarra)

FPGA design and programming

- HDL (Hardware Description Language) : **Verilog**, **VHDL**, suited to work with large structures,
- Schematic design, easier visualisation of a design.

$=>$ VHDL

VHDL - Outline

- History
- A first example: a full-adder
- VHDL language
- Concurrent domain and Sequential domain
- VHDL generics

Some milestones

- VHDL - VHSIC Hardware Description Language
  VHSIC - Very High Speed Integrated Circuits
- 1980 - developed at the behest of the U.S Department of Defense
- language influenced by ADA (concepts and syntax)
- used to describe digital and mixed-signal systems such as FPGA and integrated circuits
- can also be used as a general purpose parallel programming language

Advantages

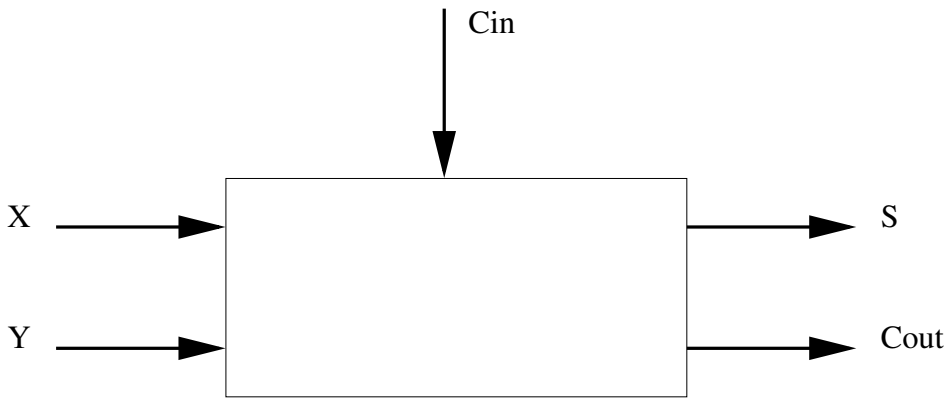
- allows to described (modeled) the behavior of a system and to verified (simulated) it before synthesis that translates the design into real hardware (wires and gates)
- allows the description of a concurrent system (dataflow language)
- modular and hierarchical conception
- portability, reusability
- strongly typed, genericity

Standardization

IEEE standard 1076 defines the VHSIC Hardware Description Language or VHDL

- 1076-1987 - First standardized revision of version 7.2 of the language from the United States Air Force.
- 1076-1993 - Significant improvements. **Probably the most widely used version with the greatest vendor tool support**.
- 1076-2002 - Minor revision. Introduces the use of protected types.
- 1076-2002 - Minor revision of 1076-2000. (buffer ports).
- 1076-2008 - Major revision released on 2009-01-26 (external signals).
- 1076-2019 - `http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/VHDL2017`

A first example: a full-adder

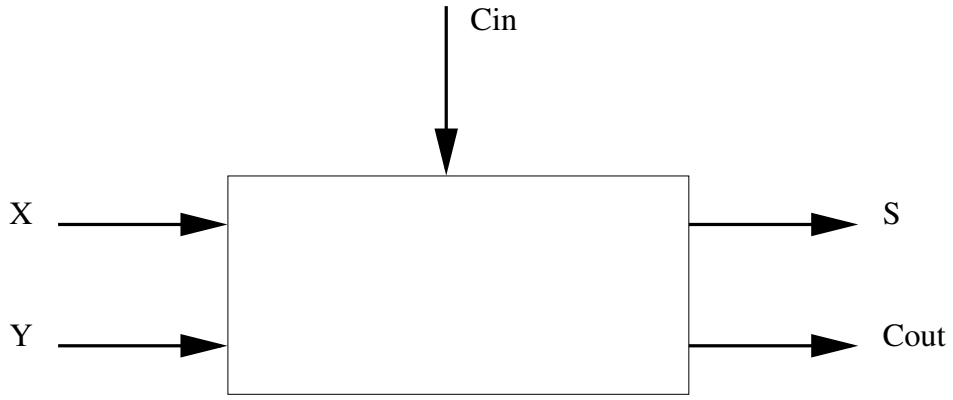


Programming structures

- VHDL is a modular language
- an electronic component is described by a VHDL module that is decomposed in two structures:
  - **entity**
  - **architecture**

The entity

- ”inside” view of a module’s interface: what are the input and output signals




```
entity full_adder is
  port (
        X, Y, Cin : in std_logic;
        S, Cout   : out std_logic
       );
end full_adder;
```

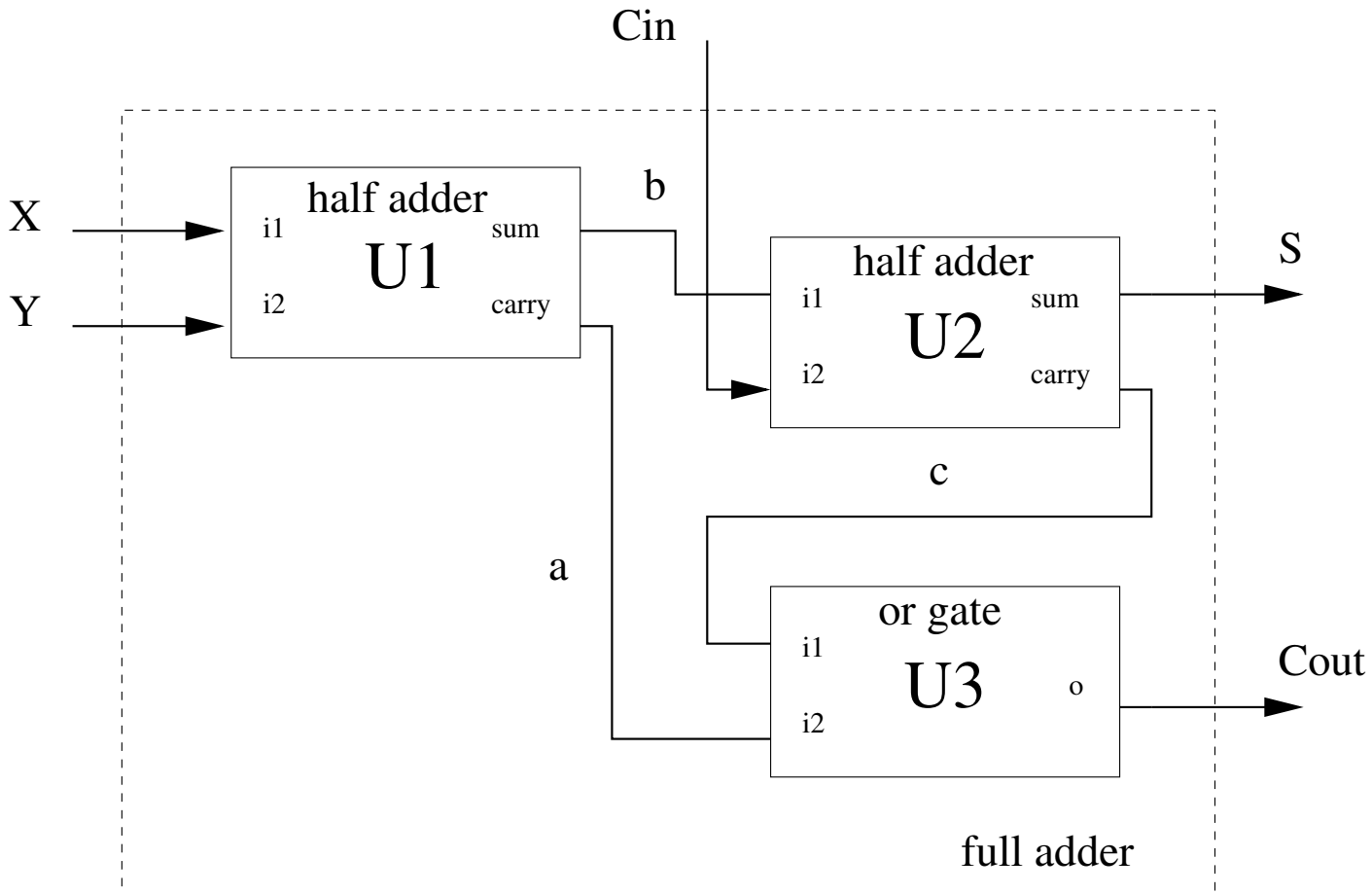
The architecture

- defines how a module works: what are the values of the output signals from the values of the input signals and, if necessary, an internal state
- it contains all the internal signals and sub-components and all the internal logic

The architecture

- different descriptions of the architecture ($\Rightarrow$ several architectures):
  - structural description: interconnected sub-modules
  - data flow description: signals computed by logic or arithmetic equations
  - behavioural description: algorithm to describe the behaviour of the module
- all the types of description can be combined in the same architecture.

Structural description of the full-adder



Structural description : VHDL code

```
architecture structural of full_adder is
  component half_adder  -- "external" view of the module
    port( i1, i2       : in  std_logic;
          carry, sum   : out std_logic);
  end component;
  component or_gate
    port( i1, i2 : in  std_logic;
          o      : out std_logic);
  end component;
  signal a, b, c : std_logic; -- internal signals
  begin
  -- "instance" statements
    U1 : half_adder port map ( X, Y, a, b );
    U2 : half_adder port map ( i1 => b, i2 => Cin,
                               carry => c, sum => S );
    U3 : or_gate    port map ( i2 => a, i1 => c,
                               o => Cout );
end structural;
```

Structural description : VHDL code

```
architecture structural of full_adder is
  component half_adder  -- "external" view of the module
    port( i1, i2       : in  std_logic;
          carry, sum   : out std_logic);
  end component;
  component or_gate
    port( i1, i2 : in  std_logic;
          o      : out std_logic);
  end component;
  signal a, b, c : std_logic; -- internal signals
  begin
  -- "instance" statements: can be written in any order
    U2 : half_adder port map ( i1 => b, i2 => Cin,
                               carry => c, sum => S );
    U1 : half_adder port map ( X, Y, a, b );
    U3 : or_gate    port map ( i2 => a, i1 => c,
                               o => Cout );
end structural;
```

Structural description : VHDL code

```
architecture structural of full_adder is
  component half_adder  -- "external" view of the module
    port( i1, i2       : in  std_logic;
          carry, sum   : out std_logic);
  end component;
  component or_gate
    port( i1, i2 : in  std_logic;
          o      : out std_logic);
  end component;
  signal a, b, c : std_logic; -- internal signals
  begin
  -- "instance" statements: can be written in any order
    U3 : or_gate    port map ( i2 => a, i1 => c,
                               o => Cout );
    U2 : half_adder port map ( i1 => b, i2 => Cin,
                               carry => c, sum => S );
    U1 : half_adder port map ( X, Y, a, b );
end structural;
```

Data flow description of the full-adder

- logical equations
  - $i = X \bigoplus Y$
  - $S = i \bigoplus Cin$
  - $Cout = X.Y + i.Cin$

- VHDL code

```
architecture data_flow of full_adder is
  signal i : std_logic;
  begin
  -- "instance" statements
     i <= X xor Y after 3 ns;
    S <= i xor Cin after 3 ns;
     Cout <= (X and Y) or (i and Cin) after 6 ns;
end data_flow;
```

Data flow description of the full-adder

- logical equations
  - $i = X \oplus Y$
  - $S = i \oplus Cin$
  - $Cout = X.Y + i.Cin$

- VHDL code

```
architecture data_flow of full_adder is
  signal i : std_logic;
  begin
  -- "instance" statements: can be written in any order
    S <= i xor Cin after 3 ns;
    i <= X xor Y after 3 ns;
    Cout <= (X and Y) or (i and Cin) after 6 ns;
end data_flow;
```

Data flow description of the full-adder

- logical equations
  - $i = X \bigoplus Y$
  - $S = i \bigoplus Cin$
  - $Cout = X.Y + i.Cin$

- VHDL code

```
architecture data_flow of full_adder is
  signal i : std_logic;
  begin
  -- "instance" statements: can be written in any order
    Cout <= (X and Y) or (i and Cin) after 6 ns;
    S <= i xor Cin after 3 ns;
    i <= X xor Y after 3 ns;
end data_flow;
```

Behavioural description of the full-adder

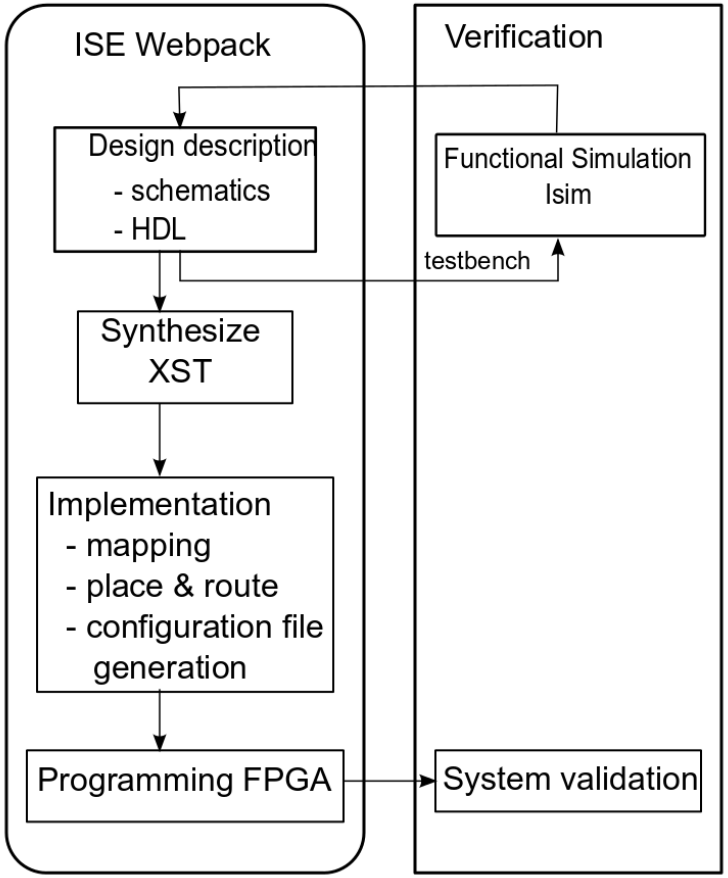
Algorithm that computes the values of the output ports from the values of the input ports.

Truth table

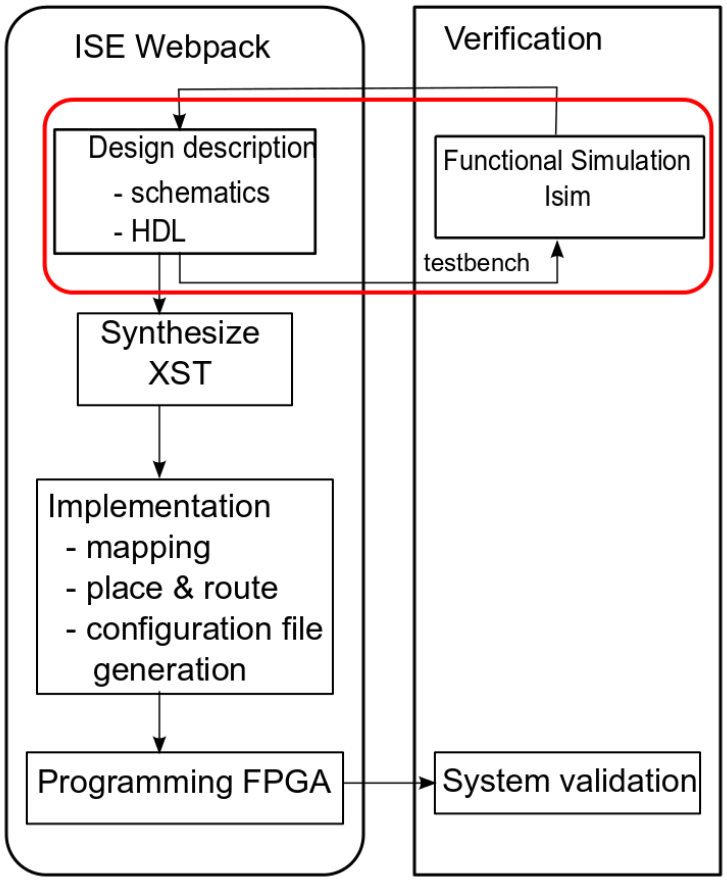
X	Y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
-- VHDL code
architecture behavioural of full_adder is
  begin
  process (X, Y, Cin)
    variable N : natural;
    constant sum_vector :
       std_logic_vector(0 to 3) := "0101";
    constant carry_vector :
       std_logic_vector(0 to 3) := "0011";
    begin
      N := 0;
       if X = '1' then N := N + 1; end if;
       if Y = '1' then N := N + 1; end if;
       if Cin = '1' then N := N + 1; end if;
      S <= sum_vector(N) after 4 ns;
      Cout <= carry_vector(N) after 6 ns;
    end process;
end behavioural;
```

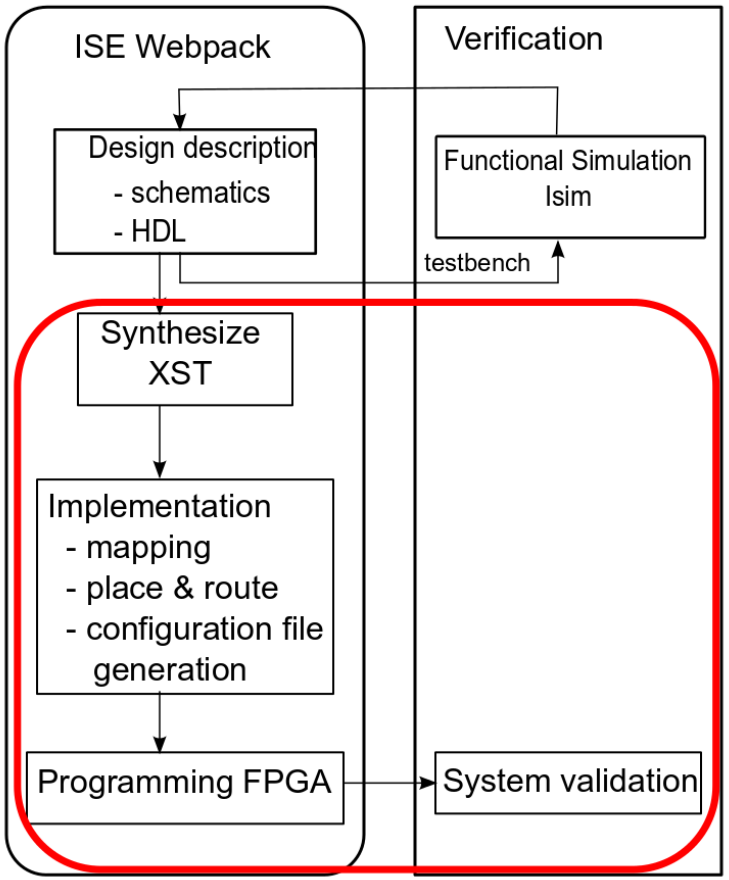
Design flow with ISE Webpack Design of Xilinx



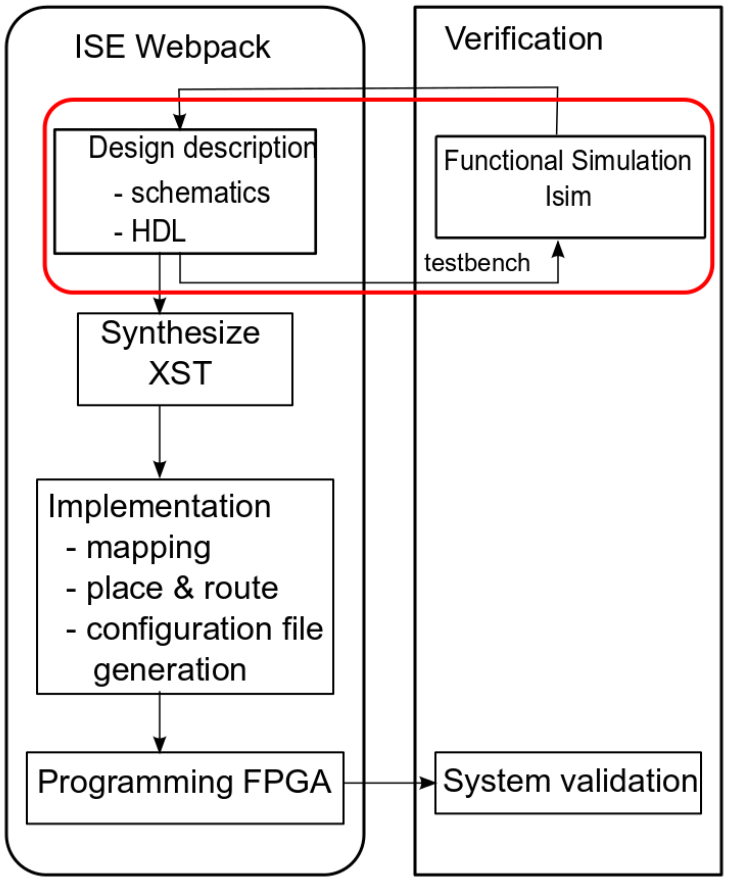
Design flow with ISE Webpack Design of Xilinx



Design flow with ISE Webpack Design of Xilinx



Design flow with ISE Webpack Design of Xilinx



Verify a module: the testbench

- before translating the VHDL module into real hardware, it needs to be tested and verified: this is the **simulation** phase
- we use a special VHDL module: the testbench
- the testbench is described by an entity with no port and by an architecture
- the component to be tested is a sub-component of the testbench
- we give values to the input signals (value/date) and verify the values of the output signals
- the testbench should be exhaustive

full-adder testbench: code

```
entity test_full_adder is
end test_full_adder;

architecture arch of test_full_adder is

  component full_adder
    port (
      X, Y, Cin : in std_logic;
      S, Cout   : out std_logic
    );
  end component;

  signal A, B, CarryIn : std_logic;
  signal Sum, CarryOut : std_logic;
```

full-adder testbench: code (cont.)

```
begin

 a1 : full_adder
   port map( X => A, Y => B, Cin => CarryIn,
             S => Sum, Cout => CarryOut);

 A <= '0', '1' after 20 ns, '0' after 40 ns,
      '1' after 60 ns, '0' after 80 ns, '1' after 100 ns,
      '0' after 120 ns, '1' after 140 ns,
      '0' after 160 ns;

 B <= '0', '1' after 40 ns, '0' after 80 ns,
      '1' after 120 ns,'0' after 160 ns ;

 CarryIn <= '0', '1' after 80 ns;

end arch_test;
```

Simulation: ISim software of Xilinx

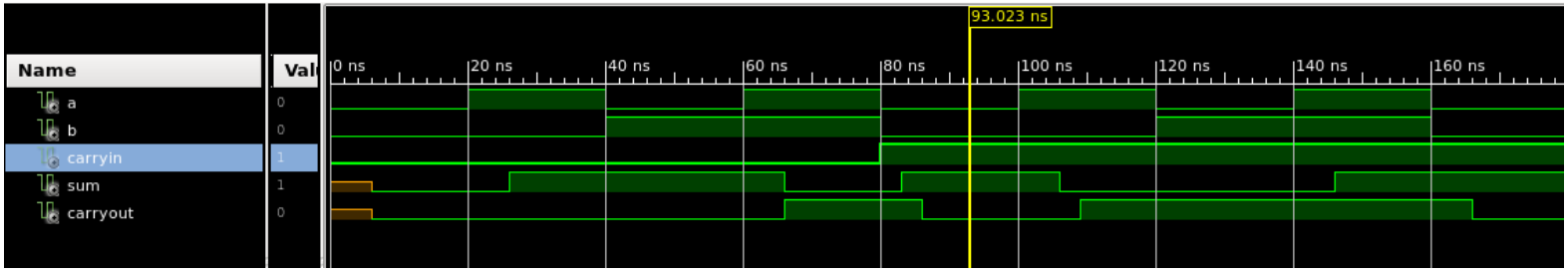


Figure: simulation of data flow architecture

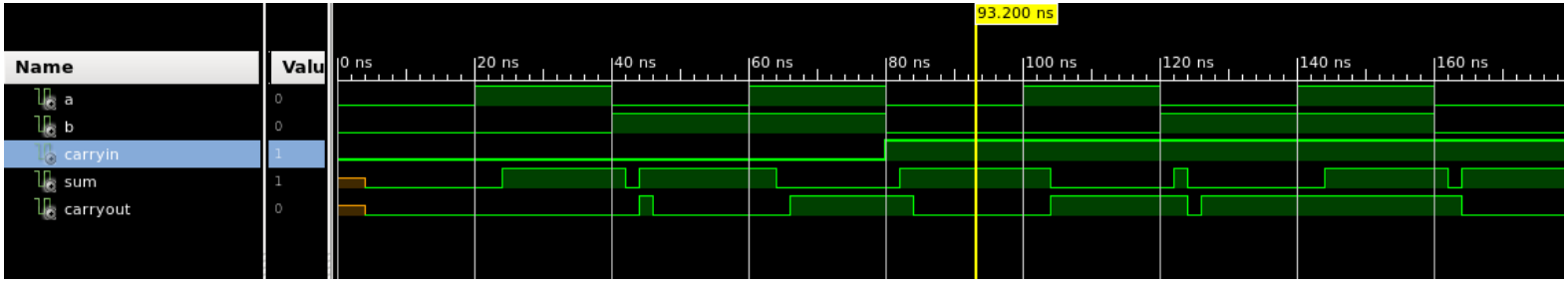
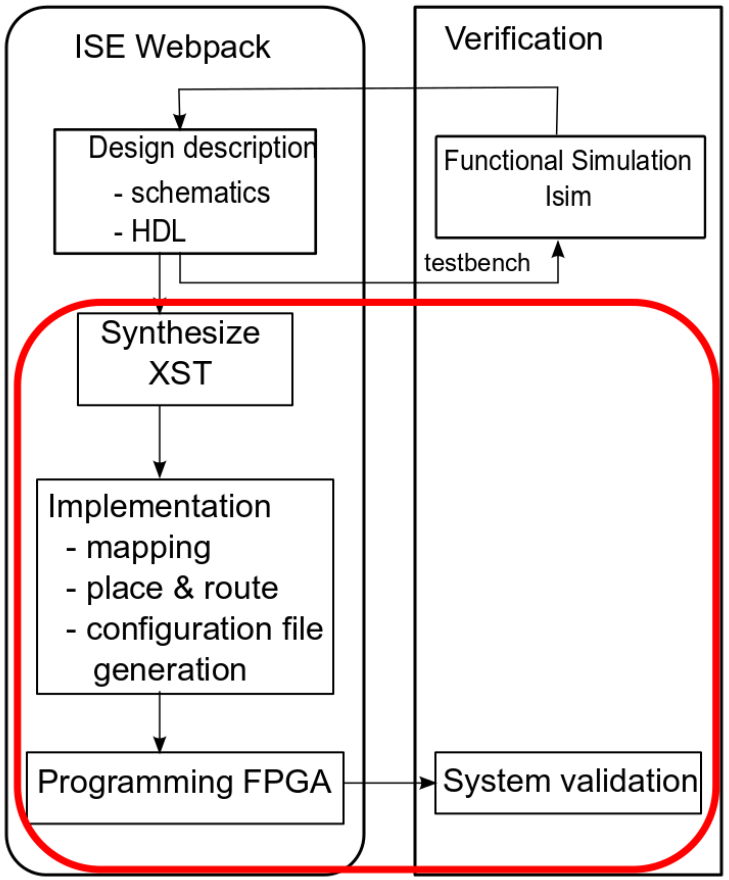


Figure: simulation of structural architecture

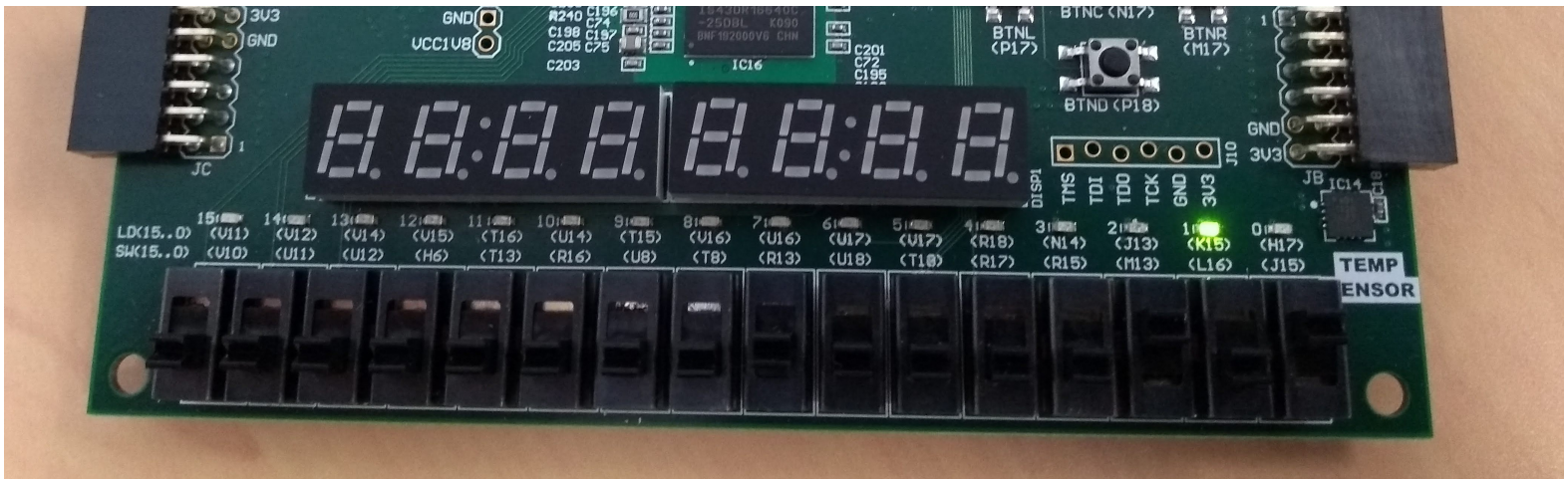
Design flow with ISE Webpack Design of Xilinx



Programming FPGA: ISE Webpack Suite

Nexys 4 Board with Artix-7 FPGA

- choose the ports of the board to connect to the ports of the component (switches/LEDs) [**ucf** file]
- generate the configuration/programming file
- transfer this file to the board
- play with the switches to check the good behaviour



VHDL language

- case insensitive
- comments: two dashes ”- -” start the comment until the end of the line
- identifiers start with a letter
- operators
- literals
- objects

Operators

- Lowest precedence first
  1. logical: **and, or, nand, nor, xor**
  2. relational: $=, /=, <, <=, >, >=$
  3. shift, rotation: **sll, srl, sla, sra, rol, ror**
  4. addition, concatenation: **+, -, &**
  5. unary: **+, -**
  6. multiplication: ***, /, mod, rem**
  7. exponentiation, absolute value, complement: ****, abs, not**

Literals

- classification:
  - numeric
    - decimal notation: integer (1345 or 1_345) or real (13.0)
    - based notation: 16#FB# ou 2#01000100#
  - character, string ('a', "Hello", "1234")
  - enumeration
  - bit, bit vector ('0', '1', "1001")

Objects

- an object contains a value
- there are four classes of object in VHDL:
  1. a **constant** has a unique fixed value
  2. a **signal** memorises the history of its past values (value, date), its current value and its predicted future values (value, date);
     only future values can be modified by the assignment of the signal
  3. a **variable** has a unique value that can be modified
  4. a **file** contains a sequence of values that can be read or written (memory, initial program)
- each object has a unique type

Types

- VHDL is strongly typed
- a type is an ordered set of values
- the type of an object is **static**; it can not be modified
- it is possible to define new types or sub-types from existing types and type constructors
- classification:
  - scalar types
  - composite types (array, record)

STD_LOGIC type

STD_LOGIC type (standard logic) is the type that we will use almost exclusively.

- it is defined in the package `std_logic_1164` of the library IEEE
- it has five values :
  - '1' - logical 1
  - '0' - logical 0
  - 'Z' - high impedance
  - 'U' - un-initialised (simulation)
  - 'X' - could not be resolved (simulation)

The associated array type: STD_LOGIC_VECTOR

- **signal** BusA : `std_logic_vector`(7 **downto** 0);

We will also use NATURAL type for convenience.

STD_LOGIC handling

- declaration/initialisation:

```
signal A, B : std_logic; -- default value = 'U'
signal C : std_logic_vector(3 downto 0) := "0011";
```

- assignment:

```
A <= '0';
C <= "0100";
C <= (others => '0'); -- aggregate
```

- concatenation:

```
C <= A & B &"01";
```

- access to a bit

```
A <= C(0); -- LSB
C(3) <= B; -- MSB
C <= (3 => '1', 1 => '1', others => '0'); -- aggregate
```

STD_LOGIC packages

- defined in package `std_logic_1164` of library `IEEE`; the following lines must be added before the entity

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

- useful packages:

```
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

Allow arithmetic operations between std_logic and integer for instance.

```
signal A : std_logic_vector(3 downto 0);
....
A <= A + 1; -- gives "0000" when A is "1111"
```

Composite types

A composite type object is one having multiple elements. There are two classes of composite types.

- **array types**: all elements of an array must be of the same type
- **record types**: elements of a record can be of different types

Arrays

- constrained
  - **type** word **is array** (0 **to** 31) **of** `std_logic`;
  - **signal** a_word : word;
  - **NOT ALLOWED:** [**signal** a_word : **array** (0 **to** 31) **of** `std_logic`;]
- unconstrained
  - **type** `std_logic_vector` **is array**(`natural` **range** <>) **of** `std_logic`;
  - The size of a particular object is specified only when it is declared.
    signal bus : `std_logic_vector`(31 **downto** 0);
- Array elements are referenced by indices and can be assigned values individually or using concatenation, aggregates, ...

```
signal lsb : std_logic;
lsb <= bus(0);
```

Record Types

- elements can be of different types
- elements (fields) are named
- example

```
type memory_cell is record
  value : word;
  valid : boolean;
end record;
signal A : memory_cell;
variable B : memory_cell;
```

- `A.value` is of type word, `A.valid` of type boolean
- Assignments

```
A <= B;
A.value <= B.value after 10 ns;
-- and using aggregates
```

Enumeration Types

- The Enumeration type is a type whose values are defined by listing (enumerating) them explicitly. This type values are represented by enumeration literals,
- It is assumed that the values are defined in ascending order. For this reason it is recommended to order the literals in such a way that the default value is the first one.

```
type couleur is (bleu, vert, rouge);
type boolean is (false, true);
type bit is ('0', '1');
type etat is (idle, emission, attente_ack);
```

Aggregates

- A basic operation that combines one or more values into a composite value of a record or array type
- Aggregates are composed of element associations, which associate expressions to elements (one expression per one or more elements)
- Element associations are specified in parentheses and are separated by commas.

Aggregates examples

```
type tab is array (1 to 3) of integer;
type art is record
  field1 : natural;
  field2 : std_logic;
  field3 : natural;
end record;
signal A : tab; signal B : art;
```

Elements can be referred to either by different associations:

- positional associations

  ```
  A <= (12, 13, 14); B <= (23, '0', 45);
  ```

- named associations

  ```
  A <= (1 => 12, 3 => 14, 2 => 13);
  ```

  ```
  B <= (field3 => 45, field1 => 5, field2 => '0');
  ```

- mixed associations

  ```
  A <= (5, others => 0); B <= (field2 => '0', others => 0);
  ```

  ```
  A <= (others => 0);
  ```

Subtypes

- A type together with a constraint
- A value belongs to a subtype of a given type if it belongs to the type and satisfies the constraint
- The given type is called the base type of the subtype
- examples

```
subtype integer_8bits is integer range 0 to 255;
subtype natural is integer range 0 to integer'high;
```

- A subtype is the same type as its base type
- No type conversion is needed when objects of a subtype and its base type are assigned (in either direction)
- The set of operations allowed on operands of a subtype is the same as the set of operations on its base type

Programming FPGA with VHDL
An introduction – Part 2

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

2024 - Sciences du Numérique
Parcours Architecture, Systèmes et Réseaux

VHDL - Outline

- History
- A first example: a full-adder
- VHDL language
- **Concurrent domain and Sequential domain**
- **VHDL generics**

Concurrent domain and sequential domain

- the description of a hardware system is concurrent
- a **VHDL description is concurrent** (unless you are inside certain constructs):
  the code lines are executed almost all at once.
- we differentiate:
  - the concurrent domain (**body of an architecture**)
  - the sequential domain (**body of a process**)

Concurrent domain

- In the concurrent domain we compute the values of signals
- The value of a signal is given by a **unique** concurrent statement:
  - a signal's assignment,
  - a sub-component instance where the signal is connected to an output port
  - a process that computes the signal's value

Signal and signal's assignment

- a signal corresponds in VHDL to the hardware support of information (wire, bus)
- **a signal has a time evolution** : past values (value,date), current value, predicted values (value, date)
- a signal is declared as port of an entity or in the declaration zone of an architecture

Simple assignment statement

```
S <= '1', '0' after 10 ns;
B <= A xor C after 15 ns;
D <= not A;
```

- assignment <=
- compatibility of types between the signal and the right expression
- right expression with a value and a delay (time type)
- this delay can be omitted (**delta delay** notion to handle this situation)

Simple assignment: execution

```
A <= not B after 10 ns;
C <= A xor D after 5 ns;
```

- before T0 : A = '1', B= '0', C = '1', D = '0'
- at *T*0, the value of *B* becomes '1' and the value of *D*, '1' (Assignment)
- Evaluation
- Prediction
- Assignment
- etc ...

Simple assignment: execution

we change the delay of A's assignment:

```
A <= not B after 5 ns;
C <= A xor D after 5 ns;
```

- before T0 : A = '1', B= '0', C = '1', D = '0'
- at *T*0, the value of *B* becomes '1' and the value of *D*, '1' (Assignment)
- Evaluation
- Prediction
- Assignment
- etc ...

Maintaining a signal

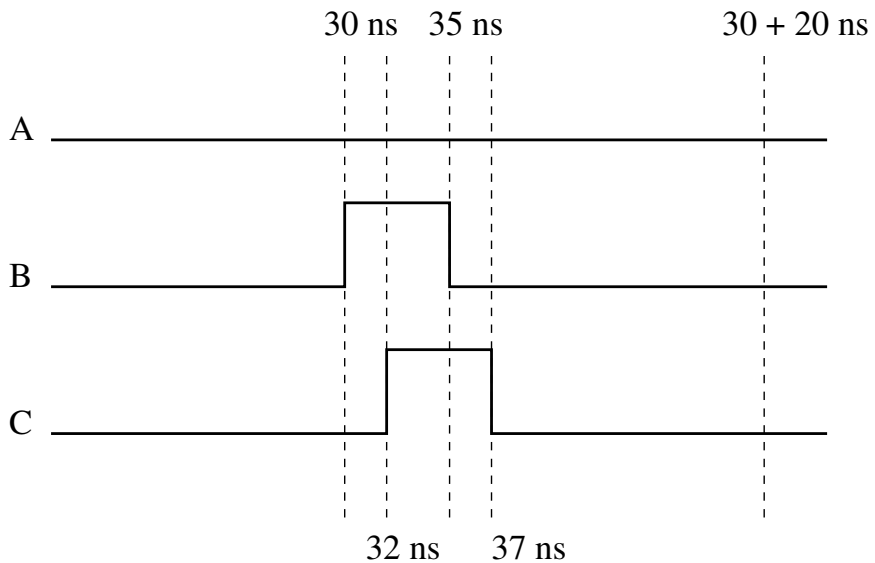
- be careful with the duration of the value of a signal from the right part of the assignment
- example

```
A <= B after 20 ns;
B <= '0', '1' after 30 ns, '0' after 35 ns;
C <= B after 2 ns;
```

Maintaining a signal

- be careful with the duration of the value of a signal from the right part of the assignment
- example

```
A <= B after 20 ns;
B <= '0', '1' after 30 ns, '0' after 35 ns;
C <= B after 2 ns;
```

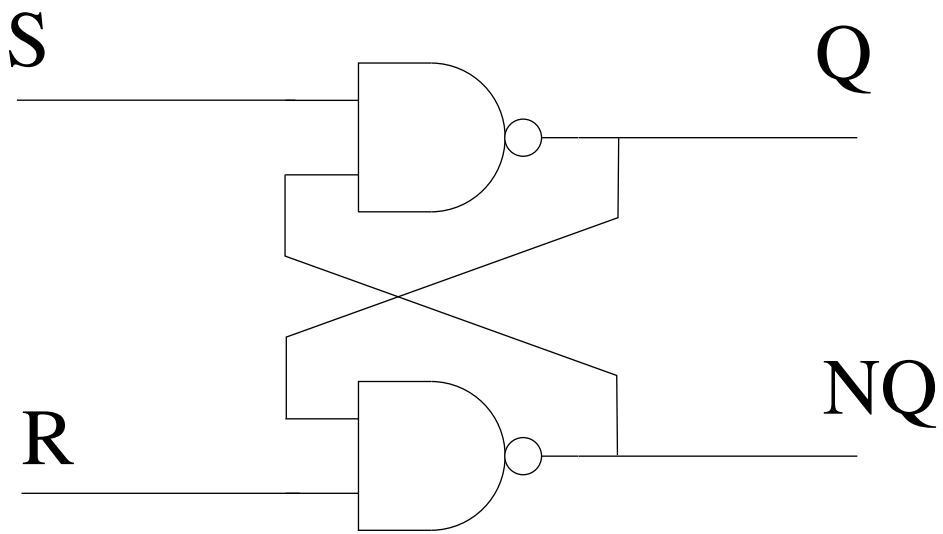


Assignment with no time delay

```
A <= B;
A <= B after 0 ns;
```

- signal A is assigned with the current value of signal B after a **delta delay**
- delta delay models causality; it is equivalent to zero delay for the simulation
- it is useful for the simulation tool in order to manage, for a given time of the simulation, a variable number of infinitesimal slices of time;
- it permits to manage a set of assignments occuring at the same time.
- example : RS flip-flop

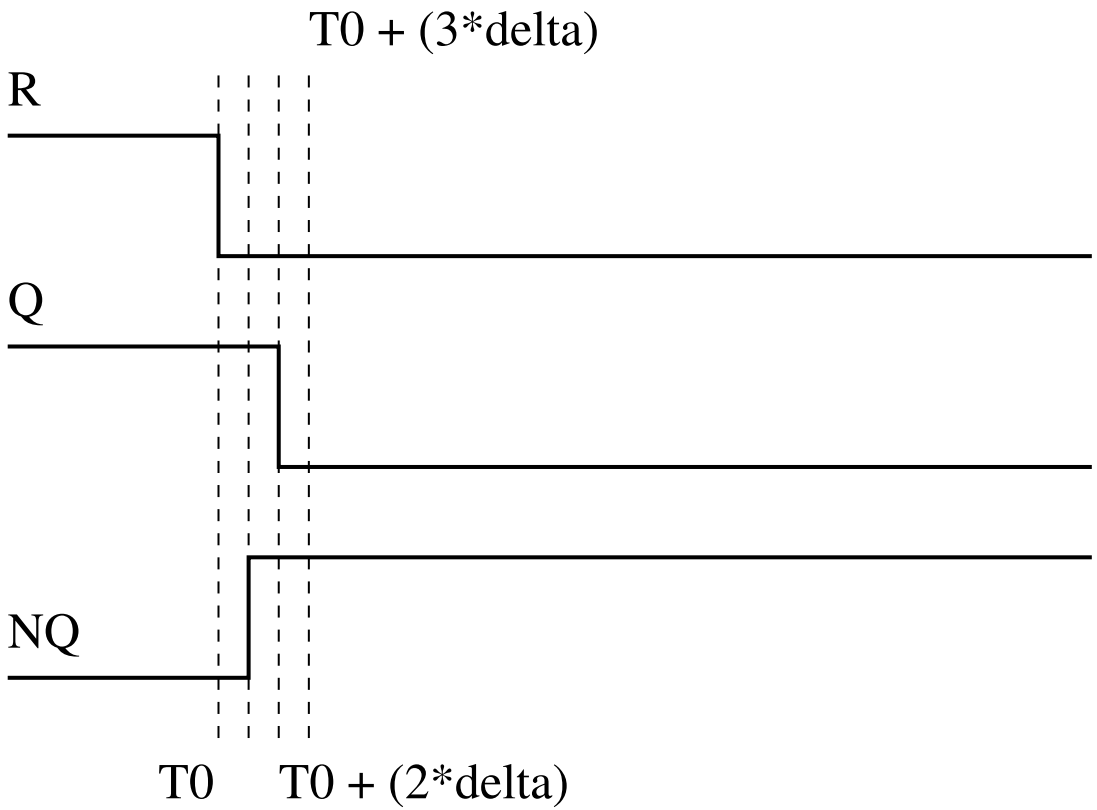
Example of an assignment with no time delay



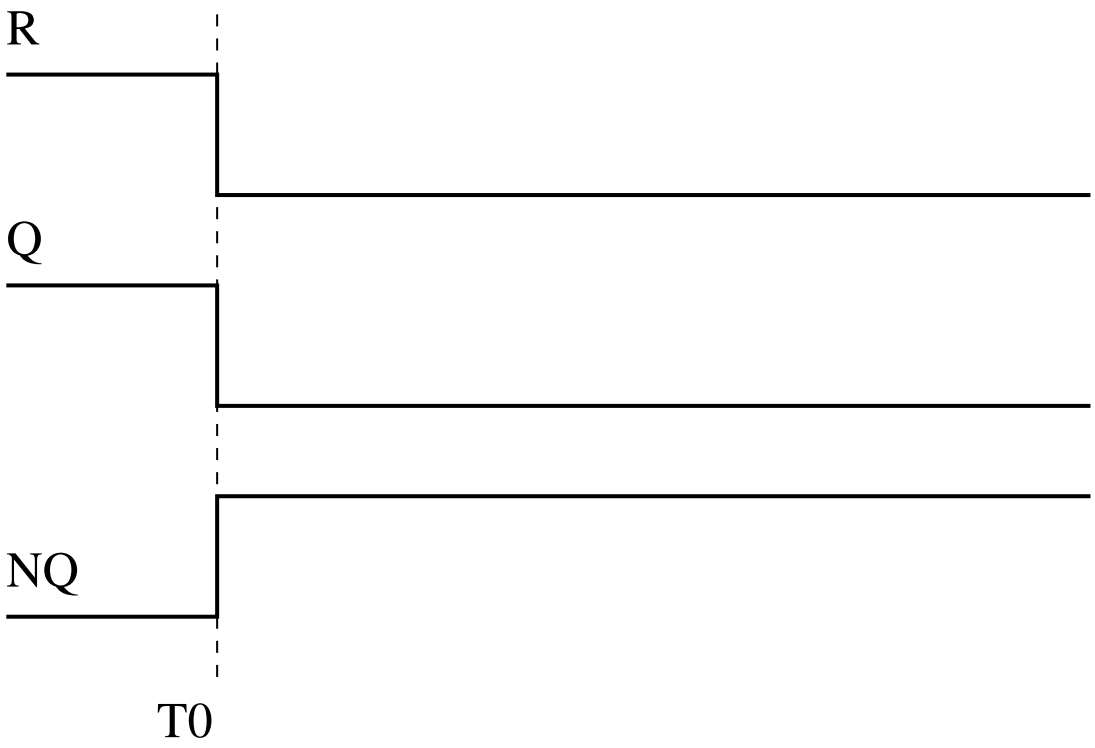

```
Q  <= S nand NQ;
NQ <= R nand Q;
```

- steady flip-flop with R, S, Q equal to '1' and NQ equals to '0'
- let's suppose that R becomes '0' at T_0

How to use delta delay for a good reasoning?

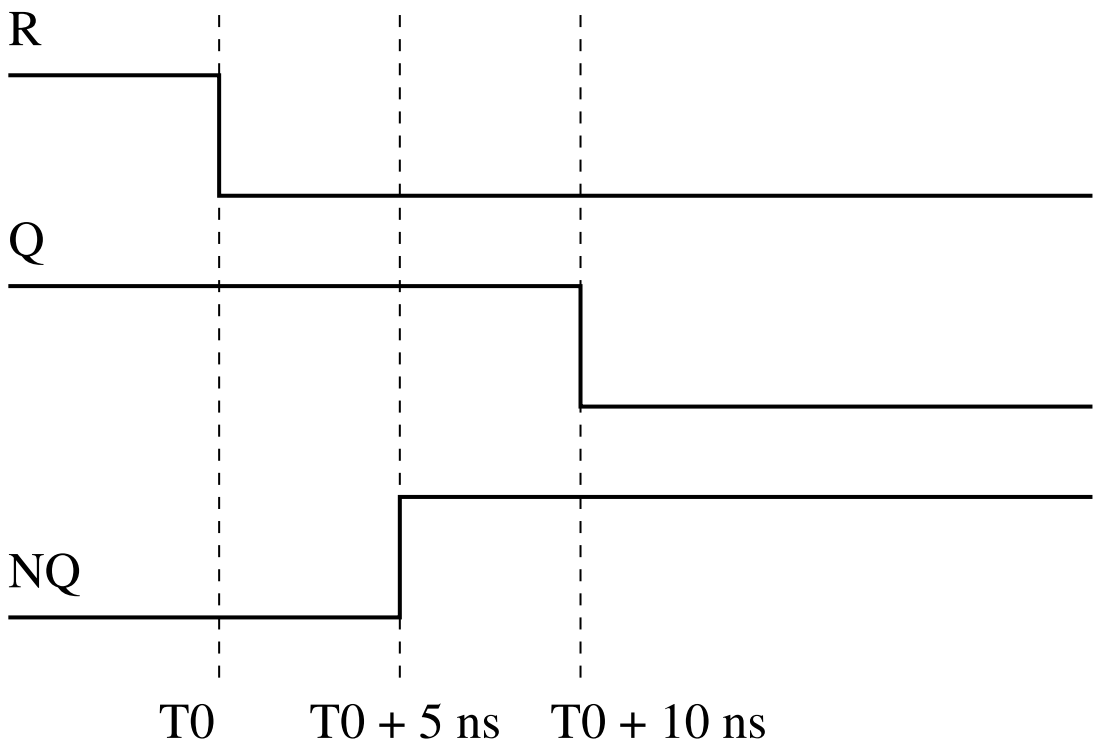


Simulation

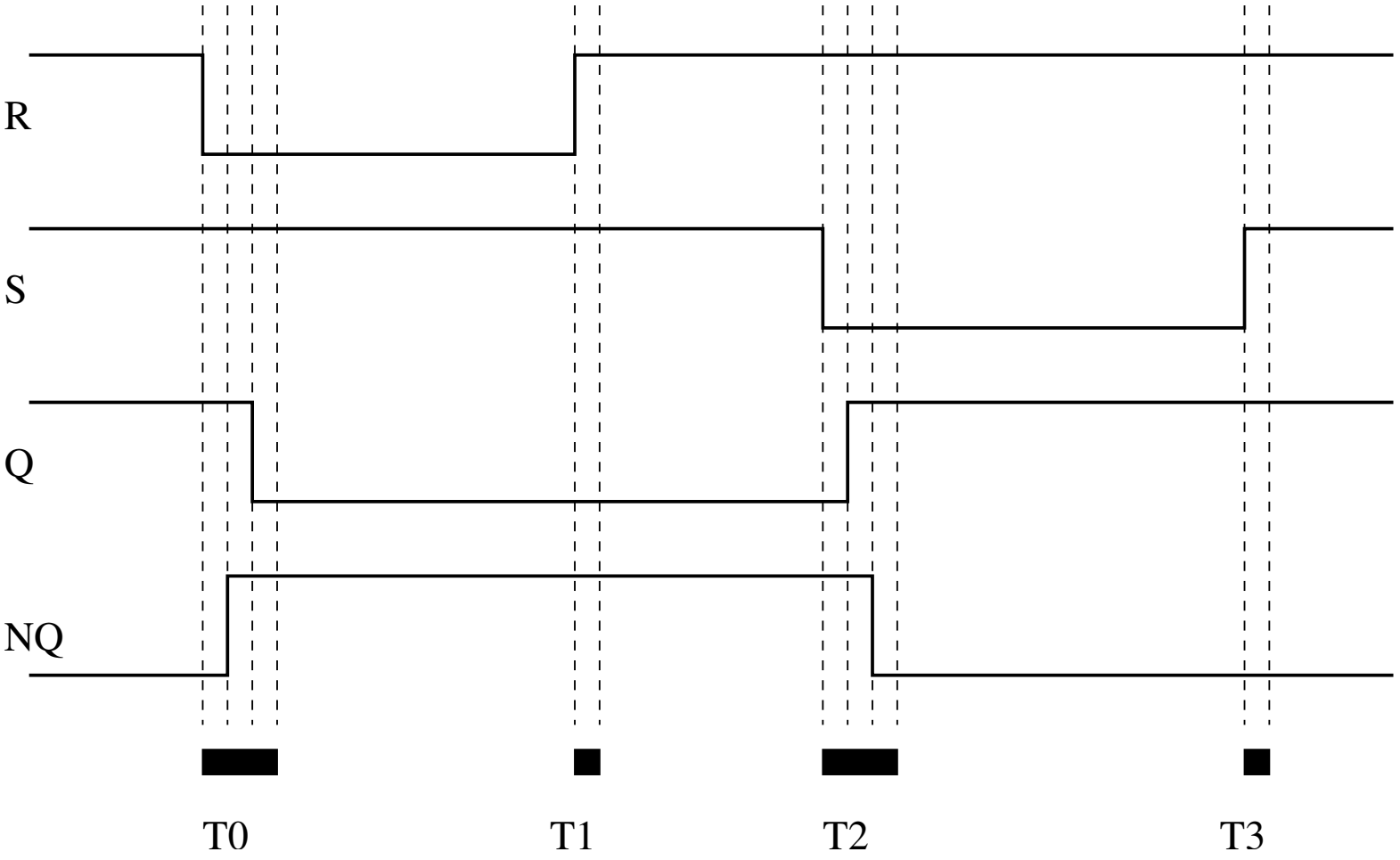


Realistic behavioral

```
Q  <= S nand NQ after 5 ns;
NQ <= R nand Q after 5 ns;
```



Waveform for all evolutions



Conditional signal assignment statement

exclusively in concurrent domain

```
T <= A after 20 ns when (X < 10) else
     B after 10 ns;
S <= 5 when A = '1' and B = '0' else
     4 when A = '1' else
     C;
```

Selected signal assignment statement

exclusively in concurrent domain

```
signal opcode : std_logic_vector(1 downto 0);
signal result, A, B : std_logic;
  ...
with opcode select
   result <= A and  B when "00",
             A or   B when "01",
             A nand B when "10",
             A nor  B when "11",
             'X' when others;   -- all values have
                                -- to be considered
```

Process statement

- **a process contains sequential statements**: each statement is executed one after another
- a process can handle signals and variables;
- a variable belongs to a unique process (declaration, assignment)
- a process is **cyclic**
- a process is sensitive to signal(s) (**sensitivity list**) or (**exclusive**) can be blocked by a **wait** statement

Syntax

```
{label:} process {(sensitivity_list)}
    { process_declarative_items }
    begin
      sequential statements
    end process {label}
```

Sensitivity list

- at each event on one of the signals of the sensitivity list, the process is executed (from the first statement to the last one) and is blocked until a new event
- a process without a sensitivity list has to be blocked (unless the code turns eternally without progressing in time): **wait** sequential statement
- sensitivity list and **wait** statement are exclusive

Example of process (1)

```
process(X, Y, Cin)
  variable N : natural;
  constant sum_vector : std_logic_vector(0 to 3) := "0101";
  constant carry_vector : std_logic_vector(0 to 3) := "0011";
  begin
    N := 0;
    if X = '1' then N := N + 1; end if;
    if Y = '1' then N := N + 1; end if;
    if Cin = '1' then N := N + 1; end if;
    S <= sum_vector(N) after 4 ns;
    Cout <= carry_vector(N) after 6 ns;
  end process;
```

Example of process (2)

```
process
  variable N : natural;
  constant sum_vector : std_logic_vector(0 to 3) := "0101";
  constant carry_vector : std_logic_vector(0 to 3) := "0011";
  begin
    N := 0;
    if X = '1' then N := N + 1; end if;
    if Y = '1' then N := N + 1; end if;
    if Cin = '1' then N := N + 1; end if;
    S <= sum_vector(N) after 4 ns;
    Cout <= carry_vector(N) after 6 ns;
    wait on X, Y, Cin;
  end process;
```

Example of process (3)

```
process
  variable N : natural;
  constant sum_vector : std_logic_vector(0 to 3) := "0101";
  constant carry_vector : std_logic_vector(0 to 3) := "0011";
  begin
    wait on X, Y, Cin;
    N := 0;
    if X = '1' then N := N + 1; end if;
    if Y = '1' then N := N + 1; end if;
    if Cin = '1' then N := N + 1; end if;
    S <= sum_vector(N) after 4 ns;
    Cout <= carry_vector(N) after 6 ns;
  end process;
```

Example of process (4): clock generator

```
process
  begin
    clk <= '0';
    wait for period/2;
    clk <= '1';
    wait for period/2;
  end process;
```

Synchronous process

- for designing components driven by a clock (binary signal),
- statements are executed on one of the edge of the clock (rising or falling edge),
- **period to execute a cycle of the process = period of the clock**
- syntax:

```
process (clk)
    -- declarations
begin
  if(clk = '1') then
    -- statements on rising edge
  end if;
end process;
```

Example: a counter

```
process (clk)
  variable cpt_aux : std_logic_vector(3 downto 0)
                     := (others => '0');
begin
  if(clk = '1') then
    cpt_aux := cpt_aux + 1;
    cpt <= cpt_aux;
  end if;
end process;
```

Addition of an asynchronous reset

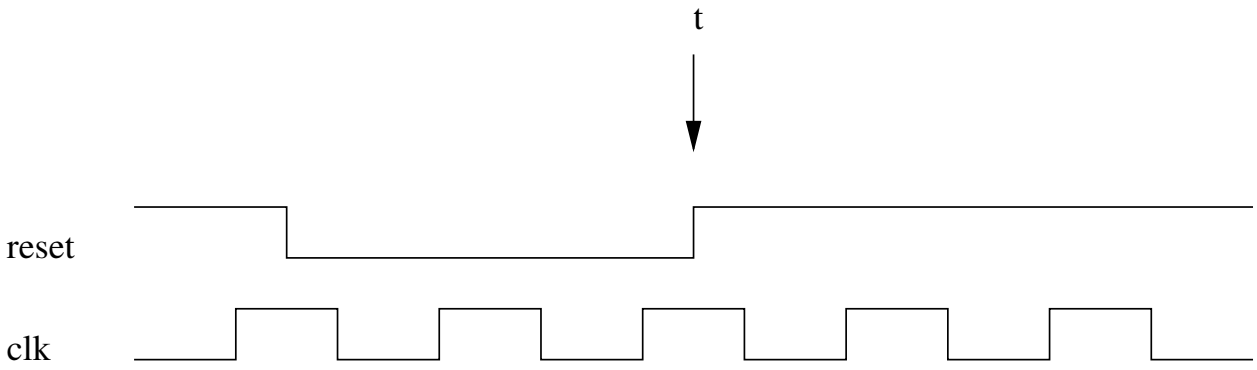
- asynchronous reset that permits to reset the component at any time
- '0' to reset

```
process (clk, reset)
  -- declarations
begin
  if(reset = '0') then
    -- initialisation statements
  elsif(clk = '1' and clk'event) then
    -- statements on rising edge
  end if;
end process;
```

Example: a better counter

```
process (clk, reset)
  variable cpt_aux : std_logic_vector(3 downto 0)
                          := (others => '0');
begin
  if(reset = '0') then
    cpt_aux := (others => '0');
    cpt <= cpt_aux;
  elsif(clk = '1' and clk'event) then
    cpt_aux := cpt_aux + 1;
    cpt <= cpt_aux;
  end if;
end process;
```

why the clk'event



At time *t*, reset rises to '1' but this is not a rising edge !

An alternative to the clk'event and clk='1'

```
process (clk, reset)
  variable cpt_aux : std_logic_vector(3 downto 0)
                          := (others => '0');
begin
  if(reset = '0') then
    cpt_aux := (others => '0');
    cpt <= cpt_aux;
  elsif(rising_edge(clk)) then
    cpt_aux := cpt_aux + 1;
    cpt <= cpt_aux;
  end if;
end process;
```

Sequential statements

The statements we will use:

- **wait**
- conditional structures
- loop structures
- nil statement (**null**)

wait statement

```
-- infinite wait
wait;

-- timeout clause, specific time delay
wait for delay;

-- sensitivity clause
wait on signal_list;

-- condition clause
wait until condition;
```

Conditional statement

- if statement

```
if condition1 then
  sequence_of_statements1
elsif condition2 then
  sequence_of_statements2
else
  sequence_of_statements3
end if;
```

- case statement

```
case expression is
  when valeur1 => sequence_of_statements1
  when valeur2 => sequence_of_statements2
  when others  => null;
end case;
```

Loop

- loop statement

```
loop
   sequence_of_statement
end loop;
```

- for statement

```
for indice in range loop
   sequence_of_statement
end loop;
```

with `range`

```
inf to sup
sup downto inf
```

Signal against Variables

	signal	variable
declaration	entity or architecture	process
assignment	concurrent or sequential domain s <= '1'; delayed	process v := '1'; immediate

A signal assignment is **unique**:

1. concurrent signal assignment
2. output port of a sub-component
3. sequential assignment in a process

VHDL generics

- VHDL generics allow to define a family of components that share the same behavior but with different characteristics (generally sizes, capacity, ...)
- It is very useful but not always convenient to handle with the synthesis tools

An example: And gate with multiple entries

- a new part of the **entity**: the generic parameter

```
entity And_gate is
  generic ( bus_size : natural := 2 );
  port ( A : in  std_logic_vector(bus_size-1 downto 0);
         B : out std_logic);
end And_gate;
```

- the value of the parameter is fixed in the component instantiation statement

```
And_gate4 : And_gate
  generic map( n )
  port map ( alpha, beta );
```

- with

```
constant n : natural := 4;
signal alpha : std_logic_vector(n-1 downto 0);
signal beta : std_logic;
```

An architecture of And_gate

```
architecture arch of And_gate is
  begin

    process(A)
      variable res : std_logic;
      begin
        res := '1';
        for i in bus_size-1 downto 0 loop
        -- or
        -- for i in A'reverse_range loop
            res := res and A(i);
        end loop;
        B <= res;
      end process;
```

Going Further

- Peter J. Ashenden, ”The Designer’s Guide to VHDL, Third Edition (Systems on Silicon)”, 2008, ISBN 0-1208-8785-1.
- FPGA tutorial
  `http://fpga-tutorials.blogspot.fr`