Hardware Modeling [VU] (191.011)
– WS24 –
VHDL Basics

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

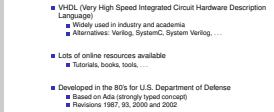
Welcome to this first lecture on VHDL, a critical language in the world of hardware design. By the end of this lecture, you will understand VHDL's basic syntax and be able to write simple programs with text output.

Hardware Modeling [VU] (191.011)
– WS24 –
VHDL Basics

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-30, 16:53 (e9b236f)



VHDL stands for Very High Speed Integrated Circuit Hardware Description Language. It is used to model electronic systems. VHDL is especially popular in industries where reliability and accuracy are crucial, such as aerospace, automotive, and telecommunications. You can find a lot of additional learning and discussion material online. We will also provide helpful resources and links on our TUWEL page. Let's start with some background information. VHDL was developed in the 80s by the U.S. Department of Defense and is based on Ada, which is known for its safety features and strong typing. The U.S. Department of Defense needed a way to specify hardware behavior, which natural language could not effectively capture.

Introduction

- VHDL (Very High Speed Integrated Circuit Hardware Description Language)
  - Widely used in industry and academia
  - Alternatives: Verilog, SystemC, System Verilog, . . .
- Lots of online resources available
  - Tutorials, books, tools, . . .
- Developed in the 80's for U.S. Department of Defense
  - Based on Ada (strongly typed concept)
  - Revisions 1987, 93, 2000 and 2002

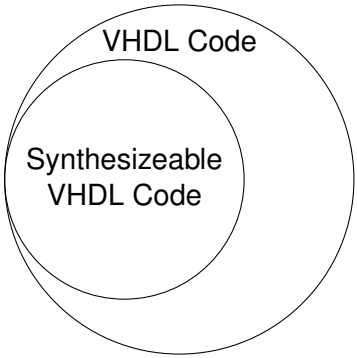
- Initially solely used to document hardware
  - Later extended by synthesis tools
  - Only subset of commands can be transferred to hardware (= synthesizable VHDL)
  - All VHDL code is simulatable
- VHDL Standard 2008 taught in this lecture
  - Attention: not all 2008 features are supported by EDA tools
  - Has to be explicitly selected in tools (not the default)

VHDL Code
Synthesizeable VHDL Code

Initially, VHDL was used primarily for documenting hardware. The goal was to create a secure and robust way to describe hardware and catch errors early in the design process. VHDL allowed for a precise description of hardware components that manufacturers could use to build the desired equipment. Later, synthesis tools were developed to convert VHDL descriptions into actual hardware. This caused some VHDL features, such as specifying delays, to be not directly translatable into hardware and considered non-synthesizable. VHDL includes both synthesizable and non-synthesizable features. However, non-synthesizable VHDL code CAN be simulated. For instance a delay statement may not be synthesizable but in a simulation it can mimic real world signal delays. Anyway, this will be covered in more detail in future lectures. In this lecture, we'll cover basic VHDL syntax, so you can start writing simple software programs with terminal outputs with the help of a simulator. We'll focus on VHDL-2008, as it is the most widely supported standard with broad tool compatibility. Keep in mind that many tools default to older VHDL standards, so you might need to configure them to use VHDL-2008. Next, let's dive into the VHDL syntax.

Introduction (cont'd)

- Initially solely used to document hardware
  - Later extended by synthesis tools
  - Only subset of commands can be transferred to hardware (= synthesizable VHDL)
  - All VHDL code is simulatable
- VHDL Standard 2008 taught in this lecture
  - Attention: not all 2008 features are supported by EDA tools
  - Has to be explicitly selected in tools (not the default)



Let's start by looking at some fundamental properties of VHDL as a language. First, it's important to know that VHDL is a case-INsensitive language. This means that whether you type your identifiers and keywords in lowercase or uppercase, VHDL will treat them as the same.

Language Properties

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`

Case insensitive
variable = VARIABLE = VaRiAbLe
Commands terminated by ';'

Another key aspect is that commands in VHDL are terminated by a semicolon, just like in Java or C.

Language Properties

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`
- Commands terminated by ';'

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`
- Commands terminated by ';'
- Comments
  - '--' single line comment
  - Since 2008 multi-line comments possible ('/*' and '*/')

Single-line comments are marked with two dashes. For multi-line comments, which were introduced in the 2008 revision, you can use the Java or C style multi line comment delimiters.

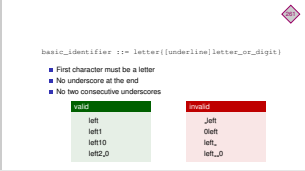
Language Properties

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`
- Commands terminated by ';'
- Comments
  - '- -' single line comment
  - Since 2008 multi-line comments possible ('/*' and '*/')

Throughout this course, you'll notice that we use specific formatting for keywords, data types, comments, and constants to make the code more readable. This will help you quickly identify different elements when you're reading VHDL code in our slides.

Language Properties

- Case insensitive
  `variable` = `VARIABLE` = `VaRiAbLe`
- Commands terminated by ';'
- Comments
  - '- -' single line comment
  - Since 2008 multi-line comments possible ('/*' and '*/')
- Format used in lecture
  - `keywords`
  - `datatype`
  - `comment`
  - `CONSTANT`
  - `everything else`



Next, let's discuss identifiers in VHDL, which are names used for variables and various other elements in your code. This slide shows a BNF, or Backus-Naur Form, which is a notation used to describe the syntax of context-free grammars. We will frequently use this notation throughout the course to present the VHDL syntax. BNF is also used extensively in third-party resources, like the VHDL standard, to explain syntax rules. If you are not already familiar with BNF, I encourage you to get comfortable with it. In BNF, curly brackets indicate repetition, square brackets indicate optional elements, and vertical bars indicate choices between alternatives. If a term appears without any brackets, it is mandatory to use that term to form a valid syntax. As an example take the BNF for basic identifiers here. Identifiers must start with a letter, and there are certain restrictions you need to be aware of. For example, you cannot have underscores at the end of an identifier or two consecutive underscores within an identifier. Valid examples include left, left1, and left underscore 0. Invalid identifiers might include names like underscore left, 0 left, or any identifier that ends with an underscore.

Basic Identifiers


```
basic_identifier ::= letter{[underline]letter_or_digit}
```

- First character must be a letter
- No underscore at the end
- No two consecutive underscores

valid	invalid
left	_left
left1	0left
left10	left_
left2_0	left__0

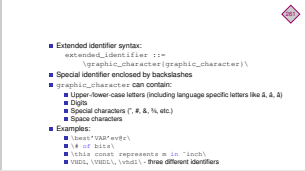
In VHDL, we have a concept called extended identifiers. These are identifiers that are enclosed within backslashes. They allow us to use characters that aren't typically allowed in standard identifiers, like special characters, spaces, or even language-specific letters. You can check out the VHDL standard for a comprehensive naming method scheme.

Extended Identifiers

- Extended identifier syntax:

```
extended_identifier ::=
    \graphic_character{graphic_character}\
```

- Special identifier enclosed by backslashes



This flexibility can be particularly useful in cases where you want your identifiers to be more descriptive or include specific characters, like special symbols, punctuation, or spaces. You can use extended identifiers to name something very precisely. For example, you can use a sentence as variable name. Extended identifiers can make your code more readable by allowing you to write names that are closer to natural language, or to conform to specific naming conventions that include special characters or symbols. You can use typical characters meant for common abbreviations like hashtags for 'number'. As in number of bits for instance. Just remember that these identifiers are enclosed in backslashes, and should be used carefully to maintain code readability and clarity. Note that, you escape the usual case-insensitivity with this feature and thus easily create three different identifiers with basically the same name as shown with the VHDL example at the bottom of the slide. Overall, extended identifiers provide additional flexibility for naming conventions. However, this kind of identifiers is rarely used in practice. Next, let's go over some important core VHDL constructs.

Extended Identifiers

- Extended identifier syntax:

```
extended_identifier ::=
    \graphic_character{graphic_character}\
```

- Special identifier enclosed by backslashes
- `graphic_character` can contain:
  - Upper-/lower-case letters (including language specific letters like ä, å, â)
  - Digits
  - Special characters (', #, &, ¾, etc.)
  - Space characters
- Examples:
  - `\best'VAR'ev@r\`
  - `\# of bits\`
  - `\this const represents m in ~inch\`
  - `VHDL`, `\VHDL\`, `\vhdl\` - three different identifiers

We start by covering entities. In VHDL, the concept of an entity is central to understanding how you describe hardware. An entity defines the external interface of a module. It tells us what the module is called and what its inputs and outputs are. Think of the entity as a blackbox that hides all the internal workings, showing only the essential connection points to the outside world. This abstraction is powerful because it allows you to focus on how components connect together without worrying about their internal details at this stage. Think of entities like a Java interface or an abstract Java object for which you define public input and output variables. In the example provided, you can see a simple entity that doesn't have any input or output. While this might seem trivial now, it forms the basis of all designs, no matter how complex they become.

Entity

- Start by characterizing a hardware `entity`
  - Defines a module's interface
  - Specifies name, inputs, and outputs
  - Encapsulates internal details
  - Blackbox definition: no knowledge of inner workings is needed
- Example `entity` with no I/O:

```
1 entity ENTITY_NAME is
2   -- I/O definitions
3 end entity;
```

Now that we've covered entities, let's move on to architectures. This is where the real work happens. Architecture describes the internal structure and behavior of an entity. In other words, it tells us how the entity's inputs are processed to produce the outputs. You can have multiple architectures for a single entity, each implementing different behaviors. Like multiple distinct Java objects for one interface. This will become more relevant later, when we start to cover hardware generation. For example, one architecture might be optimized for speed, while another is optimized for area or power consumption. This flexibility allows you to experiment with different design strategies without changing the entity's interface. In practice, architecture types are often named behaviorally, like 'behavioral' - or an abbreviation of that word - for a behavioral description, which describes how the entity works in a functional sense. The example shown here demonstrates an empty architecture with a generic name corresponding to a generic entity. Architectures start with the `architecture` keyword followed by the architecture name. Then you follow up naturally and specify "of" which entity this architecture is meant to be. Next is the declaration scope for constants and more. Elements in this scope are comparable to private class variables. After the `begin` keyword you would write the actual behavior of the specified entity. As already mentioned, in this introduction session we will write VHDL software programs rather than actual describing hardware. The goal is to make you familiar with basic VHDL constructs and syntax. We need a way to write sequential code in an architecture like we do in functions in Java or C. Thus, let's dive into a VHDL construct that supports exactly this.

Architecture

- Describes the internal structure via `architecture`
- Defines how the entity's functionality is implemented
  - Knowledge of inner workings is needed
  - How does the entity behave?
- It is possible to define multiple architectures for a single entity
  - Typical names for architectures: `beh`/`behavioral` or `struct`/`structural`

- Example: empty `architecture`

```
1 architecture ARCH_NAME of ENTITY_NAME is
2   -- constants, etc.
3 begin
4   -- description of inner workings of ENTITY_NAME
5 end architecture;
```

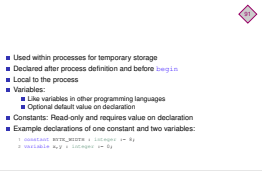
Processes are an essential part of VHDL. They allow you to describe sequential operations within an architecture. While VHDL is primarily a parallel language, processes introduce the concept of time and ordering, allowing you to describe operations that depend on previous events. For now, we will focus only on this sequential operation paradigm. Within an architecture a process starts with the `process` keyword. After that follow declarations of constants and variables. We will show an example of that later. The `begin` keyword initiates the process body. Here are all the sequential statements of a process. Note that the last statement in the process is `wait`. This is important now such that you can write a program that can be executed once per run. As is usual with VHDL scopes the process ends with the `end` keyword and the kind of scope - in this case `end process`.

Process

- Defines sequential execution within an architecture
- `wait` at the end signals process termination

- Example `process`:

```
1  architecture ARCH_NAME of ENTITY_NAME is
2    -- constants, etc.
3  begin
4    process
5      -- constants, variables
6    begin
7      -- sequential statements
8      wait;
9    end process;
10 end architecture;
```

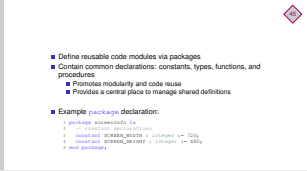


Within processes, you often need to store intermediate values, and this is where variables and constants come into play. Variables in VHDL are local to the process they're defined in. Like variables in other programming languages they are updated immediately when you assign them a new value. Constants, on the other hand, are read-only values that must be assigned when they're declared. They are useful for values that shouldn't change, like mathematical constants or configuration parameters. The example here shows declarations of a constant followed by two integer variables. Constant and variable declarations are initiated with the `constant` and `variable` keywords, respectively. Follow up with the name, colon character and type name after which you can assign default values after colon equal. Note that the assignment for constants is mandatory. For variables, they are optional but recommended. For simplicity, we will only handle some built-in types in this lecture.

Variables & Constants

- Used within processes for temporary storage
- Declared after process definition and before `begin`
- Local to the process
- Variables:
  - Like variables in other programming languages
  - Optional default value on declaration
- Constants: Read-only and requires value on declaration
- Example declarations of one constant and two variables:

```
1 constant BYTE_WIDTH : integer := 8;
2 variable x,y : integer := 0;
```



Packages in VHDL are similar to libraries or modules in other programming languages. They allow you to define reusable code components like constants and in later lectures you will also find types, functions, and procedures here. Packages help to promote modularity by keeping common definitions and utilities in a single, manageable place. This is particularly useful in larger projects where multiple entities or architectures need to share the same data types or functions. We will cover types, functions and procedures in a later lecture. To declare a package, you start with the `package` keyword followed by the package name and the `is` keyword. Within the package, you can define various reusable components. After that, you end the package declaration with `end package` optionally followed by the package name. By organizing your code into packages, you can build up a library of reusable components that makes future design work easier and more efficient.

Packages

- Define reusable code modules via packages
- Contain common declarations: constants, types, functions, and procedures
  - Promotes modularity and code reuse
  - Provides a central place to manage shared definitions

- Example `package` declaration:

```
1 package screenInfo is
2   -- constant declarations
3   constant SCREEN_WIDTH : integer := 720;
4   constant SCREEN_HEIGHT : integer := 480;
5 end package;
```

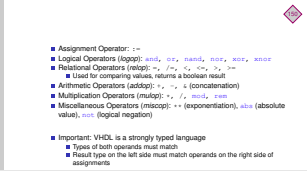
Once a package is defined, you can use its contents in your design by importing it with the 'use' clause. This provides access to all the components that the package contains. To use a package, you start with the `use` keyword, followed by the library name, package name, and the `all` keyword to include everything. Alternatively, you can specify individual elements if you don't need the entire package. In the example, the line 'use work.screenInfo.all;' is used to import everything defined in the package 'screenInfo' from the previous slide located in the 'work' library. The 'work' library is the default library in VHDL where all your design units, like entities, architectures, and packages, are compiled by default. By specifying 'work', you are referring to the current working library for your project. You will get to know other important libraries in later lectures. The `all` keyword lets you import all types, constants, functions, and procedures that the package 'screenInfo' defines.

Using Packages

- Import packages into your VHDL code with the `use` clause
- Provides access to types, constants, functions, and procedures defined in the package
- Use the following syntax:
  - `use` *library_name.package_name.*`all`**;**
  - `use` *library_name.package_name.element_name;*
  - Default library ***work***: current project working library

- Example of using a `package`:

```
1 -- make SCREEN_WIDTH, SCREEN_HEIGHT available to this file
2 use work.screenInfo.all;
3
4 -- entity declaration
5
6 -- entity architecture
```

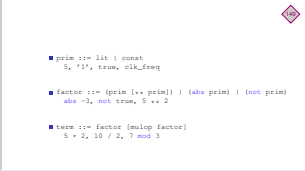


Let's look at some basic operators in VHDL that you'll often use in your designs. We have the assignment operator, represented by the colon equals sign, used for assigning values to variables. There are logical operators like 'and' and 'or', which handle boolean values. Relational operators let you compare values and return a boolean result. Note that the equality operators differ from Java and C. For equality, VHDL uses a single equals sign, while inequality is represented by a slash followed by an equals sign. Arithmetic operators like plus and minus handle numeric operations The ampersand operator is also called concatenation operator. For now, it is only relevant for building strings. There are also operations for multiplication and divisions as well as modulo and remainder. The difference between mod and rem is that rem ensures the remainder has the same sign as the dividend. Lastly, there are helpful operators for exponentiation as well as unary operators to retrieve absolute values from integers and logical negotiations. Remember, VHDL is strongly typed, meaning operand types must match exactly, or you'll get an error. We'll see examples of these operators later. Next, we look at expressions.

Basic Operators

- Assignment Operator: `:=`
- Logical Operators (*logop*): `and`, `or`, `nand`, `nor`, `xor`, `xnor`
- Relational Operators (*relop*): `=`, `/=`, `<`, `<=`, `>`, `>=`
  - Used for comparing values, returns a boolean result
- Arithmetic Operators (*addop*): `+`, `-`, `&` (concatenation)
- Multiplication Operators (*mulop*): `*`, `/`, `mod`, `rem`
- Miscellaneous Operators (*miscop*): `**` (exponentiation), `abs` (absolute value), `not` (logical negation)

- Important: VHDL is a strongly typed language
  - Types of both operands must match
  - Result type on the left side must match operands on the right side of assignments



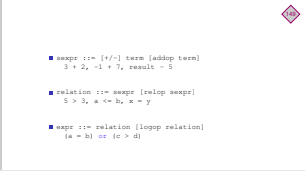
Let's start by breaking down the fundamental elements of VHDL expressions. The most basic building blocks are primitives, like literals or constants. For example, the number five, the character '1', or a predefined constant like true or a variable name. Factors are slightly more complex; they can include exponentiation, absolute values, or logical negation. A term combines factors using multiplication or division operators. Understanding these basic elements is crucial, as they form the foundation for more complex expressions.

Basic Expression Elements

- `prim ::= lit | const`
  `5, '1', true, clk_freq`

- `factor ::= (prim [** prim]) | (abs prim) | (not prim)`
  `abs -3, not true, 5 ** 2`

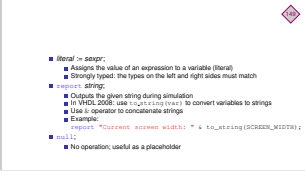
- `term ::= factor [mulop factor]`
  `5 * 2, 10 / 2, 7 mod 3`



Now let's look at how we build more advanced expressions using the basic elements from the previous slide. A simple expression, can be formed by combining terms with addition or subtraction. Relations allow us to compare two expressions, using relational operators like greater than or equal to. At the highest level, an expression can be a relation or a combination of multiple relations. By mastering these building blocks, you can construct most logical expressions you need in VHDL. Note that the operations and expression definitions are also available on the VHDL cheat sheet which you can find on this course's TUWEL page.

Combining Elements into Expressions

- `sexpr ::= [+/-] term [addop term]`
  `3 + 2, -1 + 7, result - 5`
- `relation ::= sexpr [relop sexpr]`
  `5 > 3, a <= b, x = y`
- `expr ::= relation [logop relation]`
  `(a = b) or (c > d)`



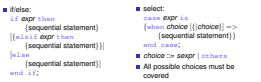
Let's review some basic sequential statements used in VHDL. Again, the assignment operator allows us to assign a value to a variable. Assignments require a simple expression on the right side. Remember, VHDL is strongly typed, so both sides of the assignment must have compatible types. The report statement is used to output messages during simulation, which can be helpful for debugging. You can convert variables to strings using the "to-string" function introduced in VHDL 2008 and concatenate multiple strings using the ampersand operator. Finally, the null statement represents a no-operation, useful when a statement is required syntactically, but no action is needed. Before writing our first program let's also talk about control flow operations.

Basic Sequential Statements

- *literal* := *sexpr*;
  - Assigns the value of an expression to a variable (literal)
  - Strongly typed: the types on the left and right sides must match
- `report` *string*;
  - Outputs the given string during simulation
  - In VHDL 2008: use `to_string(var)` to convert variables to strings
  - Use & operator to concatenate strings
  - Example:

```
report "Current screen width: " & to_string(SCREEN_WIDTH);
```

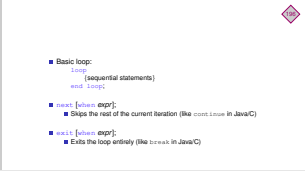
- `null`;
  - No operation; useful as a placeholder



Control flow is crucial in programming as it allows your code to make decisions based on certain conditions. Let's start with the if-else statement. The if-else statement in VHDL works similarly to other programming languages. It checks a condition represented by an expression, and executes a block of sequential statements if this condition is true. If the condition is false, the program can optionally check additional conditions using 'else-if' branches, or execute a default block of statements using the else branch. Note that "else-if" branches are introduced with somewhat uncommon `elsif` keyword, a quirk inherited form Ada. Next, we have the `case` statement, which is used when you want to select one of many possible actions based on the value of an expression. The case statement compares the expression against a series of choices. When a match is found, the corresponding block of sequential statements is executed. A default action can be specified, in case not all possible values the expression can evaluate to are covered. A choice statement consists of a simple expression or the `others` keyword. 'Others' is essentially like the 'else' keyword in if statements or 'default' in Java's and C's switch-case. The case statement is particularly useful when dealing with enumerated types or situations where multiple distinct conditions need to be handled differently. There is no fall through like in Java or C in VHDL so no 'break'-like keyword is required. A choice only executes the sequential statements below the choices. A valid case statement has to cover all possible choices from the expression. This is where the 'others' keyword as default branch comes in handy. The case statement will become very useful in later lectures. Especially when enumerated types are handled.

Control Flow: Branching

- if/else:
  `if` *expr* `then`
  {sequential statement}
  [{`elsif` *expr* `then`
  {sequential statement}}]
  [`else`
  {sequential statement}]
  `end if`;
- select:
  `case` *expr* `is`
  {`when` *choice* [{|*choice*}] =>
  {sequential statement}}
  `end case`;
- *choice* ::= *sexpr* | `others`
- All possible choices must be covered



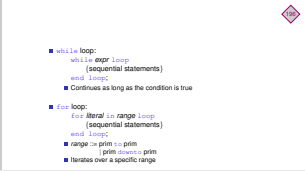
Let's begin by looking at basic loops in VHDL. A basic loop is created using the loop keyword, and it will continue indefinitely unless you use a control statement to break out of it. This is similar to an infinite loop in other programming languages. To control the flow within the loop, you can use the next statement, which skips the rest of the current iteration and moves to the next one, similar to the continue statement in Java or C. The `exit` statement, is used to exit the loop entirely, much like the "break" statement in other languages. Both `next` and `exit` can be used with an optional `when` clause to conditionally control the loop's behavior.

Control Flow: Basic Loops

- Basic loop:

```
loop
    {sequential statements}
end loop;
```

- `next` [`when` *expr*];
  - Skips the rest of the current iteration (like `continue` in Java/C)
- `exit` [`when` *expr*];
  - Exits the loop entirely (like `break` in Java/C)



Now, let's look at typed loops in VHDL. The while loop continues execution as long as the condition specified in the while expression is true. This is similar to while loops in other programming languages, where the loop checks the condition before each iteration. On the other hand, the for loop is used to iterate over a specific range of values, defined by either the `to` or `downto` keywords. With the for loop, you specify a loop parameter, which takes on each value in the range, one at a time. Note that the VHDL standard defines this loop parameter to be a constant inside the loop. Thus, when executing the sequential statements of the loop's body, the loop parameter cannot be modified. Both loops allow you to control repetitive actions in your VHDL code, but they are suited for different situations depending on whether you know the number of iterations in advance or need to rely on a condition.

Control Flow: Typed Loops

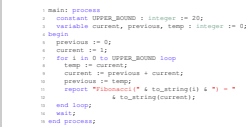
- `while` loop:
  `while` *expr* `loop`
  {sequential statements}
  `end loop;`
  - Continues as long as the condition is true

- `for` loop:
  `for` *literal* `in` *range* `loop`
  {sequential statements}
  `end loop;`
  - *range* ::= prim `to` prim
    | prim `downto` prim
  - Iterates over a specific range

Lastly, before showing a complete VHDL code example let's quickly talk about labels. In VHDL, every scope of code, such as an architecture or entity, can have an identifier. While not always required, identifiers help manage and clearly define the scope, making the code more readable and maintainable. Loops, if-statements, and process blocks can also have optional labels. Scopes are closed with the `end` keyword, and it's possible to include the identifier in the end statement as well. However, we discourage you from do this, as it introduces unnecessary work when something needs to be renamed and modern IDEs or code editors visualize this scopes anyway. Using labels can greatly improve the readability of your VHDL code, especially in complex or nested structures. It is possible to target specific nested loop hierarchies with the control flow statements 'next' and 'exit'. Overall, thoughtful use of identifiers and labels enhances code clarity and maintainability. Additionally, keep in mind that labeling structures can help you find and debug problems in simulations later.

Miscellaneous: Block Identifiers & Labels

- Every VHDL code scope can have an identifier
  - Required in certain scopes (e.g., `architecture`, `entity`)
  - Identifiers help to clearly define and manage the scope's purpose
- Scopes are closed with the `end` keyword
  - Identifier can be included in the `end` line, e.g., '`end entity myEntity;`'
  - We advise against this practice
- Loops, `if` statements, and `process` blocks can have optional labels
  - C-like syntax: e.g., '`LABELID: process`'
  - Labels can be used with `next` and `exit` to target specific loops, improving control flow within nested structures



Now lets finally put everything together and look at a bigger more comprehensive example. You are probably familiar with the Fibonacci sequence. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1. In this slide you can see the implementation of this sequence in a process. Let's start from the top. We call the process 'main' by labeling it. Then we define the constant `UPPER_BOUND` which limits the number of calculated Fibonacci numbers. Three variables are required in this implementation to calculate the sequence. We start the process by assigning 0 to 'previous' and 1 to 'current'. Then we start a loop declare a loop variable `i` and let the loop repeat from the range 0 to `UPPER_BOUND`. The loop body contains statements such that we calculate the next Fibonacci number by summing up the two previous ones. After which we print the current Fibonacci number via the 'report' command by concatenating a few constant string with two variables with the help of the `to_string` function. We end the loop, and finally, before we terminate the process we add the -for now- required 'wait' statement. We will now show you how you can easily test and play around with this code.

Example Process: Fibonacci


```
1  main: process
2    constant UPPER_BOUND : integer := 20;
3    variable current, previous, temp : integer := 0;
4  begin
5    previous := 0;
6    current := 1;
7    for i in 0 to UPPER_BOUND loop
8      temp := current;
9      current := previous + current;
10     previous := temp;
11     report "Fibonacci(" & to_string(i) & ") = "
12                    & to_string(current);
13   end loop;
14   wait;
15 end process;
```

- **Web-based platform** for simulating and sharing HDL code
  - https://www.edaplayground.com/
- Supports multiple languages:
  - VHDL, Verilog, SystemVerilog, and more
- Features:
  - Access to various simulators (e.g., GHDL, ModelSim (QuestaSim))
  - Collaborative coding with sharing links
  - No installation needed, runs in the browser
- Ideal for:
  - Learning and practicing HDL coding
  - Testing and debugging small code snippets
  - Demonstrating concepts in a classroom setting

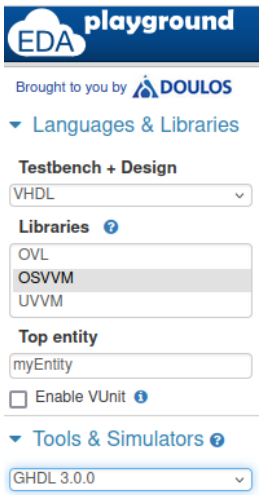
For this purpose we will use the EDA Playground. This is a Web-based platform for testing HDL code in general. It has a lot of features but for now just see it as a tool to execute VHDL code.

Testing: EDA Playground Overview

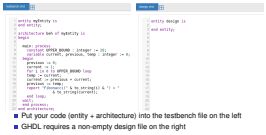
- **Web-based platform** for simulating and sharing HDL code
  - `https://www.edaplayground.com/`
- Supports multiple languages:
  - VHDL, Verilog, SystemVerilog, and more
- Features:
  - Access to various simulators (e.g., GHDL, ModelSim (QuestaSim))
  - Collaborative coding with sharing links
  - No installation needed, runs in the browser
- Ideal for:
  - Learning and practicing HDL coding
  - Testing and debugging small code snippets
  - Demonstrating concepts in a classroom setting

First, focus on the left side of the website and check out the settings. Start from the top. To run the Fibonacci example start by setting the 'Testbench + Design' language to VHDL. Then select OSVVM under 'Libraries'. Specify the so called 'Top entity'. This is generally the highest order entity in your design. We will only use one entity called 'myEntity'. So specify 'myEntity' as top entity. Lastly, select GHDL 3-O-O under Tools and Simulators.

EDA Playground Settings



- Set Testbench + Design language to VHDL
- Set Libraries to `OSVVM`
  - Open Source VHDL Verification Methodology
- Specify "Top entity"
  - "main" entity: `myEntity` in this example (see testbench.vhd on next slide)
- Select `GHDL 3.0.0` as Simulator
  - Open Source VHDL Simulator



Now it's time to code! Move your attention to the right and check out the two input fields or editors. Put your code into the default testbench.vhd file in the left editor. This includes your top entity declarations and architecture for your entity. Put your process inside the architecture. GHDL requires a non-empty design file. So just put in some random empty design on the right-most editor.

EDA Playground Code

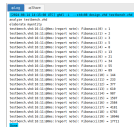
testbench.vhd

```
1  entity myEntity is
2  end entity;
3
4  architecture beh of myEntity is
5  begin
6
7    main: process
8      constant UPPER_BOUND : integer := 20;
9      variable current, previous, temp : integer := 0;
10   begin
11     previous := 0;
12     current := 1;
13     for i in 0 to UPPER_BOUND loop
14     temp := current;
15     current := previous + current;
16     previous := temp;
17     report "Fibonacci(" & to_string(i) & ") = "
18             & to_string(current);
19     end loop;
20   wait;
21   end process;
22 end architecture;
```

design.vhd

```
1  entity design is
2
3  end entity;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

- Put your code (entity + architecture) into the testbench file on the left
- GHDL requires a non-empty design file on the right



Now you can click 'run' to execute your VHDL code. If done correctly it should compile and print output on the bottom part of the webpage. In our case it printed our string as specified via the report command. Note that VHDL simulators usually print out a lot of additional information if printing via 'report'. In GHDL's case it's the filename and position of the report command in addition to the simulation timestamp. You can check the code and see that the report statement is indeed in line 16 and starts at character 11. The timestamp will become more important later when we dive into advanced simulations. And that's it for this session! You can play around with the provided example and get a better feeling of the basic VHDL syntax set as shown in this lecture. For this particular example keep in mind that the integer type is only 32-bit wide. So do not set the `UPPER_BOUND` constant arbitrarily big.

EDA Playground Output

Log | Share

```
[2024-08-14 14:15:08 UTC] ghdl -i  --std=08 design.vhd testbench.vhd
analyze testbench.vhd
elaborate myentity
testbench.vhd:16:11:@0ms:(report note): Fibonacci(0) = 1
testbench.vhd:16:11:@0ms:(report note): Fibonacci(1) = 2
testbench.vhd:16:11:@0ms:(report note): Fibonacci(2) = 3
testbench.vhd:16:11:@0ms:(report note): Fibonacci(3) = 5
testbench.vhd:16:11:@0ms:(report note): Fibonacci(4) = 8
testbench.vhd:16:11:@0ms:(report note): Fibonacci(5) = 13
testbench.vhd:16:11:@0ms:(report note): Fibonacci(6) = 21
testbench.vhd:16:11:@0ms:(report note): Fibonacci(7) = 34
testbench.vhd:16:11:@0ms:(report note): Fibonacci(8) = 55
testbench.vhd:16:11:@0ms:(report note): Fibonacci(9) = 89
testbench.vhd:16:11:@0ms:(report note): Fibonacci(10) = 144
testbench.vhd:16:11:@0ms:(report note): Fibonacci(11) = 233
testbench.vhd:16:11:@0ms:(report note): Fibonacci(12) = 377
testbench.vhd:16:11:@0ms:(report note): Fibonacci(13) = 610
testbench.vhd:16:11:@0ms:(report note): Fibonacci(14) = 987
testbench.vhd:16:11:@0ms:(report note): Fibonacci(15) = 1597
testbench.vhd:16:11:@0ms:(report note): Fibonacci(16) = 2584
testbench.vhd:16:11:@0ms:(report note): Fibonacci(17) = 4181
testbench.vhd:16:11:@0ms:(report note): Fibonacci(18) = 6765
testbench.vhd:16:11:@0ms:(report note): Fibonacci(19) = 10946
testbench.vhd:16:11:@0ms:(report note): Fibonacci(20) = 17711
Done
```

- Outputs Fibonacci sequence
- Caution if you try higher *UPPER_BOUND*: `integer` is 32-bit wide

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!