EE-334
Digital System Design

Custom Digital Circuits

Intro to VHDL

Andreas Burg

Background on VHDL

- VHDL is a **language for describing** the behaviour of **digital circuits**

  V ery High Speed Digital Integrated Circuits
  H ardware
  D escription
  L anguage

- **VHDL is used extensively by industry and academia** (especially in Europe)

- **VHDL** is **supported by all** relevant commercial and free **EDA tools**

Some VHDL History

- VHDL was developed by the VHSIC (Very High Speed Integrated Circuits) Program in the late 1970s and early 1980s
  - Development triggered by inadequate tools/methods to describe complex circuits

- VHDL has evolved over the years, with few major revisions
  - 1981: first proposal of the language
  - **1986/87:** proposed and **accepted as IEEE standard** (IEEE-1076-1987)
  - **1993:** **first revision** (IEEE-1076-1993)
  - 2002: second revision (IEEE-1076-2002)
  - **2008:** **third revision** (IEEE-1076-2002)

- **VHDL** language **is extended by** several **"packages"** that are **described in their own individual standards** (e.g., IEEE 1164-1993)

VHDL is NOT a Programming Language

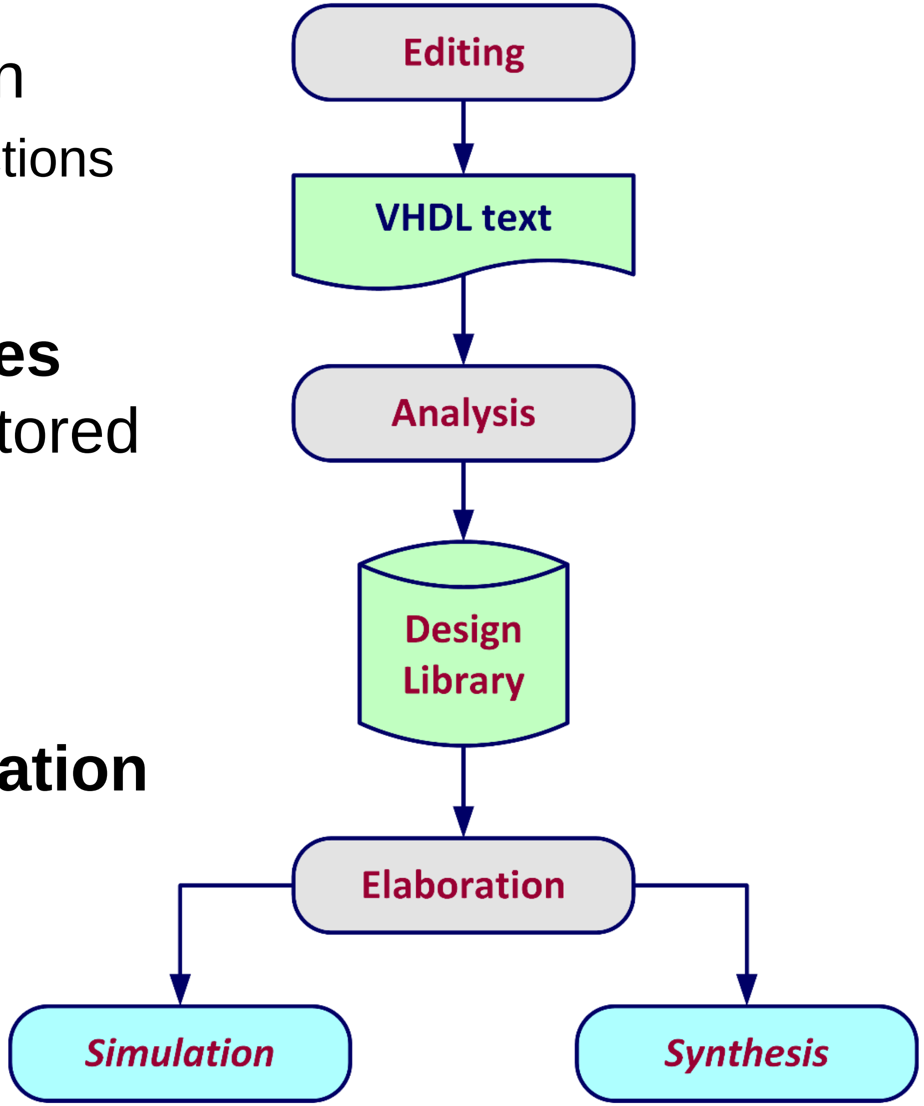
- **VHDL is different from a** classical software **programming language**
  - It describes the behaviour of hardware and is NOT a sequence of instructions

- **VHDL supports** many **concepts** that are **essential to HW design:**
  - parallelism,
  - hierarchy,
  - statements to model time/delay,
  - structural and behavioural modelling,
  - libraries and design reuse, …

- **VHDL** code **can always be simulated**, **but**
  only a subset of the language can be translated to hardware
- **Some code may translate to hardware, but may not function or be inefficient**

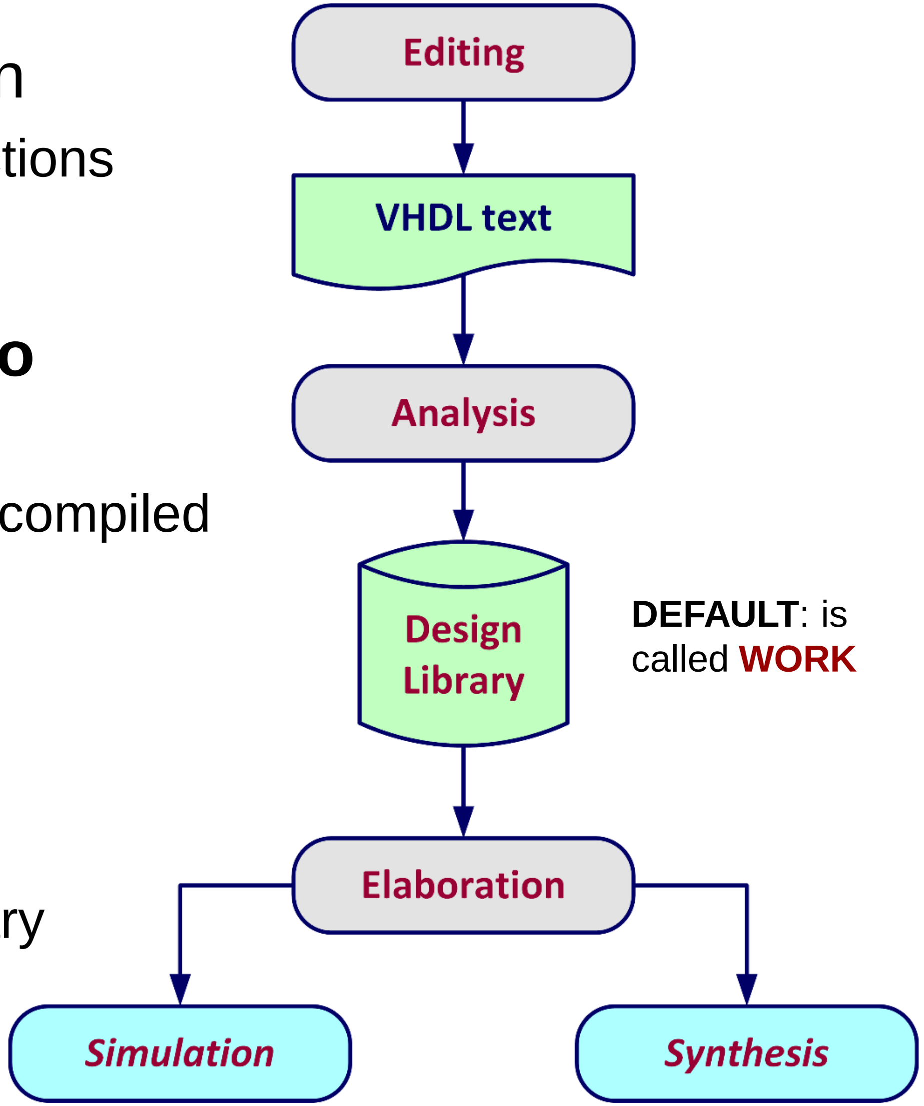
Preamble: The VHDL Library System

- **VHDL is built around design libraries** that contain
  - **Packages**: collections of constants, types, and functions
  - **Components**: hardware blocks
- An **analysis step checks** the **syntax** and **translates VHDL code into a binary representation** that is stored in the specified design library
  - **Default library** is typically called **“work”**
- An **elaborate step expands the binary representation** and **prepares it for simulation or synthesis**
  - This step is often hidden in the simulator
  - It is more complex for synthesis



Reminder: The VHDL Library System

- **VHDL is built around design libraries** that contain
  - **Packages**: collections of constants, types, and functions
  - **Components**: hardware blocks
- **Packages and components can be compiled into different libraries**
  - HOWEVER: In most cases, user-defined packages will be compiled into the same library as your design
  - Additional libraries are typically language extensions or technology libraries
- The **default library is** the usually called **`WORK`**
  - This library contains your design and is used when no library is explicitly specified when compiling



VHDL Packages

- **VHDL packages are similar to INCLUDE files in C/C#**
  - They include often used declarations to avoid the need to repeat them in many files

- **VHDL packages can include** for example
  - **CONSTANTS** that are used throughout a design
  - **COMPONENT** declarations
  - **DATA TYPE** definitions
  - Common **FUNCTIONS** and **PROCEDURES**

- **HINT:** Packages are convenient, to avoid the need to change design-wide parameters in one common place to avoid inconsistencies and simplify changes

Using Libraries and Packages

- **VHDL components and packages** can **“use”** the **content of other packages**

- To **use a package**, we need to **specify**
  - **The library** from which it is included
  - **The** name of the **package**
  - **The function/constant** to be used (or `all`)

```
LIBRARY library name;
USE library name.package_name.function_name¦ALL;
```

- Project specific packages are usually compiled into the default `WORK` library
  - The `WORK` library **is** always **already declared by default** and **can be referred to without explicitly declaring it**
  - **To include a package from the WORK library use:**
    - Note: packages included in that package are not visible in the design

```
USE WORK.my_package.ALL;
```

Using Libraries and Packages

- **Example**: packages from `ieee` library
  - for multi-valued logic (`std_logic_1164`) and
  - signed/unsigned arithmetic (`numeric_std`)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;
```

HINT: this will be the first three lines of almost any VHDL code you write

- **NOTE**: **components are "included"** in a different way than packages, **through instantiation** (see later)

VHDL Package Declaration

- **Packages are described in .vhd/.vhdl files**, same as design entities
  - HINT: always use a separate file for each package
  - The file name is not relevant, but should match the package name specified in the file

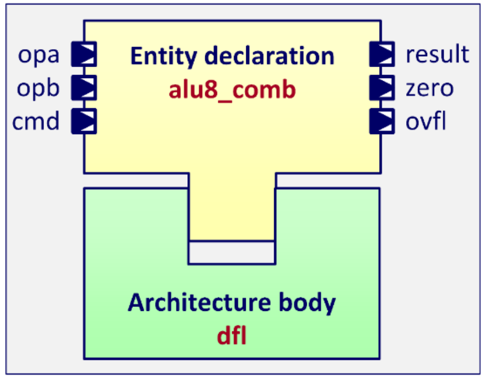
- The package name is the name under which the package is accessible from the library

- **Packages can themselves include other packages** from any library
  - Typical example: packages from the IEEE library
  - NOTE: a package included by a package is only visible in that package, not in the level above (visibility of package inclusions is limited to same package or design file)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

PACKAGE package_name IS

  -- PACKAGE CONTENT
  -- DECLARATION of CONSTANTS, TYPES, FUNCTIONS, …

END PACKAGE package_name;
```

Components: ENTITY and ARCHITECTURE

- Multiple **alternative architectures** can be provided **for the same entity**
  - Used for abstraction and refinement: e.g., behavioural, RTL, and gate-level netlist




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

entity alu8_comb is
   port (
      ...
   );
end entity alu8_comb;

architecture rtl of alu8_comb is
   ...
begin
   ...
end architecture rtl;
```

ENTITY
external interface of a component

ARCHITECTURE1 | **ARCHITECTURE2** | **ARCHITECTURE3**

implementation of the component

- **CONFIGURATIONS define which architecture is** used if multiple exist

ENTITY Declaration Syntax

- VHDL **entity defines the interfaces** of a VHDL component

- **Entity declaration** defines
  - the **name** of the component
  - **ports** : interface (inputs and outputs) of the component
  - **generics** : instance specific parameters
    - **NOTE: generics must resolve to a constant (be known) at compile-time**

```
ENTITY entity_name IS
          GENERIC(
                    generic_1_name : generic_1_type;
                    generic_2_name : generic_2_type
          );
          PORT(
                    port_1_name : port_1_dir port_1_type;
                    port_2_name : port_2_dir port_2_type
          );
END entity_name;
```

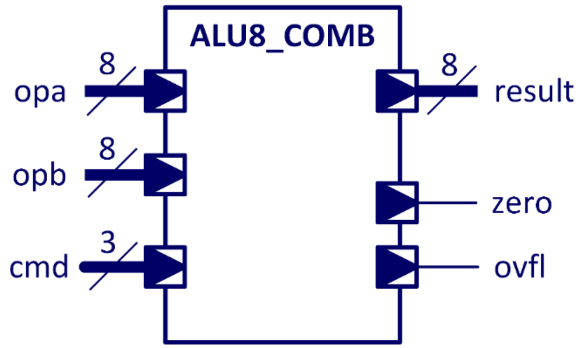
Port directions can be

`IN`	Input only
`OUT`	Output only
`INOUT`	Input/output for tristate
`BUFFER`	Output that can also be read

- **`PORT`** and **`GENERIC`** sections **are optional**

ENTITY Declaration Example

- **Example:** 8-bit ALU, purely combinational (no clock signal), no parameters




```
ENTITY alu8_comb IS
         PORT (
                  opa, opb : IN  std_logic_vector(7 downto 0);
                  cmd      : IN  std_logic_vector(2 downto 0);
                  result   : OUT std_logic_vector(7 downto 0);
                  zero     : OUT std_logic;
                  ovfl     : OUT std_logic
         );
END alu8_comb;
```

Architecture Declaration

- VHDL **architecture specifies the** model or **implementation** of a component

- **Architecture** declaration defines
  - the **name of the architecture**
  - the **name of** the **associated entity**
  - the **architecture body** that includes signal declarations and the actual code

```
ARCHITECTURE architecture_name OF entity_name IS
         -- signals to be used are declared here
BEGIN
         -- Insert VHDL statements to assign outputs to
         -- each of the output signals defined in the
         -- entity declaration.
END architecture_name;
```

- **Example:** `and_gate`

```
ENTITY and_gate IS
PORT (    a: IN  std_logic;
          b: IN  std_logic;
          c: OUT std_logic);
END and_gate;
-- Architecture
ARCHITECTURE comb_logic OF and_gate IS
BEGIN
          c <= a AND b;
END comb_logic;
```

Signals

- **Signals represent (are) wires**

- Signals are defined only inside an architecture
- The **ports** of an entity **allow signals to be connected from outside** a component
  - The ports of an entity can be treated as signals inside its architecture

- **Signals** are **declared in** the **preamble of the architecture**
- **Signals** are **associated with data types** which abstract electrical behaviour

```
ARCHITECTURE architecture_name OF entity_name IS
         -- signals to be used are declared here
         SIGNAL signal_1_name : signal_1_type;
         SIGNAL signal_2_name : signal_2_type;
BEGIN
         -- VHDL statements
END architecture_name;
```

Constants

- Constants serve two purposes (but as the same object)
  - Definition of **fixed values** that are **known at compile-time** (i.e., before e.g., hardware is constructed during elaboration)
    - **Examples**: width of signals, number of instances of a component, iterations of a generate-loop
  - Definition of fixed electrical signals that can be thought of as connections to VDD or GND

- Constants can be declared in
  - Packages
  - As GENERICS in an ENTITY
  - As CONSTANTS in the preamble of the architecture
- **Constants can be derived** with valid expressions **from other constants**

```
ARCHITECTURE architecture_name OF entity_name IS
        -- constants to be used are declared here
        CONSTANT constant_1_name : constant_1_type := expression;
        -- EXAMPLE
        CONSTANT WIDTH_A : integer := 8-1;
        CONSTANT WIDTH_B : integer := WIDTH_A + 1;
BEGIN
        -- VHDL statements
END architecture_name;
```

Built-In Data Types

- VHDL knows **several** standard **built-in data types**

- **VHDL** supports 6 **native scalar** (singe value) **data types**
  - `Bit       : 1/0`
  - `Boolean   : true/fals`
  - `Integer   : defined by a range (default is 32-bit)`
  - `Char      : 8-bit`
  - `Real      : floating point`
  - `Time      : for modeling of delays [ps,ns]`

- Not all types are equally well suited for synthesis
  - Time has no hardware equivalent
  - Real leads to excessive complexity and is often not understood by tools

Extended Multi-Value Data Types: `std_logic`

- **Binary 0/1** representation **of the basic bit type is** often **insufficient** to
  - describe complex electrical states of wires (driver conflicts)
  - model design intent that provides flexibility for optimization
- IEEE standardized of **more capable data types** in `ieee.std_logic_1164` with up to **nine possible values (encoded as characters in '…')**

`std_logic`

Logic-0	`'0'`	Weak-0	`'L'`
Logic-1	`'1'`	Weak-1	`'H'`
Don't Care	`'-'`	Weak-X	`'W'`
High Impedance	`'Z'`	Uninitialized	`'U'`
Unknown	`'X'`		

- **Values for `std_logic`** are **encoded as CHARACTERS: '…'**

Extended Multi-Value Data Types: `std_logic`

- **For synthesis:**
  - **Logic-0 and Logic-1**: straightforward binary value
  - **Don't care '-'**: heavily used for synthesis to specify no preference for a particular value.
  - **High impedance 'Z':** used to define tri-state drivers (**ONLY USE IN EXCEPTIONAL CASES and with greatest care**)

- **For simulation:**
  - **Unknown 'X':** useful in simulations to identify driver conflicts or unknown logic levels (e.g., during signal rise time)
  - **Uninitialized 'Z':** indicates that a signal was never assigned a value
  - **Weak-0, Weak-1, Weak-X**: mostly unused and appear only in simulation to resolve conflicts

- Multi-value data types can resolve driver conflicts
  - This feature should not be used for design intent during synthesis

Concurrent Signal Assignments

- **Concurrent assignments** are **part of the architecture body**
- To **assign the value of a r.h.s. expression to a signal on the l.h.s.** use

```
signal <= expression;
```

  - Interpret as driving the signal with the output of the circuit that evaluates the expression

- **Concurrent assignments** are always **carried out in parallel**
- A **signal should never have more than one driver**
  - Only assigned in one single concurrent statement
  - Exceptions: tri-state signals (ideally do not use)

```
ARCHITECTURE DONT_DO_THIS OF BAD_DESIGN IS
         -- signal declaration
         SIGNAL A : std_logic;
BEGIN
         -- DRIVER CONFLICT
         A <= '0';
         A <= '1';      => 'X'
END DONT_DO_THIS;
```

Boolean Expressions and Constant Assignments

- **Valid values can be assigned directly to a signal with a compatible data type**

```
Signal_1 <= <valid_value>;
```

- **VHDL supports all common Boolean operators** between signals

<boolean_operator>		
AND	NAND	NOT
OR	NOR	
XOR	XNOR	

```
Signal_3 <= Signal_1 <boolean_operator> Signal_2;
```

General Remarks

- **VHDL** is **not case sensitive**. **Nevertheless, be consistent** in your names

- Many naming styles are OK, here is one recommendation
  - Write **VHDL keywords in ALL UPPER CASE**
  - Write **signals in CamelCase**

- **Comments** are **initiated by** a leading **--** until the end of the line

```
-- This is a VHDL comment
```

VHDL Naming Conventions

- **Motivation**: signal name describes clearly the origin and nature of the signal
  - VHDL code is often long. Hints where a signal comes from are incredibly useful

<Name>x<Signal Class>[<State>][<Low Active>][<PortDirection>]

Class	Label	Example
Clock	C	CLKxCI
Async Reset	R	RSTxRBI
Control/status	S	ClearCNTxS
Data/address	D	SamplexDN
Test signals	T	ScanENxT

Active	Label	Example
LOW	B	ENxSB
HIGH		ENxS

State	Label	Example
Present	P	REGxDP
Next	N	REGxDN

Direction	Label	Example
IN	I	AxDI
OUT	O	ZxDO