Mentor Graphics Introduction to VHDL

July 1994

TABLE OF CONTENTS

TABLE OF CONTENTS [continued]

TABLE OF CONTENTS [continued]

LIST OF FIGURES

LIST OF FIGURES [continued]

LIST OF FIGURES [continued]

LIST OF TABLES

About This Manual

This manual introduces some of the language concepts and provides some general coding techniques for the modeling language based on IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*.

Manual Organization

This manual is organized into the following sections:

- Section 1, "Overview," provides an introduction to VHDL and a description of the modeling principles supported by the language.
- Section 2, "VHDL Fundamentals," introduces basic VHDL concepts and defines many of the terms associated with the language.
- Section 3, "Foundation for Declaring Objects--Types," defines objects, types, and the different type classes provided by VHDL.
- Section 4, "Constructs for Decomposing Design Functionality," identifies a number of VHDL constructs that allow the designer to decompose a complex design into smaller and more manageable modules.
- Section 5, "Global Considerations," describes some of the general issues you must consider when designing and modeling with VHDL.
- Section 6, "Coding Techniques," provides a number of hardware problems and some possible corresponding VHDL solutions.
- "Glossary," defines terms that appear in this manual and the *Mentor Graphics VHDL Reference Manual*.

Notational Conventions

For information about VHDL syntax conventions used in this manual, refer to the *Mentor Graphics VHDL Reference Manual*. Also refer to the "BNF Syntax Description Method" section in the *Mentor Graphics VHDL Reference Manual*.

For information about general documentation conventions, refer to *Mentor Graphics Documentation Conventions*.

Related Publications

In an effort to consolidate this information, the related publications list for VHDL can be found in the Related Publications section of the *Mentor Graphics VHDL Reference Manual*.

Section 1 Overview

This section provides an overview of VHDL, which is based on IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*. VHDL stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. The following list outlines the major topics covered in this section:

VHDL is a design and modeling language specifically designed to describe (in machine- and human-readable form) the organization and function of digital hardware systems, circuit boards, and components.

Figure 1-1 shows some of the various things that you can model with VHDL. This manual refers to a VHDL model as a textual description of a hardware design or piece of a design that, when simulated, mimics the design's behavior. A VHDL model can describe the behavior of either a new design or of pre-existing hardware. The terms VHDL description and VHDL model are used interchangeably. A design refers to the product under development whether it be a system, circuit board, or component.

Also refer to the discussion on "Designing or Modeling?" in the *Digital Modeling Guide*.

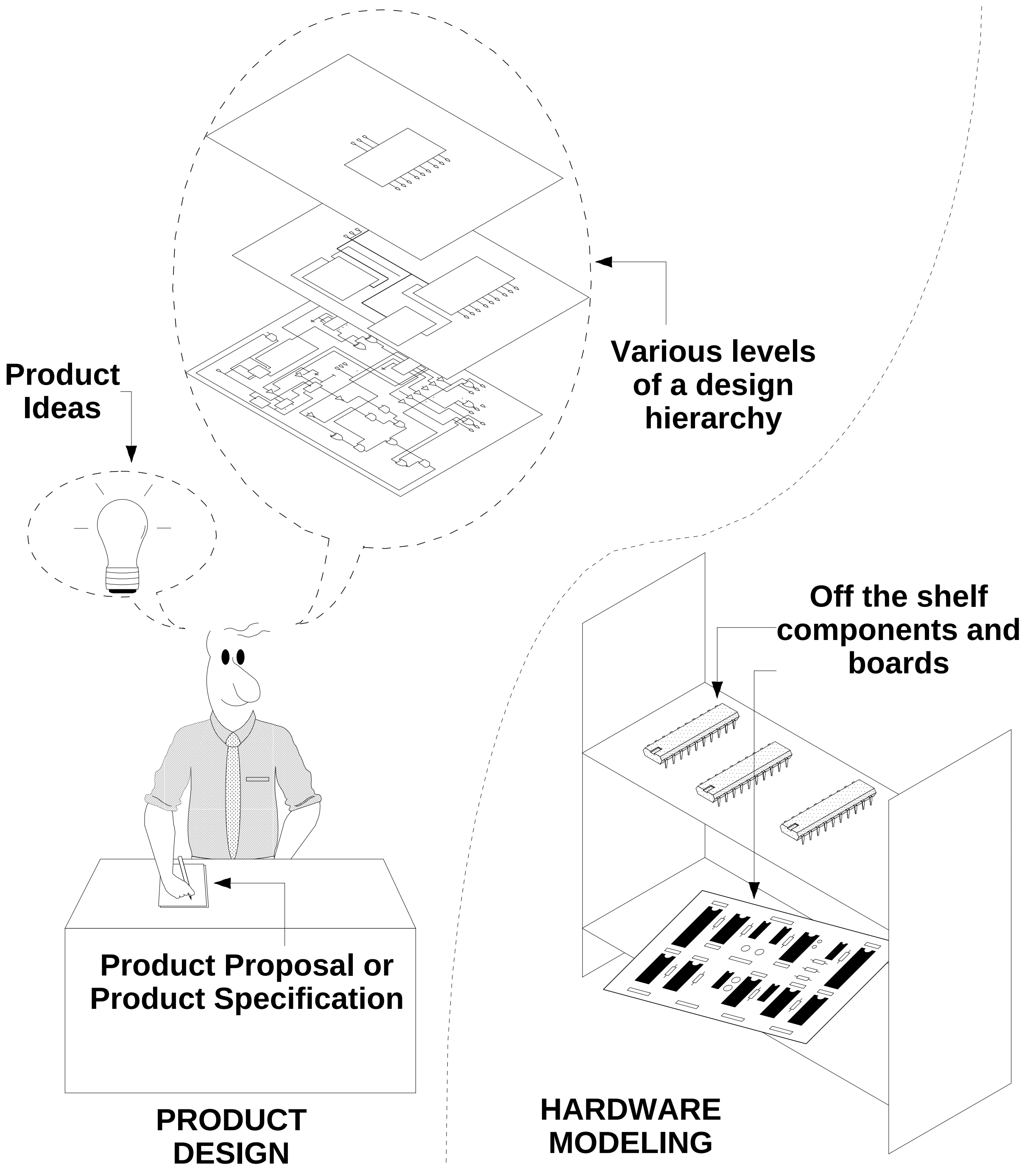

Product Ideas
Various levels of a design hierarchy
Off the shelf components and boards
Product Proposal or Product Specification
PRODUCT DESIGN
HARDWARE MODELING

Figure 1-1. Various Things You Can Describe with VHDL

Designing refers to the process of developing new ideas into a product. Modeling usually refers to the process of creating a simulateable description (model) of the behavior of pre-existing products. (In Figure 1-1, this process is represented by the term Hardware Modeling.) Designing with VHDL involves creating and using models that you can simulate.

When you use VHDL as a design tool, you can describe a product idea, a product proposal (possibly the next step after the idea), a product specification, and then various design abstraction levels.

The compiled VHDL code provides a software model of your design or pre-existing hardware that you can test using digital simulators. In the hardware design process, you can analyze and refine your VHDL design description on a workstation before reaching the prototype stage.

VHDL descriptions that are created following VHDL synthesis guidelines can be run through a synthesis tool to create a gate-level implementation of the design.

General VHDL Modeling Principles

Because VHDL is a powerful language, you can write hardware descriptions that satisfy two important goals:

- The VHDL descriptions will be easy to understand.
- The VHDL descriptions will be modifiable.

Easy-to-understand code benefits anyone who must read the code, especially if the original designer is not available to clarify any ambiguity.

Modifiable code is equally beneficial. There are several reasons why you might need to change your VHDL hardware description. For example, the hardware requirements may have changed as the design developed, or you may have found an error or timing problem during simulation of the software model.

In either case, VHDL supports the following principles that make it possible to write, modify, and maintain complex hardware design descriptions:

- Top-down design
- Abstraction
- Uniformity
- Modularity
- Information-hiding

The following subsections further describe these modeling principles and show how they help make complex hardware descriptions readable, easy to understand, and modifiable.

Top-Down Design

When designing a complex hardware system or ASIC (application-specific integrated circuit), an engineer usually conceptualizes the design function with block diagrams at a high abstraction level. VHDL, within a Mentor Graphics simulation environment, allows you to

- Model the behavior of the high-level blocks
- Analyze (simulate) them
- Refine the high-level functionality as required before reaching the lower abstraction levels of design implementation

With the addition of the Mentor Graphics synthesis application, a VHDL design can be synthesized to the gate level.

Correcting design errors earlier in the design process is less costly than at the silicon or component implementation level.

Modularity

Modularity is the principle of partitioning (or decomposing) a hardware design and associated VHDL description into smaller units. Figure 1-2 shows a flat design (no hierarchy) partitioned into smaller units. When you design hardware with VHDL, the function of each hardware partition can be described with a module of code (represented as three-dimensional rectangular boxes in Figure 1-2). This makes the hardware description easier to manage and understand.

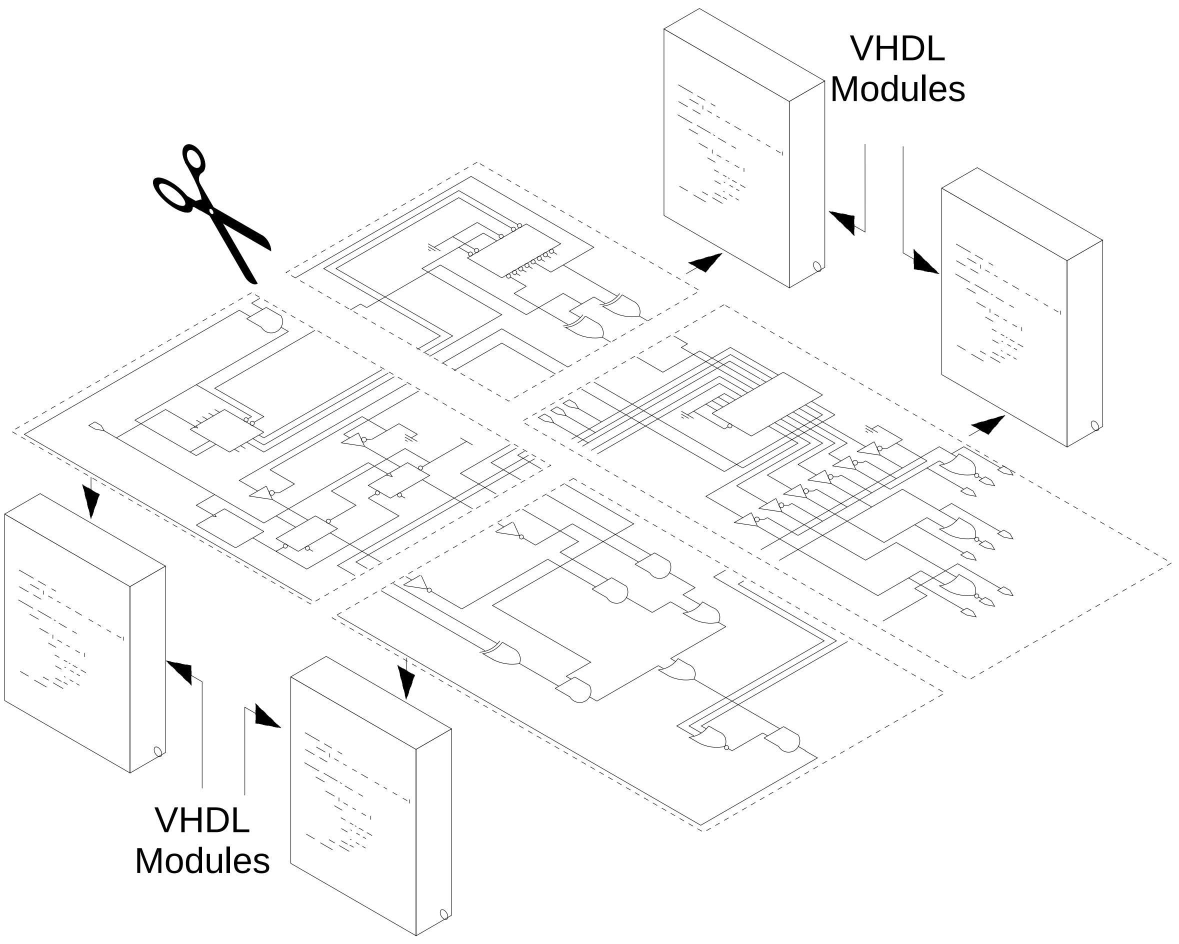


Figure 1-2. Flat-Level Partitioning of a Hardware Design

VHDL is composed of language building blocks that consist of over 75 reserved words and about 200 descriptive words or word combinations. Figure 1-3 illustrates how each VHDL module consists of various language building blocks. Figure 1-3 shows a module that represents a description of a NAND gate.

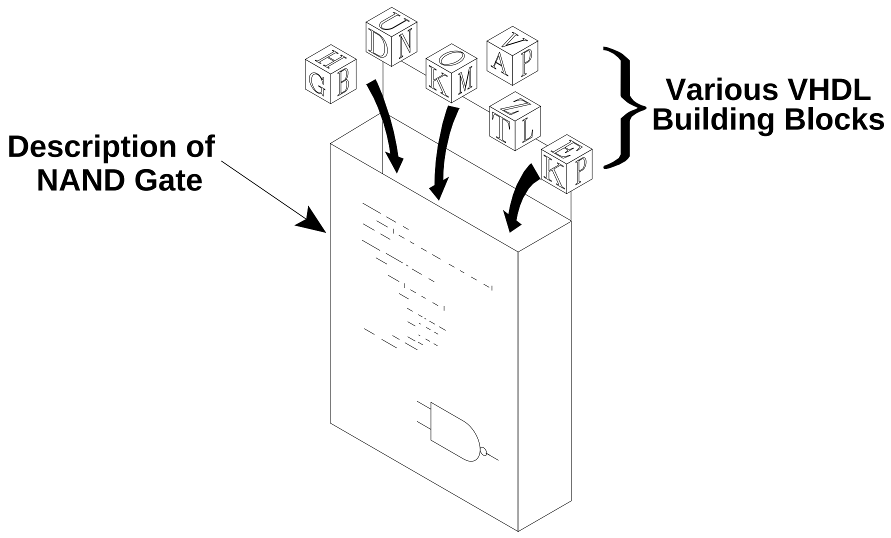


Figure 1-3. A Hardware Module Created from VHDL Building Blocks

Figure 1-4 shows a hierarchical method of partitioning a design into smaller modules. The VHDL description of a shifter is partitioned into modules that describe the underlying structure. The shifter description contains an indirect reference to the NAND gate module. The structure of the shifter is described in a higher-level module as a series of interconnected flip-flops. In a lower-level module, the structure of the flip-flop is described as two interconnected NAND gates. In a separate module at an even deeper level, the function of a NAND gate is described. Each module is a self-contained description of the various parts used to describe a shifter.

The self-contained modules need to interface to other modules in a design in order to work as one unit. At the highest level, the shifter module in Figure 1-4 contains a well-defined interface that couples it to the flip-flop module. The flip-flop description at the middle level contains an interface that couples it to the lower-level description of the NAND gate.

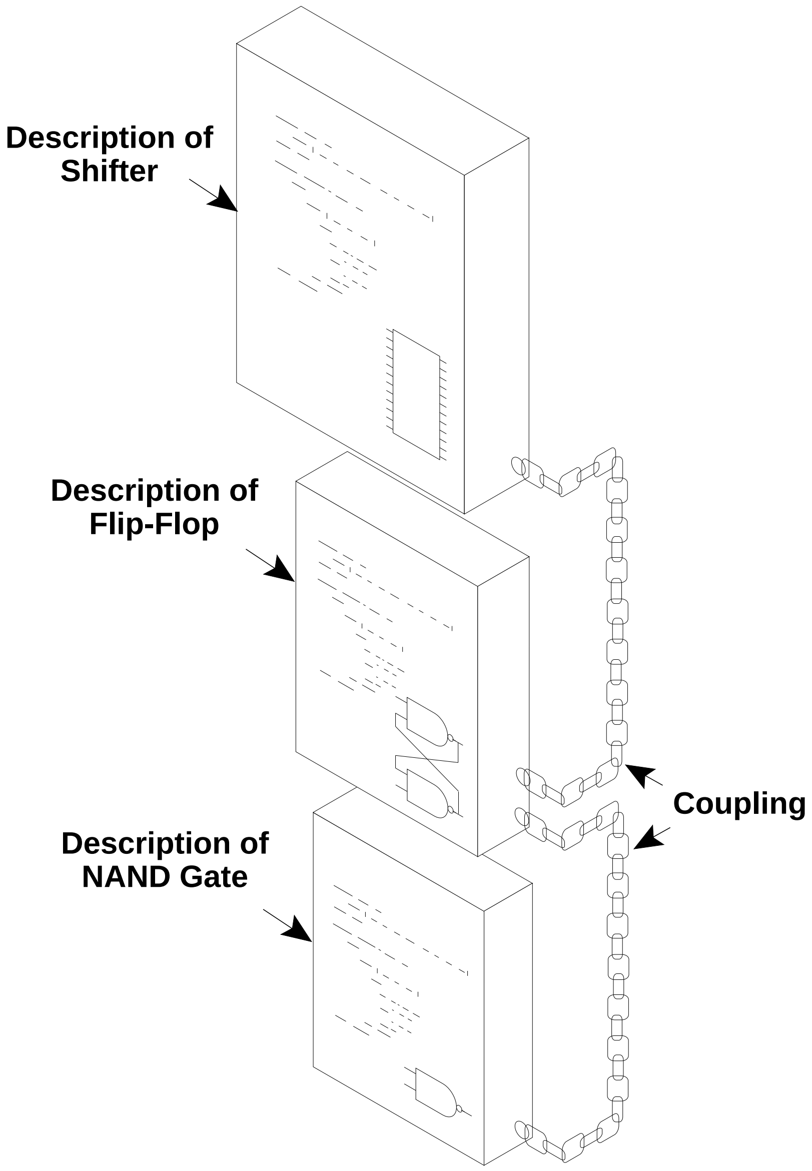


Figure 1-4. Hierarchical Partioning of a VHDL Shifter Description

One reason to keep the description of the NAND gate and the flip-flop localized in separate modules is to make it possible to couple more than one high-level description to the lower-level modules. Localization makes the lower-level modules reusable and eliminates repetition. Another reason to localize the flip-flop and NAND gate modules is that the high-level description of the shifter is kept relatively simple and uncluttered.

It would not be a difficult task to modify the shifter description (such as adding more inputs and outputs) without changing the flip-flop or NAND gate descriptions. By using modularity in your VHDL descriptions, you meet the goals of keeping your description easy to understand and modifiable.

Abstraction

An *abstraction* will group details (in a module) that describe the function of a design unit but does not describe how the design unit is implemented. This principle is closely related to modularity. In Figure 1-4, the flip-flop is an abstraction of the NAND gate level, and the shifter is an abstraction of the flip-flop level. Each abstraction is built from lower levels.

Figure 1-5 shows another way you can describe a hardware design using various levels of abstraction. A Read-Only-Memory (ROM) device is described at a high level as a series of address locations with corresponding data bytes stored in each location. At this level you do not care about address lines, data lines, or control lines. You can concentrate on the data byte assignments to selected addresses without thinking about the many signal lines that must be controlled at a lower level.

In the lower-level module, you can describe how each signal on the ROM pins must be configured to read or program each data storage location. If you needed to change the data stored in a given ROM location, you could go to the higher-level module and change a hex value associated with an address rather than redefine the states of many data lines. You again meet the modeling goals of keeping the design easy to understand and maintainable by using abstraction.

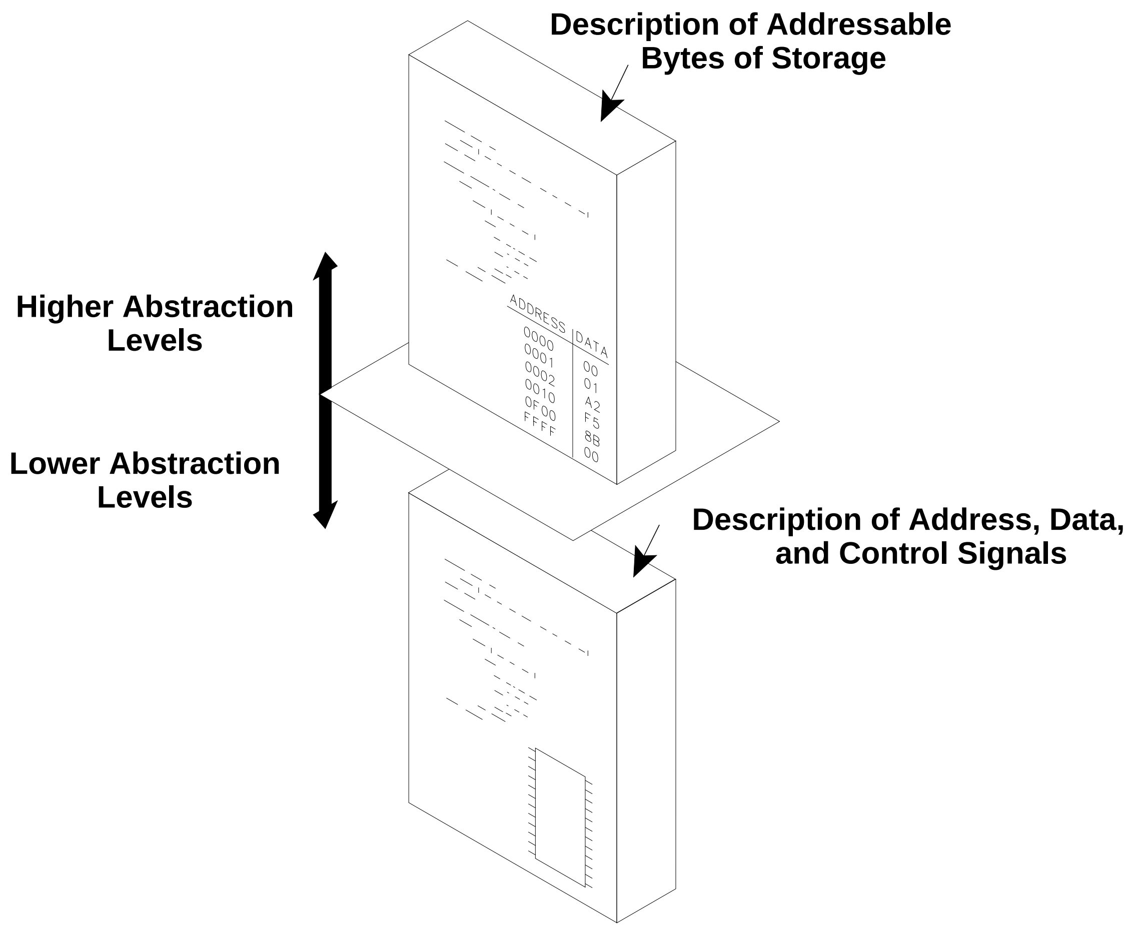

Description of Addressable Bytes of Storage
Higher Abstraction Levels
Lower Abstraction Levels
ADDRESS
DATA
0000 00
0001 01
0002 A2
0010 F5
0F00 8B
FFFF 00
Description of Address, Data, and Control Signals

Figure 1-5. Applying Abstraction to a ROM Description

Information Hiding

When coding a particular hardware module, it may be desirable to hide the implementation details from other modules. *Information hiding* is another useful principle for making VHDL designs manageable and easier to read. This principle complements abstraction, which extracts the functional details in a given module. By hiding implementation details from other modules, a designer's attention is focused on the relevant information while the irrelevant details are made inaccessible.

In the preceding VHDL shifter description (shown in Figure 1-4), the NAND gate level of abstraction can be hidden from the person who is coding the flip-flop description. Figure 1-6 shows a representation of this principle.

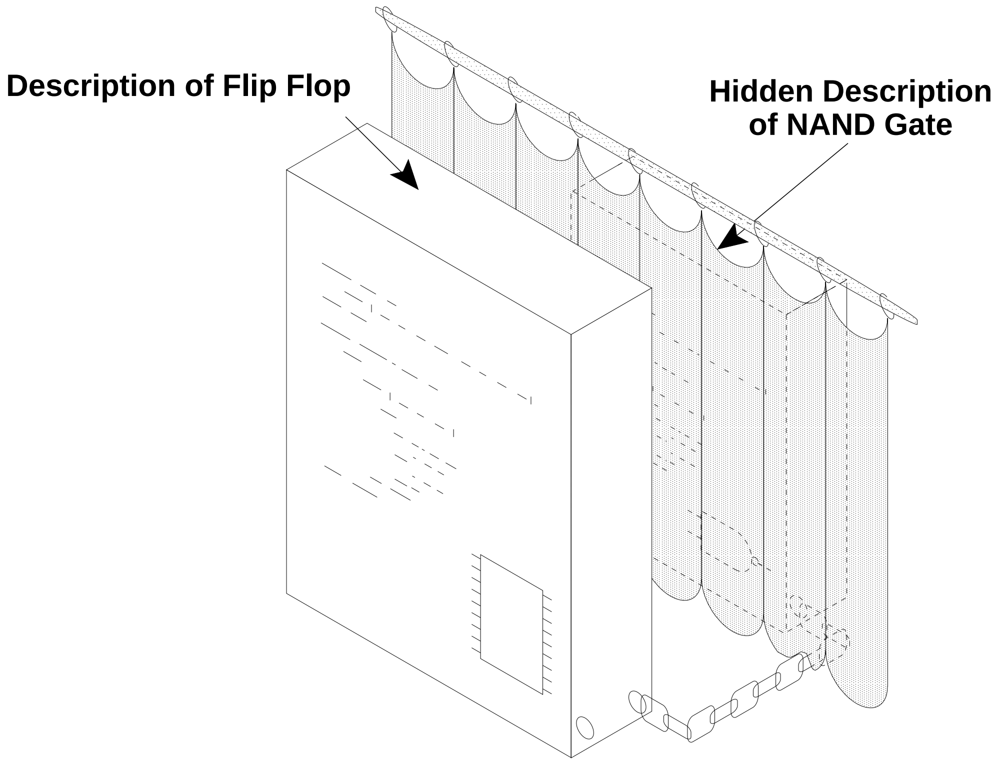


Figure 1-6. Hiding Unessential Details of NAND Gate Level

The person describing the flip-flop does not really care (at this level) how the NAND gate internals work. The NAND gate can be a previously-coded description that was compiled and stored in a library. The designer needs only to know how to interface to the input and output pins of the NAND gate. In this way, the flip-flop designer can ignore the details of how the NAND gate is implemented.

Another function of information hiding is to protect proprietary information when distributing VHDL models outside a company. By distributing only the compiled code (executable), the proprietary information (source code) can be hidden from its users.

Uniformity

In addition to the principles of modularity, abstraction, and information-hiding, uniformity is another principle that helps to make your hardware description readable. *Uniformity* means that you create each module of code in a similar way by using the various VHDL building blocks. Uniformity implies good coding style, such as consistent code indentation and informative comments. For further information, refer to the "General VHDL Coding Guidelines" subsection on page 6-1.

Summary

- VHDL is a modeling and design language specifically designed to describe (in machine- and human-readable form) the organization and function of digital hardware systems, circuit boards, and components.

- The following list describes several reasons why you would use VHDL to design and model your new product ideas or pre-existing hardware:

  - VHDL allows you to design, model, and test a system from the high level of abstraction down to the structural gate level.

  - VHDL descriptions created following by VHDL synthesis guidelines can be run through a synthesis tool to create gate-level implementations of designs.

  - Because this hardware language is based on IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, engineers throughout the design industry can use this language to minimize communication errors and incompatibility problems.

  - At Mentor Graphics, VHDL is integrated into one overall design environment. It is possible to do a system-level simulation mixing high-level, abstract descriptions with detailed gate-level models.

- VHDL supports the following principles that make it possible for you to write, modify, and maintain complex hardware design descriptions:

  - Top-down design--the method of describing (modeling) the behavior of the high-level blocks, analyzing (simulating) them, and refining the high-level functionality as required before reaching the lower abstraction levels of design implementation

- ❍ Modularity--the principle of partitioning (or decomposing) a hardware design and the associated VHDL description into smaller units
- ❍ Abstraction--grouping details (in a module) that describe the function of a design unit but do not describe how the design unit is implemented
- ❍ Information-hiding--hiding the implementation details of one module from other modules
- ❍ Uniformity--creating the design modules from the language building blocks in a consistent way

Section 2
VHDL Fundamentals

This section introduces fundamental VHDL concepts and defines many of the terms associated with the language. Simple hardware examples are used throughout the section to illustrate many of the concepts. It is important to note that these examples may not represent ideal design or model solutions. In addition, they are not meant to provide you with a full understanding of each language building block but to give you a good introduction to the VHDL form.

Section 6 covers specific design tasks in further detail. The following list outlines the major topics covered in this section:

VHDL Building Blocks

VHDL is composed of language building blocks that consist of more than 75 reserved words and about 200 descriptive words or word combinations. These building blocks are used to create the data types and instructions that make up a VHDL description.

Reserved words are words that have specific meaning to a VHDL compiler, such as the word **port**. Certain characters, such as the left and right parentheses and the semicolon, are also classified as reserved words. Do not use reserved words except as defined by VHDL.

Examples of descriptive word combinations are "port clause" and "port list". Although these word combinations would not appear in actual code, they provide a name to the building blocks that you use when building a VHDL description.

The building blocks of VHDL are called *language constructs*. A language construct is an item that is constructed from basic items such as reserved words or other language building blocks. For example, the syntax diagram in Figure 2-1 shows that the language construct called port_clause is composed of the following: the reserved word **port** followed by the reserved word "**(**", another building block called a port_list, the reserved word "**)**", and finally the reserved word "**;**". Those items enclosed in an oval or circle appear verbatim in the VHDL code. Items enclosed in rectangles are other language constructs that are defined in separate syntax diagrams.

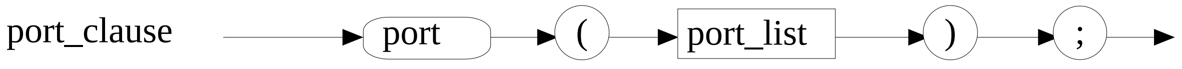


Figure 2-1. Port Clause Syntax Diagram

For a complete listing of VHDL reserved words and syntax diagrams for each language construct, refer to the *Mentor Graphics VHDL Reference Manual*.

This manual presents syntax information in a summary format that suggests how the real code looks. The right column of the following syntax example shows the syntax summary for port clause:

port clause ....   **port (** port_list **) ;**

Examples presented in this format do not always show which items in the syntax are optional or which ones you can use more than once. For the detailed information, you should consult the syntax diagrams and the BNF diagrams in the *Mentor Graphics VHDL Reference Manual*.

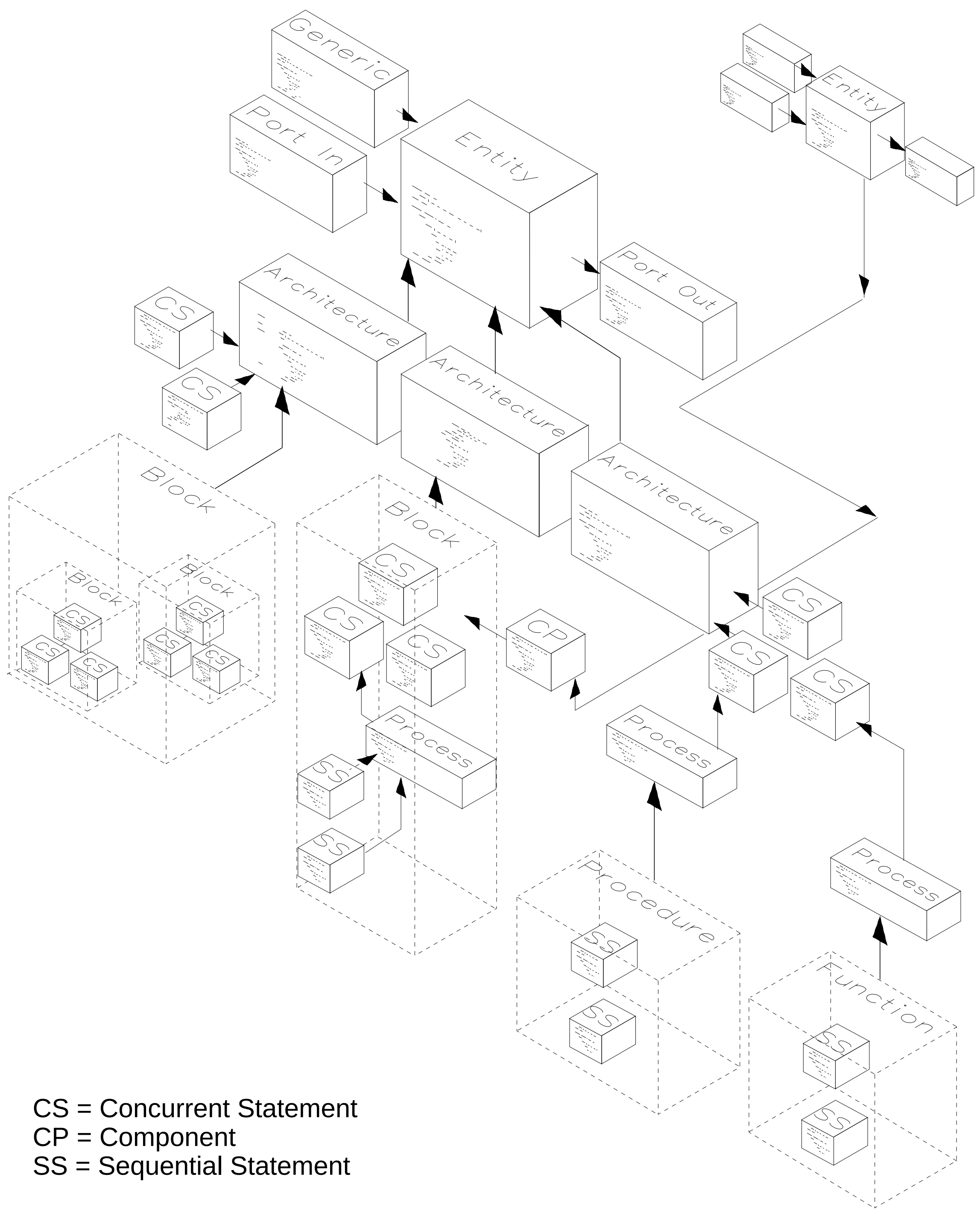
Major Language Constructs

Figure 2-2 shows the hierarchy of the major language constructs. Each block in the figure represents a major language construct and shows its position relative to other constructs. The following paragraphs briefly introduce some of the constructs that are further explained in later subsections. You might want to refer to Figure 2-2 when reading about the various language constructs in the later subsections.

At the top of the pyramid-like structure in Figure 2-2 is the *design entity*. A design entity is the basic unit of a hardware description.

At the next level, the figure shows that a design entity is composed of one or more architectures. The architecture describes the relationships between the design entity inputs and outputs. Each architecture consists of concurrent statements, denoted as CS in Figure 2-2. Concurrent statements define interconnected processes and blocks that together describe a design’s overall behavior or structure.

You can group concurrent statements using the block statement. This grouping is represented by a dashed block in Figure 2-2. Groups of blocks can also be partitioned into other blocks. At this same level, a VHDL component (denoted as CP in Figure 2-2) can be instantiated and connected to defined signals within the blocks. The VHDL component is a reference to an entity.



CS = Concurrent Statement
CP = Component
SS = Sequential Statement

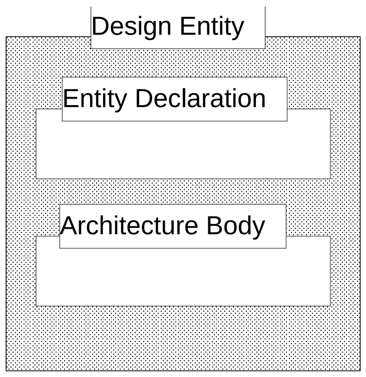
Figure 2-2. Major Language Construct Hierarchy

A process can be a single signal assignment statement or a series of sequential statements (denoted as SS in Figure 2-2). Within a process, block, or package, procedures and functions can partition the sequential statements. Refer to the "Locating Language Constructs" appendix in the *Mentor Graphics VHDL Reference Manual* for information on where major constructs can be positioned within a VHDL design description.

A package (not shown in the figure) allows you to group a collection of related items for use by one or more separate modules of code.

Primary Language Abstraction

During the design process you usually decompose hardware designs into smaller, more manageable units. VHDL supports this hardware decomposition and makes it possible for you to write a hardware description so that many of the smaller parts are reusable by different portions of the overall design (or even other designs).

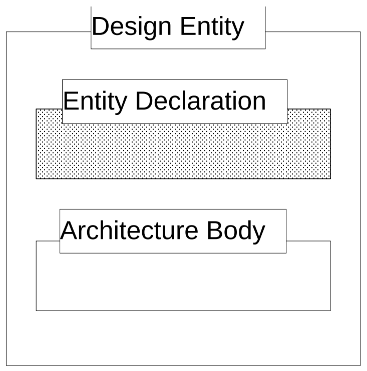


The primary abstraction level of a VHDL hardware model is the design entity. The design entity can represent a cell, chip, board, or subsystem.

A design entity is composed of two main parts: an entity declaration and an architecture body*.

Entity declarations and architecture bodies are two of the VHDL language library units. A library unit is a portion of the hardware description (model) that can be contained and compiled in a separate design file. (Package declarations and package bodies are two other library units.) This capability allows you to modularize a design description by compiling each entity or package declaration separate from the corresponding body.

*Refer to the appropriate syntax diagrams in the *Mentor Graphics VHDL Reference Manual* for a listing of all the possible building blocks in an entity declaration and an architecture body.



An entity declaration defines the interface between the design entity and the environment outside of the design entity. The structure of an entity declaration is shown in the following example:

```
entity identifier is
   entity_header
     -- (generic and/or port clauses)
  entity_declarative_part
     -- (declarations for subprograms,
     --  types, signals, ...)
begin
entity_statement_part
end identifier ;
```

The entity identifier is a descriptive name that you assign. Each design entity receives information from the outside via a port (of mode **in**) or a generic interface. The design entity sends out information via a port (of mode **out**). Also see the top of Figure 2-2 on page 2-4.

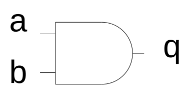
A generic interface defines parameters (such as delay data) that can be passed into the entity when it is instantiated*. Generics allows you to define a reusable design entity with variable parameters that can be customized for each use of the design entity.

The basic format of a generic clause and a port clause are shown as follows:

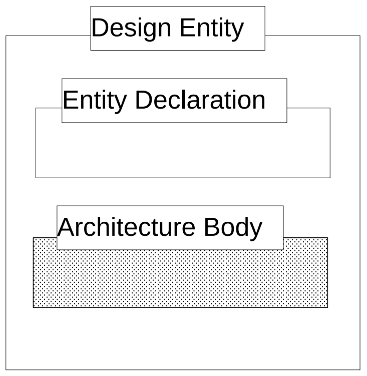
generic clause	**generic (** generic_list **) ;**
port clause	**port (** port_list **) ;**

The following example shows an entity declaration (including a port clause) for the simple two-input AND gate shown at the left. The convention used in this manual to identify reserved words within code examples is to display them in all uppercase characters. Also refer to the Notational Conventions section in the *Mentor Graphics VHDL Reference Manual*.

*Passing parameters in this way is further explained in the "Component Instantiation" subsection on page 4-7.



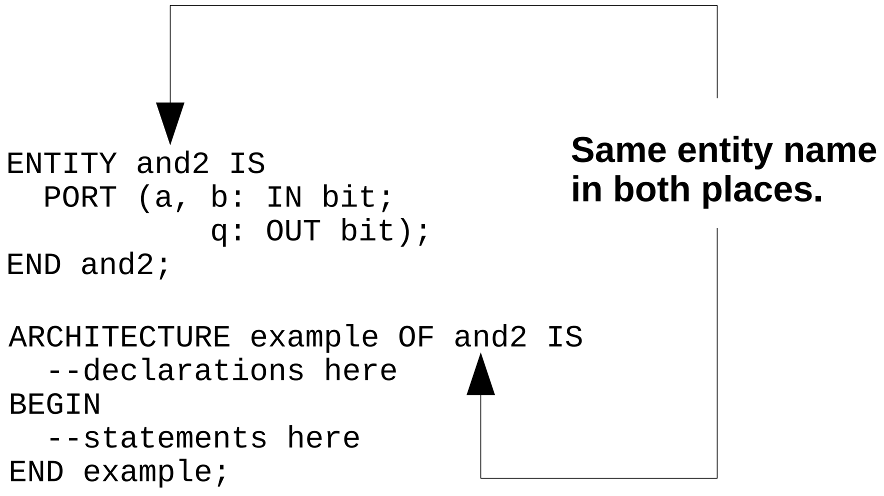
```
ENTITY and2 IS
  PORT (a, b: IN bit;
           q: OUT bit);
 END and2;
```



The architecture body describes the relationships between the design entity inputs and outputs. The structure of this construct is shown in the following example:

architecture identifier **of** entity_name **is**
 architecture_declarative_part
begin
architecture_statement_part
end identifier**;**

The identifier and entity_name are words that you provide in your VHDL code. The entity name in the architecture body must be the same as the identifier of the corresponding entity declaration as shown in Figure 2-3.



```
ENTITY and2 IS
  PORT (a, b: IN bit;
           q: OUT bit);
END and2;

ARCHITECTURE example OF and2 IS
  --declarations here
BEGIN
  --statements here
END example;
```

Figure 2-3. Entity Name Usage in Entity Declaration and Architecture Body

You define the behavior or structure of a design entity in the architecture body using one or more methods described in the "Design Description Methods" subsection, beginning on page 2-10.

A given design entity may have more than one architecture body to describe its behavior and/or structure as shown in Figure 2-4.

You would write the entity declaration (entity name could be "trfc_lc" as indicated at the top of Figure 2-4) and compile it. Then you could write and compile a high-abstraction level behavioral description of the circuit. The architecture name could be "behav" as shown in the lower-right corner of Architecture Body 1 in Figure 2-4.

Once you are satisfied that the circuit behavior (at the high-abstraction level) is functioning, you can write another architecture body to test circuit functions at a lower-abstraction level. Architecture Body 2 in Figure 2-4 includes some structure and data flow details. The architecture name of this body is "dflow."

Then you can simulate this second level architecture and make refinements to Architecture Body 2 as required until the expected results are achieved.

The lowest abstraction level you write could be a structural description of the circuit that implements the design function at the component level. Architecture Body 3 in Figure 2-4 represents this abstraction level. The architecture name of this body is "struct."

Using three different architecture bodies for this one design allows you to develop the circuit description using top-down methodology. Each abstraction level is documented and saved in a separate design file.

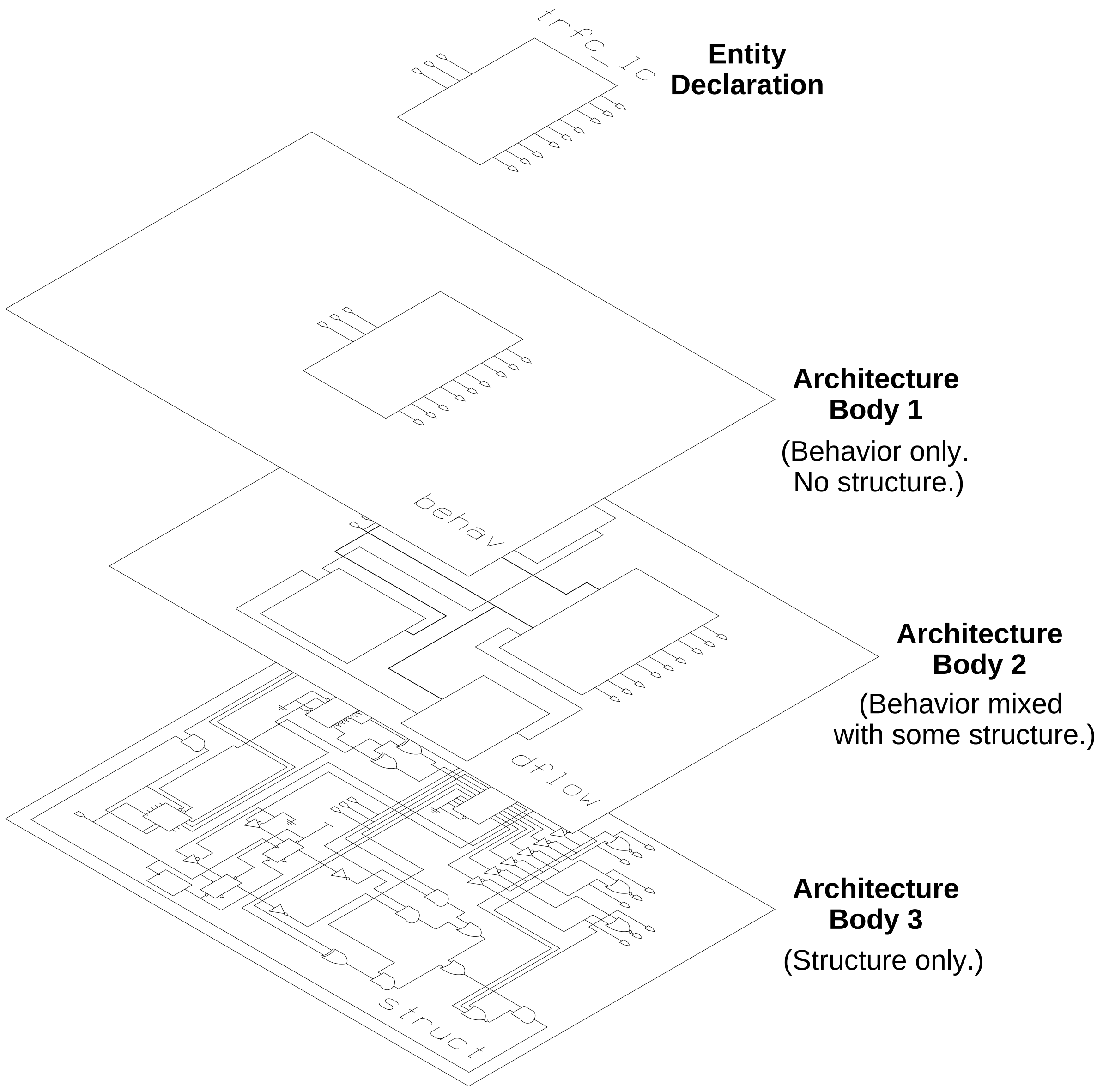


Figure 2-4. Multiple Architecture Bodies for One Entity Declaration

Design Description Methods

VHDL provides a textual method of describing a hardware design in place of a schematic representation. The following list shows the various VHDL methods for describing hardware architectures:

- *Structural description* method expresses the design as an arrangement of interconnected components.
- *Behavioral description* method describes the functional behavior of a hardware design in terms of circuits and signal responses to various stimuli. The hardware behavior is described algorithmically without showing how it is structurally implemented.
- *Data-flow description* method is similar to a register-transfer language. This method describes the function of a design by defining the flow of information from one input or register to another register or output.

All three methods of describing the hardware architecture can be intermixed in a single design description.

Structural Description

This subsection uses a two-input multiplexer to identify some of the language constructs in a VHDL structural description. This description provides an overview and not a complete representation of all the language building blocks found in a structural description. Refer to the appropriate syntax diagrams in the *Mentor Graphics VHDL Reference Manual* for a complete flow of the language constructs described in this subsection.

A VHDL structural description of a hardware design is similar to a schematic representation because the interconnectivity of the components is shown. This similarity is illustrated in this subsection with a comparison of a simple schematic design to a VHDL structural description of the same circuit.

Figure 2-5 shows the symbol of a two-input multiplexer (MUX). This MUX is a hierarchical design, as shown in Figure 2-6, with the bottom sheet containing the schematic representation or description of the internal structure, as shown in Figure 2-7. Note the pin names on the inside of the MUX symbol in Figure 2-5 match the net names of the inputs and output of the schematic in Figure 2-7.

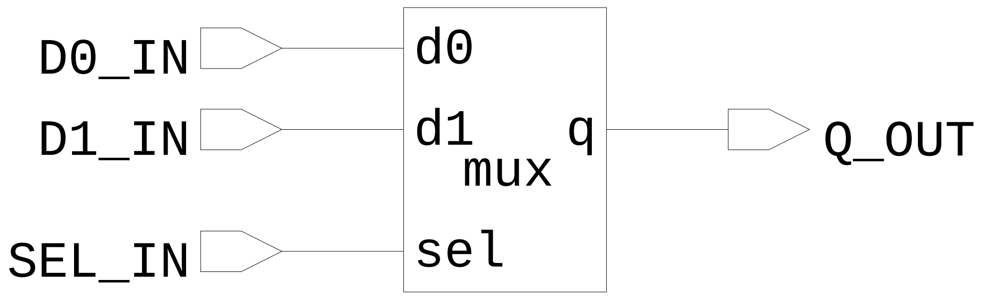


Figure 2-5. Symbol Representation of Two-Input Multiplexer

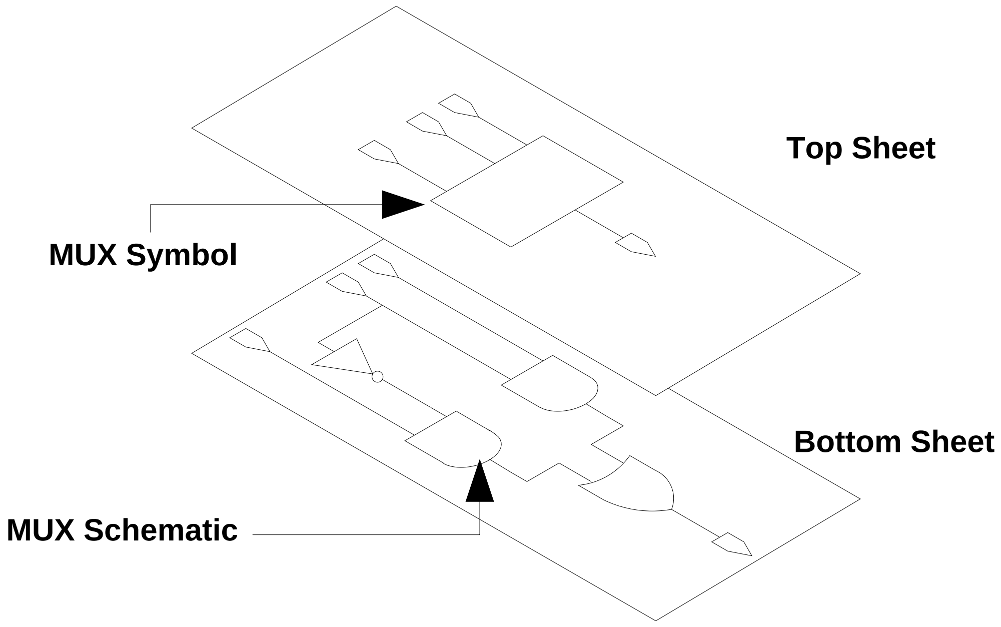


Figure 2-6. A Schematic Editor Hierarchical Design of a Multiplexer

Figure 2-8 shows a VHDL structural description of the two-input multiplexer. The VHDL code contains comments that are set off with a double dash (--). Any text appearing between the double dash and the end of a line is ignored by the compiler. (See lines 1,2, 5, 7, 17, 19 through 21, and 24 in Figure 2-8.) Descriptive comments make the code easier to read.

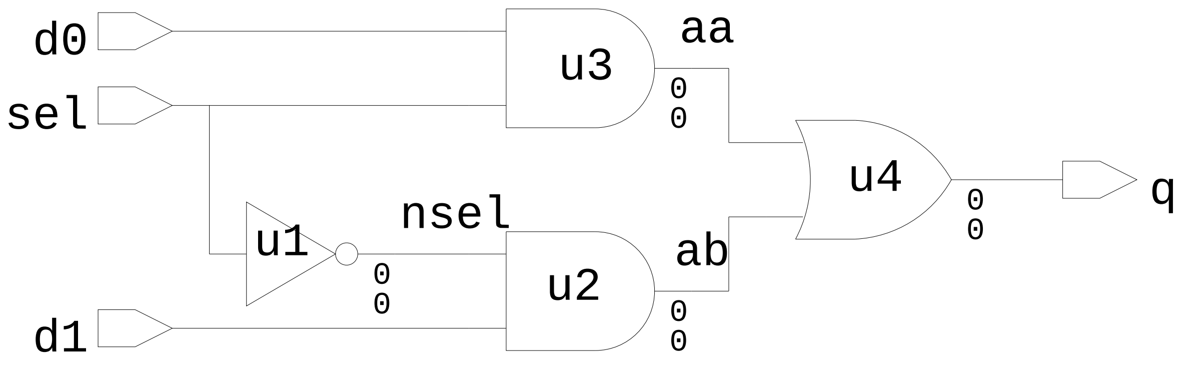


Figure 2-7. Gate-Level Representation of Two-Input Multiplexer

```
ENTITY mux IS                  -- entity declaration
  PORT (d0, d1, sel: IN bit; q: OUT bit); --port clause
END mux;

                             -- architecture body
ARCHITECTURE struct OF mux IS
  COMPONENT and2               --architecture decl. part
    PORT(a, b: IN bit; c: OUT bit);
  END COMPONENT;
  COMPONENT or2
    PORT(a, b: IN bit; c: OUT bit);
  END COMPONENT;
  COMPONENT inv
    PORT (a: IN bit; c: OUT bit);
  END COMPONENT;

  SIGNAL aa, ab, nsel: bit;        --signal declaration

  FOR u1    :inv  USE ENTITY WORK.invrt(behav); -- config.
  FOR u2, u3:and2 USE ENTITY WORK.and_gt(dflw); -- specif.
  FOR u4    :or2  USE ENTITY WORK.or_gt(arch1); --

BEGIN
  u1:inv  PORT MAP(sel, nsel);--architecture statement part
  u2:and2 PORT MAP(nsel,d1,ab);
  u3:and2 PORT MAP(d0, sel,aa);
  u4:or2  PORT MAP(aa, ab, q);
END struct;
```

Figure 2-8. Code of Structural Description for a Multiplexer

The two-input MUX represented by Figure 2-8 is a basic design unit. The entity declaration at the top of Figure 2-8 (lines 1 through 3) defines the interface between the design entity and the environment outside of the design entity.

This entity declaration contains a port clause that provides input channels (signals `d0`, `d1`, and `sel` in Figure 2-8, line 2) and an output channel (signal `q` in Figure 2-8, line 2). The signals are of a predefined type called `bit` which is declared elsewhere to describe all possible values (0 or 1) for each signal. (Types are described on page 3-1.) This entity declaration can be compared with the MUX symbol in the schematic design in Figure 2-6.

The architecture body in Figure 2-8 (lines 6 through 28) describes the relationships between the design entity inputs and outputs structurally. This architecture body performs a function similar to the bottom sheet in the schematic design in Figure 2-6.

The various components (`and2`, `or2`, and `inv`) that form the `mux` design entity in Figure 2-8 are declared in the architecture declarative part (lines 7 through 15). Signals (`aa`, `ab`, and `nsel`) are also declared in the architecture body (line 17) to represent the output of the two AND gates (`u2` and `u3`) and the inverter (`u1`).

The configuration specifications in lines 19 through 21 bind each component instance to a specific design entity which describes how each component operates. For example, the component `u1` used in line 24 of Figure 2-8 is bound to an architecture body called `behav` for a design entity called `invrt`.

The architecture statement part (lines 24 through 27) describes the connections between the components within the design entity. In this part, the declared components are instantiated. (For more information on component declaration and instantiation, see "Component Instantiation" on page 4-7.)

Figure 2-9 shows how a schematic sheet could contain a MUX symbol with an associated VHDL structural description. Instead of using an underlying schematic sheet, the VHDL structural description defines the internal structure of the component.

In the design shown in Figures 2-6 through 2-9, the behavior of the MUX was determined by the connections between the inverter, the AND gates, and the OR gate. The function of these gates is generally understood.

In a more complex design, the components `u1` through `u4` in Figure 2-8 could represent entities that have complicated functions such as a central processing unit or a bus controller. When function and not structure is most important, you can describe each component with a corresponding behavioral description.

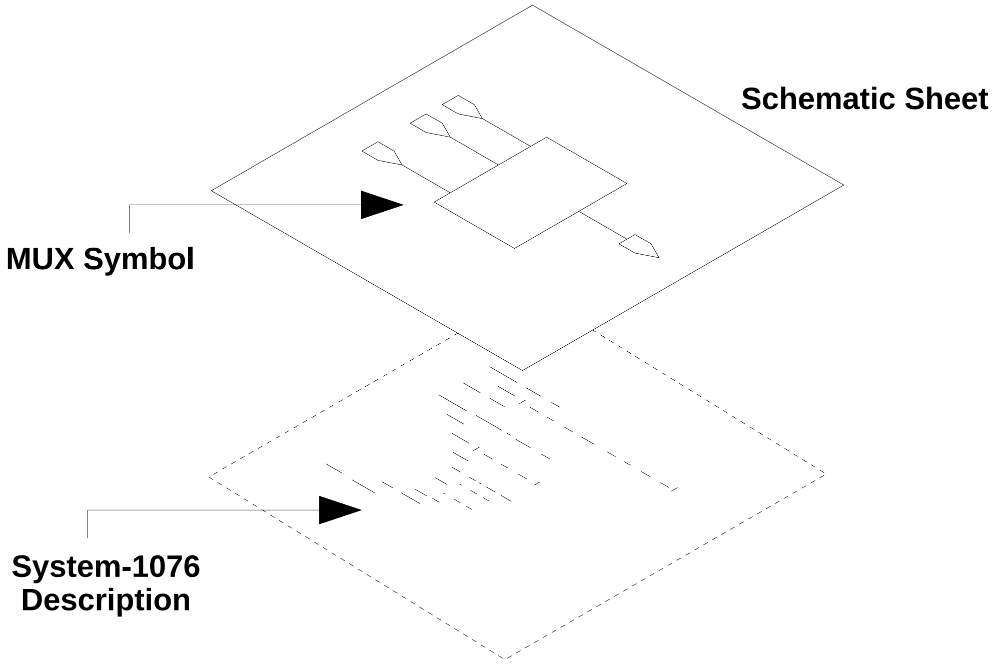


Figure 2-9. Two-Input Multiplexer with Associated Structural Description

Behavioral Description

A VHDL behavioral description represents the function of a design in terms of circuit and signal response to various stimulus. This subsection identifies some of the major language constructs found in a behavioral description using the previous MUX example and a four-bit shifter example. Refer to the appropriate syntax diagrams in the *Mentor Graphics VHDL Reference Manual* for a complete flow of the language constructs described in this subsection. After reading the previous subsection on structural descriptions, you can compare that method with the behavioral description method that is described in this subsection.

Figure 2-10 shows a behavioral description of the mux example described in the structural description subsection. Like the structure description, you can include the MUX symbol on a schematic sheet, except this time, the VHDL model defines the behavior, of the component during circuit simulation.

The behavioral description in Figure 2-10 and the structural description in Figure 2-8 both contain an entity declaration and an architecture body. In practice, you most likely would not have both the behavioral and structural architecture body shown in Figures 2-8 and 2-10 in one source file (although it is possible). You can first write the entity declaration in one design file, then the behavioral architecture in another design file, and the structural architecture in still another design file.

In an actual design, after the entity declaration is written and compiled, you might next write a behavioral architecture to allow testing of the overall circuit functions. After you simulate and refine the functional model, you then might write a structural architecture. You can substitute the structural architecture body for the behavioral and then the model can be simulated again.

```
ENTITY mux IS           --  entity declaration
  PORT (d0, d1, sel: IN bit; q: OUT bit); --port clause
END mux;
                              --  architecture body
ARCHITECTURE behav OF mux IS
BEGIN
  f1:                          -- process statement
  PROCESS (d0, d1, sel)        -- sensitivity list
  BEGIN
    IF sel = ’0’ THEN            -- process statement part
        q <= d1;
    ELSE
        q <= d0;
    END IF;
  END PROCESS f1;
END behav;
```

Figure 2-10. Code of Behavioral Description for a Multiplexer

A behavioral description model is also useful to stimulate inputs of other VHDL models during simulation. For example, you might have designed a traffic light controller using a structural description and now you wish to test it. The traffic light controller has inputs that connect to traffic sensors. For simulation purposes, you could include a behavioral model that stimulates the sensor inputs in a predefined test pattern.

The major difference between the structural and behavioral descriptions of the MUX is that the architecture body in Figure 2-10 contains a process statement. The process statement describes a single, independent process that defines the behavior of a hardware design or design portion. The basic format of a process statement is shown as follows:

```
process statement .............   label :
                                  process ( sensitivity_list )
                                    process_declarative_part
                                  begin
                                    process_statement_part
                                  end process label ;
```

The process statement in Figure 2-10 begins with the process label `f1` followed by a colon (line 7). The process label is optional but is useful to help differentiate this process from other processes in a larger design.

Following the reserved word **process** is an optional sensitivity list (located between the parentheses). The sensitivity list in Figure 2-10 (line 8) consists of the signal names `d0, d1,` and `sel`. During simulation, whenever a signal in the sensitivity list changes state, the statements in that process are executed. In the MUX example, whenever `d0, d1,` or `sel` changes state, process `f1` is executed and the state of the output signal is changed accordingly. Each process in a VHDL design description is executed once during initialization of the VHDL model.

The heart of the process statement in Figure 2-10 is the ’if’ statement that is contained in the process statement part. The basic format of an if statement is shown as follows:

if statement	**if** condition **then** sequence_of_statements **elsif** condition **then** sequence_of_statements **else** sequence_of_statements **end if ;**

The VHDL if statement is interpreted similarly to an English sentence. For example, look at the following sentence:

> **If** the traffic light is green, **then** proceed across the intersection or **else** (if the traffic light is not green) remain stopped.

The sentence has a condition that must be satisfied (If the traffic light is green) before the command (proceed across the intersection) is executed. The "else" part of the sentence gives the alternative command (or else remain stopped) if the condition is not satisfied.

The if statement in Figure 2-10 (lines 10 through 14) can be rewritten as the following sentence:

> **If** signal `sel` (select) is equal to `0`, **then** assign the value of the waveform on signal `d1` to target signal `q` or **else** assign the value of the waveform on signal `d0` to target signal `q`.

Once the if condition or the else condition in this example is satisfied (`sel = '0'` or its opposite where `sel` does not equal `'0'`), target signal `q` is modified according to the appropriate signal assignment statement. The basic format of a signal assignment statement is as follows:

signal assignment statement: target **<= transport** waveform **;**
(Note that **transport** is optional.)

The following is the first signal assignment statement in Figure 2-10:

```
11         q <= d1;
```

This statement assigns the waveform on signal `d1` to target signal `q`. The optional reserved word **transport*** is not used in this example. The signal assignment delimiter consists of the two adjacent special characters <=, also called a compound delimiter. The following is the second signal assignment statement in this example. This statement assigns the waveform of signal `d0` to the target signal `q`:

```
13         q <= d0;
```

Another use of the compound delimiter <= is as the relational operator "less than or equal to" in conditions such as the following:

```
IF z <= '1' THEN
```

The other relational operators are shown in Table 5-1 on page 5-11.

In summary, the signal assignment delimiter <= is used to assign the value on the right side of the delimiter to the target on the left side. The same compound delimiter <= is used as the relational operator "less than or equal to" in test conditions such as the if statement. How this delimiter is used in context determines whether it is a signal assignment delimiter or a relational operator.

Figure 2-11 shows a VHDL behavioral description of a four-bit shifter. To see how accurate and succinct the VHDL description is, compare it with the following textual description:

```
The four-bit shifter has four input data lines, four output
data lines, and two control lines.  When both control lines
are low, the input levels are passed directly to the
corresponding output.  When control line 0 is high and
control line 1 is low, output line 0 is low; input line 0
is passed to output line 1; input line 1 is passed to
output line 2; and input line 2 is passed to output line 3.
When control line 0 is low and control line 1 is high,
input line 1 is passed to output line 0; input line 2 is
passed to output line 1; input line 3 is passed to output
line 2; and output line 3 is low.  When both control lines
are high, input line 0 is passed to both output line 0 and
line 1; input line 1 is passed to output line 2; and input
line 2 is passed to output line 3.
```

*Refer to the Glossary entry for the reserved word **transport** for further information.

```
 1   ENTITY shifter IS               -- entity declaration
 2     PORT ( shftin  : IN  bit_vector(0 TO 3);   --port clause
 3            shftout : OUT bit_vector(0 TO 3);
 4            shftctl : IN  bit_vector(0 TO 1) );
 5   END shifter;
 6
 7   ARCHITECTURE behav OF shifter IS  -- architecture body
 8   BEGIN
 9     f2:                                -- process statement
10     PROCESS (shftin, shftctl)
11      VARIABLE shifted : bit_vector(0 TO 3);--proc. decl. part
12     BEGIN
13       CASE shftctl IS                      --proc. stmnt part
14         WHEN "00" => shifted := shftin;
15         WHEN "01" => shifted := shftin(1 TO 3) & ’0’;
16         WHEN "10" => shifted := ’0’ & shftin(0 TO 2);
17         WHEN "11" => shifted := shftin(0) & shftin(0 TO 2);
18       END CASE;
19       shftout <= shifted AFTER 10 ns;
20     END PROCESS f2;
21   END behav;
```

Figure 2-11. Code Example of Behavioral Description for a Shifter

The port clause in Figure 2-11 (lines 2 through 4) identifies the input ports as `shftin` (shifter data in) and `shftctl` (shifter control) and the output port as `shftout` (shifter data out). This port clause defines the input and output ports as an array of bits using the predefined type `bit_vector`. Types are described on page 3-1.

The arrays can be compared with containers that have labeled compartments for data storage as shown in Figure 2-12. For example, the array named `shftin` has four elements referred to as `shftin(0)`, `shftin(1)`, `shftin(2)`, and `shftin(3)`. Each element is a storage area for data; in this case, they are storage areas for bit information.

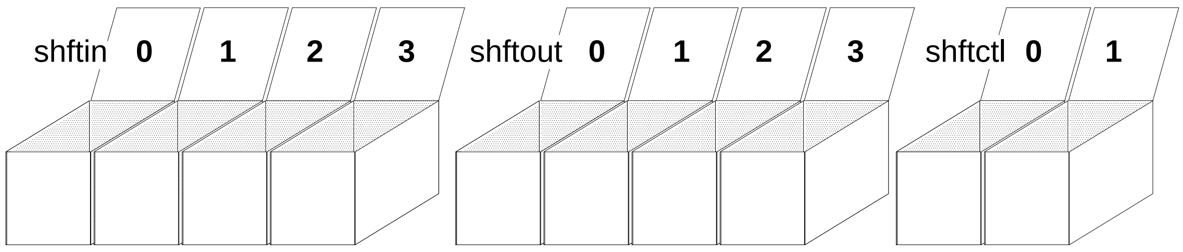


Figure 2-12. Arrays Represented as Data-Storage Containers

The architecture body in Figure 2-11 contains a process statement (lines 9 through 20) as does the previous MUX behavioral example. One difference between the two examples is that the process statement in the shifter example contains a process declarative part (line 11) composed of a variable declaration. A variable declaration has the following format:

variable declaration .... **variable** identifier_list **:**
subtype_indication **:=** expression **;**

The variable declaration in Figure 2-11 does not include the optional "**:=** expression" part. The variable `shifted` holds the shifted value of the `shftin` bit vector. It is important that `shftin` and `shifted` are of the same type, in this case, an array of bits with four elements.

The variable declaration states that `shifted` is an array of bits from 0 to 3. The bit vector `shifted` appears later (after the case statement) in the signal assignment:

```
  19    shftout <= shifted AFTER 10 ns;
```

This signal assignment statement includes the reserved word **after** to specify the propagation time expected for the `shftin` array of waveforms (stored in the variable array `shifted`) to reach the `shftout` array of target signals. Figure 2-13 shows how the elements in array `shifted` map one-for-one to the elements in array `shftout`. The 10 ns delay is represented by a timer that determines the time when the waveforms are transferred. Labels S0 through S3 represent the values that were stored in `shftin` and then passed to the variable `shifted`.

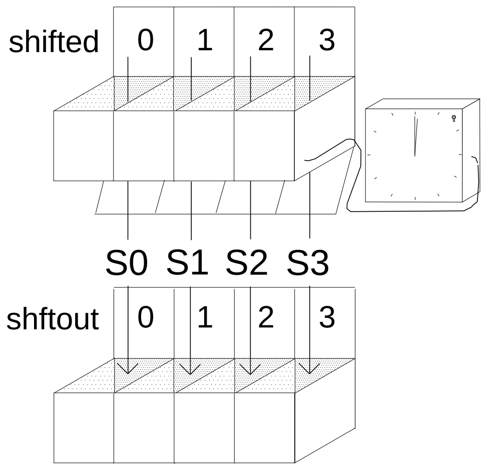


Figure 2-13. Variable Assignment for SHFTOUT Array After 10 ns

The previous MUX behavioral example uses an if statement to assign a waveform to a target signal when a given condition is satisfied. The shifter example uses a case statement to perform a similar function. A case statement executes one out of a number of possible sequences of statements as determined by the value of an associated expression. The basic format of a case statement is shown as follows:

case statement .... **case** expression **is**
when choices => --case stmnt alternative
sequence_of_statements
end case ;

The case statement in Figure 2-11 (lines 13 through 18 in the process statement part) contains four case statement alternatives for the `shftctl` array, `shftctl(0)` and `shftctl(1)`. When one of the alternatives is true, the associated variable assignment statement is executed. A variable assignment statement replaces the current variable value (target) with a new value as specified by an expression. A variable assignment statement has the following format:

variable assignment stmnt target **:=** expression

The characters **:=** are used together as the variable assignment delimiter. To better understand the variable assignment process, consider each of the case statement alternatives from Figure 2-11 one at a time starting with the following:

```
WHEN "00" => shifted := shftin;
```

According to this case statement alternative, when `shftctl(0)` equals 0 and `shftctl(1)` equals 0, then the variable array `shifted` is assigned the values in the array `shftin`. The compound delimiter **'**=>' separates the choices `(WHEN "00")` from the sequence of statements `(shifted := shftin;)`.

The alignment of data in an array is determined by the order in which the array is declared. The port clause defines array `shftctl` as follows:

```
  shftctl : IN bit_vector(0 TO 1)
```

The order of data in the array `shftctl` is 0 to 1 (ascending). Any reference to `shftctl` follows this ordering. Therefore, the first 0 (from the left) in the phrase `WHEN "00"` refers to the state of `shftctl(0)` and the second 0 (from the left) refers to the state of `shftctl(1)`.

The order of data in the arrays `shftin` and `shftout` is defined in the port clause as 0 to 3 (ascending order). The variable `shifted` is defined in the process declarative part as 0 to 3 (ascending).

Figure 2-14 shows how the elements in array `shftin` map one-for-one to the elements in array `shifted` during execution of the following case statement alternative:

```
WHEN "00" => shifted := shftin;
```

Labels V0 through V3 represent the values that are passed from each `shftin` array element.

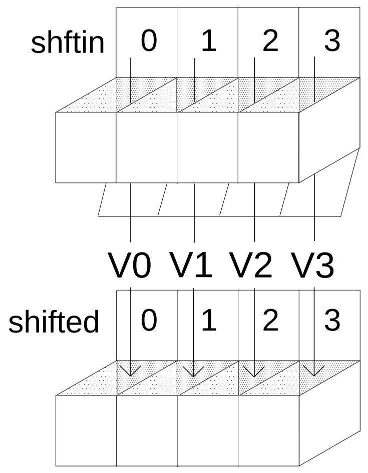


Figure 2-14. Variable Assignment for Array When SHFTCTL = 00

Figure 2-15 shows how the elements in array `shftin` map to the elements in array `shifted` during execution of the following case statement alternative:

```
   15   WHEN "01" => shifted := shftin(1 TO 3) & ’0’;
```

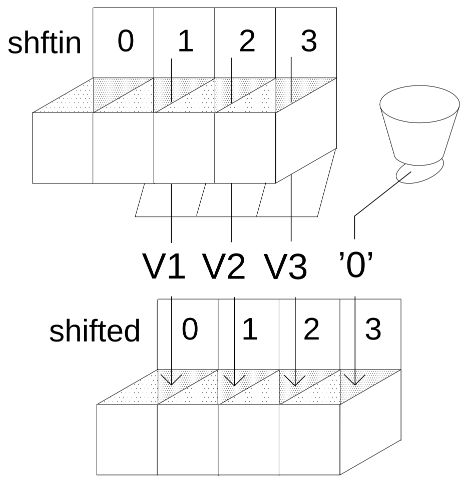


Figure 2-15. Variable Assignment for Array When SHFTCTL = 01

Note that only three elements (1, 2, and 3) of the `shftin` array are transferred to the `shifted` array. The fourth value (’0’) is concatenated to the array`shftin` by using the concatenation operator **&**. The ’0’ is transferred along with the other `shftin` values. For a complete description of the concatenation operator, see the

"Adding Operators" section in the *Mentor Graphics VHDL Reference Manual*.

Figure 2-16 shows how the elements in array `shftin` map to the elements in array `shifted` during execution of the following case statement alternative:

```
  16   WHEN "10" => shifted := ’0’ & shftin(0 TO 2);
```

In this alternative, the value ’0’ is assigned to the first element of array`shifted` (`shifted(0)`) and the values of `shftin(0 to 2)` are concatenated to the ’0’ and assigned to array elements `shifted(1)` through `shifted(3)`.

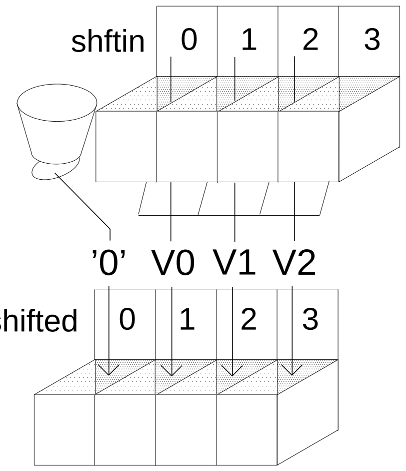


Figure 2-16. Variable Assignment for Array When SHFTCTL = 10

Figure 2-17 shows how the elements in array `shftin` map to the elements in array `shifted` during execution of the following case statement alternative:

```
  17   WHEN "11" => shifted := shftin(0) & shftin(0 to 2);
```

The value of `shftin(0)` is assigned to two elements of array `shifted` (`shifted(0)` and `shifted(1)`).

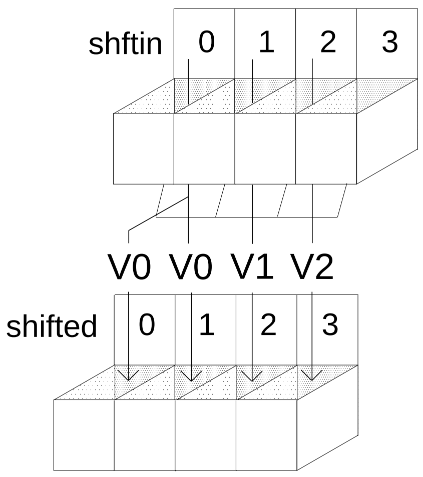


Figure 2-17. Variable Assignment for Array When SHFTCTL = 11

To show further what happens when process `f2` is executed in the VHDL shifter example, three conditions of the `shftctl` array are represented in the waveform drawing of Figure 2-18 as follows:

1. `WHEN "00" => shifted := shftin;`

2. `WHEN "10" => shifted := ’0’ & shftin(0 to 2);`

3. `WHEN "01" => shifted := shftin(1 to 3) & ’0’;`

Each number at the top of the waveform drawing relates to the corresponding condition in the previous numbered list. Arbitrary waveforms (data values) have been assigned to the `shftin` array elements `shftin(3)` to `shftin(0)`. The values of the `shftin` array elements are labeled V3 to V0 for each condition represented. The arrows show how the values flow from the `shftin` array to the `shifted` array (when the `shftctl` signals change state) and then to the `shftout` array 10 ns later.

1. Because `shftctl(1)` and `shftctl(0)` are both low in condition 1, the values on `shftin(3)` to `shftin(0)` (V3 to V0) pass to the corresponding `shifted` array elements. Ten nanoseconds later, the same values pass from the `shifted` array elements to the `shftout` array elements as determined by the conditional signal assignment:

```
  19   shftout <= shifted AFTER 10 ns;
```

A conditional signal assignment is further described on page 2-31.

2. Because `shftctl(1)` is high and `shftctl(0)` remains low in condition 2, the following conditions occur: a low (0) is forced on `shifted(0)`, `shifted(1)` takes on the high value from `shftin(0)`, `shifted(2)` remains high because of the high value from `shftin(1)`, and `shifted(3)` takes on the low value from `shftin(2)`.

3. Because `shftctl(1)` is low and `shftctl(0)` is high in condition 3, the following conditions occur: a low (0) is forced on `shifted(3)` so it remains low, `shifted(0)` takes on the low value from `shftin(1)` so it remains low, `shifted(1)` takes on the low value from `shftin(2)`, and `shifted(2)` takes on the low value from `shftin(3)`.

To provide a complete picture of the four-bit shifter example, the structure is shown in the schematic of Figure 2-19.

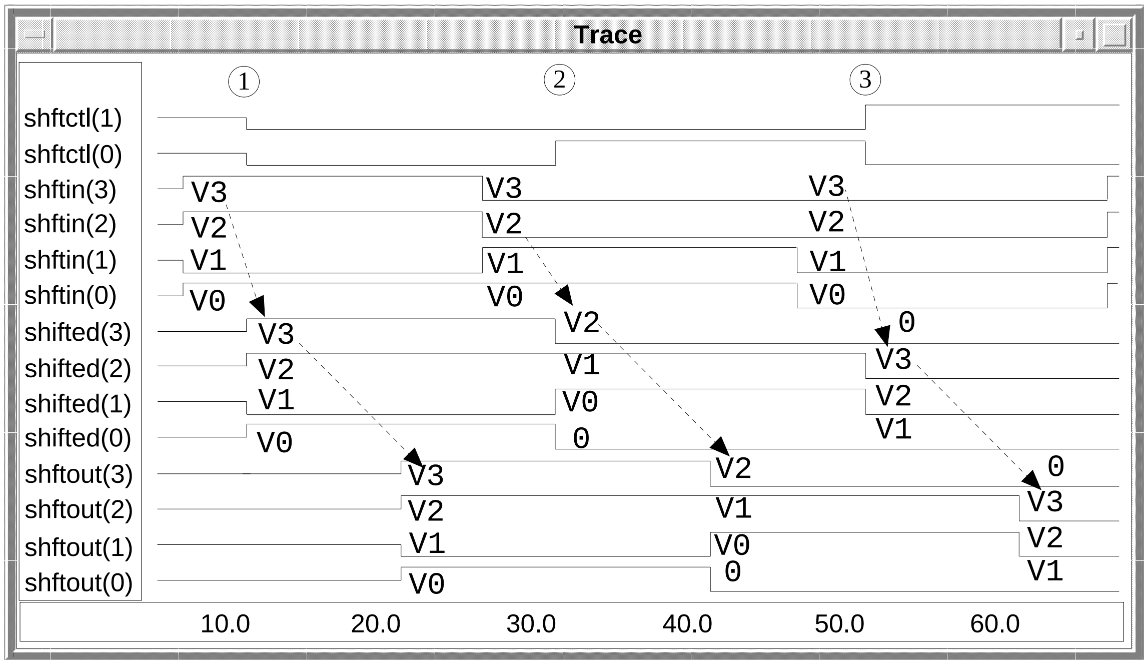


Figure 2-18. Four-Bit Shifter Waveforms

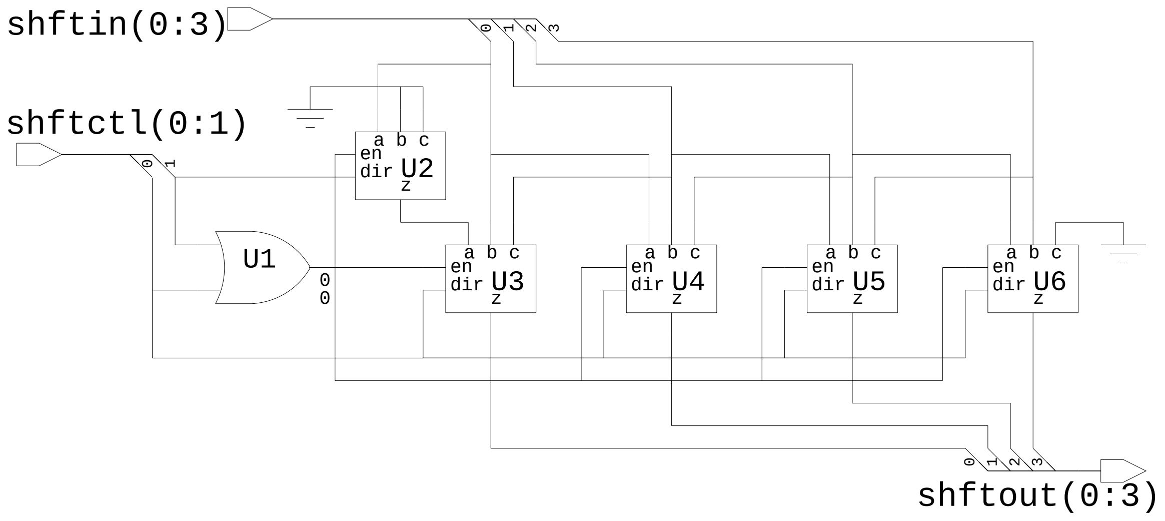


Figure 2-19. Schematic for a Four-Bit Shifter

Structural and Behavioral Description Summary

To summarize the preceding structural and behavioral description methods:

> A VHDL structural description defines the interconnectivity of various components. A behavioral description algorithmically defines circuit and signal response to various stimuli.
>
> A design entity is the basic unit of a hardware description that represents a cell, chip, board, or subsystem. Both the structural and behavioral descriptions declare each design entity with an entity declaration. An associated architecture body describes the relationships between the design entity inputs and outputs.
>
> The structural and behavioral descriptions largely differ in the architecture body, as shown in the comparison of the MUX examples in Figure 2-20. The architecture body of the structural description, as shown in the top part of Figure 2-20, contains an architecture statement part that describes the interconnectivity of the components within the design entity. The architecture body of the behavioral description, shown in the bottom part of Figure 2-20, contains a process statement that describes the behavior of the declared design entity.

If a model contains a signal assignment statement or a concurrent statement that has an associated signal assignment statement, it is *not* a structural description. If a model contains a component instantiation statement, it is *not* a behavioral description.

```
ENTITY mux IS -- STRUCTURAL ------ entity declaration
  PORT (d0, d1, sel: IN bit; q: OUT bit); --port clause
END mux;

ARCHITECTURE struct OF mux IS  -- architecture body
  COMPONENT and2                   --architecture decl. part
    PORT(a, b: IN bit; c: OUT bit);
  END COMPONENT;
  COMPONENT or2
    PORT(a, b: IN bit; c OUT bit);
  END COMPONENT;
  COMPONENT inv
    PORT (a: IN bit; c: OUT bit);
  END COMPONENT;

  SIGNAL aa, ab, nsel: bit;        --signal declaration
  FOR u1    :inv  USE ENTITY WORK.invrt(behav); -- config.
  FOR u2, u3:and2 USE ENTITY WORK.and_gt(dflw); -- specif.
  FOR u4    :or2  USE ENTITY WORK.or_gt(arch1); --

BEGIN
  u1:inv  PORT MAP(sel, nsel);   --architecture statement part
  u2:and2 PORT MAP(nsel,d1,ab);
  u3:and2 PORT MAP(d0, sel,aa);
  u4:or2  PORT MAP(aa, ab, q);
END struct;
-------------------------------------------------------------------
```

```
ENTITY mux IS  -----BEHAVIORAL-------  entity declaration
  PORT (d0, d1, sel: IN bit; q: OUT bit);  --port clause
END mux;
                                --  architecture body
ARCHITECTURE behav OF mux IS
BEGIN
  f1:                            -- process statement
  PROCESS (d0, d1, sel)          -- sensitivity list
  BEGIN
    IF sel = ’0’ THEN              -- process statement part
        q <= d1;
    ELSE
        q <= d0;
    END IF;
  END PROCESS f1;
END behav;
```

Figure 2-20. Comparing Structural and Behavioral Descriptions

Data-Flow Description

The following identifies some of the major language constructs found in a data-flow description using the previous MUX and four-bit shifter examples.

A VHDL data-flow description and a register-transfer language description are similar in that they describe the function of a design by defining the flow of information from one input or register to another register or output.

The data-flow and behavioral descriptions are similar in that both use a process to describe the functionality of a circuit. A behavioral description uses a small number of processes where each process performs a number of sequential signal assignments to multiple signals. In contrast, a data-flow description uses a large number of concurrent signal assignment statements. Concurrent statements used in data-flow descriptions include the following:

- Block statement (used to group one or more concurrent statements)
- Concurrent procedure call
- Concurrent assertion statement
- Concurrent signal assignment statement

In addition to these language constructs, the process statement, generate statement, and component instantiation statement are also concurrent statements. These three additional concurrent statements are not usually found in a data-flow description.

Concurrent statements define interconnected processes and blocks that together describe a design's overall behavior or structure. A concurrent statement executes asynchronously with respect to other concurrent statements. The subsection "Contrasting Concurrent and Sequential Modeling" on page 4-28 provides more information on how concurrent statements execute.

Figure 2-21 can be considered a data-flow description of the same MUX example used in the previous behavioral and structural description examples. This example is too simple to show the usefulness of a data-flow description because it is almost identical to the behavioral description in Figure 2-10 on page 2-15.

Both examples use one process statement (implied with the concurrent signal assignment statement in Figure 2-21, lines 8 through 10) to define signal behavior.

```
 1   ENTITY mux IS                -- entity declaration
 2     PORT (d0, d1, sel: IN bit; q: OUT bit);--port clause
 3   END mux;
 4
 5                                -- architecture body
 6   ARCHITECTURE data_flow OF mux IS
 7   BEGIN
 8     cs1 :                      --concurrent sig. assgnmnt stmnt
 9       q <= d1 WHEN sel = ’0’ ELSE --conditional sig. assgnmnt
10            d0 ;
11   END data_flow;
```

Figure 2-21. Example of Data-Flow Description for a Multiplexer

The data-flow description contains the same entity declaration (lines 1 through 3) used in the previous structural and behavioral description examples. The architecture body contains a concurrent signal assignment statement that represents an equivalent process statement that has the same meaning. The format of a concurrent signal assignment statement is shown as follows:

concurrent signal assignment statement	label **:** conditional_signal_assignment
	-- or
	label **:** selected_signal_assignment

In Figure 2-21 (lines 9 and 10), a conditional signal assignment performs the signal assignments (`q <= d1` or `q <= d0`) based on the conditions defined in the conditional waveform. The format of a conditional signal assignment and the associated conditional waveform is:

conditional signal assignment	target <= options conditional_waveforms **;**
conditional waveforms	waveform **when** condition **else** **--** . . . waveform **when** condition **else** waveform

The conditional signal assignment represents a process statement that uses an if statement in the signal transform. The options (**guarded** and **transport**) are not used in the conditional signal assignment in Figure 2-21.

For comparison, the behavioral description of the four-bit shifter from Figure 2-11 is shown again in Figure 2-22 (at the top of the figure) along with the equivalent data-flow description of the same shifter (at the bottom of the figure).

The major difference between the two descriptions is that four process statements are implied in the data-flow description with the four conditional signal assignments (lines 9 through 20). In the behavioral description, one process statement is explicitly called (lines 9 through 20).

The data-flow example in Figure 2-22 uses the same entity declaration and corresponding port clause as the equivalent behavioral description (lines 1 through 5). The architecture body in the data-flow description uses a concurrent signal assignment statement that is composed of four conditional signal assignments; one for each element of the `shftout` array. This concurrent signal assignment statement does not use the optional label as does the one shown in Figure 2-21, line 8.

```
ENTITY shifter IS --BEHAVIORAL--------- entity declaration
  PORT ( shftin  : IN  bit_vector(0 TO 3);   --port clause
         shftout : OUT bit_vector(0 TO 3);
         shftctl : IN  bit_vector(0 TO 1) );
END shifter;

ARCHITECTURE behav OF shifter IS   -- architecture body
BEGIN
  f2:                                    --process statement
  PROCESS (shftin, shftctl)
    VARIABLE shifted : bit_vector(0 TO 3);--process decl. part
  BEGIN
    CASE shftctl IS                       --proc. statement part
      WHEN "00" => shifted := shftin;
      WHEN "01" => shifted := shftin(1 TO 3) & ’0’;
      WHEN "10" => shifted := ’0’ & shftin(0 to 2);
      WHEN "11" => shifted := shftin(0) & shftin(0 TO 2);
    END CASE;
    shftout <= shifted AFTER 10 ns;
  END PROCESS f2;
END behav;
--------------------------------------------------------------------

ENTITY shifter IS -------DATA-FLOW----------- entity declaration
  PORT ( shftin  : IN  bit_vector(0 TO 3);  -- port clause
         shftout : OUT bit_vector(0 TO 3);
         shftctl : IN  bit_vector(0 TO 1) );
END shifter;

ARCHITECTURE data_flow OF shifter IS -- architecture body
BEGIN                                --concurrent sig. assignment
  shftout(3) <= ’0’       AFTER 10 ns WHEN shftctl = "01" ELSE
                shftin(3) AFTER 10 ns WHEN shftctl = "00" ELSE
                shftin(2) AFTER 10 ns;--end cond. sig. assign. 1
  shftout(2) <= shftin(3) AFTER 10 ns WHEN shftctl = "01" ELSE
                shftin(2) AFTER 10 ns WHEN shftctl = "00" ELSE
                shftin(1) AFTER 10 ns;--end cond. sig. assign. 2
  shftout(1) <= shftin(2) AFTER 10 ns WHEN shftctl = "01" ELSE
                shftin(1) AFTER 10 ns WHEN shftctl = "00" ELSE
                shftin(0) AFTER 10 ns;--end cond. sig. assign. 3
  shftout(0) <= shftin(1) AFTER 10 ns WHEN shftctl = "01" ELSE
                ’0’       AFTER 10 ns WHEN shftctl = "10" ELSE
                shftin(0) AFTER 10 ns;--end cond. sig. assign. 4
END data_flow;
```

Figure 2-22. Comparison of Behavioral and Data-Flow Shifter Descriptions

Constructs Found in Each Design Description Method

The following list itemizes the language constructs and functions found in each type of VHDL design description method. The constructs common to all three methods are listed above the dashed line, followed by the constructs that are unique to a particular description method (below the dashed line).

Constructs Common to Structural, Behavioral, and Data-Flow Methods

Entity declarations	Package declarations	Constant declarations
Architecture bodies	Package bodies	Subtype declarations
Function declarations	Type declarations	Concurrent assertions
Ports	Generics	Signals
Aliases	Attributes	Blocks

- -

Constructs Unique to a Particular Design Description Method

Structural	**Behavioral**	**Data Flow**
Components	Register and bus signals	Register and bus signals
Config. specifications	Concurrent assignments	Concurrent assignments
Config. declarations	Guards	Guards
Generate statement	Disconnection spec.	Disconnection spec.
	Procedure declarations	
	Procedure calls (seq. and concurrent)	
	Sequential statements	
	Process statements	
	Variables	
	Assignments (variable and signal)	
	Dynamic allocation	

Section 3
Foundation for Declaring Objects--Types

This section defines objects, types, and the different type classes provided by VHDL. The section is organized into the following topics:

In code examples in Section 2, objects are declared such as signals `d0`, `d1`, `sel`, and `q` found in the following port clause example (extracted from the code in Figure 2-10):

```
PORT (d0, d1, sel: IN bit; q: OUT );  --port clause
```

Objects are the containers for values of a specified type. Objects are either signals, variables, or constants. Object values are manipulated with a set of operators or subprograms. Each of the objects in the previous port clause example is declared as being of the type named `bit`. The type `bit` is declared in the predefined package called "standard" as having a value of 0 or 1 as shown in the following example:

```
TYPE bit IS ('0', '1'); -- predefined type declaration
```

The types that you define, along with various predefined types, form templates that you use when declaring objects. By declaring the signals `d0`, `d1`, `sel`, and `q` to be of a specific, well defined `bit` type, the hardware designer's intent for these signals (objects) is clearly documented.

Once an object is declared of a certain type, operations can be performed on the object within the bounds set in the type declaration. In the case of the type `bit`, the type declaration specifies that you can set the value of an object of this type to either a ’1’ or a ’0’. If you try to set `q` to 10, an error is generated because the operation result for `q` is outside the bounds of ’0’ and ’1’. The operations performed on the signals declared in Figure 2-10 are as follows:

```
IF sel = ’0’ THEN  q <= d1;
ELSE               q <= d0;
```

When an object is declared to belong to a certain type, it takes on the structure or boundaries set by the type declaration. This characteristic allows you tight control over these objects. If you mix objects of different types or exceed boundaries set by the type declaration, you are notified of an error condition. The format of a type declaration is as follows:

type declaration .... **type** identifier **is** type_definition **;**

Within the type declaration are various classes of type definitions as shown in Figure 3-1. The type definitions are described in the following subsection.

CAUTION

You should not use predefined type identifiers, such as those declared as part of the predefined standard package, in your type declarations. Using these identifiers can make your hardware description very confusing or hard to understand later on. The contents of package "standard" is documented in the Mentor Graphics VHDL Reference Manual.

There may be times when you want to declare an object that uses a subset of values of a given type. In this case, you can define a subtype and associate an object with that subtype. For example:

```
TYPE control_valves IS (on, off, standby, shutdown);
 SUBTYPE off_controls IS control_valves RANGE off TO shutdown;
```

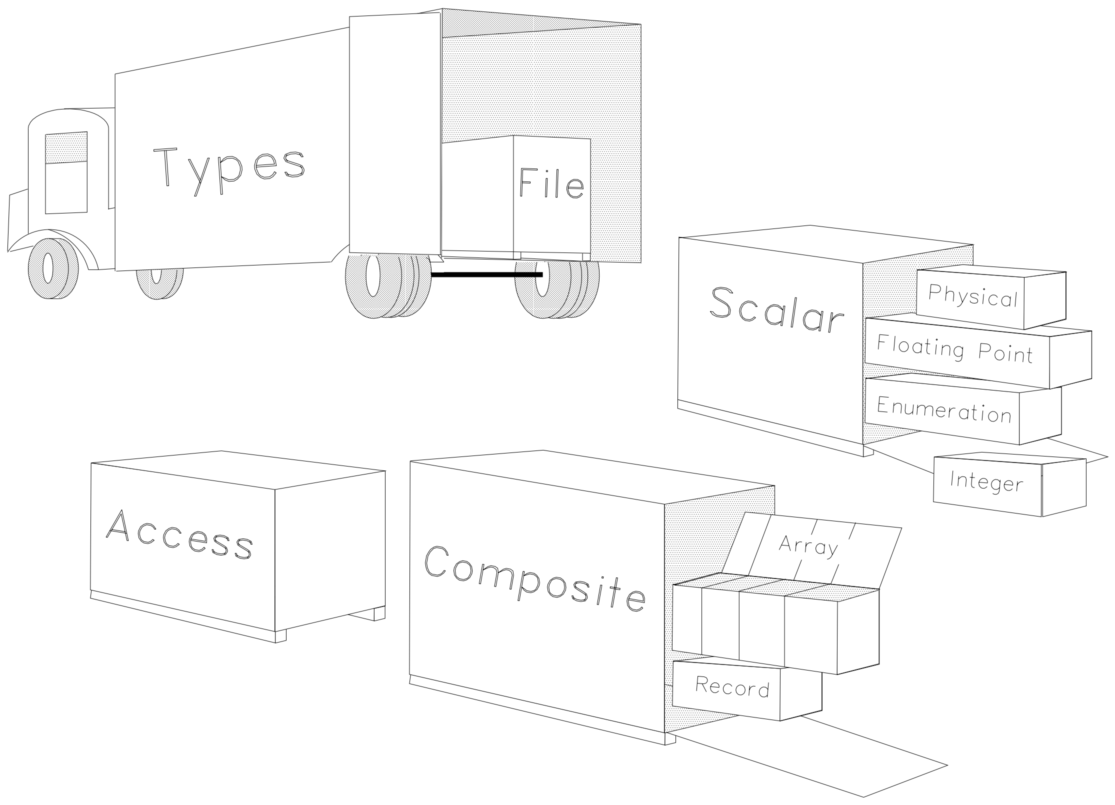


Figure 3-1. Various Classes of Type Definitions Within a Type Declaration

A subtype definition is not a new type; it is a new name for a contiguous subset of the base type. A subtype declaration declares a contiguous subset of values of a specified type and has the following format:

subtype declaration .... **subtype** identifier **is** subtype_indication **;**

The subtype indication identifies restrictions placed on the subtype you declare. Another subtype declaration example follows:

```
TYPE address_size IS RANGE 0 TO 255; --integer type decl.
  SUBTYPE add_piece IS address_size RANGE 0 TO 128;
```

Various Classes of Type Definitions

Within a type declaration, the following classes of type definitions are available. They are explained in the following subsections.

- Scalar type definition
  - ❍ Physical
  - ❍ Floating Point
  - ❍ Enumeration
  - ❍ Integer
- Composite type definition
  - ❍ Array
  - ❍ Record
- File type definition
- Access type definition

Scalar Types

Scalar types (physical, floating point, enumeration, and integer) completely specify an item using an appropriate scale. The following subsections describe each of these types in the order listed.

Physical Types

A physical type describes a quantity measurement of an item. This quantity is expressed in a multiple of the base unit of measurement in a range that you specify. A valid range boundary for most implementations includes integers from -2,147,483,648 to +2,147,483,647. Some implementations can use a range of integers with a 64-bit boundary. The format of a physical type definition is as follows:

```
physical type definition .....   range range
                                  units
                                    identifier ;   --base_unit_declaration
                                    identifier = abstract_literal  name ;
                                  end units
```

In the physical type definition format, the first line includes both the reserved word **range** and a language construct called "range". The following example shows the format for the language construct "range". This construct is also used in other type definitions.

range .... attribute_name
  -- or
simple_expression **to** simple_expression
  -- or
simple_expression **downto** simple_expression

The following example shows a type declaration that includes a physical type definition for an item called `measure`. A range for `measure` is specified from 0 to 1,000,000,000. The base unit declaration (line 3) defines the root measure of `measure` as millimeters. The secondary units are defined in multiples of the base unit (lines 4 and 6) or in multiples of a previously defined secondary unit (line 5).

```
1   TYPE measure IS RANGE 0 TO 1000000000
2     UNITS         --type decl., physical type def.
3        mm;           -- base_unit_declaration:
    millimeter
4        cm = 10 mm;  -- secondary_unit_declaration:
    centimeter
5        dm = 10 cm;  -- secondary_unit_declaration:  decimeter
6        m = 1000 mm; -- secondary_unit_declaration:  meter
7     END UNITS;
```

Once the previous code has been inserted into a given design, you can define objects of type `measure`. The following example shows objects called `w` and `h` declared (as type `measure`) within the process declarative part (line 3). Operations are performed on these objects within the process statement part.

```
1   process_size:            --process label
2   PROCESS (sig1)             --sig1 is declared elsewhere.
3     VARIABLE w, h : measure;  --process declarative part
4   BEGIN
5     w:= (100 cm + 1 m) - 10 mm;--process statement part
6     h:= 20 cm + w;          --w and h results in base unit -
    mm
7     sig2 <= z;        --"sig2" and "z" are declared
    elsewhere.
8    END PROCESS process_size;
```

A predefined physical type called "time" is declared in the standard package. The actual range of type "time" is implementation dependent. To avoid confusion by overriding this predefined type, you should not use "time" as the identifier for your own type declarations .

Floating Point Types

Floating point types define a collection of numbers that provide an approximation to real numbers. It is not possible for hardware to handle an infinitely long real number such as the result of dividing seven by three. In this case you can approximate the real number as 2.33333. The basic format of a floating point definition is shown as follows:

floating type definition ..... **range** range

The following example declares a type called `half_hour`. A range is specified between 0 and 29.99. Once this type declaration is added to a hardware design description, objects can be declared of the type `half_hour`, and floating point operations can be performed on objects of this type.

```
TYPE half_hour IS RANGE 0.0 TO 29.99; --floating point def.
```

If the operation result is not within the range specified in the type definition (as shown in the following example), an error occurs. In this example the object `test_t1` is declared as type `half_hour`. The result of the operation in the process statement part causes object `test_t1` to equal 30.05. This is outside of the range specified in the type declaration so an error is reported.

```
test_time:                 --process label
PROCESS
  test_t1: half_hour;      --process declarative part
BEGIN
  test_t1:= 15.05 + 15.00; --Generates a result
END PROCESS test_time;     --out of range (ERROR)
```

A predefined floating point type called "real" is declared in the standard package as a range of numbers from -1.79769E308 to + 1.79769E308 (platform dependent). To avoid confusion by overriding this predefined type, you should not use "real" as the identifier for your own type declarations .

Enumeration Types

Enumeration types permit you to define a customized set of values. The format of an enumeration type definition is as follows:

enumeration type definition **(**enumeration_literal1**,** ...**,** enumeration_literaln **)**

An enumeration literal can be either an identifier (letters, underscores, and/or digits) or a character literal (made up of a graphic character within single quotes). When you specify a list of enumeration literals (separated by commas) each one has a distinct enumeration value. The first literal listed, on the left, has the predefined position of zero.

An enumeration type you might want to create when modeling a hardware design could be as follows:

```
TYPE wire_color IS (red, black, green); --custom enum. type
```

In the previous example the enumeration literal `red` occupies position zero, `black` occupies position one, and `green` occupies position two. If you create another type declaration that uses the same enumeration literals as a previous declaration, as shown in the following example, the repeated literals are said to be overloaded if they appear in the same area (scope) of code. Overloading is further described on page 5-6.

```
TYPE traffic_light IS (red, yellow, green, flashing);
```

A number of predefined enumeration types are declared in the package called "standard" including the following:

```
TYPE bit IS (’0’, ’1’); --predefined type with char. literals
TYPE boolean IS (FALSE, TRUE);  --predefined enumeration type
```

In addition to those previously shown predefined types, the enumeration types called "character" and "severity_level" are also predefined in the standard package but are not shown here.

Integer Types

Integer types are sets of positive and negative (including zero) whole numbers that you define. On a 32-bit (two's complement) system, any range of integers you define must be between -2,147,483,648 and +2,147,483,647. Integer types include values of an infinitely larger type called universal integers. Universal integers are an unbounded, anonymous type that represents all possible integer literals. Anonymous types are those that cannot be referred to directly because they have no name. The concept of universal integers is used to derive a definition of an integer type. The following shows the integer type definition format:

integer type definition ...... **range** range

The range bounds you specify must be expression that can be evaluated during the current design unit analysis (locally static expression), and they must be integer types. Following are two examples of illegal range bounds to illustrate these points. The first example shows an illegal range bound with one bound (black) not being of an integer type (it is an enumerated type).

```
 TYPE test_int IS RANGE 0 TO black; --black is illegal range
```

The second illegal example is shown as follows to illustrate how a range bound must be a locally static expression. Assume that a package called "external" contains a deferred constant declaration called `ext_val`. Now in a separate design unit (using "external"), you try to declare the following integer type:

```
 TYPE test_integer2 IS RANGE 0 TO ext_val; --illegal boundary
```

The variable `ext_var` is an integer, but it cannot be evaluated in the current design unit, therefore an error results. If `ext_var` had been defined within the same design unit as the `test_int` type declaration, `ext_var` could have been evaluated and the boundary would be legal.

In addition to those already described, there is another consideration when declaring and using integer types. The result of any operation performed on an integer must be within the range boundary set in the integer definition. (A similar situation is described in the "Floating Point Types" subsection starting on page 3-6.

There is one predefined integer type definition contained in the standard package. It has the following format in a 32-bit system (platform dependent):

```
TYPE integer IS RANGE -2147483648 TO +2147483647;
```

Composite Types

Composite types allow you to specify groups of values under a single identifier. The composite types available in VHDL are array types and record types, as explained in the following subsections.

Array Types

A named array is a collection of elements that are of the same type. Arrays may be configured in one or more dimensions. Each array element is referenced by one or more index values, depending whether it is a single or a multiple dimension array. See Figure 2-11 on page 2-19 for an example that shows how objects are declared as one dimensional arrays of bits.

An array definition can be one of two kinds: an unconstrained array definition or a constrained array definition. As shown in the following example, a constrained array has a specified index constraint that specifies the number of array elements.

```
TYPE arr1 IS ARRAY (0 TO 4) OF integer; --const. array def.
```

The index constraint in the `arr1` declaration is `(0 TO 4)`. Any object declared to be of type `arr1` is an array with five elements as shown in Figure 3-2. In the `arr1` declaration, the type "integer" is used as the subtype indication, which means each element of the array must be of type "integer". The general format of a constrained array definition is shown as follows:

constrained array definition	**array** index_constraint **of** subtype_indication

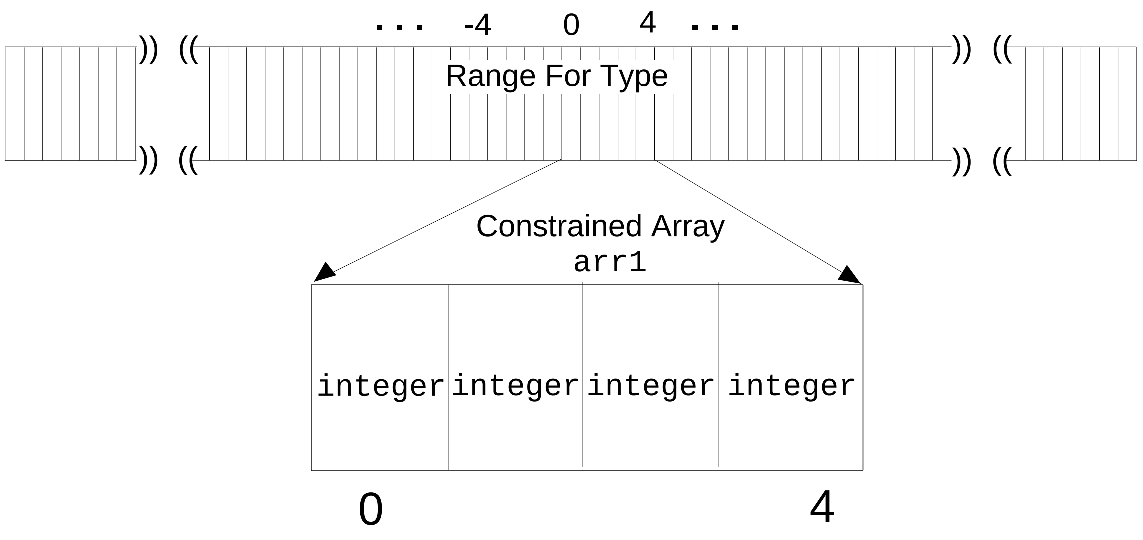


Figure 3-2. Defining a Constrained Array Type

An example of a two dimensional array definition and related type declarations is shown in the following code example, and the corresponding array structure is shown in Figure 3-3. A specific type (enumeration) is defined for each dimension of the array and is followed by an array definition to specify the structure for a type called `mtrx`. All elements within this array structure are of the type `integer`. An object can now be declared that has the `mtrx` structure.

```
TYPE index_acrs IS ('a','b','c','d');  --enumeration def.
TYPE index_dwn IS ('e','f','g');       --enumeration def.
TYPE mtrx IS ARRAY (index_dwn  RANGE 'e' TO 'g',
                    index_acrs RANGE 'a' TO 'd') OF integer;
                    --constrained, two-dimensional array
```

A type declaration for an unconstrained array is different from a constrained array, in that the unconstrained array has no specification of the index. This means that you can define an array type whose number of elements is not known as shown in the second line of the following example:

```
TYPE mem_arr IS ARRAY (0 TO 1023) OF integer;   --constrained
TYPE arr2 IS ARRAY (integer RANGE <>) OF mem_arr; --unconstrn
```

The first line of the example defines a constrained array type `mem_arr` to create an array structure with 1024 elements (0 to 1023). The second line defines unconstrained array `arr2` to have elements of the type `mem_arr` but without any specific index boundaries. Each element of `arr2` is an array structure with 1024 elements of type integer.

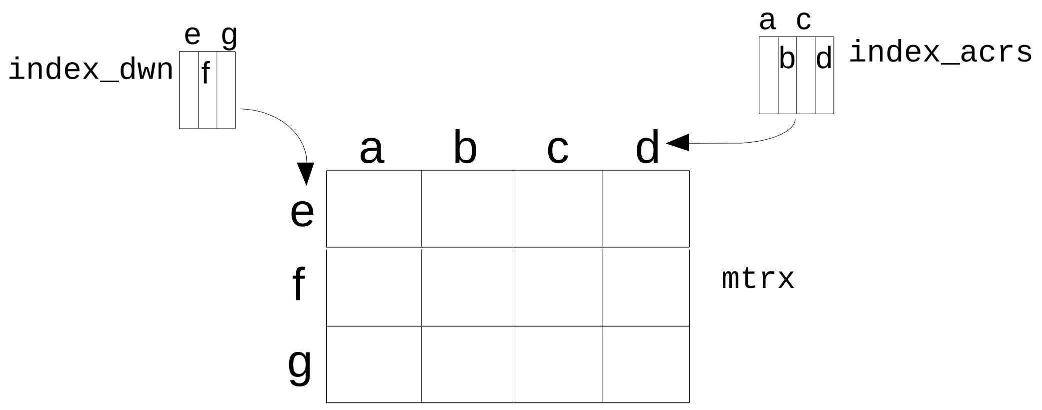


Figure 3-3. Defining a Constrained Array Matrix

The phrase `RANGE <>` (range box) indicates that the `arr2` index can range over any interval allowed in the index subtype `integer`, as shown in Figure 3-4.

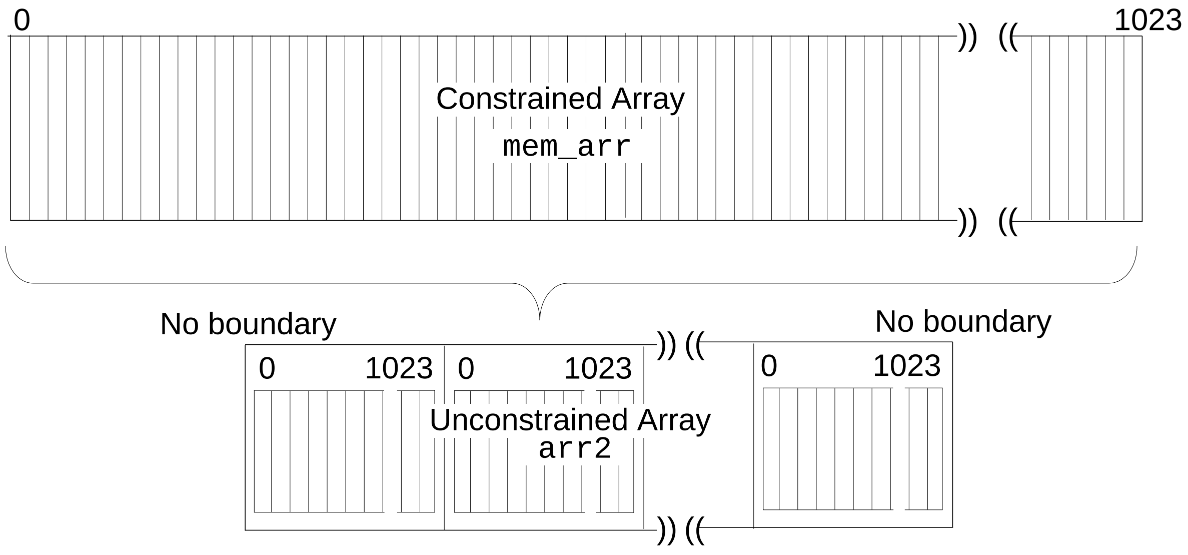


Figure 3-4. Defining an Unconstrained Array with Array Elements

The format of an unconstrained array definition is shown as follows:

unconstrained array definition	**array (** type_mark **range <>) of** subtype_indication

Record Types

A record is a composite type whose elements can be of various types. The purpose of a record is to group together objects of different types that can then be operated on as a single object.

The record type definition defines a particular record type. It has the following format:

record type definition	**record** identifier_list **:** element_subtype_definition . . . **end record**

The record type definition contains a series of element declarations, each of which contains one or more element identifiers and a subtype indication for those elements. All the element identifiers must be unique. The following example shows two record type definitions.

```
TYPE coordinates IS RECORD
   xvalue,yvalue : integer;
END RECORD;

TYPE half_day IS (am, pm);
TYPE clock_time IS RECORD
   hour : integer RANGE 1 TO 12;
   minute, second : integer RANGE 1 TO 60;
   ampm : half_day;
END RECORD;
```

You can read from and assign data to individual elements of a record. To access an individual record element, you use the selected-name construct, as shown in the following examples:

```
VARIABLE time_of_day : clock_time;
. . .
time_of_day.minute := 35; -- loads 35 into element "minute"

start_hour := time_of_day.hour; -- assigns value of element
                                -- "hour" to "start_hour"
```

When assigning values to or reading from record elements, the types of the record elements must match the types of the variables; otherwise, an error occurs. You can also access a record as an aggregate, in which case all the elements are assigned at once, as shown in the following example:

```
VARIABLE time_of_day : clock_time;
. . .
time_of_day := (12, 05, 23, am);
```

File Types

This subsection introduces the concept of file types but does not go into specific details because it is beyond the scope of this manual. For more information on file types, refer to the *Mentor Graphics VHDL Reference Manual.*

File types allow you to declare external files that contain objects of the type you specify. External files refer to those that are external to the main hardware description model file. The following example shows the format of a file type definition:

file type definition .... **file of** type_mark

Access Types

This subsection introduces the concept of access types but does not go into specific details because it is beyond the scope of this manual. For more information on file types, refer to the *Mentor Graphics VHDL Reference Manual.*

Access types let you designate objects that are variable. The following example shows the format of an access type definition:

access type definition ... **access** subtype_indication

Retrieving Information on Certain Kinds of Objects

Once an object has been declared, you might want to retrieve some information from the object and use the result in an operation or test condition. VHDL provides a number of predefined attributes that can examine certain parameters of any one of the following kinds of objects:

- Arrays
- Blocks
- Signals (scalar or composite)
- Types (scalar, composite, or file)

This description of predefined attributes serves only as an overview. Refer to the *Mentor Graphics VHDL Reference Manual* for a complete discussion of all predefined attributes. The following is an example format of an attribute name, which is used with both user-defined and predefined attributes to denote a value, function, type, range, signal, or constant associated with a design entity:

attribute name .... prefix**’***attribute*_simple_name

You use the apostrophe (’) character followed by an attribute identifier to designate a particular attribute. The prefix is the object name or function call that the attribute will check.

The following example uses a standard predefined attribute (’event) in the condition in line 10.

```
 1   LIBRARY my_lib; USE my_lib.my_qsim_logic.ALL;
 2   ENTITY incomplete_counter IS
 3      PORT (clock, data:  IN my_qsim_state;
 4                  q_out: INOUT my_qsim_state);
 5   END incomplete_counter ;
 6
 7
 8   ARCHITECTURE behav OF incomplete_counter IS
 9   BEGIN
10     q_out <= data WHEN clock'event AND clock = '1' ELSE
11               q_out;
12   END behav;
```

Attribute `'event` (commonly pronounced "tic event") in line 10 is a signal attribute. The attribute checks the signal `clock` and returns a Boolean value of TRUE when there is an event on `clock`, or a value of FALSE if there is no event on `clock`. Figure 3-5 illustrates the returned values for `clock'event` in relation to the `clock` signal. Also shown in the figure is the returned values for the entire condition `clock'event AND clock = '1'`. When the condition in line 10 of the previous code example is true, a rising clock pulse has occurred so the value of `data` is assigned to `q_out`.

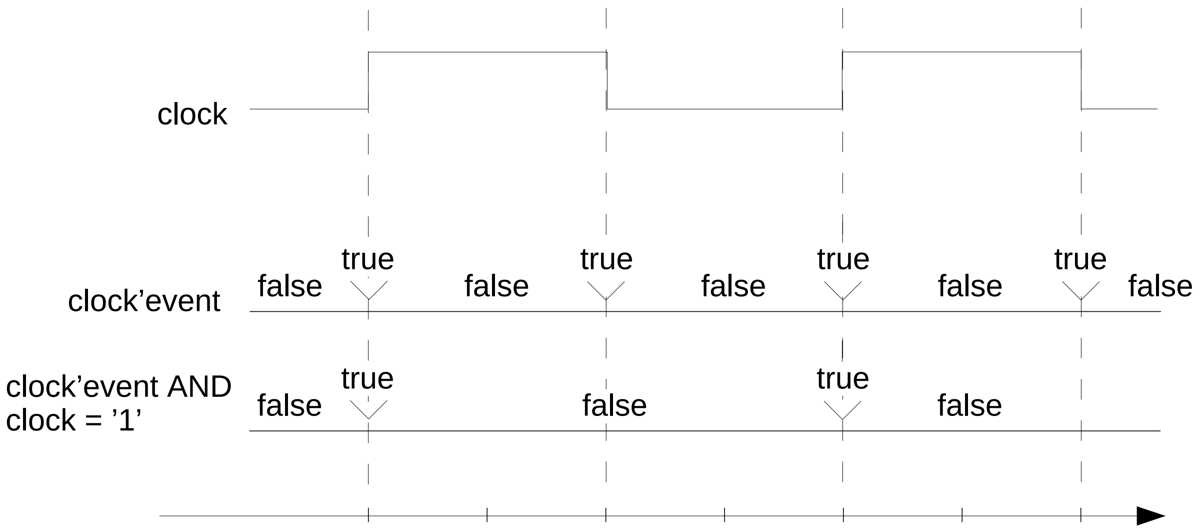


Figure 3-5. Signal Attribute Example

The following example shows two predefined attributes, ’left and ’right, that operate on types. Type high_byte is declared in line 1 to have a range from 28 to 31. The bound values 28 and 31 are used in the loop parameter specification in line 3 by using the ’left and ’right attributes. Attribute ’left returns the high_byte bound value 28 and ’right returns the bound value 31.

```
1   TYPE high_byte IS RANGE 28 TO 31;
2
3   FOR i IN high_byte’left TO high_byte’right LOOP
4       --something happens
5   END LOOP;
```

The previous example is used only to show the function of ’left and ’right. VHDL provides a more efficient method of extracting the range of type high_byte for a loop range as shown in the following code:

```
3  FOR i IN high_byte LOOP --The range is 28 to 31
```

The following example shows another predefined attribute, ’pos(x), that operates on a type. Attribute ’pos(x) returns an integer value that equals the position of the item that you define by supplying a parameter (x) to the attribute.

In line 1 of the following example, an enumerated type called opcode is declared. The opcode type contains nine instruction mnemonics. The comment in line 2 shows the positional location of each mnemonic. Line 4 declares constant stop to be an integer that is assigned a value related to the position of the corresponding halt (hlt) mnemonic. In other words, the result of opcode’pos(hlt) (which equals 8) is assigned to constant stop.

```
1   TYPE opcode IS (mov,lda,sta,jmp,ret,add,sub,nop,hlt);
2      --positions   0   1   2   3   4   5   6   7   8
3
4   CONSTANT Stop : integer := opcode’pos(hlt);  -- stop = 8
```

The previous examples give you a basic look at predefined attributes and how you might use them in your hardware models. Refer to the *Mentor Graphics VHDL Reference Manual* for a complete description of all the predefined attributes and how to create user-defined attributes.

Section 4 Constructs for Decomposing Design Functionality

Many of the VHDL constructs have been previously introduced in the structural, behavioral, and data-flow descriptions. The examples used in those subsections solved relatively simple design problems.

This section identifies a number of VHDL constructs that allow the designer to decompose a complex design into smaller and more manageable modules. This subsection is divided into the following topics:

Consider the principles of concurrent and sequential operation as they relate to hardware simulation. Figure 4-1 shows the execution order for the VHDL statements in a hypothetical design during a given simulation timestep. The term *timestep* is used to denote the smallest time increment of a simulator. Time is shown at the top increasing from left to right, the same as in the trace window of a simulator (or on the screen of an oscilloscope).

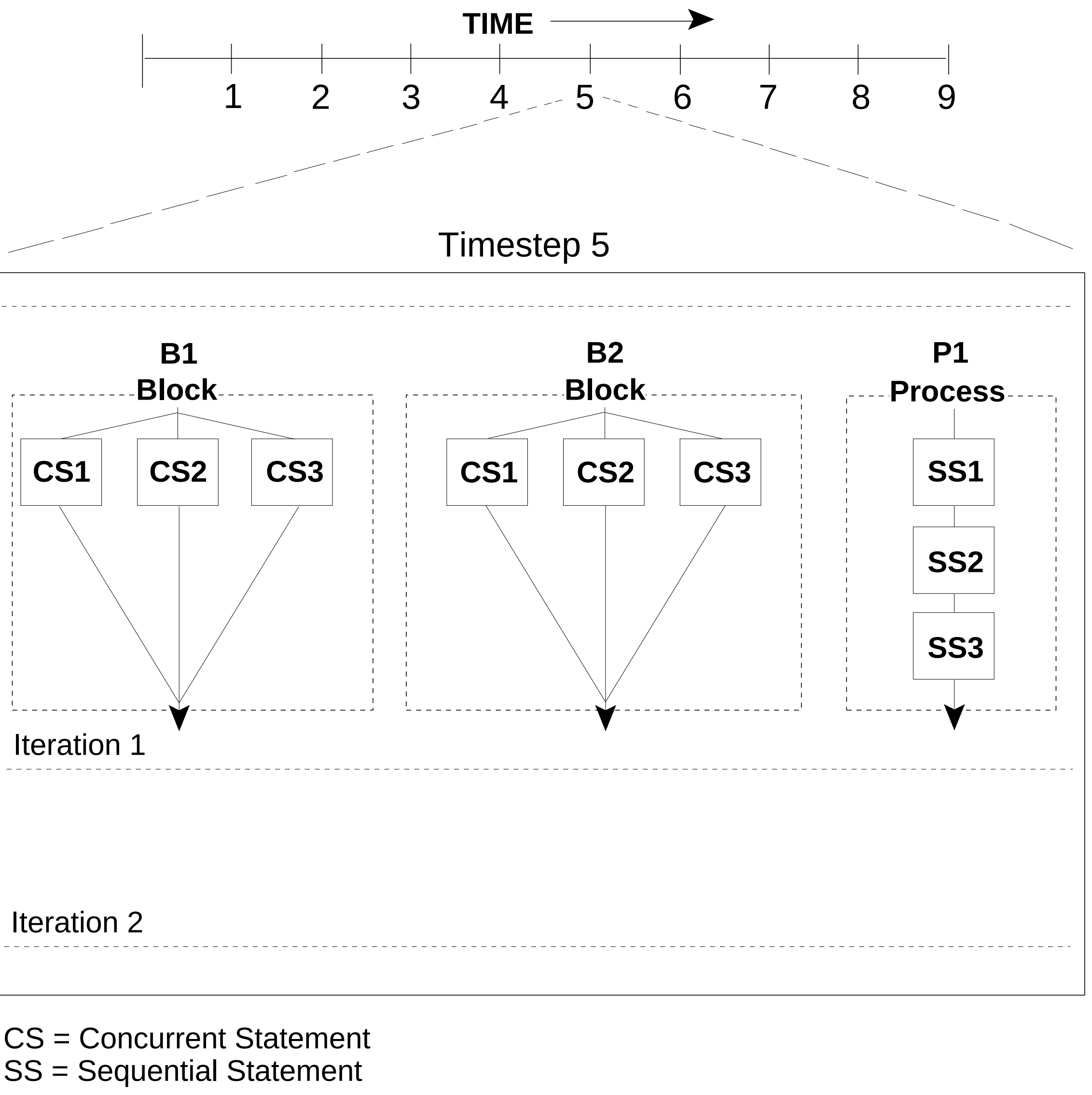


Figure 4-1. Concurrent and Sequential Operations

Most simulators run on systems with a single processor. With just one processor, concurrent processes are not actually evaluated in parallel on the simulator hardware. VHDL uses the concept of delta delay to keep track of processes that should occur in a given timestep but are actually evaluated in different machine cycles. A *delta delay* is a unit of time as far as the simulator hardware is concerned, but as far as the simulation is concerned, time has not advanced.

The processes in the two blocks (B1 and B2) and a process (P1) in Figure 4-1 are scheduled to execute (concurrently) within iteration 1 of timestep 5. The term *iteration* is used to denote a delta delay unit. (This example does not require a second iteration of timestep 5.)

Within blocks B1 and B2 are three concurrent statements that execute in parallel. Within process P1 are three sequential statements. Statement SS1 executes first, followed by SS2, and then SS3. Once all statements have finished execution, the simulator can advance to the next iteration or timestep. As far as the simulator is concerned, all of these operations (processes in B1 and B2, and P1) occur at the same time (during iteration 1 of timestep 5).

Concurrent Decomposition

This subsection identifies the constructs that define hardware functions that execute concurrently. The main focus of this subsection is to describe the following:

- The block statement, which is the primary concurrent statement used to decompose the hardware functionality into smaller modules

- The component instantiation statement, which defines the actual use of declared components

The following list identifies the VHDL statements that execute concurrently within a particular simulation timestep.

- Concurrent signal assignment statement (See Figures 2-21 and 2-22 on pages 2-31 and 2-33, respectively, for examples.)

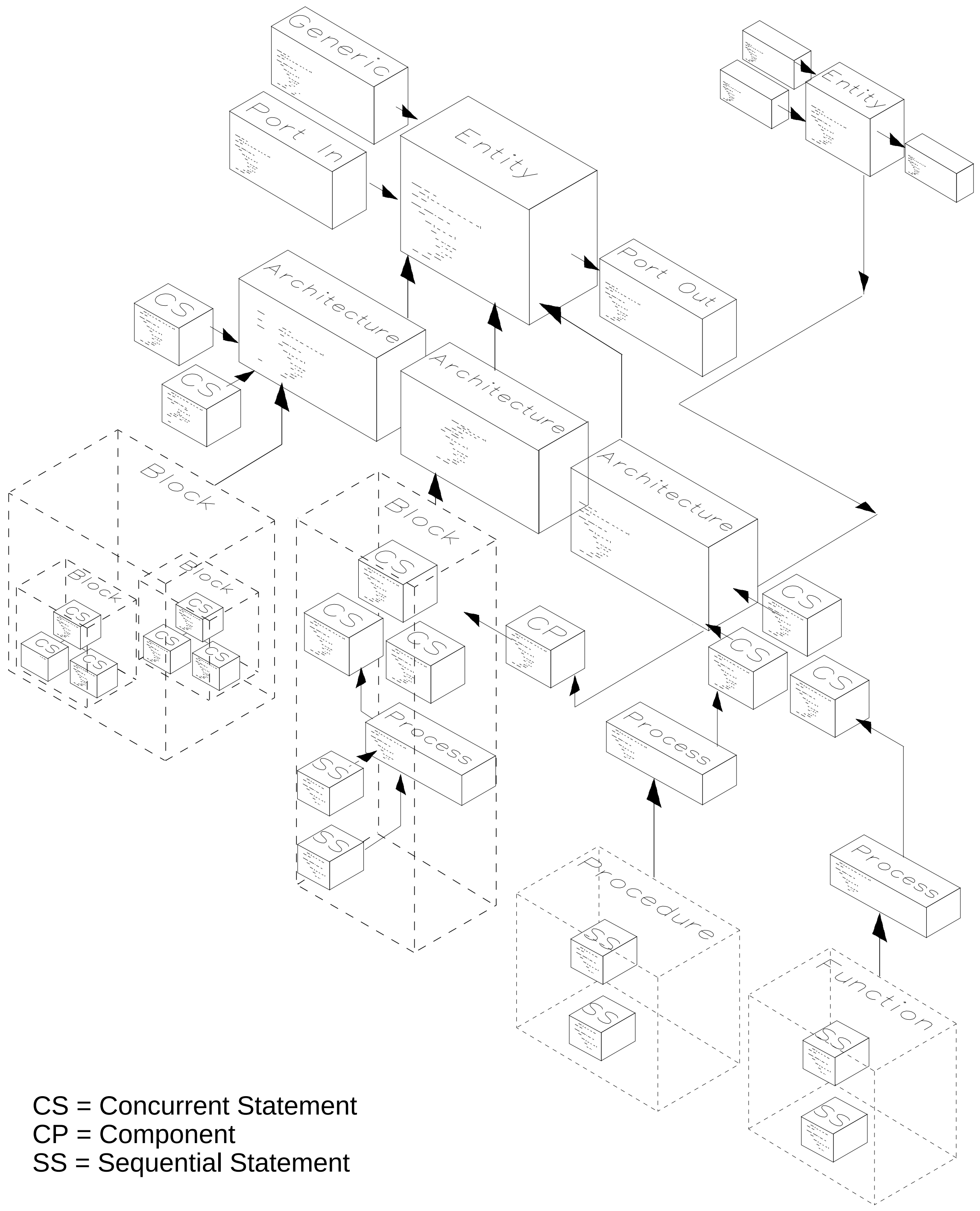
- Process statement (See Figures 2-10 and 2-11 on pages 2-15 and 2-19, respectively, for examples.)
- Concurrent procedure call
- Concurrent assertion statement
- Block statement (used to group concurrent statements)
- Component instantiation statement
- Generate statement

Block Statement

The block statement allows you to group concurrent statements into one logical unit, which describes a portion of your design. Figure 4-2 shows the blocks in the overall VHDL hierarchy highlighted in bold-dashed lines. As can be seen in the figure, blocks can be nested to support the decomposition of the design.

The basic format of a block statement is as follows:

```
block statement ...............     label :
                                    block  ( expression )
                                     block_declarative_item
                                    begin
                                     concurrent_statement
                                    end block label ;
```



CS = Concurrent Statement
CP = Component
SS = Sequential Statement

Figure 4-2. Relating Blocks to the VHDL Hierarchy

You can use the optional guard expression to control the operation of certain statements within a block. To illustrate this point, Figure 4-3 shows the code description and corresponding schematic for a bistable latch using a guard expression.

```
 1   ENTITY bistable_latch IS
 2      PORT (enable, data: IN  bit;
 3               q, q_not : OUT bit );
 4   END bistable_latch;
 5
 6   ARCHITECTURE example OF bistable_latch IS
 7   BEGIN
 8      latch1 :                  -- block label
 9      BLOCK (enable = ’1’)      -- guard expression
10        SIGNAL d_in : bit;       --block decl. item
11
12      BEGIN
13        d_in  <= GUARDED data ; -- guarded sig. assignment
14        q     <= d_in ;
15        q_not <= NOT d_in ;
16      END BLOCK latch1 ;
17   END example ;
```

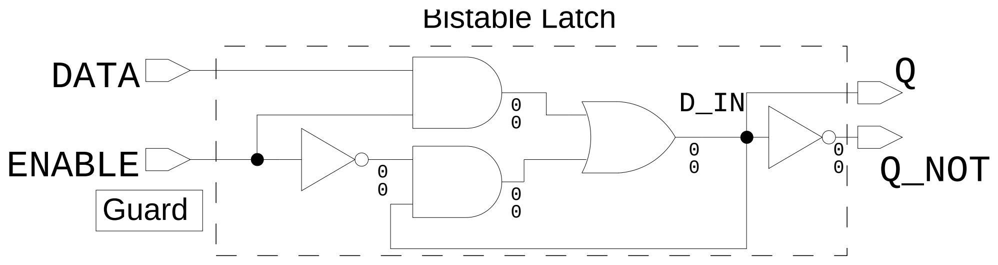


Figure 4-3. Using the Guard Expression on a Bistable Latch

The guard expression `(enable = ’1’)` in line 9 causes all guarded signal assignments within the block to execute only when the guard expression is true. In the case of the bistable latch, the enable signal must equal a 1 before the guarded signal assignment `d_in <= GUARDED data` in line 13 will execute. In other words, the `d_in` signal does not take on the value of the `data` signal until the `enable` signal is at a 1 value. (The `data` value is latched until the `enable` signal goes high).

If `d_in` is not updated with a new `data` value during a given timestep, the signals `q` and `q_not` are assigned the value of `d_in`, which has not been scheduled to change in the current timestep.

Two predefined block attributes (’behavior and ’structure) are available so you can check if a block is a behavioral or structural description. The expression block_label’behavior returns a Boolean value of TRUE if the block *does not* contain a component instantiation statement. The expression block_label’structure returns a Boolean value of TRUE if the block *does not* contain a signal assignment statement or a concurrent statement that contains a signal assignment statement.

If the following code was inserted between lines 16 and 17 of Figure 4-3, the message "`Block latch1 is not a structural description`" would appear from the assertion statements during simulation.

```
16a ASSERT latch1’behavior  --If condition false, show report
16b   REPORT "Block latch1 is not a behavioral description";
16c     SEVERITY note;
16d ASSERT latch1’structure --If condition false, show report
16e   REPORT "Block latch1 is not a structural description";
16f     SEVERITY note;
```

The assert statement generates the associated report if the condition is false. In line 16a, the condition `latch1’behavior` is true so the report in line 16b is not generated. In line 16d, the condition `latch1’structure` is false so the report in line 16e is generated.

Component Instantiation

The language constructs for components are unique to the structural description method. (Components are not used in pure behavioral or data-flow descriptions.) The component declaration and the component instantiation statement are described in this subsection along with the generic clause and port clause.

Before a component is instantiated within an architecture body, you must declare it. Component declarations are used within an architecture body to describe the components to be connected within a given design entity. A component declaration has the following format:

component declaration ...	**component** identifier generic_clause port_clause **end component ;**

Once a component has been declared and specified, you use the component instantiation statement to describe each specific instance of the component and to map the signals on each instance to the ports identified in the declaration. The following format of a component instantiation statement is followed by the format of a generic map aspect, port map aspect, and association list:

component instantiation statement	label **:** name generic_map_aspect port_map_aspect **;**
generic map aspect	**generic map** **(** association_list **)**
port map aspect	**port map** **(** association_list **)**
association list	association_element **,** association_element ...

Figure 4-4 repeats the MUX structural code example from Figure 2-8 but shades the areas that do *not* pertain to this description. The figure shows the areas of code that pertain to the declaration, specification, and instantiation of an AND gate.

The MUX ports (`d0`, `d1`, `sel`, and `q`) are defined in the entity declaration port clause (line 3). These ports (called formal ports) define external connections to the MUX design entity. Subsequent code describes the connection of internal components to these formal ports as necessary.

The AND gate `and2` is declared (component declaration) in the architecture body (lines 8 through 10). The AND gate symbol on the right graphically shows the port configuration of this internal component. These ports (called local ports) are local to the component declaration within the `mux` architecture body.

Further down in the architecture body of Figure 4-4 is a signal declaration statement (line 18) that defines signals `aa`, `ab`, and `nsel`. These signals allow the interconnection of the various components within the `mux` architecture body.

The configuration specification in line 21 binds the separate `u2` and `u3` instances to a particular design entity (`and_gt`) and corresponding architecture body (`dflw`) that describes the behavior of the AND gate. It is possible to bind a different architecture (or entity declaration and architecture body) to `u3` and leave `u2` bound to the `dflw` architecture of the `and_gt` design entity.

The component instantiation statements are located in the architecture statement part (lines 25 through 28). In this example, `u2` and `u3` are two separate instances of the `and2` gate. The port map in each instantiation statement maps the ports to the appropriate signals. In this way, circuit connection is described. The bottom of Figure 4-4 shows the graphic representation of how the AND gates fit into the total MUX description.

```
                       -- entity declaration
ENTITY mux IS
  PORT (d0, d1, sel : IN bit ; q: OUT bit);
END mux;

                        -- architecture body
ARCHITECTURE struct OF mux IS
  COMPONENT and2
    PORT(a, b: IN bit; z: OUT bit);
  END COMPONENT;
  COMPONENT or2
    PORT(a, b: IN bit; z: OUT bit);
  END COMPONENT;
  COMPONENT inv
    PORT (i: IN bit ; z: OUT bit);
  END COMPONENT;

  SIGNAL aa, ab, nsel: bit ;

  FOR u1   :inv  USE ENTITY WORK.invrt(behav);
  FOR u2,u3:and2 USE ENTITY WORK.and_gt(dflw);
  FOR u4   :or2  USE ENTITY WORK.or_gt(arch1);

BEGIN
  u1:inv  PORT MAP(sel, nsel);
  u2:and2 PORT MAP(nsel,d1,ab);
  u3:and2 PORT MAP(d0, sel,aa);
  u4:or2  PORT MAP(aa, ab, q);
END struct;
```

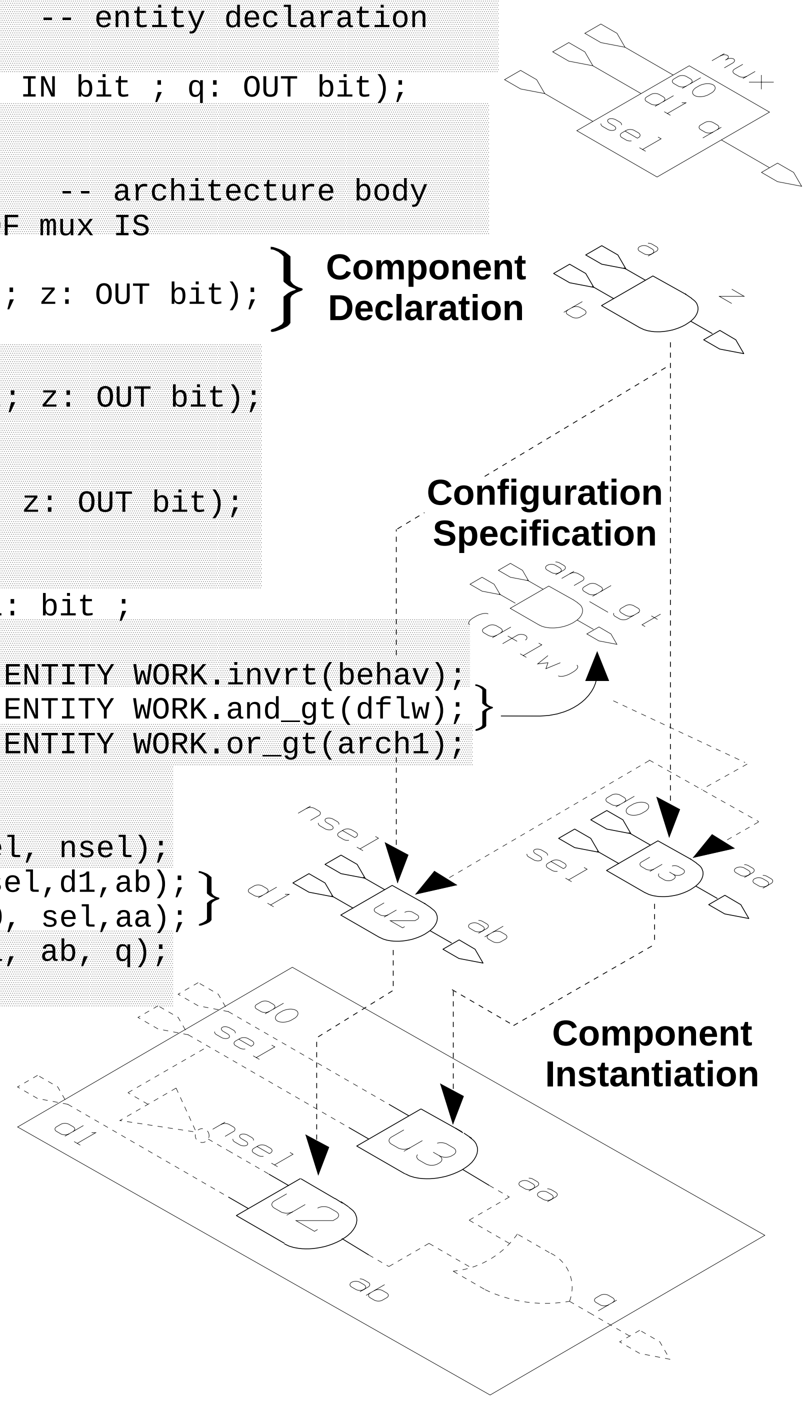


Figure 4-4. Instantiating an AND Gate in a MUX Description

Figure 4-5 shows how a generic clause in an entity declaration allows parameter customization when the design entity is instantiated in a circuit description. The entity declaration `and2` is shown instantiated in a circuit description in three places. The generic clause (`prop_delay : time`) defines a parameter called `prop_delay` (propagation delay) that has a value of a type called `time`. All values passed back to `prop_delay` must be of the same type (in this case, `time`).

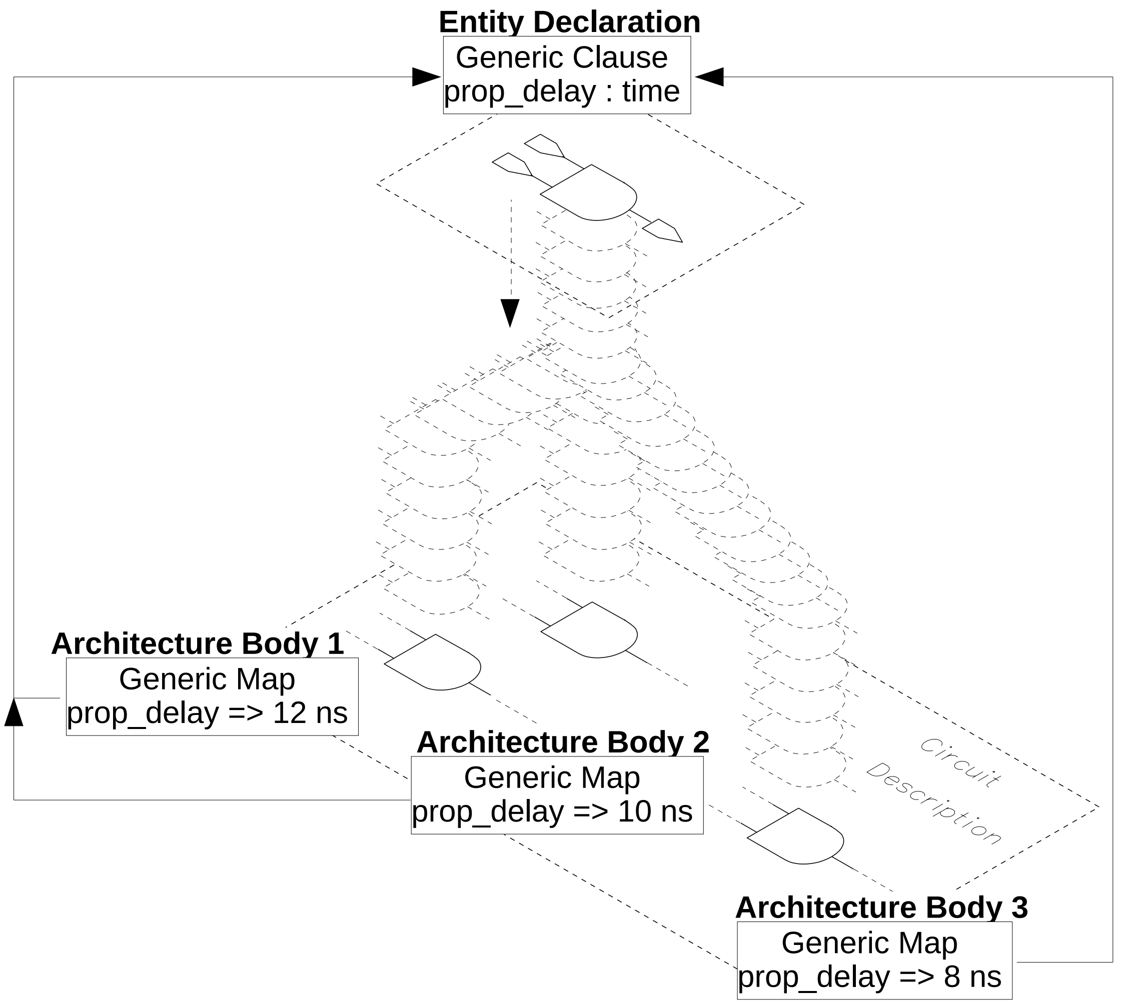


Figure 4-5. Instantiating a Component Using Different Parameters

The actual value for `prop_delay` is passed back into the entity declaration with a generic map in each architecture body. In this example, architecture body 1 passes back a time value of 12 ns, architecture body 2 passes back a time value of 10 ns, and architecture body 3 passes back a time value of 8 ns.

A port map performs a similar function to the generic map. A port map defines which signals within the architecture body correspond with the ports that are defined in the entity declaration or a component declaration.

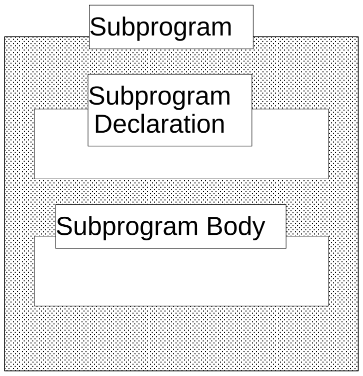
Sequential Decomposition

This subsection identifies the constructs that define hardware functionality that execute sequentially. The main focus of this subsection is to describe the two parts of the subprogram construct: procedures and functions.

The following list identifies the VHDL statements and operations that execute sequentially within a particular timestep.

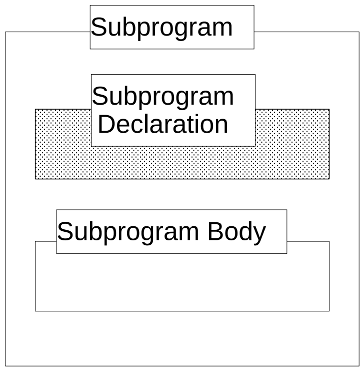
- Control Statements
  - If statement
  - Case statement
  - Wait statement
- Looping statements
  - Loop statement
  - Next statement
  - Exit statement
- Assignments
  - Signal assignments
  - Variable assignments
- Assertion statement
- Return statement
- Null statement
- Subprograms

Subprograms--Functions and Procedures



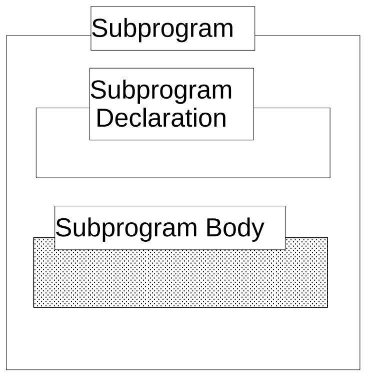
A subprogram allows you to decompose the hardware description into behavioral descriptions or operations using algorithms for computing values. The hardware's high-level activities are declared using a subprogram declaration. The actual operations are implemented in a subprogram body.

Subprograms have two forms: procedures and functions.



The subprogram declaration specifies the interface between the subprogram and the external (to the subprogram) environment for either a procedure or function, as shown in the following example:

procedure designator **(**formal_parameter_list**) ;**
 -- or
function designator **(** formal_parameter_list **)**
 return type_mark **;**



The subprogram body contains an algorithm or behavioral description. It has the basic format as shown in the following example:

procedure designator **(**formal_parameter_list**) is**
-- or
function designator **(** formal_parameter_list **)**
return type_mark **is**
--
subprogram_declarative_part
begin
subprogram_statement_part
end designator **;**

Subprograms are useful for decomposing large systems into smaller modules. In a top-down design approach, you can decompose the solution into its primary functions. At the high abstraction level for a hardware system, you create an architecture body that contains the "what happens" details without including the "how it happens" details. The "how" details can be placed in subprograms (which can be placed in packages) that are called from the system architecture body. This makes the system-level architecture description easier to read and understand.

Because a subprogram can be a procedure or function, you need to decide which one best fits your needs. It is best to use a function when the "how it happens" module returns a single value to the calling routine in the "what happens" section. You should use a procedure either to return multiple values, to return no value, or to effect a change in some remote part of the system (such as change a data value at a RAM location).

Figure 4-6 shows an example* of a system design description that uses three procedures and one function. The large block on the left represents an architecture body that contains the "what happens" design description. From this description, various procedures and a function are called to perform specific actions. For this discussion, assume that the procedures are located in a package called "ram_package".

*This same example is used in the *Mentor Graphics VHDL Reference Manual* to describe subprograms in further detail.

The "how it happens" details are located in the procedures and function (highlighted with bold lines in Figure 4-6). The inset at the lower-right corner of Figure 4-6 shows the coupling between the modules of code in another format.

The following code is required to support the example in Figure 4-6 and is shown for completeness. These lines are located throughout the "what happens" code, not in the package that contains the procedures and function.

The following line is located before the entity declaration of the calling code to identify the logical name of the library containing the ram design unit and the logical name of the library containing the ram_package:

```
LIBRARY lib_stuff, package_stuff;
```

Each VHDL implementation uses some scheme to map each logical name to a physical location where the libraries reside on the node or network.

The following line can be declared in a declarative region, such as the entity declaration or the architecture body of the calling code:

```
USE lib_stuff.ram_package.all;--Makes package containing
                               --function and procedure
                               --visible to calling code.
```

The following lines are declared in the ram_package declarative part:

```
TYPE op_code_array IS ARRAY (0 TO 255)
                           OF bit_vector (0 TO 7);
```

See the library and use clause descriptions on page 4-48 for more information on these constructs.

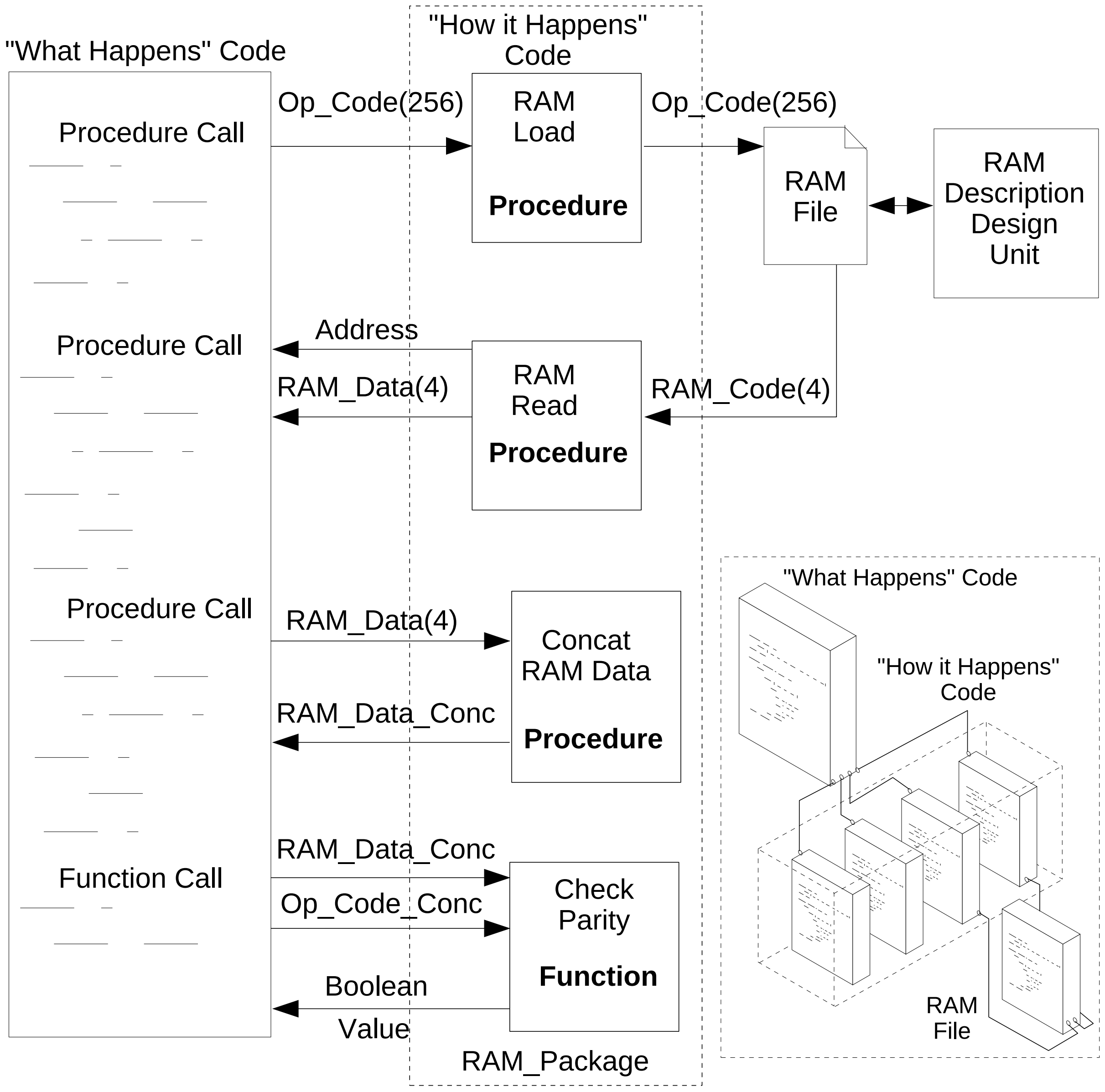


Figure 4-6. Using Procedures and Functions

The RAM Load procedure shown in Figure 4-7 accepts operation code (Op-code) supplied by the calling description and loads it into 256 lines in a RAM file that is also used by a separate RAM design unit. The code in Figure 4-7 is not good use of a procedure because it only accepts one parameter, but it is used here to show how to cause a change in a remote design unit without affecting a change in the calling code.

```
 1     ------------- Subprogram Declaration - Procedure ------
 2   PROCEDURE ram_load (CONSTANT Op_code : IN op_code_array);
 3
 4     ------- Subprogram Body - Procedure -------------------
 5   PROCEDURE ram_load(CONSTANT Op_code : IN op_code_array) IS
 6     FILE ram_cntnts: op_code_array IS IN
 7                          "/idea/user/vhdl_lib/ram1_file";
 8
 9    BEGIN
10     FOR a IN Op_code'RANGE LOOP
11      write(ram_cntnts, Op_code(a));--load op_code into file
12     END LOOP;
13   END ram_load;
```

Figure 4-7. Code Example of a Subprogram--RAM Load Procedure

The RAM Load procedure in Figure 4-7 consists of a declaration and a body. The subprogram declaration (line 2) couples the procedure to the calling code by specifying one input constant (`Op_code`) expressed as having a type of `op_code_array` (a template for a 256-element array {0 to 255} of 8-bit {0 to 7} vectors).

The subprogram body (line 5) starts by repeating the subprogram specification from the subprogram declaration. This procedure does not return a value to the calling code, but it does change the contents of a file for the separate RAM design unit.

The subprogram statement part (line 11 of Figure 4-7) calls a predefined procedure `write` to load an element of the `op_code` array to file `ram1_file` declared in lines 6 and 7. The write procedure is implicitly defined with the `ram_cntnts` file declaration in lines 6 and 7 of Figure 4-7 as is a read and endfile procedure.

The `write` procedure requires two parameters: first the file declaration identifier (`ram_cntnts`) and then the data to be passed (`op_code`). The file declaration identifier (`ram_cntnts` in line 11) associates the file declaration with the actual file `ram1_file`. The loop statement (lines 10 and 12) causes the write procedure to execute for each element of the `Op_code` array.

In general, Figure 4-7 shows how you can use a procedure to cause a side-effect without returning a value to the calling code.

Figure 4-8 shows the code for the RAM Read procedure (from the example in Figure 4-6). The `ram_read` procedure returns two separate values to the calling routine; `ram_data` (a four-element array) and `test_add_start` (test address start). The purpose of this procedure is to read four consecutive data values from the `ram1_file` randomly and to return these values to the calling routine. Later the returned values are checked against the original op-code data to check if the RAM data has been corrupted. This procedure also returns the starting address value so the later operation knows where to start for a comparison check.

The following type declaration must be declared in the `ram_package` declarative part so this type declaration is visible to the `ram_read` procedure and the code that calls the `ram_package`. This type declaration creates a template for a four-element array, each containing an eight-bit vector.

```
TYPE ram_data_array IS ARRAY(0 TO 3) OF bit_vector(0 TO 7);
```

The subprogram declaration and subprogram body in Figure 4-8 start out by specifying the two output variables that pass the data (`ram_data`) and address (`test_add_start`) information back to the calling routine.

Line 17 in Figure 4-8 generates a starting address (`address`) by performing a number of operations. The first operation calls a predefined function `rand` from the predefined math package which is declared as follows:

```
FUNCTION rand( seed : real ) RETURN real;
```

The use clause in line 10 makes the math package visible. Only the `rand` function is directly visible from the math package because it was explicitly called in the use clause.

The `rand` function in line 17 generates a random floating point number between 0 and 1.0. The function accepts a seed value which is declared as a constant in line 14 and set to 0.1. As an example, assume this function returns a value of 0.654321.

The second operation in line 17 multiplies the returned value from the `rand` function by 63.0 to generate a floating point number between 0 and 62.99. Continuing with the 0.654321 value from the previous paragraph, the second operation returns 41.2222 (0.654321 % 63.0 = 41.2222).

The third operation in line 17 uses the type conversion construct to convert the floating point value between 0 and 62.99 into an integer value between 0 and 63. The number 63 is derived by dividing the 256 address locations into equal blocks of 4 consecutive addresses (64 blocks, from 0 to 63). The conversion would take value 41.2222 and convert (round) it to integer 41.

The fourth operation in line 17 multiplies the integer value of 0 to 63 by four to generate the starting address for one of the address blocks. The last address block starts at address 252, so the final result from the operations in line 17 should not exceed this value. (This example yields 41 % 4 = 164.) The final operation in line 17 assigns the result to variable `address`.

Line 18 assigns the starting address to variable `test_add_start`, which is returned to the calling code.

```
------------------ Subprogram Declaration - Procedure --------
PROCEDURE ram_read (VARIABLE ram_data: OUT ram_data_array;
              VARIABLE test_add_start: OUT integer );

 ------------------- Subprogram Body - Procedure ---------------
PROCEDURE ram_read (VARIABLE ram_data: OUT ram_data_array;
              VARIABLE test_add_start: OUT integer ) IS
 FILE ram_cntnts: op_code_array IS IN
                            "/idea/user/vhdl_lib/ram1_file";
 USE std.math.rand;    -- Makes rand function from math
                       -- package visible to this procedure.
 VARIABLE address : integer ;
 VARIABLE op_code : op_code_array;
 CONSTANT Seed    : real := 0.1;

BEGIN                          -- generate random address
  address := integer(rand(Seed) * 63.0) * 4; --between 0 & 252
  test_add_start := address;
  FOR a IN 0 TO (address + 3) LOOP     -- Read file until
desired
    read(ram_cntnts, op_code(a));      -- data is reached.
    IF a >= address THEN
      ram_data(a - address) := op_code(a); --extract desired
data
    END IF;
  END LOOP;
END ram_read;
```

Figure 4-8. Code Example of a Subprogram--RAM Read Procedure

The loop statement (lines 19 and 24) in Figure 4-8 is set to loop from 0 up to the starting address plus 3. In this example it loops from 0 to 167. For each pass through the loop statement, the predefined read procedure* in line 20 is executed. The first time through the loop, the read procedure returns the value of line 1 of file `ram1_file` to the first element (0) of array `op_code`. Each consecutive loop retrieves the next eight-bit array value from `ram1_file` and loads it to the appropriate element of the `op_code` array. When the loop counter (a) of line 19 in Figure 4-8 equals the `address` value (164 in this example), the if statement condition in line 21 is true and line 22 is executed. As long as the loop counter is greater than or equal to the `address` value, the if statement is executed. In this example the value of `op_code(164)` is assigned to `ram_data(0)`, on the next pass through the loop the value of `op_code(165)` is assigned to `ram_data(1)`, and so it goes until all four elements of `ram_data` have been filled.

Figure 4-8 shows how you can use a procedure to return multiple values to the calling routine.

Figure 4-9 shows the code for the Concatenate Data procedure (from the example in Figure 4-6). This procedure receives the `ram_data` array from the calling routing and returns one parameter (`ram_data_conc`). This procedure concatenates the four 8-bit `ram_data` bit vectors into one 32-bit `ram_data_conc` bit vector (lines 11 and 12 of Figure 4-9).

*The read procedure is implicitly defined with the `ram_cntnts` file declaration in lines 8 and 9 of Figure 4-8 as is a write and endfile procedure.

```
  ------------ Subprogram Declaration - Procedure ------
PROCEDURE concat_data (
  CONSTANT Ram_data      : IN ram_data_array;
  VARIABLE ram_data_conc : OUT bit_vector (0 TO 31) );

  -------- Subprogram Body - Procedure -----------------
PROCEDURE concat_data (
  CONSTANT Ram_data      : IN ram_data_array;
  VARIABLE ram_data_conc : OUT bit_vector (0 TO 31) ) IS
BEGIN
  ram_data_conc:= Ram_data(0)& Ram_data(1)&
                  Ram_data(2)& Ram_data(3);
END concat_data;
```

Figure 4-9. Example of a Subprogram--Concatenate Data Procedure

Now that you have seen various uses of procedures, examine the function example in Figure 4-10. The purpose of this function is to check the parity on the 32-bit `ram_data_conc` and `op_code_conc` bit vectors, compare the results, and then return a TRUE or FALSE Boolean value to the calling code.

The function in Figure 4-10 starts out the same as does the previous procedure examples. The input parameters* are specified in the subprogram specification in both the declaration and body. In this example, the function receives two input bit vectors; `ram_data_conc` and `op_code_conc`.

Two local variables are declared in line 10 of the subprogram body (`sum1` and `sum2`) to hold the parity result for each input bit vector. The loop (lines 12 through 19) cycles through 32 times, once for each of the 32 bits in the concatenated arrays. The return statement (line 20) returns a TRUE Boolean value if sum1 = sum2; otherwise a FALSE value is returned.

*Functions cannot use "out" or "in/out" parameters in the subprogram specification because a function returns only a single value.

```
------------------- Subprogram Declaration - Function ---------
FUNCTION chk_pty (CONSTANT Ram_data_conc: IN bit_vector(0 TO
31);
                  CONSTANT Op_code_conc : IN bit_vector(0 TO
31))
  RETURN boolean;

-------------------- Subprogram Body - Function ----------------
FUNCTION chk_pty (CONSTANT Ram_data_conc: IN bit_vector(0 TO
31);
                  CONSTANT Op_code_conc : IN bit_vector(0 TO
31))
  RETURN boolean IS
  VARIABLE sum1, sum2 : boolean := false;
BEGIN
  FOR i IN 0 TO 31 LOOP
    IF Ram_data_conc(i) = ’1’ THEN  -- compute parity for
       sum1 := NOT sum1;            -- concatenated ram data
    END IF;
    IF Op_code_conc(i) = ’1’ THEN   -- compute parity for
       sum2 := NOT sum2;            -- concatenated op code data
    END IF;
  END LOOP;
  RETURN sum1 = sum2; -- return true if sum1=sum2,
END chk_pty;          -- false if not equal
```

Figure 4-10. Code Example of a Subprogram--Parity Checker Function

The following list summarizes the main features of the subprogram types: functions and procedures.

Functions	Procedures
● Produce no side-effects	● Can produce side-effects
● Only accept input (**in**) parameters	● Accept input (**in**), output (**out**), and input/output (**inout**) parameters
● Return just one value	● Do not have to return any value or can return multiple values
● Always use the reserved word **return**	● Do not require the reserved word **return**

To complete the description of procedures and functions, the constructs for calling a function or procedure are described in the following subsections.

Function Call

A function call is used as part of an expression to execute the named function, specify actual parameters (if any), and return one value to the expression. The function call can be part of a sequential or concurrent decomposition. This description is included here to show how subprograms can be used. The format of a function call is shown in the following paragraph along with the format of the association element construct:

function call .... name **(** association_element**,** ... **)**

association element .... formal_part => actual_part

The Check Parity function call in Figure 4-6 could appear as the Boolean expression in an assert statement such as the following:

```
ASSERT chk_pty (op_code_conc  => op_code_c,
                ram_data_conc => ram_data_c)
  REPORT "Parity Check Failed."  --report if condition false
   SEVERITY note;
```

The string literal "`Parity Check Failed`" in the report is sent from the assert statement if `chk_pty` returns a FALSE value. The association elements shown in this example use the optional formal part. The formal part associates a formal parameter (from the function) with the related actual parameter (in the calling code), as shown in Figure 4-11.

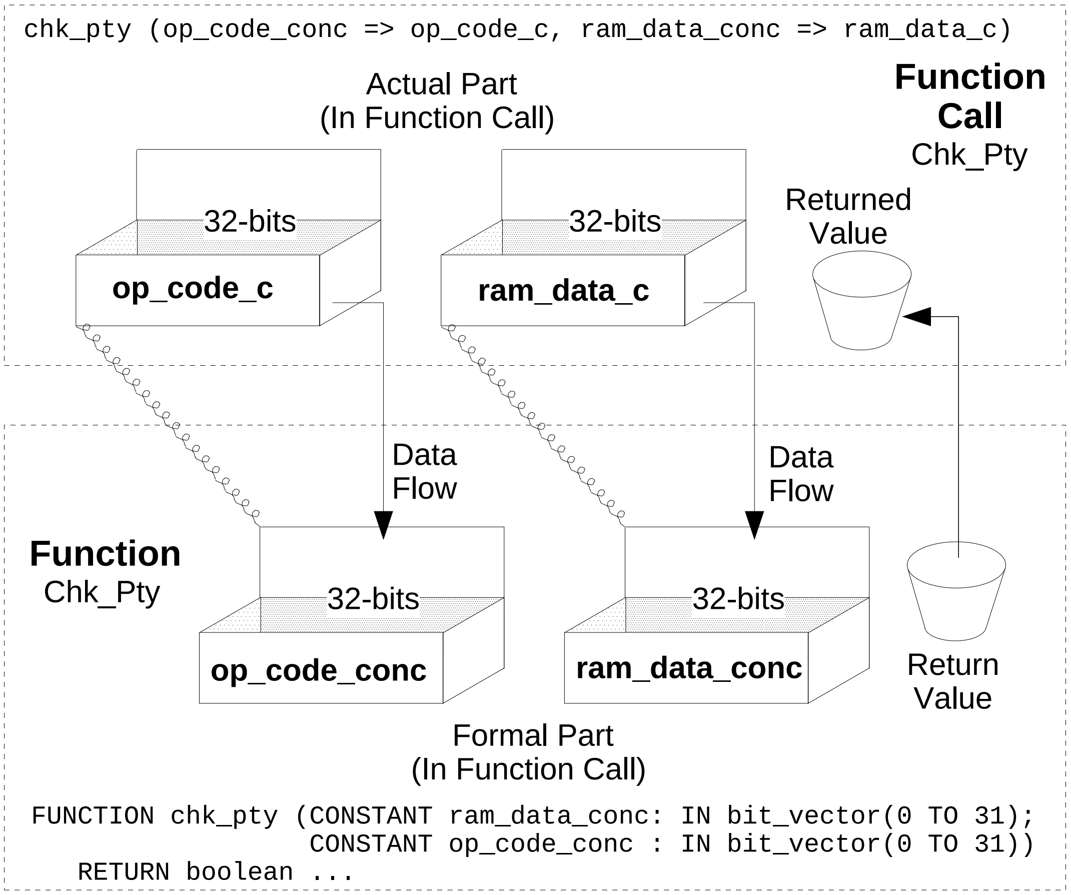
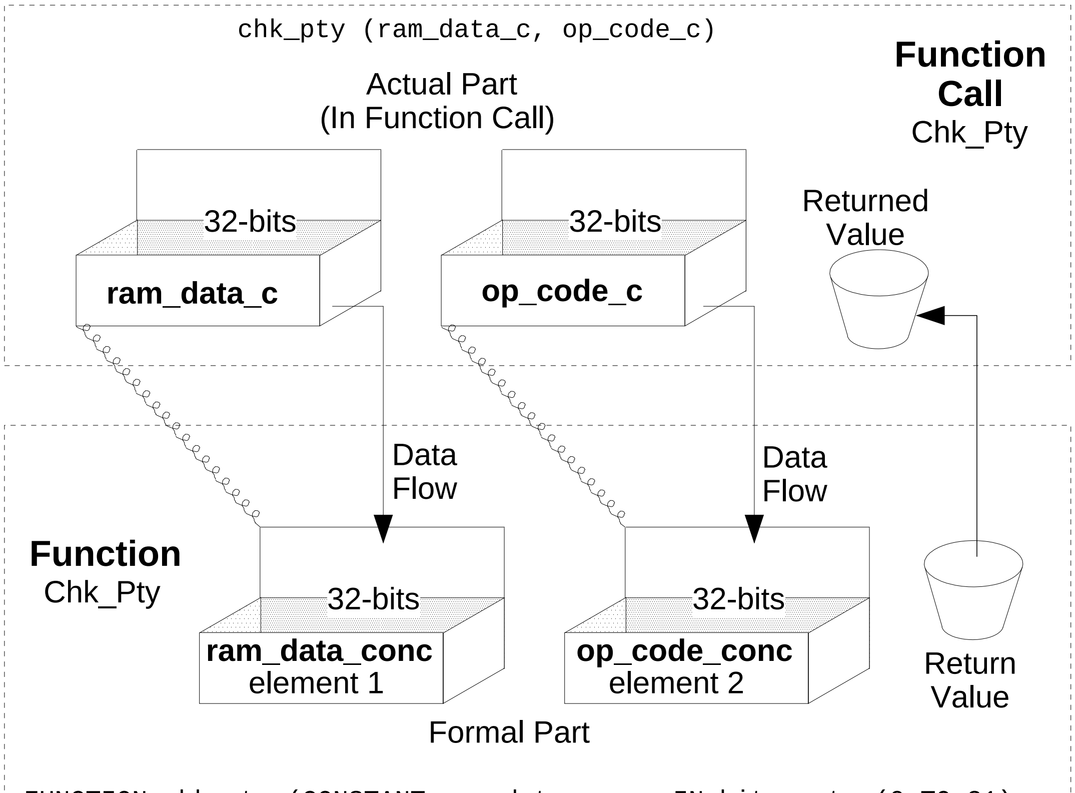


Figure 4-11. Associating Actual Parameters to Formal Parameters

The formal part `op_code_conc` from the function `chk_pty` is explicitly associated with `op_code_c` (denoted with a coiled line in Figure 4-11) in the function call, and `ram_data_conc` is explicitly associated with `ram_data_c` in the function call. This association is termed "named notation" or "named association". It does not matter how the association elements are ordered in the function call when you use the named association. You must, however, be careful that the type of the each formal parameter matches the type of the associated actual parameter.

If the optional named notation is not used in a function call, the association is determined by the order in which the parameter(s) are listed in the function and the function call. Figure 4-12 shows a modified version of the previous function call example that does not include the optional formal part.



```
FUNCTION chk_pty (CONSTANT ram_data_conc: IN bit_vector(0 TO 31);
                  CONSTANT op_code_conc : IN bit_vector(0 TO 31))
   RETURN boolean ...
```

Figure 4-12. Positional Parameter Notation in a Function Call

The order of the actual parameters (in the function call) determines which actual parameter is associated with the corresponding formal parameter (in the function). This ordering is called "positional notation" or "positional association".

In Figure 4-12, since `ram_data_c` is the first parameter listed in the function call, it is associated with the first element in the function (`ram_data_conc`). The second parameter in this function call (`op_code_c`) is associated with the second element in the function (`op_code_conc`).

Procedure Call

A procedure call can be either a concurrent statement or a sequential statement. A procedure call simply executes the named procedure. The format of each type of call (concurrent or sequential) is shown in the following examples:

Concurrent Statement

concurrent procedure call	label **:** name **(** association_element**,** ... **) ;**

Sequential Statement

procedure call statement ..	name **(** association_element**,** ... **) ;**

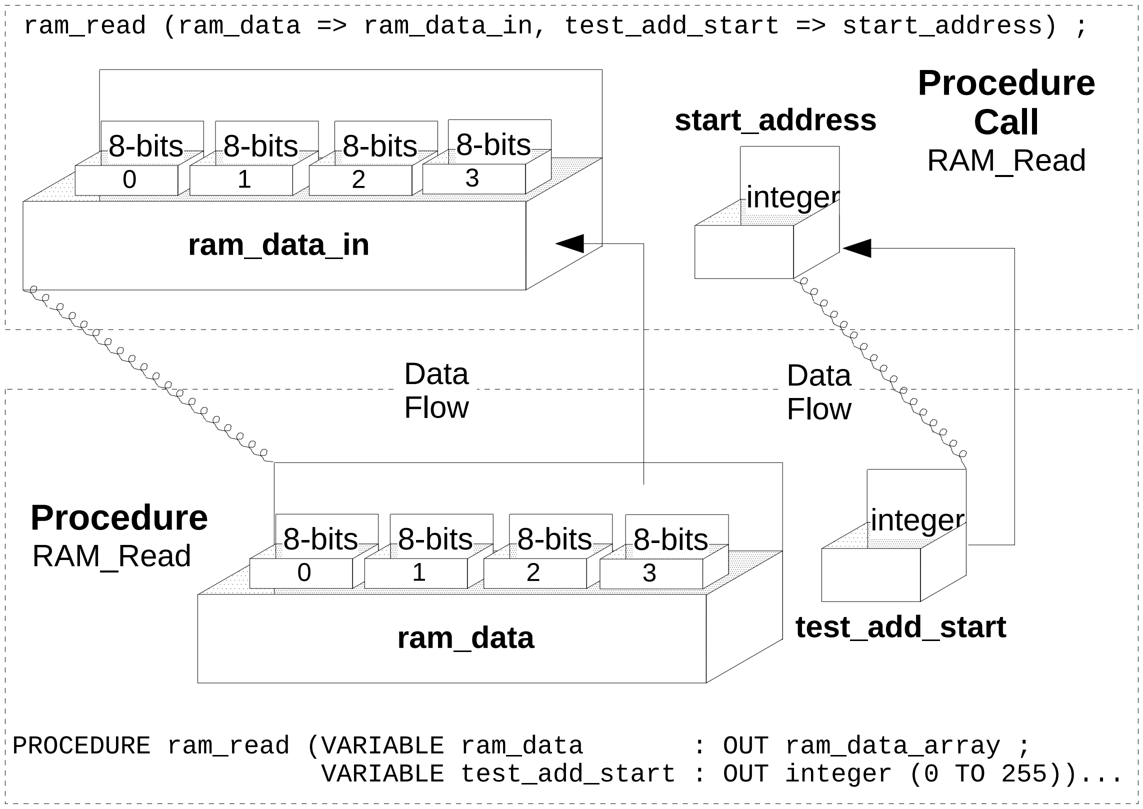
The format of a procedure call statement is similar to the format of a function call. Figure 4-13 shows how the RAM Read procedure call might look from the example shown in Figure 4-6. This procedure call associates formal part `ram_data` with actual part `ram_data_in` and associates formal part `test_add_start` with actual part `start_address`.

This procedure call receives the four, eight-bit `ram_data` array elements and the `test_add_start` integer from the called procedure. These values can then be used within the calling code.

If the optional named association is not used in a procedure call, the association is determined by the order in which the parameter(s) are listed in the procedure and the procedure call. This feature is identical to the positional association for a function call. The procedure call from Figure 4-13 could be rewritten as the following:

```
ram_read (ram_data_in, start_address);
```

The order of the actual parameters (in the procedure call) determines which actual parameter is associated with the corresponding formal parameter (in the procedure). Also refer to the previous description of a function call on page 4-24.



```
ram_read (ram_data => ram_data_in, test_add_start => start_address) ;
```

```
PROCEDURE ram_read (VARIABLE ram_data       : OUT ram_data_array ;
                    VARIABLE test_add_start : OUT integer (0 TO 255))...
```

Figure 4-13. Procedure Call Parameter Association

Contrasting Concurrent and Sequential Modeling

When modeling a complex circuit or system function, you must make sure to model the correct behavior regarding concurrent and sequential tasks. The following simple design example shows the modeling tasks from the truth table down through several model abstraction levels.

Table 4-1 shows the truth table for an AND/OR/Invert (AOI) circuit function which is used as this design example. This circuit has four inputs (A through D) and one output (E). These inputs and output are of type my_qsim_state. Assume a zero propagation delay through the circuit.

Table 4-1. AND/OR/INVERT Truth Table

A	B	C	D	E
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

The my_qsim_state type is defined in a package called "my_qsim_logic". This example assumes the package resides in a library with a logical name of "my_lib". The use clause in line 1 makes this package visible to the aoi design. For reference, the my_qsim_state type declaration is shown here:

```
  TYPE my_qsim_state IS ('X', '0', '1', 'Z');
```

Using the information you have so far, you can write the entity declaration (in a design file), as shown in the following example:

```
LIBRARY my_lib; USE my_lib.my_qsim_logic.ALL;
ENTITY aoi IS
   PORT (A, B, C, D : IN  my_qsim_state;
                  E : OUT my_qsim_state );
END aoi;
```

Next you can simplify the truth table using a Karnaugh map as shown in Table 4-2. The resulting Boolean equation is: E = $\overline{AB + CD}$.

Table 4-2. AND/OR/Invert Karnaugh Map

CD \ AB	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	0	0	0	0
10	1	1	0	1

Using the Boolean equation, you can write the following architecture body in a separate design file or in the same design file as the entity declaration:

```
ARCHITECTURE behav1 OF aoi IS

BEGIN
 E <= NOT((A AND B) OR (C AND D)); --con. sig. asgnmt stmt
END behav1;
```

This code can now be compiled and simulated to verify that it performs according to the truth table. The concurrent signal assignment statement performs as the name implies; it assigns the result of the expression on the right side to the signal (`E`) on the left side. This assignment is done in parallel with any other concurrent statements that occur (there are none in this example) in the architecture body.

If you need a behavioral model only to describe a certain set of output conditions given multiple input conditions, you would not need to go down to a lower abstraction level. However, if you want to implement the design into hardware/silicon, you would continue as follows.

The equation `E <= NOT((A AND B) OR (C AND D))` implies three basic logic functions; AND, OR, and an inverter. The next abstraction level description you write could be the following data-flow description (`dflow1`) on the left or the equivalent behavioral description (`behav2`) on the right. (The comments CS1 through CS4 indicate concurrent statements and SS1 through SS4 indicate sequential statements.)

```
 1   ARCHITECTURE dflow1               1   ARCHITECTURE behav2
 2                      OF aoi IS      2                 OF aoi IS
 3   SIGNAL O1,O2,O3:                  3   BEGIN
 4                  my_qsim_state;     4    PROCESS (A, B, C, D)
 5   BEGIN                             5     VARIABLE O1, O2,
 6      E  <= NOT O3;   --CS1          6        O3: my_qsim_state;
 7      O1 <= A AND B;  --CS2          7    BEGIN
 8      O2 <= C AND D;  --CS3          8     O1 := A AND B; --SS1
 9      O3 <= O1 OR O2; --CS4          9     O2 := C AND D; --SS2
10   END dflow1;                      10     O3 := O1 OR O2;--SS3
                                      11     E  <= NOT O3;  --SS4
                                      12    END PROCESS;
                                      13   END behav2;
```

When writing either the `dflow1` or the `behav2` descriptions, be sure to model the concurrency of the actual tasks correctly. Because this example assumes a zero propagation delay, all assignments are performed in parallel. The evaluation flow within a given timestep for `behav1`, `dflow1`, and `behav2` is shown in Figure 4-14.

In the `dflow1` architecture, the order of the concurrent statements (lines 6 through 9) is not important since the simulator executes each statement in arbitrary order within a given timestep. You probably would order them with line 6 (`E <= NOT O3;`) at the bottom for clarity, even though it does not affect the order of evaluation during simulation.

The `behav2` architecture requires some special considerations regarding the concurrency of the circuit operation. A process is used to group the statements that define the circuit behavior. Because a process is a concurrent statement, it executes in parallel with any other concurrent statement. It is important to realize that statements within a process execute sequentially. Variables are declared (lines 4 and 5) within the process to mimic the internal networks shown as O1, O2, and O3 in Figure 4-15. Signals cannot be declared within a process.

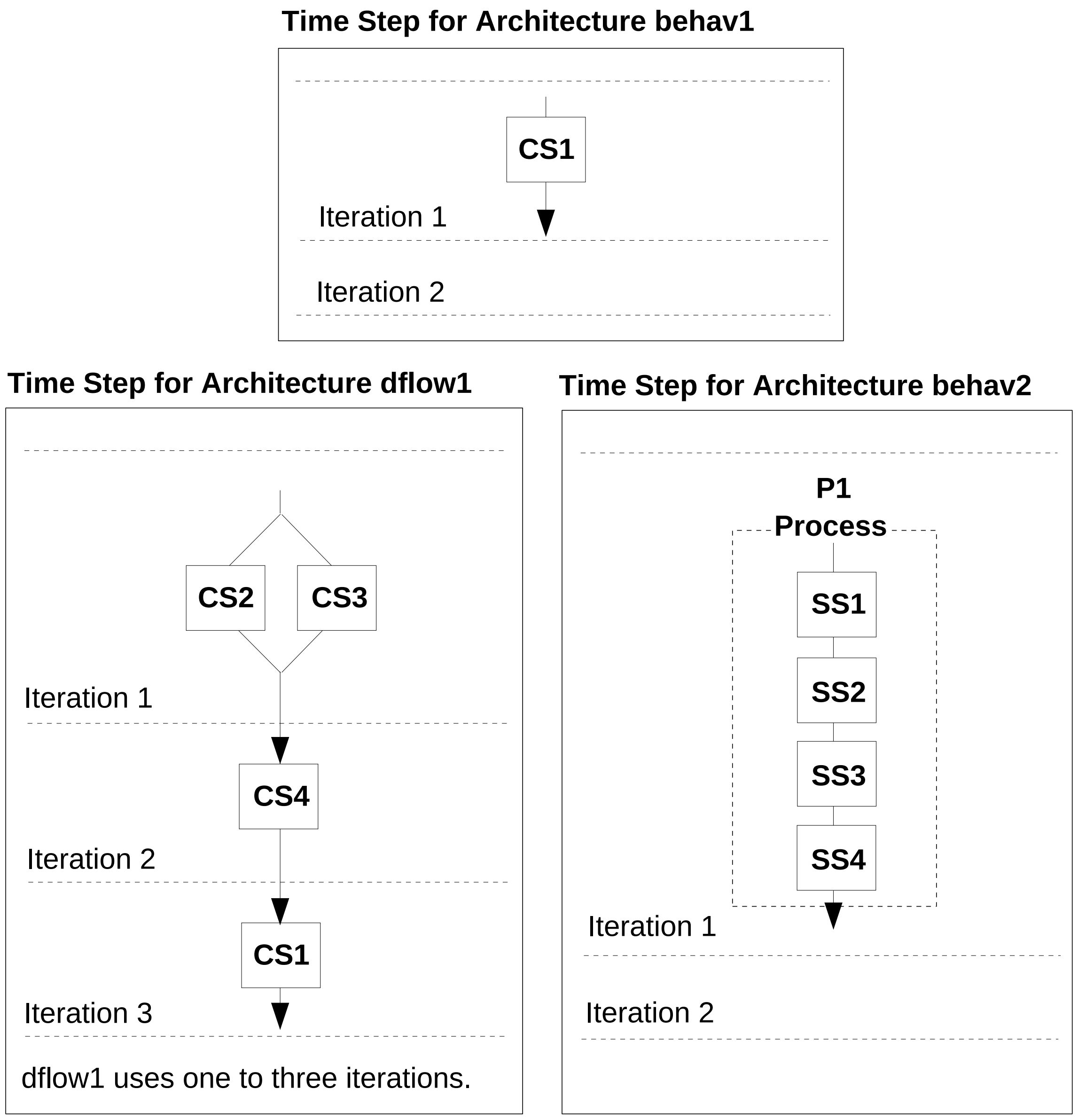

Time Step for Architecture behav1
CS1
Iteration 1
Iteration 2
Time Step for Architecture dflow1
CS2
CS3
Iteration 1
CS4
Iteration 2
CS1
Iteration 3
dflow1 uses one to three iterations.
Time Step for Architecture behav2
P1
Process
SS1
SS2
SS3
SS4
Iteration 1
Iteration 2

Figure 4-14. Evaluation Flow for First Behavioral AOI Models

The statements in architecture `behav2` must be ordered as shown to be sure that the circuit behavior is modeled properly. If you start a simulation run by forcing values of ’0’ on the A and B inputs and ’1’ on the C and D inputs, the code will execute in a sequential manner (shown in the comments) as follows:

```
 1   ARCHITECTURE behav2 OF aoi IS
 2   BEGIN
 3     PROCESS (A, B, C, D) -- 1. Changes on signals activates process.
 4       VARIABLE O1, O2,
 5                O3: my_qsim_state;
 6     BEGIN
 7       O1 := A AND B;      -- 2. O1 is assigned a '0', '0' AND '0' = '0'
 8       O2 := C AND D;      -- 3. O2 is assigned a '1',  '1' AND '1' = '1'
 9       O3 := O1 OR O2;     -- 4. O3 is assigned a '1', '0' OR  '1' = '1'
10       E  <= NOT O3;       -- 5. E is assigned a '0', INVERT '1' = '0'
11     END PROCESS;
12   END behav2;
```

Now consider the same code, but this time with the statements out of order as shown in the following example. The same values as previously described are used on this model, but this time you will notice the model does not perform correctly.

```
 1   ARCHITECTURE behav2 OF aoi IS
 2   BEGIN      --WRONG BEHAVIOR
 3      PROCESS (A, B, C, D) --1. Changes on signals activates process.
 4        VARIABLE O1, O2,
 5                 O3: my_qsim_state;
 6      BEGIN
 7      O3 := O1 OR O2;     --2. O3 is assigned an 'X',  'X' OR  'X' = 'X'
 8      O1 := A AND B;      --3. O1 is assigned a '0', '0' AND '0' = '0'
 9      O2 := C AND D;      --4. O2 is assigned a '1',  '1' AND '1' = '1'
10      E  <= NOT O3;       --5. E is assigned an 'X', INVERT 'X' = 'X'
11      END PROCESS;
12   END behav2; --WRONG BEHAVIOR
```

In line 7 of the previous example, the assignment was made using the previous simulator state of 'X' for O1 and O2 because A, B, C, and D were all 'X' before values were applied. The model does not perform properly because the statements are executed sequentially, and they are not ordered properly.

When denoting an 'X' or 'Z' state of type my_qsim_state, you must use uppercase characters. The 'X' and 'Z' of the my_qsim_state type are character literals, which means they are case-sensitive.

By now the structure of this circuit may be apparent and is shown in the schematic in Figure 4-15 to be used as a reference for the rest of this discussion.

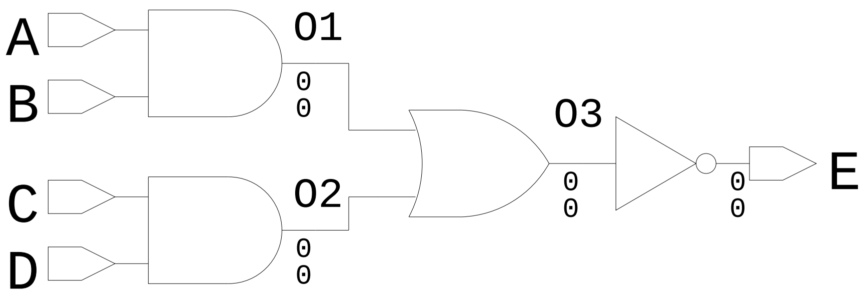


Figure 4-15. AND/OR/Inverter Circuit

Figure 4-16 shows the three distinct functions in this circuit. The OR function is dependent on the outcome of the AND operation, and the invert function is dependent on the outcome of the OR operation. This arrangement shows a necessity for three iterations in a given timestep.

The following description shows another way to describe the behavior of the circuit that will be implemented in hardware. Process P1 is equivalent to concurrent statement "`O3 <= O1 OR O2;`", which could appear in the architecture statement part. Process P2 is equivalent to concurrent statement "`E  <= NOT O3;`".

```
ARCHITECTURE behav3 OF aoi IS
  SIGNAL O1,O2,O3: my_qsim_state;

BEGIN
   O1 <= A AND B;   -- CS1
   O2 <= C AND D;   -- CS2

   P1: PROCESS (O1, O2)
   BEGIN
     O3 <= O1 OR O2;  -- SS1
   END PROCESS;

   P2: PROCESS (O3)
   BEGIN
     E  <= NOT O3;    -- SS1
   END PROCESS;
```

```
17   END behav3;
```

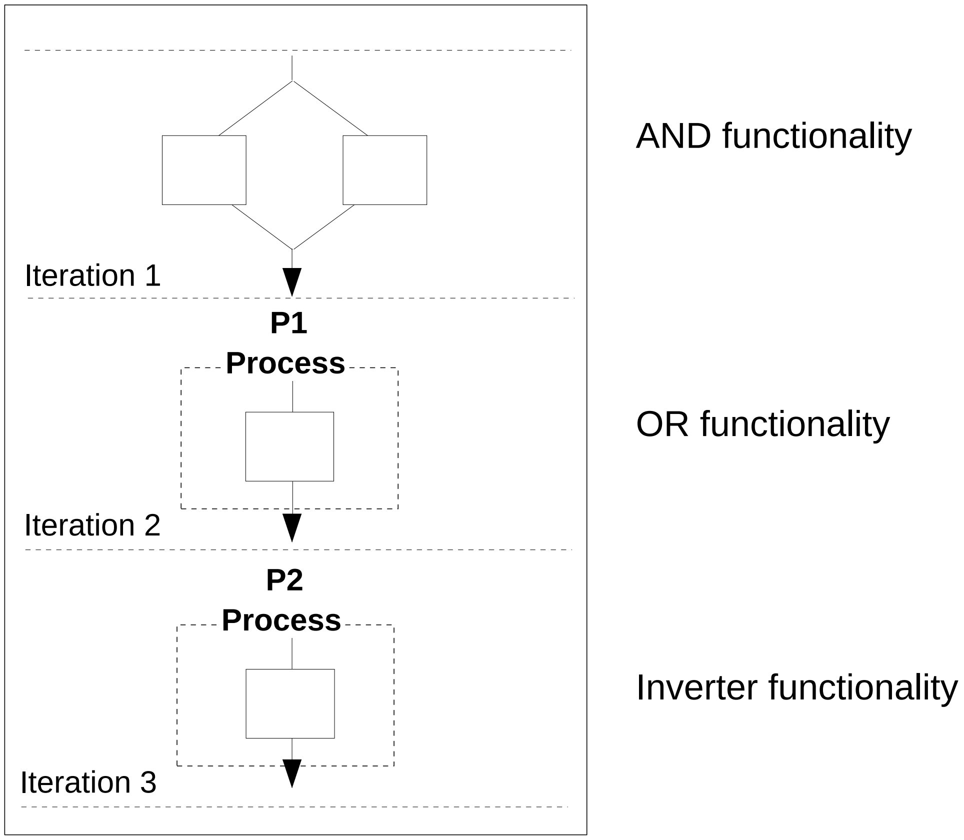


Figure 4-16. Operation-Flow Diagram for AOI Circuit

In architecture `behav3`, lines 5 and 6 perform the AND operations concurrently as before. The simulation requires one iteration of a timestep. Process P1 does not execute in iteration 1 because O1 and O2 will not have changed value until the end of the iteration. Process P2 does not execute because O3 has not changed value.

When iteration 2 begins, Process P1 executes and causes a change to O3 (at the end of the iteration). Process P2 executes in iteration 3 and makes the assignment to output `E`. Architecture `behav3` best represents the internal behavior (apart from a structural representation) of the circuit.

If you intend your model to be connected and simulated with other circuits in a larger design, it would be best to use architecture `behav1`. Architecture `behav1` is simple and quicker than the other architectures to simulate. If the intent of the model is to document the actual data flow through the circuit, either `dflow1` or a structural representation will be best.

As shown in the examples in this subsection, there are many ways to model the behavior of one design. You should use concurrent and sequential statements to best represent the design functionality as you progress through the various abstraction levels of a design.

How Values Get Assigned to Signals and Variables

Just as you must take care to model the correct behavior regarding concurrent and sequential tasks, you must also consider how signals and variables are treated during simulation. The improper use of signals and variables can cause simulation results that are different than expected. This subsection shows how values get assigned to signals and variables during simulation.

Of the three kinds of objects (signals, variables, and constants), signals provide the interconnection between components, blocks, and processes. Variables and constants provide containers for values used in the computation of signal values. For every signal that you declare and assign values, there is a least one driver associated with the signal. Each driver holds the projected waveform for the signal.

The most important thing to remember about variables and signals is that a variable assignment updates the variable value within the current simulation iteration; a signal assignment causes an event to be scheduled at least one iteration later (one delta delay*), when the signal then receives the new value. If a signal assignment contains a delay value more than zero, the event is scheduled for the appropriate timestep in the future. If the signal assignment specifies a zero delay (or if none is given, zero nanoseconds is the default), the signal driver does not update the signal value until one iteration after the

*For more information on delta delay concepts, see the *Mentor Graphics VHDL Reference Manual* assignment statement was evaluated.

Consider the code variations in Figure 4-17 of the AOI circuit, which is used in the previous subsection. These architectures declare O1, O2, and O3 as signals. Again the ordering within the process is important for the same reasons shown for architecture `behav2` in the previous subsection. You would not expect `behav5` to simulate properly because of the ordering of the signal assignment statements. However, you might think that `behav4` would simulate correctly because it appears to be ordered correctly.

In addition to the sequential statement ordering, you now also need to be concerned with how and when the signal values are updated during simulation. A delay of 0 ns is added to each signal assignment statement in examples `behav4` and `behav5` as a reminder that there is some unit of delay (even for a 0 ns delay) before the signal receives a new value. Even though the order of statements is correct in `behav4`, this model does not generate simulation results one might expect.

The signal states during initialization of the `behav4` architecture is as follows:

```
INIT.     X  X  X  X  X
          ^A ^B ^C ^D ^E
```

If at time zero you force all the inputs to 0 you might expect output `E` to go to ’1’. The following shows results from a simulator (using a resolution of 0.1 ns per timestep) for this condition:

```
Time 0.0  0  0  0  0  X   #END OF TIMESTEP
          ^A ^B ^C ^D ^E
```

If you step through the `behav4` process, you can see why output `E` did not change to ’1’. When all the inputs are forced to ’0’, the sensitivity list activates the process in timestep 0.0, iteration 1. The statement `"01 <= A AND B AFTER 0 ns;"` executes first during iteration 1 and *schedules an event* on signal `01` (to change to ’0’ after 0 ns). The current value (in iteration 1) of signal `O1` is ’X’ as determined during initialization.

```
ARCHITECTURE behav4 OF aoi IS
 SIGNAL O1,O2,O3:
              my_qsim_state;
BEGIN    --WRONG BEHAVIOR
 PROCESS (A, B, C, D)
 BEGIN
  O1 <= A AND B  AFTER 0 ns;
  O2 <= C AND D  AFTER 0 ns;
  O3 <= O1 OR O2 AFTER 0 ns;
  E  <= NOT O3   AFTER 0 ns;
 END PROCESS;
END behav4; --WRONG
```

```
ARCHITECTURE behav5 OF aoi IS
 SIGNAL O1,O2,O3:
              my_qsim_state;
BEGIN    --WRONG BEHAVIOR
 PROCESS (A, B, C, D)
 BEGIN
  O3 <= O1 OR O2 AFTER 0 ns;
  O1 <= A AND B  AFTER 0 ns;
  O2 <= C AND D  AFTER 0 ns;
  E  <= NOT O3   AFTER 0 ns;
 END PROCESS;
END behav5; --WRONG
```

Figure 4-17. Changing Model Behavior by Moving Sequential Statements

Statement "`O2 <= C AND D AFTER 0 ns;`" executes next during iteration 1 and *schedules an event* on signal `O2` (to change to '0' after 0 ns). The current value (in iteration 1) of signal `O2` is also 'X' as determined during initialization.

Statement "`O3 <= O1 OR O2 AFTER 0 ns;`" executes next during iteration 1 and *schedules an event* on signal `O3` (to change to 'X' after 0 ns). The evaluation of this statement uses the current value (in iteration 1) of signal `O1` (X) and `O2` (X).

Statement "`E <= NOT O3 AFTER 0 ns;`" executes next during iteration 1. It *schedules an event* on signal `E` (to change to 'X' after 0 ns). The evaluation of this statement uses the current value (in iteration 1) of signal `O3` (X). The last statement that executes during iteration 1 is the implied wait statement from the process sensitivity list.

Now that all statements have executed, the simulator is advanced 0.1 ns and input `D` is forced to a '1' using the Force command. The status of each signal input and out signal at the end of this timestep is shown as follows:

```
Time 0.1  0  0  0  1  X    #END OF TIMESTEP
          ^A ^B ^C ^D ^E
```

Stepping through the `behav4` process, you can again see why output `E` still did not change to ’1’ in timestep 0.1. When input `D` is forced to ’1’, the sensitivity list activates the process in timestep 0.1, iteration 1. At the beginning of this iteration the values stored in each driver for signals `O1`, `O2`, `O3`, and `E` are placed on each signal. At the beginning of iteration 1 for timestep 0.1, the complete status of all the signals are as follows:

```
Time 0.1  0  0  0  1  X   0   0   X   #AT BEGINNING OF
          ^A ^B ^C ^D ^E  ^O1 ^O2 ^O3 #ITERATION 1
```

These signal states were derived from the simulator forces and the events that were scheduled in iteration 1 of timestep 0.0. Using these values, you can again step through the sequential statements within the process.

Statement "`O1 <= A AND B AFTER 0 ns;`" executes first during iteration 1 and *schedules an event* on signal `01` (to change to ’0’ after 0 ns). The current value (in iteration 1) of signal `01` is ’0’ as determined during the previous timestep.

Statement "`O2 <= C AND D AFTER 0 ns;`" executes next during iteration 1 and *schedules an event* on signal `O2` (to change to ’0’ after 0 ns). The current value (in iteration 1) of signal `O2` is also ’0’ as determined during the previous timestep.

Statement "`O3 <= 01 OR 02 AFTER 0 ns;`" executes next during iteration 1 and *schedules an event* on signal `O3` (to change to ’0’ after 0 ns). The evaluation of this statement uses the current value (in iteration 1) of signal `O1` (0) and `O2` (0). The current value (in iteration 1) of `O3` is ’X’.

Statement "`E <= NOT O3 AFTER 0 ns;`" executes next during iteration 1. It *schedules an event* on signal `E` (to change to ’X’ after 0 ns). The evaluation of this statement uses the current value (in iteration 1) of signal `O3` which is ’X’. The last statement that executes during iteration 1 is the implied wait statement from the process sensitivity list.

Using the same procedure as before you can work through the process again for timestep 0.2 using the following signal conditions:

```
Time 0.2  0  0  1  0  X   0   0   0  #BEGINNING OF IT 1
Time 0.2  0  0  1  0  1   0   0   0  #END OF TIMESTEP
          ^A ^B ^C ^D ^E  ^O1 ^O2 ^O3
```

You can work through the statements in architecture `behav5` and you will see that the results are the same as for `behav4`.

There are two ways to fix the problems encountered in the `behav4` example. One way is to use variables for `01`, `02`, and `03` as in architecture `behav2` in the previous subsection. Values are assigned to variables within the current iteration.

Another way to fix the simulation problem in architecture `behav4` is to make the process sensitive to changes on signals `O1`, `O2`, and `O3` as shown in the following example. This causes the process to execute again within the same timestep (but during later iterations) when `O1`, `O2`, or `O3` is scheduled to change in 0 ns.

```
ARCHITECTURE behav4_fixed OF aoi IS
  SIGNAL O1,O2,O3: my_qsim_state;

BEGIN
  PROCESS (A, B, C, D, O1, O2, O3)
  BEGIN
    O1 <= A AND B  AFTER 0 ns;
    O2 <= C AND D  AFTER 0 ns;
    O3 <= O1 OR O2 AFTER 0 ns;
    E  <= NOT O3   AFTER 0 ns;
  END PROCESS;
END behav4_fixed;
```

Using this revised architecture, follow through the same steps as before. The signal states during initialization of the `behav4_fixed` architecture is as follows:

```
INIT.     X  X  X  X  X
          ^A ^B ^C ^D ^E
```

At time zero all inputs are forced to ’0’. The following shows a simulator result (using a resolution of 0.1 ns per timestep) for this condition:

```
Time 0.0  0  0  0  0  1   #END OF TIMESTEP
          ^A ^B ^C ^D ^E
```

Note that this time the output `E` changed to ’1’ as expected. As in the`behav4` example, when all the inputs are forced to ’0’, the sensitivity list activates the process in timestep 0.0, iteration 1. The four sequential statements execute as previously described. The last statement in iteration 1 to execute is the implied wait statement from the sensitivity list. The implied wait statement is where architectures `behav4_fixed` and `behav4` behave differently.

The sensitivity list now includes `O1`, `O2`, and `O3`, which have events scheduled to occur after 0 ns. Adding these signals means that the process will again trigger during timestep 0.0 (iteration 2) and the sequential statements are executed a second time. The results of iteration 2 cause yet another event to schedule after 0 ns (within the current timestep). Because the process is sensitive to the signal that changed during iteration 2, another iteration is required to finish the timestep. The complete simulation results are shown as follows:

```
Time 0.0  0  0  0  0  X   X   X   X   #BEGINNING OF IT 1
Time 0.0  0  0  0  0  X   0   0   X   #BEGINNING OF IT 2
Time 0.0  0  0  0  0  X   0   0   0   #BEGINNING OF IT 3
Time 0.0  0  0  0  0  1   0   0   0   #END OF TIMESTEP
          ^A ^B ^C ^D ^E  ^O1 ^O2 ^O3
```

From this example you can see that the more efficient way (only one iteration required) to fix the `behav4` architecture is to use variables for `O1` through `O3` instead of signals within the process as shown in the `behav2` architecture.

Another example is included to show how the use of a signal within a loop can produce results that were not intended. The following architectures (`arch_err` and `arch_ok`) use a signal called "t" to hold the integer sum of the contents of the 10 elements of array "arr_a" (line 12). If each element of `arr_a` equals the integer value 1, you might expect the sum to equal 10 after the loop in lines 11 through 13 finishes executing for the architecture `arch_err` on the left. For reasons previously described, `t` will contain the integer value 1, not 10, after execution.

```
 1   ARCHITECTURE arch_err
 2       OF ch_val IS
 3    TYPE ar1 IS ARRAY
 4       (1 TO 10) OF integer;
 5    SIGNAL t: integer := 0;
 6    SIGNAL arr_a : ar1;
 7
 8   BEGIN  --INCORRECT
 9    PROCESS (cntrl)
10    BEGIN
11      FOR i IN 1 TO 10 LOOP
12       t <= t + arr_a(i);
13      END LOOP;
14    END PROCESS;
15   END arch_err;
```

```
 1   ARCHITECTURE arch_ok
 2       OF ch_val IS
 3    TYPE ar1 IS ARRAY
 4       (1 TO 10) OF integer;
 5    SIGNAL t: integer := 0;
 6    SIGNAL arr_a : ar1;
 7
 8   BEGIN    --CORRECT
 9    PROCESS (cntrl)
10    VARIABLE a: integer:= 0;
11    BEGIN
12     FOR i IN 1 TO 10 LOOP
13      a := a + arr_a(i);
14     END LOOP;
15     t <= a;
16    END PROCESS;
17   END arch_ok;
```

The sequential signal assignment in line 12 for architecture `arch_err` on the left does not update the value of `t` during each pass through the loop. When line 12 is executed, the value for `t` is *scheduled to change* after 0 ns. During all 10 passes through the loop, the value for `t` on the right side of the expression remains 0.

To correct the behavior, declare a variable (a) to compute and hold the sum within the loop. Then assign that value (a) to signal `t` outside the loop as done in `arch_ok` on the right. When simulating architecture `arch_ok`, if each element of `arr_a` equals the integer value 1, the value for `t` (line 15) is *scheduled to change* after 0 ns to integer value 10.

Resolving a Signal Value When Driven by Multiple Assignment Statements

Every signal assignment statement assigns a projected waveform to a driver. It is possible (and probable in hardware designs) that your model contains more than one signal assignment statement (each with its own driver) that attempts to assign different values to the same signal at the same time. When this happens, your model must provide a resolution function that specifies how to resolve the assignment.

Each signal that requires a resolution function makes reference to the appropriate function in the signal declaration. For example, the following signal declaration refers to a resolution function called "wired_or":

```
SIGNAL total : wired_or integer ;
```

For more information on resolution functions, refer to "Multiple Drivers and Resolution Functions" in the *Mentor Graphics VHDL Reference Manual*.

Creating Shared Modules--Packages

VHDL provides the package construct to allow you to group a collection of related items for use by one or more separate modules of code. Among the items that can be grouped together in packages are:

- Type and subtype declarations
- Subprograms (functions and procedures)
- Constants
- Signals

Figure 4-18 shows how you might use a package in a hardware description. One package definition is coupled to two separate modules within a hardware design. Packages directly support the principles of modularity, abstraction, and information-hiding.

Packages can be compiled and stored separately from the rest of the hardware description (in a design file) to facilitate sharing between hardware designs. Entity declarations and architecture bodies also share this capability. It also makes a hardware design easier to manage if a logical collection of resources is stored separately from the main hardware description. A good example of these features is with the standard VHDL package called "standard".

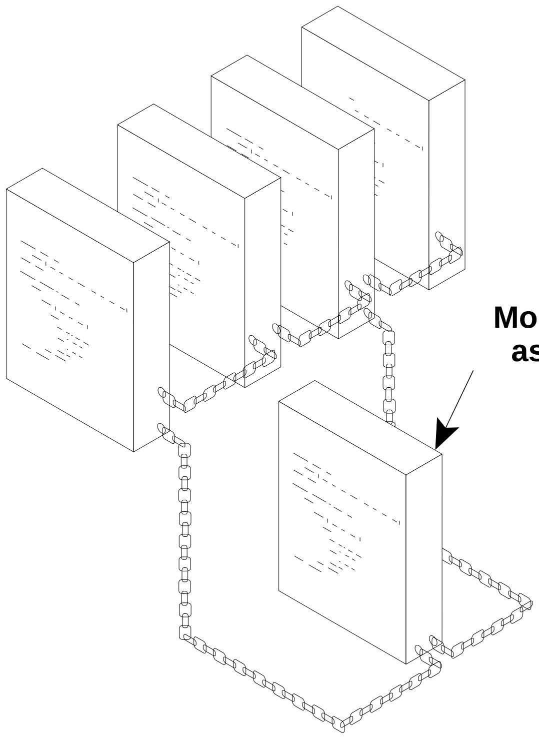
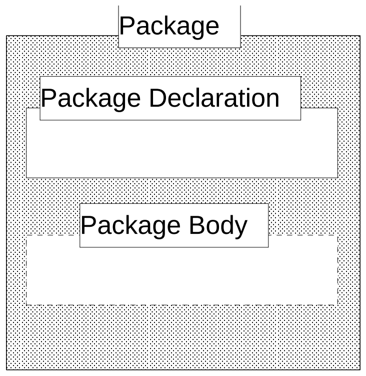


Figure 4-18. Shared Module of Code Defined as a Package

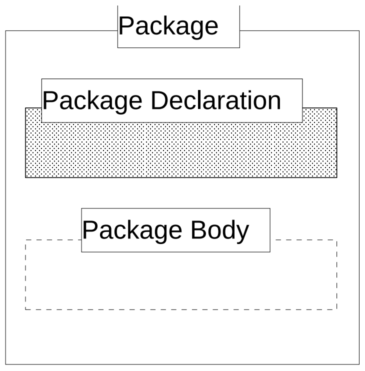
The package "standard" predefines a number of things consisting of types, subtypes, and functions. Figure 4-19 shows a portion of the standard package code (predefined enumeration types and predefined array types). The contents of the package "standard" are described in the "Predefined Packages" section of the *Mentor Graphics VHDL Reference Manual*. Note the type definitions for `bit` and `bit_vector` that are used in previous code examples. You can create your own packages to serve a similar purpose.

```
PACKAGE standard_portion IS
  --predefined enumeration types:
    TYPE bit IS ('0', '1');
    TYPE boolean IS ( false, true );
  --predefined array types
    TYPE string IS ARRAY ( positive RANGE <> ) OF character;
    TYPE bit_vector IS ARRAY ( natural RANGE <> ) OF bit;
 END standard_portion;
```

Figure 4-19. Portion of Code from a Package Called Standard

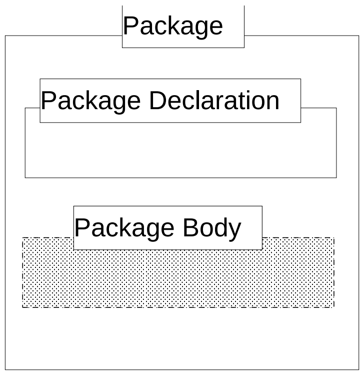


Packages can be defined in two parts: a package declaration (also called a package header) and a package body. This two part structure is common to the design entity construction described on page 2-5. You can create a package using only a package declaration. The package body is not required (unless the package header contains deferred constants or subprogram declarations). A package declaration can be stored and compiled in a separate design file from the package body.



A package declaration, like that in Figure 4-19, defines the contents that are visible outside of the package. The package declaration begins with the reserved word **package** as shown in the following example:

package identifier **is**
 package_declarative_part
end identifier**;**



A package body contains program details that are not visible outside of the package. It begins with the reserved words **package body** as shown in the following example:

package body identifier **is**
 package_body_declarative_part
end identifier**;**

The package body is not always required.

Anything you declare in a package declaration can be made visible to design files outside of the package. You must compile the package header before you compile any code that uses the package. The package body does not need to be compiled until you are ready to simulate the design that uses the package.

If you want to make a change to a package header after a corresponding entity declaration or architecture body has been compiled, you must recompile the package header, package body (if used), entity declaration (if it contains a use clause for or direct reference to an item in the changed package), and architecture body. Making a change in the package body does not mean you have to recompile the package header, entity declaration, and architecture body. This is a good reason to separate the package header and package body into separate design files. There is more chance that you would make a change to the code in a package body than modify the declarations in the package header.

To illustrate how this affects the way you model with VHDL, consider the following example. Figure 4-20 shows a VHDL model of a traffic light controller that is located in four design files. The entity declaration (in file *entity*) contains a library clause and a use clause that makes the `tlc_types` package visible to the `tlc` entity declaration and associated architecture body.

The package declaration is located in a file called *pkg_hdr* and the package body is located in a file called *pkg_body*. The architecture body in file *arch1* contains a procedure call to a procedure that is located in the package.

Before the tlc design is simulated, the files are compiled: first *pkg_hdr*, second *entity*, third *arch1*, and finally *pkg_body*. If during simulation you note that a change needs to be made in the procedure body (in file *pkg_body*), only file *pkg_body* needs to be recompiled before running the simulation again.

Assume that during the second simulation you decide that the procedure could better serve the model if an additional parameter is added along with a few more lines of code. This time the procedure specification must be changed in both the package header (in file *pkg_hdr*) and the package body (in file *pkg_body*). Once these changes are made, all the design files must again be compiled as before the first simulation. Adding or changing a declaration in the package header file affects all design units that use that package. They must all be recompiled.

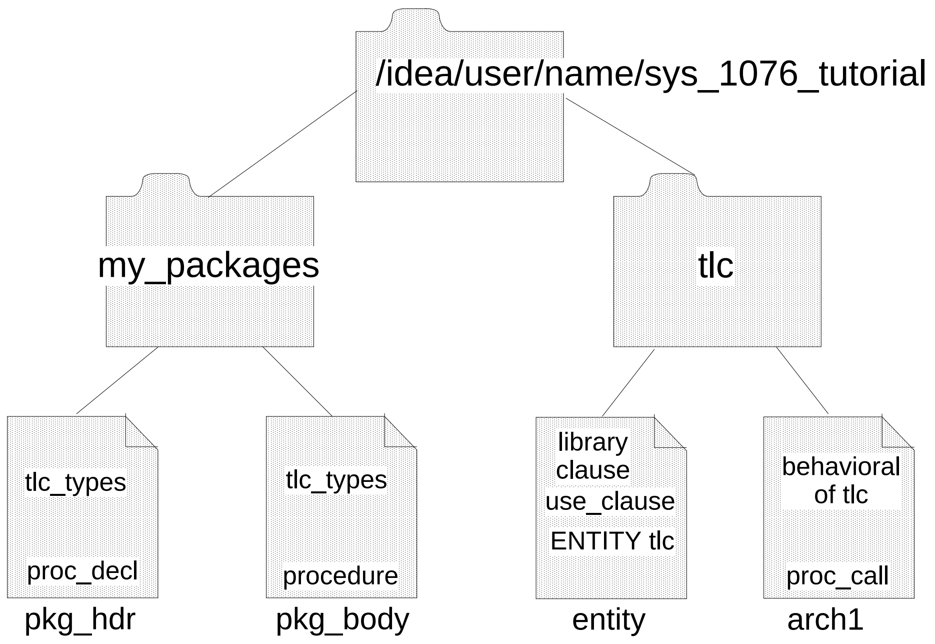


Figure 4-20. Effects on Entity and Architecture When Changing Package Header

Making a Package Visible--Library Clause and Use Clause

To make a package or another design entity visible to the current code abstraction, you first need to write a library clause to identify where the library containing the given design entity or package is. Then you need to write a use clause to make selected declarations in the library visible at the current level of abstraction.

Figure 4-21 shows two library directories; */idea/user/package_lib* and */idea/user/vhdlsim_lib*. Under the *vhdlsim_lib* directory is a directory for a VHDL design entity (*and2*) and a corresponding package (*package2*). The package1 and package2 databases are made visible to the and2 design entity by

1. Placing a library clause at the head of the design file *entity* to identify a logical name for each referenced library.

2. Designating a logical-to-physical map to identify where in the directory structure the physical library (directory) resides. (Each VHDL implementation has a specific method for doing this map.)

3. Putting a use clause where the specific package contents are required.

The library clause and use clause have the following format:

library clause	**library** logical_name_list **;**
use clause	**use** prefix**.**suffix**,** prefix**.**suffix**,** ... **;**

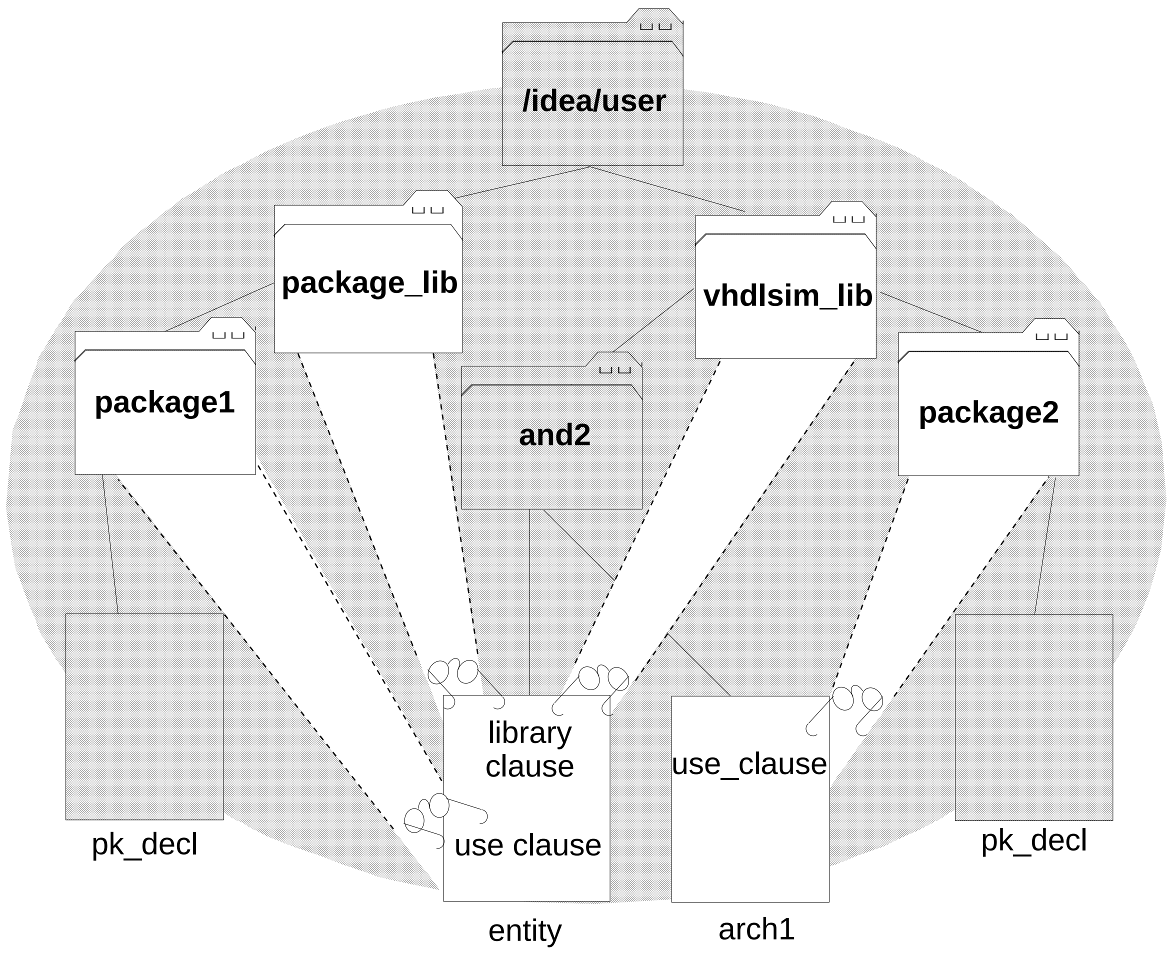


Figure 4-21. Making Packages Visible with Library and Use Clauses

The design file for *entity* in Figure 4-21 might look like the following:

```
LIBRARY package_lib, vhdlsim_lib;--logical library names
USE package_lib.package1.ALL

ENTITY test IS
  PORT (a : IN  my_lsim_logic;  q : OUT my_lsim_logic);
END test;
```

To access a package declaration and declarations within the package, you insert a use clause in the body of a procedure, function, design entity, another package, etc.

To make a package visible at a given abstraction level, you include the logical library name, followed by the desired package name as the prefix and the desired internal declaration as the suffix (or all of the internal package declarations can be called by using the word **all** as the suffix). As an example, look at the following use clause:

```
   USE lib_stf.ram.ALL; --use clause to access ram package
```

This use clause makes all the declarations in a RAM description package visible and accessible to a procedure called `ram_load`. Figure 4-22 shows how the use clauses in the `ram_read` and `ram_load` procedures couple them to the RAM description package.

When compiling code that contains logical references to libraries, you must supply a logical-to-physical map for each library name as determined by the particular VHDL implementation.

If you needed to access just 2 out of 20 routines from a particular package, it would be more efficient to make visible just the desired routines, not the whole package. It would also make your code easier to understand if the use clause specifically identified which routines you were using from a package.

As an example, assume a package called `mem_ops` contains 20 procedures and functions (including those shown in Figure 4-22). You need only `ram_load` and `ram_read`. The following use clause specifies the logical library name, package name, and the particular procedure that you want to make visible:

```
USE lib_stf.mem_ops.ram_load,   -- library.package.procedure
     lib_stf.mem_ops.ram_read;   -- library.package.procedure
```

This use clause makes only the two named procedures visible to your current level of abstraction. It also makes it clear which operations you are using from the `mem_ops` package.

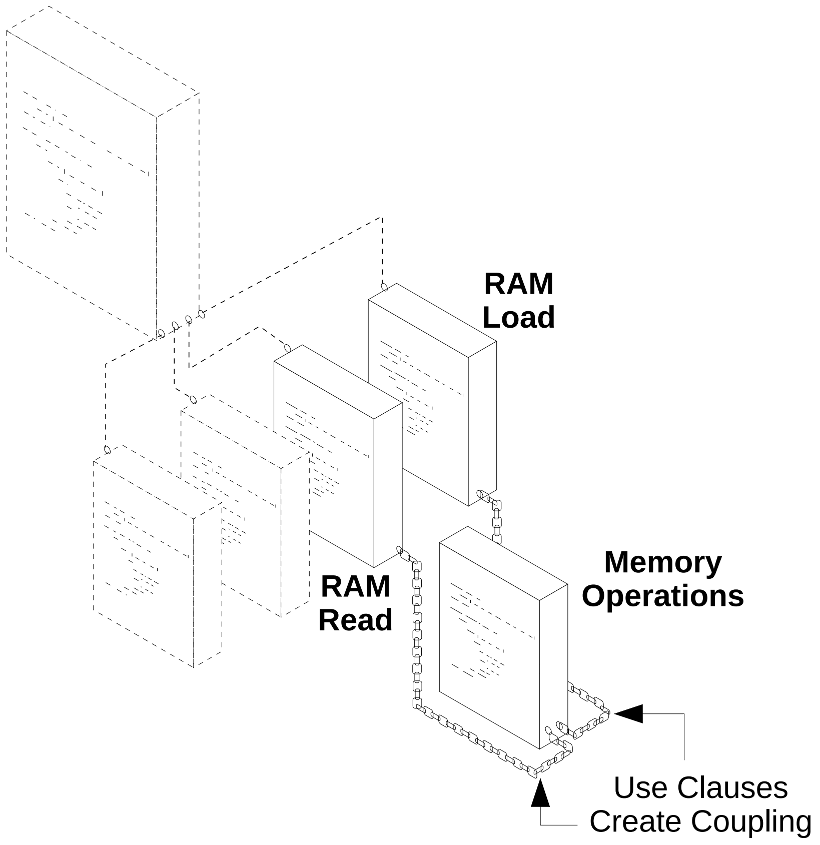


Figure 4-22. Coupling Two Procedures to One Package with Use Clauses

Section 5
Global Considerations

This section describes some of the general issues you must consider when designing and modeling with VHDL. The topics in this section include the following:

Scope and Visibility

When decomposing large hardware descriptions into smaller units (such as packages, subprograms, processes, and blocks), you must consider each declaration and how it relates to the whole description. For example, if you want to declare a signal that is accessed globally throughout the hardware description, you must position it in the proper location within your code. You must also be sure that you do not use the same name for a different purpose elsewhere in your code.

This subsection describes some of the rules you must be aware of that govern the region of text where a declaration has effect (*scope*) and which areas of text a declaration is visible (*visibility*). In general, the scope of a declared identifier starts at the point where the identifier is first named and extends to the end of the description unit (subprogram, block, package, process) that contains the declaration. This concept is best understood by examining the example in Figure 5-1.

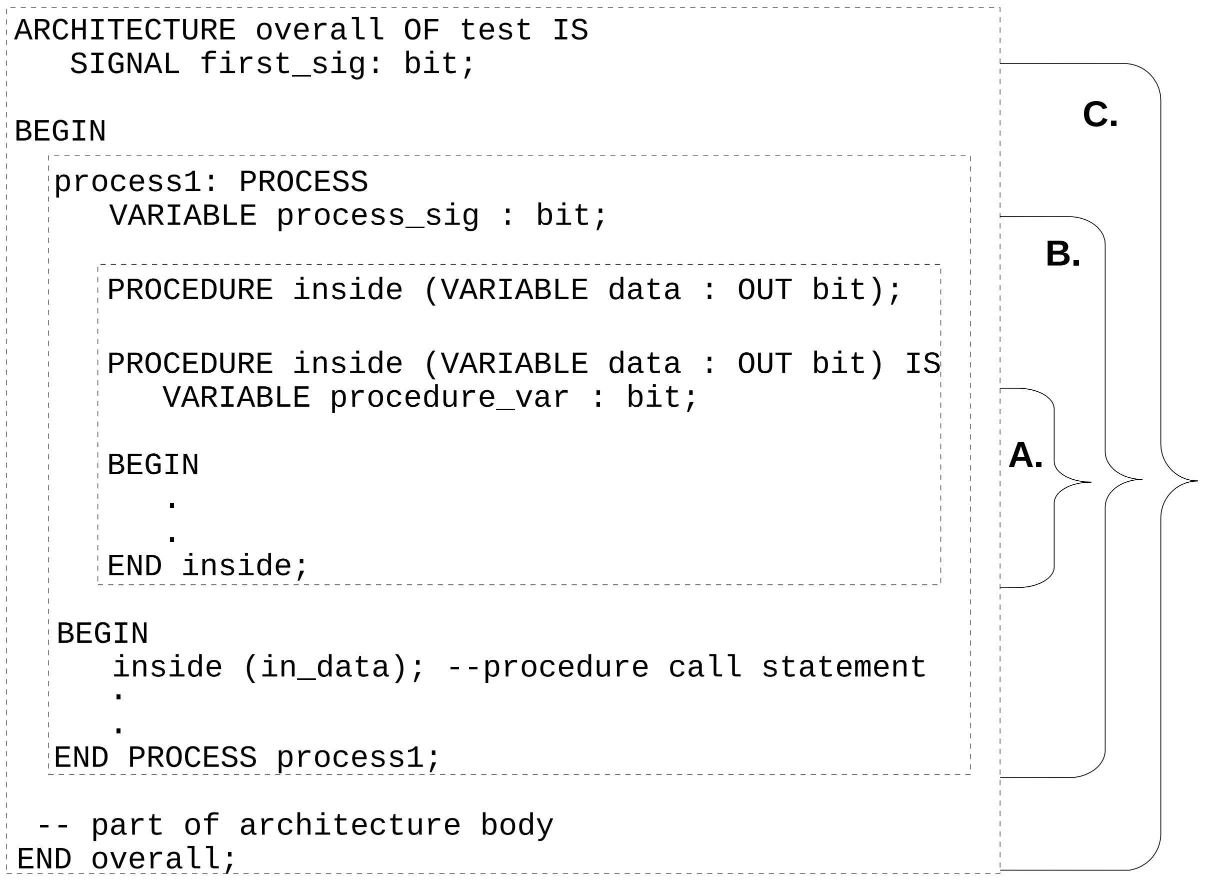
Figure 5-1 shows an architecture body that contains a procedure within a process. In this example, a signal and various variables are declared in the three different text regions identified as A, B, and C.

The signal declared in the architecture declarative part (`first_sig`) has a scope identified by the region labeled C. In this example, `first_sig` is directly visible to all regions within region C. This means that `first_sig` could be used in the process called `process1` and the procedure called `inside`.

The process declarative part in `process1` declares a variable called `process_sig`. This variable has a scope identified by region B in Figure 5-1. Variable `process_sig` is directly visible inside region B but is not visible outside of this region.

A subprogram (the procedure called `inside`) is also declared in the `process1` process declarative part. This makes the procedure directly visible to the process statement part, but declarations within the procedure (such as `procedure_var`) are not directly visible in region B. Even though the procedure variable `data` (formal part) is associated with the actual part `in_data` in the procedure call, `data` is not directly visible to region B. Refer to the "Procedure Call" subsection on page 4-27 and the "Function Call" subsection on page 4-24 for more information on formal-to-actual part association.

In the subprogram declarative part for the procedure in Figure 5-1, one variable is declared (`procedure_var`). This declaration is directly visible only within the scope defined by region A.



```
ARCHITECTURE overall OF test IS
   SIGNAL first_sig: bit;

BEGIN
  process1: PROCESS
     VARIABLE process_sig : bit;

     PROCEDURE inside (VARIABLE data : OUT bit);

     PROCEDURE inside (VARIABLE data : OUT bit) IS
        VARIABLE procedure_var : bit;

     BEGIN
        .
        .
     END inside;

  BEGIN
     inside (in_data); --procedure call statement
     .
     .
  END PROCESS process1;

 -- part of architecture body
END overall;
```

Figure 5-1. Example of Scope and Visibility

Now that you have examined some of the basic scope and visibility rules, look at the following features that allow you more visibility control over each declared item. The remainder of this subsection shows you how to make a declared variable in region B from Figure 5-1 visible to the architecture statement part in region C. Also described is how to hide a declaration in region C from region B.

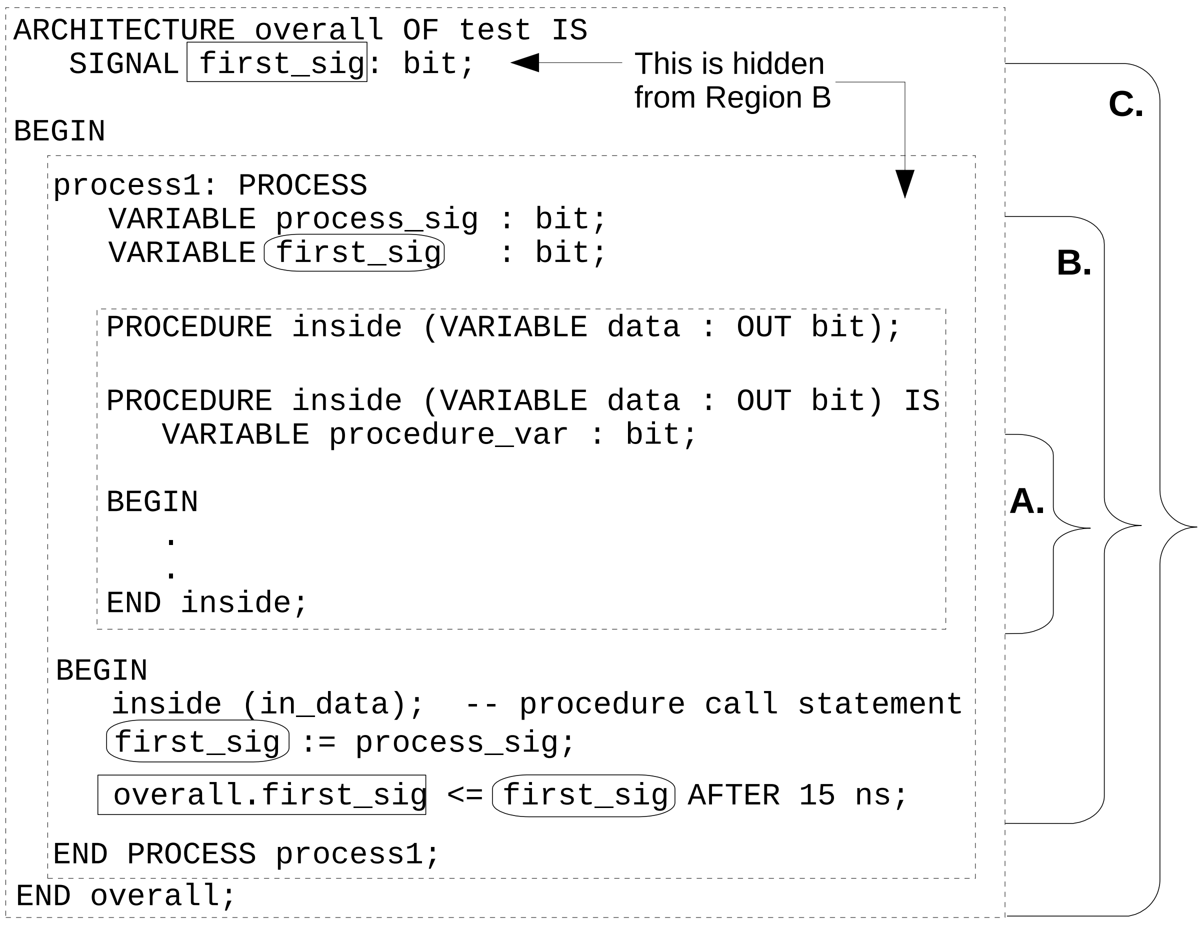
Figure 5-2 repeats the example from Figure 5-1 but adds text to demonstrate how a global declaration (region C) can be hidden from the inner text regions (regions B and A). As previously described, the signal `first_sig` in the architecture declarative part of Figure 5-2 is normally visible to all regions within region C. This signal name is bounded by a rectangle to differentiate it from the variable of the same name which is defined in `process1`.

In Figure 5-2, a variable with the name `first_sig` is declared in the process declarative part within region B. This variable is bounded by an oval. The figure uses the square and oval boundaries to help identify which signal/variable is used in the signal assignment statements within the process statement part.

The variable `first_sig` in the process declarative part (bounded by an oval) and the signal `first_sig` in the architecture declarative part (bounded by a rectangle) are homographs* of each other. This relationship causes signal `first_sig` in the architecture declarative part to be hidden from inner region B. The `first_sig` signal bounded by a rectangle is no longer directly visible throughout region B. The `first_sig` variable bounded by an oval takes precedence within region B.

The signal assignment statement (following the procedure call) in the process statement part assigns the value of `process_sig` to the region B `first_sig` variable.

*Refer to the *Mentor Graphics VHDL Reference Manual* for more information on homographs.



```
ARCHITECTURE overall OF test IS
   SIGNAL first_sig: bit;

BEGIN

  process1: PROCESS
     VARIABLE process_sig : bit;
     VARIABLE first_sig   : bit;

     PROCEDURE inside (VARIABLE data : OUT bit);

     PROCEDURE inside (VARIABLE data : OUT bit) IS
        VARIABLE procedure_var : bit;

     BEGIN
        .
        .
     END inside;

  BEGIN
     inside (in_data);  -- procedure call statement
     first_sig := process_sig;
     overall.first_sig <= first_sig AFTER 15 ns;
  END PROCESS process1;
END overall;
```

Figure 5-2. Hiding a Declaration Using a Homograph

The signal assignment statement shows how you can make the signal `first_sig` (from region C) visible once it has been hidden from the inner regions. To specifically access this signal, the architecture body name is used as a prefix (`overall`), followed by a period, and then the signal name (`first_sig`). The result (`overall.first_sig`) is shown in Figure 5-2 bounded by a rectangle because it corresponds with the declaration at the top of the figure also bounded by a rectangle. The signal `overall.first_sig` takes on the value of variable `first_sig` after 15 ns.

In this subsection you have seen how to use the same declaration identifier purposefully in more than one location to perform specific functions. Overloading is another related topic, which is described in the next subsection.

Reusing Predefined Names --Overloading

When creating large design descriptions, you may want to reuse a predefined identifier for either an enumeration literal or a subprogram (function or procedure). A particular identifier might clearly state the intended purpose for more than one function, procedure, or enumeration literal. The VHDL method to reuse these predefined names is called overloading. It is also possible to overload operators such as "+", "=", and so on. The following topics describe overloading for each of these constructs.

Overloading Enumeration Literals

This subsection describes how to overload enumeration literals. The following example shows how enumeration literals `red` and `green` are overloaded by appearing in two separate enumeration definitions in an area of code that has an overlapping scope.

```
TYPE wire_color IS(green,black,red); --custom enumeration type
TYPE traffic_lt IS(red, yellow, green, flashing); --Overloaded
```

In this example, it does not make sense to create unique literals because the overloaded literals specifically describe the intentions for each type declaration. Once these declarations have been made (repeated in the architecture declarative part, lines 3 and 4, of Figure 5-3), objects of these types can be declared (see the process declarative part, lines 9 and 10, of Figure 5-3) and then values can be assigned (see the process statement part, lines 13 and 14, of Figure 5-3).

The compiler uses the type of `pwr_hot` and `top_lt` in Figure 5-3 to determine which literal `red` is assigned to each object. Ambiguity is avoided by using the context of the statements (`pwr_hot := red;`) and (`top_lt := red;`).

```
ARCHITECTURE physical OF hardware IS    --architecture
 SIGNAL sig1: bit;                      --declarative part
 TYPE wire_color IS (green, black, red);
 TYPE traffic_lt IS (red, yellow, green, flashing);

BEGIN                 -- architecture statement part
 example :                    -- process label
 PROCESS (sig1)               -- process declarative part
  VARIABLE pwr_hot, pwr_neutral, pwr_ground : wire_color;
  VARIABLE top_lt, middle_lt, bottom_lt : traffic_lt;

 BEGIN                    -- process statement part
  pwr_hot := red ;          -- from type "wire_color"
  top_lt := red ;           -- from type "traffic_lt"
 END PROCESS example;
END hardware;
```

Figure 5-3. Overloading Enumeration Literals

Overloading Subprograms

As previously mentioned, procedures and functions can also be overloaded. To illustrate this feature, an example package is shown in Figure 5-4. The related type declarations are shown at the top of the figure, followed by four subprogram specifications (functions), each with the same designator `my_to_qsim_12state`. These are four unique functions that each have a similar purpose: to convert a qsim value to a qsim 12-state value. The subprogram body of each function is not shown.

The first `to_my_qsim_12state` function (line 19) receives the value of object `val` from the calling code. The object `val` must have a value of the type `my_qsim_value`, which is defined as either 'X', '0', or '1'. The next object value received by this function is `str`, which has a type of `my_qsim_strength` (either 'Z', 'R', 'S', or 'I'). This function returns a value (to the calling code) of the type `my_qsim_12state`.

The second `my_to_qsim_12state` function (line 22) looks for a value of object `val` of the type `my_qsim_state`, which is defined as either 'X', '0', '1', or 'Z'. This function returns a value (to the calling code) of the type `my_qsim_12state`, as does the first function.

The third (line 24) and fourth (line 27) functions are the same as the first and second functions, respectively, except that they expect vectors rather than single object values.

The compiler determines which `my_to_qsim_12state` function to use by the context of the calling code. Figure 5-5 shows an example of how to specifically call the first and second function from Figure 5-4. Also included in Figure 5-5 is a function call that cannot be resolved and that generates an error.

Line 3 of Figure 5-5 makes the package "my_qsim_logic" visible to this process. The `my_lib.` prefix is required to specify in which library this package is located. A library clause is required (not shown) in the same design unit as the process to define the logical name for the library where the package resides.

In the process declarative part of Figure 5-5 (lines 3 through 8), a number of variables of various types are declared (`var1`, `var2`, `var3`), and values are assigned to each. One constant is declared (`Sig_strngth` in line 4) to represent a signal strength value of "strong". Variable `reg_in_a` is declared (line 8) to be of the type `my_qsim_12state`.

```
PACKAGE my_qsim_logic IS
TYPE my_qsim_state IS ('X','0','1','Z');-- X:Unknown logic level
 SUBTYPE my_qsim_value IS my_qsim_state RANGE  'X' TO '1' ;

                                  --Leading S: denotes qsim
state
TYPE my_qsim_12state IS(SXR,SXZ,SXS,--'Z': Hi imped. sig.
strngth
                     SXI,S0R,S0Z,--'R': Resistive sig. strngth
                     S0S,S0I,S1R,--Trailing 'S':Strong sig.
strth
                     S1Z,S1S,S1I);--'I': Indeterminate sig.
strth
TYPE my_qsim_strength IS ('Z','R','S','I');
TYPE my_qsim_state_vector IS ARRAY(positive RANGE<>) OF

my_qsim_state;
TYPE my_qsim_value_vector IS ARRAY(natural RANGE <>) OF

my_qsim_value;
TYPE my_qsim_12state_vector IS ARRAY(natural RANGE<>)
                            OF my_qsim_12state ;
TYPE my_qsim_strength_vector IS ARRAY(natural RANGE<>)
                            OF my_qsim_strength;
FUNCTION my_to_qsim_12state (val : my_qsim_value;          -- 1.
                            str : my_qsim_strength)
                           RETURN my_qsim_12state;
FUNCTION my_to_qsim_12state (val : my_qsim_state)          -- 2.
                           RETURN my_qsim_12state;
FUNCTION my_to_qsim_12state (val : my_qsim_value_vector;   -- 3.
                            str : my_qsim_strength_vector)
                           RETURN my_qsim_12state_vector
FUNCTION my_to_qsim_12state (val : my_qsim_state_vector)   -- 4.
                           RETURN my_qsim_12state_vector;
END my_qsim_logic;
```

Figure 5-4. Overloading Subprograms--Functions

The first statement (line 11) in the process statement part of Figure 5-5 assigns the `my_qsim_value` `’0’` and `my_qsim_strength` value `’S’` to variable `reg_in_a` by calling function `my_to_qsim_12state`. The function is required to perform the conversion because it is not possible to assign values of one type to a variable declared as a different type.

Because the first function call passes a value of type `my_qsim_value` and a value of type `my_qsim_strength`, the compiler matches this function call with the first function of Figure 5-4. None of the other functions (lines 24 through 28 of Figure 5-4) matches this function call so the ambiguity of using the same name for four different functions is resolved.

```
 1   example2 :                      -- process label
 2   PROCESS (sig1)                  -- process declarative part
 3     USE my_lib.my_qsim_logic.ALL;
 4     CONSTANT Sig_strngth : my_qsim_strength := ’S’;
 5     VARIABLE var1        : my_qsim_value    := ’0’;
 6     VARIABLE var2        : my_qsim_state    := ’Z’;
 7     VARIABLE var3        : bit              := ’0’;
 8     VARIABLE reg_in_a    : my_qsim_12state;
 9
10   BEGIN                            -- process statement part
11     reg_in_a := my_to_qsim_12state(var1, Sig_strngth);--  #1
12     reg_in_a := my_to_qsim_12state(var2);   --Uses funct. #2
13     reg_in_a := my_to_qsim_12state(var3, Sig_strngth); --ERR
14   END PROCESS example2;
```

Figure 5-5. Calling a Specific Overloaded Subprogram--Function

The second statement (line 12 of Figure 5-5) assigns the `my_qsim_state` value `’Z’` to the variable `reg_in_a` by calling function `my_to_qsim_12state` again. In this case, the function call passes only one value of type `my_qsim_state`, so the compiler matches this function call with the second function (line 24) of Figure 5-4. None of the other functions matches this function call, so again the ambiguity is resolved.

The third statement (line 13 of Figure 5-5) attempts to assign the `bit` value `’0’` and `my_qsim_strength` value `’S’` to the variable `reg_in_a` by calling function `my_to_qsim_12state`. If you look at all four functions in Figure 5-4, you will not find any that expect a `bit` value as an argument. Even though the value of `var3` is `’0’`, it belongs to a type not recognized by any of the functions with the name `my_to_qsim_12state`. In this case, a compiler cannot resolve the ambiguity and an error is generated.

Overloading Operators

Any predefined operator can be overloaded. Table 5-1 shows all the predefined operators by operator class. The operator classes are listed in the order of precedence from highest precedence (top) to lowest precedence (bottom). All the predefined operators in a given operator class have the same precedence.

Table 5-1. Operators by Precedence

Operator Class	**Binary Operators**	**Unary Operators**
Miscellaneous	**	**abs not**
Multiplying	*** / mod rem**	
Sign		**+ -**
Adding	**+ - &**	
Relational	= /= < <= > >=	
Logical	**and or nand nor xor**	

An operator is overloaded by defining a function; however, the function format is slightly different from the examples shown in the previous subsection on overloading subprograms. Figure 5-6 shows how the operator "=" is overloaded in the a package called "my_qsim_logic". Notice the symbol "=" (a string literal) is enclosed by double quotes. When an operator is overloaded, it must be enclosed by double quotes.

```
1   PACKAGE my_qsim_logic IS
2    TYPE my_qsim_state IS ('X', '0', '1', 'Z');
3    FUNCTION "="(l,r: my_qsim_state)RETURN my_qsim_state;--1
4    FUNCTION "="(l,r: my_qsim_state)RETURN boolean;      --2
5   END my_qsim_logic;
```

Figure 5-6. Overloading Operators

Because the operator "=" is a binary operator (see Table 5-1) it must have two parameters. The parameter "l" accepts the left primary and the parameter "r" accepts the right primary when the overloaded operator symbol is used in an expression. Both parameters are of type `my_qsim_state`.

Figure 5-7 shows how one of the overloaded "=" operators is chosen from the context of the calling code. The variable `result1` is declared to be of type `my_qsim_state` (line 6). When the first expression in line 10 `(var1 = var2)` is evaluated and assigned to variable `result1`, it must return a value of type `my_qsim_state`. Therefore, a VHDL compiler matches the "=" operator with the first function shown in line 3 of Figure 5-6. Variable `result1` receives a value of '0'.

The overloaded operator function is called by using the operator notation (left_parameter = right_parameter). An equivalent function call to the "=" function would look like the following:

```
 result1 :=  "="(var1, var2);--equivalent function call to "="
```

The second expression in Figure 5-7 (line 11) expects variable `result2` to receive a Boolean value when `(var1 = var2)` is evaluated because `result2` is declared as type Boolean in line 7. A VHDL compiler matches the "=" operator with the second function shown in line 4 of Figure 5-6. Variable `result2` receives a value of 'F'. Signal`sig1` in line 16 does not pertain to this discussion but is required in the process sensitivity list in line 2.

```
example3 :                     -- process label
PROCESS (sig1)                 -- process declarative part
  USE my_lib.qsim_logic.ALL;
  VARIABLE var1    : my_qsim_state := ’0’;
  VARIABLE var2    : my_qsim_state := ’1’;
  VARIABLE result1 : my_qsim_state;
  VARIABLE result2 : boolean;

BEGIN                      -- process statement part
 result1:= (var1 = var2); --Uses funct #1 (result1 = ’0’)
 result2:= (var1 = var2); --Uses funct #2 (result2 = ’F’)
                            -- (Remaining code shows other
 var2   := ’0’;             --  possible results.)
 result1:= (var1 = var2);  --Uses funct #1 (result1 = ’1’)
 result2:= (var1 = var2);  --Uses funct #2 (result2 = ’T’)
 sig1   <= result1;
END PROCESS example3;
```

Figure 5-7. Calling a Specific Overloaded Operator

Section 6 Coding Techniques

This section presents the following tasks that a modeler might want to accomplish and certain considerations when modeling with VHDL:

General VHDL Coding Guidelines

VHDL provides many constructs for the description of complex hardware systems. It is how you use these constructs, that is, your coding style, that determines the readability of your source code. This subsection presents some general guidelines that you can use to keep your description of a complex hardware system readable.

The following list summarizes a suggested style of presenting code to improve readability:

- Group logically related declarations and statements. Use blocks to group concurrent statements. Use packages to group the following:
  - ❍ Subprograms (functions and procedures)
  - ❍ Type and subtype declarations
  - ❍ Signals and constants
- Use indentation to show nesting and subordination.
- Use white space (blank lines) to separate logical code divisions.
- Line up similar words (such as reserved words) or punctuation such as colons on adjacent lines of code.
- Use comments to describe code functions that are not obvious.
- Use all uppercase characters for reserved words and all lowercase characters for user-defined identifiers.
- Use capitalized words (first character uppercase) for generic and constant identifiers.
- Use the optional label (identifier) for processes, concurrent signal assignment statements, concurrent procedure calls, etc., to give them a descriptive name.
- Enter code with the reader in mind. You enter code once, but it may be read and reviewed many times.

Figure 6-1 shows a variation of the shifter example, used earlier in this section, that uses these coding guidelines to improve readability. The code has been divided and grouped into three separate files. This separation gives the added benefit of being able to compile and maintain each of these files separately. Packages (`my_ qsim_logic` and `standard`) are used to group the type declarations. Indentation shows the subordination. Blank lines help break the code into visually discernible units.

Wherever possible, text is aligned in columns such, as lines 5 through 8 of the *my_qsim_logic* file and lines 10 through 13 of the *shifter_architecture* file.

Comments are used in the examples found throughout this section to help familiarize you with how to identify basic constructs in real code, such as the port clause comment in line 6 of the *shifter_entity file*. In your code, use comments to help describe code functionality in areas that are not clear.

Compare the readability of the code in Figure 6-1 with the equivalent code in Figure 6-2. The code in Figure 6-2 ignores all but one guideline (capitalizing reserved words). The code may perform as expected, but it is not very easy to read. If reserved words were not capitalized, it would be almost impossible to read.

```
-----------STORED IN FILE my_qsim_logic--------------------
PACKAGE my_qsim_logic IS --Following is a portion of qsim_logic

                         --Leading ’S’: denotes qsim state
TYPE my_qsim_12state IS(SXR,SXZ,SXS,--’Z’: Hi imped. sig. strngth
                     SXI,S0R,S0Z,--’R’: Resistive sig. strength
                     S0S,S0I,S1R,--Trailing ’S’:Strong sig. strth
                     S1Z,S1S,S1I);--’I’:Indeterminate sig. strth
TYPE my_qsim_12state_vector IS ARRAY(natural RANGE <>)
                         OF my_qsim_12state;
END my_qsim_logic;
-----------------------------------------------------------------
```

```
------------STORED IN FILE shifter_entity-------------------
LIBRARY my_lib; USE my_lib.my_qsim_logic.ALL;

ENTITY shifter IS
  GENERIC(Prop_delay: time); --time declared in standard package
  PORT (shftin  : IN  my_qsim_12state_vector(0 TO 3); --port cl.
        shftout : OUT my_qsim_12state_vector(0 TO 3);
        shftctl : IN  my_qsim_12state_vector(0 TO 1));
END shifter;
-----------------------------------------------------------------
```

```
-----------STORED IN FILE shifter_architecture-------------
ARCHITECTURE behav1 OF shifter IS      -- architecture body
BEGIN
  shift_proc:                            --process statement
  PROCESS (shftin, shftctl)
    VARIABLE shifted : my_qsim_12state_vector(0 TO 3);

  BEGIN
    CASE shftctl is                     --process statement part
      WHEN (S0S, S0S) => shifted := shftin;
      WHEN (S0S, S1S) => shifted := shftin(1 TO 3) & S0S;
      WHEN (S1S, S0S) => shifted := S0S & shftin(0 TO 2);
      WHEN (S1S, S1S) => shifted := shftin(0) & shftin(0 TO 2);
    END CASE;
    shftout <= shifted AFTER Prop_delay;
  END PROCESS shift_proc;
END behav1;
-----------------------------------------------------------------
```

Figure 6-1. Good Presentation Style for Shifter Description

```
--------STORED IN FILE shifter-----------------------
TYPE my_qsim_12state IS (SXR,SXZ,SXS,SXI,S0R,S0Z,S0S,S0I,
S1R,S1Z,S1S,S1I);
TYPE my_qsim_12state_vector IS ARRAY(natural RANGE <>)
                         OF my_qsim_12state;
ENTITY shifter IS
GENERIC (prop_delay : time);
PORT (shftin : IN my_qsim_12state_vector(0 TO 3);
shftout : OUT my_qsim_12state_vector(0 TO 3);
shftctl : IN my_qsim_12state_vector(0 TO 1));
END shifter;
ARCHITECTURE behav1 OF shifter IS
BEGIN
PROCESS (shftin, shftctl)
VARIABLE shifted : my_qsim_12state_vector(0 TO 3);
BEGIN
CASE shftctl IS
WHEN (S0S, S0S) => shifted := shftin;
WHEN (S0S, S1S) => shifted := shftin(1 TO 3) & S0S;
WHEN (S1S, S0S) => shifted := S0S & shftin(0 TO 2);
WHEN (S1S, S1S) => shifted := shftin(0) & shftin(0 TO 2);
END CASE;
shftout <= shifted AFTER prop_delay;
END PROCESS;
END behav1;
----------------------------------------------------
```

Figure 6-2. Poor Presentation Style for Shifter Description

Various Techniques for Modeling Timing

VHDL provides a number of alternatives for modeling timing and other parameters that are dependent on different technologies. Parameters can be embedded in a model either as fixed values or variables, or the parameters can be customized outside the model. These techniques are described in the following subsections using a simple AND gate example.

Embedding Fixed-Delay Parameters Within a Model

If you want a model that describes one kind of technology, you may choose to embed delay values within the model. The following example of an AND gate (`and2_gate`) is a simple behavioral model that includes a propagation delay of 8 ns (lines 8 and 10). The behavior of this model might be accurate enough for use in certain simulations.

```
 1   ENTITY and2_gate IS
 2     PORT (in0, in1 : IN  bit;
 3               out1 : OUT bit );
 4   END and2_gate;
 5
 6
 7   ARCHITECTURE fixed_delay OF and2_gate IS
 8     CONSTANT Typical_delay : time := 8 ns;
 9   BEGIN
10    out1 <= in0 AND in1 AFTER Typical_delay; --fixed delay
11   END fixed_delay;                          -- of 8 ns
```

When dealing with more complicated models, you should declare a constant (as in line 8) to hold your fixed-delay value. Multiple statements in your model may use the delay value, but you can modify the value at a later time in just one place (the constant declaration) of the model. To change the delay value, you have to edit the file and recompile it.

Embedding Variable-Delay Parameters Within a Model

One step up in complexity (and accuracy) from the `fixed_delay` architecture in the previous example involves a model that has different propagation delays depending on the state of the output. The following example also embeds the delay information in the model but adds a variable-delay to the output.

```
 1   LIBRARY my_lib; --Define Logical library name.
 2   USE my_lib.logic_example.ALL; --Calls package that defines
 3                               --my_lsim_LOGIC. See Figure 6-15,
 4   ENTITY and2_gate IS         --                     page 6-28.
 5     PORT (in0, in1 : IN  my_lsim_LOGIC;
 6               out1 : OUT my_lsim_LOGIC );
 7   END and2_gate;
 8
 9
10   ARCHITECTURE variable_delay OF and2_gate IS
11    CONSTANT Tplh_typ: time:= 5 ns; --low-to-high typ. delay
12    CONSTANT Tphl_typ: time:= 8 ns; --high-to-low typ. delay
13
14   BEGIN
15     and_inputs : PROCESS (in0, in1)
16     BEGIN
17       IF (in0 AND in1) = ’1’ THEN
18         out1 <= ’1’ AFTER Tplh_typ;
19       ELSIF (in0 AND in1) = ’0’ THEN
20         out1 <= ’0’ AFTER Tphl_typ;
21       ELSIF (Tplh_typ >= Tphl_typ) THEN
22         out1 <= ’X’ AFTER Tplh_typ;
23       ELSE
24         out1 <= ’X’ AFTER Tphl_typ;
25       END IF;
26     END PROCESS and_inputs;
27   END variable_delay;
```

Figure 6-3. Embedding Variable-Delay Parameters Within Model

In the `variable_delay` architecture, two constants are declared: one to hold the delay value of a low-to-high transition (line 11), and one to hold the delay value of a high-to-low transition (line 12). These delay values are used in the process (`and_inputs`) that determine which state has occurred (with if/else conditions) at the inputs and sets the output accordingly. Wherever the constant `Tplh_typ` is used, a delay value of 5 ns occurs; similarly, wherever constant `Tphl_typ` is used, a delay value of 8 ns occurs.

This model does not actively take into account ’X’ (unknown) or ’Z’ (high impedance) states on the inputs. These conditions could be trapped in "if conditions" and appropriate action taken accordingly.

The next example uses generics instead of constants to gather and pass the delay information to the model architecture.

Using Generics to Parameterize a Model

In this subsection a behavioral model of an AND gate is used to show one way a model can accept different propagation timing values for the output signal from outside the model. This technique can be used to pass any parameters such as load capacitance and temperature

The `and2_gate` model is shown in Figure 6-5. The generic clause (line 22) allows you to parameterize a rise and fall time of the `out1` signal with values from outside the model in either a compiled VHDL test bench file containing a generic map, shown in Figure 6-4, or by some other implementation-dependent method.

```
 1   LIBRARY my_lib; --Define Logical library name.
 2   USE my_lib.my_qsim_logic.ALL;
 3   ENTITY test_and2_gate IS
 4   END test_and2_gate;
 5
 6
 7   ARCHITECTURE test_bench OF test_and2_gate IS
 8     COMPONENT and2
 9       GENERIC (Rs, Fl : time );
10       PORT(a,b: IN my_qsim_12state; c: OUT my_qsim_12state);
11     END COMPONENT;
12
13     FOR a1 :and2 USE ENTITY and2_gate(behav)
14       GENERIC MAP (Rs, Fl)
15       PORT MAP (a, b, c);
16     SIGNAL x, y, z : my_qsim_12state;
17   BEGIN
18     a1: and2
19      GENERIC MAP (7 ns, 10 ns);
20      PORT MAP (x ,y, z);
21   . . .  --The code here can exercise the AND gate.
22   END test_bench;
```

Figure 6-4. Test Bed Code for AND Gate Model

After the `and2_gate` model is compiled, the `test_and2_gate` model can be compiled. At that time the values from the VHDL test bench file are passed into the `and2_gate` VHDL model as shown in Figure 6-6. In this example, the `Rs` value is set to 7 ns and the `Fl` value is set to 10 ns (line 19 of the `test_bed` architecture in Figure 6-4). Where the `and2_gate` model in Figure 6-5 uses `out1_rs` (lines 32, 36, and 37), the value 7 ns is inserted. Similarly, where `out1_fl` is used in the `and2_gate` model (lines 35, 36, and 39 in Figure 6-5), the Fl value of 10 ns is inserted.

As shown in line 22 of Figure 6-5, you can set default values for the generic parameters. The default takes affect if there is no value associated with the signal from outside the `and2_gate` model. Default binding also occurs by other ways when certain conditions are met.

```
PACKAGE my_qsim_logic IS --Package defines qsim_12state
                           --and function qsim_state_from(),
                           --which returns qsim_state value.
TYPE my_qsim_state IS (’X’,’0’,’1’,’Z’);--X: Unknown logic level
 SUBTYPE my_qsim_value IS qsim_state RANGE  ’X’ TO ’1’ ;
                                  --Leading S: denotes qsim state
TYPE my_qsim_12state IS(SXR,SXZ,SXS,--Z: Hi imped. sig. strngth
                     SXI,S0R,S0Z, --R: Resistive sig. strngth
                     S0S,S0I,S1R, --Trailing S: Strong sig. strth
                     S1Z,S1S,S1I);--I: Indeterminate sig. strth
                                              -- CONVERSION TABLE
FUNCTION my_qsim_state_from(val: my_qsim_12state)-- State  Result
                        RETURN my_qsim_state;--S0S, S0R,     ’0’
                                             --S1S, S1R,     ’1’
                                   --S0I, S1I, SXS, SXR, SXI ’X’
                                             --SXZ, S0Z, S1Z ’Z’
END my_qsim_logic;  --PACKAGE BODY NOT SHOWN HERE.


USE my_qsim_logic.ALL;
ENTITY and2_gate IS
  GENERIC (Out1_rs, Out1_fl: time := 0 ns);
  PORT (in0, in1 : IN  my_qsim_12state;
            out1 : OUT my_qsim_12state );
END and2_gate;

ARCHITECTURE behave OF and2_gate IS
BEGIN
 and_inputs : PROCESS (in0, in1)
 BEGIN
  IF((my_qsim_state_from(in0) AND my_qsim_state_from(in1))= ’1’)
     THEN  out1 <= S1S AFTER Out1_rs;
  ELSIF ((my_qsim_state_from(in0) AND
                             my_qsim_state_from(in1)) =’0’)
     THEN out1 <= S0S AFTER Out1_fl;
  ELSIF (Out1_rs >= Out1_fl) THEN
    out1 <= SXS AFTER Out1_rs;
  ELSE
    out1 <= SXS AFTER Out1_fl;
  END IF;
 END PROCESS and_inputs;
END behave;
```

Figure 6-5. AND Gate Model Using Generics to Receive Timing Parameters

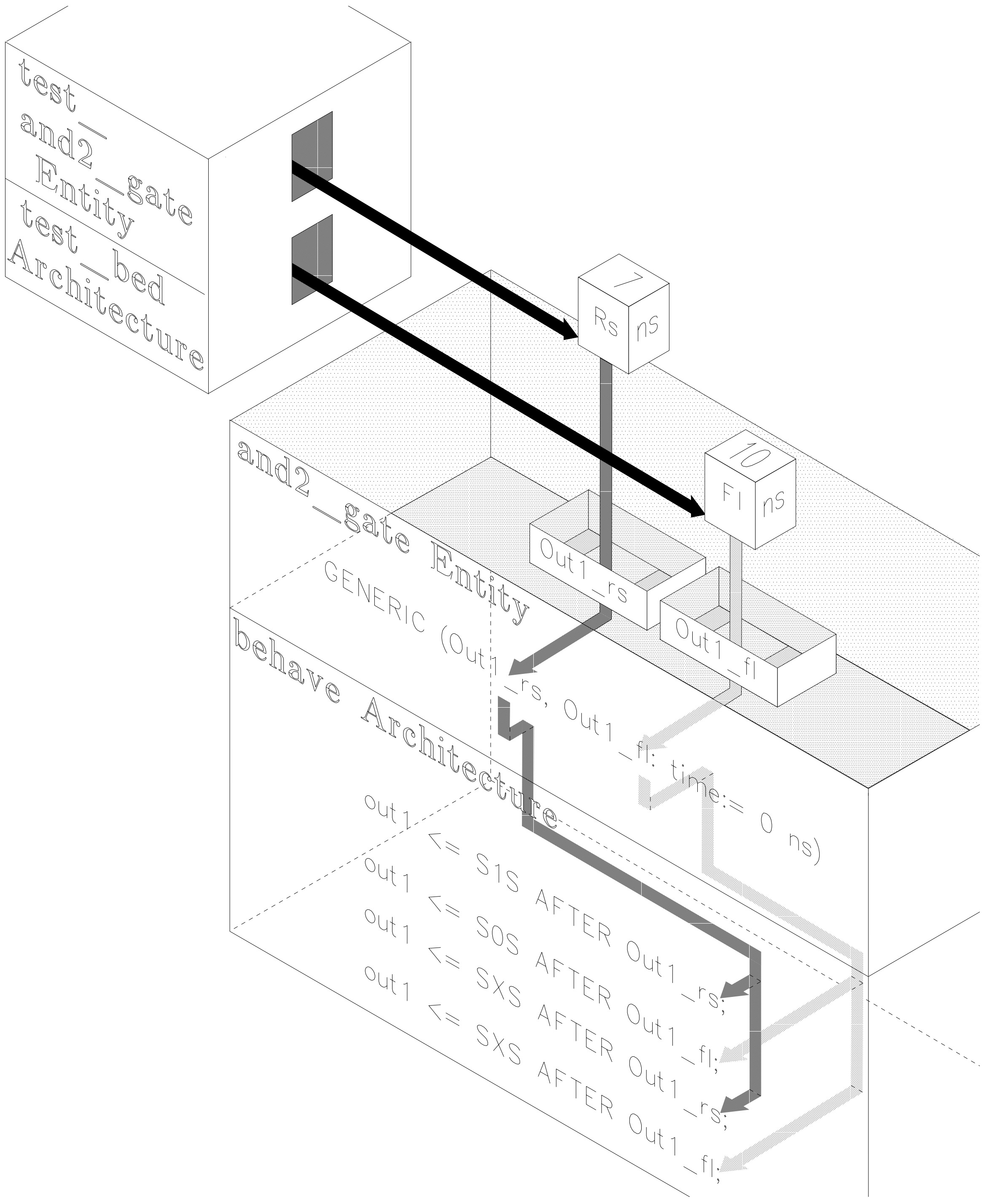

test_
and2__gate
Entity
test__bed
Architecture
7
Rs
ns
10
Fl
ns
Out1_rs
Out1_fl
and2__gate Entity
GENERIC (Out1_rs, Out1_fl: time:= 0 ns)
behave Architecture
out1 <= S1S AFTER Out1_rs;
out1 <= S0S AFTER Out1_fl;
out1 <= SXS AFTER Out1_rs;
out1 <= SXS AFTER Out1_fl;

Figure 6-6. Using Generics to Pass Customized Parameters to a Model

Parameterizing Rise/Fall Delays with Generics

The example in Figure 6-7 shows the entity declaration (interface) for a one-bit latch description that includes parameterized rise and fall delays. The rise and fall delay values are set in a related VHDL test bench file. The values from the compiled test bench model are passed into the latch model through generics. The latch code in Figure 6-7 reserves space for error checking code that is added and described in the following subsection.

Unlike the previous `and2_gate` design entity in Figure 6-5, the rise/fall delay values for the `latch` design entity are treated in the VHDL model as strings. Lines 17 through 20 of Figure 6-7 define the generic rise/fall constants that are passed into the model. Each of the rise/fall constants is declared as a string and given a default value of `"0, 0, 0"`. If the `latch` model is instantiated in another model without a generic map to set the corresponding propagation delay values, the default values are used. This string assumes the first number is the minimum delay, the second number is the typical delay, and the third number is the maximum delay (in nanoseconds). Line 21 receives the timing mode information.

A function called "my_qsim_get_time" is declared in package "my_qsim_extended" (not shown), which chooses the appropriate delay value from each generic constant string based on which timing parameter is chosen (`min`, `max`, or `typ`). To choose the correct string, a type is also defined in the "my_qsim_extended" package (`timing_type`) and has a value set of `min` (minimum), `typ` (typical), or `max` (maximum). You assign one of the following values: `min`, `typ`, or `max` to `Timing_mode` in a generic map in a test bench file or other implementation-dependent method. Line 2 in Figure 6-7 calls the `my_qsim_extended` package to make the `my_qsim_get_time` function and the `timing_mode` type visible to the `latch` design entity. Lines 28 through 32 in Figure 6-7 also describe the `my_qsim_get_time` function.

Figure 6-8 illustrates how a string is passed to the `latch` design entity for each rise and fall value on the enable and data signals. Once the strings are passed to the model, the appropriate value (`min`, `typ`, or `max`) from each string must be selected and converted to type time.

```
 1   USE my_lib.my_qsim_logic.ALL; --This package defines qsim_state.
 2   USE my_lib.my_qsim_extended.ALL;--This package defines
 3                                   --my_qsim_get_time function and
 4                                   --type timing_type
 5   ENTITY latch IS
 6                          --This line reserved (error check code)
 7                          --This line reserved (error check code)
 8                          --This line reserved (error check code)
 9       --Data_rise   = Prop. Delay low-to-high from data to q_out
10       --Data_fall   = Prop. Delay high-to-low from data to q_out
11       --Enable_rise = Prop. Delay low-to-high from enable to q_out
12       --Enable_fall = Prop. Delay high-to-low from enable to q_out
13       --Enable_fall = Prop. Delay high-to-low from enable to q_out
14     GENERIC (            --This line reserved (error check code)
15                          --This line reserved (error check code)
16                          --This line reserved (error check code)
17              CONSTANT Data_rise    : string := "0, 0, 0" ;
18              CONSTANT Data_fall    : string := "0, 0, 0" ;
19              CONSTANT Enable_rise  : string := "0, 0, 0" ;
20              CONSTANT Enable_fall  : string := "0, 0, 0"
21              CONSTANT Timing_mode  : timing_type := typ );
22         --Each string is ordered as min, typ, and max time values.
23         --Each string assumes values in nanoseconds.
24
25     PORT (enable, data : IN  my_qsim_state ;
26           q_out        : OUT my_qsim_state );
27
28     -- my_qsim_get_time function from my_qsim_extended package
29     -- accepts two parameters:
30     --  1 A string of characters that is converted to a time value
31     --  2 Constant "Timing_mode", which is declared in test_bed
32     --    model and set to a value of either min, typ, or max
33
34     CONSTANT Data_tplh  : time := my_qsim_get_time( Data_rise,
35                                                    Timing_mode);
36     CONSTANT Data_tphl  : time := my_qsim_get_time( Data_fall,
37                                                    Timing_mode);
38     CONSTANT Enable_tplh: time := my_qsim_get_time( Enable_rise,
39                                                    Timing_mode);
40     CONSTANT Enable_tphl: time := my_qsim_get_time( Enable_fall,
41                                                    Timing_mode);
42   BEGIN
 43 - 58 . . .             --These lines reserved (error check code)
 59  END latch;
```

Figure 6-7. Entity for One-Bit Latch Using Parameterized Rise/Fall Delays

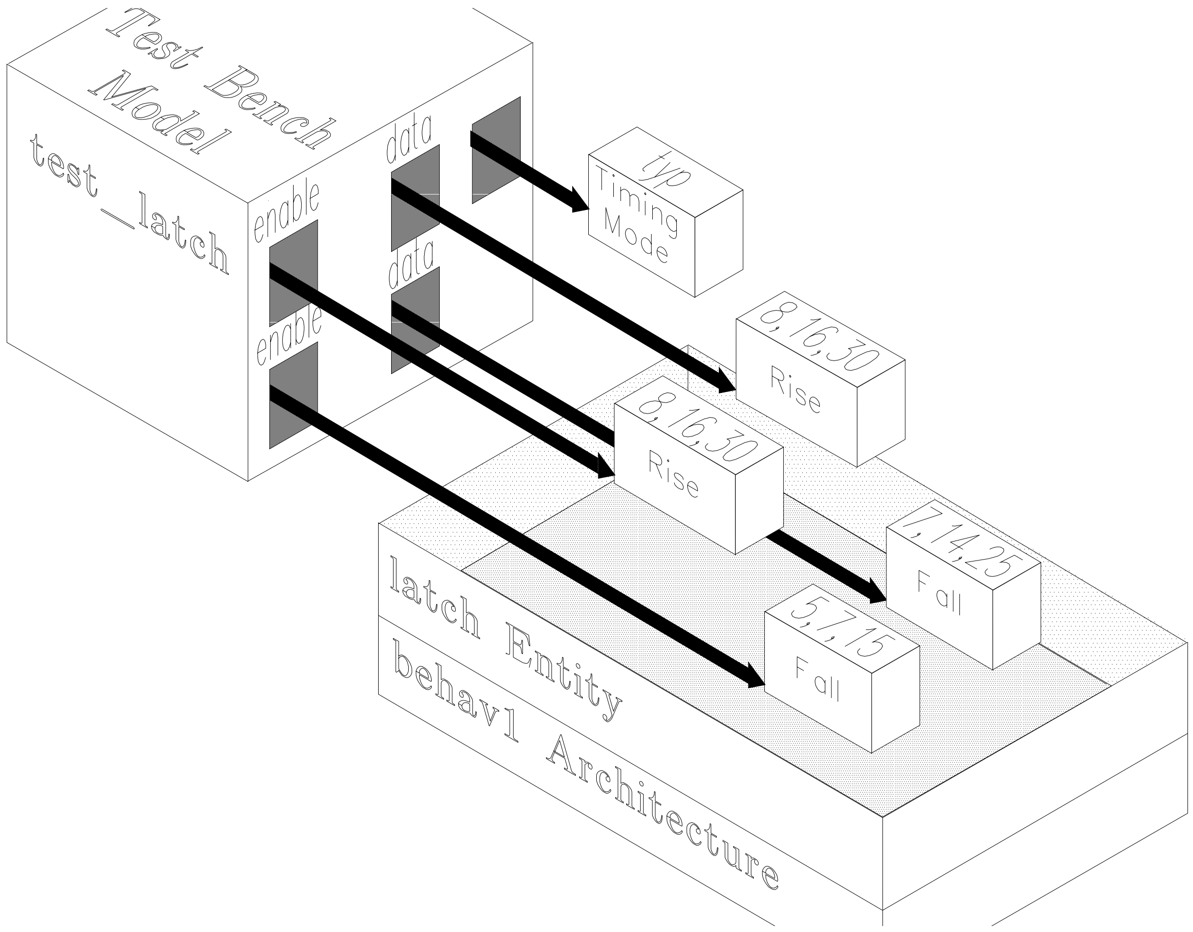


Figure 6-8. Rise/Fall Values Passed to One-Bit Latch Model

Four constants in Figure 6-7, declared in lines 34 through 41, correspond with the four generic constants in lines 17 through 20. The constants in lines 34 through 41 hold the actual propagation delay values used in the architecture of the model. Each of these are declared as type time and set to a specific value determined by the function `my_qsim_get_time`. The function and its parameters are described in the code comments in lines 28 through 32.

In this example, generic constant `Data_rise` (line 17 of Figure 6-7) receives string "8,16,30" as shown in Figure 6-8. This value comes from the fourth parameter in the generic map of line 25 in the test bench model of Figure 6-9. The eighth parameter in the generic map (Figure 6-9, line 26) sets the `Timing_mode` constant (Figure 6-7, line 21) to `typ`. The function call in line 34 of Figure 6-7 passes the string "8,16,30" and constant `Timing_mode` (which is set to `typ`) to function `my_qsim_get_time`. In this case, the function returns the time value 16 ns and assigns it to constant `Data_tplh` (Figure 6-7, line 34). The constant timing values are used throughout the `behav1` architecture body shown in Figure 6-10. The other rise/fall parameters are also set in this way.

```
LIBRARY my_lib; --Define Logical library name.
USE my_lib.my_qsim_logic.ALL;
USE my_lib.my_qsim_extended.ALL;  --Declares timing_type as
                   --TYPE timing_type IS (min, typ, max);
ENTITY test_latch IS
END test_latch

ARCHITECTURE test_bed OF test_latch IS
  COMPONENT latch1
    GENERIC (En_width, Da_setup, Da_hold       : time;
             Da_rise, Da_fall, En_rise, En_fall: string;
             Time_mode: timing_type );
    PORT (da, en: IN my_qsim_state; qo: OUT my_qsim_state);
  END COMPONENT;

  FOR L1 :latch1 USE ENTITY latch(behav1)
    GENERIC MAP (En_width, Da_setup, Da_hold,
                 Da_rise, Da_fall,
                 En_rise, En_fall, Time_mode);
    PORT MAP (da, en, qo);
  SIGNAL data, enable, q_out : my_qsim_state;
BEGIN
  L1: latch1
   GENERIC MAP( 20 ns, 20 ns, 5 ns,  --This line supports
               "8,16,30", "7,14,25", --timing chks
               "8,16,30", "5,7,15", typ);
   PORT MAP (data, enable, q_out);
. . .  --The code here can exercise the Latch.
END test_bed;
```

Figure 6-9. Test Bed Code for Latch Model

```
ARCHITECTURE behav1 OF latch IS

BEGIN
 PROCESS (enable, data)
 BEGIN
  IF enable’event THEN -- -------------------------------------
    IF enable = ’1’ THEN      --Behavior when enable is ’1’
      IF data = ’1’ THEN
        q_out <= data AFTER Enable_tplh;
      ELSIF data = ’0’ THEN
        q_out <= data AFTER Enable_tphl;
      ELSIF Enable_tplh >= Enable_tphl THEN
        q_out <= ’X’ AFTER Enable_tplh;
      ELSE
        q_out <= ’X’ AFTER Enable_tphl;
      END IF;                  --End Behavior when enable is ’1’

    ELSIF enable /= ’0’ THEN --Behavior when enable is ’X’ or ’Z’
      IF enable_tplh >= Enable_tphl THEN
        q_out <= ’X’ AFTER Enable_tplh;
      ELSE
        q_out <= ’X’ AFTER Enable_tphl;
      END IF;            --End Behavior when enable is ’X’ or ’Z’
    END IF;
  ELSE  -- No event on enable ---------------------------------
    IF enable = ’1’ THEN    --Model behavior when enable is ’1’
      IF data = ’1’ THEN
        q_out <= data AFTER Data_tplh;
      ELSIF data = ’0’ THEN
        q_out <= data AFTER Data_tphl;
      ELSIF Data_tplh >= Data_tphl THEN
        q_out <= ’X’ AFTER Data_tplh;
      ELSE
        q_out <= ’X’ AFTER Data_tphl;
      END IF;
    END IF;                 --End Behavior when enable is ’1’
  END IF;
END behav1;
```

Figure 6-10. Architecture Body for One-Bit Latch with Parameterized Delays

Increasing Model Accuracy with Error Checking

The previous example of the one-bit latch in Figure 6-7 is used here to show one way to add hold and setup violation checks to your simulation models. As before, generics are used to pass data into the VHDL model.

In this example the latch test bench model associated with the latch model assumes the values shown in Table 6-1 (set in lines 24 through 26 in Figure 6-9). The values added to the latch for the setup and hold checks in this example are Setup, Hold, and Width.

Table 6-1. Constant Values Set in Latch Test Bed Model

Constant	Type	Value
Enable_rise	string	8,16,30
Enable_fall	string	5,7,15
Enable_width	time	20 ns
Data_rise	string	8,16,30
Data_fall	string	7,14,25
Data_setup	time	20 ns
Data_hold	time	5 ns

Figure 6-11 shows the lines that are added to the code in Figure 6-7 to handle a pulse-width check for the enable signal, a check of the data-to-enable setup time, and a check of the data-from-enable hold time.

Lines 14 through 16 in Figure 6-11 show the generic constant declarations which receive the information from outside the model during simulation. Each of these constants are set to a default time value of 0 ns if there is no associated data from outside the model. When the latch model is instantiated in either a test bench model or another design entity and contains the values listed in Table 6-1, the associated VHDL model assigns the proper values to the constants.

```
    --Enable_width  = Width of enable pulse
    --Data_setup    = Setup Time
    --Data_hold     = Hold Time
. . .
  GENERIC ( CONSTANT Enable_width : time   := 0 ns ;
            CONSTANT Data_setup   : time   := 0 ns ;
            CONSTANT Data_hold    : time   := 0 ns ;
. . .

-----------------------------------------------------------------
  chk_width: ASSERT (enable = ’1’) OR (enable’delayed = ’0’) OR
               (enable’delayed’last_event >= Enable_width)
             REPORT "Enable signal has insufficient pulse width."
              SEVERITY error;
-----------------------------------------------------------------
  chk_setup: ASSERT (enable’stable) OR (enable /= ’0’) OR
               (data’last_event >= Data_setup)
             REPORT "Data-to-enable setup time violation."
              SEVERITY error;
-----------------------------------------------------------------
  chk_hold:  ASSERT (data’stable) OR (enable /= ’0’)
               (enable’last_event >= Data_hold)
             REPORT "Data-from-enable hold time violation."
              SEVERITY error;
-----------------------------------------------------------------
END latch;
```

Figure 6-11. Error-Checking Code Added to Entity for One-Bit Latch

The error checks are performed with three concurrent assertion statements within the entity statement part (lines 43 through 58). The purpose of each assertion statement is to report a timing violation during simulation.

The first error check (lines 43 through 47) evaluates the width of the enable pulse. With a pulse width (`enable_width`) value of 20 ns, the assertion statement generates an error if the enable pulse is not at a '1' value for at least 20 ns. Two predefined attributes are used to determine if the enable pulse width is within specification: 'delayed and 'last_event.

In Figure 6-11, the first condition in line 44 (`enable = '1'`) is ORed with two other conditions. All three conditions must be false before the assertion statement issues the report in line 46. If the condition `enable = '1'` is true, the report is not issued.

To understand the other two conditions in lines 44 and 45, you must understand how 'delayed* and 'last_event operate in a simulator environment. Figure 6-12 shows two positive enable pulses: one that fails the pulse-width check and issues the report (16 ns wide), and one that passes the pulse-width check and does not issue the report (20 ns wide). This example assumes that the constant `Enable_width` equals 20 ns. The figure shows two signals: `enable` and `enable'delayed`. Each simulator event (from E1 through E8) is indicated on the figure at the appropriate signal transition. Each event shown causes the assertion statement conditions to evaluate.

When the predefined attribute 'delayed is appended to the `enable` signal with no time delay value specified, a new signal is created, as shown in Figure 6-12 by `enable'delayed`. This new `enable'delayed` signal is delayed from the original `enable` by 0 ns. Even though 0 ns is the delay time (called a delta delay), the simulator still provides one iteration of delay to the `enable'delayed` signal. This is an important concept to understand when using a 0 ns delay.

The condition (`enable'delayed'last_event >= enable_width`) in line 45 does the actual pulse-width check. The predefined attribute 'last_event returns the elapsed time since the last event occurred on the associated signal. At event E3 (30.0 ns, iteration 1), `enable'last_event` returns a value of 0 ns.

*Also refer to the *Mentor Graphics VHDL Reference Manual*, the "Signal Attributes" subsection.

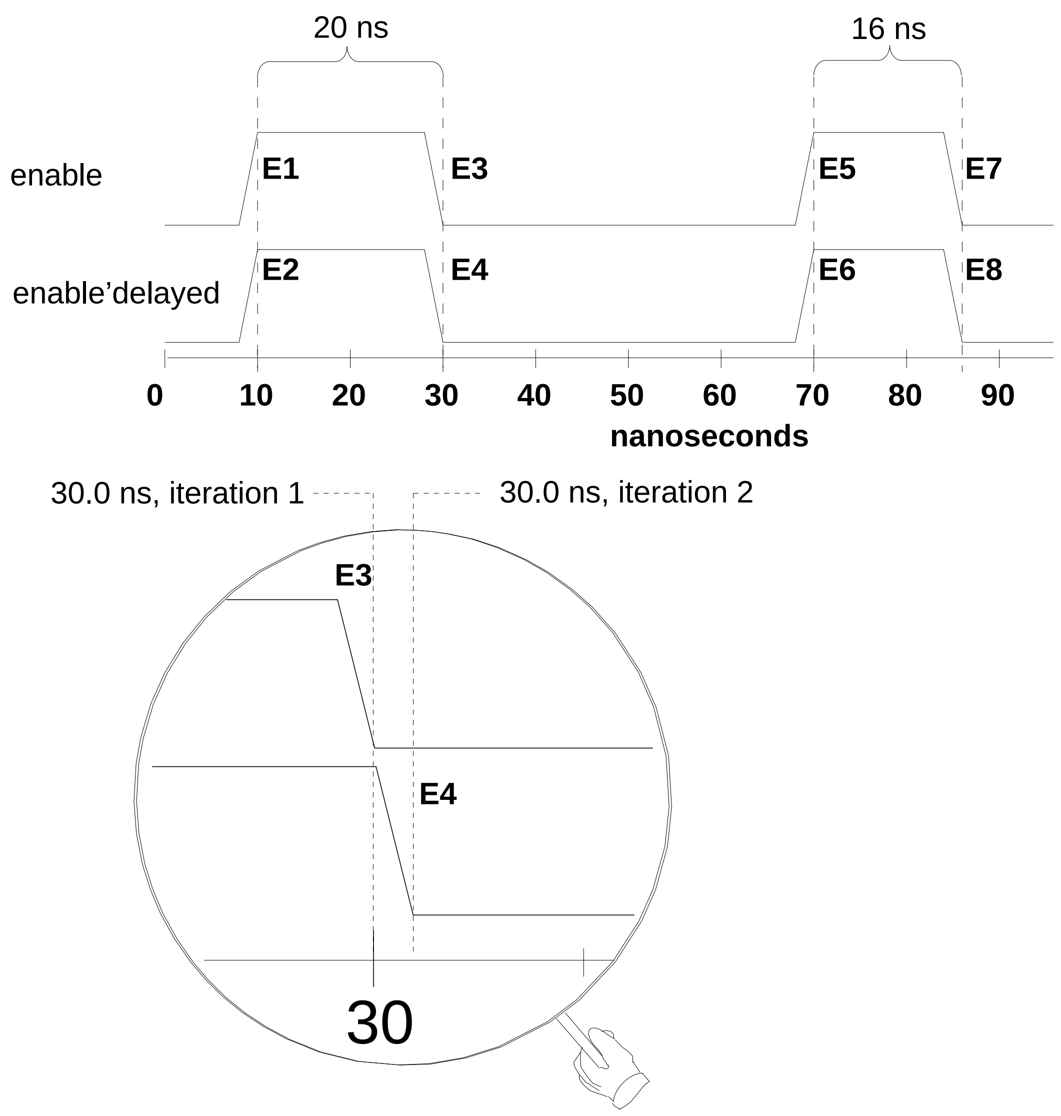


Figure 6-12. Comparing a Signal with Its Delayed Counterpart

The elapsed time since the last event at timestep "30.0 ns" (E3) for the enable signal is 0 ns. This is why the 'last_event attribute is applied to the delayed version of the `enable` signal in line 45. If the 'last_event attribute is applied to the `enable` signal instead of the delayed version of the `enable` signal, the elapsed time between events E3 and E1 *is not* returned at event E3.

At event E3, (when the `enable` signal changes to a '0') the`enable'delayed` signal has not yet changed state (no event). Therefore, when the 'last_event attribute is applied to the delayed version of the `enable` signal at event E3, a result of 20 ns is returned. (The last timestep where `enable’delayed` changed state was 10.0 ns, iteration 2.) At event E3, the condition in line 45 is true and the report is not issued. If the result of the condition in line 45 is false* (along with the other two conditions in line 44), the assertion statement issues the report in line 46.

The second condition (`enable’delayed = ’0’`) in line 44 is required to prevent the `enable’delayed` signal from triggering the assertion statement report. When `enable’delayed` is ’0’, such as at event E4, the assertion statement will not issue the report because all three conditions have to be false.

In Figure 6-11, lines 49 through 52 (repeated below) check the setup time of the latch.

```
49  chk_setup: ASSERT (enable’stable) OR (enable /= ’0’) OR
50                (data’last_event >= Data_setup)
51             REPORT "Data-to-enable setup time violation."
52              SEVERITY error;
```

The `chk_setup` assertion statement also has three conditions that must evaluate to a Boolean FALSE value before the report in line 51 is generated. Figure 6-13 shows the waveforms and parameters that relate to the hold and setup checks. The setup check relates to the events labeled E1 and E3 in the figure.

This example assumes that constant `Data_setup` is set to a value of 20 ns. The first condition in line 49 (`enable’stable`) produces a FALSE value whenever there is an event on the enable signal in the current simulator iteration. The predefined attribute ’stable with no time value specified produces a Boolean value of TRUE if there has been no event on a signal in the current simulator iteration. This condition in line 49 is necessary to trigger the assertion statement checking whenever the enable signal changes state.

The second condition in line 49 (`enable /= ’0’`) causes the assertion statement to look for a setup error when `enable` is ’0’ (not at ’1’, ’X’, or ’Z’). The third condition in line 50 performs the actual setup violation check. The expression `data’last_event` returns a time value that designates the elapsed simulator

*The result of the condition in line 45 (Figure 6-11) is false at the event labeled E7 in Figure 6-12.

time since `data` last changed. If the returned value is less than or equal to 20 ns in this example, a setup violation has occurred. The setup time in Figure 6-13 (from event E1 to E3) is within the 20 ns specification and will not generate an error report.

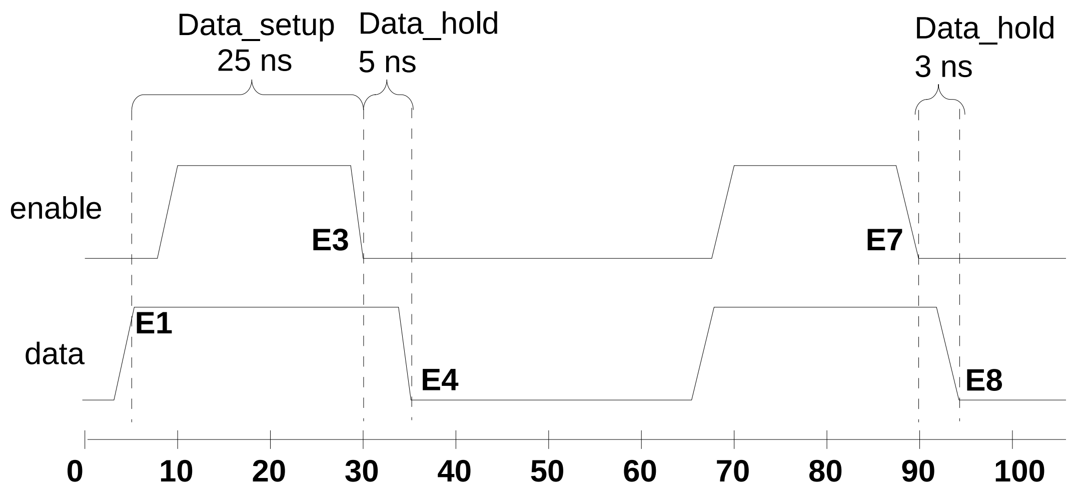


Figure 6-13. Setup/Hold Timing for Latch

The `chk_hold` assertion statement in lines 54 through 57 is similar to the `chk_setup` assertion statement in lines 49 through 52.

```
54  chk_hold:  ASSERT (data’stable) OR (enable /= ’0’) OR
55                (enable’last_event >= Data_hold)
56             REPORT "Data-from-enable hold time violation."
57              SEVERITY error;
```

The `chk_hold` assertion statement has three conditions that all must evaluate to FALSE before the report is issued. The first condition (`data’stable`) causes the assertion statement to "look" at the other two conditions only when there is an event on `data`.

The second condition (`enable /= ’0’`) causes the assertion statement to perform the hold check only when `enable` is ’0’ (not ’1, ’X’, or ’Z’’). The third condition (`enable’last_event >= Data_hold`) performs the hold check. If the first two conditions are false, then this statement determines whether the report will be sent. Assuming that constant `Data_hold` has a value of 5 ns, the result of the third condition applied to event E4 in Figure 6-13 would be true and no error would be reported. However, the result for the third condition at event E8 would be false and an error would be reported.

Modeling for Increased Simulation Performance

Because VHDL contains many constructs to cover a wide range of modeling problems, there will be times that you are faced with multiple ways to model a given task. Given two or more ways of modeling a given task, you may decide that maximum simulation performance is your goal. This subsection describes some considerations regarding performance for some of the tasks that can be modeled in more than one way.

When to Use Variables Within a Loop Instead of Signals

The following paragraphs describe when a variable assignment within a loop is more efficient during simulation than a signal assignment.

For every signal assignment statement that appears in a simulation model, a queue of projected waveforms for each signal must be maintained by the simulator. Signal assignments do not cause the simulator to assign a new value to the signal immediately (within the current iteration). Signal assignments encountered during simulation cause a simulator to keep a list of projected values for future times (either a future simulator iteration or future simulator timesteps). For more information on this principle, refer to "How Values Get Assigned to Signals and Variables" on page 4-36.

The architecture `a1` on the left of Figure 6-14 shows one way to use a signal (`sig1` in line 14) to find and store the maximum integer value from a 64-element array. The signal assignment statement is executed (as determined by the if condition in line 13), and an event *is scheduled after 0 ns* every time a value is found in one of the array elements that is greater than the value stored in the `max` variable. Putting the signal assignment statement inside the loop (lines 12 to 17) causes the simulator to operate on the `sig1` queue up to 64 times. Every time the simulator updates the `siq1` queue, more time is required for the simulation run.

```
 1   ARCHITECTURE a1 OF en IS
 2     TYPE var_arr IS ARRAY
 3       (1 TO 64) OF integer;
 4     SIGNAL sig1  : integer;
 5
 6   BEGIN      --Inefficient
 7    PROCESS (cntrl)
 8     VARIABLE max : integer;
 9     VARIABLE var : var_arr;
10
11    BEGIN
12     FOR elmnt IN 1 TO 64 LOOP
13      IF var(elmnt) >max THEN
14        sig1 <= var(elmnt);
15        max  := var(elmnt);
16      END IF;
17     END LOOP;
18    END PROCESS;
19   END a1;
```

```
 1   ARCHITECTURE a2 OF en IS
 2     TYPE var_arr IS ARRAY
 3       (1 TO 64) OF integer;
 4     SIGNAL sig1  : integer;
 5
 6   BEGIN     --More Efficient
 7    PROCESS (cntrl)
 8     VARIABLE max : integer;
 9     VARIABLE var : var_arr;
10
11    BEGIN
12     FOR elmnt IN 1 TO 64 LOOP
13      IF var(elmnt) >max THEN
14        max  := var(elmnt);
15      END IF;
16     END LOOP;
17     sig1 <= max;
18    END PROCESS;
19   END a2;
```

Figure 6-14. Signal Assignment Within a Loop

In Figure 6-14, architecture `a2` on the right provides the same basic function as architecture `a1`. The signal assignment has been moved outside the loop (line 17). The variable `max` keeps track of the highest integer value encountered (line 14) and is assigned to `sig1` only once during the process (lines 7 through 18). This causes the simulator to *schedule an event* for `sig1` just once instead of up to 64 times, a method which decreases the time it takes to simulate this version of the model.

Using Resolution Functions Only When Needed

Your VHDL models will most likely contain more than one signal assignment statement (each with its own driver) that assigns different values to the same signal at the same time. When this happens, your model must provide a resolution function that specifies how to resolve the assignment. For each signal that is declared as a resolved signal, the associated resolution function is called whenever a signal assignment is executed.

For example, if you have two signals declared as type bit, you might declare them on the same line as shown below on the left. If signal `s1` requires a resolution function, you must add the resolution function name (`wired_or` in this example) to the signal declaration. If signal `s2` does not require a resolution function, the declaration should be separated from the `s1` signal declaration as shown on the right.

```
SIGNAL s1,s2:wired_or bit;       SIGNAL s1: wired_or bit;
                                 SIGNAL s2:          bit;
```

For a more efficient simulation model, use a resolution function only when necessary. For more information on resolution functions, refer to the *Mentor Graphics VHDL Reference Manual* subsection titled "Multiple Drivers and Resolution Functions."

Using Attribute ’event Instead of ’stable When Possible

The basic rule to follow when coding is to check if there is a construct or attribute that specifically performs the required modeling task, and use that construct instead of one that is more general. For example, if you require a test condition to check if a signal has an event scheduled during the current simulator timestep, you can use either the attribute ’event or the attribute ’stable. As will be shown, ’event is preferred over ’stable for this particular case if maximum simulation performance is desired.

Consider the following code examples:

```
IF (sig’stable = false) THEN
  --something happens
END IF;
```

```
IF (sig’event = true) THEN
  --something happens
END IF;
```

The if condition in both examples (on the left and the right) is true whenever `sig` is scheduled to change state in the current simulator timestep. The major difference between these examples is the amount of work the simulator must perform when evaluating the attributes. Attribute ’stable takes more work to evaluate than ’event.

Attribute ’event returns a Boolean value TRUE whenever the associated signal is scheduled to change value in the current simulator timestep. Attribute ’stable(t) returns a Boolean value TRUE if the associated signal has been stable for the specified time (t). If no time is specified, such as in the previous example on the left, the attribute ’stable returns the value TRUE as long as the associated signal remains stable.

The attribute ’stable is more general and has more information to process than attribute ’event. For this reason, you should use the more specific attribute ’event for conditions shown in the previous example to make your model more efficient during simulation.

For a comparison of these attributes with other signal attribute, refer to the *Mentor Graphics VHDL Reference Manual* subsection titled "Signal Attribute Example."

Creating Lookup Tables for Logic Operations

A lookup table is an efficient way to define and reference Boolean and logic operations. The table is created as a constant, which is defined as an array with two or more dimensions. In the following example, a function is declared, which uses a lookup table to implement an AND logic function for signals defined as type my_lsim_LOGIC.

In Figure 6-15, the function `AND` is defined to accept two my_lsim_LOGIC inputs, `a` and `b`, and return a my_lsim_LOGIC value (line 3). A type is declared (`a2_lookup` in lines 11 through 14) to form the array template for declaring the constant. The array is two dimensional; each dimension consists of four elements. The indices range from ’0’ to ’Z’ as defined in the my_lsim_LOGIC type declaration.

The constant is declared in lines 15 through 20 as type `a2_lookup`. Each array element holds a particular my_lsim_LOGIC value that corresponds to the AND logic function on the two corresponding index values. The `a` and `b` inputs provide the x and y index for the array. Line 23 returns the appropriate array element value to the code that called the function. An example of the calling code could be as follows:

```
sig3 <= "AND"(sig1,sig2); --sig3= 0 if sig1= 1 AND sig2= 0
```

Because the function `AND` is defined as an overloaded operator, the following line performs the same function as the previous line:

```
sig3 <= sig1 AND sig2; --sig3= 0 if sig1= 1 AND sig2= 0
```

```
PACKAGE logic_example IS
  TYPE my_lsim_LOGIC IS (’0’, ’1’, ’X’, ’Z’);
 FUNCTION "AND" (a, b : IN my_lsim_LOGIC)
    RETURN my_lsim_LOGIC;
END logic_example;


PACKAGE BODY logic_example IS
 FUNCTION "AND" (a, b : IN my_lsim_LOGIC)
    RETURN my_lsim_LOGIC IS
  TYPE a2_lookup IS ARRAY
                     (my_lsim_LOGIC’(’0’) TO my_lsim_LOGIC’(’Z’),
                      my_lsim_LOGIC’(’0’) TO my_lsim_LOGIC’(’Z’))
                        OF my_lsim_LOGIC;
  CONSTANT Output : a2_lookup :=
   --  0    1    X    Z
    ((’0’, ’0’, ’0’, ’0’),  -- 0
     (’0’, ’1’, ’X’, ’X’),  -- 1
     (’0’, ’X’, ’X’, ’X’),  -- X
     (’0’, ’X’, ’X’, ’X’)); -- Z

  BEGIN
   RETURN Output(a, b);
  END "AND";
END logic_example;
```

Figure 6-15. Logic Example Package

Process Statements--Avoiding Infinite Loops

When using process statements, you should be aware that you could create an undesired infinite loop condition during simulation.

Once a process is executed during simulation, it is always active (it never stops). Each process may contain one or more sequential statements that continue to execute without advancing simulation time until a condition is encountered that suspends the sequence. If you do not provide a condition in your code that will suspend the execution, the sequential statements will loop forever.

Compilers can help check for some infinite loop conditions in processes. An infinite loop occurs during simulation if a process statement *does not* contain any of the following:

- A sensitivity list (which has an implied wait statement at the end of the process)
- A wait statement
- A procedure call statement that calls a procedure containing a wait statement

A compiler can generate an error or warning if no sensitivity list, wait statement, or procedure call is present within a process. A compiler can generate a warning (not an error) if no sensitivity list or wait statement is present, but the process does contain a procedure call. You must have a wait statement in the procedure to avoid an infinite loop. A VHDL compiler cannot determine whether a wait statement should be in a procedure that is called from a process, so only a warning can be issued. A compiler warning helps make you aware that a possible infinite loop exists in your code.

In Figure 6-16, the code at the left shows an error condition that could be caught at compile time because two processes (`P1` and `P2`) satisfy the infinite loop condition (no sensitivity list, no wait statement, and no procedure call). The code at the right of Figure 6-16 is the same code but with additional text (lines 11 and 20) to correct the infinite loop condition.

```
ARCHITECTURE behav3 OF
     aoi IS
  SIGNAL O1,O2,O3:
    my_lsim_logic;

BEGIN
  O1 <= A AND B;
  O2 <= C AND D;

  P1:  -- ERROR CONDITION
  PROCESS -- Infinite Loop
  BEGIN
    O3 <= O1 OR O2;
  END PROCESS;

  P2:  -- ERROR CONDITION
  PROCESS -- Infinite Loop
  BEGIN
    E  <= NOT O3;
  END PROCESS;
END behav3;
```

```
ARCHITECTURE behav3 OF
    aoi IS
  SIGNAL O1,O2,O3:
    my_lsim_logic;

BEGIN
  O1 <= A AND B;
  O2 <= C AND D;

  P1: --Good code includes
  PROCESS (O1, O2) --sens.
  BEGIN            --list.
    O3 <= O1 OR O2;
  END PROCESS;

  P2: --Good code includes
  PROCESS--wait statement
  BEGIN
    E  <= NOT O3;
    WAIT ON O3;
  END PROCESS;
END behav3;
```

Figure 6-16. Correcting an Infinite Loop in a Process Statement

In Figure 6-17, the code at the left contains a process that uses a procedure call (`procedure1` in line 15) and does not use a sensitivity list or a wait statement. A compiler can generate a warning (not an error) when this code is compiled.

The package at the right of Figure 6-17 contains the procedure (`procedure1`) called from the process in the code at the left of the figure. In this example an infinite loop is avoided because the called procedure contains a wait statement that halts the execution of sequential statements within the process.

```
LIBRARY contains_package;
USE pkg_at_right.ALL;

ENTITY expl IS
   PORT (sig1 : IN bit);
END expl ;


ARCHITECTURE behav OF expl
  IS
BEGIN
 PROCESS
   VARIABLE var1 : bit;
 BEGIN
  --something happens here
  procedure1(var1, sig1);
 END PROCESS;
END behav;
```

```
PACKAGE pkg_at_rt IS
 --procedure1 Declaration-
  PROCEDURE procedure1 (
   VARIABLE frst: OUT bit;
   VARIABLE scnd: IN bit);
  ------------------------
END pkg_at_rt;

PACKAGE BODY pkg_at_rt IS
  --procedure1 --Body---
  PROCEDURE procedure1 (
   VARIABLE frst: OUT bit;
   VARIABLE scnd:IN bit)
    IS
  BEGIN
   frst := NOT scnd;
   WAIT FOR 10 ns;
  END procedure1;
  ------------------------
END pkg_at_rt;
```

Figure 6-17. Wait Statement in a Procedure Avoids an Infinite Loop

Using VHDL for Simulation Stimulus

Depending on the VHDL environment, there can be a number of ways to provide simulation stimulus for your VHDL models. One way is to use a VHDL test bench model. The example in Figure 6-18 provides stimulus for a two-input AND gate that uses the my_qsim_12state type.

The model in Figure 6-18 is completely contained in one design file to make it simple to move around your system or store in the library with the model to be tested. The model includes comments that provide brief instructions on how the model should be used to test a two-input AND gate.

The entity declaration does not define any interface to the test bench model. Internal signals x and y are declared (line 16) to map to the inputs of the AND gate, and internal signal z maps to the output of the AND gate.

Within the architecture body (lines 7 through 38), the two concurrent statements (lines 22 through 33 and lines 35 through 37) force the outputs of the stimulus model (the inputs of the AND gate to which the outputs are mapped). All you need to do from the simulator is to run the simulation for about 150 ns. You can then check the simulation results against the logic table provided in the stimulus code and against the AND gate model delay parameters. This example does not test all the input conditions possible using the 12-state type my_qsim_12state. Only the states with a "strong" strength are tested. The ’Z’ state is also ignored.

```
LIBRARY my_lib; --Define Logical library name.
USE my_lib.my_qsim_logic.ALL;
ENTITY test_and2_gate IS
END test_and2_gate;


ARCHITECTURE test_bed OF test_and2_gate IS
  COMPONENT and2
    GENERIC (Rs, Fl : time );
    PORT (a, b: IN my_qsim_12state; c: OUT
my_qsim_12state);
  END COMPONENT;

  FOR a1 :and2 USE ENTITY and2_gate(behav)
    GENERIC MAP (Rs, Fl)
    PORT MAP (a, b, c);
  SIGNAL x, y, z : my_qsim_12state;
BEGIN
  a1: and2
   GENERIC MAP (7 ns, 10 ns);
   PORT MAP (x ,y, z);

    x <= S0S AFTER  10 ns, --   Abbreviated Directions:
         S1S AFTER  20 ns, -- 1. Compile this model.
         SXS AFTER  30 ns, --
         S0S AFTER  40 ns, -- 2. Invoke simulator
         S1S AFTER  50 ns, --    on this model
         SXS AFTER  60 ns, -- 3. Run the simulator
         S0S AFTER  70 ns, --    for 150 ns.
         S1S AFTER  80 ns, -- 4. Check the results against
         SXS AFTER  90 ns, --    following logic table
         S0S AFTER 100 ns,  --          out1
         S1S AFTER 110 ns,  --    \  X   0   1   Z
                            -- out2\_______________
                            --  X  | X   0   X   X
    y <= S0S AFTER  40 ns,  --  0  | 0   0   0   0
         S1S AFTER  70 ns,  --  1  | X   0   1   X
         SXS AFTER 130 ns;  --  Z  | X   0   X   X
END test_bed;
```

Figure 6-18. VHDL Model Used as Stimulus for AND Gate

Glossary

abstract literal

An abstract literal is one of two types of a numeric literal. (The other numeric literal is a physical literal). An abstract literal is either an integer literal (such as 0, 16e2, and 1_024) or a real literal (such as 0.0, 24.0, and 66.33e-9). (*Also refer to* literal).

abstraction

Abstraction is a coding principle that groups details (in a module) describing the function of a design unit but does not describe how the design unit is implemented. This principle is closely related to modularity.

actual

An actual is a port, signal, variable, or expression that is associated with a corresponding formal.

actual port

Refer to port.

adding operators

The adding operators consist of the "+", "-", and "&" operators. The predefined adding operator "+" has the conventional "addition" definition. The operator "-" has the conventional "minus" or "subtraction" definition. The operator "&" performs a concatenation operation on the left and right operand.

aggregate

An aggregate is the combination of one or more values into a composite value of an array. The following example shows the four-bit wide array variable `mem_load` assigned four distinct bit values (line 5) to produce a composite value of 1010:

```
1   PROCESS (sig1)
2    TYPE mem4 IS ARRAY (0 to 3) OF bit;
3    VARIABLE mem_load : mem4;
4   BEGIN
5    mem_load := ('1', '0', '1', '0'); -- aggr. is in paren's
6   END PROCESS;
```

allocator

An allocator is an expression that, when evaluated, creates an anonymous object and yields an *access value* that designates that object. The access value can be thought of as the address of the object. The access value may be assigned to an *access-type* variable, which then becomes a *designator* of (hereafter called a pointer to) the unnamed object. Such pointers allow access to structures like FIFOs (first in, first out registers) and linked lists that contain unnamed elements for which storage is dynamically allocated. For additional information, refer to the *Mentor Graphics VHDL Reference Manual*, in the "Allocators" subsection.

anonymous types

An anonymous type is created implicitly and has no name to reference within code. Numeric types (such as integer and floating point types) have an implied base type that are anonymous. Because the base type is not explicitly named, you cannot refer directly to the anonymous base type. The base type of an array is also anonymous.

architecture body

An architecture body is a VHDL construct that describes the relationships between the design entity inputs and outputs. In the architecture body, the design entity behavior, data-flow, or structure is described.

array types

An array type is a form of a composite type. Objects that are declared as an array type contain a collection of elements that are of the same type. An array type definition is either constrained or unconstrained. (*Also refer to* constrained array definition and unconstrained array definition). For example, the following array definition creates a template (declared as type `test_array`) for objects which contain four elements that each have a value of type `integer`:

```
TYPE test_array IS ARRAY(0 TO 3) OF integer;--constr. array
```

ascending

A range, such as `0 TO 3`, is considered an ascending range.

ASCII

ASCII is an acronym for American Standard Code for Information Interchange. The predefined package called "standard" contains a definition of type `character`, which represents the ASCII character set.

assertion violation

This term describes when a condition in an assertion statement evaluates to false.

association list

An association list provides the mapping between formal or local generics, ports, or subprogram parameter names and local or actual names or expressions.

attribute

An attribute defines a particular characteristic of a named item. The kinds of attributes are function, range, signal, type, and value. These five attribute kinds operate on the following kinds of items: array, block, signal (scalar or composite), or type (scalar, composite, or file). A number of predefined attributes are provided with VHDL.

base type

Every type and subtype declaration has a base type. Consider the following type declarations:

```
TYPE volume IS (height, width, depth);
  SUBTYPE area IS volume RANGE height TO width;
```

For the `area` subtype declaration, the base type is `volume`. In the `volume` type declaration, the base type is itself (`volume`).

Backus-Naur

This term refers to a semi-algebraic notation for documenting the syntax of a programming language. Graphical syntax diagrams are used in Appendix A of the *Mentor Graphics VHDL Reference Manual* to convey syntax information in addition to the Backus-Naur format used throughout that manual. (*Also refer to* the "BNF Syntax Description Method" subsection in the *Mentor Graphics VHDL Reference Manual*).

behavioral description

The VHDL method that allows you to describe the function of a hardware design in terms of circuit and signal response to various stimulus. The hardware behavior is described algorithmically without showing how it is structurally implemented.

binding indication

Binding indication is a language construct used within a configuration specification to associate (bind) the component instance to an entity declaration.

block

A block is a smaller unit of an overall design. A design entity is a major unit at the top of a design hierarchy. A design entity is considered an external block. Within a design entity, functionality can be decomposed into smaller units (with the VHDL block statement). The block statement defines internal blocks. Blocks defined within blocks indicate the design hierarchy.

box

The symbol <> (box) is used in an index subtype definition to denote an undefined range. The phrase `RANGE <>` (range box) in the following type declaration indicates that the `arr2` index can range over any interval allowed in the index subtype `integer`:

```
TYPE ar2 IS ARRAY(integer RANGE <>) OF m_arr; --unconst. arr.
```

bus

The reserved word **bus** is used in a signal declaration so you can control the assignment of a value to a signal. Signals that are declared as a bus re-evaluate the output value when all drivers are disconnected. (*Also refer to* guarded signal and register).

character literal

A character literal is a single ASCII symbol enclosed in single quotes ('). These characters are case-sensitive. The character 'Z' does not equal the character 'z'. A character literal is one type used in an enumeration literal. (*Also refer to* enumeration literal).

comment

Comments are used within VHDL code to document areas or lines of code that may not be clear to the reader. Comments are phrases or sentences that start with a double dash (--) symbol. Any text appearing between the double dash and the end of a line is ignored by the compiler. Descriptive comments make the code easier to read.

compiler

A VHDL compiler is a program that checks the source code for proper syntax and semantics, displays any errors encountered, and (once you correct any errors) translates the source code into a simulator-compatible database (called the object code).

complete context

The term complete context is used in conjunction with overload resolution. A complete context is either a statement, specification, or declaration. When the compiler sets out to resolve an overloaded name, it looks for exactly one interpretation of each part of the innermost complete context. (*Also refer to* overloading and overload resolution).

component

A VHDL component is the basic unit of a structural description. Components allow you to declare a device and instantiate it within a design entity's architecture body without needing to specify the actual architecture of the device.

component binding
Component binding is the method used to interconnect components using three kinds of ports: formal, local, and actual. (*Also refer to* port).

composite type
A composite type specifies groups of values under a single identifier. One composite type is an array type, which allows you to group items that are naturally represented as a table or are logically bundled together. Another composite type is a record type.

concatenation
Concatenation is the process of combining two or more smaller elements into a larger element. For example, you can combine two string literals to form one larger string. Also, two smaller arrays can be concatenated to form a larger array. The ampersand (**&**) is used to indicate a concatenation.

concurrent statements
Concurrent statements define interconnected processes and blocks that together describe a design's overall behavior or structure. A concurrent statement executes asynchronously with respect to other concurrent statements.

configuration declaration
A configuration declaration provides a mechanism for deferring the binding of a component instance in a given block to a specific design entity, which describes how each component operates.

configuration specification
A configuration specification binds a component instance to a specific design entity, that describes how each component operates.

constant
A constant is one type of VHDL object. (Signals and variables are also objects.) The value of a constant is set in a declaration and cannot be changed in a statement. (*Also refer to* deferred constant).

constrained array definition

A constrained array is an array that has a defined range of the array indices such as `(1 TO 25)` in the following example:

```
TYPE int_array IS ARRAY(1 TO 25) OF integer;--constr. arr.
```

constraint

A constraint (index or range) defines a value subset of a given type.

construct

A language construct is one of the many building blocks of VHDL. Each construct is an item that is constructed from basic items such as reserved words or other language building blocks.

data-flow

Data-flow is the VHDL description method, which is similar to register-transfer languages. This method describes the function of a design by defining the flow of information from one input or register to another output or register.

declaration

A declaration is code you write to introduce items (such as types, objects, and entities) to a certain scope of the hardware model. You provide a name for each declaration, which can be referenced throughout the scope of the model that is visible to the declaration.

default expression

A default expression is an expression that supplies a default value in signal declarations, interface constant declarations, interface variable declarations, or interface signal declarations. The default value is used during initialization of the simulator, or when a signal is left unconnected or the interface object is left unassociated.

deferred constant

A deferred constant is specified if you do not use an expression after the ":=" delimiter in a constant declaration. A deferred constant allows you to declare a constant but not to specify the value immediately. Deferred constants can only appear in package declarations and must have a constant declaration in the package body. The following example shows a deferred constant declaration and the corresponding full constant declaration:

```
PACKAGE common_info IS
   CONSTANT Xtal_value : real;  -- deferred constant
END common_info;

PACKAGE BODY common_info IS
   CONSTANT Xtal_value : real := 1.556E6; --full constant
           . . .
```

delta delay

The term delta delay refers to a very small amount of time (greater than zero but less than one timestep of the simulator). When a signal assignment is specified with zero delay (the default), the simulator makes the assignment after a delta delay unit. You can think of one delta delay as one iteration in the simulation environment. (*Also refer to* iteration).

design entity

Design entity is the primary abstraction level of a VHDL hardware model. The design entity represents a cell, chip, board, or subsystem. A VHDL design entity is composed of two main parts: an entity declaration and an architecture body.

design file

A design file contains source code for one or more design units. (*Also refer to* design unit).

design library

Refer to library.

design unit

A design unit is a portion of the hardware description (model) that can be contained and compiled in a separate design file. A design unit contains a context clause (library clause and/or use clause) and a library unit. The following are library units: entity declarations, configuration declarations, architecture bodies, package declarations, and package bodies. The ability to store design units in separate files allows you to modularize a design description by compiling each entity or package declaration separate from the corresponding body. This ability is also useful so packages can be shared by multiple entities.

discrete array

A discrete array is a one-dimensional array that contains elements that are enumeration or integer types.

driver

A driver contains the projected output waveform for a signal. You use the signal assignment statement to change the value of the projected output waveforms that are in the driver for a signal. The value of a signal is related to the current values of its drivers.

entity

Refer to design entity.

entity declaration

Entity declaration is a language construct that defines the interface between the design entity and the environment outside of the design entity. The entity declaration begins with the reserved word **entity.**

entity header

Entity header is a language construct that declares the interface for the design entity (using ports and generics), which enables it to communicate with other items in the design environment.

enumeration literal

An enumeration literal is used in an enumeration type definition to declare specific values composed of an identifier(s) or character literal(s). The following type declaration defines a type called `my_qsim_state` to have values of ’X’, ’0’, ’1’, and ’Z’:

```
TYPE my_qsim_state IS (’X’,’0’,’1’,’Z’); --Uses 4 char. lit.
```

errors

You can encounter two kinds of errors when using a VHDL implementation: compile-time and run-time. Errors encountered when running the compiler are called compile-time errors. The compiler checks the source code for proper syntax and semantics and displays any errors encountered. Errors encountered while simulating a VHDL model are called run-time errors.

event

In model simulation, event refers to a change in a signal value.

execute

To execute means to carry out the instructions and/or evaluate the algorithms described in an explicit or implied VHDL process.

expression

An expression is a mathematical formula that, when evaluated, computes a value or set of values.

external block

Refer to block.

formal

A formal is a generic or port of a design entity or a parameter of a subprogram.

formal port

Refer to port.

format effector

A format effector is a non-printable control character you use to format the text in your source file. There are five format effectors used in VHDL. The following list shows the VHDL format effectors:

- Tab
- Line feed
- Vertical tab
- Form feed
- Carriage return

function

A function is one kind of subprogram. (*Refer to* subprogram). A function has the following characteristics: it produces no side-effects; it accepts only input (**in**) parameters; it returns just one value; and it always uses the reserved word **return**.

generic

A generic is a channel for static information to be passed from an environment to an internal or external block. A generic allows you to reuse a single design entity by passing in constants such as delays, temperature, and capacitance. With each different use of the design entity, different values can be supplied for the constants.

globally static expression

A globally static expression is an expression that can be evaluated when the design hierarchy where the expression appears is elaborated. The values for globally static expressions may depend upon declarations that appear in other design units. The values for globally static expressions are determined when the design unit is elaborated.

guard

Refer to guard expression.

guard expression

A guard expression is a mathematical formula that evaluates to a boolean value that is used to control the operation of certain statements within a block. When a guard expression is evaluated and found to be true, all guarded assignments within the block are executed. If the guard expression is false, the guarded assignments do not execute. (*Also refer to* guarded assigment).

guarded assignment

A guarded assignment is a concurrent signal assignment statement that uses the optional reserved word **guarded**. The statement does not execute unless the associated guard expression evaluates to a true condition.

guarded signal

A guarded signal allows you to control the assignment of signal values. The guard is a boolean expression. If the Boolean expression is FALSE, the guard assigns a null transaction to the drivers of the guarded signal, which turns off the drivers. If the value of the guard is TRUE, the signal assignment is made. Guarded signals must have resolution functions if they are a bus or register. There are two methods for guarding signals:

- Specifying **register** or **bus** as the signal kind in a signal declaration.
- Specifying **guarded** in a concurrent signal assignment.

hidden declaration

A hidden declaration is a declaration that can not be seen within a given scope. With homographs in different scopes, the inner declaration hides the corresponding outer declaration within the inner scope region. If a homograph exists, then one of the declarations is not visible. Homographs within the same scope create an error. (*Also refer to* homograph).

homograph

A homograph is a pair of declarations that have a special relationship to each other. Two declarations are homographs of each other if they both use a common identifier and overloading is allowed for at most one of the two declarations. There are two homograph cases: one declaration can be overloaded and the other cannot; or both declarations can be overloaded and they have the same parameter and result type profile. Only enumeration literals or subprogram declarations can be overloaded.

index constraint

An index constraint is used with constrained arrays to specify a subset of values for the range of the array indices such as `(1 TO 25)` in the following example:

```
TYPE int_array IS ARRAY (1 TO 25) OF integer; --constr. arr.
```

inertial (delay)

Inertial delay refers to a type of delay used in VHDL signal assignments. If the reserved word **transport** is not used in the right-hand side of a signal assignment, a default inertial delay for the waveform is assumed. An inertial delay applied to a waveform indicates that pulses with a width shorter than the specified delay time will not be transmitted to the target.

information hiding

Information hiding is a coding principle that means certain information from a module of code is hidden from other modules. This principle helps make VHDL designs manageable and easier to read. When coding a particular hardware module, it may be desirable to hide the implementation details from other modules. This principle complements abstraction, which extracts the functional details in a given module. By hiding implementation details from other modules, a designer's attention is focused on the relevant information, while the irrelevant details are made inaccessible.

iteration

An iteration is a simulator time unit that is greater than zero but less than one timestep. A timestep is the smallest time increment in the simulator. Iterations are used because the simulator is actually a serial processor that must process concurrent hardware events. All concurrent processes are evaluated in the same timestep as far as the simulation is concerned, but multiple iterations may be required to evaluate the concurrent processes completely.

iteration scheme

An iteration scheme is a VHDL construct used within a loop statement to control the execution of a loop.

iterative statements

Iterative statements include the loop statement, next statement, and the exit statement. These iterative statements allow you to write code that can repeatedly execute a sequence of statements.

language construct

Refer to construct.

lexical element

Lexical elements are the items used to form the VHDL language. A lexical element is one of the following: an identifier (or a reserved word), a comment, a literal (numeric, character, or string), or a delimiter.

library

VHDL libraries are classified into two groups: working libraries and resource libraries. The working library is the library in which the compiled design unit is placed. The analogy to the working library is your working directory. When you compile the design unit, it exists in the working directory in which you performed the compilation. There is only one working library during the compilation of a design unit.

The resource library is a library that is referenced within the design unit when it is compiled. There can be any number of resource libraries for a given design unit. The working library itself can be a resource library.

literal

A literal is a lexical element such as a number, character, or string that represents themselves. For example, the numbers "1064" represents a decimal literal for integer one thousand, sixty-four.

local

A local is a special name for a generic or port in a component declaration.

locally static expression

A locally static expression is an expression that can be completely evaluated when the design unit in which it appears is evaluated. The values for locally static expressions depend only on those declarations that are local to the design unit or on any packages used by the design unit. For more information, refer to the *Mentor Graphics VHDL Reference Manual* in the "Static Expressions" subsection.

local port

Refer to port.

mode

This VHDL construct is optionally used in an interface declaration to specify which direction that information flows through an object's channel of communication. The mode is designated with one of the following reserved words:

in:	The interface object can only be read.
out:	The interface object value can be updated but not read.
inout:	The interface object can be read and updated by 0 or more sources.
buffer:	The interface object can be read and updated by, at most, one source.
linkage:	The interface object can be read and updated only by appearing as an actual corresponding to an interface object of **linkage** mode.

modularity

Modularity is a coding principle that refers to the partitioning (or decomposing) of a hardware design and associated VHDL description into smaller units.

named notation

(Also called named association.) An association is considered named when an association element explicitly matches the actual part with a formal part. For example, the formal parameters in the following procedure (lines 1 and 2) are explicitly associated with a corresponding actual part in the function call in line 3. The association in line 3 uses the named notation. *(Also refer to* positional notation).

```
1 PROCEDURE integer_4bit(CONSTANT i_in    : IN integer;
2                  SIGNAL  i3, i2, i1, i0: OUT my_qsim_state);
3  integer_4bit(i_in => count, i3 => qd, i2 => qc,
4              i1 => qb, i0 => qa);
```

named

Each declared item must have a name. Names formally designate one of the following: explicitly or implicitly declared items, subelements of composite items, or attributes.

objects

Objects are the containers for values of a specified type. Objects are either signals, variables, or constants. Object values are used and manipulated with a set of operators or subprograms.

object code

Object code is the simulator-compatible database generated by the VHDL compiler from the VHDL source code. Users cannot directly modify object code with an editor. Object code is modified by changing the source code contents and then recompiling.

overloading

Overloading is the term that describes the process of using the same name for two or more different enumeration literals or subprograms (functions or procedures) within the same scope. The following example shows how enumeration literals `red` and `green` are overloaded by appearing in two separate enumeration definitions in an area of code that has an overlapping scope:

```
TYPE wire_color IS (red, black, green); --custom enum. type
TYPE traffic_light IS (yellow,red,green,flashing); --Ovrload
```

overload resolution

Overload resolution is the method used by a VHDL compiler to determine the actual meaning of an overloaded enumeration literal or subprogram (function or procedure) name. Using the type declarations (`wire_color` and `traffic_light`) from the overloaded definitions, a compiler uses the overload resolution method to determine that the enumeration literal `red` in the following example actually refers to the one in the `wire_color` declaration:

```
SIGNAL pos: wire_color; --Declare sig. of type wire_color
 ...
pos := red; --Uses "red" from wire_color type declaration
```

package

A package consists of the VHDL package declaration and package body constructs to allow you to group a collection of related items for use by one or more separate modules of code. Among the items that can be grouped together in packages are: type and subtype declarations, subprograms (functions and procedures), constants, and signals. Packages can be compiled and stored separately from the rest of the hardware description (in a design file) to facilitate sharing between hardware designs.

passive process

The term passive process describes one kind of process that can appear in a VHDL model. A process is called a passive process if no signal assignment statement appears in a process or a procedure called by the process. A passive process can appear in the entity statement part of a design entity. *(Also refer to* persistent process).

persistent process

Persistent process is a process that, once executed, exists forever. You use the process statement to define a process, which is used to contain a series of sequential actions that execute during simulation. *(Also refer to* passive process).

port

A port is the channel for signal input/output communications between an internal or external block and the environment. Three possible configuration modes for ports are **in**, **inout**, **out**, **buffer**, and **linkage**. There are three kinds of ports:

- Formal ports: These are specified in the entity declaration.
- Local ports: These are specified in the component declaration.
- Actual ports: These ports, within a component instantiation statement, map to the local ports of the component declaration with the reserved words **port map**. Actual ports in an instance can be connected to formal ports by using the configuration specification.

positional notation

(Also called positional association.) Positional notation is one way to associate an actual port to a corresponding formal port. When an association element does not explicitly specify which actual port matches a corresponding formal port, the association is made by the position of each element (positional association). For example, the order of elements in line 3 of the following example causes `count` (in the procedure call) to associate with `I_in` (in the procedure of line 1). They are both the first element. The second element in the procedure call (`qd`) associates with the second element in the procedure (`i3`). This positional association continues until all actual parts in the procedure call are associated with a corresponding formal part in the procedure.

```
1  PROCEDURE integer_4bit(CONSTANT I_in  : IN integer;
2                   SIGNAL i3, i2, i1, i0: OUT my_qsim_state);
3   integer_4bit (count, qd, qc, qb, qa) ; -- positional not.
```

primary

A primary (also known as an operand) is a quantity on which an operator performs an operation within an expression.

procedure

A procedure is one kind of subprogram. (*Refer to* subprogram). A procedure has the following characteristics: it can produce side-effects; it does not have to return any value or can return multiple values; it does not require the reserved word **return**; and it accepts input (**in**), output (**out**), input/output (**inout**), **buffer**, and **linkage** parameters.

process

Refer to passive process and persistent process.

register

The reserved word **register** in a signal declaration allows you to control the assignment of a value to resolved signal. Signals that are declared as a register, retain the last output value when all drivers are disconnected. *(Also refer to* guarded signal and bus).

record type

A record type is a composite type whose values consist of named elements.

reserved words

A reserved word is one that has specific meaning to a VHDL compiler, such as the word **port**. Certain characters, such as the left and right parentheses and the semicolon, are also classified as reserved words.

resolution function

A resolution function is a user-defined subprogram that determines what single value a signal should have when there are multiple drivers for that signal. Every signal you define that is the target of a signal assignment has a driver. If the signal has more than one driver (is a target for more than one signal assignment statement), you need to define a resolution function. The resolution function is called every time the signal is active.

resolved signal

A resolved signal is a signal with an associated resolution function.

resource library

Refer to library.

RLL

RLL (rotate left logical) is one of the System-1076 predefined multiplying operators. The left operand type is a one-dimensional array of any type or any integer type. The right operand is a non-negative value of any integer type.

RRL

RRL (rotate right logical) is one of the System-1076 predefined multiplying operators. The left operand type is a one-dimensional array of any type or any integer type. The right operand is a non-negative value of any integer type.

scope

Scope is the region of code where a declaration has effect. The scope of a declared identifier starts at the point where the identifier is first named and extends to the end of the description unit (subprogram, block, package, process) that contains the declaration.

semantics

Semantics are the rules that determine the meaning of the VHDL constructs as they are used in a hardware description. The following example shows a line of code (line 4) that has the correct syntax but incorrect semantics. A signal declared to be of type integer cannot be assigned to a signal of type my_qsim_state. The compiler checks for both semantic errors and syntax errors and displays an error message when a rule has been violated. *(Also refer to* syntax).

```
1  SIGNAL sig1 : integer;
2  SIGNAL sig2 : my_qsim_state;
3    ....
4  sig2 <= sig1;  -- Syntax correct, Semantics incorrect.
```

separators

Separators and delimiters are characters that divide and establish the boundaries of lexical elements. When you put lexical elements together, sometimes you must use a separator between the elements. Otherwise, the adjacent lexical elements could be construed as being a single element. There are three lexical separators:

- Space character. Except when the space is in a comment, string or character literal.
- Format effector. Except when the format effector is in a comment or string literal.
- End of line. Consists of the line feed character.

sequential statements

Sequential statements represent hardware algorithms that define the behavior of a design. You use sequential statements in a process or a subprogram (procedure or function). Each statement executes in the order encountered.

short-circuit operation

Short-circuit operation is a predefined logical operation (**and**, **or**, **nand**, and **nor**) for operands of types bit and boolean. In a short-circuit operation, the right operand is evaluated only if the left operand does not possess adequate information to determine the operation result. In other than short-circuit operations, both the left and right operands are evaluated by the compiler before the predefined operator is applied.

signal

A signal is a VHDL object that you can assign projected values to. It has a history and a time dimension.

slice

A slice name designates a one-dimensional array that is created from a consecutive portion of another one-dimensional array. A slice of a design item equates to a new design item of the same type. In the following example, line 9 assigns a slice of `array_a` to `array_b`.

```
1  PROCESS (sens_signal)
2    TYPE ref_arr IS ARRAY (positive RANGE <>) OF integer;
3    VARIABLE array_a : ref_arr (1 TO 12); --declare array_a
4    VARIABLE array_b : ref_arr (1 TO 4);  --declare array_b
5  BEGIN
6    FOR i IN 1 TO 12 LOOP  -- load array with values 1 - 12
7      array_a (i) := i + 1;
8    END LOOP;
9    array_b := array_a (6 TO 9); -- slice of "array_a"
10 END PROCESS;
```

SLL

SLL (shift left logical) is one of the System-1076 predefined multiplying operators. The left operand type is a one-dimensional array of any type or any integer type. The right operand is a non-negative value of any integer type.

source code

Source code refers to the combination of VHDL constructs that model a hardware system’s behavior. Source code is created and stored in files that can be edited by the user. The source code (files) are compiled to produce the simulator-compatible database.

specification

A specification associates additional information with a VHDL description. There are three types of specification: attribute, configuration, and disconnection.

SRA

SRA (shift right arithmetic) is one of the System-1076 predefined multiplying operators. The left operand type is a one-dimensional array of any type or any integer type. The right operand is a non-negative value of any integer type.

SRL

SRL (shift right logical) is one of the System-1076 predefined multiplying operators. The left operand type is a one-dimensional array of any type or any integer type. The right operand is a non-negative value of any integer type.

structural description

A structural description is the VHDL method for describing the hardware design as an arrangement of interconnected components.

subprogram

A subprogram allows you to decompose a hardware system into behavioral descriptions or operations using algorithms for computing values. The hardware's high-level activities are declared using a subprogram declaration. The actual operations are implemented in a subprogram body. Subprograms have two forms: procedures and functions.

subtype

A subtype is a subset of a previously-declared type defined with a specific range or index constraint. You can use a subtype if you know that a set of values is within a range of the original type. The system automatically checks the subtype value range for you in this case. *(Also refer to* base type).

syntax

Syntax refers to the formal rules that specify the form of a VHDL description. The syntax specifies how specifications, declarations, statements, and other constructs should be written. The VHDL compiler checks the correctness of a VHDL description against the set of syntax and semantic rules. The compiler generates error messages when discrepancies are found.

The following example shows a line of code (line 1) that has incorrect syntax. *(Also refer to* semantics).

```
1  SIGNAL sig1 : my_qsim_state --Syntax incorrect, missing (;)
2  SIGNAL sig2 : my_qsim_state;
```

timestep

A timestep is the unit of time assigned to the smallest time increment in the simulator. The timestep value (simulator resolution) can be changed within the simulator.

transaction

A transaction is a value and a time for the value to occur on a driver. In the statement `a <= ’1’ AFTER 10 ns;` a transaction is defined for the driver of signal "a". The value is ’1’ and the time is the relative delay of 10 ns.

transport (delay)

Transport delay refers to a type of delay used in VHDL signal assignments. The optional reserved word **transport** is used in the right-hand side of various signal assignments to specify the type of delay associated with the first waveform element. A transport delay indicates any pulse will be transmitted to the target, no matter how short the duration/width. The waveform exhibits an infinite frequency response. If the reserved word **transport** is not used in a signal assignment, a default inertial delay for the waveform is assumed.

type

A type declaration forms a template that describes objects that you declare, such as signals, constants, and variables. There are predefined types (such as bit and bit_vector) and an infinite number of types (templates) that you can specify. The following declaration shows how the type `bit` is declared:
`TYPE bit IS (’0’, ’1’);` Signals declared to be of type `bit` can have a value of ’0’ or ’1’. No other value is allowed.

unconstrained array definition

An unconstrained array is an array in which you specify the type of the indices, but do not specify the range. In place of specifying the range, you use the box symbol "<>". In this way you can declare the array type without declaring its range, and then you can declare as many arrays of the same type with the range you desire. This feature allows you to pass arrays of arbitrary sizes as parameters. The following example shows an unconstrained array definition:

```
TYPE data_array IS ARRAY(integer RANGE <>) OF integer;
```

uniformity

Uniformity is a coding principle that means each module of code is created in a similar way using the various VHDL building blocks. Uniformity helps to make your hardware description readable. This implies good programming style such as consistent code indentation and informative comments.

variables

A variable is one type of VHDL object. (Signals and constants are also objects.) A variable has one current value associated with it that can be changed in a variable assignment statement using the variable assignment delimiter (:=). A variable has no history.

visible

An item is visible at a point in the code if it is legal (according to the rules of visibility) to refer to that item at that point.

visibility

Visibility refers to the region of code where a declaration is visible.

waveform

A waveform is a series of transactions associated with a driver. The transactions indicate the future values of the driver in an order with respect to time.

working library

Refer to library.

INDEX

INDEX [continued]

INDEX [continued]

INDEX [continued]

INDEX [continued]