cadence®

Cadence VHDL-AMS Overview

Product Version 22.09
April 2022

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Preface

This manual describes the analog and mixed-signal aspects of the Cadence® VHDL-AMS language. With VHDL-AMS, you can create and use modules that describe the high-level behavior and structure of analog, digital, and mixed-signal components and systems. The guidance given here is designed for users who are familiar with the development, design, and simulation of circuits and with high-level programming languages.

The preface discusses the following:

Related Documents

For more information about VHDL-AMS and related products, consult the sources listed below.

- *Cadence AMS Designer Environment User Guide*
- *Cadence AMS Designer Simulator User Guide*
- *Virtuoso Analog Design Environment User Guide*
- *Virtuoso Mixed-Signal Circuit Design Environment User Guide*
- *SimVision Analysis Environment User Guide*
- *Spectre Circuit Simulator Reference*
- *Spectre Circuit Simulator User Guide*
- *Cadence Hierarchy Editor User Guide*
- *Component Description Format User Guide*
- *IEEE Standard VHDL Analog and Mixed-Signal Extensions, IEEE Std 1076.1-1999. Available from IEEE.*
- *Instance-Based View Switching Application Note*
- *Cadence Library Manager User Guide*
- *Virtuoso Schematic Composer User Guide*

Internet Mail Address

You can send product enhancement requests and report problems to Customer Support. For current phone numbers and e-mail addresses, go to Cadence Online Support and click the *Contact Us* link on the Home page.

Please include the following in your e-mail:

- The license server host ID

  To determine what your server's host ID is, use the Subscription Service of Cadence Online Support for assistance.

- A description of the problem
- The version of the VHDL-AMS product that you are using

- Analog simulation control files, top-level modules and all included files including hardware design language (HDL) modules so that Customer Support can reproduce the problem
- Output logs and error messages

Typographic and Syntax Conventions

Special typographical conventions distinguish certain kinds of text in this document. The formal syntax in this reference uses the definition operator, `::=`, to define the more complex elements of the VHDL-AMS language in terms of less complex elements.

- Lowercase words represent syntactic categories. For example,

  ```
  module_declaration
  ```

  Some names begin with a part that indicates how the name is used. For example,

  ```
  node_identifier
  ```

  represents an identifier that is used to declare or reference a node.

- Boldface words represent elements of the syntax that must be used exactly as presented. Such items include keywords, operators, and punctuation marks. For example,

  ```
  end block
  ```

- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,

  ```
  logical_operator ::=
          and
      |   or
      |   nand
      |   nor
      |   xor
      |   xnor
  ```

- Square brackets enclose optional items. For example,

  ```
  subprogram_body ::=
      [pure|impure]
  ```

- Braces enclose an item that can be repeated zero or more times. For example,

  ```
  architecture_body ::=
      architecture identifier of entity_name is
          {block_declarative_part}
  ```

Code examples are displayed in constant-width font.

```
---This is an example of the font used for code.
```

Within the text, variables are in italic font, like this: *allowed_errors*.

Within the text, keywords, reserved words and filenames are set in constant-width font, like this: reserved_word, file_name

If a statement is too long to fit on one line, the remainder of the statement is indented on the next line, like this:

```
qgf = width*length*cfbb*(vgfs - wkf - qb/(2*cbb) -
        (vgbs - vfbb + qb/(2*cob))) + qgf_par ;
```

1

Introduction

What is VHDL-AMS?

The VHDL language has been extended to support analog and mixed-signal design. This extension is referred to as VHDL-AMS and is defined in the IEEE 1076.1 standard. The extensions to the core IEEE 1076 standard support analog behavioral modeling. These extensions support both conservative system and signal flow system semantics.

When you use the Cadence AMS Designer environment, you can directly instantiate textual VHDL-AMS views — along with VHDL (digital), Verilog-A, Verilog-AMS, Verilog (digital), and Spectre primitive views — from within the Virtuoso schematic composer tool.

Extensions to VHDL

Since VHDL first became a standard, a number of extensions have been added to the language. These extensions allow the language to be used for digital synthesis, gate level simulation, and now analog and mixed-signal modeling and simulation.

The Cadence solution currently supports the following extensions:

- *ANSI/IEEE Std 1076-1993, IEEE standard VHDL Language Reference Manual*
- *IEEE Std 1076-1987 IEEE, standard VHDL Language Reference Manual*
- *IEEE Std 1076.1-1999 IEEE, standard VHDL Analog and Mixed-signal Extensions*
- *IEEE Std 1076.2-1996 IEEE, Standard VHDL Mathematical Packages*
- *IEEE Std 1076.3-1997 IEEE, Standard VHDL Synthesis Packages*
- *IEEE Std 1076.4-2000, IEEE standard for VITAL ASIC (application specific integrated circuit) Modeling Specification*
- *IEEE Std 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*

If you plan to use VHDL for digital synthesis, be aware that digital synthesis tools support only the subset of the language specified by the 1076.6-1999 and 1076.3-1997 extensions.

Benefits of VHDL-AMS

VHDL-AMS provides many benefits, including:

- The ability to model discrete and continuous time systems at various levels of abstraction. Most of the constructs required for modeling are built into the language; others are provided by standardized packages.
- Support for a variety of modeling styles, including dataflow modeling, modeling conservative systems, and creating structural designs. The support that VHDL-AMS provides for simulation, description, and digital synthesis means the language is useful throughout the design cycle. The ability to describe designs at the data flow and analog behavioral levels allows you to concentrate on the concept of the design rather than on low level details.
- Support for analog behavioral models, which allow for significantly faster simulations at the system level.
- The ability to access libraries of predefined components.
- Support for packages, which facilitates sharing a group of design data, types, functions, and procedures under a single name.
- Support for hierarchies, so that design entities can be instantiated within other design entities as components. The process of binding design entities to the component references in hierarchical descriptions supports an iterative, fluid approach to development.
- Support for binding components with entity architecture pairs under the control of configurations. Both Cadence configurations and VHDL configurations are supported.
- Wide support and portability to other design environments. The fact that VHDL-AMS is non-proprietary means that there is a large selection of VHDL and VHDL-AMS models available.
- The ability, when using the AMS Designer environment, to incorporate and simulate two standard, public languages (VHDL-AMS and Verilog-AMS) in the same design.
- Support for bottom-up verification, which allows for faster full chip verification.

Implications of Using VHDL-AMS

The potential lifetime of a design is extended beyond the lifetime of the underlying hardware technology, protecting against technology changes. A shorter development time is also likely due to the ability to verify design concepts before deciding on hardware.

Textual descriptions augment schematics by allowing definition of the behavior of a cell rather than just selecting predefined cells from a library.

VHDL-AMS promotes the use of more simulation during the development phase, allowing designers to explore a wider set of alternative solutions to a given problem.

Adopting a consistent style of coding with the ability to make modifications quickly encourages designers to share their models and designs thus increasing IP reuse. Further advantages are also found as you consider digital synthesis, design for test, and characterization solutions.

References to the VHDL-AMS Language Reference

This document refers frequently to the *IEEE Standard VHDL Analog and Mixed-Signal Extensions Language Reference*, using a notation like, “See LRM 3.2,” where the numbers indicate the relevant section and subsection in the reference.

Using VHDL-AMS with Other Languages

For more information about using VHDL-AMS with other languages in a mixed-signal design, see chapter 4, “Preparing the Design: Using Mixed Languages,” in *Virtuoso AMS Designer Simulator User Guide*.

2

VHDL-AMS Modeling Styles

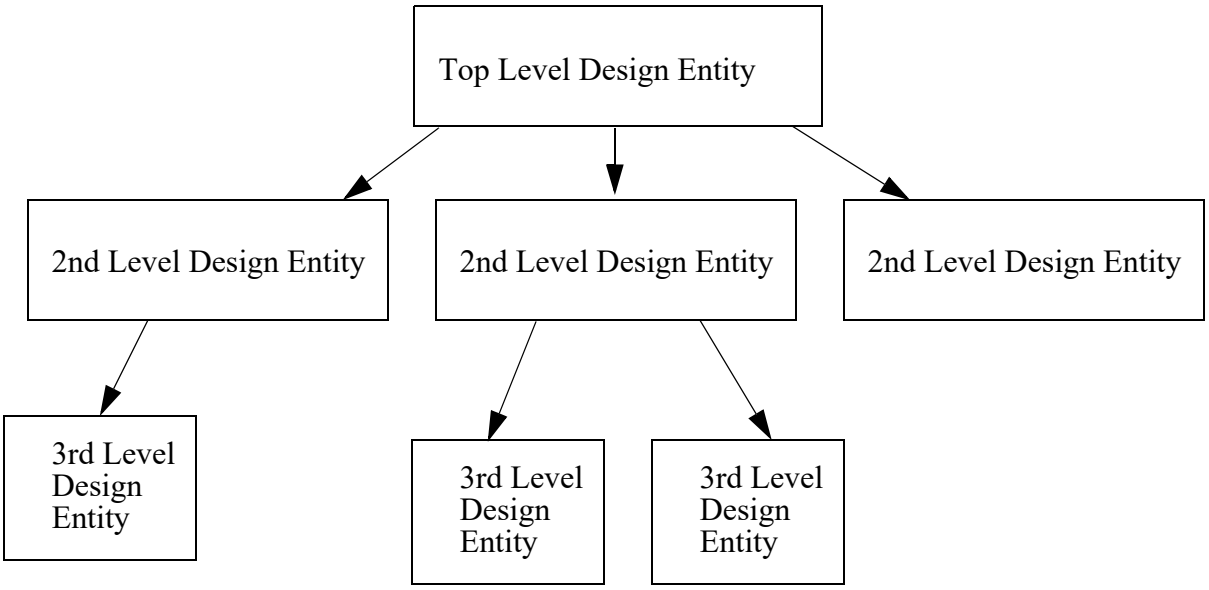
You can read about the following topics in this chapter:

Levels of Abstraction

Design abstraction is a strategy for managing complexity and hiding and showing details. At lower levels of abstraction, more details are exposed and the behavior of the model is closer to the complexity of the physical implementation of a design. At higher levels of abstraction, less detail is exposed and the behavior of the model is further from the physical implementation of the design. Typically, designs using higher levels of abstraction simulate faster that designs using lower levels of abstraction.

VHDL-AMS provides blocks that you can use to partition and organize designs into hierarchies. Multiple levels of nested blocks are allowed. The internal details of these relationships can be shown or hidden from view, as desired. The hierarchy consists of behavioral and structural elements. The structural elements define the parent/child

relationships. As illustrated in the following diagram, these relationships can be represented as a tree structure



Analog Abstraction Hierarchy

The tables below list the levels of abstraction most often associated with analog circuits.

.

Table 2-1 Abstraction Hierarchy for the Analog Functional Domain

Level	Description
Functional or behavioral signal flow	Describes signal flow input/output relations by mathematical functions. For example, you might model an op-amp as an output that is a function of its input.
Functional or behavioral conservative	Uses equations to describe relations between terminals. You might use this level of abstraction to model a capacitor that takes second order effects into account.
Ideal equations	Uses idealized device models
Characteristic equations	Uses device models that include second order effects

Table 2-2 Abstraction Hierarchy for the Analog Structural Domain

Level	Description
System transfer functions	Models signal flow between system level blocks
Macro Models	Uses a parameterizable block of device models to model some complex function
Ideal sources and devices	Uses idealized device models
Nets and devices	Uses primitive devices and wire models that include second order effects

Conservative Systems

A *conservative system* is one that obeys the laws of conservation described by Kirchhoff's Potential and Flow laws. Across quantities define the potential between two terminals. Through quantities define the flow through a terminal (to another terminal).

Terminals

A terminal is a point of physical connection between devices of continuous-time descriptions. Terminals obey conservation-law semantics.

Reference Terminal

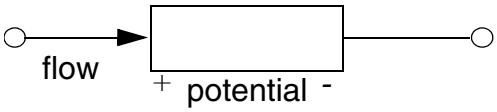
The potential of a single terminal is defined with respect to a reference terminal. The reference terminal, often called *ground* in electrical systems, has a potential of zero.

```
NATURE electrical IS voltage ACROSS current THROUGH ground REFERENCE;
```

The reference terminal is declared as part of the nature declaration, can have any name, but is a unique terminal. A terminal (node in Verilog) cannot be arbitrarily declared to be the reference. See LRM 4.8.

Reference Directions

Each branch quantity has a reference direction. For example, consider the following schematic. With the reference direction shown, the potential in this schematic is positive whenever the potential of the terminal marked with a plus sign is larger than the potential of the terminal marked with a minus sign.



The direction of a branch quantity is determined by the order of the terminals in the terminal_aspect of the branch quantity declaration. See LRM 4.3.1.6. So, for example:

```
QUANTITY vx ACROSS ix THROUGH t1 TO t2;
```

declares an across quantity `vx` from terminal `t1` to terminal `t2`. The value of `vx` is positive if terminal `t1` has a higher potential than terminal `t2` (it would be negative if `t2` had a higher potential than `t1`). It also declares a through quantity `ix` through `t1` to `t2`. If the flow is from `t1` to `t2` then the value of `ix` is positive (if the flow is from `t2` to `t1` then the value of `ix` is negative). If we change the declaration to:

```
QUANTITY vx ACROSS ix THROUGH t2 TO t1;
```

the value of `vx` and `ix` change signs.

Analog Systems

For analog systems, the simulator uses Kirchhoff's laws to develop equations that define the values and flows in the system. Because the equations can be differential and nonlinear, the simulator does not solve them directly. Instead, the simulator uses an approximation and solves the equations iteratively at individual time points (also called solution points). The simulator controls the interval between the time points to ensure the accuracy of the approximation.

At each time point, iteration continues until two convergence criteria are satisfied. The first criterion requires that the approximate solution on this iteration be close to the accepted solution on the previous iteration. The second criterion requires that Kirchhoff's Flow Law be adequately satisfied. To determine the required accuracy for these criteria, the simulator uses the tolerances specified in the design.

An analog model contains three equation sets: the explicit set, the structural set, and the augmentation set. The explicit set is derived from the simultaneous statements that describe the signal flow behavior of free quantities and the branch behavior of branch quantities. The structural set, which is derived from Kirchoff's Laws, constrains a network of branches to obey conservation of charge and potential. The augmentation set describes how `'DOT` and `'INTEG` are defined under various conditions such as computing initial state or solving a time integration step. At each solution point, the analog solver finds a simultaneous solution to the combined equations from all three sets.

Simultaneous Statements

Simultaneous statements are algebraic and differential equations used to define the analog behavior of a system. For example:

```
simple_expression == simple_expression;
```

`simple-expression` can contain linear, nonlinear, or differential equations involving operations on any value bearing construct (such as quantities, variables, constants, attributes, and arrays of constants, generics or signals).

For example, the following signal flow description produces the sum and product of its inputs:

```
entity am is
    port(quantity in1, in2: in real; quantity outsum, outmult: out real);
end am;

architecture am_behav of am is
begin
    outsum == in1 + in2;
    outmult == in1 * in2;
end am_behav;
```

You can define dynamic relationships between the inputs and outputs of modules. For example, here is a conservative model behavioral description for a capacitor:

```
entity cap is
    generic(c : real := 1.0e-3);
    port (TERMINAL n, p: electrical);
end cap;

architecture cap_behav of cap is
    quantity vcap across icap through n to p;
begin
    icap == vcap'dot * c;
end cap_behav;
```

The `'integ` attribute is used to relate quantities to time integrals of other quantities. For example, the following signal flow model sets the output to the integral of the input (with respect to time). The `break` statement specifies the initial conditions on `input'integ`, which are needed during DC analysis.

```
entity integrator is
    port (quantity input: in real; quantity output : out real);
end integrator;

architecture integrator_behav of integrator is
begin
    break input'integ => 0.0;
    output == input'integ;
end integrator_behav;
```

You can perform index and slice operations on signal arrays in simultaneous equations. For example:

```
Signal s : real_vector(0 to 3) := (0.0, 0.0, 0.0, 0.0);
Quantity q1 : real_vector (0 to 3);
Quantity  q2 : real;
q1 == s;
q2 == s(2);
```

If you are using the Spectre or UltraSim solver with the simulation front end (SFE) parser, you can also perform index and slice operations on constant and generic arrays in simultaneous equations. For example:

```
s == paramB( 1 downto 0 );
```

Note: The array slice must have the same direction (such as `downto`) as the array declaration.

Important

You can perform index and slice operations on constant, generic, and signal arrays only in simultaneous statements. You must be using the SFE parser in order to use constant and generic arrays in this way.

You can assign a whole quantity array to a whole constant, generic, or signal array. For example:

```
r == paramA;
```

You can access individual bits of a constant, generic, or signal array using another constant (such as `constIndex` here):

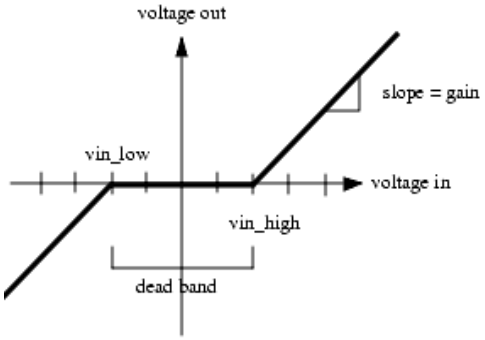
```
t == paramB( constIndex );
```

In an analog context (and using the SFE parser), you can pass the result of a constant or generic indexed expression to a user-defined function. For example:

```
x == addParams( q(0) - paramB(0), q(1) + paramB(1) );
```

Conditional Behavior in Simultaneous Statements

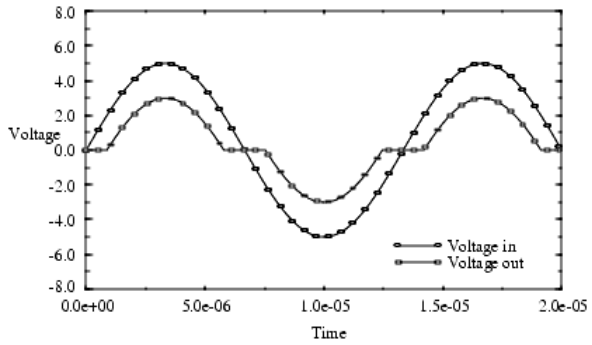
You can use simultaneous conditional statements to define behavior in regions. The following architecture describes a voltage deadband amplifier `vdba`. For example, the conditional `if-else` structure describes the piecewise linear approximation of the characteristics of the amplifier. If the input voltage is greater than `vin_high` or less than `vin_low`, the amplifier is active. When the amplifier is active, the output is the `gain` times the differential voltage between the input voltage and the edge of the deadband. When the input is in the deadband between `vin_low` and `vin_high`, the amplifier is quiescent, and the output voltage is zero.




```
ENTITY vdba IS
    GENERIC (vin_low : REAL := -2.0; vin_high : REAL := 2.0; gain : REAL := 1.0);
    PORT (QUANTITY input : IN REAL; QUANTITY output : OUT REAL);
END vdba;

ARCHITECTURE vdba_behav OF vdba IS
BEGIN
    IF (input >= vin_high) USE
        output == gain * (input - vin_high);
    ELSIF (input <= vin_low) USE
        output == gain * (input - vin_low);
    ELSE
        output == 0.0;
    END USE;
END vdba_behav;
```

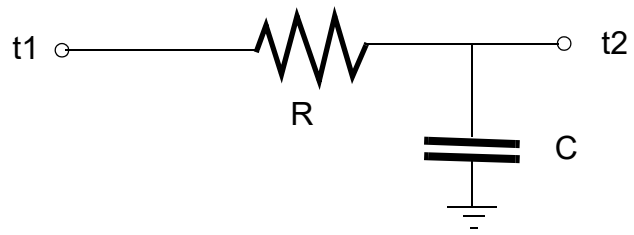
The following graph shows the response of the amplifier to a sinusoidal source.



Design Hierarchy

Hierarchy is supported by means of *instantiation*, which refers to the process of creating an instance of a design entity inside another design entity. Such instances are referred to as components.

Here is an example of an architecture that contains component instantiation.




```
ENTITY rc IS
    PORT (TERMINAL t1, t2: electrical);
END rc;

ARCHITECTURE rc_behav OF rc IS
    component cap IS
        port (terminal t1, t2: electrical);
        generic (rval : real := 1.0e-3);
    end component;
    for all : cap_comp use entity work.cap(cap_behav);
    component resistor is
        port (terminal t1, t2: electrical);
        generic (cval := real := 1000.0);
    end component;
    for all : resistor_comp use entity work.resistor(resistor_behav);
    r1: resistor_comp
        generic map(2000.0)
        port map(t1, t2);
    c1 : capacitor_comp
        generic map (2.0e-3)
```

```
        port map(t2, gnd);
END rc_behav;
```

Digital Abstraction Hierarchy

Just as analog blocks have different levels of abstraction, digital models can also be categorized according to their level of abstraction. The following table shows the abstraction hierarchy associated with digital models and systems.

Level	Modeling Method	Structural Primitive	Time Model
System level	Cooperating processing units	CPU, memory, bus	Causality
Chip level	Parallel algorithms	Controller, RAM, ROM, UART	Discrete (fine/coarse granularity)
Register transfer level	Guarded commands	Register, counter, ALU, multiplexor	Discrete (coarse granularity)
Logic gate level	Boolean logic equations	Gate, flip-flop	Discrete (fine granularity)
Circuit level	Differential equations	Transistor, R, L, C	Continuous

A digital description can use several of these different styles simultaneously.

System Level

As the name suggests, a system-level design can describe a complete system, comprising multiple PCBs connected to a backplane bus, although a system level description does not always contain a vast amount of structural information. In addition to the models of the hardware components making up a system, system-level designs often model the bus systems or networks used to interconnect the components. For example, a VHDL description of a computer interface card might be verified by using a VHDL model of the computer system bus and connecting the two of them together to form a testbench.

Chip Level

Chip-level VHDL descriptions fall into two general categories: those intended for synthesis, and those intended for simulation. The first category consists of RTL descriptions forming a top-level structure without any detailed timing information other than basic clocking relationships. The chip being described is typically an FPGA or cell-based ASIC.

The second category of chip-level descriptions are those intended for simulation. These descriptions are primarily behavioral models whose purpose is to accurately simulate the behavior of the chip at the pin level. The internal organization of the model is not important. To achieve this, the models generally include detailed timing behavior, extracted from the manufacturer's data sheets, concerning the behavior of the device interface pins under most conditions.

Many IC manufacturers are producing accurate VHDL models for their standard devices. Models are available for a large range of devices covering simple gates to full-blown microprocessors. The primary purpose of these models is to enable a designer to perform a realistic simulation of a full system.

Register Transfer Level

Register transfer level (RTL) descriptions are the most commonly used style of VHDL description. At this level, systems are described in terms of combinational and sequential functions at the behavioral level. The functions are often described using individual processes and interconnected using signals to form a dataflow. Typical functions included in an RTL description are registers, counters, and state machines along with combinational functions such as multiplexors, arithmetic units, and decoders. The RTL style can include some structural information via the dataflow. However, the individual functional blocks are described using behavioral constructs rather than instantiated gates and flip-flops. Many designs consist of registers interspersed with combinational functions and together these form the datapath. Transfers between registers and operations carried out by combinational functions are controlled by the controller or control path, which is usually a behavioral description of a finite state machine.

The RTL level of description is particularly relevant to users of logic synthesis, because this is the accepted level for designs that are intended for synthesis.

Logic Gate Level

VHDL-AMS provides many features for the support of gate-level simulation, the most useful being the Boolean operators (AND, OR, XOR, etc.), which correspond directly to the gate-level primitives. Boolean circuit algebra does not directly address the important delay time of the components so time values must be attributed by additional models.

In addition to basic logic gates, this level includes other low-level primitives such as flip-flops and latches. VHDL-AMS descriptions written at this level are referred to as netlists or wirelists because they consist of a list of components and interconnections. VHDL netlists are often generated automatically from a schematic or as part of the output of a synthesis tool.

Gate-level simulation is often performed during the latter stages of the design process after the design has been synthesized from a higher level description. Such synthesized designs can contain hundreds of thousands of gates so simulation performance is an important issue.

Circuit Level

The circuit level consists of interconnected active and passive components. Circuit behavior within time domain is represented by means of nonlinear differential equations. This level acts as a link to physical representations of digital circuits.

Mixed-Signal Systems

Mixed-signal systems and circuits generate and consume both continuous and discrete signals in the digital partition, while in the analog partition the values are continuous. Digital operates in the discrete time domain while analog operates with continuous time values so signal values must be converted as they move from one domain to the other. Each partition can be denoted at any abstraction level.

3

Example: Design Entity

A design entity represents a portion of a hardware design that performs a well-defined function and has well-defined inputs and outputs. A design entity can represent an entire system, a subsystem, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A design entity is defined by an entity declaration together with a corresponding architecture body.

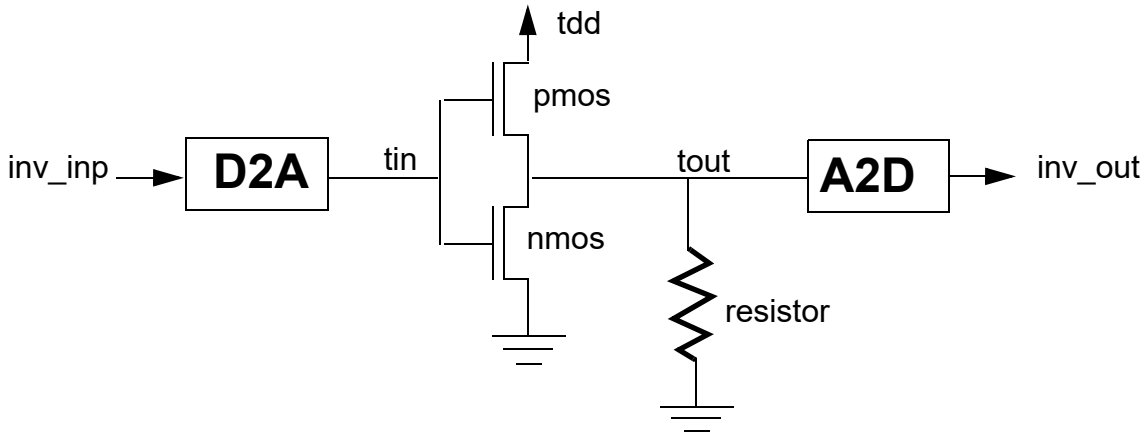
Design entities can be described by using a hierarchy of blocks where each block represents a portion of the whole design. The design entity itself is the top-level block, known as an external block, that resides in a library and can be used as a component of other designs. Internal blocks are nested in the hierarchy and defined by block statements.

A design entity can also be described in terms of interconnected components. Each component can be bound to a lower-level design entity to define the structure or behavior of that component. Successive decomposition of a design entity into components, and binding those components to other design entities that can be decomposed in the same manner, results in a hierarchy of design entities representing a complete design. A collection of design entities is a design hierarchy. The bindings necessary to identify a design hierarchy are specified in a configuration of the top-level entity in the hierarchy. A configuration is defined by a configuration declaration. A configuration is used to describe how design entities are put together to form a complete design.

Illustrated Example of an Inverter Model

This example is a mixed-signal model of an inverter. The input is a bit signal that is converted to a 5V or 0V. This value is then given to the input of the cmos inverter and the output of the

cmos inverter is then given to an analog to digital converter. The final output is a bit signal that is the inverse of the input bit signal.



This inverter can be modeled with the following entity and architecture.

```
entity inverter is
    port(inv_inp : in  bit;
         inv_op  : out bit);
end entity inverter;

architecture inv_behav of inverter is
    terminal tin, tout, tdd : electrical;
    quantity vdd across idd through tdd;
    quantity vin across iin through tin;
    quantity vout across tout;
    quantity vres across ires through tout;
    signal vsig : real := 0.0;
    constant power : real := 5.0;
    constant r : real := 1.0;
    component nmos is
        port (terminal g, s, d : electrical);
    end component;
    for all : nmos use entity work.nmos(nmos_behav);
    component pmos is
        port (terminal g, s, d : electrical);
    end component;
    for all : pmos use entity work.pmos(pmos_behav);
begin
    vres == r * ires;
    vdd == power;
    vin == vsig'slew(1.0,-2.0);
    break on vsig;
    nm : nmos port map(tin, ELECTRICAL_REF, tout);
    pm : pmos port map(tin, tdd, tout);
    d2a: process(inv_inp)
    begin
        if (inv_inp = '0') then
            vsig <= 0.0;
        else
            vsig <= power;
        end if;
    end process;
```

```
    a2d : process (vout'above(power/2.0))
    begin
        if(vout'above(power/2.0) = true) then
    inv_op <= '1';
    else
         inv_op <= '0';
    end if;
    end process;
end architecture inv_behav;  ---- of inverter
```

The interfaces for the nmos and pmos components are declared in the architecture description `inv_behav`. In this case the architecture is written in the dataflow style because the design uses signal ports with direction. The syntax for declaring and instantiating components are covered in more detail in Component on page 48.

The resistor is modeled with a simple simultaneous statement. (For more information, see "Simultaneous Statements" on page 54.)

```
vres == r * ires;
```

The `d2a` and the `a2d` are modeled as processes. The `d2a` process assigns the `vsig` signal a value of 0.0 if the input is 0, otherwise it assigns a value of 5.0 (`power`). The following simple simultaneous statement specifies that the `vin` across quantity follows the value of `vsig` but with a rising slope of 1 and a falling slope of -2.:

```
vin == vsig'slew(1.0,-2.0);
```

The `vdd` across quantity has a constant value of 5.0 (`power`):

```
vdd == power;
```

The `tin` terminal is connected to the gate port and the `tout` terminal is connected to the drain ports on the `nmos` and `pmos` instances. The `nmos` source port is tied to ground (the reference terminal) and the `pmos` source port is tied to the `tdd` terminal:

```
nm : nmos port map(tin, ELECTRICAL_REF, tout);
pm : pmos port map(tin, tdd, tout);
```

The `a2d` process converts the analog value back to digital. If the value of `vout` is greater than 2.5 (`power/2`) then `inv_op` is assigned a value of 1. Otherwise `inv_op` is assigned a value of 0. The expression `vout'ABOVE(power/2.0)` tests whether `vout` is greater than 2.5.

4

VHDL-AMS Language Elements

A VHDL-AMS design consists of any number of design files (ASCII text files) containing descriptions of the basic design units. There is no fixed relationship between design files and design units: a single design file can contain one or more design units. You cannot split a design unit across more than one design file.

The VHDL-AMS language is not case sensitive, so you can freely mix upper and lower case characters (apart from when they are used for character literals).

For an overview of the main features of the VHDL-AMS language you can use with the Virtuoso AMS Designer simulator, see

- Entity and Architecture on page 34
- Packages and Libraries on page 36
- Declarations on page 37
- Statements on page 51
- Expressions on page 55

For detailed information on these topics, see the VHDL-AMS LRM.

Entity and Architecture

Design units are divided into two categories: design entities and architectures. A combination of one of each of these two types forms the basic hardware unit of a VHDL-AMS description. You can have only one entity of a given name but you can pair that entity with different architectures, as needed. Each alternative architecture can describe a different implementation, but the interface, as defined by the entity, is always the same.

Entities

An entity declaration defines the interface between a given design entity and the environment in which it is used. It can also specify declarations and statements that are part of the design entity. An entity declaration can be shared by multiple design entities, each with a different architecture. An entity declaration can potentially represent a class of design entities, each with the same interface.

```
entity_declaration ::=
    entity identifier is
        [generic (generic_interface_list);]
        [port (port_interface_list);]
        {entity_declarative_part}
    end [entity][identifier];
```

Architectures

An architecture body defines the body of a design entity. It specifies the relationships between the inputs, outputs, quantities, and terminals of a design entity and can be expressed in terms of structure, dataflow, equations, or behavior. Such specifications can be partial or complete.

```
architecture_body ::=
    architecture identifier of entity_name is
        {block_declarative_part}
    begin
        [concurrent_statement_part}
    end [ architecture ] [ architecture_simple_name] ;

architectural_statement ::=
    simultaneous_statement | concurrent_statement
```

The `identifier` defines the simple name of the architecture body; distinguishing architecture bodies associated with the same entity declaration. The `entity name` identifies the name of the entity declaration.

Both the entity declaration and architecture body must reside in the same library.

Multiple implementations of One Interface

VHDL-AMS supports the use of multiple architectures for a given entity. This is useful, for example, in the context of logic synthesis because the output of synthesis tools is often a VHDL-AMS gate-level netlist that has the same entity declaration as the original RTL description:

```
RTL Description --> work.entityname(behaviour)
Gate-level net-list --> work.entityname(post_synth)
```

If the component `entityname` is instantiated within a design, you use a VHDL-AMS configuration to specify which of the two alternative architectures to load prior to simulation. There are two supported methods of specifying configurations: configuration specifications (an architecture declarative item), and configuration declaration (a primary library unit).

Configuration specification allows a component instance (or instances) to be bound to a specific entity/architecture pair by specifying the binding within the architecture that instantiates the design entities.

Packages and Libraries

Packages and libraries are used to group and organize the information used in designs.

Packages

Packages are used to group multiple elements under a single name. Packages might group related declarations into a separate, independent, and reusable set of subprograms that provides operations on a particular type of data, or might group the declarations needed to model a design.

Packages can also be used to separate the implementation of items from the external view of the items. In this case, a package declaration specifies the external view and the implementation is defined in the body of the package.

```
package_declaration ::=
    package identifier is
        {package_declarative_item}
    end [package][identifier];
```

The package declaration is a primary design unit and as such can be analyzed and placed in the working library (or in a different library, if required). The declarations contained in a package (or in a library) can be made available to a design entity by using a `use` clause.

```
use_clause ::=
    use selected_name {, selected_name };
```

To make all the declarations contained within the package visible, use the reserved word `all` as the suffix of the identifier. For example, the following statement makes visible all the declarations in the `fastpak` package.

```
use fastpak.all
```

Libraries

Libraries are used to store packages and design units. Analyzed designs are typically stored in a working library, named `work`. Items in the `work` library can be accessed simply by using a `use` clause, because all design entities have access to the `work` library by default.

Library units can also be stored in other libraries, referred to as resource libraries. To access the items in a resource library, you insert a library clause immediately before the design unit that accesses the resource libraries.

```
library_clause ::=
    library identifier {, ...};
```

Declarations

A declaration introduces the name of an object, defines its type, and can provide an initial value. See the following topics for descriptions of specific declarations:

Natures

A nature declaration declares a nature and defines the across and through types of the nature.

```
nature_declaration ::=
    nature identifer is nature_definition ;
nature_definition ::=
    scalar_nature_definition | composite_nature_definition
subnature_declaration ::=
    subnature identifier is  subnature_indication;
nature_mark ::=
    nature_name | subnature_name
subnature_indication ::=
    nature_mark [index_constraint]
```

Types

You can declare the following VHDL-AMS types:

- Scalar Types on page 38
- Composite Types on page 39
- Access Types on page 39
- File Types on page 39
- Scalar Nature on page 40

Scalar Types

Scalar types consist of integer types, enumeration types, physical types, and floating point types.

Integer types have whole number values.

```
integer_type_definition ::=
    range simple_expression (to | downto) simple_expression
```

Enumeration types allow for setting names for encoded values of some signals.

```
enumeration_type_definition ::=
    ((identifier | character_literal){,...})
```

Physical types represent physical properties like mass, length, and time.

```
physical_type_definition ::=
    range simple_expression (to | downto) simple_expression
        units
```

```
            identifier ;
            {identifier = physical_literal ;}
        end units [identifier]

physical_literal ::=
    [decimal_literal | based_literal] unit_name
```

Floating point types represent real numbers.

```
floating_type_definition ::=
    range simple_expression (to | downto) simple_expression
```

Composite Types

Terminal objects of an array nature consist of identical elements associated with some index value. The branch types defined by an array nature definition are array types.

```
array_nature_definition ::=
    unconstrained_nature_definition | constrained_nature_definition

unconstrained_nature_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
        of subnature_indication

constrained_nature_definition ::=
    array index_constraint of subnature_indication
```

A record nature is a composite nature where terminal objects of a record nature consist of named terminals. The branch types defined by a record nature definition are record types.

```
record_nature_definition ::=
    record
         nature_element_declaration
         {nature_element_declaration}
    end record [record_nature_simple_name]

nature_element_declaration ::=
    identifier_list: element_subnature_definition

element_subnature_definition ::=
    subnature_indication
```

Access Types

Access types link data objects, similar to pointer types found in programming languages.

```
access_type_definition ::=
    access subtype_indication
```

File Types

File types define objects representing files in the host system environment.

```
file_type_definition ::=
    file of type_mark
```

The `type_mark` defines the subtype of the values contained in the file.

Scalar Nature

A scalar nature declaration defines a scalar nature and its branch types, and declares the name of the reference terminal that is of the nature.

```
scalar_nature_definition ::=
    type_mark across
    type_mark through
    identifier reference
```

The type marks must denote floating-point types. The simple nature of a scalar nature is the nature itself.

Objects and Interface Objects

An *object* is a named item that has a value of a specified type. In the case of a terminal, it has a specified nature. See the following topics for more information:

Constant

You use a constant to provide a name and an explicitly defined type for a value. You must declare a constant before you can use it in a model.

```
constant_declaration ::=
    constant identifier {,...}:subtype_indication [:=expression];
```

You can declare constants that are real, integer, and boolean types.

If you are using the Spectre or UltraSim solver with the simulation front end SFE parser, you can also declare constants that are arrays of these types and use these arrays in analog simultaneous statements. For example, you can declare the following:

```
constant const_array: Real_vector := ( 1.0, 2.0 );
```

Generic

A generic is a constant in the interface of an entity.

```
entity_declaration ::=
    entity identifier is
        [generic (generic_interface_list);]
        ...
    end [entity][identifier];
```

A generic interface list that is accessed from an analog context (such as in a simultaneous statement) can consist of scalar objects or arrays of real, integer, or boolean objects.

```
generic_interface_list ::=
    (identifier {,...}:subtype_indication[:=expression])
    {;...}
```

For example, if you are using the Spectre or UltraSim solver with the simulation front end (SFE) parser, you can declare a generic in the following way and use it in an analog simultaneous statement:

```
entity child is
    generic ( genericArg : real_vector(0 to 1) := (0.0, 0.0) );
    ...
end child;
```

Generic declarations specify parameters you can change when you instantiate an architecture in a design.

For each generic, you can specify a type and an optional default value. You can define a type to have a valid range of values.

The following example illustrates how to declare a generic in an entity. The `resistor` entity declares `r`, which represents the resistance value, as a generic. Terminal ports `tr1` and `tr2` represent circuit nodes whose nature is `electrical`. The `rbehavior` architecture declares `Vr` as an analog branch quantity representing the branch voltage across `tr1` and `tr2`.

```
entity resistor is
    generic(r : real := 1.0 );
    port(terminal tr1, tr2 : electrical);
end resistor;

architecture rbehavior of resistor is
    quantity Vr across Ir through tr1 to tr2;
begin
    Vr == Ir*r;
end architecture rbehavior;
```

Variable

Variables must be declared before they can be used in a model. Variables are data objects and serve for local storage of temporary data. Non-shared variables can occur in sequential environment only, i.e., inside a `PROCESS` statement or a subprogram. Shared variables can occur in any statement and can be accessed by more than one process.

```
variable_declaration ::=
    [shared] variable identifier {,...}:subtype_indication [:=expression];
```

Signal

Internal signals in an architecture body are defined using signal declarations. Signals represent interconnection wires.

```
signal_declaration ::=
    signal identifier {,...} : subtype_indication [:=expression];
```

Attribute

The attributes of signals can be used to find information about their history of transactions and events. The table below lists the predefined attributes that Cadence supports. Additional information on these attributes can be found in LRM 14.1. In the following table, `T` = scalar type, subtype, or terminal; `A` = array type; `S` = signal; `E` = named entity; `N` = nature; and `Q` = quantity.

Array quantities cannot be used in attributes.

Attribute	Value Returned
`A'ascending[(N)]`	(Digital only): `TRUE` if the Nth index range of `A` is defined with an ascending range; `FALSE` otherwise.
`A'high[(N)]`	(Digital only): Upper bound of the Nth index range of `A`.
`A'left[(N)]`	(Digital only): Left bound of the Nth index range of `A`.
`A'length[(N)]`	(Digital only): Number of values in the Nth index range.
`A'low[(N)]`	(Digital only): Lower bound of the Nth index range of `A`.
`A'range[(N)]`	(Digital only): The range `A'left(N) to A'right(N)` if the Nth index range of `A` is ascending; otherwise the range `A'left(N) downto A'right(N)`.
`A'reverse_range[(N)]`	(Digital only): The range `A'right(N) downto A'left(N)` if the Nth index range of `A` is ascending; otherwise the range `A'right(N) to A'left(N)`.
`A'right[(N)]`	(Digital only): Right bound of the Nth index range of `A`.
`E'instance_name`	(Digital only): A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, *including* the names of instantiated design entities.

`E'path_name`	(Digital only): A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, *excluding* the names of instantiated design entities.
`E'simple_name`	(Digital only): The simple name, character literal, or operator symbol of the named entity.
`Q'above(E)`	`TRUE` if `Q - E` is sufficiently larger than 0.0, `FALSE` if `Q - E` is sufficiently smaller than 0.0, `Q'above(E)'delayed` otherwise.
`Q'delayed[(T)]`	A quantity equal to `Q` delayed by `T`.
`Q'dot`	The derivative with respect to time of `Q` at the time when the software evaluates the attribute.
`Q'dot'dot`	The second derivative with respect to time of `Q` at the time when the software evaluates the attribute.
`Q'integ`	The time integral of `Q` from time 0 to the time when the software evaluates the attribute.
`Q'ltf(NUM,DEN)`	The Laplace transfer function of each scalar subelement of `Q` with `NUM` as the numerator and `DEN` as the denominator polynomials. `Num` and `Den` must be aggregates of constants or scalar references. For examples of use, see "Examples of Attribute Use: Q'ltf" on page 46.
`Q'ramp[(MAX_T_RISE [,MAX_T_FALL])]`	A quantity where each scalar subelement follows the corresponding scalar subelement of `Q` linearly with the specified rise and fall times.
`Q'slew[(MAX_RISING_SLOPE [,MAX_FALLING_SLOPE])]`	A quantity where each scalar subelement follows the corresponding scalar subelement of `Q`, but its derivative with respect to time is limited by the specified slopes.
`S'delayed[(T)]`	A signal equivalent to signal `S` delayed `T` units of time.
`S'driving`	(Digital only): `FALSE`, if the prefix denotes a scalar signal and the current value of the driver for `S` in the current process is determined by the null transaction; `TRUE` otherwise.

`S'driving_value`	(Digital only): If `S` is a scalar signal, the current value of the driver for `S` in the current process. If `S` is a composite signal, the aggregate of the values of `R'driving_value` for each element `R` of `S`. If `S` is a null slice, a null slice.
`S'event`	`TRUE` or `FALSE`, depending on whether an event has just occurred on signal `S`.
`S'last_active`	The amount of time that has elapsed since the last time at which signal `S` was active.
`S'last_event`	The amount of time that has elapsed since the last event occurred on signal `S`.
`S'last_value`	The previous value of `S`, immediately before the last change of `S`.
`S'quiet[(T)]`	A signal that has the value `TRUE` when the universal time of the last transaction on `S` is less than the universal time correspond to `NOW - T`, and the value `FALSE` otherwise.
`S'ramp[(TRISE[,TFALL])]`	A quantity where each scalar subelement follows the corresponding scalar subelement of `S`. If the value of any parameter is greater than 0.0, the corresponding value change is linear from the current value of the scalar subelement of `S` to its new value, whenever that subelement has an event. `S` can also be a signal array.
`S'slew[(RISING_SLOPE [,FALLING_SLOPE])]`	A quantity where each scalar subelement follows the corresponding scalar subelement of `S`. If the value of `RISING_SLOPE` is less than `REAL'HIGH`, or if the value of `FALLING_SLOPE` is greater than `REAL'LOW`, the corresponding value change is linear from the current value of the scalar subelement of `S` to its new value, whenever that subelement has an event. `S` can also be a signal array.
`S'stable[(T)]`	A signal that has the value `TRUE` when the universal time of the last event on `S` is less than the universal time corresponding to `NOW - T`, and the value `FALSE` otherwise.

`S'transaction`	A signal whose value toggles to the inverse of its previous value in each simulation cycle in which signal `S` becomes active.
`T'ascending`	`TRUE` if `T` is defined with an ascending range; `FALSE` otherwise.
`T'base`	The base type of `T`.
`T'high`	The upper bound of `T`.
`T'left`	The left bound of `T`.
`T'leftof(X)`	The value that is to the left of the parameter in the range of `T`.
`T'low`	The lower bound of `T`.
`T'pos(X)`	The position number of the value of the parameter.
`T'pred(X)`	The value whose position number is one less than that of the parameter.
`T'reference`	The across quantity whose plus terminal is `T` and whose minus terminal is the reference terminal of the nature of `T`.
`T'right`	The right bound of `T`.
`T'rightof(X)`	The value that is to the right of the parameter in the range of `T`.
`T'succ(X)`	The value whose position number is one greater than that of the parameter.
`T'val(X)`	The value whose position number is the universal_integer value corresponding to `X`.

Examples of Attribute Use: Q'ltf

The parameters to the `Q'ltf` attribute must be constants or aggregates of scalar references. For example, given the following declarations:

```
entity lpf_1 is
  generic ( fp : real := 1.0;
            gain : real := 1.0 );
  port ( quantity input : in real;
         quantity output : out real);
end entity lpf_1;

library ieee;
use ieee.math_real.all;
```

```
architecture simple of lpf_1 is
  constant wp : real := math_2_pi*fp;
  constant wpGain : real := wp * gain;
  constant num : real_vector := (0 => wp * gain);
  constant den : real_vector := (wp, 1.0);
begin
  ...
end architecture simple;
```

Then the following `'ltf` statements, when they appear in the body of the architecture, are valid or invalid as described in the comments:

```
output == input'ltf(num, den);  // Invalid, parameters are array references.
                                // The parameters must be aggregates.

output == input'ltf((0=>num[0]), (den[0], den[1]));
                                // Invalid, parameters are aggregates of
                                // array references, not scalar references.

output == input'ltf((0=>wp*gain), (wp, 1.0));
                                // Invalid, first parameter aggregates
                                // contains an expression.

output == input'ltf((0=>wpGain, (wp, 1.0));
                                // Valid, parameter aggregates are scalar
                                // references and constants.
```

Terminal

A terminal declaration declares a terminal, reference quantity, and contribution quantity of the terminal.

```
terminal_declaration ::=
    terminal identifier_list : subnature_indication ;
```

A terminal represents a physical connection point in the system.

Although Cadence VHDL-AMS does not support a general ability to create aliases, you can declare aliases to terminals used for GROUND connections.

Quantity

A quantity declaration declares a quantity.

```
quantity_declaration ::=
    free_quantity_declaration | branch_quantity_declaration

free_quantity_declaration ::=
    quantity identifier_list : subtype_indication [:= expression];

branch_quantity_declaration ::=
    quantity [across_aspect] [through_aspect] terminal_aspect ;

across_aspect ::=
    identifier_list [:= expression] across
```

```
through_aspect ::=
    identifier_list [:= expression] through
terminal_aspect ::=
    plus_terminal_name  [ to minus_terminal_name ]
```

Across quantities specify the voltage across two terminals. Through quantities define the current through a terminal. Free quantity values are determined solely by the specified equations and are not constrained by any terminal.

Branch quantity arrays are supported, so you can declare branch quantities referencing bits, slices, or a whole terminal arrays and use the quantity arrays in VHDL-AMS behavioral statements.

Component

To instantiate a design entity within another design entity, the lower level design entity is declared in the form of a component. The component declaration is placed in the architecture declaration of the high level design entity, creating a socket into which a design entity is placed. The component declarations indicate that a lower-level design entity is used. They do not specify which entity/architecture pair is plugged into the socket.

Specifying an entity name and a corresponding architecture body name binds the instance. Different instances of a given component can be bound to different entity./architecture pairs. The keyword `all` can also be used if all instances of a particular component type have the same binding.

```
component_declaration ::=
    component identifier [ is ]
        [ generic (generic_interface_list);]
        [ port (port_interface_list); ]
    end component [identifier] ;
```

It is common practice to use the same name and ports for the component and the entity to which it is bound.

The component instantiation statement specifies how the component is used in the design.

```
component_instantiation_statement ::=
    instantiation_label:
        [component](component_name)
        [generic map (generic_association_list)]
        [port map (port_association_list)];
```

The `generic map` arguments are expressions that map a particular value to a particular constant or constant array for an instance. You can use positional or named associations or a combination of both to specify the mapping.

If you are using the Spectre or UltraSim solver with the simulation front end (SFE) parser, you can specify constant or generic arrays on the component instantiation:

```
i1 : entity work.bottom
        generic map ((5.0, 2.0*g1), (2.0, g1-g2))
```

Note: You must specify the entire array. You cannot specify an index or a slice.

A `port map` statement maps ports of an entity to terminals, signals, or quantities in an architecture body. You can provide a value for the expression or you can unassociate the port using the `open` keyword. You can use positional or named associations or a combination of both to specify the mapping.

If you are using the Spectre or UltraSim solver with the simulation front end (SFE) parser, you can specify constant or generic arrays in a `port map` statement as follows:

```
port map ( formalArg => actualArg )
```

where the *`formalArg`* and the *`actualArg`* can be an array name representing an entire array or a scalar value. You must not specify an index or slice expression for the *`formalArg`*, such as *`formalArg`*`(0)` or *`formalArg`*`(0 to 1)`. The *`actualArg`* can also be any of the following:

- An array with an index expression

  Note: The index expression can contain integers and enumerated expressions.

- An array with a slice expression such as
  - an ascending or descending range where the slice is passed down to a VHDL child
  - an ascending range in a VHDL parent where the slice is passed down to a Verilog child
  - an ascending range in a Verilog parent where the slice is passed down to a VHDL child
  - an ascending range with an enumerated index, such as `arg(enum1 to enum2)`, where the slice is passed down to a VHDL child
  - an integer or enumerated locally static or globally static expression as either the left or the right value of the slice, such as

    ```
    port map ( formalArg => actualArg((g1+2)/2-2+3 to (g1/2+1)*2) )
    ```

Note: You must not use VHDL predefined index attributes, `'left` or `'right`, in an index or slice expression for *`actualArg`*.

Interface Declarations (Ports)

The ports of a block are defined by a port interface list. Interface elements in the port interface list declare a formal port. A formal port can be a signal port, a quantity port, or a terminal port.

Interface objects also include interface terminals and interface quantities that appear as ports of a design entity, component, or block.

```
interface_declaration ::=
    interface_terminal_declaration
    interface_quantity_declaration
    interface_signal_declaration

interface_terminal declaration ::=
    terminal identifier_list : subnature_indication

interface_quantity_declaration ::=
    quantity identifier_list : [in | out] subtype_indication [:=
    static_expression]

interface_signal_declaration ::=
    signal identifier_list : [in | out] subtype_indication [:=
    static_expression]
```

Subprograms

The subprogram facility allows for the division of complex behavioral models into self-contained sections. There are two kinds of subprograms: procedures and functions. A procedure summarizes a collection of sequential statements that are executed for their effect, and is therefore a generalization of a statement. A function summarizes a collection of sequential statements that are executed for their result, and is therefore a generalization of an expression.

Procedures

When using a procedure in a model, it must first be declared. Declarations can include types, subtypes, constants, variables, and nested subprogram declarations.

```
subprogram_body ::=
    procedure identifier [(parameter_inteface_list)] is
        {subprogram_declarative_part}
    begin
        {sequential_statement}
    end [procedure][indentifier]
```

To use a procedure, code a `call` statement.

```
procedure_call_statement ::=
    [label:] procedure_name;
```

Functions

A function is a way of defining a new operation using an expression. A collection of sequential statements are written that calculate the result.

```
subprogram_body ::=
    [pure|impure]
    function identifier [(parameter_inteface_list)] return type_mark is
        {subprogram_declarative_item}
    begin
        {sequential_statement}
    end [function][indentifier]
```

Unlike a procedure call, a function call is part of an expression, not a sequential statement on its own.

```
function_call ::=
    function_name [(parameter_association_list)]
```

Statements

The three types of statements are sequential, concurrent, and simultaneous:

- Sequential statements allow selection between alternative courses of action, repetitive action, and are executed in sequence.
- Concurrent statements describe the operation of a module.
- Simultaneous statements express the equations that govern the behavior of a model.

Sequential Statements

Sequential statements define algorithms for the execution of a subprogram or process; they execute in the order they appear.

Sequential Break Statement

A sequential break statement is included in a process.

```
break_statement ::=
    [label:] break [break_element {, ...}][when boolean_expression];
break_element ::=
    [for quantity_name use] quantity_name => expression
```

Signal Assignment

Assignment of a signal statement provides a new value of a signal. A delay mechanism can be specified that determines when a new value should be applied. An event occurs on a signal when a new value is different than the old value it replaces.

```
signal_assignment_statement ::=
    [label:] name <=[delay_menchanism] waveform;
```

```
waveform ::=
    (value_expression [after time_empression])(,...)
```

Variable Assignment

The value of a variable can be modified once assigned by an assignment statement.

```
variable_assignment_statement ::=
    [label :] name := expression ;
```

Concurrent Statements

Concurrent statements define interconnected blocks and processes that jointly with simultaneous statements describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other. The concurrent statement is declared using the syntax for architecture bodies:

```
achitecture_body ::=
    architecture identifier of entity_name is
        {block_declarative_item}
    begin
        {concurrent_statement}
    end [architecture][identifier];
```

Block Statement

The primary concurrent statements are the block statement, which groups together other architecture statements, and the process statement, which represents a single independent sequential process. A block statement defines an internal block representing a portion of a design. Blocks can be hierarchically nested to support design decomposition.

```
block_statement ::=
    block_label :
        block [ ( guard_expression ) ] [ is ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;
```

Process

A process statement defines an independent sequential process representing the behavior of some portion of the design.

```
process_statement ::=
    [ process_label : ]
        [ postponed ] process [ ( sensitivity_list ) ] [ is ]
            process_declarative_part
        begin
```

```
        process_statement_part
    end [ postponed ] process [ process_label ] ;
```

Component Instantiation

A component instantiation is a concurrent statement for writing a structural implementation. The component instantiation performs direct instantiation of an entity.

```
component_instantiation_statement ::=
    instantiation_label:
        entity entity_name [(architecture_identifier)]
            [port map (port_association_list)];
```

Concurrent Break Statement

A concurrent break statement is used in mixed-signal systems when the operating conditions of an analog part are changed by the process representing a digital part. A discontinuity is indicated by using a break statement to signal the analog solver that a discontinuity has occurred at the current simulation time.

```
concurrent_break_statement ::=
    [label:]
    break [break_element {, ...}]
        [on signal_name {, ...}]
        [when boolean_expression];

break_element ::=
    [for quantity_name use] quantity_name => expression
```

Procedure

The concurrent procedure call statement is a process whose body contains a sequential procedure call statement.

```
concurrent_procedure_call_statement ::=
    [label:] procedure_name [(parameter_association_list)];
```

Concurrent Assert

A concurrent assertion statement is used to represent a process whose body contains an ordinary sequential assertion statement.

```
concurrent_assertion_statement ::=
    [label:]
    assert boolean_expression
        [report expression][severity expression]
```

Simultaneous Statements

Simultaneous statements describe the analog behavior of a model by expressing explicit differential and algebraic equations. Together with implicit equations, these explicit equations constrain the values of quantities in a model.

```
architecture_body ::=
    architecture identifier of entity_name is
        {block_declarative_item}
    begin
        {concurrent_statement | simultaneous_statement}
    end [architecture][identifier]
```

See also "Simultaneous Statements" on page 21.

Simple

A simple simultaneous statement specifies zero or more characteristic expressions.

```
simple_simultaneous_statement ::=
    [label :] expression == expression ;`
```

If

A simultaneous if statement selects for evaluation one of the enclosed simultaneous statement parts depending on the value of one or more conditions.

```
simultaneous_if_statement ::=
     [if_label :]
        if condition use
            simultaneous_statement_part
        {elsif condition use
            simultaneous_statement_part}
        [else
             simultaneous_statement_part]
        end use  [if_label];
```

Case

A simultaneous case statement selects for evaluation one of a number of alternative simultaneous statement parts; the chosen alternative is defined by the value of an expression.

```
simultaneous_case_statement ::=
    [case_label:]
     case expression use
         simultaneous_alternative
         {simultaneous_alternative}
     end use [case_label];
```

```
simultaneous_alternative ::=
     when choices =>
        simultaneous_statement_part
```

Expressions

Predefined Operators and Operator Precedence

Operators belong to classes that have the same precedence level; in the following table the classes of operators are listed in order of increasing precedence.

logical_operator	::=	**and**	|	**or**	|	**nand**	|	**nor**	|	**xor**	| **xnor**
relational_operator	::=	=	|	/=	|	<	|	<=	|	>	| >=
shift_operator	::=	**sll**	|	**srl**	|	**sla**	|	**sra**	|	**rol**	| **ror**
adding_operator	::=	+	|	–	|	&					
sign	::=	+	|	–							
multiplying_operator	::=	*	|	/	|	**mod**	|	**rem**			
miscellaneous_operator	::=	**	|	**abs**	|	**not**					

Operators of higher precedence are associated with their operands before operators of lower precedence. When a sequence of operators is allowed, operators with the same precedence level are associated with their operands in textual order, from left to right. The precedence of an operator is fixed and cannot be changed. Parentheses can be used to control the association of operators and operands.

Static and Non- Static Expressions

Locally static expressions are those expressions that can be determined when a design unit is compiled. A globally static expression can be determined at the time of elaboration. Non-static expressions can change during simulation.

5

Mixed-Signal Value Conversions

The analog extensions of the VHDL-AMS language introduce some complexities not present in a digital only language. Because values in the analog domain are continuous and values in the digital domain are discrete, converting across domains now involves issues of both the translation of values and the timing of the exchange.

In the digital domain, it is the signals and variables that contain dynamic data values. These objects are assigned values in processes and change at discrete times. In the analog domain, the terminals and quantities are the objects containing values. The values of these objects are continuous functions with respect to time. These differences mean that when you model a mixed signal system, you must consider the mechanism for exchanging values between digital and analog.

The method you use to exchange values across domains depends on whether the value is being converted from an analog value to a digital value or from a digital value to an analog value.

Analog to Digital Conversion

You can, from a digital context, sample an analog value at various time points and convert the sampled value to a digital value, or you can monitor an analog value and use the value to trigger an event based on a threshold crossing. The following sections illustrate these approaches.

Sampling Analog Values

You can directly use the value of an analog-valued quantity in an assignment statement of a digital process. For example, the following model of a flip-flop uses a clock to determine when to sample the quantity `q`. The value of the quantity `q` then determines the new digital value of `out`.

```
library ieee;  use ieee.math_real.all;
library ncvhdl_lib;

ENTITY sampler IS
```

```
END sampler;

ARCHITECTURE behavior_sampler OF sampler IS
    signal clk : bit;
    signal output : bit;
    QUANTITY q   : real;

    -- Source constants
    constant thresh : real := 3.4;
    constant amplitude : REAL := 2.0 * thresh;
    constant frequency : REAL := 0.16 * 0.1e9  ;

BEGIN

    clock: process(clk)
    BEGIN
        clk <= not clk after 10 ns;
    END PROCESS clock;

    flipflop: PROCESS (clk)
    BEGIN
        if (clk = '1') then
            if (q > thresh) then
                output <= '1';
            else
                output <= '0';
            end if;
        end if;
    END PROCESS flipflop;

    -- sin source
    q == amplitude * sin(2.0 * MATH_PI * frequency * now);

END behavior_sampler;
```

Using Analog Values to Trigger a Digital Event

You can also use the `'above` attribute to trigger a digital event based on analog changes. The expression `Q'above(E)` represents a boolean-valued signal that is true when the value of `Q` is greater than the value of the expression `E` and is false otherwise. You can use the `Q'above(E)` signal in a process sensitivity list or in a `wait` statement to trigger the execution of a digital process.

For example, in the following model of an inverter the statement

```
wait on v_in'above(threshold)
```

ensures that the value of `output` changes only when the value of the analog signal crosses the `threshold` value.

```
inverter: process is
    constant threshold : real := 2.5;
begin
    if (v_in > threshold) then
        output <= '0';
    else
        output <= '1';
    end if;
```

```
    wait on v_in'above(threshold);
end process inverter;
```

Digital to Analog Conversion

Simultaneous statements can directly access digital signals. For example, if you have a bit-valued signal `S` and an across quantity `Q` you can legally use a simultaneous statement like the following:

```
Q == real(S) * maxval;
```

Unfortunately, the above statement introduces a discontinuity into quantity `Q` whenever signal `S` changes value. As a result, you need to use the `S'ramp` or `S'slew` attributes to smooth the transitions. Avoiding discontinuities allows the simulator to run much more efficiently.

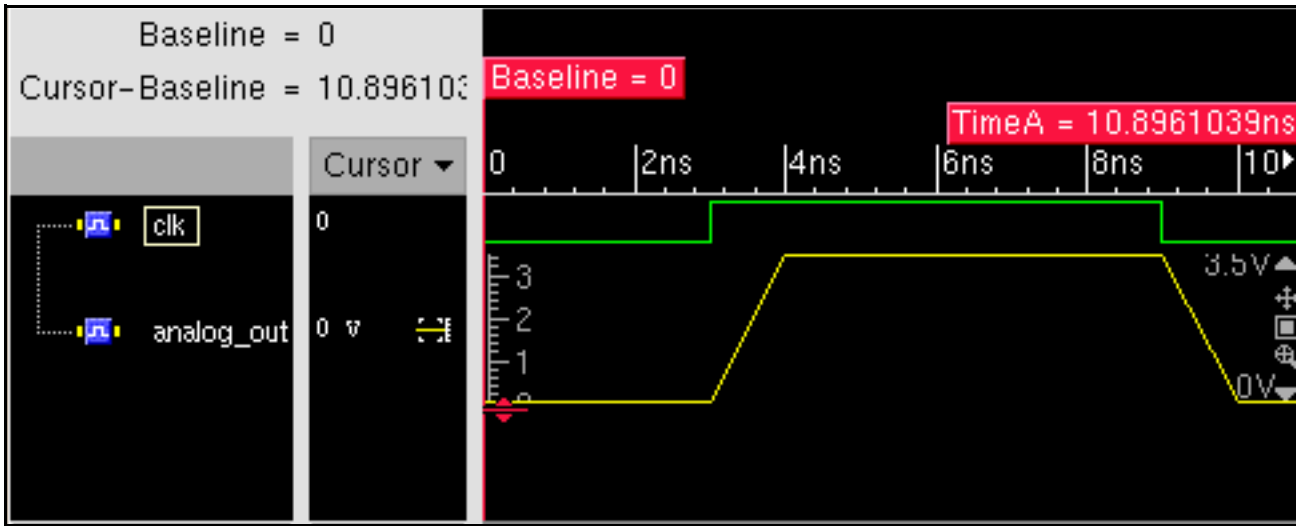
For example, the following example converts the digital signal `clk` to the analog quantity `analog_out`, and uses the `s'ramp` attribute to smooth the output.

```
library ieee;  use ieee.math_real.all;
library ncvhdl_lib;
use ncvhdl_lib.std_decls.all;

entity v_source is
end entity v_source;

architecture behavior of v_source is
    constant maxv : real := 3.5;
    signal rclk : real := 0.0;
    signal clk : bit := '0';
    terminal t1: electrical;
    quantity analog_out across analog_curr through t1;
begin
    clock: process is
    begin
        if (clk = '0') then
            wait for 3 ns;
            rclk <= maxv;
            clk <= '1';
        else
            wait for 3 ns;
            rclk <= 0.0;
            clk <= '0';
        end if;
    end process clock;
    analog_out == rclk'ramp(1.0e-9);
end architecture behavior;
```

Notice, in the following figure, how using the `S'ramp` attribute results in a ramp, rather than a step function, on the `analog_out` signal.



Break Statement

Signal or variable changes in a process sometimes cause discontinuities in the analog part of a design. If this happens in your design, you need to ensure that your digital code runs a `break` statement whenever a process introduces a discontinuity.

There are two types of `break` statements: concurrent and sequential.

The following example illustrates how to use a concurrent `break` statement. When the switch turns on or off (that is when `sw` changes value) the conversion process introduces a discontinuity into the `v_out` signal. To announce the discontinuity, the concurrent break statement runs whenever there is an event on the signal `sw`.

```
library ieee;  use ieee.math_real.all;
library ncvhdl_lib;
use ncvhdl_lib.std_decls.all;

entity v_source is
end entity v_source;

architecture behavior of v_source is
    signal sw : bit := '0';
    quantity v_out : real := 0.0;
begin
    if (sw = '0') use
        v_out == 0.0;
    else
        v_out == 5.0;
    end use;
    toggle: process is
    begin
```

```
        sw <= not sw after 5 ns;
        wait on sw;
    end process toggle;
    break on sw;
end architecture behavior;
```

A

Standard Packages Supported

This chapter includes information on:

- The IEEE libraries that are included in the installation. See "IEEE Libraries for VHDL-AMS" on page 64.
- Support for the IEEE standard VHDL mathematical packages. See "IEEE Standard VHDL Mathematical Packages" on page 64

IEEE Libraries for VHDL-AMS

The software supplied by Cadence includes two libraries designed for use with VHDL-AMS. The first library is a superset of the default IEEE library. The second library is a superset of the IEEE (pure) library. Both libraries include the VHDLAMS specific packages:

```
package IEEE.ELECTRICAL_SYSTEMS
package IEEE.ENERGY_SYSTEMS
package IEEE.FLUIDIC_SYSTEMS
package IEEE.FUNDAMENTAL_CONSTANTS
package IEEE.MATERIAL_CONSTANTS
package IEEE.MECHANICAL_SYSTEMS
package IEEE.RADIANT_SYSTEMS
package IEEE.THERMAL_SYSTEMS
```

Note: These VHDLAMS packages are in draft form. They are not yet official IEEE standards.

To use the VHDL-AMS superset of the IEEE library, include the following lines in your `cds.lib` files.

```
SOFTINCLUDE $AMSHOME/tools/xcelium/files/cds.lib
UNDEFINE ieee
DEFINE ieee $AMSHOME/tools/xcelium/files/IEEE_vhdlams/IEEE
```

To use the VHDL-AMS superset of the IEEE (pure) library, include the following lines in your `cds.lib` files.

```
SOFTINCLUDE $AMSHOME/tools/xcelium/files/cds.lib
UNDEFINE ieee
DEFINE ieee $AMSHOME/tools/xcelium/files/IEEE_vhdlams_pure/IEEE
```

With this preparation, you can access the VHDL-AMS specific definitions that are included in the packages. For example, to access electrical natures in your design, you specify the following in your code:

```
library ieee;
use ieee.electrical_systems.all
```

For more information, see the VHDL-AMS package source located in

your_install_directory`/tools/xcelium/files/IEEE_vhdlams.src`

IEEE Standard VHDL Mathematical Packages

The AMS Designer simulator supports the VHDL mathematical packages defined in *IEEE Standard VHDL Mathematical Packages* (IEEE Std 1076.2-1996). This standard defines two package declarations: `MATH_REAL` and `MATH_COMPLEX`. The packages define a standard for designers to use in describing VHDL models that make use of common REAL or COMPLEX constants and common REAL or COMPLEX mathematical functions and operators.

Because of the IEEE copyright on these packages, Cadence cannot redistribute the source code. The source files (MATH_COM.vhd and MATH_REAL.vhd) that are shipped with the AMS Designer simulator have been stripped of all VHDL code.

The precompiled packages are included with the simulator. You can use the precompiled mathematical packages without the source code if you know which constants, functions, procedures, and operators are defined in the packages. The following sections tell you what the two packages contain.

Note: See the NC-VHDL Simulator Known Problems and Solutions document for a list of known issues in using the IEEE math libraries.

If, for some reason, you need the VHDL source code for the mathematical packages (for example, if you must recompile the packages because a library on which they depend has changed), you can get the source code, and a copy of the IEEE Std 1076.2-1996 standard, by contacting the IEEE directly at:

```
http://www.ieee.org
```

As explained in the previous section, two sets of IEEE packages are shipped with the simulator (the Synopsys packages and the “IEEE_pure” packages). However, unlike the other packages, the IEEE mathematical packages are compiled under both the IEEE and IEEE_pure directories. The AMS Designer simulator also supports the Synopsys version of the packages. To use the Synopsys version, you must first purchase it and then compile it into the IEEE library. Before you compile the source code, remove the following line from MATH_REAL.vhd:

```
attribute FOREIGN : string;
```

This line of code redefines the FOREIGN attribute, which is already defined in the STANDARD package. When the AMS Designer simulator detects the redefinition of the FOREIGN attribute, it hides both definitions of the attribute, in compliance with the VHDL LRM.

To avoid warning messages while compiling the Synopsys version, remove the dummy VHDL implementations of RAND, SRAND, and GET_RAND_MAX from MATH_REAL.vhd. Because these three functions are defined as FOREIGN functions, the AMS Designer simulator issues the following warning message when it sees the VHDL implementations of the three functions that have the FOREIGN attribute:

```
ncvhdl_p: *W,FATSPB : FOREIGN subprogram has body which will never be called.
```

If you are using the Synopsys version, you can use the “C_NATIVE” functions RAND, SRAND, and GET_RAND_MAX.

MATH_REAL

This section lists the constants, functions and procedures available in the `MATH_REAL` package.

Constants

The `MATH_REAL` package defines the following constants:

- MATH_E
- MATH_1_OVER_E
- MATH_PI
- MATH_2_PI
- MATH_1_OVER_PI
- MATH_PI_OVER_2
- MATH_PI_OVER_3
- MATH_PI_OVER_4
- MATH_3_PI_OVER_2
- MATH_LOG_OF_2
- MATH_LOG_OF_10
- MATH_LOG2_OF_E
- MATH_LOG10_OF_E
- MATH_SQRT_2
- MATH_1_OVER_SQRT_2
- MATH_SQRT_PI
- MATH_DEG_TO_RAD
- MATH_RAD_TO_DEG

Functions and Procedures

The `MATH_REAL` package contains the following functions and procedures:

- function SIGN (X : in REAL) return REAL;
- function CEIL (X : in REAL) return REAL;
- function FLOOR (X : in REAL) return REAL;
- function ROUND (X : in REAL) return REAL;
- function TRUNC (X : in REAL) return REAL;
- function "MOD" (X, Y : in REAL) return REAL;
- function REALMAX (X, Y : in REAL) return REAL;
- function REALMIN (X, Y : in REAL) return REAL;
- procedure UNIFORM (variable SEED1,SEED2:inout POSITIVE; variable X:out REAL);
- function SQRT (X : in REAL) return REAL;
- function CBRT (X : in REAL) return REAL;
- function "**" (X : in INTEGER; Y : in REAL) return REAL;
- function "**" (X : in REAL; Y : in REAL) return REAL;
- function EXP (X : in REAL) return REAL;
- function LOG (X : in REAL) return REAL;
- function LOG2 (X : in REAL) return REAL;
- function LOG10 (X : in REAL) return REAL;
- function LOG (X : in REAL; BASE : in REAL) return REAL;
- function SIN (X : in REAL) return REAL;
- function COS (X : in REAL) return REAL;
- function TAN (X : in REAL) return REAL;
- function ARCSIN (X : in REAL) return REAL;
- function ARCCOS (X : in REAL) return REAL;
- function ARCTAN (X : in REAL) return REAL;

- function ARCTAN (Y : in REAL; X : in REAL) return REAL;
- function SINH (X : in REAL) return REAL;
- function COSH (X : in REAL) return REAL;
- function TANH (X : in REAL) return REAL;
- function ARCSINH (X : in REAL) return REAL;
- function ARCCOSH (X : in REAL) return REAL;
- function ARCTANH (X : in REAL) return REAL;

MATH_COMPLEX

The `MATH_COMPLEX` package defines two complex types, constants, functions, and arithmetic operators.

Types

The types are defined as follows:

```
type COMPLEX is
   record
         RE: REAL;         -- Real part
         IM: REAL;         -- Imaginary part
   end record;
subtype POSITIVE_REAL is REAL range 0.0 to REAL'HIGH;
subtype PRINCIPAL_VALUE is REAL range -MATH_PI to MATH_PI;
type COMPLEX_POLAR is
   record
         MAG: POSITIVE_REAL;    -- Magnitude
         ARG: PRINCIPAL_VALUE;  -- Angle in radians; -MATH_PI is illegal
   end record;
```

Constants

- MATH_CBASE_1 defined as COMPLEX'(1.0, 0.0)
- MATH_CBASE_J defined as COMPLEX'(0.0, 1.0)
- MATH_CZERO defined as COMPLEX'(0.0, 0.0)

Functions and Operators

- function "=" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return BOOLEAN;

- function "/=" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return BOOLEAN;
- function CMPLX(X: in REAL; Y: in REAL:= 0.0) return COMPLEX;
- function GET_PRINCIPAL_VALUE(X: in REAL) return PRINCIPAL_VALUE;
- function COMPLEX_TO_POLAR(Z: in COMPLEX) return COMPLEX_POLAR;
- function POLAR_TO_COMPLEX(Z: in COMPLEX_POLAR) return COMPLEX;
- function "ABS"(Z: in COMPLEX) return POSITIVE_REAL;
- function "ABS"(Z: in COMPLEX_POLAR) return POSITIVE_REAL;
- function ARG(Z: in COMPLEX) return PRINCIPAL_VALUE;
- function ARG(Z: in COMPLEX_POLAR) return PRINCIPAL_VALUE;
- function "-" (Z: in COMPLEX) return COMPLEX;
- function "-" (Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function CONJ (Z: in COMPLEX) return COMPLEX;
- function CONJ (Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function SQRT(Z: in COMPLEX) return COMPLEX;
- function SQRT(Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function EXP(Z: in COMPLEX) return COMPLEX;
- function EXP(Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function LOG(Z: in COMPLEX) return COMPLEX;
- function LOG2(Z: in COMPLEX) return COMPLEX;
- function LOG10(Z: in COMPLEX) return COMPLEX;
- function LOG(Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function LOG2(Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function LOG10(Z: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function LOG(Z: in COMPLEX; BASE: in REAL) return COMPLEX;
- function LOG(Z: in COMPLEX_POLAR; BASE: in REAL) return COMPLEX_POLAR;
- function SIN (Z : in COMPLEX) return COMPLEX;
- function SIN (Z : in COMPLEX_POLAR) return COMPLEX_POLAR;

- function COS (Z : in COMPLEX) return COMPLEX;
- function COS (Z : in COMPLEX_POLAR) return COMPLEX_POLAR;
- function SINH (Z : in COMPLEX) return COMPLEX;
- function SINH (Z : in COMPLEX_POLAR) return COMPLEX_POLAR;
- function COSH (Z : in COMPLEX) return COMPLEX;
- function COSH (Z : in COMPLEX_POLAR) return COMPLEX_POLAR;

Arithmetic operators

- function "+" (L: in COMPLEX; R: in COMPLEX) return COMPLEX;
- function "+" (L: in REAL; R: in COMPLEX) return COMPLEX;
- function "+" (L: in COMPLEX; R: in REAL) return COMPLEX;
- function "+" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "+" (L: in REAL; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "+" (L: in COMPLEX_POLAR; R: in REAL) return COMPLEX_POLAR;
- function "-" (L: in COMPLEX; R: in COMPLEX) return COMPLEX;
- function "-" (L: in REAL; R: in COMPLEX) return COMPLEX;
- function "-" (L: in COMPLEX; R: in REAL) return COMPLEX;
- function "-" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "-" (L: in REAL; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "-" (L: in COMPLEX_POLAR; R: in REAL) return COMPLEX_POLAR;
- function "*" (L: in COMPLEX; R: in COMPLEX) return COMPLEX;
- function "*" (L: in REAL; R: in COMPLEX) return COMPLEX;
- function "*" (L: in COMPLEX; R: in REAL) return COMPLEX;
- function "*" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "*" (L: in REAL; R: in COMPLEX_POLAR) return COMPLEX_POLAR;

- function "*" (L: in COMPLEX_POLAR; R: in REAL) return COMPLEX_POLAR;
- function "/" (L: in COMPLEX; R: in COMPLEX) return COMPLEX;
- function "/" (L: in REAL; R: in COMPLEX) return COMPLEX;
- function "/" (L: in COMPLEX; R: in REAL) return COMPLEX;
- function "/" (L: in COMPLEX_POLAR; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "/" (L: in REAL; R: in COMPLEX_POLAR) return COMPLEX_POLAR;
- function "/" (L: in COMPLEX_POLAR; R: in REAL) return COMPLEX_POLAR;

B

Reserved Words

The identifiers listed below are called reserved words and are reserved for significance in the language. A reserved word must not be used as an explicitly declared identifier.

Note: Reserved words differing only in the use of corresponding uppercase and lowercase letters are considered as the same (see LRM 13.3.1). The reserved word range is also used as the name of a predefined attribute.

Note: An extended identifier whose sequence of characters inside the leading and trailing backslashes is identical to a reserved word is not a reserved word. For example, `\next\` is a legal (extended) identifier and is not the reserved word `next`. See LRM 13.3.2.

`abs`	`body`	`entity`
`access`	`break`	`exit`
`across`	`buffer`	`file`
`after`	`bus`	`for`
`alias`	`case`	`function`
`all`	`component`	`generate`
`and`	`configuration`	`generic`
`architecture`	`constant`	`group`
`array`	`disconnect`	`guarded`
`assert`	`downto`	`if`
`attribute`	`else`	`impure`
`begin`	`elseif`	`in`
`block`	`end`	`inertial`

inout
is
label
library
limit
linkage
literal
loop
map
mod
nand
nature
new
next
noise
nor
not
null
of
on
open
or
others
out
package
port
postponed
procedural
procedure
process
pure
quantity
range
record
reference
register
reject
rem
report
return
rol
ror
select
severity
signal
shared
sla
sll
spectrum
sra
srl
subnature
subtype
terminal
then
through
to
tolerance
use
transport
type
unaffected
units
until
variable
wait
when
while
with
xnor
xor

C

Advice and Solutions for VHDL-AMS Compiler Issues

- When the ncvhdl compiler encounters a problem while compiling VHDL-AMS entities or architectures, it can return an error message acronym. Try one of the following compiler options to bypass the issue.
  - For the following error message

    `ncvhdl_p: *E,SSTNTQ`

    try the following compiler option

    `ncvhdl -relax`
  - For the following error message

    `ncvhdl_p: *E,SCSUMM`

    try the following compiler option

    `ncvhdl -nosolvecheck`
  - For problems with VHDL compilation order rules, try the `-SMARTORDER` option.
- Case Sensitivity Issues

  While VHDL-AMS is case-insensitive, the DFII 5X library database is case sensitive. For new cells and new models, use primarily lower case characters for the `cell name`, `entity names`, `generic names`, `port names` and `architecture names`.

  The IUS583 release introduces the `ncvhdl -keepcase4use5x` option. This option forces the `ncvhdl` compiler to retain case sensitivity during compilation. The `-use5xkeepcase` option keeps the case as for a VHDL cell name in a 5X library.

  For example, see the following `Mixed_case` cell in VHDL code.

```
entity Mixed_case
 ...
end entity Mixed_case;
```

This code creates a `Mixed_case` cell instead of a `mixed_case` cell in a 5X library database.

In addition, there is a UNIX environment variable named `CDS_ALT_NMP`. When you set it to `MATCH`

```
setenv CDS_ALT_NMP MATCH
```

then AMS-D in HED and AMS-D in ADE compile using the command

```
ncvhdl -specificunit Abc
```

instead of

```
ncvhdl -specificunit \Abc\
```

In case you have case-sensitive cell name matching with the entity name, you should set: the UNIX "`setenv CDS_ALT_NMP MATCH`" variable and use the ncvhdl option "`-keepcase4use5x`".

Glossary

Across quantity

A quantity that represents the potential across two terminals.

Analog context

Anything evaluated by the analog simulator such as an analog simultaneous statement.

Architecture body

A body associated with an entity declaration to describe the internal organization or operation of a design entity. An architecture body is used to describe the behavior, data flow, or structure of a design entity. There can be multiple architecture associated with a single entity.

Attribute

A definition of some characteristic of a named entity. Some attributes are predefined for types, ranges, values, signals, and functions. The remaining attributes are user defined and are always constants.

Block

The representation of a portion of the hierarchy of a design. A block is either an external block or an internal block.

Break statement

A statement that notifies the analog solver to be prepared to deal with a discontinuity in an analog value at the next analog solution point. During a QUIESCENT_DOMAIN (DC solution), the break statement can also force named quantities to assume specified reset values.

Chip level

A level of abstraction at which a designer can perform realistic simulation of a full system:

1) Synthesis: Without any detailed timing information like FPGAs.

2) Simulation: Accurately simulates the chip.

Circuit level

Interconnected active and passive components such as resistors and capacitors.

Component

Describes instances of an entity and connects signals, quantities, and terminals to the ports of the instances.

Concurrent statement

A statement that executes asynchronously, with no defined relative order. Concurrent statements are used for dataflow and structural descriptions.

Conservative system

One that obeys Kirchhoff's laws.

Constant

An object whose value cannot be changed. Constants must be declared before they are used in models.

Continuous time simulation

A simulation in which values are a continuous function of time.

Discrete time simulation

A simulation in which all changes to the system occur at precise simulation times.

DFII

The former Cadence Design Framework II, which has now become Virtuoso Design Environment in IC 6.1.

Entity declaration

A definition of the interface between a given design entity and the environment in which it is used. Entities can also specify declarations and statements that are part of the design entity. A given entity declaration can be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface.

Function

A collection of sequential statements that are executed for their result.

Generic

An interface constant declared in the block header of a block statement, a component declaration, or an entity declaration. Generics provide a channel for static information to be communicated to a block from its environment. Unlike constants, however, the value of a generic can be supplied externally, either in a component instantiation statement or in a configuration specification.

Generic map

The mapping between generic constants and the actual values they receive when an entity is used in a component instantiation.

Logic gate level

Corresponds directly to gate level primitives such as `AND` and `OR`.

Nature

A definition that specifies values that can be accessed through the attributes of a terminal.

Object

An named entity that has a value of a specified type. An object can be a constant, signal, variable, quantity, terminal, or file.

Package

Used to organize data and subprograms in a model.

Port

A channel for dynamic communication between a block and its environment.

Port interface list

An interface list that declares the inputs and outputs of a block, component, or design entity. The ports can be signals, quantities, or terminals.

Port map

A mapping used in component instantiations that connects actual signals, quantities, or terminals to the formal ports defined in the port interface list for the component.

Procedure

Summarizes a collection of sequential statements that are executed for their effects.

Quantity

An object with a floating point value and which has a continuous solution set by the analog solver.

Quantity port

A quantity, used to model signal flow, that is declared in a port interface list. A direction, either in or out, must be specified on quantity ports.

Reference direction

A direction determined by the order of two terminals that form a branch. For a through quantity, a positive direction represents a flow from the first terminal to the second. Similarly, for an across quantity, a higher potential on the first terminal than on the second is considered a positive potential.

Reference terminal

A terminal that is used by all terminals of a given nature as the zero for the values of its across type. Intuitively, the ground terminal for all terminals of that nature.

Register transfer level (RTL)

Systems described in the form of combinational and sequential functions such as registers, counters and decoders.

Scalar type

A type whose values have no elements. The integer, enumeration, physical, and floating point types.

Sequential statements

Statements that run in sequence.

Shared variable

A variable that can be accessed by more than one process, and by analog expressions. The order of process execution, and therefore of access to shared variables, is unspecified within any one simulation cycle.

Signal

An object with a past history of values. Signals represent interconnection wires.

Signal port

A signal, used to model signal flow, that is declared in a port interface list. Signal ports can be specified with an optional direction: in, out, inout, buffer, or linkage.

Simultaneous statements

Algebraic and differential equations used to specify the analog behavior of a system.

System level

A level of abstraction that describes a complete system.

Terminal

A point of physical connection between devices. A terminal implicitly creates two quantities: `T'reference` (an across quantity from this terminal to the reference

terminal) and `T'contribution` (a through quantity from this terminal to the reference terminal).

Terminal port

A terminal, with no direction, that is declared in a port interface list. Terminal ports are used to model conservative systems.

Through quantity

A quantity that represents the flow through a terminal to a second terminal.

Variable

An object with a single current value. Variables must be declared before being used in a model.

Index

T

V