**Chapter 8: Names**

Chapter 8 focuses on the concept of names in VHDL, which are identifiers used to refer to objects, types, subprograms, and other design elements. Names ensure clarity, organization, and accessibility in a VHDL design.

**8.1 General**

* Purpose: Names uniquely identify design elements and enable referencing them in expressions, statements, and configurations.
* Types of Names: Includes simple names, selected names, indexed names, slice names, attribute names, and external names.

**8.2 Simple Names**

* Refer to declared entities within a specific scope**.**



**8.3 Selected Names**

* Refer to elements within a hierarchical design or package.
* Syntax: <prefix>.<suffix>

A close-up of a standard

Description automatically generated

**8.4 Indexed Names**

* Access individual elements in arrays or vectors using an index.
* Syntax: <array\_name>(<index>)



**8.5 Slice Names**

* Refer to a subset of an array or vector.
* Syntax: <array\_name>(<range>)

A screenshot of a computer error

Description automatically generated

**8.6 Attribute Names**

* Refer to attributes of objects, such as predefined attributes ('event, 'length, etc.) or user-defined attributes.
* Syntax: <object>'<attribute>

A white background with black text

Description automatically generated

**8.7 External Names**

* Refer to objects or entities external to the current design unit, useful for modular and hierarchical designs.



**From :** <https://adaptivesupport.amd.com/s/question/0D54U00007hX4mrSAC/vhdl-external-name-alias-for-internal-array-signal-not-being-resolved?language=en_US>

**Chapter 9: Expressions**

Chapter 9 focuses on expressions in VHDL, which are combinations of operators and operands used to compute values, control behavior, or specify conditions. Expressions are fundamental for describing hardware behavior.

**9.1 General**

Expressions: Formed by combining operands (constants, variables, signals, or literals) with operators.

Usage: Used in assignments, conditions, and computations throughout a design.

**9.2 Operators**

Operators define operations on operands. They are categorized as:

1. **Arithmetic Operators**:
   * Perform mathematical operations.
   * Examples: +, -, \*, /, mod, rem.
2. **Relational Operators**:
   * Compare two operands and return a boolean.
   * Examples: =, /=, <, <=, >, >=.
3. **Logical Operators**:
   * Perform bitwise or boolean operations.
   * Examples: and, or, nand, nor, xor, not.
4. **Shift Operators**:
   * Shift the bits of operands.
   * Examples: sll, srl, sla, sra, rol, ror.
5. **Concatenation Operator**:
   * Combines elements into an array or string.
   * Example: &.
6. **Miscellaneous Operators**:
   * Include unary operators like + and - for sign operations.

**9.3 Operands**

Operands are the entities acted upon by operators:

* **Literals**: Constant values like 10, '1', or "1010".
* **Names**: References to signals, variables, or constants.
* **Function Calls**: Results of subprogram executions.
* **Qualified Expressions**: Specify the type of a literal or expression.

**9.4 Static Expressions**

* Computed entirely at compile-time.
* Used in constant declarations or constraints.

A white rectangular object with black text

Description automatically generated

**9.5 Universal Expressions**

* Expressions involving universal types like integers or real numbers.
* Automatically converted to specific types when necessary.

**Chapter 10: Sequential Statements**

Chapter 10 describes sequential statements in VHDL, which are used to control the flow of execution in processes, functions, and procedures.

**10.1 General**

* Sequential Statements: Specify the behavior of a design in processes or subprograms.
* Execute in a defined sequence, unlike concurrent statements.

**10.2 Wait Statement**

* Suspends execution of a process until a condition is met.

**10.3 Assertion Statement**

* Verifies conditions during simulation and reports errors or warnings if conditions are false.

**10.4 Report Statement**

* Displays messages during simulation for debugging.

A white background with black text

Description automatically generated

**From:** <https://peterfab.com/ref/vhdl/vhdl_renerta/mobile/source/vhd00007.htm>

**10.5 Signal Assignment Statement**

* A white screen with black text

  Description automatically generatedAssigns values to signals

**10.6 Variable Assignment Statement**

* Updates the value of a variable immediately.
* Updates the value of a variable immediately.

**10.7 Procedure Call Statement**

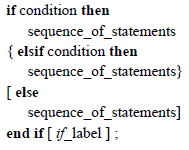
* Invokes a procedure to perform a specific task.

A computer code with black text

Description automatically generated

**From:** <https://peterfab.com/ref/vhdl/vhdl_renerta/mobile/source/vhd00052.htm>

**10.8 If Statement**

* Conditional execution of statements.

**10.9 Case Statement**

* A screenshot of a computer program

  Description automatically generatedExecutes one of several branches based on a value.

**10.10 Loop Statement**

* A close up of a text

  Description automatically generatedRepeats a set of statements

**10.11 Next Statement**

* A white screen with black text

  Description automatically generatedSkips the remaining statements in the current iteration of a loop.

**From:** https://ics.uci.edu/~jmoorkan/vhdlref/nexts.html

**10.12 Exit Statement**

* A white background with black text

  Description automatically generatedTerminates a loop prematurely.

**From:** <https://peterfab.com/ref/vhdl/vdlande/exits.html>

**10.13 Return Statement**

* Exits a subprogram and optionally returns a value.

A screenshot of a computer code

Description automatically generated

**From:** <https://docs.amd.com/r/en-US/ug901-vivado-synthesis/Relaxed-Return-Rules-for-Function-Return-Values>

**10.14 Null Statement**

* A black text with white text

  Description automatically generated with medium confidencePlaceholder for no operation.

From: <https://peterfab.com/ref/vhdl/vhdl_renerta/source/vhd00045.htm>

**Chapter 11: Concurrent Statements**

Chapter 11 describes concurrent statements in VHDL, which are fundamental to modeling parallelism in hardware. These statements operate independently and concurrently, reflecting the behavior of hardware components.

**11.1 General**

* **Concurrent Statements**: Specify behavior that executes in parallel, unlike sequential statements which execute in a defined order.
* Used in architectures, block statements, and generate statements.
* Types of concurrent statements:
  + Process statements.
  + Component instantiations.
  + Concurrent procedure calls.
  + Signal assignments.

**11.2 Block Statement**

* Groups concurrent statements into a single block with its own declarative region.
* Supports hierarchy and scope management.

**11.3 Process Statement**

* Encapsulates sequential statements in a concurrent environment.
* Executes when any signal in the sensitivity list changes.

**11.4 Concurrent Procedure Call Statements**

* Calls a procedure to perform a task in a concurrent environment.

**11.5 Concurrent Assertion Statements**

* Asserts conditions during simulation to verify design correctness.

**11.6 Concurrent Signal Assignment Statements**

* Assigns values to signals, including:
  1. **Simple Signal Assignment**: Updates the value of a signal.
  2. **Conditional Signal Assignment**: Assigns values based on conditions.
  3. **Selected Signal Assignment**: Assigns values based on a selection expression.

**11.7 Component Instantiation Statements**

* Used to instantiate design components in an architecture.
* Connects the component's ports to signals or expressions.

**11.8 Generate Statements**

* Creates repetitive or conditional structures in designs.
* **For-Generate**: Repeats a block or statement for a range of values.
* **If-Generate**: Includes blocks conditionally.

**Chapter 12: Scope and Visibility**

Chapter 12 focuses on how declarations and objects in VHDL are organized, made accessible, and referenced within a design. It explains the rules of scope, visibility, and how declarations interact within different regions of a VHDL description.

**12.1 Declarative Region**

* A **declarative region** is a logical area in a VHDL design where declarations (e.g., signals, types, constants) can appear.
* Declarative regions are defined by constructs such as:
  + Entity declarations.
  + Architecture bodies.
  + Processes, blocks, and packages.

**12.2 Scope of Declarations**

* **Scope** refers to the region of the design where a declared entity can be referenced.
* **Rules**:
  + The scope begins immediately after the declaration and extends to the end of the enclosing declarative region.
  + Nested regions can reference declarations in their parent regions unless shadowed by local declarations.

**12.3 Visibility**

* **Visibility** determines whether a name in a declaration is accessible in a given part of the design.
* **Direct Visibility**:
  + A name is directly visible if it is declared in the current declarative region or explicitly imported using a use clause.
* **Selected Visibility**:
  + Names declared in packages or external entities are accessed using a selected name.
* **Overloading**:
  + Multiple declarations with the same name are allowed if they are distinguishable by context.

**12.4 Use Clauses**

* **Purpose**: Make external declarations visible in the current scope.

**12.5 The Context of Overload Resolution**

* When multiple entities with the same name are visible, the compiler resolves which entity to use based on the context.

**Chapter 13: Design Units and Their Analysis**

Chapter 13 explains the organization of VHDL designs into modular units called design units. It describes how these units are analyzed, stored, and reused in a VHDL environment.

**13.1 Design Units**

* A VHDL design is divided into **design units** that can be compiled, stored, and reused independently.
* **Types of Design Units**:
  1. **Entity Declaration**: Defines the interface (inputs/outputs) of a design.
  2. **Architecture Body**: Implements the behavior or structure of the design.
  3. **Package Declaration**: Contains reusable declarations (e.g., types, constants, and subprograms).
  4. **Package Body**: Implements subprograms and defines private data for a package.
  5. **Configuration Declaration**: Specifies the bindings of components to entities and architectures.
  6. **Context Declaration**: Specifies references to libraries and packages used across design units.

**13.2 Design Libraries**

* A **design library** is a storage area for compiled design units.
* Design units are analyzed (compiled) and stored in libraries for later use.
* The **work library** is the default library where newly compiled units are stored.

**13.3 Context Declarations**

* Context declarations simplify reuse by centralizing references to libraries and packages.

**13.4 Context Clauses**

* Used to include library and package references directly in a design unit.

**13.5 Order of Analysis**

* Design units are analyzed in dependency order:
  + Entities must be analyzed before their architectures.
  + Packages must be analyzed before their bodies or dependent units.
  + Configuration declarations depend on the availability of entities and architectures.
* Errors in dependencies will halt the analysis process.

**Chapter 14: Elaboration and Execution**

Chapter 14 explains the process of **elaboration** and **execution** in VHDL. These are critical steps in preparing and running a VHDL design simulation, ensuring that all design components are properly initialized and interconnected before simulation begins.

**14.1 General**

* **Elaboration**: The process of constructing the simulation model from design units, initializing data, and resolving dependencies.
* **Execution**: The simulation phase where the VHDL model operates according to the specified behavior.

**14.2 Elaboration of a Design Hierarchy**

* Involves constructing the complete hierarchy of a design, starting from the top-level entity.
* Key steps:
  1. **Binding**: Components in the design are bound to specific entities and architectures.
  2. **Instance Creation**: Design entities and components are instantiated in the hierarchy.
  3. **Connection**: Signals and ports are connected to ensure proper communication.

**14.3 Elaboration of a Block, Package, Subprogram, or Protected Type Header**

* Each block or design unit undergoes elaboration to:
  + Declare and initialize constants, variables, signals, and types.
  + Prepare subprograms (procedures and functions) and protected types for use.

**14.4 Elaboration of a Declarative Part**

* Handles declarations within entities, architectures, and packages.
* Ensures that:
  + Objects like signals, variables, constants, and attributes are initialized.
  + Dependencies between declarations are resolved.

**14.5 Elaboration of a Statement Part**

* Prepares the statements within architectures, processes, and subprograms for execution.
* Concurrent statements (e.g., signal assignments) are scheduled for execution during simulation.

**14.6 Dynamic Elaboration**

* Some constructs (e.g., generate statements) are elaborated dynamically based on conditions or ranges.
* Enables conditional or repetitive instantiation of components and blocks.

**14.7 Execution of a Model**

* Simulation begins after elaboration is complete.
* **Phases of Execution**:
  1. **Initialization**: All signals and objects are initialized to their default or declared values.
  2. **Simulation Cycles**:
     + Evaluate concurrent statements and processes.
     + Update signals and propagate changes.
     + Repeat until the simulation ends.
  3. **Termination**: Simulation ends when a specified condition is met, or no further updates occur.

**Chapter 15: Lexical Elements**

Chapter 15 focuses on the **lexical elements** in VHDL, which are the basic building blocks of the language. These include characters, identifiers, literals, and other tokens that make up VHDL code.

**15.1 General**

* **Lexical Elements**: Basic components of VHDL syntax that are combined to form statements and declarations.
* VHDL is case-insensitive (e.g., signal, SIGNAL, and Signal are treated the same).

**15.2 Character Set**

* VHDL uses a subset of the ISO Latin-1 character set, which includes:
  + Letters (A–Z, a–z).
  + Digits (0–9).
  + Special symbols (e.g., +, -, \*, /, =, ;, :).
  + Space, tab, and newline for formatting.

**15.3 Lexical Elements, Separators, and Delimiters**

* **Tokens**: Atomic units of VHDL syntax, including identifiers, reserved words, literals, and operators.
* **Separators**: Include spaces, tabs, and newlines, which separate tokens.
* **Delimiters**: Symbols that mark the beginning or end of constructs (e.g., parentheses ( and ), commas ,).

**15.4 Identifiers**

* Names for entities like signals, variables, constants, and types.
* Rules:
  + Must begin with a letter.
  + Can include letters, digits, and underscores (\_), but cannot end with an underscore or have consecutive underscores.
  + Cannot be a reserved word.

**15.5 Abstract Literals**

* Represent numeric values.
* Types:
  + **Integer Literals**: Whole numbers (e.g., 42, -10).
  + **Real Literals**: Decimal numbers (e.g., 3.14, -0.5).

**15.6 Character Literals**

* Enclosed in single quotes (') and represent a single character.

**15.7 String Literals**

* Enclosed in double quotes (") and represent sequences of characters.

**15.8 Bit String Literals**

* Represent binary, octal, or hexadecimal values.
* Prefixes:
  + B for binary, O for octal, X for hexadecimal.
* Enclosed in double quotes.

**15.9 Comments**

* Begin with -- and extend to the end of the line.
* Used for documentation or explanations.

**15.10 Reserved Words**

* Reserved words have predefined meanings and cannot be used as identifiers.
* Examples: architecture, entity, signal, process, begin.

**15.11 Tool Directives**

* Special comments understood by tools (e.g., simulation or synthesis tools).
* Begin with -- pragma and include tool-specific commands.

**Chapter 16: Predefined Language Environment**

Chapter 16 explains the **predefined environment** of VHDL, which consists of built-in attributes, types, packages, and subprograms that form the standard foundation for VHDL designs. These predefined elements ensure portability and consistency across tools and designs.

**16.1 General**

* VHDL provides a set of predefined elements to support commonly needed functionality.
* These elements are always available without additional declarations or imports.

**16.2 Predefined Attributes**

* Attributes provide information about objects, types, and signals, or define specific behaviors.
* Categories:
  1. **Value Attributes**:
     + Examples: 'left, 'right, 'high, 'low, 'length.
     + Provide information about ranges and bounds.
  2. **Functional Attributes**:
     + Examples: 'event, 'active, 'last\_value.
     + Provide runtime information about signals or objects.
  3. **Type Attributes**:
     + Examples: 'base, 'pos, 'val.
     + Provide operations or details about scalar types.
  4. **Array Attributes**:
     + Examples: 'range, 'reverse\_range, 'ascending.
     + Describe properties of arrays or vectors.

**16.3 Package STANDARD**

* The **STANDARD** package is automatically available and defines basic data types, such as:
  + **Scalar Types**: integer, real, boolean, character.
  + **Physical Types**: time.
  + Operators for arithmetic, relational, and logical operations.
* Includes basic subprograms for type conversion and mathematical operations.

**16.4 Package TEXTIO**

* Supports file input/output operations.
* Provides predefined types and procedures for reading and writing to text files.
* **Types**:
  + line: Represents a line of text.
* **Procedures**:
  + read, write, endfile

**16.5 Standard Environment Package**

* Defines utilities for simulation and testing.
* Includes functions like assert and stop for simulation control.

**16.6 Standard Mathematical Packages**

* Provide extended mathematical operations.
* Examples: Trigonometric functions (sin, cos), exponential functions, and logarithms.

**16.7 Standard Multivalue Logic Package**

* **std\_logic\_1164**: Defines std\_logic and std\_ulogic types for multi-valued logic.
  + Used for modeling digital circuits.
  + Includes logical operators (and, or, xor) and resolution functions.

**16.8 Standard Synthesis Packages**

* Define attributes and subprograms for synthesis tools.
* Ensure compatibility between simulation and synthesis.

**16.9 Fixed-Point and Floating-Point Packages**

* **Fixed-Point Package**:
  + Provides fixed-point arithmetic types and operations.
  + Example: ufixed, sfixed (unsigned and signed fixed-point).
* **Floating-Point Package**:
  + Provides floating-point arithmetic for high-precision calculations.
  + Example: float, float\_pkg.

**16.12 Reflection Package**

* Provides meta-programming capabilities.
* Allows querying and manipulating types, attributes, and objects dynamically.

**Chapter 17: VHDL Procedural Interface (VHPI) Overview**

Chapter 17 introduces the **VHDL Procedural Interface (VHPI)**, a standardized mechanism to integrate external software tools and programs with VHDL simulation environments. VHPI is essential for extending VHDL's functionality, enabling interaction with external systems for simulation, analysis, or tool integration.

**17.1 General**

* VHPI provides a procedural (function-call) interface for accessing and controlling VHDL simulation environments.
* **Purpose**:
  + Enhance interoperability between VHDL simulators and external software tools.
  + Allow runtime manipulation and observation of VHDL models.

**17.2 Organization of the Interface**

* VHPI is organized into a well-defined hierarchy and categorized into capability sets, access functions, and handles.
* Key components:
  1. **Capability Sets**: Define what functionalities a simulator provides.
  2. **Handles**: Represent VHDL objects or simulator elements (e.g., signals, variables, or processes).
  3. **Access Functions**: Facilitate interactions with the simulation environment.

**17.3 Capability Sets**

* Capabilities are sets of features that a simulator supports, allowing users to query its compatibility and functionality.
* Examples of capabilities:
  + Reading and writing values of VHDL objects.
  + Accessing simulation time.
  + Triggering callbacks for specific events.
* Simulators may implement a subset of VHPI features, and capability queries ensure compatibility checks.

**17.4 Handles**

* Handles are the central abstraction in VHPI, representing simulation objects (signals, variables, instances, etc.).
* **Types of Handles**:
  + **Object Handles**: Represent VHDL objects like signals, constants, or variables.
  + **Entity/Instance Handles**: Represent design entities or instantiated components.
  + **Process Handles**: Represent concurrent processes in the simulation.
* **Use of Handles**:
  + Handles are retrieved using specific VHPI functions and are used to perform operations like value reading, writing, or querying properties.

**Key Concepts in VHPI**

1. **Access Functions**:
   * Provide procedural access to simulation elements through handles.
   * Allow querying, reading, or modifying simulation states dynamically.
2. **Callback Mechanism**:
   * Enable event-driven interaction.
   * Users can register functions to be executed upon specific simulation events (e.g., signal changes or process activations).
3. **Simulator Interoperability**:
   * VHPI ensures a standardized way of interacting with different simulators, improving portability of tools and extensions.

**Applications of VHPI**

1. **Debugging and Monitoring**:
   * External tools can observe signals or variables during simulation to aid debugging.
2. **Dynamic Model Interaction**:
   * Allows runtime injection of stimuli or modification of simulation states.
3. **Custom Verification**:
   * Integrate custom verification tools or frameworks with VHDL simulators.
4. **Co-Simulation**:
   * Enable integration of VHDL models with other simulation environments (e.g., system-level or mixed-signal simulation).