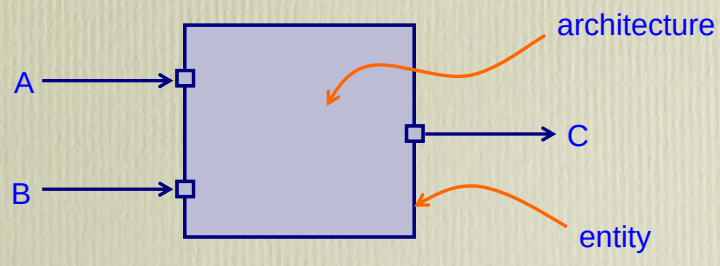
A VHDL review

Eduardo Sanchez
EPFL

Entity and architecture

- A piece of hardware is visualized as a "black box". The interface of the black box is well defined, while its inside is invisible
- In VHDL the black box is known as an entity
- VHDL allows to associate an implementation with the black box, describing its contents: this implementation is called the architecture



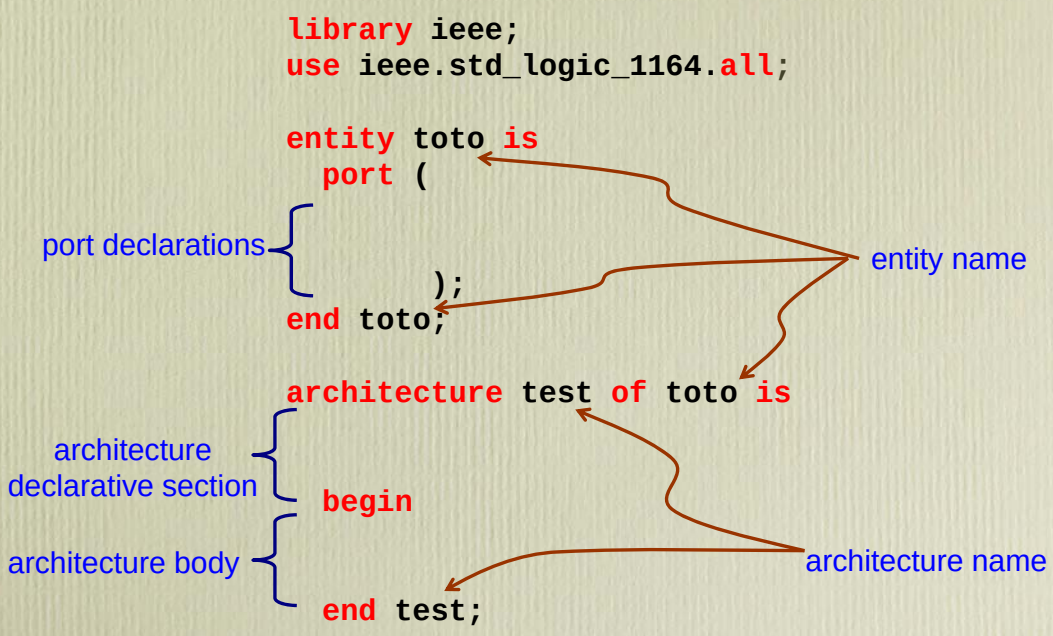
Signals and types

- In order to connect different parts of a design, VHDL uses signals (equivalent to wires in real hardware)
- Every signal in VHDL has a type. For synthesis, the most used types are **`std_logic`**, for 1-bit signals, and **`std_logic_vector`**, for buses. These types are defined in the **`std_logic_1164`** package, which resides in the IEEE library

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


- The std_logic type has nine values:
  - **`'U'`** uninitialized
  - **`'X'`** forcing unknown
  - **`'0'`** forcing 0
  - **`'1'`** forcing 1
  - **`'Z'`** high impedance
  - **`'W'`** weak unknown
  - **`'L'`** weak 0 (pull-down)
  - **`'H'`** weak 1 (pull-up)
  - **`'-'`** don't care
- Most synthesis tools treat these values slightly differently than simulators: **`'L'`** is for 0, **`'H'`** is for 1, **`'-'`** and **`'X'`** for don't care, and **`'-'`** is a wildcard for comparisons
- For assignments, the used values are **`'X'`**, **`'0'`**, **`'1'`**, **`'Z'`**

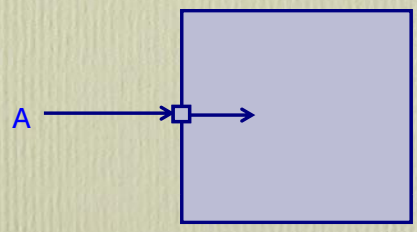
Structure of a VHDL program



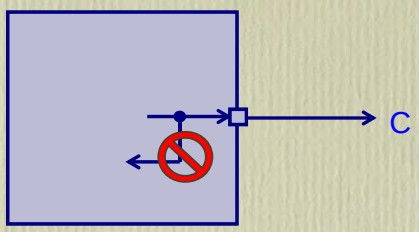
- VHDL is not case sensitive
- VHDL is free format
- Comments are indicated with a double-dash. The carriage return terminates a comment

The entity ports

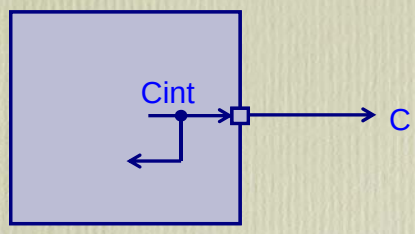
- The signals of an entity are its ports. They have an associated type and a mode, indicating its driver direction and whether or not the port can be read from within the entity

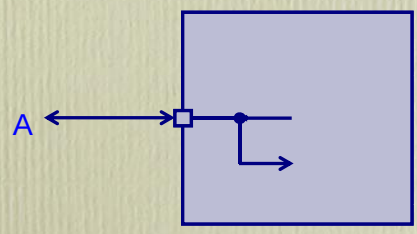


Mode **in**
The port signal can be read within the entity (but it cannot be written)



Mode **out**
The port signal cannot be read within the entity





Mode **inout**
The port signal can be read within the entity

Concurrency

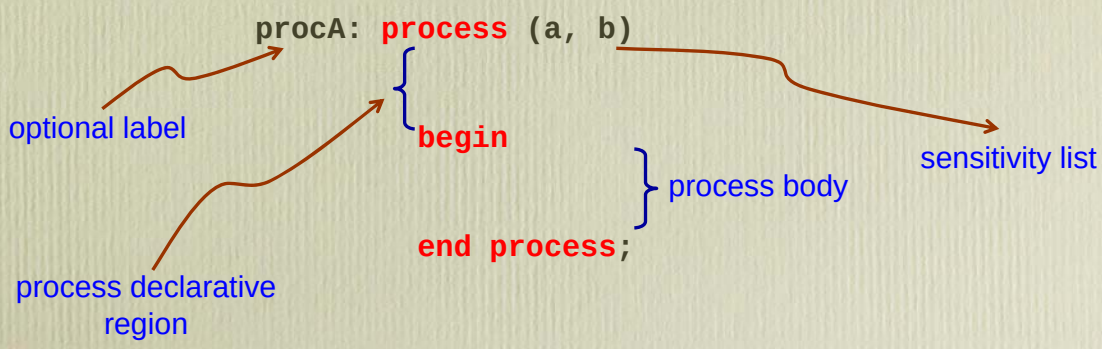
- VHDL supports the notion of "concurrent execution" or "concurrence", with two methods of describing such concurrence inside an architecture: processes and concurrent assignments

```
architecture toto of test is
begin
c <= a and b;
z <= c when oe='1' else 'Z';
seq: process (clk, reset)
     begin
       .
       .
     end process;
end toto;
```

concurrent assignments

sequential statements

- The relative order of concurrent assignments and processes within an architecture is unimportant
- A process is a collection of sequential statements: inside a process, the order of statements is important

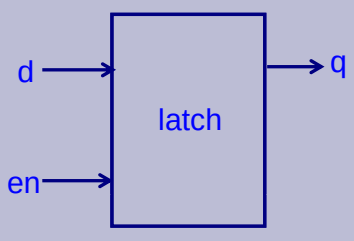


- The sensitivity list is a list of all signals that trigger execution of the process. A process is not executed unless one (or more) of the signals in the sensitivity list changes in value
- The statements of a process are executed simultaneously (during the **`wait`**), even though they are evaluated sequentially
- Most synthesis tools accept only one **`wait`** statement per process, which is usually required at the beginning or end of a process
- Signals have implicit memory: they retain their previous value, and are not updated until some later time or event
- Corollary: when multiple assignments are made to a signal within a process, only the last assignment takes effect

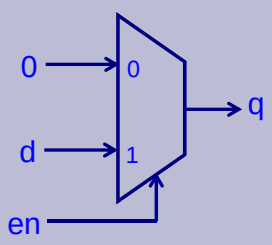
Latch implementation

- The implicit memory property of signals in VHDL is used to infer latches. But there are times when an implicit latch may be generated "accidentally". Most often, the resulting logic will be functionally correct; however, it will be slow and result in a much larger circuit than necessary

```
process (en, d)
  begin
  if en='1'
    then q <= d;
  end if;
end process;
```




```
process (en, d)
  begin
  if en='1'
    then q <= d;
    else q <= '0';
  end if;
end process;
```



- For fastest, most efficient designs, we must do the following:
  - completely specify all conditions within a process: if the **`else`** condition is not specified, then the output signals within the **`if…then`** will retain their last value due to implicit memory
  - completely specify all output possibilities within a process: the outputs must be specified completely for all selection options in a **`case`** statement. We can do this by setting the default value at the beginning of the process: we assert only the outputs necessary within each selection in the **`case`** statement (this takes advantage of the fact that signal values do not get updated until the end of a process)

Register implementation

- The preferred way of describing flip-flops is using the **`if…then`** statement along with a sensitivity list

```
process (clk)
begin
  if clk'event and clk='1'
      then if en='1'
             then q <= d;
           end if;
  end if;
end process;
```

- Asynchronous reset:

```
process (clk, reset)
begin
  if reset='1'
    then q <= '0';
    elsif clk'event and clk='1'
      then q <= d;
  end if;
end process;
```

- Synchronous reset:

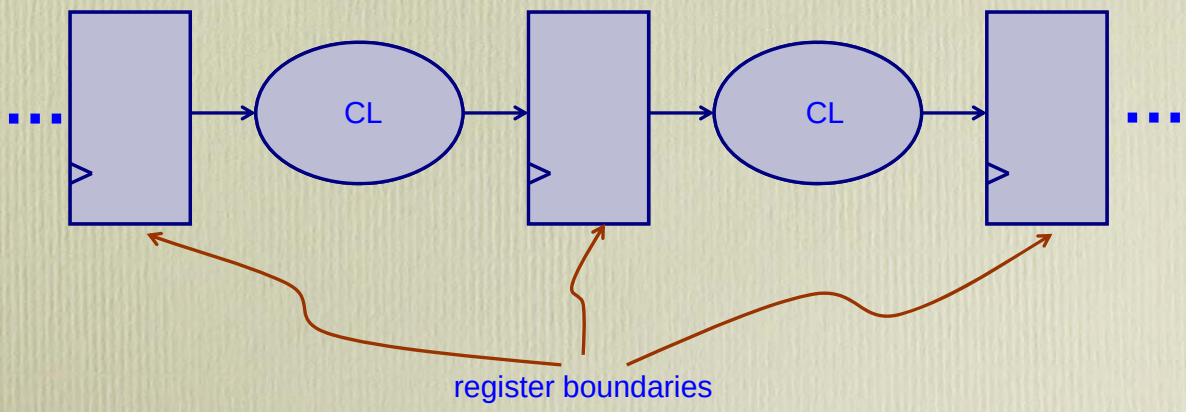
```
process (clk)
begin
  if clk'event and clk='1'
    then if reset='1'
           then q <= '0';
           else q <= d;
         end if;
  end if;
end process;
```


Counter implementation

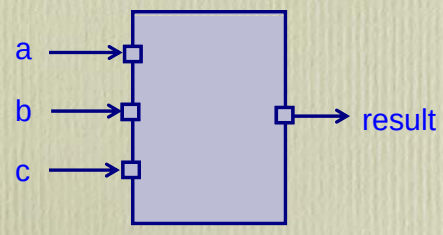
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port (clk : in std_logic;
        reset : in std_logic;
        count: out std_logic_vector(3 downto 0));
end counter;

architecture simple of counter is
  signal countL : std_logic_vector(3 downto 0);
begin
  process (clk)
  begin
  if (clk'event and clk='1')
    then if reset='1'
           then countL <= "0000";
           else countL <= countL + 1;
         end if;
  end if;
  end process;

  count <= countL;
end simple;
```

Design styles

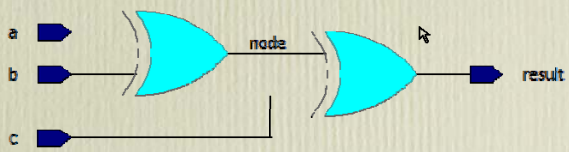
- The focus of VHDL is design for RTL (Register Transfer Level) synthesis: it means that logic is treated as if were combinational logic in between registers



- The combinational logic can be implemented in any of the VHDL design styles:
  - structural: a design composed of sub-blocks, analogous to writing a netlist of a schematic design
  - dataflow: Boolean equations
  - behavioral: algorithms
- Example:



entity



architecture

```
architecture dataflow of toto is
  signal node : std_logic;
  begin
  node <= a xor b;
  result <= node xor c;
  end dataflow;
```

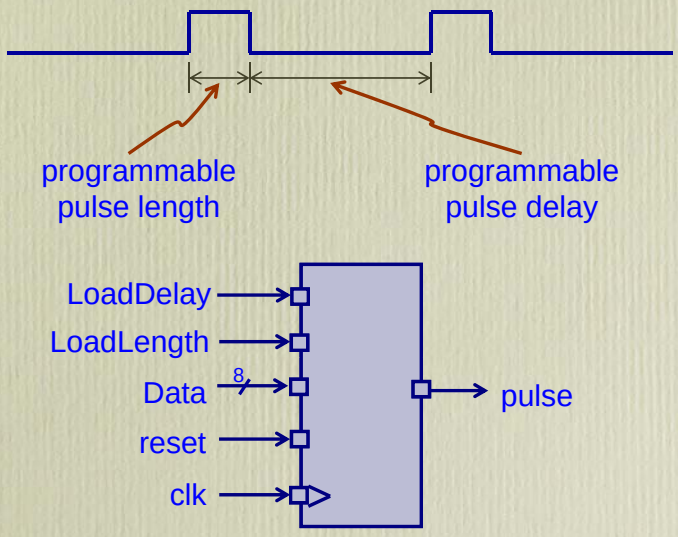
```
architecture behavior of toto is
  begin
  process (a, b, c)
    begin
    if ((a xor b xor c)='1')
      then result <= '1';
      else result <= '0';
    end if;
    end process;
  end behavior;
```



```
architecture structural of toto is
  signal u1_out : std_logic;
  begin
  u1: xor2 port map (i1 => a,
                     i2 => b,
                       y => u1_out);
  u2: xor2 port map (i1 => u1_out ,
                     i2 => c,
                     y => result);
  end structural;
```

Exercise

- To design a programmable pulse generator. The delay between pulses, and the length of duration of each pulse is programmable



Design partitioning

- Partitioning is the process of organizing and sub-dividing a design into smaller units for the purpose of:
  - managing design complexity
  - enhancing reliability via design reuse
  - making it easier to maintain design
  - incrementally synthesizing the design as it is developed
- A natural partition is the one that tends to separate datapath from control logic
- Hierarchy is a method of modularizing a design at different levels. Each successive level represents modules of lesser complexity than the one above it until, finally, the design is completely specified in terms of relatively simple blocks
- VHDL has the native capability to handle hierarchy

- We need to place (or *instantiate*) one or more copies of a black box (an entity-architecture pair): a *component* must be declared, which can be instantiated
- Components only describe the ports for a given black box. The logical behavior of the component is based on a corresponding entity-architecture pair
- The component name must match the corresponding entity name
- The signal names in the component should be the same as the signal name in the entity
- Each component instance has a label and has its port association mapped to the appropriate signals in the design. Each signal in the port list (the *formal* name) of the component is connected to a signal within the architecture (the *actual* name). This association can be explicit (named) or positional
- Each component instantiation is done concurrently within the architecture and can never be placed within a process

- Example:


```
u4: OR2 port map (i1 => toto,
                  i2 => titi,
                  y => y);

u4: OR2 port map (toto,
                  titi,
                  y);
```

- Output ports may be left unconnected: it is indicated with a port assignment of **open**

Libraries and packages

- There are two libraries that are implicitly visible, and do not need the **`library`** clause for visibility:
  - **`std`**: it contains the various types as well as operators that are part of the VHDL standard
  - **`work`**: it is the default working library
- It is often useful to keep logical building blocks of a design in separate files. To do so, however, these lower design units, or entity-architecture pairs, must be analyzed (into the **`work`** library) before the top level of the design is analyzed
- A *package* is a collection of declarations, which can be accessed by the **`use`** clause. In its simplest form, a package may be used to declare components, which can then be made visible not only for the current design, but by any design that references the package


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

package toto is
      •
      •    } component declarations
      •
end toto;
```

- We must first add the package to the work library before it can be used. For most synthesis tools, this implies that the compilation of the file containing the package must occur prior to that of the top level design that uses the package
- **`use work.toto.all`**
  A library declaration and **`use`** clause are only valid for one design unit (entity-architecture pair) that follows the clause
- In addition, types and sub-types may also be placed into packages

Component configuration

- In VHDL, a component declaration may apply to several entities, which in turn may be expressed using several architectures. The process of associating a specific entity and architecture to a component is known as *configuration*
- There are two kinds of configuration:
  - configuration declarations bind each component to an entity-architecture pair in a design
  - configuration specifications are written within an architecture, and only bind components for that architecture


```
configuration toto of entityname is
  for architecturename
    for u1: and2
              use entity work.and2(rt1);
    end for;
     •
     •
     •
end toto;
```

component

instance

entity-architecture pair

- If we chose to bind all instances of a given component to a given entity-architecture pair, we can use the **`for all`** statement in the configuration declaration

Scalable and parameterizable design

- Unconstrained arrays: VHDL allows port signals to be declared unconstrained
- Exemple:

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

entity sdffe is
  port (d   : in std_logic_vector;
        en  : in std_logic;
        clk : in std_logic;
        q   : out std_logic_vector);
end sdffe;
```

```
architecture synt of sdffe;
begin
  process
  begin
  wait until clk='1';
  if en='1'
    then q <= d;
  end if;
  end process;
end synt;
```


- To instantiate this entity-architecture pair, we must create a corresponding component declaration:

```
component sdffe
  port (d   : in std_logic_vector;
        en  : in std_logic;
        clk : in std_logic;
        q   : out std_logic_vector);
end component;
```

After placing this component declaration into the primitive package, we can then instantiate our scalable flip-flop into a design: by changing only the size of the input and output vectors the design is automatically scaled

- Generics: generics are constants that are passed into the entity declaration. They are not only used to pass information to an entity, but are useful when sizing an array in the port declaration, and even within the architecture
- Exemple:

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

entity pdffe is
  generic (n   : integer := 2);
  port    (d   : in std_logic_vector(n-1 downto 0);
           en  : in std_logic;
           clk : in std_logic;
           rst : in std_logic;
           q   : out std_logic_vector(n-1 downto 0));
end pdffe;
```



```
architecture synt of pdffe;
begin
  process (rst, clk)
  begin
  if rst='1'
    then q <= (others => '0');
    elsif (clk'event and clk='1')
      then if en='1'
             then q <= d;
           end if;
  end if;
  end process;
end synt;
```

- The default size indicated in the generic declaration is not only good design practice, but is also required by most synthesis tools

- Loops
- Exemple:

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

entity oddParityLoop is
  generic (width     : integer := 8);
  port    (ad        : in std_logic_vector(width-1 downto 0);
           oddParity : out std_logic);
end oddParityLoop ;
```



```
architecture synt of oddParityLoop;
begin
  process (ad)
    variable loopXor : std_logic;
  begin
  loopXor := '0';
  for i in 0 to width-1 loop
    loopXor := loopXor xor ad(i);
  end loop;
  oddParity <= loopXor;
  end process;
end synt;
```

- Generate: the **`generate`** statement in VHDL generates logic by repeating a slice of logic. Unlike the **`for…loop`**, however, the **`generate`** statement encapsulates concurrent statements
- Exemple:

```
toto: for i in 1 to n generate
        .
        .
      end generate;
```

- It is possible to create logic conditionally. The condition must be static, such as a generic or a constant, at the time of the synthesis:

```
toto: if condition generate
        .
        .
      end generate;
```