

This document is for the sole use of Giovanni Ferreira of Aluno

# VHDL Language and Application

Course Version 9.0

Lecture Manual

Revision 1.0

cadence®

# This document is for the sole use of Giovanni Ferreira of Aluno

© 1990-2022 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

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

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

**Restricted Print Permission:** This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

The publication may be used solely for personal, informational, and noncommercial purposes;

The publication may not be modified in any way;

Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and

Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence customers in accordance with, a written agreement between Cadence and the customer.

Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

## Table of Contents

### VHDL Language and Application

<b>Module 1</b>	<b>About This Course .....</b>	<b>2</b>
<b>Module 2</b>	<b>VHDL Application .....</b>	<b>9</b>
<b>Module 3</b>	<b>Language Introduction .....</b>	<b>20</b>
<b>Module 4</b>	<b>Signals and Data Types .....</b>	<b>38</b>
<b>Module 5</b>	<b>VHDL Operators .....</b>	<b>60</b>
<b>Module 6</b>	<b>Sequential Statements .....</b>	<b>73</b>
Lab 6-1	Familiarization with the Multiplexer Design	
Lab 6-2	Familiarization with Your VHDL Simulator	
Lab 6-3	Familiarization with Your Synthesis Tool	
Lab 6-4	Expanding the Multiplexer	
Lab 6-5	Adding the Alarm Signal	
<b>Module 7</b>	<b>Concurrent and Sequential Statements .....</b>	<b>93</b>
<b>Module 8</b>	<b>The Simulation Cycle and Process Control .....</b>	<b>108</b>
Lab 8-1	Adding a Seven-Segment Display Driver	
<b>Module 9</b>	<b>Variables and Sequential Statements .....</b>	<b>132</b>
<b>Module 10</b>	<b>Arithmetic Operators .....</b>	<b>149</b>
<b>Module 11</b>	<b>VHDL Coding Styles .....</b>	<b>170</b>
<b>Module 12</b>	<b>The Synthesis Process .....</b>	<b>184</b>
<b>Module 13</b>	<b>Definition of RTL Code .....</b>	<b>196</b>
Lab 13-1	Alarm Register	

<b>Module 14</b>	<b>Synthesis of Mathematical Operators .....</b>	<b>217</b>
	Lab 14-1 Pulse Generator	
<b>Module 15</b>	<b>Finite State Machine (FSM) Design and Analysis .....</b>	<b>238</b>
	Lab 15-1 Alarm Clock Controller	
<b>Module 16</b>	<b>Synthesis Coding Styles .....</b>	<b>257</b>
	Lab 16-1 Full Counting Mechanism for the Alarm Clock	
<b>Module 17</b>	<b>Functions and Procedures .....</b>	<b>271</b>
<b>Module 18</b>	<b>Advanced Concurrent VHDL .....</b>	<b>288</b>
	Lab 18-1 Time and Alarm Adjustment	
<b>Module 19</b>	<b>Advanced Data Types .....</b>	<b>301</b>
	Lab 19-1 Putting It All Together	
<b>Module 20</b>	<b>Testbench Coding Styles .....</b>	<b>319</b>
<b>Module 21</b>	<b>Testbench Applications .....</b>	<b>338</b>
	Lab 21-1 Script-Driven Testbench	
<b>Module 22</b>	<b>Gate-Level Simulation .....</b>	<b>350</b>
<b>Module 23</b>	<b>Application of Configurations .....</b>	<b>363</b>
<b>Module 24</b>	<b>Design Organization and Management .....</b>	<b>378</b>
<b>Module 25</b>	<b>Course Conclusions .....</b>	<b>391</b>
<b>Module 26</b>	<b>Next Steps.....</b>	<b>394</b>
<b>Appendix A</b>	<b>VHDL200X .....</b>	<b>398</b>

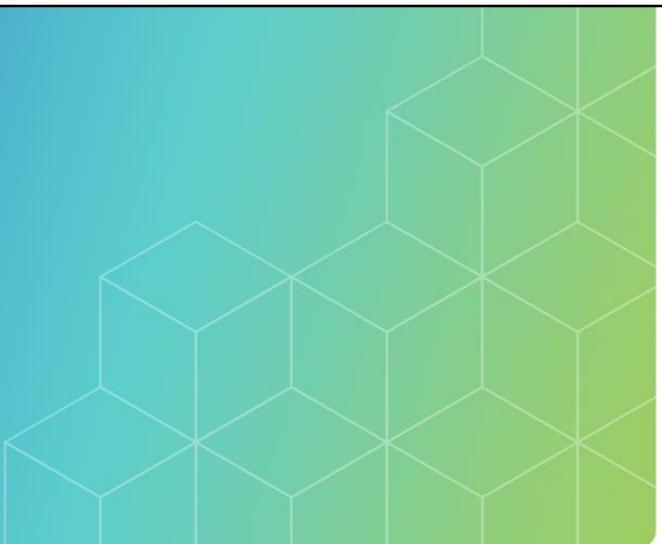
VHDL Language and Application

Version VHDL 9.0

Estimated time: 5 Days

**cadence®**

*This page does not contain notes.*



## About This Course

### Module

**1**

Revision

**1.0**

Version

**VHDL 9.0**

Estimated Time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Course Prerequisites

Before taking this course, you need to:

- Have a good working knowledge of any programming language
- Have experience with coding
- Have knowledge of logic design

Knowledge of the following is helpful but not essential:

- Some familiarity with Cadence® Xcelium™ simulator



Before taking the course, it is good to have an excellent working knowledge of any programming language, have some coding experience, and knowledge of logic design. Also, knowledge of the Xcelium simulator is helpful though not necessary

## Course Objectives

In this course, you

- Use basic and advanced VHDL language details, and advanced application issues, including:
  - Synthesis coding styles
  - Synthesis coding styles to build sophisticated designs and testbenches
- Analyze design organization and management
- Use real-world hardware-orientated examples
- Set up and run simulation using the Xcelium simulator

Following the course

- Spend several months of “hands-on” work to become experienced

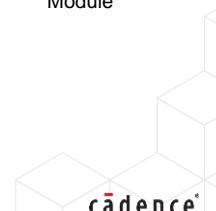


In this course, you will use basic and advanced VHDL language details and advanced application issues, including synthesis coding styles, to build sophisticated designs and testbenches, analyze design organization and management; use real-world, hardware-orientated examples; and set up and run a simulation using Xcelium simulator.

## Course Agenda

<b>Day 1</b>	<b>Day 3</b>
1. Course Introduction	12. The Synthesis Process
2. Application Introduction	13. Definition of RTL Code
3. Language Introduction	14. Synthesis of Mathematical Operators
4. Signals and Data Types	15. FSM Design and Synthesis
5. VHDL Operators	16. Synthesis Coding Styles
6. Sequential Statements	
<b>Day 2</b>	<b>Day 4</b>
7. Concurrent and Sequential Statements	17. Functions and Procedures
8. Simulation Cycle and Process Control	18. Advanced Concurrent VHDL
9. Variables and Sequential Statements	19. Advanced Data Types
10. Arithmetic Operators	20. Testbench Coding Styles
11. VHDL Coding Styles	21. Testbench Applications
	<b>Day 5</b>
	22. Gate-Level Simulation
	23. Application of Configurations
	24. Design Organization and Management
	25. Course Conclusions

Introduction Module      Synthesis      Advanced Module



5 © Cadence Design Systems, Inc. All rights reserved.

The instructor-led training follows this schedule. The first two days, we cover only basic concepts. On the third day, we deal with synthesis, and on days 4 and 5, we deal with more advanced topics. Over five days, the course covers about 85% of the language, the parts that are used 99% of the time!

## Software and Licenses

For the software and licenses used in the labs for this course, go to:

[https://www.cadence.com/content/cadence-www/global/en\\_US/home/training/all-courses/82109.html](https://www.cadence.com/content/cadence-www/global/en_US/home/training/all-courses/82109.html)

If there is additional information regarding the specific software, it is detailed in the lab document and/or the README file of the database provided with this course.



*This page does not contain notes.*

## Conventions: VHDL Code

lowercase:  
VHDL keywords

UPPERCASE:  
User-defined names

```
entity CONVENTIONS is
  port (A, B : in bit;
        C    :out bit);
end CONVENTIONS;

architecture RTL of CONVENTIONS is
begin
  C <= A XOR B;
end RTL;
```



These conventions may differ from those  
used inside your own company

Beware



Throughout the course, we follow this convention, where VHDL keywords are lowercase, and user-defined names are in uppercase. These conventions may differ from those you use in your own company.

## Conventions: Icons



Synthesis

Use of VHDL that affects typical synthesis tools



Tip

A tip or hint in using the language effectively



Beware

A particular issue to watch out for that may cause problems



Update

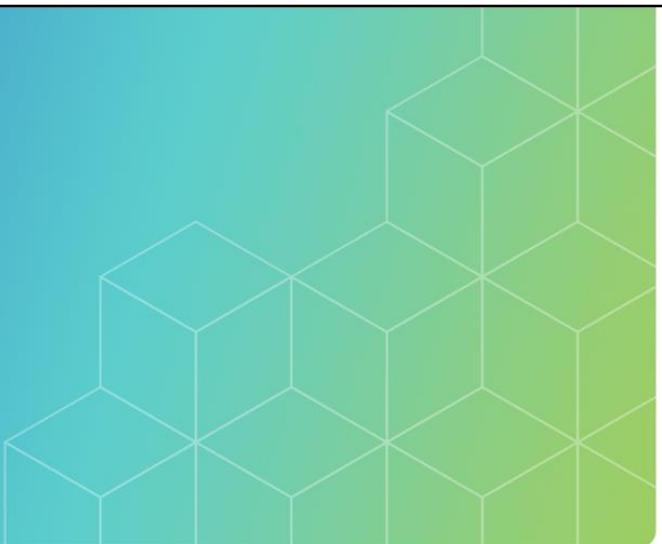
Language features made available in the VHDL'08 update  
(may not yet be supported)



Throughout this class, we use icons to draw your attention to certain kinds of information. Here are the icons we use and what they mean.



## VHDL Application



**Module** **2**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Define and describe the below application areas of VHDL
  - VHDL definition
  - Levels of abstraction
  - Design flow
  - Benefits and issues



In this module, you define and describe below the application areas of VHDL, VHDL definition, levels of abstraction, design flow, benefits, and issues.

## What Is VHDL?

VHSIC Hardware Description Language

- VHSIC – Very High-Speed Integrated Circuit
- Modeling DIGITAL Electronic Systems

International Standard

- IEEE Std 1076

Versions

- VHDL'87, VHDL'93, VHDL2001, VHDL2006, VHDL2008 (major update), VHDL2019

Pure Language Definition

- Not application/methodology standard

Applications

- ASIC and FPGA designers writing RTL code for synthesis
- System architects doing high-level simulations
- Verification engineers writing test benches for all levels of simulation
- Model developers describing ASIC/FPGA cells, or macros

11 © Cadence Design Systems, Inc. All rights reserved.



VHSIC was a US Department of Defense research initiative begun in the early 1980s to look into ASIC technology for defense applications. One of the objectives for VHSIC was to improve project documentation by designing a language to document ASIC designs.

The language was focused on digital hardware design. It included constructs to handle concurrency (things happening simultaneously, working in parallel like hardware) and software-like, sequential constructs to provide a powerful way of describing complex behavior.

In 1987 the IEEE released the language as a full IEEE standard, IEEE 1076. IEEE rules require that standards are reviewed every five years to keep them open to user input, so an updated version of VHDL was released in 1993. A minor update was released in 2001, but more significant changes are included in the VHDL2006 standard.

The IEEE standards only define the language, not a methodology to use the language. We will look into these issues as we go through the training course.

VHDL is still developing – the IEEE has agreed upon analog extensions to the language. Other issues, such as adding object-orientated features to VHDL, are being looked into as part of the VHDL200x initiative.

Applications of VHDL include:

- ASIC and FPGA designers writing RTL code for synthesis
- System architects doing high-level simulations
- Verification engineers writing test benches for all levels of simulation
- Model developers describing ASIC/FPGA cells or macros

## Hardware Description Language (HDL)

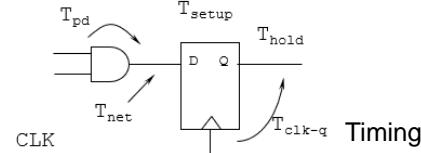
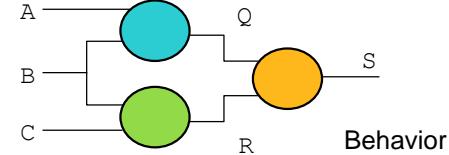
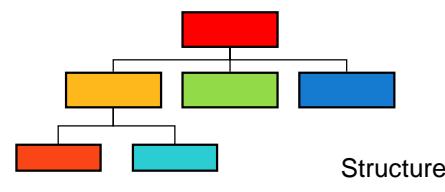
VHDL is not a software language

VHDL is a Hardware Description Language (HDL)

- Programming language with special constructs for modeling hardware

VHDL supports:

- Structure
  - Physical (netlist and hierarchy)
  - Software (subprograms)
- Hardware behavior
  - Serial (sequential)
  - Parallel (concurrent)
- Timing
- Abstraction levels



12 © Cadence Design Systems, Inc. All rights reserved.

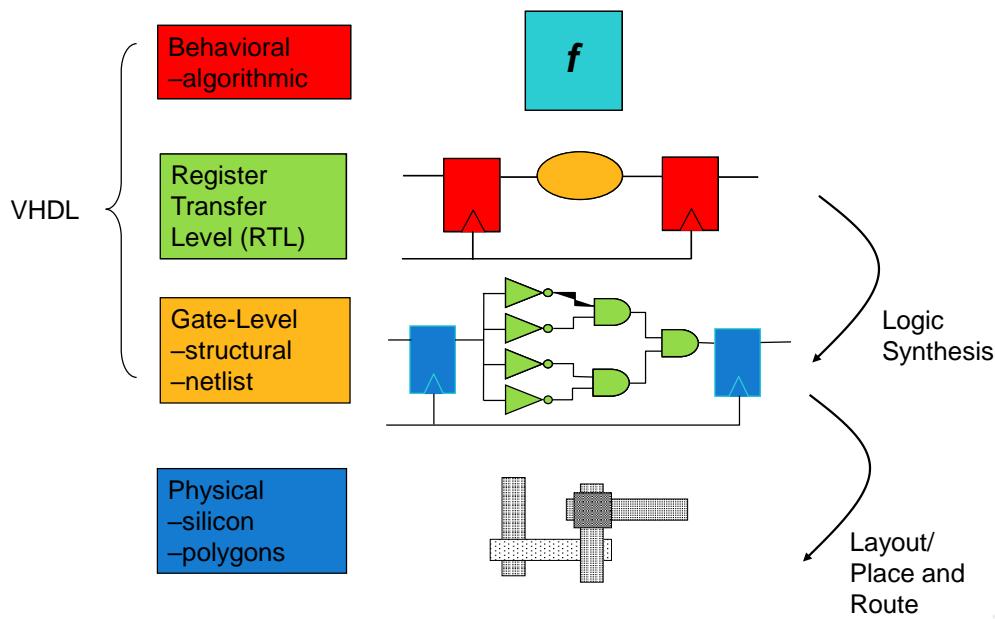


VHDL is an HDL, not a conventional software language like C, Pascal, Fortran, or Basic. An HDL contains high-level programming language constructs and special constructs describing digital electronic systems.

VHDL contains features and constructs to support the description of:

- Structure – Both physical, such as hierarchical block diagrams and component netlists, and software, such as subprograms. This allows us to describe large, complex systems and manage design complexity.
- Hardware behavior – both serial and parallel. In serial behavior, the output of one functional block is passed to the input of another, which is similar to the behavior of a conventional software language. However, in parallel behavior, a block output can be passed to the inputs of several blocks acting in parallel, and the outputs of these blocks may eventually converge back together. VHDL has to support concurrent behavior, where many separate events will be happening at the exact moment in time
- Time – Typically, programming languages have no concept of time. There are propagation delays, clock periods, and timing checks to be modeled in hardware.
- Different abstraction levels – As we shall see, a hardware function can be described at several levels of abstraction, from a high-level algorithmic model to a low-level netlist.

## Levels of Abstraction: Definition



13 © Cadence Design Systems, Inc. All rights reserved.



At each level of abstraction, you can describe a system as a group of hierarchical models in varying amounts of detail.

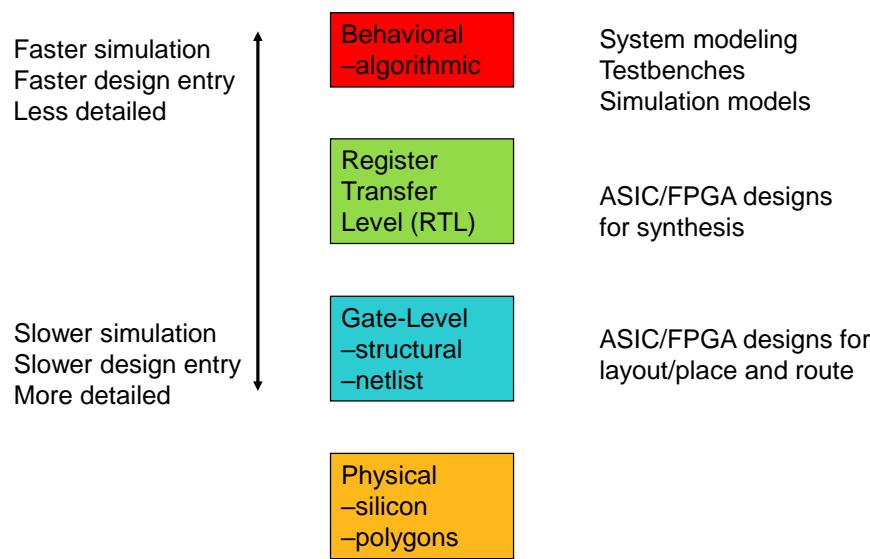
Within VHDL, models can be written with different levels of detail. The three primary levels of abstraction are:

- **Behavioral** – System described using mathematical equations. No timing may be defined – the system may simulate like a software program in zero time.
- **Register Transfer Level (RTL)** – System partitioned into combinational and clocked logic, using constructs and coding styles supported by synthesis. Timing is defined in terms of cycles based on the defined clock(s)
- **Structural (Gate-Level)** – A netlist of technology-specific low-level gates and cells from a specific vendor's technology linked by net and bus connections. Structural models may also be used for block-diagram representations of design hierarchy.

Layout deals with silicon structures and polygons to create basic logic functions.

EDA tools facilitate the translation between abstraction levels, as shown.

## Tradeoffs Between Abstraction Levels



14 © Cadence Design Systems, Inc. All rights reserved.



It is important to understand that as you move to higher abstraction levels, you give up details about your design to gain faster capture and simulation.

So, test benches and simulation models are written in behavioral level code for faster design entry and simulation.

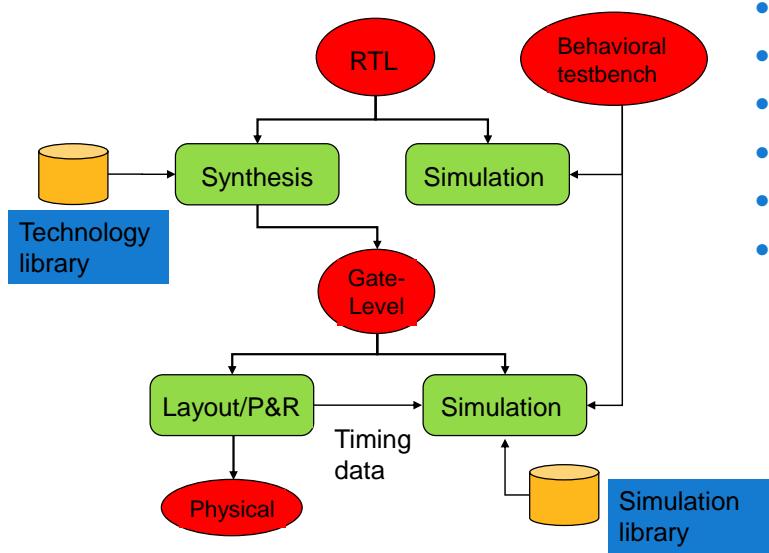
ASIC or FPGA designs must be entered in more detail at RTL. This will take more time to enter and simulate since it must include combinational/registered logic details for the synthesis tool. The synthesizer's output is translated to layout for place and route by EDA tools.

Some areas of design may be entered at the logic level (e.g., hand-crafted asynchronous logic).

Higher abstraction levels allow you to describe your design functionality without being concerned about the details of its implementation.

Some hardware designers find this concept hard to understand or accept at first.

## Simplified Design Flow Sequence



- Create an RTL design
- Create a behavioral testbench
- Simulate to check functionality
- Synthesize to gate-level netlist
- Simulate to check functionality and timing
- Layout/P&R

15 © Cadence Design Systems, Inc. All rights reserved.



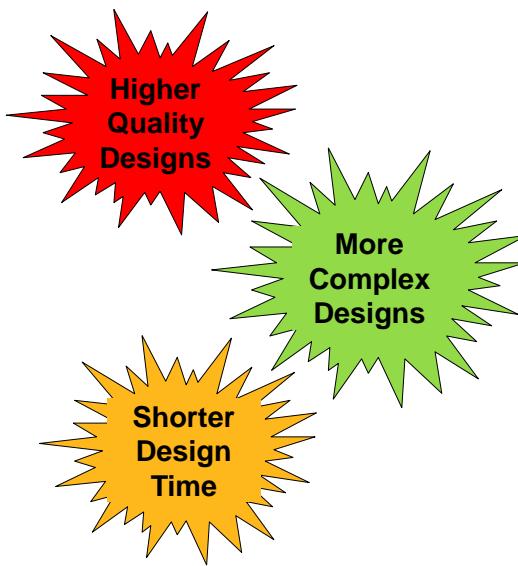
Typically, a design is entered directly in RTL VHDL. The RTL must be functionally verified using a VHDL simulator. A testbench model is written to apply a stimulus, extract results and verify results. Since the testbench will not be synthesized, there is no restriction on the use of language constructs or modeling styles. Hence, they are written in behavioral level VHDL.

The design is read into a synthesis tool and the specified target technology for the design. The design is then synthesized into a gate-level netlist. This can be verified using the original RTL testbench.

The layout is then performed for ASIC designs or Place and Route (P&R) for FPGA/CPLD designs. Timing information can be extracted from the layout tools and "back-annotated" into the netlist to perform function and timing verification using a simulator.

The flow described here is very simplified, and many variations exist. If your company uses HDLs, it should have a design flow. Otherwise, if you are new to using HDLs, understanding and developing a design methodology is a crucial first step in using HDLs.

## The Benefits of Using VHDL



Design at a higher level

- Find problems earlier
- Explore alternatives

Implementation independent

- Last-minute changes
- Delay implementation decisions

Flexibility

- Re-use
- Choice of tools, vendors

Language-based

- Faster design capture
- Easier to manage



16 © Cadence Design Systems, Inc. All rights reserved.

Higher-level VHDL design entry allows the simulation to be carried out sooner and quicker than other design entry methods (e.g., schematic). Earlier simulation allows earlier debugging – easier and quicker with HDL code. Quicker simulation allows the possibility for different architectures of an algorithm to be investigated. This allows a better chance of selecting an optimal architecture and partitioning.

RTL code is mainly implementation-independent – the target ASIC or FPGA technology is selected during synthesis. Choice of technology can be deferred until most of the design is entered. Also, an alternate technology or vendor can be selected with a minimum of the redesign.

The standard format allows the reuse of designs from previous projects or commercial Intellectual Property (IP) providers. Can also move or switch between different tools/vendors without (in theory) re-entry, reformat, or translation of design description.

ASCII language description is simpler to enter and edit (with a simple text editor).

## Issues in Using VHDL

- Steep learning curve
  - New programming language to understand
  - Simulation and synthesis of EDA tools to learn
- No standard "off-the-shelf" methodology
  - Needs to be planned and implemented
  - Probably mixing tools from different vendors
- Aimed at digital design only
  - Although analog extensions do exist (VHDL-AMS)
- Correct coding styles influence project success
- Much design planning and partitioning required before coding
- Other specifics you will see during the course

17 © Cadence Design Systems, Inc. All rights reserved.



Adopting VHDL requires learning a new language, knowing at least two new EDA tools (simulator and synthesis tool), and changing to a more software-like design methodology.

How you write your code can affect the speed and efficiency of simulation and synthesis tools and the performance and area of the final design. Cadence courses can teach you efficient, flexible, and portable coding styles for describing your designs and using specific language constructs for easy design entry, maintenance, and reuse.

## VHDL Application: Summary Quiz

1. What level of VHDL is used in:
  - a) Testbenches
  - b) Synthesizable designs
  - c) Netlists
2. How is VHDL implementation-independent, and why is this an advantage?



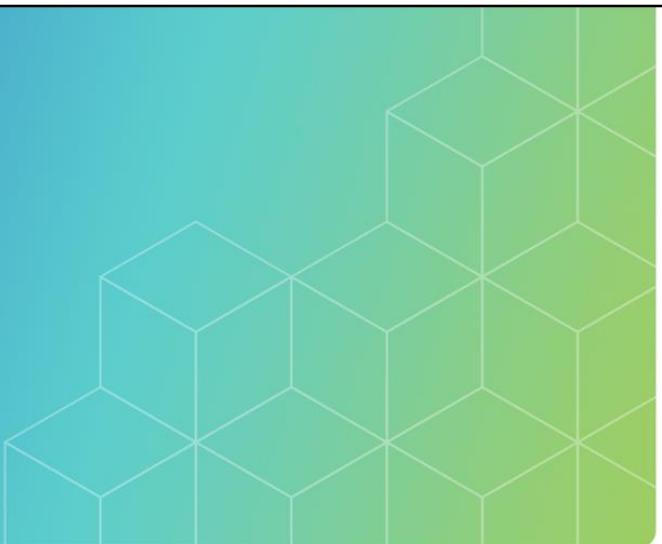
*This page does not contain notes.*

## Solutions: VHDL Application: Summary Quiz

1. What level of VHDL is used in:
  - a) Testbenches
    - Behavioral
  - b) Synthesizable designs
    - Register Transfer Level (RTL)
  - c) Netlists
    - Gate-level or structural
2. How is VHDL implementation-independent, and why is this an advantage?
  - VHDL can be implementation-independent because we don't need to specify the target implementation technology until synthesizing the design. This allows us to retarget a design at a different technology simply by selecting a different technology in synthesis.



*This page does not contain notes.*



## Language Introduction

**Module** **3**

Revision **1.0**

Version **VHDL 9.0**

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Explore main language concepts
- Define and describe:
  - VHDL design units
    - Entity, architecture, configuration, package, and package body
  - VHDL connection model
- Outline hierarchy and describe how it is built in VHDL
- Define compilation libraries and state compilation order for a given VHDL model
- List all the naming rules and regulations



By the end of this module, you will explore main language concepts, define and describe basic VHDL design units such as entity, architecture, configuration, package, and package body, define how these blocks are connected to build a VHDL model, outline hierarchy and describe how it is built in VHDL, define compilation libraries and state compilation order for given VHDL model.

List all the naming rules and regulations.

## Entity

Describes interface only

No definition of behavior

Port list must include:

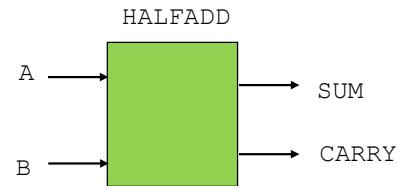
- Port name (A, B..)
- Port mode (in/out/inout)
- Port type, e.g. std\_logic

library and use statements enable the use of std\_logic type

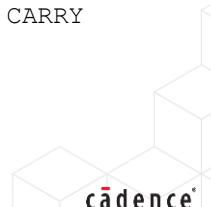
- More later

```
library IEEE;
use IEEE.std_logic_1164.all;

entity HALFADD is
    port ( A, B      : in  std_logic;
           SUM, CARRY : out std_logic);
end HALFADD;
```



22 © Cadence Design Systems, Inc. All rights reserved.



First, we look at the entity.

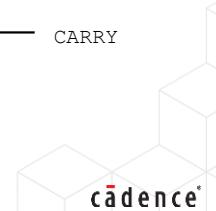
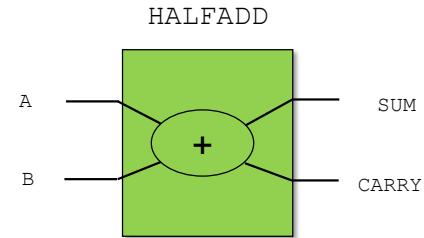
Here is the entity declaration for a half-adder block. Notice that the entity defines a name for the block (HALFADD in this case) and the inputs and outputs for the block (the port statement defines two inputs, A and B, and two outputs, SUM and CARRY). Notice that we also define what sort of information passes through the port, std\_logic, in this case. This is called the "type" of the port; we will discuss this more in the next section. To use the std\_logic type, we must add a library and use statements before the entity. More on this later.

You can think of an entity as similar to a symbol in a schematic design; it defines the name of the block and the interface to the outside world, but it doesn't define the behavior of the block. To do this, VHDL uses another construct, the architecture.

## Architecture

- Describes the behavior of the entity
- Must be associated with a specific entity
- Single entity can have many architectures
- begin separates declarations from functional code

```
architecture RTL of HALFADD is
  -- declarations can go here
begin
  SUM  <= A xor B;
  CARRY <= A and B;
end RTL;
```



23 © Cadence Design Systems, Inc. All rights reserved.

Here is the architecture declaration for our half-adder. The outputs SUM and CARRY are defined in terms of inputs A and B, so we have described the behavior of our half-adder. The xor and functions used here are not gates but VHDL operators. This way of describing the behavior uses concurrent VHDL constructs (we can reverse the order of the statements and still get the same behavior!). We will discuss the differences between concurrent and sequential VHDL in later sections.

Notice that the architecture is associated with the HALFADD entity and has a unique name, RTL. VHDL allows multiple architectures for each entity to have different versions of the same design stored in your VHDL database, e.g., a behavioral model, an RTL description, and the synthesized gates. Later we will see how to select the one you want to use. The begin keyword separates the declarative region from the functional code.

In this architecture, we use concurrent signal assignments to calculate values for the output ports of the entity from the input ports. We will look at signal assignments and operators in more detail in later modules of this course.

## Creating Hierarchy: Component Declaration

```

library IEEE;
use IEEE.std_logic_1164.all;
entity T_HALFADD is
end T_HALFADD;

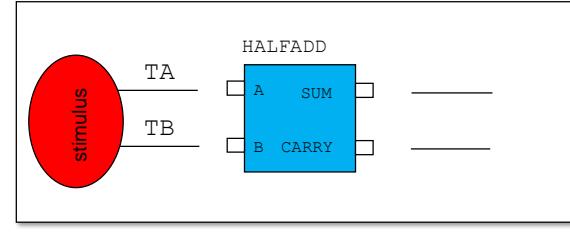
architecture BENCH of T_HALFADD is

component HALFADD
port( A, B      : in  std_logic;
      SUM, CARRY : out std_logic);
end component;

signal TA, TB : std_logic;
signal TSUM, TCARRY : std_logic;

begin
-- component instantiations
-- stimulus generation
end BENCH;

```



T\_HALFADD

- Components to be instantiated must be declared within the architecture
- Signals must also be declared to connect to component ports



Component and signal declarations  
before begin of architecture

Beware



24 © Cadence Design Systems, Inc. All rights reserved.

Now we have fully described our half-adder, we can write a testbench to check if it works correctly. We need to create an “instances” of the half-adder block in the testbench; connect local “wires” to the instance ports, and drive input stimulus into the half-adder.

First, we need an entity for the testbench. The testbench is the top-most block in our design hierarchy; it does not have any external connections. Therefore, it does not have a port list. Then we need an architecture; in which we will "instantiate" a half-adder block and connect it up to the stimulus generation process.

VHDL requires that we declare any hierarchical blocks we instantiate as components. The declaration must define the name of the block and its interface so that the VHDL compiler can check that the block is used consistently. The component declaration is very similar to the entity.

We also need some internal “wires” to connect the half-adder; we create these using the signal declaration. Notice that we have defined the “type” of the signals (std\_logic in this case) just as we did with ports, but with signals, we don't define a “direction.”

## Creating Hierarchy: Component Instantiation

```

library IEEE;
use IEEE.std_logic_1164.all;

entity T_HALFADD is
end T_HALFADD;

architecture BENCH of T_HALFADD is

component HALFADD
  port ( A, B      : in  std_logic;
         SUM, CARRY : out std_logic);
end component;

signal TA, TB : std_logic;
signal TSUM, TCARRY : std_logic;

begin

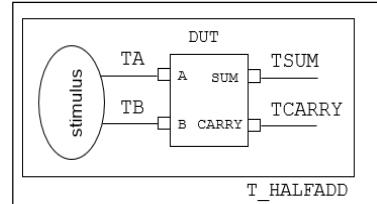
DUT : HALFADD port map (A => TA, B => TB, SUM => TSUM, CARRY => TCARRY);
-- stimulus generation

end BENCH;

```

Each instantiated component has a:

- Unique name
- Port map to connect component ports to local ports and signals
  - Here, using named association



25 © Cadence Design Systems, Inc. All rights reserved.

Once we have declarations, we can instantiate and connect our hierarchical block. We create unique copies of a component using a component instantiation statement, which is placed in the architecture body after the begin statement. The component instantiation gives a unique name to the instance, DUT; in this case, it defines which component we are instantiating and specifies a port map. The port map defines how the components' ports are connected to the local ports and signals in the architecture.

In the code, you can see DUT: HALFADD; this creates an instance of the HALFADD block and labels it DUT, just like placing a symbol down on a schematic sheet. The port map statement connects up the block.

There are two alternative methods of specifying port mapping. The first method uses named association. Each connection is defined by explicitly mapping each component port to the required signal or port of the architecture, i.e.:

<name>: <component> port map (<port> => <signal>, ...)

This is known as named association port mapping because we state the name of the component port to which each signal or port is mapped. The individual port mappings can be placed in any order.

## Positional Association

```

library IEEE;
use IEEE.std_logic_1164.all;

entity T_HALFADD is
end T_HALFADD;

architecture BENCH of T_HALFADD is

component HALFADD
  port ( A , B      : in  std_logic;
         SUM, CARRY : out std_logic);
end component;

signal TA, TB : std_logic;
signal TSUM, TCARRY : std_logic;

begin

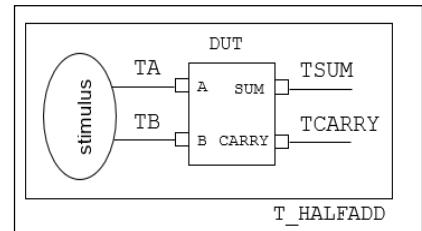
DUT : HALFADD port map (TA,      TB,  TSUM,  TCARRY); <----- Connection here

-- stimulus generation

end BENCH;

```

- Local ports/signals mapped to component ports based on order of declaration



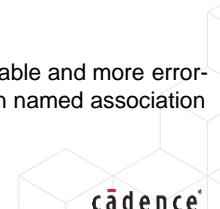
1st item in the port map (TA) linked to 1st component port (A)

2nd item in the port map (TB) linked to 2nd component port (B), etc.



Less readable and more error-prone than named association

Beware

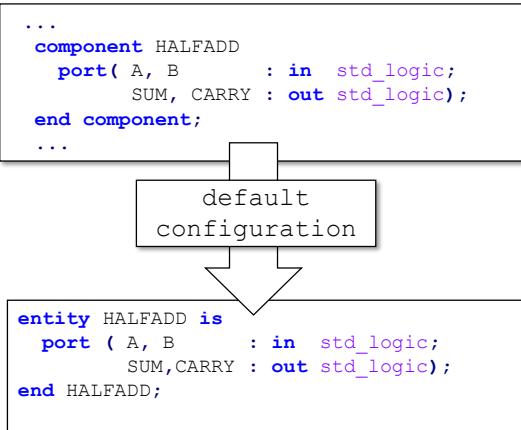


Positional association port mapping can also be used. In this method, the port map contains a list of signals/ports local to the architecture. Connection is made on a positional basis – the first signal/port is mapped to the first port in the component declaration, the second signal/port to the second component port, etc.

The positional association may be simpler to use, but it is less readable than named mapping and more error-prone.

For example, consider if we accidentally swapped TSUM and TCARRY in the port map above. The design would still compile and simulate, but with the SUM and CARRY output data for instantiation DUT swapped over. This would lead to strange functional behavior, which may be very difficult to debug. Any ambiguity and possible misunderstanding are reduced by explicitly stating which component port is mapped to which local port/signal using a named association.

## Connecting Hierarchy: Configurations



Use copy/paste to ensure port lists match

Tip

- Component declaration/instantiation creates a "socket"
  - Not a direct connection to an entity
- If certain rules are followed, component declaration and an existing entity are automatically linked
  - Called default configuration
- Rules for a default configuration
  - Entity and component names *must* match
  - Port names/types *must* match (can be declared in a different order)
- Otherwise, an explicit configuration must be written to link the hierarchy
  - Covered later



To build up a hierarchy in VHDL, we declare components and instantiate them to provide the “sockets” into which we plug “chips.” In this case, the “chip” is an entity and a single architecture of that entity that defines the interface and behavior. The VHDL construct that “plugs” the entity/architecture into the component “socket” is the configuration statement.

The configuration statement defines which entity and which architecture of that entity will be used to plug into a particular component socket. You can think of it as a “parts list” for the design.

When the names and port lists of the component and the entity are identical, we don't need to write an explicit configuration. The simulator can link the component and entity automatically. This is called a default configuration. If you are using a default configuration but have more than one architecture for an entity, the last architecture to be compiled is the one that is used. Otherwise, an explicit configuration is to be written to link hierarchy which is covered later.

A default configuration is the only configuration method generally supported by synthesis tools.

## Package

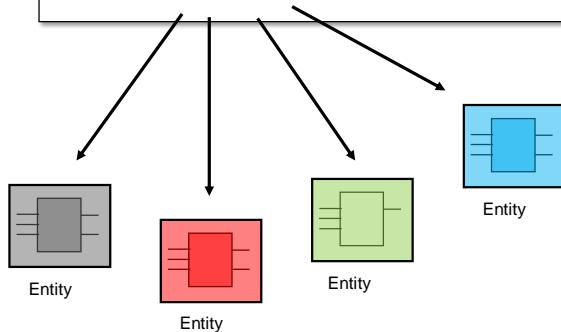
Collection of declarations

- Types
- Constants
  - Typed object with a specific value
- Components
- Subprograms

Can be referenced by any entity/architecture

```
library IEEE;
use IEEE.std_logic_1164.all;

package SYSTEM_CONSTS is
    constant HIGH : std_logic := '1';
    constant LOW : std_logic := '0';
end SYSTEM_CONSTS;
```



We said earlier that some of the basic constructs in VHDL help you manage your design data rather than describe the behavior or structure of the design. One of these constructs is the package; you can use a package to group together a number of related definitions, which can then be shared by several designs. Using a package ensures that the same definitions are used consistently by all the designs.

Not all VHDL constructs can be used in packages; you cannot put entities or architectures in packages, but you can define types, VHDL constants, components, and VHDL subprograms (a collective term for functions and procedures, which we will discuss later).

## Package Body

```
package DEMO_PACK is
  -- constants
  -- data types
  -- component declarations
  -- subprogram declarations
end DEMO_PACK;
```

```
package body DEMO_PACK is
  -- constant values
  -- local declarations
  -- subprogram definitions
end DEMO_PACK;
```

It cannot exist without a package

Most commonly contains:

- Definitions of subprograms
- Values of constants
- Local declarations

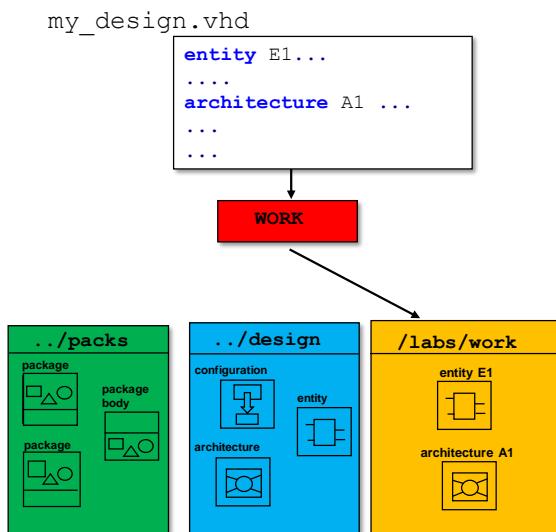


VHDL packages can have a package body associated with them. The most common use for a package body is to hold VHDL subprogram definitions (functional code as opposed to declarations), as we will see later. They can also be used for deferred constants, where a constant is declared in the package but only given a value in the package body. This gives us advantages in design re-compilation.

Subprograms in packages are described in the section on functions and procedures and covered in the section on arithmetic operators.

Deferred constants are described in Design Organization and Management, which contains much more information on packages.

## Compilation Library



Design files compiled into a library

- Physically exists as a directory
- Stores compiled design units
  - Entity, architecture, package, package body, configuration
- File "holder" thrown away

Large projects may use many libraries

The simulator option defines the library into which units will be compiled

- Called *working library* or *work*
- Mapped to the specific library by the simulator

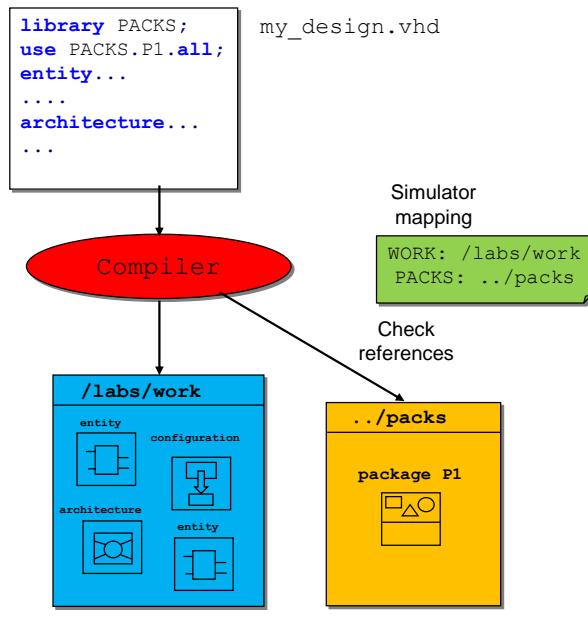


To simulate a VHDL model, we must first convert our source files into a binary form that the simulator can recognize. The process of checking the syntax and producing the binary file is known as compilation.

When you compile a design in VHDL, you compile it into a data structure known as a library. Libraries have logical and physical names. A library is usually a directory on the computer's disc where the compiled, binary design units like entity, architecture, package, package body, etc., are placed. Large projects use many libraries. The actual physical directory is referenced by a logical name called `work`, which acts as a pointer to this directory. The analyzed code is compiled into an implicitly declared `work` library and is mapped to the actual path by the simulator. So the simulator-specific steps to compile a VHDL design are:

1. Create a compilation library or libraries (if not already created).
2. Point the `WORK` logical name to the library into which you wish to compile.
3. Compile the VHDL code.

## Referencing Packages



The design unit can reference a package from the same or a different library

Referenced via:

- Library declaration
- Use clause

Library referenced by *logical name*

Simulator specifies mapping:

- Logical name -> library directory

References create dependencies between design units

Standard libraries are available:

- IEEE is a built-in library
- std\_1164 is a predefined package



When a simulator creates a compilation library directory, the library is given a logical name. There will be a simulator-specific mapping mechanism to map a library logical name to a particular library's directory.

Declarations in a package can be accessed in another design unit by referencing the package. The package is referenced by specifying the library into which the package has been compiled (library declaration) and then the package's name in this library (use clause).

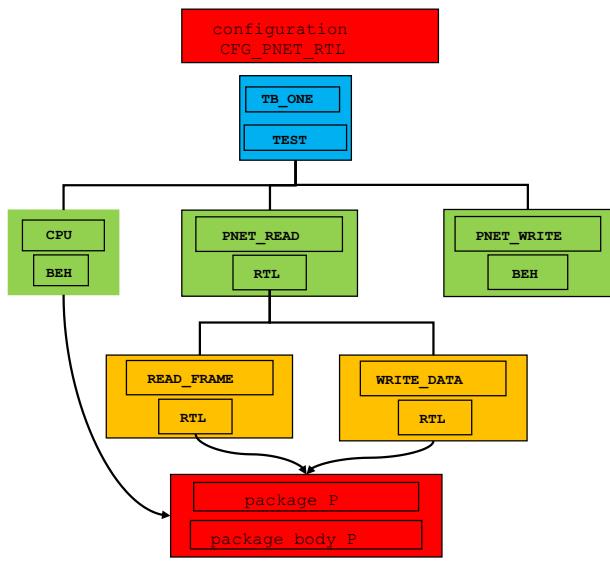
The library declaration uses the logical name for the library, e.g., library PACKS.

The use clause also references the logical name with the name of the package use PACKS.P1.all; i.e., reference all the declarations from package P1 in the library with the logical name PACKS.

The compiler will use the simulator mapping to find the library directory for PACKS; look for the compiled package P1 and check the declarations.

The simulator always compiles code into the current working library, selected using the logical name WORK. References create dependencies between design units. There are few standard libraries, like IEEE, that contain standard 1164 and other packages.

## Compilation Order



Entity before architecture

Package before body

Design unit before referenced

- Package before entity/architecture referencing it
- Configuration last

Compilation order

- Packages first
- Entities/architectures
- Configurations last

Bottom-up

32 © Cadence Design Systems, Inc. All rights reserved.



A hierarchical design contains several dependencies; we need to understand these to compile the design successfully.

Architectures reference the name of the entity they are associated with, so we should compile the entity before any architectures. Similarly, packages should be compiled before their package bodies because a package body references a package. If the entity and architecture or the package and body are in the same file, they will be compiled simultaneously.

If an entity uses definitions provided in a package, then the package should be compiled before the entity. Similarly, configurations should be compiled after the entities/architectures they reference. The general rule is that a design unit should be compiled before any other design unit that references it.

We can use these rules to work out the compilation order for a design; package and package body first, then entities followed by their architectures, and finally the configuration.

**NOTE:** Although technically it doesn't matter in which order we compile the design components, some simulators prefer the details to be compiled "bottom-up" (e.g., READ\_FRAME and WRITE\_DATA; then PNET\_READ, etc.). This is a good approach to follow.

## Naming Rules

Names may consist of letters, numbers, and underscores

VHDL is case insensitive

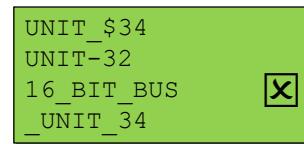
- These names all refer to the same object
  - ABC, Abc, abc



Names must start with a letter

Names can be of any length

- Tool or methodology may restrict name lengths
- From VHDL'93, illegal names can be used for an object
  - By delimiting the name with the \ character
  - Called an extended identifier



33 © Cadence Design Systems, Inc. All rights reserved.



VHDL has strict rules for naming design units (entities, architectures, packages, and configurations) and design objects (signals, types, subprograms, etc.).

The only allowable characters are letters (uppercase or lowercase), numbers, and underscores. This means characters like \$ and - are not allowed.

VHDL is case insensitive – both names and language keywords can be either uppercase or lowercase, but the same name will always refer to the same object, e.g., ABUS, Abus, and abus are all the same object.

Many companies adopt specific naming styles to make code more readable and maintainable.

From VHDL'93, illegal names can be used for an object by delimiting the name with the \ character, e.g., \UNIT\_\$32\ is allowed. These are called extended identifiers. This feature is intended to help with VHDL code generators (e.g., schematic to VHDL translators where schematic names may include illegal characters) or gate-level component libraries (which may contain cell names like 2AND or 4OR). Note that extended identifiers are distinct from normal identifiers and are case sensitive, i.e., all the following are separate objects:

unit, \unit\, \Unit\, \UNIT\

## Comments and Spacing

```
-- this is a line comment.  
-- each line must begin with a --  
-- comments end with a new line  
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity HALFADD is  
  port (A, B      : in std_logic;  
         SUM, CARRY : out std_logic);  
end HALFADD;      -- end of line comment  
  
architecture BEHAVE of HALFADD is  
begin  
  -- VHDL is a free-format language  
  -- additional space can be used to enhance readability  
  SUM  <= A xor B;  
  CARRY <= A and B;  
end BEHAVE;
```

34 © Cadence Design Systems, Inc. All rights reserved.

cadence®

VHDL comments are created using --; anything on a line to the right of the “double-dash” is taken as a comment. If the line begins with -- then the whole line is a comment. VHDL does not provide multi-line comments; if you want to use comment headers at the beginning of VHDL files to define version control information, change histories, and block descriptions, then every line in the header must begin with --.

## VHDL Introduction: Summary Quiz

1. With what must every architecture be associated?
2. How is a particular component instance linked to an entity/architecture pair?
3. Write the component instantiation for the following:



35 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## A Strict Language

- VHDL is stringent
- Types and directions of signals connected must match
- All objects must be declared before they can be used
- Goal: trap errors at compile time, not after hours of simulation
- You will grow to like it!



We have now seen enough VHDL to get a flavor of the language, and it is clear already that VHDL forces you to be precise in your design description. For example, you must declare a type for every signal and port, and you can only connect signals and ports of the same type; all objects must be declared before they are used and compiled correctly.

The purpose of all this precision is not to frustrate you! (Although sometimes you may feel it is!) VHDL forces you to be unambiguous in your design definition to eliminate many bugs during compilation and not after hours of simulation time. Your VHDL compiler is trying to help you!. With more hands-on experience, you will grow to like it.

## Solutions: VHDL Introduction: Summary Quiz

1. With what must every architecture be associated?
  - An entity
2. How is a particular component instance linked to an entity/architecture pair?
  - A configuration, either explicit or default, links an instantiation of a component to a specific entity/architecture pair.
3. Write the component instantiation for the following:



```
XAP1 : ALU port map (A => DATA, B => CTRL, CLK => CLK,  
RST => RST, AVER => AVER, DATA => ACCUM);
```

37 © Cadence Design Systems, Inc. All rights reserved.



### Question 3:

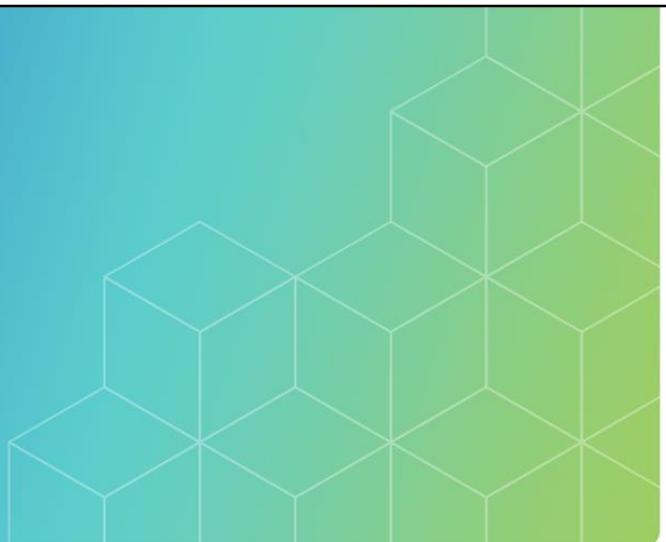
The format for the instantiation is as follows (using named port mapping):

```
<instance> : <component> port map ( <port> => <signal>, ...);
```

Therefore, the instantiation is:

```
XAP1 : ALU port map (A => DATA, B => CTRL, CLK => CLK,  
RST => RST, AVER => AVER, DATA => ACCUM);
```

Note: There is a signal named DATA in the instantiation and a port named DATA in the component, but the signal and port are mapped to different connections in the port map. This is fine, and VHDL can deal with this situation, but it isn't very clear. In a real-life example, you might consider renaming the DATA signal so it could not be confused with the port.



## Signals and Data Types

**Module** **4**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

We have looked at how we pass information around in VHDL using signals and the different types these signals can have. Now we will look at how we can do useful work on the information in the signals using operators.

## Module Objectives

In this module, you

- Define and describe the type concept
- List and use all predefined types and packages
- Define your own user-defined data types
- List all the rules regarding signal assignments
- Declare and perform basic array operations
- Work with multi-dimensional arrays
- Define enumerated data types and their use in FSM
- Build an array of arrays
- Build large arrays using aggregates
- Use records as data structures



In this module, you will be able to define and describe the type concept, list and use all predefined types and packages, define your own user-defined data types, list all the rules regarding signal assignments, declare and perform basic array operations, work with multi-dimensional arrays, define enumerated data types and their use in FSM, build an array of arrays, build large arrays using aggregates and use records as data structures.

## Concept of a Type

The type must be defined when the signal is declared

Either in:

- architecture, in a signal declaration
- Port section of an entity

Types must match up!

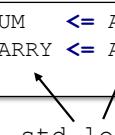
```
library IEEE;
use IEEE.std_logic_1164.all;

entity FULLADD is
    port (A, B, CIN      : in std_logic;
          SUM, CARRY : out std_logic);
end FULLADD;

architecture STRUCTURAL of FULLADD is

    signal N_SUM,N_CARRY1,N_CARRY2 : std_logic;
    -- component declarations
begin
    -- component instantiations
end STRUCTURAL;
```

$\text{SUM} \leftarrow A \text{ xor } B;$   
 $\text{CARRY} \leftarrow A \text{ and } B;$



std\_logic on both sides

40 © Cadence Design Systems, Inc. All rights reserved.



Every time we declare a signal in VHDL, we need to specify its type. The type defines a set of possible values that the signal can carry.

There are two ways to declare signals in VHDL; the obvious one is the signal declaration in an architecture where we use the keyword signal followed by one or more names and the type of the signal(s) created. (Commas separate multiple names).

The other way of creating signals is in port declarations; signals are implicitly declared when you declare ports. In the case of a port, you also say a direction (or "mode" in VHDL terminology) and the type of the signal.

We can assign values to a signal using a signal assignment statement such as:

```
OUTPUT_SIG <= INPUT_SIG;
```

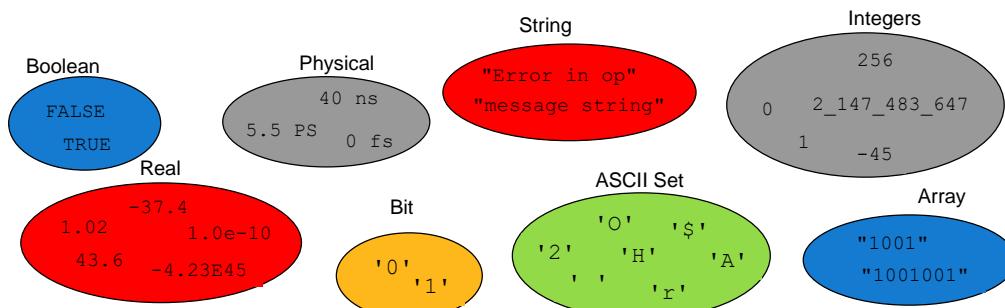
(This statement is equivalent to connecting two schematic nets or wires.) The important rule you need to remember is that the types of the signals on either side of the  $<=$  must be the same.

## Standard Predefined Data Types

Every type has a set of potential values

- Standard types – defined by language
- Built-in packages of additional types (std\_logic)
- Users can define their own types, as we shall see

```
package STANDARD is
    type BOOLEAN is (FALSE, TRUE);
    type BIT is ('0', '1');
    type CHARACTER is (-- ASCII set );
    type INTEGER is range <min 32 bit>;
    subtype NATURAL is integer range 0 to ...
    type REAL is range <min 64 bit>
    -- BIT_VECTOR
    -- STRING
    -- TIME
end STANDARD;
```



41 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Type **BOOLEAN** has the values FALSE and TRUE. Type **BIT** can have the values '0' and '1'.

VHDL also allows you to work with integers and real numbers using the types **INTEGER** and **REAL**. The range for integer is simulator-dependent but is at least:

-2,147,483,647 to +2,147,483,647

VHDL differentiates between **INTEGER** and **REAL** values using the decimal point: 42.0 is a real number, whereas 42 is an integer.

**TIME** is a physical type. Physical types have units (e.g., ms, ns) and relationships between units (1000ns = 1ms). Values of type **TIME** can be either integers or real numbers and must have units and a numeric value.

Array types have several elements of the same type grouped as one object, e.g., **BIT\_VECTOR** (an array of **BIT** elements) and **STRING** (an array of **CHARACTER** elements).

Note that **BIT** and **CHARACTER** values have single quotes around them. This helps us distinguish them from other values, e.g., **INTEGER** 0 or 1. This also allows us to define values for all the elements of a **BIT\_VECTOR** or **STRING** by enclosing the values inside double quotes, e.g., "100001" or "this is a string value."

The **STANDARD** package is listed in the *VHDL Reference Guide*.

## Review: Using Standard Logic Package

```

library clause      use clause
↓                ↓
library IEEE;
use IEEE.std_logic_1164.all;

entity STDDEC is
    port (A, B, SEL : in std_logic;
          OP      : out std_logic);
end STDDEC;

architecture RTL of STDDEC is
    signal CTRL : std_logic;
    ...
begin
    ...

```

- std\_logic\_1164 package supplied pre-compiled with every simulator
  - In predefined library IEEE
- Add library and use clauses to make the package visible
  - Can then use-package declarations



library and use clauses  
come before entity  
Beware

42 © Cadence Design Systems, Inc. All rights reserved.



The library clause makes the contents of a library visible (IEEE is the VHDL logical name for the library, which holds IEEE standard packages). The use clause makes all the definitions visible from the package std\_logic\_1164 compiled into the library IEEE.

The library IEEE and its packages are supplied pre-compiled into every VHDL simulator. The simulator contains a predefined mapping file that maps the library logical name IEEE to a directory installed with the simulator software. The use clause tells the compiler that we wish to use the package named std\_logic\_1164 from this library.

The .all part of the use clause tells the compiler to reference all the declarations found in the package. It is possible to pick individual declarations from a package via a use clause, but it usually indicates badly designed code or poorly partitioned packages. Therefore it is always better to use the .all option.

For any VHDL types which are not locally declared, the compiler will search the package(s) referenced in the use clause(s) for the type declarations. So when we use std\_logic, the compiler will look in the pre-compiled package std\_logic\_1164 in the library IEEE for the declaration of std\_logic.

We will get compilation errors if we use a type that is not declared in a referenced package or omit the use clause.

## Standard Logic Type

```

package std_logic_1164 is
    type std_ulogic is ( 'U', -- Uninitialized
                           'X', -- Forcing 0 or 1
                           '0', -- Forcing 0
                           '1', -- Forcing 1
                           'Z', -- High Impedance
                           'W', -- Weak 0 or 1
                           'L', -- Weak 0
                           'H', -- Weak 1
                           '-' ); -- don't care
    ...

```

Two forms with the same values:

std\_logic: resolved – allows multiple drivers

std\_ulogic: unresolved – does not allow multiple drivers



Standard Logic defines nine values that can describe the various conditions required for simulation and synthesis of logic signals, including signal drive strength (“forcing” or “weak”) which is useful for circuit-level simulation (modeling pullups, etc.).

Note that the first (leftmost) value in the `std_ulogic` declaration is ‘U,’ representing an un-initialized state. Therefore, objects of `std_ulogic`-based types will take this value at the beginning of the simulation. Successful design reset will assign the object to a meaningful value ('0', '1', 'Z' etc.). An object which remains at 'U' may not have been successfully reset. This allows us to detect and debug initialization problems.

`std_ulogic` is not a resolved type, so you cannot connect two drivers to a signal of type `std_ulogic` (the “u” in `std_ulogic` is for “unresolved”). However, real hardware often requires multiple drivers on the same signal, so another type, a resolved type called `std_logic`, is defined in the 1164 package. `std_logic` is closely related to `std_ulogic` and shares the same type values (closely related types are discussed in a later section).

## Signals and Drivers

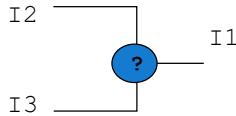
The signal value changed with a *signal assignment statement*

- This creates a *signal driver*

Types must match on either side of the assignment

Multiple drivers to the same target can cause compile errors

- Need a special *resolved type*



```
architecture RTL of E1 is
  signal A, B : std_logic;
  signal I1, I2, I3 : integer;
begin
  ✓ A <= B;
  ✗ B <= I2; -- type mismatch
  ✗ I1 <= I2 -- multiple...
  I1 <= I3; -- ...drivers
  ...
end RTL;
```

44 © Cadence Design Systems, Inc. All rights reserved.



We use the signal assignment statement to change the value of a signal.

In some ways, a signal assignment statement is like a hardware gate driving a wire; in fact, the proper VHDL technical term for the object putting a value onto a signal is a signal driver. The type of the signal driver and the signal must match – otherwise, your compiler will give an error message. Multiple drivers to the same target cause compile errors. We need a special resolved type for it.

## Resolving Multiple Drivers

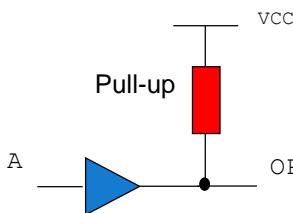
```

library IEEE;
use IEEE.std_logic_1164.all;

entity PULLUP is
    port (A : in std_logic;
          OP : out std_logic);
end PULLUP;

architecture BEHAV of PULLUP is
begin
    OP <= 'H';
    OP <= A;
end BEHAV;

```



All *standard* types are unresolved

- Multiple concurrent assignments to the same target give compile errors

Need a *resolved* type

- Multiple concurrent assignments allowed signals of resolved types
- Resolution functions* work out value on resolved signal
- Essential for modeling real hardware, e.g., tri-state buses, pullups

`std_logic` is the only resolved type commonly used

45 © Cadence Design Systems, Inc. All rights reserved.



When you connect the outputs of two gates together in hardware, you get problems. The same is true in VHDL; two signal assignments to the same signal would normally give a compiler error in VHDL because it is impossible to determine the value of the signal.

VHDL does allow you to drive the same signal with two different signal assignment statements in one special case; when the signal is of a resolved type. The basic idea is that when you make two or more signal assignments to a signal of a resolved type, a special function called a resolution function is called up to sort out the value on the resolved signal. We will discuss resolved types and resolution functions in detail later on. Resolved types are useful in modeling hardware such as tri-state buffers, pullups, etc.

## Using Standard Logic Types

```
library IEEE;
use IEEE.std_logic_1164.all;

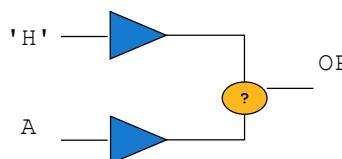
entity PULLUP is
    port (A : in std_logic;
          OP :out std_logic);
end PULLUP;

architecture BEHAV of PULLUP is
begin
    OP <= 'H';
    OP <= A;    
end BEHAV;
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity PULLUP is
    port (A : in std_ulogic;
          OP :out std_ulogic);
end PULLUP;

architecture BEHAV of PULLUP is
begin
    OP <= 'H';
    OP <= A;    
end BEHAV;
```



- `std_logic` is resolved
  - Multiple drivers are allowed
- `std_ulogic` is unresolved
  - Multiple drivers cause compilation errors

46 © Cadence Design Systems, Inc. All rights reserved.



`std_ulogic` and `std_ulogic_vector` are unresolved types. Therefore, you will get a compilation error if you assign two or more drivers to a signal of type `std_ulogic` or `std_ulogic_vector`. This would help debug models where you do not expect multiple drivers on signals.

`std_logic` and `std_logic_vector` are resolved types. Therefore, you can assign two or more drivers to a type `std_logic` or `std_logic_vector` signal. The resolution function works out what the final value of the signal should be. Multiple drivers are essential for modeling tri-state and bi-directional signals. Resolution functions are discussed in more detail in the Functions and Procedures module.

You can mix `std_logic` and `std_ulogic` types (although the strict VHDL type rules prevent mixing the array versions). In practice, you would use one type of family only.

The following is a lookup table for assignments of '`1`', '`0`', '`Z`' to `A`. The resulting value on `OP` is obtained by referencing the driven value on `A` with the constant drive value of '`H`' in the resolution function:

<code>A</code>	<code>'1'</code>	<code>'0'</code>	<code>'Z'</code>
<code>OP</code>	<code>'1'</code>	<code>'0'</code>	<code>'H'</code>

## Resolved or Unresolved Types

STD\_ULOGIC

STD\_LOGIC

std\_ulogic benefits:

- Gives errors when accidentally having two drivers

std\_logic benefits:

- Other standards based on it
  - Gate-level simulation
  - Mathematical functions
- Synthesis tools only output netlists in one data type
  - std\_logic required for tri-state logic
- Generally, no simulation speed overhead

Hence std\_logic is best used for RTL



Cadence recommends std\_logic for use in RTL design. Although using std\_ulogic provides a degree of extra error checking where we can know if there are any unintentional multiple drivers. std\_logic has the major advantage that many other standards are based on, such as gate-level simulation, arithmetic packages (IEEE1076.3), and others. Synthesis tools output netlists in one kind of data type. Hence, std\_logic is required for modeling tri-state. Std\_logic has no simulation speed overhead, therefore, best used for RTL.

## Arrays

```
signal A_BUS, Z_BUS : std_logic_vector(3 downto 0);
```

Collection of signals of the same data type

Z\_BUS <= A\_BUS;

std\_logic\_1164 defines

- std\_logic\_vector (array of std\_logic)

Language defines

- bit\_vector (array of bit)
- string (array of characters)

Define size when declaring the object



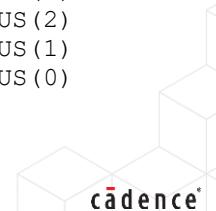
Size of the array on left and right  
of assignment must be equal

Beware

Z\_BUS (3) ← A\_BUS (3)  
Z\_BUS (2) ← A\_BUS (2)  
Z\_BUS (1) ← A\_BUS (1)  
Z\_BUS (0) ← A\_BUS (0)

Z\_BUS (3) <= A\_BUS (0);

Z\_BUS (3) ← A\_BUS (3)  
Z\_BUS (2) ← A\_BUS (2)  
Z\_BUS (1) ← A\_BUS (1)  
Z\_BUS (0) ← A\_BUS (0)



48 © Cadence Design Systems, Inc. All rights reserved.

VHDL has an equivalent to the idea of a “bus” in hardware design, where several wires are grouped together and treated as a single object. The VHDL equivalent to a bus is an array; you can declare signals or ports of array types and specify the number of elements in the array, just like specifying the number of bits in a bus. (Remember, two array types are provided as standard types; bit\_vector and string.)

The (3 downto 0) defines the range of the array; in this case, we will have four elements in the array A\_BUS referred to as A\_BUS (3), A\_BUS (2), A\_BUS (1), and A\_BUS (0). The leftmost element is A\_BUS (3) because our signal was declared as (3 downto 0). This is important when assigning our array signal because the assignment is by position (left to right) and not by element number.

We can make signal assignments using the individual elements of the array using the form signal\_name(index\_number), for example, Z\_BUS (3) <= A\_BUS (1);

VHDL also defines:

bit\_vector – an array of bits, rarely used.

string – an array of characters, useful in test benches and verification.

std\_logic\_1164 also defines:

std\_ulogic\_vector – an array of std\_ulogic, rarely used for reasons given previously.

## Array Index Direction

```
signal Z_BUS : std_logic_vector(3 downto 0);  
signal B_BUS : std_logic_vector(3 downto 0);  
signal C_BUS : std_logic_vector(0 to 3);
```

Z\_BUS <= B\_BUS;

Z\_BUS (3) ← B\_BUS (3)  
Z\_BUS (2) ← B\_BUS (2)  
Z\_BUS (1) ← B\_BUS (1)  
Z\_BUS (0) ← B\_BUS (0)

- Arrays can be defined in either direction
- `downto` is the de-facto standard

Z\_BUS <= C\_BUS;

Z\_BUS (3) ← C\_BUS (0)  
Z\_BUS (2) ← C\_BUS (1)  
Z\_BUS (1) ← C\_BUS (2)  
Z\_BUS (0) ← C\_BUS (3)



Elements are assigned by position, not element number

Beware



Be consistent in defining the direction of your arrays

Tip



49 © Cadence Design Systems, Inc. All rights reserved.

We can define the range of an array as `(0 to 3)` instead of `(3 downto 0)`. In this case, the leftmost array element would be `(0)` and not `(3)`.

We can connect two signals defined in these two ways because the number of array elements is the same in both cases. However, when we assign a signal declared as `(3 downto 0)` to another signal declared as `(0 to 3)`, we will follow the rule in VHDL of connecting by position, in order from left to right, and connect the two signals as shown on the slide.

In our example, if we meant element `(3)` to be the MSB, then we have just reversed the bus! Remember that MSBs and LSBs are not built into VHDL, so to avoid confusion, be consistent in defining the direction of your arrays. The `downto` direction is the most commonly used.

## Array Literal Assignment

```
signal Z_BUS : std_logic_vector(3 downto 0);
```

Z\_BUS <= 11;

Types must match on either side of the assignment

Z\_BUS <= "1011";

The size of the array on the left and right of the assignment must be equal

Z\_BUS <= "10111";

Underscores can be used in a literal value

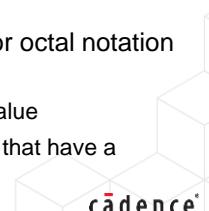
- Aids readability
- Double underscores are not allowed

Z\_BUS <= "10\_11";

Z\_BUS <= "10\_\_11";

Array literals can be specified using hex or octal notation

- Array and value size must match!
  - Each hex literal is a 4-element binary value
  - Can only be directly assigned to arrays that have a length of a multiple of 4



50 © Cadence Design Systems, Inc. All rights reserved.

From VHDL'93, array values can be assigned using hexadecimal (`X"<value>"`) or octal (`O"<value>"`) notation. Note: the size of the value and the size of the array must still match, i.e., hex values can only be assigned to arrays of length divisible by four, and octal values to arrays of length divisible by 3.

e.g. `O"23" = "010_011" = 6 elements`

Specifically, each hex literal expands to a four-element value and octal literal to a 3-element value. Therefore you could not assign a 6-element array using hex notation.

e.g. `H"23" = "0010_0011" = 8 elements`

Underscores can break up any literal value, e.g., `2_147_483_647` or `"1001_0011"`. Underscores must be separated by at least one digit.

## VHDL2008: Sized Literals

```
signal Z_BUS : std_logic_vector(5 downto 0);
```

VHDL2008 allows *sized literals*

- Literals are truncated or padded (by default with 0) to a set width
  - 6X"f"** = "001111"
- Works with X and Z values too
  - 6X"XX"** = "XXXXXX"
- Decimal D notation also allowed
  - 6D"27"** = "011011"
- Optional S modifier allows sign extension
  - 6SX"f"** = "111111"

VHDL  
08  
Update

vhdl2002

-- length mismatch 6 <= 8  
Z\_BUS <= X"27"



vhdl2008

-- length match 6 <= 6  
Z\_BUS <= **6X"27"**



X"7"	=	"0111"
X"ZZ"	=	"ZZZZZZZZ"
3X"7"	=	"111"
9X"F"	=	"000001111"
9SX"F"	=	"111111111"
8D"17"	=	"00010001"

**cadence®**

51 © Cadence Design Systems, Inc. All rights reserved.

From VHDL'93, array literals can be expressed using hex or octal notation. However, a single hex literal is always 4 bits, and rules requiring array sizes to be the same on both sides of an assignment limit the use of hex literal assignments to arrays with a length equal to a multiple of 4.

In VHDL2008, an optional size integer can be added before the hex or octal base specifier. The literal value is then truncated or padded (by default with 0 in the MSB) to the length specified by the size (which must match the length of the target array). The literal value can also include X and Z values. Now hex literals can be used to assign arrays of any size.

In addition, a decimal specifier is also defined for VHDL2008 literals.

By default, if the literal value is smaller than the size specifier, VHDL pads out the extra bits, from the MSB, with '0' up to the size specifier. For signed 2s-complement values, this would convert a negative value into a positive. Therefore, an extra S modifier is placed between size and base specifiers for signed literals. The S modifier pads out the literal value to the size width using sign extension, i.e., if the MSB of the literal is '1', the extra bits are padded with '1', maintaining sign information.

## Aggregates

```
signal A_BIT, B_BIT, C_BIT, D_BIT : std_logic;
signal Z_BUS : std_logic_vector(3 downto 0);
signal BYTE : std_logic_vector(7 downto 0);
```

- Aggregates create a value for an array from individual elements
  - Size of aggregate must match the size of the array

```
-- positional association
Z_BUS <= (A_BIT, B_BIT, C_BIT, D_BIT);
```

```
-- named association
BYTE <= (7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0');
```

- We can assign all the bits of BYTE to '0', independent of the width of BYTE, using just others

```
BYTE <= (others => '0');
```

52 © Cadence Design Systems, Inc. All rights reserved.



An aggregate allows us to construct an array from individual values by listing the values within brackets, separated by commas. The aggregate must be the same size as the array, and each aggregate value must be of the correct type for the array.

Aggregates can use named or positional associations.

In positional association, values are matched to array elements in order of declaration – first value to first array element, second value to second element, etc.

Each array element index is mapped to a specific value in a named association. Note that we can assign several elements using a "slice" construct:

```
5 downto 1 => '1',
```

but you can only assign individual bits in an aggregate; you cannot assign:

```
5 downto 1 => "10101", -- wrong
```

Using named association, we can use the other keyword to assign a value to any element in the array that has not already been explicitly assigned. We can also use others to assign all elements of an array without specifying the width.

Note that we cannot mix named and positional associations in any one aggregate signal assignment; we can use only one form or the other.

Aggregates can be used on either side of an assignment statement.

We can assign all the bits of BYTE to '0', independently of the width of BYTE used.

```
BYTE <= (others => '0');
```

## Enumerated Type Definitions

VHDL allows users to define their own types

- Declare type first, then declare objects of the type

One form is the *enumerated type*, used in designing FSM

Remember – types cannot be intermixed!

```
type MY_STATE is (RESET, IDLE, DMA_CYCLE, INT_CYCLE);
...
signal STATE : MY_STATE;
signal TWO_BIT : std_logic_vector(1 downto 0);
...
STATE <= RESET; 
STATE <= "00"; 
STATE <= TWO_BIT; 
```



Synthesis tools usually offer a way to map each enumeration to a bit pattern

Synthesis



As we mentioned, it is possible to define your own types in VHDL. One form is the enumerated type. These are particularly convenient for state machine descriptions where you can have a different type value for each state, e.g., RESET, IDLE, DMA\_CYCLE, etc. This is much more readable than bit patterns, e.g., "0110", and is easier to review, maintain, and re-use.

It is important to understand that enumerated types are different types covered by the same strict typing rules as other VHDL types. In the example on the slide, the values of type MY\_STATE cannot be replaced by a 2-element vector, even though we know that when we synthesize the code, any signals of type MY\_STATE would be represented by 2-bit busses. (Note that most synthesis tools provide a way of controlling how the enumerated type values are represented as bit patterns in the synthesized design.)

## User-Defined Arrays

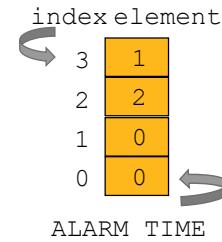
VHDL also allows users to define their own arrays

```
type T_CLOCK_TIME is array (3 downto 0) of integer;
signal CURRENT_TIME, ALARM_TIME : T_CLOCK_TIME;
```

```
ALARM_TIME(3) <= 1;
ALARM_TIME(2) <= 2;
ALARM_TIME(1) <= 0;
ALARM_TIME(0) <= 0;

-- assigned with aggregate
CURRENT_TIME <= (1,2,0,0);
```

```
CURRENT_TIME <= ALARM_TIME;
```



54 © Cadence Design Systems, Inc. All rights reserved.



As well as declaring your own enumerated types, you can declare your own array types. In the example on the slide, we created a 4-element array of integers to represent a 4-digit clock display. We can declare signals of this type, and we can assign to individual elements of these signals or assign all elements with the aggregate construct.

Notice that we need an aggregate assignment to assign all elements of the array; we cannot assign a value of "0123"; the rule in VHDL is that we can only use the "<value>" form if the type values of the array elements are all single ASCII characters (defined using single quotation marks, e.g., '1' in the type declaration for BIT).

In the case of integers, we can have multiple characters for each value – 10, 100, 4000. Therefore, we must have an aggregate assignment, where commas separate each element value.

## Arrays of Arrays

The element of an array can be another array

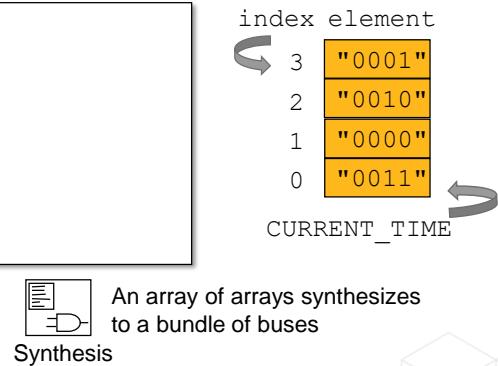
- This is called an "array of arrays"
- Multi-dimensional arrays also allowed (covered later)

```
type T_4X4 is array (3 downto 0) of std_logic_vector(3 downto 0);
signal CURRENT_TIME, ALARM_TIME : T_4X4;
```

```
-- element assignments
ALARM_TIME(3) <= "0001";
ALARM_TIME(0) <= (others => '0');

-- array assigned with aggregate
CURRENT_TIME <=
("0001", "0010", "0000", "0011");
ALARM_TIME <= (others => "0000");
```

```
CURRENT_TIME <= ALARM_TIME;
```



An array of arrays synthesizes  
to a bundle of buses



55 © Cadence Design Systems, Inc. All rights reserved.

Multi-dimensional arrays can also be declared. The simplest form is where an element of an array is itself an array object. This gives you “nested” arrays or “array-of-arrays,” as they are commonly called. These are very useful for creating data structures. In the example on the slide, we created a 4-element array of 4-bit vectors to represent a 4-digit clock display. We can declare signals of this type and assign them to individual elements of these signals or assign all elements with the aggregate construct.

Array-of-arrays are synthesizable to a collection (or bundle) of individual buses.

We can also declare true 2-dimensional arrays (or larger), although these are not synthesizable and less useful.

The Advanced Data Types module covers complex arrays in much more detail.

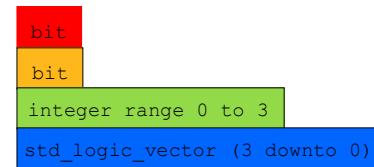
## Records

```

type T_PACKET is record
    BYTE_ID : std_logic;
    PARITY   : std_logic;
    ADDRESS  : integer range 0 to 3;
    DATA     : std_logic_vector(3 downto 0);
end record;
signal TX_DATA, RX_DATA : T_PACKET;
...
RX_DATA <= TX_DATA;
Dot notation
RX_DATA.ADDRESS <= 3;

-- positional aggregate assignment
TX_DATA <= ('1', '0', 2, "0101");
-- named aggregate assignment
RX_DATA <= ( BYTE_ID => '1',
              PARITY  => '0',
              ADDRESS => 1,
              DATA    => "0100");

```



Collection of signals of different types

- Elements explicitly named

Assignment with aggregate

- Named or positional

Individual elements accessed with "dot" notation

Records can be nested



Records are  
synthesizable

Synthesis



56 © Cadence Design Systems, Inc. All rights reserved.

While an array is a structure where each element is of the same type, VHDL allows us to use records, where each element can be different. (Arrays and records are collectively referred to as "composite types" because they are composed of several individual elements.)

In records, we cannot use the index number to identify the individual elements of the record. We need to declare unique names for the elements or fields within the record and define the type for each field because they can be different.

In the example on the slide, we are using a record to describe the contents of a data packet because we can be more precise and specific about the information in the different fields of the packet using a record.

Records can be nested, where an element of a record is itself a record type.

We can assign records using aggregate assignments or assign individual elements of the record using the `record_signal.field_name` syntax.

Synthesis tool usually supports the synthesis of records if every element type of a record is synthesizable (e.g., a record containing an element of type `real` or `time` would not be synthesizable). Not all synthesis tools support nested records.

## Signals and Types: Summary Quiz

1. Are array assignments made by position or by number?
2. What is the difference between `std_logic` and `std_logic_vector`?
3. Which of the following statements are legal VHDL?

```
library IEEE;
use IEEE.std_logic_1164.all;
...
signal C_BUS      : std_logic_vector(0 to 3);
signal Z_BUS      : std_logic_vector(3 downto 0);
signal A_BIT      : std_logic;
signal BYTE       : std_logic_vector(7 downto 0);
type T_INT_ARRAY is array (0 to 3) of integer;
signal INT_ARRAY : T_INT_ARRAY;
...
a -> BYTE      <= (OTHERS => '1');
b -> Z_BUS     <= C_BUS;
c -> Z_BUS     <= ('1', A_BIT, '0');
d -> INT_ARRAY <= "0123";
```

57 © Cadence Design Systems, Inc. All rights reserved.



Let us take a short quiz on this module.

Are array assignments made by position or by number?

What is the difference between `std_logic` and `std_logic_vector`?

Which of the following statements are legal VHDL?

## Signals and Types: Summary Quiz (Solutions)

1. Are array assignments made by position or by number?

- Array assignments are made by position.

Given signal AVEC : std\_logic\_vector(3 downto 0);

signal BVEC : std\_logic\_vector(0 to 3);

BVEC <= AVEC; then the first element of AVEC (3) is assigned to the first element of BVEC (0)

2. What is the difference between std\_logic and std\_logic\_vector?

- std\_logic\_vector is an array of std\_logic, i.e., std\_logic is a single value, whereas std\_logic\_vector is collection of individual std\_logic values.



*This page does not contain notes.*

## Signals and Types: Summary Quiz (Solutions) (continued)

3. Which of the following statements are legal VHDL?

```

library IEEE;
use IEEE.std_logic_1164.all;
...
signal C_BUS      : std_logic_vector(0 to 3);
signal Z_BUS      : std_logic_vector(3 downto 0);
signal A_BIT      : std_logic;
signal BYTE       : std_logic_vector(7 downto 0);
type T_INT_ARRAY is array (0 to 3) of integer;
signal INT_ARRAY : T_INT_ARRAY;
...
BYTE    <= (OTHERS => '1');
Z_BUS   <= C_BUS;
Z_BUS   <= ('1', A_BIT, '0');
INT_ARRAY <= "0123";

```

c. Incorrect; array length mismatch

d. Incorrect-type mismatch

a. Correct; assigns all elements of byte to 1

b. Correct, assignment is made by position and not by index number



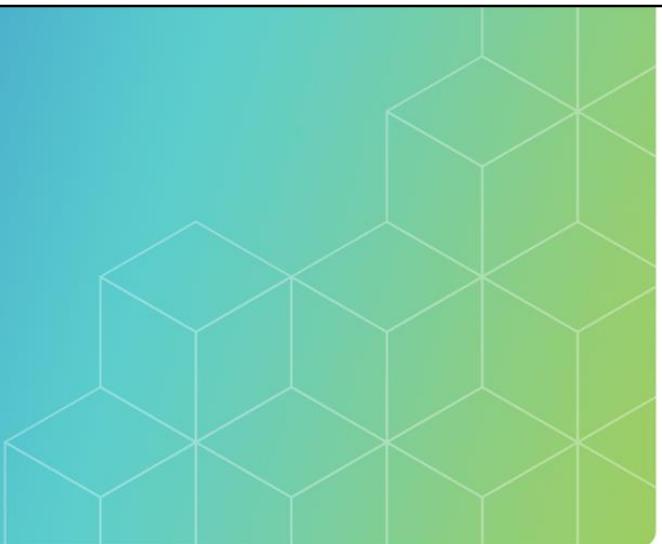
### Solutions (continued)

- Statement a – Correct. Assigns all elements of BYTE to “1”, giving a value of “11111111.”
- Statement b – Correct, but as Z\_BUS and C\_BUS are declared in different directions, and position rather than a number makes an assignment, C\_BUS(0) is assigned to Z\_BUS(3), etc.
- Statement c – Incorrect – array length mismatch. The aggregate on the right-hand side contains three values, but the target Z\_BUS of the assignment is of length 4.
- Statement d – Incorrect – type mismatch. The right-hand side is a string of 4 characters, whereas the target INT\_ARRAY is an array of integers. We would need to use an aggregate to make the assignment correctly.

`INT_ARRAY <= (0,1,2,3);`



## VHDL Operators



**Module**

**5**

Revision

1.0

Version

VHDL 9.0

Estimated time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Module Objectives

In this module, you

- Define and list all the main VHDL operators, such as
  - Logical operators
  - Relational operators
  - Rotate and Shift operators
  - Concatenation operators
- Describe the concept of array slices
- Write and analyze code to perform these operations



In this module, you define and list all the main VHDL operators, such as logical operators, relational operators, rotate and shift operators, and concatenation operators; describe the concept of array slices; and write and analyze code to perform these operations.

## Logical Operators

and, or, nand, nor, xor, xnor (equal precedence)

not (higher priority)

Predefined for

- bit
- bit\_vector
- boolean

Defined in the std\_logic\_1164 package for

- std\_ulogic, std\_logic
- std\_ulogic\_vector, std\_logic\_vector

Shift operators added in VHDL'93

- sll, srl, sla, sra, rol, ror
  - Predefined for bit\_vector only

62 © Cadence Design Systems, Inc. All rights reserved.



First, let's look at the logical operators available in VHDL. These are shown on the slide; and, or, nand, nor, xor, xnor have equal precedence but do not takes higher precedence.

These operators are defined in the standard package for the predefined types bit, bit\_vector, and boolean. They are also defined for the IEEE 1164 types (std\_ulogic, std\_logic and their vector versions) in the std\_logic\_1164 package.

VHDL'93 added some new operators; xnor and six shift operators. The shift operators are logical shift left and right (sll and srl), arithmetic shift left and right (sla and sra), and rotate left and right (rol and ror). These shift operators have higher precedence than logical and relational operators but lower than the not operator. The shift operators are only defined for the built-in type bit\_vector. We will look at performing shift operations with std\_logic later in this module. As we will see, there are simple ways to describe shift and rotate operations using aggregates and bit-slices.

## Logical Operators: Examples

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MVLS is
port (A, B, C : in std_logic;
      OP       : out std_logic);
end MVLS ;
architecture EX of MVLS is
begin

  OP <= A and not(B or C);

end EX;
```

Use parentheses to control how an expression is evaluated and for readability

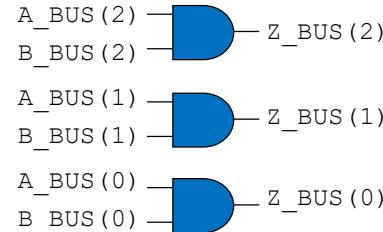


Again, the code is a simple example. We have used brackets here because of the precedence rules. It's a good idea to use parentheses anyway, then you know it's right. Also, it's easier for another engineer to understand who may not be so familiar with VHDL. We know that not has higher precedence over other logical operators. Also, “AND” and OR have equal precedence. Here, the term inside the parenthesis, B or C, is executed first, then AND is performed on this result and A. Supposably, if the parenthesis is removed, then not B is executed first, AND is performed on this result and A as it comes first when we move from left to right. The result of this is ORed with C.

## Logical Operations on Arrays

```
signal A_BUS, B_BUS, Z_BUS : std_logic_vector(2 downto 0);
...
Z_BUS <= A_BUS and B_BUS;
...
```

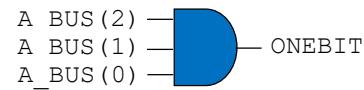
- Operands must be of the same length and type
- Operations carried out on matching elements



VHDL  
08  
Update

Can be used as reduction operators to reduce array to a single value

```
signal A_BUS : std_logic_vector(2 downto 0);
signal ONEBIT : std_logic;
...
ONEBIT <= and A_BUS
```



64 © Cadence Design Systems, Inc. All rights reserved.



We can do logical operations on arrays and signals with single element types like `std_logic`. The rule is that the two array operands need to be of the same length and type, and the result is another array of the same length and type.

When you carry out a logical operation on two arrays, you perform that logical operation on matching elements in the two arrays to produce an element of the result array. You can see this in the example on the slide.

If we changed the declaration of `B_BUS` in the example on the slide so that the range was reversed (0 to 2) instead of (2 downto 0) then the result of the array would be slightly different:

`A_BUS(2) and B_BUS(0)`

`A_BUS(1) and B_BUS(1)`

This is because elements are “matched” by their position in the array (left to right) and not by the index number.

From VHDL2008, the logical operators can be applied to a single vector value. In this situation, the operator acts as a reduction operator, performing the logical operation on every array element to reduce it to a single value.

## Relational Operators

The result is boolean (TRUE or FALSE)

Check the result with the `if` statement to create conditional logic

```
signal BOOLA, BOOLB : boolean;
signal INTA, INTB : integer;
...
BOOLA <= INTA = INTB;
BOOLB <= INTA <= INTB; -- OK!
```

```
if A = B then
  OP <= '1';
else
  OP <= '0';
end if;
```

```
type T_COLOR is (RED, BLUE,
GREEN);
---RED < BLUE < GREEN
---for std_logic '0' < '1' <
'Z'
```

=  
equality

/=  
inequality

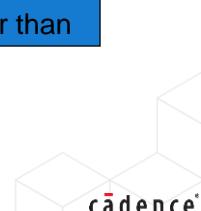
>=  
greater than  
or equal to

<=  
less than  
or equal to

<  
less than

>  
greater than

65 © Cadence Design Systems, Inc. All rights reserved.



VHDL provides a number of relational operators to allow you to compare two values. The result of the operator is of type boolean, which has two values, TRUE and FALSE.

Note the symbol `<=` is used for both signal assignments and as the “less than or equal” operator. The compiler can check the correct use of the symbol from the context.

The most common application of relational operators is in `if` statements. (We'll discuss `if` statements in the Sequential Statements module.)

Relational operators can be used with almost any type, including enumerated and user-defined types. In the case of enumerated types, the rules for applying relational operators come from the declaration of the type; the value declared first in the type declaration (the leftmost) is treated as value 0, the next as value 1, and so on. For example, if we had a type declared as:

```
type T_COLOR is (RED, BLUE, GREEN);
```

then, in a relational operation, red is treated as 0, blue as one, green as 2, and so:

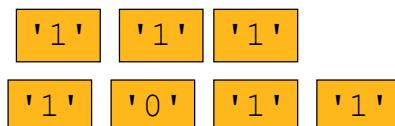
```
RED < BLUE < GREEN
```

so for `std_logic` declared as ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-') ;

```
'0' < '1' < 'Z'
```

Ambiguity can be avoided using equality/inequality instead of less/greater than operators.

## Relational Operations on Arrays



### Arrays

- May be of different lengths
- Aligned left and compared
- Can lead to unusual results!

Arrays have no numerical meaning

- Anyway – are values binary or 2's complement?

```
signal A : std_logic_vector(2 downto 0);
signal B : std_logic_vector(3 downto 0);
...
if A > B then
  ...
end if;
```



All composite types (arrays and records) are compared from left to right, element by element.

Arrays can be of different lengths.

Different length arrays are aligned to the left and compared element by element.

This can lead to results that you wouldn't expect. In the example on the slide, "111" is greater than "1011". Specifically, when one array is smaller than the other, the smaller array is padded to the same size as the larger one using the value nul. The two vectors can then be compared element by element, where nul is always treated as the lowest possible value for comparison.

This is the first example of an important rule in VHDL: By default, an array does not have implied numerical meaning. If you want to use arrays as numbers, you have a problem! VHDL has a way around this which we will discuss in the Arithmetic Operators module.

## Concatenation

```
signal A_BUS, B_BUS, Z_BUS : std_logic_vector(3 downto 0);
signal A_BIT, B_BIT, C_BIT, D_BIT : std_logic;
signal BYTE : std_logic_vector(7 downto 0);
```

```
Z_BUS <= A_BIT & B_BIT & C_BIT & D_BIT;
```

Z\_BUS (3) ← A\_BIT  
Z\_BUS (2) ← B\_BIT  
Z\_BUS (1) ← C\_BIT  
Z\_BUS (0) ← D\_BIT

- Can be used like aggregate to assign array from individual elements
- Can join arrays to create larger arrays
- Normal rules about matching array lengths apply

```
BYTE <= A_BUS & B_BUS;
```



VHDL has a concatenation operator, which allows you to build up arrays. We can concatenate individual elements together to form arrays (similar to an aggregate assignment) or concatenate arrays to form larger arrays.

## Slice of an Array

```
signal Z_BUS, A_BUS : std_logic_vector(3 downto 0);
signal B_BIT : std_logic;
signal BYTE   : std_logic_vector(7 downto 0);
```

```
A_BUS <= BYTE(5 downto 2);
Z_BUS(1 downto 0) <= '0' & B_BIT;
```

BYTE(5 downto 2) <= A_BUS;	<input checked="" type="checkbox"/>
Z_BUS(0 to 1) <= '0' & B_BIT;	<input type="checkbox"/>



The slice direction must be the same as the signal declaration

Beware

- An array slice can be used on either side of an assignment

Slicing allows us to take a “slice” out of a larger array and assign it to a smaller array. The slice can appear on either side of a signal assignment statement, so we can also assign a slice of a larger array with a smaller array. In the example on the slide, we see that array a-bus of 3 down to 0 is assigned with 5 down to 2 bits of the byte. Similarly, an array slice of z-bus of 1 down to 0 is set with the concatenation of 0 and b-bit.

There is a restriction in VHDL in that an array can only be sliced in the same direction as it was declared. Otherwise, it is a compiler error. In this code, z-bus is displayed as three down to zero. Hence we can't take 0 to 1 slice from it as it doesn't comply with the declared index direction.

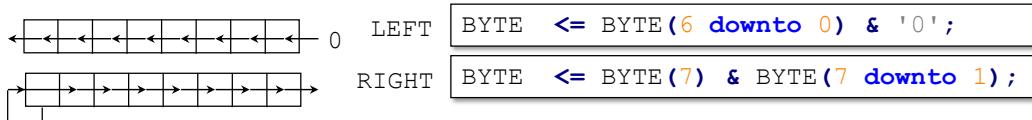
That is, if an array is declared as (3 downto 0), you cannot take a slice (0 to 1) from it.

## Rotate and Shift Basics

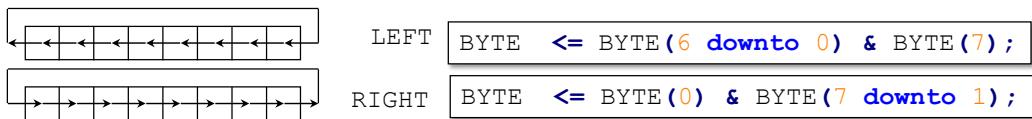
### Logical Shift



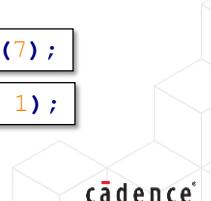
### Arithmetic Shift



### Rotation



69 © Cadence Design Systems, Inc. All rights reserved.



Slices and concatenation can be used to describe rotating and shift operations.

With logical shifts, bits that are shifted outside the vector are lost, and '0' values are added to maintain vector length.

For example, the logical right shift of BYTE by 2 bits ('0's passed in at MSB, and 2 LSB's lost).

```
BYTE <= "00" & BYTE(7 downto 2);
```

Arithmetic left shifts are identical to logic left shifts. However, an arithmetic right shift maintains the current value of the MSB. This maintains the sign of the vector (in 2's complement, a negative value has a '1' in the MSB).

For example, the arithmetic right shift of BYTE by 2 bits (value of MSB maintained).

```
BYTE <= BYTE(7) & BYTE(7) & BYTE(7 downto 2);
```

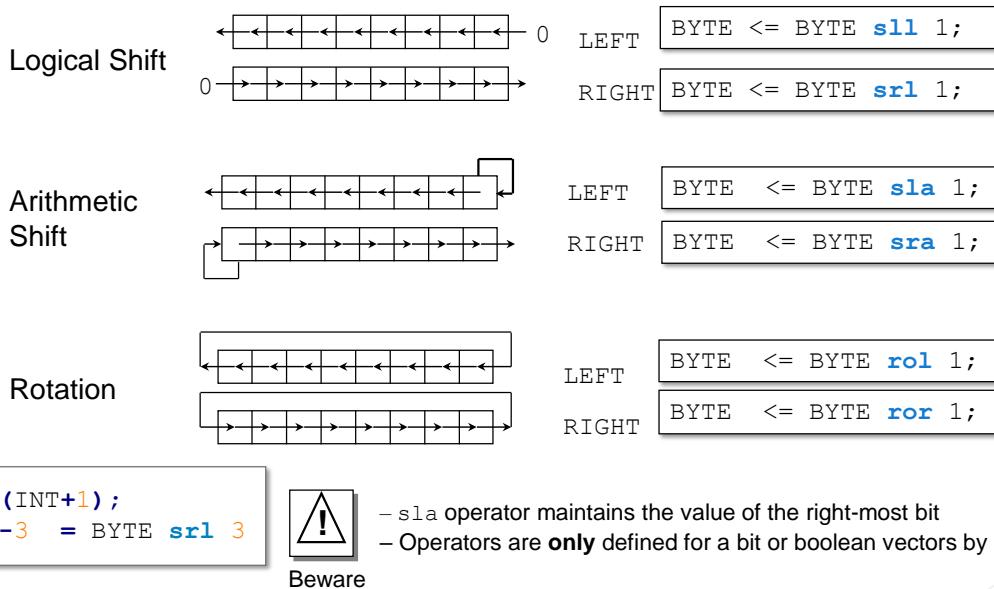
Rotates circulate data within the vector but do not lose bits – the original vector value can be recovered.

For example, left rotation of BYTE by 2 bits.

```
BYTE <= BYTE(5 downto 0) & BYTE(7 downto 6);
```

Shifts can be used for simple multiplication and division by powers of 2. A logical shift right 2 bits is equivalent to a divide by 4; a logical shift left 1 bit is multiplied by 2. Arithmetic shifts are used for signed data.

## Rotate and Shift Operators



70 © Cadence Design Systems, Inc. All rights reserved.



VHDL'93 defines operators for rotating and shift operations. These operators are only defined for arrays of bit (bit\_vector) or boolean by default. This restricts the use of these operators, particularly since shifts and rotates can be easily implemented with bit slices and concatenation.

Note that the shift left arithmetic operator (**sla**) maintains the value of the right-most bit. This is for correct operation in designs using “little-endian” data (bit 0 is the MSB) or arrays using the “to” index direction instead of “downto”.

The second operand must be of type integer, so expressions are allowed, for example:

```
BYTE ror (INT+1)
```

Negative values are also allowed, which shift in the opposite direction:

```
BYTE sll -3 = BYTE srl 3
```

Arithmetic packages, e.g., `numeric_std`, define these shift operators for array types `unsigned` and `signed`. This allows us to use these operators on `std_logic_vector` objects using closely-related type conversions – see the Arithmetic Operators module for more information.

## Operators: Summary Quiz

1. Which statements are correct, VHDL?

```
1 architecture DODGEY of OPERATORS is
2   signal BOOL           : boolean;
3   signal A_INT, B_INT    : integer range 0 to 15;
4   signal Z_BIT          : std_logic;
5   signal A_VEC, B_VEC, Z_VEC : std_logic_vector (3 downto 0);
6   begin
7
8     Z_BIT <= A_INT = B_INT;
9     BOOL  <= A_INT > B_VEC;
10    Z_VEC <= A_VEC & B_VEC;
11    Z_VEC <= A_VEC(0 to 1) & A_VEC(1 downto 0);
12    Z_VEC <= A_VEC(1 downto 0) & B_VEC(1 downto 0);
13
14 end DODGEY;
```

71 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Operators: Summary Quiz (Solution)

1. Which statements are correct, VHDL?

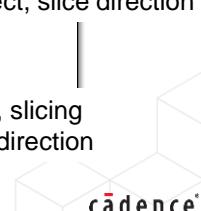
```

1  architecture DODGEY of OPERATORS is
2      signal BOOL           : boolean;
3      signal A_INT, B_INT   : integer range 0 to 15;
4      signal Z_BIT          : std_logic;
5      signal A_VEC, B_VEC, Z_VEC : std_logic_vector (3 downto 0);
6      begin
7          8: Incorrect, wrong return type
8              Z_BIT <= A_INT = B_INT;
9              BOOL <= A_INT > B_VEC;
10             Z_VEC <= A_VEC & B_VEC;
11             Z_VEC <= A_VEC(0 to 1) & A_VEC(1 downto 0);
12             Z_VEC <= A_VEC(1 downto 0) & B_VEC(1 downto 0);
13
14 end DODGEY;

```

8: Incorrect, wrong return type  
 9: Incorrect, type mismatch  
 10: Incorrect, type mismatch  
 11: Incorrect, slice direction  
 12: Correct, slicing in right direction

72 © Cadence Design Systems, Inc. All rights reserved.



Here is the solution.

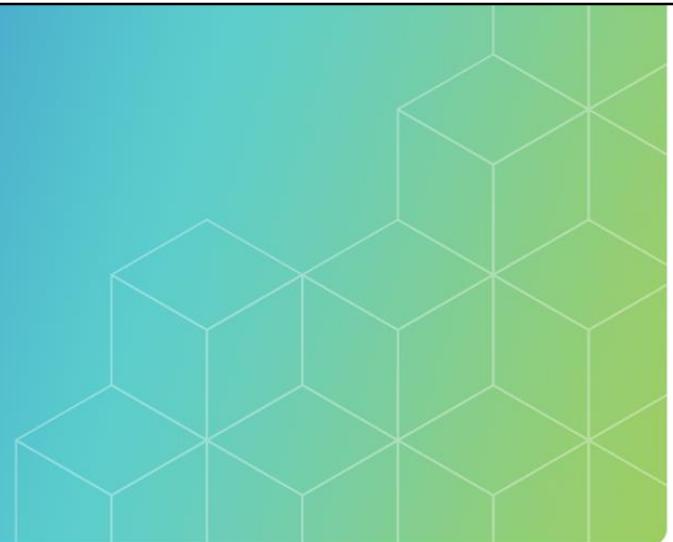
Line 8 – Incorrect due to the wrong return type. We know that relational operators return a boolean value, whereas here, rhs z-bit is of std\_logic type, which is not allowed.

Line 9 – Incorrect due to a type mismatch. The relation operator is more significant than is not defined by default to compare an integer (A\_INT) with a vector (B\_VEC). This is a limitation, and in the Arithmetic Operators module, we see how it is possible to compare an integer and a vector, using the numeric\_std package and defining the vector as either signed or unsigned.

Line 10 – Incorrect – array length mismatch. The concatenation gives a vector of length eight which cannot be assigned to the target, Z\_VEC, of length 4

Line 11 – Incorrect – slice direction error. A\_VEC can only be sliced in the direction in which it is declared. As A\_VEC was declared in the `downto` direction, the `to` slice is illegal.

Line 12 – Correct. The slicing of A\_VEC and B\_VEC is in the correct direction. The concatenation gives us a std\_logic\_vector of length four, and this matches the type and length of the target Z\_VEC.



## Sequential Statements

**Module** **6**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

We have looked at some of the operators available in VHDL, which allow us to do some useful work with signal values. We are still limited in capturing our design in terms of logic functions and comparators with what we know.

This section will look at some sequential statements in VHDL that allow us to describe pure behavior. VHDL will enable you to describe behavior using concurrent or sequential statements. We will look at sequential style in this section.

## Module Objectives

In this module, you

- Define and describe the working of the process and list the types of process
  - Process with a sensitivity list
  - Process without sensitivity list
  - Process all
- Code and analyze examples using sequential statements like
  - if then else
  - case
- List the rules associating case statements
- Describe vector case expressions



In this module, you will be able to define and describe the working of processes and list the types of processes like process with sensitivity list, process without sensitivity list, and process all. You will be able to code and analyze examples using sequential statements like if then else and case statements, list the rules associating case statements, and describe vector case expressions.

## Process

More complex design functionality requires sequential statements

- For example, conditional `if` statement

Sequential statements must be placed in a process

Processes are defined in architecture, where they:

- Read input ports and local signals
- Write output ports and local signals

Process label is optional

Multiple processes interact concurrently.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX is
    port (A, B, SEL : in std_logic;
          OP        : out std_logic);
end MUX;

architecture BEHAVE of MUX is
begin

    MUXPROCESS: process (A, B, SEL)
    begin

        if (SEL = '1') then
            OP <= A;
        else
            OP <= B;
        end if;
    end process MUXPROCESS;

end BEHAVE;
```

Order of execution

Sequential statements give VHDL a lot of power when describing complex behavior.

When we defined the behavior of our half-adder, we used concurrent VHDL statements.

We could have described the behavior using a VHDL process, such as the example on the slide. We can use VHDL sequential statements inside a process, which work like conventional software. Inside a process, the statements are executed line-by-line, so the order of statements is important. VHDL has a variety of sequential statements, many of which are very similar to statements in conventional software languages, such as the `if` statement in the example.

Processes are placed only inside an architecture. They read input and/or local signals and update the output and/or local signals. In the example shown, the MUX-process is used to read signals `a`, `b`, and `SEL` and update the `OP` value.

The process label is optional. It makes code more readable when we have multiple processes. Single architecture can have many processes that execute concurrently and communicate simultaneously using signals.

When we defined the behavior of our half-adder, we used concurrent VHDL statements.

We could have described the behavior using a VHDL process, such as the example on the slide. We can use VHDL sequential statements inside a process, which work like conventional software. Inside a process, the statements are executed line-by-line, so the order of statements is important. VHDL has a variety of sequential statements, many of which are very similar to statements in conventional software languages, such as the `if` statement in the example. Sequential statements give VHDL a lot of power when describing complex behavior.

VHDL processes can only be used inside an architecture. Single architecture can have many functions executing concurrently and communicating simultaneously using signals.

## Process Sensitivity List

Sensitivity list

```
MUX:
process (A, B, SEL)
begin
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process MUX;
```

The process is executed when any signal in the sensitivity list changes the value

- Signal event

Output signals are only updated when the process suspends

Any signal can be on the list

- No simulator compilation checks
- Most synthesis tools checklists



What would the behavior be if SEL was missing from the sensitivity list?

Question



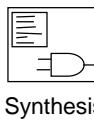
Remember, a process is a section of code in which statements execute sequentially, line-by-line, like conventional software.

Notice the brackets after the keyword process. This is a list of signals known as the sensitivity list; whenever any of these signals changes value (an event in VHDL terminology), the process will run, executing the statements line-by-line until the end process is reached. When this happens, the process suspends, and any signals assigned in the process are updated with their new values. It is important to understand that signals are only updated when the process suspends, as we will see later.

Any signals can be in the sensitivity list, but only the signals whose values are read in the process need to be in the sensitivity list. If the signal SEL is not included in the sensitivity list, the process will not execute when SEL changes. Such a process will compile and simulate successfully.

If you synthesize the process with SEL missing from the sensitivity list, you should get errors from your synthesis tool. However, some synthesis tools may only generate warnings and automatically generate a complete sensitivity list. It is possible to miss the warnings and have different simulation behavior between RTL and gate-level designs. The IEEE standard for synthesizable VHDL clearly states that the sensitivity list for a combinational process must be complete.

## Incomplete Sensitivity List

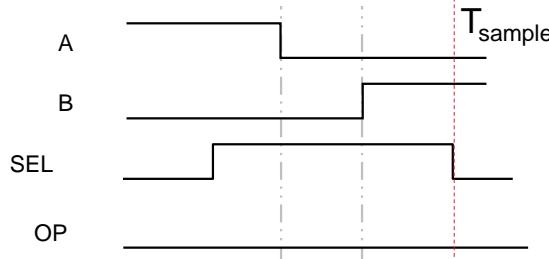


For combinational logic,  
sensitivity list must contain  
all signals read in the process

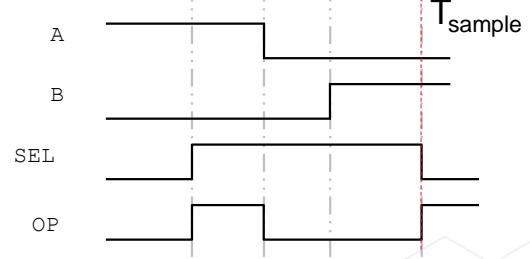
Synthesis

```
MUX: process (A, B, SEL)
begin
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process MUX;
```

```
MUX: process (A, B)
...
```



```
MUX: process (A, B, SEL)
...
```



77 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

To examine the effects of an incomplete list, consider the process MUX.

If SEL is missing, the assignment to OP is only made on an event (change of value) on A or B. Changes in the value of SEL do not trigger the process and so do not immediately update the output.

Synthesis tools should require a complete sensitivity list to synthesize combinational logic.

Incomplete sensitivity lists can be difficult to debug in simulation: outputs can lag the expected values in some circumstances. In this slide, we see that, in the waveform on the left, OP is evaluated only when A or B changes, changes on SEL don't trigger the process. In the waveform on the right, which includes SEL in the process sensitivity list, the OP value is evaluated whenever A, B, or SEL changes, as indicated by the vertical lines.

At the time T sample, the SEL value changes from 1 to 0. This should trigger the process, and OP should be assigned with B, which is one at this instant. This happens in the waveform shown on the right as SEL is contained in the sensitivity list. In contrast, in the waveform on the left, at T sample OP= 0, as the process is not triggered due to the incomplete sensitivity list, the value is not updated with B.

## process (all)

- 2008 update
- Used in combinational processes
- Implied process sensitivity list with all inputs
- Safe to use even with procedures
- Performance wise no much change; it takes negligible time by the compiler to determine the sensitivity list
- Avoids any mistakes of missing signals, renamed or added signals

```
MUX: process (all)
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process MUX;
-- compiler expands process (all) to
-- process(A, B, SEL)
-- as A, B, SEL are inputs
```



VHDL-2008 introduces `process (all)` for combinational processes. All is equivalent to a sensitivity list with all inputs to the process. It can be used even when the process contains procedures that work on signals. Performance-wise, there is not much change as the compiler takes no extra time to figure out the implied sensitivity list. Process all gives the advantage of avoiding the mistakes that may occur due to accidental missing of any signal listing, renaming, or any addition of a new signal.

In the code on the slide, `process (all)` implies `process (A, B, SEL)` as A, B and SEL are the inputs read by the process MUX.

## Processes and Signals

Architectures can contain multiple processes

- Connected by signals
- Every signal must have a type

Processes execute statements in sequence, like "software"

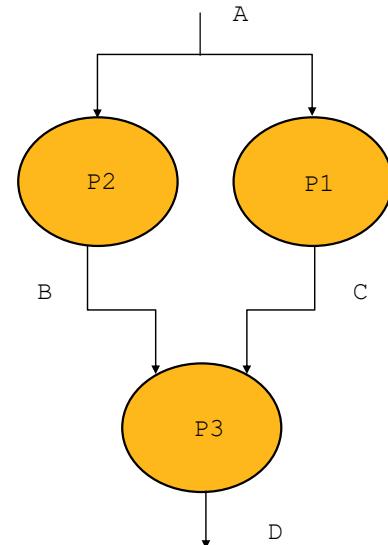
- Serial hardware

Processes with the same sensitivity list are executed simultaneously

- Concurrent hardware

An output of one process may be the input of another

- Serial (sequential) hardware



79 © Cadence Design Systems, Inc. All rights reserved.

This capability to have many processes communicating concurrently with each other via signals is the basic model which VHDL uses to describe hardware. The processes may be in a single architecture or partitioned into a number of separate entities/architectures in a hierarchical structure, but the basic model remains the same.

Signals handle communication between processes, and in VHDL, each signal must have a type that defines a set of values the signal is allowed to carry. VHDL provides several built-in types, such as `bit`, `integer`, `real` and `bit_vector`, similar to a bus in schematic design (a number of wires grouped together). VHDL also allows you to define your own types, making your code more readable and providing better error checking because you can be very precise about the information on a signal.

Type checking is very strict in VHDL; you cannot connect two signals of different types. VHDL forces you to be precise and consistent.

## if Statement Syntax

```

if CONDITION then
    -- sequential statements
end if;

if CONDITION then
    -- sequential statements
else
    -- sequential statements
end if;

if CONDITION1 then
    -- sequential statements
elsif CONDITION2 then
    -- sequential statements
elsif CONDITION3 then
    -- sequential statements
else
    -- sequential statements
end if; -- condition1,2,3
--can overlap

```

- Standalone **if** executes sequential statements only if the condition is true
- .. add unconditional **else** to execute other sequential statements when **if** condition false
- ..add **elsif** to check further conditions
  - Conditions prioritized in order of appearance
  - Final **else** is optional
- Priority encoder behavior

80 © Cadence Design Systems, Inc. All rights reserved.



So having looked at the theory behind concurrent and sequential statements, we will spend the rest of this section looking at three different examples of sequential statements: **if**, **case**, and the **for loop**.

The syntax for the **if** statement is shown on the slide. At the heart of the **if** statement is the condition, the test we do to decide which branch of the **if** statement to execute. The condition must be an expression that delivers a boolean result, typically from one of the relational operators we have seen.

VHDL allows a lot of flexibility in the syntax for the **if** statement; the **else** clause is optional. An unconditional **else** statement is added to handle scenarios when **if** condition fails. Also, we can have any number of **elsif** branches to check further conditions. Each **elsif** branch will have a condition that can overlap with the conditions in other **elsif** branches or the condition in the first **if** branch. Overlapping conditions are allowed because the **if** and **elsif** conditions are evaluated in order; the first branch with a true condition is the one that is executed, and the **if** statement then completes. Hence, it acts as a priority encoder.

Each branch in the **if** statement can contain any number of sequential statements, including other “nested” **if** statements, **case**, etc.

## if Statement Example

```

library IEEE;
use IEEE.std_logic_1164.all;

entity IF_EXAMPLE is
    port (A,B,C,S : in std_logic_vector(3 downto 0);
          OP       : out std_logic_vector(3 downto 0));
end IF_EXAMPLE;
architecture RTL of IF_EXAMPLE is
begin
    IFEX: process (all)
    begin
        if (S = "0000") then      -- 1
            OP <= A;
        elsif (S <= "0101") then -- 2
            OP <= B;
        else
            OP <= C;
        end if;
    end process IFEX;
end RTL;

```

- if is a conditional statement
- if conditions are tested in order of appearance
  - Condition is a boolean expression
- if conditions can overlap
- Statement(s) of first true condition executed

```

if (S <= "0101") then -- 2
    OP <= A;
elsif (S = "0000") then -- 1
    OP <= B;
...

```



What would be the effect of swapping conditions 1 and 2?

81 © Cadence Design Systems, Inc. All rights reserved.



The slide shows an example of an if statement with an else clause and a single elsif branch. Conditions in if, elsif can overlap. Note here that the conditions in the if and elsif branches overlap when S="0000", then the first and second conditions hold good. But the first one gets priority because it was the first branch defined in the if statement. Hence, statements in the if block is executed sequentially.

This behavior will explain the consequences of swapping the two conditions over:

```

if (S <= "0101") then      -- 2
    OP <= A;
elsif (S = "0000") then   -- 1
    OP <= B;
else
    OP <= C;
end if;

```

Now when S = "0000", the first condition (2) is true, and so OP is A. Because we have swapped the conditions, and condition 1 overlaps with condition 2, condition 1 can never be true, so OP can never be B.

## Structuring if Statements

```

if (ENABLE = '1') and (ALARM = '1') then
  -- ENABLE = '1', ALARM = '1'
else
  -- ENABLE = '1', ALARM = '0' or
  -- ENABLE = '0', ALARM = '1' or
  -- ENABLE = '0', ALARM = '0'
end if;
    -- same condition
    if (ENABLE and ALARM) = '1' then
      ...
if ENABLE = '1' then
  if ALARM = '1' then
    -- ENABLE = '1', ALARM = '1'
  else
    -- ENABLE = '1', ALARM = '0'
  end if;
else
  -- ENABLE = '0'
end if;
    if ENABLE = '1' then
      -- ENABLE = '1'
    elsif ALARM = '1' then
      -- ENABLE = '0', ALARM = '1'
    else
      -- ENABLE = '0', ALARM = '0'
    end if;
  
```

- if condition can contain multiple checks
  - Any boolean expression can be used
  - Typically, relational operators are linked by logical operators
- if statements can also be nested
- Think carefully about which conditions must be checked

--Bad Code  
if ENABLE = '1' and ALARM = '1' then  
 ...

-- vhdl2008  
if ENABLE and ALARM then  
 ...

VHDL 08      std\_logic.'1' = true

Update



82 © Cadence Design Systems, Inc. All rights reserved.

Any boolean expression can be used for the condition of an if statement. This expression can contain any number of separate checks, as long as the resulting value is boolean. We use relational operators to compare an object to a value and logical operators to link several checks into a single boolean expression. In the example on the slide, the equality relational operator checks if Enable and Alarm equal one; if so result is true, else false. Logical and is performed over these two boolean results to produce a single boolean value. Now, if this is true, if branch is executed, else branch is executed.

Brackets are important in boolean expressions. Without brackets, expressions are evaluated left to right in order of declaration. If you wrote:

```
if ENABLE = '1' and ALARM = '1' then it is a bad code.
```

The compiler would compare ENABLE to '1', then try to and the boolean result with ALARM which would fail due to type mismatching.

If statements can also be nested. The slide shows various examples of if statements and how they can be written.

In VHDL2008, the requirement to add the value check for every std\_logic signal is removed. In VHDL2008 *only*, the std\_logic value '1' is interpreted as a boolean TRUE. This means instead of writing:

```
if (ENABLE = '1') and (ALARM = '1') then
```

In VHDL2008, we can write:

```
if ENABLE and ALARM, then
```

The programmer must carefully choose the condition to be checked depending on the design requirement.

## case Statement Syntax

```

case expression is
  when VALUE_1 =>
    -- single value
    <sequential statements>
  when VALUE_2 | VALUE_4 =>
    -- list of values
    <sequential statements>
  when VALUE_M to VALUE_N =>
    -- range of values
    <sequential statements>
  when others =>
    -- remaining values
    <sequential statements>
end case;

```

case is a multi-way conditional statement

- Case expression evaluated
- Statement(s) of matching item executed

Optional others capture unspecified



Now let's look at the case statement. Suppose you have not come across this construct in other languages. In that case, the idea is that we evaluate an expression and select one of a number of sequences of statements depending on the expression's value.

We can check for explicit case expression (S) values as in the first case branch.

We can also test a number of discontinuous values in a single when branch using the | (or) option or by specifying range with the “to” operator. We can also cover any values that have not been specifically tested using the when others statement.

## case Statement Example

```
entity CASE_EXAMPLE is
port (A, B, C, S : in std_logic_vector(2 downto 0);
      OP        : out std_logic_vector(2 downto 0));
end CASE_EXAMPLE;
architecture RTL of CASE_EXAMPLE is
begin
  CASEG: process (all)
  begin
    case S is
      when "001" =>
        OP <= A;
      when "100" =>
        OP <= C;
      when "110" | "011" =>
        OP <= A;
      when others =>
        OP <= "0000";
    end case;
  end process CASEG;
end RTL;
```

No item values may overlap

All possible expression values must be covered

- Specifically, or with `others` item

Compiler error if rules violated

84 © Cadence Design Systems, Inc. All rights reserved.



The slide shows an example case statement. It is placed inside the process CASEG. There are two important rules for the case statement:

- The values are selected by `when` branches cannot overlap (in technical terms, they must be “disjoint”).
- All possible values of the expression must be specified either explicitly or with the `others` statement.

Hence, the `case` statement differs from the `if-then-elsif` with respect to both of these rules. In this example, when  $S = 0\ 0\ 1$ ,  $OP$  is assigned with  $A$ , when  $S = 1\ 0\ 0$ ,  $OP$  is assigned with  $C$ , and when  $S$  is either  $1\ 1\ 0$  or  $0\ 1\ 1$ ,  $OP$  is assigned with  $A$ , in all other cases,  $OP$  is assigned with  $0000$ .

## Defining case Ranges

```
entity INTEGER_CASE is
  port ( A, B, C, SINT : in integer range 0 to 15;
        OP           : out integer range 0 to 15);
end INTEGER_CASE;

architecture RTL of INTEGER_CASE is
begin
  CASEINT: process (all)
  begin
    case SINT is
      when 0 to 4 =>
        OP <= B;
      when 5 =>
        OP <= C;
      when others =>
        OP <= A;
    end case;
  end process CASEINT;
end RTL;
```

- Case ranges can only be used with discrete types
  - For example, integer



Arrays do not have a discrete sequence of values associated with them  
Beware

<pre>case S_VECTOR is   when "0101" =&gt; <input checked="" type="checkbox"/>   ...   when "0000" to "0100" =&gt; <input type="checkbox"/>   ...   when others =&gt; <input checked="" type="checkbox"/> end case;</pre>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/>
--	--

85 © Cadence Design Systems, Inc. All rights reserved.



It is possible to define a range of values using the “x to y” syntax with a case statement. This will work with integers and enumerated types but not with an array type. In this example, we see that SINT is declared as an integer and can be tested for a range of values like 0 to 4 in the case statement.

Ranges can't be defined for array types. This is because an array is just a collection of elements of the same type. We have seen that it has no numeric value associated with it and, therefore, no sequence that would allow us to define a range of values.

## VHDL2008: Matching Case

New "matching case" construct

- case? ... end case?;

Allows std\_logic.'-' as wildcard value in case choice expressions

Restrictions

- Only allowed in choice expression
- No overlapping between values
- bit, boolean, std\_logic types or vectors only

VHDL  
08

Matching case construct a  
case? ... end case?

Update

```
vhdl2002
case? IP is
when "100" | "101" | "110" | "111" => OP <= "11";
when "010" | "011" => OP <= "10";
when "001" | "000" => OP <= "01";
when others => OP <= "00";
end case;
```

```
vhdl2008
case? IP is
when "1--" => OP <= "11";
when "01--" => OP <= "10";
when "00--" => OP <= "01";
when others => OP <= "00";
end case?;
```

86 © Cadence Design Systems, Inc. All rights reserved.



In a VHDL2008 case, the statement can use std\_logic don't care values (--) in case choice expressions as a true wildcard, matching any other std\_logic value. This can greatly simplify case statements, particularly for priority encoders by removing the requirement for the case choice expression to explicitly list every value for a branch.

However, the standard VHDL case restrictions still apply, specifically that there can be no overlapping values between case choices and that every value of the case-select expression should be covered in the case choices.

In this example, we see that instead of listing four IP values, 1 0 0, 1 0 1, 1 1 0, and 1 1 1 case, we can write 1-- in the case statement – both checks for the same condition. When we expand 1--, we get these four values.

## Vector Case Expressions

```
signal AVEC : std_logic_vector(1 downto 0);
signal OP   : std_logic_vector(2 downto 0);
```

```
case AVEC is
  when "00" => OP <= "000";
  when "01" => OP <= "001";
  when "10" => OP <= "010";
  when "11" => OP <= "100";
end case;
```

Compilation error

```
case AVEC is
  when "00" => OP <= "000";
  when "01" => OP <= "001";
  when "10" => OP <= "010";
  when "11" => OP <= "100";
  when others => null; 
end case;
```

- std\_logic has 9 values
  - Two element vector has 81 possible value combinations
- Covering only binary values in case branches will cause compile errors
  - For example, what about "UU"?
- when others needed
- null can be used to indicate nothing to be done



null is supported  
by synthesis tools

Synthesis



Case rules state that all values of a case expression must have one branch (and one branch only) in the case body.

When using std\_logic vector types in a case expression, remember that there are other values besides '1' and '0'. A simple 2-element vector has 81 possible values, and you must cover all of these in the case body. This is usually done by specifying functional behavior for all binary ('1' and '0') combinations and using the when others clause to deal with all other cases.

It is unlikely we would define any functional behavior for the non-binary combinations. Hence, one way to tell the simulator that we don't wish to do anything for all the undefined values covered by the others branch is to use a null statement. Null statements are supported by synthesis. We can even attach a null statement to a case branch for a binary vector value to indicate that we don't wish to do anything in the case statement for that particular value. However, this can cause synthesis problems with incomplete assignments. (See the Definition of RTL Code module for more details.)

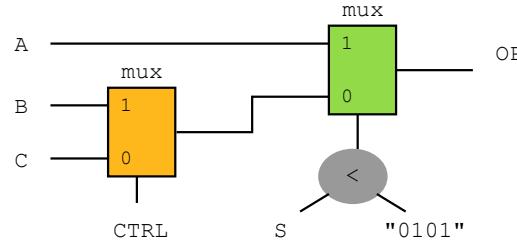
Note that you cannot use the “don't care” std\_logic value '-' as a wildcard comparison character in case or if conditions. Compilers will only allow a '-' value to be matched to another '-' value, not a '1' or '0', for example.

In the above code, signal AVEC is declared as a standard logic vector; hence, every element can take 9 values. In the first case statement, we check for a subset of values it can take; the simulator will not know what to do if AVEC takes other values like X, Z, etc. Hence, it leads to compiler errors. This is handled in the second case statement by specifying every other value under others and doing nothing by using the null keyword.

## Sequential Statements: Summary Quiz

1. What are the two rules regarding the `case` statement?
2. If more than one condition of an `if-elsif` statement is true, which condition takes priority?
3. Write a process for the following combinational logic:

Hint – Use a single `if` statement



*This page does not contain notes.*

## Sequential Statements: Summary Quiz (Solutions)

- What are the two rules regarding the `case` statement?

There are two important rules for the `case` statement:

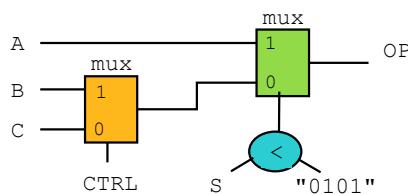
- The values selected by `when` branches cannot overlap (in technical terms, they must be "disjoint").
- All possible values of the expression must be specified either explicitly or with the `others` statement.

- If more than one condition of an `if-elsif` statement is true, which condition takes priority?

- Overlapping conditions are allowed because the `if` and `elsif` conditions are evaluated in order; the first branch with a true condition is the one that is executed.

- Write a process for the following combinational logic:

Hint – Use a single `if` statement



```
process comb(all)
test :begin
    if ( S < "0101" )           -- condition1
        OP <= A;
    elsif ( CTRL == '0' )         -- condition2
        OP <= C;
    elsif ( CTRL == '1' )         -- condition3
        OP <= B;
    endprocess : test
```

89 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Let's look at the solutions. There are two important rules for the `case` statement:

- The values selected by `when` branches cannot overlap.
- All possible values of the expression must be specified either explicitly or with the `others` statement.

If more than one condition in `if-elsif` holds true, then the first branch with a true condition is executed first.

The process code for the circuit is as shown. The circuit on the slide contains two MUXes, with CTRL and output of a comparator acting as select lines. The output of the comparator is 1 if S is less than 0 1 0 1; otherwise, it is 0. Since two MUXes are cascaded, it implies priority logic. Since the output of the comparator is the select line of the high priority MUX, the one on the right, if S less than 0 1 0 1 condition is satisfied, then irrespective of what value control takes, output is assigned to A. If S less than 0 1 0 1 fails, we look at low priority MUX to which control is the select line. In this case, if control is 0, the output of this MUX is C. If control is one, the output of this MUX is B. The output of this MUX is, in turn, assigned to the OP by high priority MUX when S equals 0 1 0 1 fails.

## Quiz (to prepare for tomorrow!)

```
entity ALL_WRONG is
port (A, B : in integer range 0 to 3;
      Z : out std_logic_vector(3 downto 0));
end ALL_WRONG;

architecture ALL_WRONG of A is
  BAD_ONE : process (A,B,X)
begin;
  Y <= "0000";
  Z <= "0000";
  if (A = "00") then
    Z <= "0011";
  elseif (A = 2) then
    Z <= "0111";
  else
    case B is
      when 0 =>
        Z <= "0000";
      when 0 to 2 =>
        Z <= "1111";
    end case;
  endif;
  end BAD_ONE;
end A;
```

Find at least twelve things wrong!



*This page does not contain notes.*

## Solutions: Quiz (to prepare for tomorrow!)

```

----1 library and use missing
entity ALL_WRONG is
    port (A, B : in integer range 0 to 3;
          Z : out std_logic_vector(3 downto 0));
end ALL_WRONG;

architecture ALL_WRONG of A is ----2,3
  BAD_ONE : process (A,B,X)-----4,5
  begin;-----6
    Y <= "0000";-----7
    Z <= "0000";
    if (A = "000") then -----8
      Z <= "0011";
    elsif (A = 2) then-----9
      Z <= "0111";
    else
      case B is
        when 0 => -----10
          Z <= "0000";
        when 0 to 2 => -----10
          Z <= "1111";
      end case; -----11
    endif;
  end process BAD_ONE; -----12
end A;

```

91 © Cadence Design Systems, Inc. All rights reserved.



Here is the solution:

1. library IEEE; and use IEEE.std\_logic\_1164.all; lines required to access std\_logic\_vector type. This could be counted as two errors.
2. Architecture and entity names were swapped over in architecture declarations. Should be architecture A of ALL\_WRONG;
3. Architecture is missing begin before process declaration.
4. Comma missing between A and B in the process sensitivity list.
5. Undeclared identifier X is listed in the process sensitivity list.
6. Incorrect semi-colon at the end of begin.
7. Undeclared identifier Y is the target of an assignment.
8. Type mismatch in equivalence operator comparing A (integer type) to literal "00" of unknown type (std\_logic\_vector? bit\_vector? string?). It would be helpful to mix types in operators. The Arithmetic Operators module will describe how, but with the current code, this operator use is illegal.
9. Keyword elseif should be elsif.
10. Value 0 is included in both branches of the case statement.
11. Value 3 is omitted from all branches of the case statement.
12. Keyword process missing from end BAD\_ONE;

## Labs

- Lab 6-1 Familiarization with the Multiplexer Design
- Lab 6-2 Familiarization with Your VHDL Simulator
- Lab 6-3 Familiarization with Your Synthesis Tool
- Lab 6-4 Expanding the Multiplexer
- Lab 6-5 Adding the Alarm Signal



You will now have the opportunity to perform some self-paced labs to reinforce the ideas presented in this module.

## Concurrent and Sequential Statements

**Module**

**7**

Revision

**1.0**

Version

**VHDL 9.0**

Estimated Time:

- Lecture
- Lab

**cadence®**

In this section, we will introduce some of the most commonly used concurrent and sequential statements.

## Module Objectives

In this module, you

- Compare and contrast concurrent and sequential statements
- Differentiate between multiple concurrent and sequential assignments
- Define and describe conditional signal assignments
- Describe selected signal assignments
- Analyze and write code using these statements



In this module, you will be able to compare and contrast concurrent and sequential statements, differentiate between multiple concurrent and sequential assignments, define and describe conditional signal assignments, describe selected signal assignments, and analyze and write code using these statements.

## Concurrent and Sequential Statements

```
architecture DESIGN of E is
  -- signal declarations
begin
```

```
  P1 : process
  begin
    -- sequential statements
  end process P1;
```

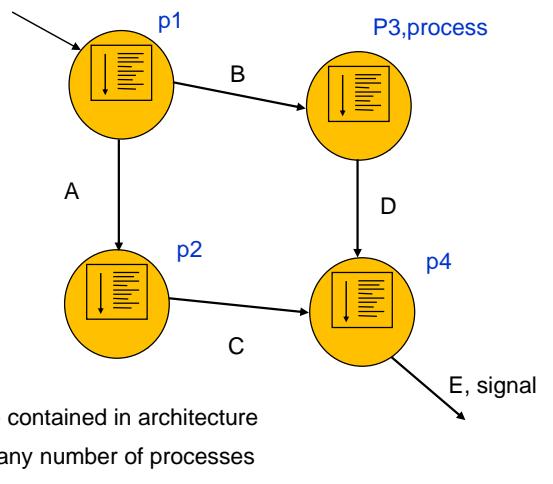
```
-- concurrent statements
```

```
  MUX : process (all)
  begin
    if SEL = '1' then
      OP <= A;
    else
      OP <= B;
    end if;
  end process MUX;
```

Sequential inside

```
  Y <= A and B;
end DESIGN;
```

Concurrent outside



- Processes are contained in architecture
- There can be any number of processes
- Processes are concurrent outside and sequential inside
- Swapping processes don't affect the behavior
- Concurrent statements are allowed along with processes

Processes must be contained within an architecture; we can have any number of processes within one architecture. The processes execute concurrently, and statements inside the process execute sequentially; hence, we could swap the processes around in the architecture and still get the same simulation behavior.

Outside processes, we are allowed to have concurrent statements. These concurrent statements are considered to be like miniature processes to fit into the VHDL connectivity model, i.e., a network of processes communicating concurrently via signals that we have seen. In the given code, process P1, process MUX, and the concurrent signal assignment all execute simultaneously. Hence, their order can be changed with no change in behavior. Whereas statements within each process execute sequentially, the order of these statements can't be changed.

## Equivalent Process

All concurrent signal assignment statements have an equivalent process form

```
architecture SEQ of DES is
begin
  P1 : process (all)
  begin
    Y <= A and B;
  end process P1;
  ...
end RTL;
```



```
architecture CON of DES is
begin
  Y <= A and B;
  ...
end RTL;
```

### Sequential statements

- Inside of a process
- Process sensitivity list must be defined
- Can have multiple signal assignments in one process
- Combinational and registered logic (RTL)

- ### Concurrent statements
- Outside of a process
    - Equivalent to a process
  - Sensitivity list added by the compiler
  - Multiple signal assignments require multiple concurrent statements
    - Equivalent to multiple processes
  - Combinational logic only (RTL)



All concurrent signal assignment statements have an equivalent process form. Indeed the VHDL compiler replaces each concurrent signal assignment with its equivalent process as it builds a simulation kernel for your design. It may be helpful to consider the equivalent process for a concurrent assignment when trying to understand code.

We know that sequential statements occur inside a process. It must have a defined sensitivity list, there can be multiple signal assignments in one process, and it can infer both combinational and registered logic.

A concurrent statement is just a short-hand method of writing a process. It occurs outside a process.

The advantages of concurrent statements are shorter code, and you don't have to define a sensitivity list explicitly; it is added automatically by the compiler.

Disadvantages are that each concurrent statement is equivalent to a process, and we need multiple concurrent statements to assign to multiple signals. As multiple assignments can be achieved in one process with several sequential statements, concurrent statements are less efficient in this case. It infers combinational logic only.

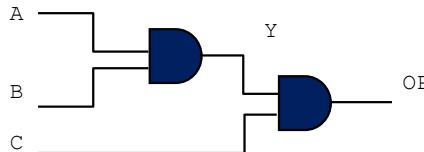
For simple combinational logic, we choose to use a process or concurrent statement(s). Readability and ease of design are crucial in choosing which to use.

In the code given, we see that a single concurrent assignment, Y assigned with A and B, is expanded to process P1 with sensitivity list A and B and a sequential statement, Y assigned with and B. Both are equivalent.

## Concurrent Assignment Statements

Order independent

What you write is what you get



```
architecture CON1 of ANDS is
begin
    Y <= A and B;
    OP <= C and Y;
end CON1;
```

```
architecture CON2 of ANDS is
begin
    OP <= C and Y;
    Y <= A and B;
end CON2;
```

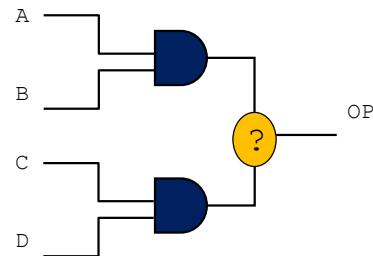
Concurrent assignment statements are, as the name suggests, concurrent; in other words, the order in which they appear in your architecture does not affect the behavior of your model. What you write is what you get. The best way to understand what is happening is to visualize each operation like a schematic: such a drawing is always concurrent.

In the example on the slide, we see that Y assigned with A and B infers, AND gate with A and B as inputs, and Y as output. Similarly, OP assigned with C and Y infers a AND gate, with C and Y as input and OP as output. When we change the order of statements, in case two, it only changes which gate we draw first and which next, but the overall circuit remains the same.

## Multiple Concurrent Assignments

- Multiple concurrent assignments to a single signal are “wired together”
- Signal must be of a *resolved* type
- `std_logic` and `std_logic_vector` are the only resolved types commonly used
- Uses a *resolution function* to determine the final value

```
architecture CONC of MULT is
  signal OP : std_logic;
begin
  OP <= C and D;
  OP <= A and B;
  ...
end CONC;
```



98 © Cadence Design Systems, Inc. All rights reserved.

Let's look at two concurrent signal assignment statements with one target, OP. The outputs of A and B and C and D are wired together. Also, OP is of the type `std_logic`, which is of resolved type. `std_logic` and `std_logic_vector` are the only resolved types in VHDL, and VHDL requires a resolution function to determine the final OP value when any of A, B, C, etc., or D change value. You can think of the resolution function as modeling the wired logic functionality of the technology you are using. If a resolution function is not defined, your compiler will give you an error message. Resolution functions are discussed in more detail in the Functions and Procedures module.

## Multiple Assignment Application

```

architecture RTL of TRISTATE is
  signal ENA, ENB : std_logic;
  signal OPA, OPB, OP : std_logic;
begin

  TRI_A : process (all)
  begin
    if ENA = '1' then
      OP <= OPA;
    else
      OP <= 'Z';
    end if;
  end process TRI_A;

  TRI_B : process (all)
  begin
    if ENB = '1' then
      OP <= OPB;
    else
      OP <= 'Z';
    end if;
  end process TRI_B;

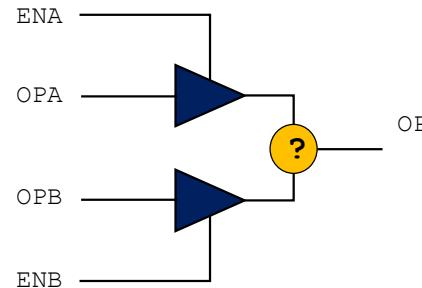
end RTL;

```

Multiple assignments to the same signal results usually an error

One exception is a tri-state bus

- Tri-state controllers drive data onto the bus when enabled



Multiple drivers on the same signal would normally be an error in hardware design. However, there is one common application of multiple drivers in hardware design is tri-state buses. Tri-state drivers control multiple accesses to a single bus under the control of an enable signal. In a tri-state buffer, when enable is high, the output is driven by input else, Z is driven. In the code given on the slide, there are two processes TRI\_A, which updates OP with OPA when enabled, ENA goes high, else drives Z, and process TRI\_B, which drives OP with OPB when enabled, ENB is high, else Z. Since, both the process drive onto same signal OP, which is of the type standard logic, which is of resolved type, OP is assigned with a value as dictated by the resolution function.

If OP were not of resolved type, then the compiler would throw an error, which is not the case here.

## The Process (A Reminder!)

```

architecture RTL of E is
begin

  SELECTOR :
  process (all)
    -- process (A1, A2, B1, B2, S1, S2)
  begin
    if S1 = '1' then
      OP1 <= B1;
    else
      OP1 <= A1;
    end if;
    if S2 = '1' then
      OP2 <= B2;
    else
      OP2 <= A2;
    end if;
  end process SELECTOR;

end RTL;

```

Sequential execution

Multiple processes execute concurrently

- Contains statements that execute in sequence
- Executed by an event on the signal in the sensitivity list
- Signals are updated at the end of the process



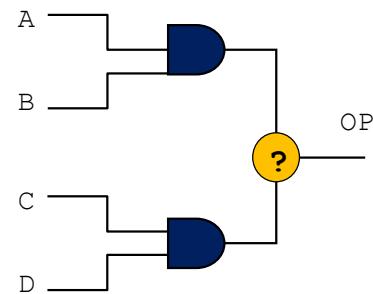
Let's recall the process concept. A process is used inside an architecture. It may have a label, but this is optional. It has a sensitivity list which contains a list of those signals to which the process is sensitive. It has a body delimited by the words begin and end process. Inside the body are sequential statements. A process will execute if there is an event, that is, a change of value, on any of the signals in the sensitivity list. After executing all statements in its body, a process will “suspend” until the next event occurs on any signals in its sensitivity list. Any signals assigned within the process are not updated until after the process has been suspended.

In the code on the slide, we have a process with A1, A2, B1, B2, S1, and S2 in the sensitivity list; hence, if any of their value changes, it triggers the process execution and statements inside the process gets executed sequentially. Process updates output, OP1, and OP2 depending on the values taken by the inputs.

## Multiple Sequential Assignments

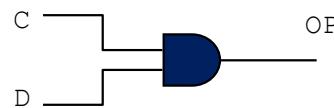
Multiple assignments lead to different results in sequential and concurrent assignments

```
architecture CONCURRENT of MULTIPLE is
  signal A, B, C, D, OP : std_logic;
begin
  OP <= A and B;
  OP <= C and D;
end CONCURRENT;
```



Multiple concurrent assignments to the same target assign a value based on the resolution function

```
architecture SEQUENTIAL of MULTIPLE is
  signal A, B, C, D, OP : std_logic;
begin
  SEQ : process (all)
  begin
    OP <= A and B;
    OP <= C and D;
  end process SEQ;
end SEQUENTIAL;
```



Multiple sequential assignments to the same target use the last assignment

Sequential assignment statements behave differently from concurrent assignment statements. The behavior of multiple sequential assignments to the same target differs from that of multiple concurrent signal assignments to the same target. This is because signals assigned within a process are not updated until after the process suspends. The question therefore arises: if a signal is assigned more than once, which assignment will be used to update it? The designers of the language decided that it would be the last assignment. In the code on the slide, in-process SEQ, OP will get C and D – it will never see A and B, not even as a glitch! Whereas, in case1, where there are multiple concurrent assignments onto OP, the value is updated as decided by the resolution function, as OP is of the resolved data type.

## Multiple Sequential Assignments: Example

```
NODEF : process (all)
begin
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process NODEF;
```

```
DEF : process (all)
begin
  OP <= B; -- default
  if SEL = '1' then
    OP <= A;
  end if;
end process DEF;
```

- Last assignment takes effect
- Signals updated after process suspends
- Two processes are equivalent
- The Second example is a specific coding style
  - Helps prevent the accidental generation of latches in the synthesis
  - Can be more efficient for simulation if the most common assignment is the default

102 © Cadence Design Systems, Inc. All rights reserved.



Let's look at two equivalent styles of coding. Since the last assignment takes effect within a process and signals are updated after the process ends, these two processes behave identically.

In both of them, assignments to OP are complete; that is to say, OP gets assigned to whatever the value of SEL. In the first code, when SEL equals one, we see that OP is assigned with A else B. If SEL = '1', OP will get B's value in the bottom one. However, if SEL = '1', OP gets assigned twice, and since only the last assignment takes effect, OP will get A. Hence, these two processes are equivalent. The DEF process uses the default assignment statement. This is a way of ensuring that assignments are complete without having to put else clauses into all our if statements. We will see an example of the usefulness of default assignments in the Definition of RTL Code module.

## Conditional Signal Assignment

```
entity BRANCH is
  port (A, B, C, T : in std_logic_vector(3 downto 0);
        OP       : out std_logic_vector(3 downto 0));
end BRANCH;
```

- Concurrent form of if
  - Only one target
  - Unconditional else branch
- Equivalent to a process containing a single if statement

```
architecture USE_IF of BRANCH is
begin
  IFSEQ : process (all)
  begin
    if (T > "0101") then
      OP <= A;
    elsif (T < "0101") then
      OP <= B;
    else
      OP <= C;
    end if;
  end process IFSEQ;
end USE_IF;
```



Beware

Conditional signal assignment  
cannot be used inside a process

```
architecture USE_CONDITIONAL of BRANCH is
begin
  OP <= A when T > "0101" else
  B when T < "0101" else
  C;
end USE_CONDITIONAL;
```

103 © Cadence Design Systems, Inc. All rights reserved.



VHDL offers a concurrent version of the if statement, known as the Conditional Signal Assignment.

You cannot use the conditional signal assignment inside a process – it is a concurrent statement. This limitation is removed in VHDL2008.

The language defines the concurrent statement and the process shown above to be equivalent.

The conditional signal assignment is just a “shorthand” way of writing a process with a single if statement containing a single target and unconditional else.

You cannot use the if statement outside a process – it is a sequential statement.

From VHDL'93, the requirement for an unconditional else does not apply – if none of the if conditions are true, the signal target remains unchanged.

## Selected Signal Assignment

```
entity BRANCH is
  port (A, B, C, T : in std_logic_vector(3 downto 0);
        OP       : out std_logic_vector(3 downto 0));
end BRANCH;
```

- Concurrent form of `case`
  - Only one target
- Equivalent to process containing a single `case` statement

```
architecture USE_IF of BRANCH is
begin
  IFSEQ : process (all)
  begin
    case T is
      when "0000" | "0100" =>
        OP <= B;
      when "0101" =>
        OP <= C;
      when others =>
        OP <= A;
    end case;
  end process IFSEQ;
end USE_IF;
```



Beware

Selected signal assignment  
cannot be used inside a process

```
architecture USE_SELECTED of BRANCH is
begin
  with T select
    OP <= B when "0000" | "0100",
    C when "0101",
    A when others;
end USE_CONDITIONAL;
```

104 © Cadence Design Systems, Inc. All rights reserved.

Similar to the concurrent form of the `if`, VHDL gives the Selected Signal Assignment's concurrent state of the `case` statement.

You cannot use the selected signal assignment inside a process – it is a concurrent statement. This limitation is removed in VHDL2008.

As for the `case` statement, the rules are:

- All values in the `case` expression have branches specified.
- No branches overlap.

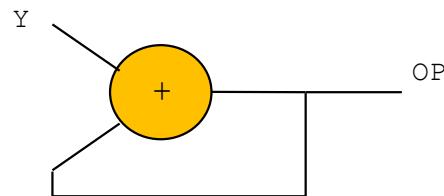
The language defines the concurrent statement and the process to be equivalent.

The selected signal assignment is just a “shorthand” way of writing a process with a single `case` statement containing a single target.

## Don't Create Zero-Delay Feedback Loops!

Concurrent statement

```
OP <= OP + Y;
```



Equivalent process

```
process (all)
begin
    OP <= OP + Y;
end process;
```

- Concurrent assignment with delay-less feedback is syntactically allowed in VHDL but doesn't infer sensible hardware
  - An event on  $Y$  triggers the process and updates  $OP$  with a new value
  - As  $OP$  is in the sensitivity list, this re-triggers the process...
  - ... which assigns a new value to  $OP$
  - So the process repeats indefinitely
- Simulator runs, but simulation time doesn't advance

105 © Cadence Design Systems, Inc. All rights reserved.


A concurrent assignment statement with feedback translates into concurrently operating hardware with a delay-less feedback path. This is syntactically correct VHDL. However, it is not sensible hardware.

In addition, when the process is first executed, triggered by an event on  $Y$ , for instance, a new value of  $OP$  will be assigned.

When the process suspends and  $OP$  is updated with the new value, this will re-trigger the process since  $OP$  is in the sensitivity list of the equivalent process.

Re-executing the process gives us a new value of  $OP$ , which re-triggers the process, and this repeats.

The delay-less combinational feedback gives us an infinite programming loop during simulation. If your simulator is running but simulation time is not advancing, then you may have a simulation loop somewhere in your code.

## Concurrent and Sequential Statements: Summary

1. What happens if multiple sequential assignments are made to the same signal?
2. What are the two rules for using a conditional signal assignment?
3. Are conditional and selected signal assignment statements concurrent or sequential?
4. Why should you not create combinational feedback loops?

106 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## Concurrent and Sequential Statements: Summary (Solutions)

1. What happens if multiple sequential assignments are made to the same signal?
  - The last one in wins.
2. What are the two rules for using a conditional signal assignment?
  - There can be only one target for the assignment.
  - There must be an unconditional `else`.
3. Are conditional and selected signal assignment statements concurrent or sequential?
  - Concurrent. They cannot be used inside a process unless your compiler supports VHDL2008.
4. Why should you not create combinational feedback loops?
  - Because they create infinite simulation loops.



*This page does not contain notes.*

## The Simulation Cycle and Process Control

**Module** 8

Revision 1.0

Version VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In the last section, we introduced some of the most commonly used concurrent and sequential statements. In this section, we will look at the time domain behavior of the simulation.

## Module Objectives

In this module, you

- Outline how the VHDL simulator works
- Describe the concept of event-driven verification
- Define the following concepts
  - Simulation cycle
  - Delta cycle
  - Processes with and without sensitivity lists
  - Wait statement



In this module, you will be able to outline how the VHDL simulator works, describe the concept of event-driven verification, define the following concepts, simulation cycle, delta cycle, processes with and without sensitivity lists, and wait statement.

## The Wait Statement

- Suspends execution of process
- Execution continues when the wait condition is satisfied
- There are four formats

```
-- wait formats can be combined  
wait on A, B for 10 ns;
```

Waits for:

`wait for <specific time>;`

A specific time

`wait on <signal_list>;`

An event on signal(s)

`wait until <condition>;`

A condition to occur (requires event)

`wait;`

Indefinitely (never to be reactivated)



A wait statement suspends the execution of a process. Execution will continue with the next statement when the wait condition is satisfied.

There are four formats of wait statements, as shown.

Wait for waits for a specific time to be reached.

Wait on waits for an event, which is a change of value, on any signal in the list.

Wait until waits for the Boolean condition to be true. Note that this condition requires an event on one of the signals in the boolean expression before it is evaluated. We will see an example of that later.

Finally, a standalone wait suspends process execution forever. This format is only used in certain special cases.

You can combine formats, for example:

```
wait on <signal list> for <specific_time> is valid
```

## Wait Examples

```
STIMULUS: process
begin
    SEL <= '0';
    BUS_B <= "0000";
    BUS_A <= "1111";

    wait for 10 ns;
    SEL <= '1';
    wait for 10 ns;
    BUS_B <= "0101";
    BUS_A <= "1010";

    wait for 10 ns;
    -- stop process
    wait;
end process STIMULUS;
```

-- wait formats can be combined  
**wait on A, B for 10 ns;**

```
library IEEE;
use IEEE.std_logic_1164.all;
entity FLOP is
    port (D, CLK : in std_logic;
          Q      : out std_logic);
end FLOP;

architecture BEHAVE of FLOP is
begin
    EDGE_TRIGGERED : process
    begin
        wait until CLK='1';
        Q <= D;
    end process EDGE_TRIGGERED;
end BEHAVE;
```

time	SEL	BUS_B	BUS_A
0 ns	'0'	"0000"	"1111"
10 ns	'1'	"0000"	"1111"
20 ns	'1'	"0101"	"1010"

111 © Cadence Design Systems, Inc. All rights reserved.



Let us look at some wait examples. The STIMULUS process will execute until the first wait statement. It will then suspend with SEL, BUS\_B, and BUS\_A being updated to the values shown in the table. These values will be maintained for the next 10 nanoseconds, after which the process will resume executing. The process assigns a new value for SEL, then executes another wait statement. This causes the process to suspend, SEL to be updated to '1', and process execution to be paused for another ten nanoseconds. After resuming, BUS\_B and BUS\_A are assigned with new values; on encountering the third wait, the process is suspended, and BUS\_B and BUS\_A are updated with new values and are maintained for another ten nanoseconds. The process resumes and executes the last wait, which stops the process forever.

Note that no simulation time will be used up while executing the process.

In the FLOP model, the wait statement causes suspension of the process until CLK='1'. Remember that the expression CLK='1' will only be evaluated when there is an event upon CLK. Every clock cycle, there will, of course, be two events on CLK: when it goes from '0' to '1' and when it changes from '1' to '0'. The expression is tested one delta cycle after the event, so it is only on a rising edge that it will evaluate true. Therefore, this simple statement will, rather elegantly, detect a rising edge.

Finally, wait conditions can also be combined, as shown. In the example above, the statement will wait on an event on signals A or B for up to 10 nanoseconds.

## General Form of the Process

```
SL : process (all)
begin
    <sequential statement>;
    <sequential statement>;
    ...
end process SL;
```

```
WS : process
begin
    <sequential statements>
    <wait statement>
    <sequential statements>
    <wait statement>
    ...
end process WS;
```

Processes come in two forms:

- With sensitivity list
- Without sensitivity list

The process with a sensitivity list:

- Triggers when there is an event on the sensitivity list
- Executes from the first line
- Suspends at the last line
- Retriggered on new event

Process without a sensitivity list:

- Infinite loop, broken by wait statements
- Suspends and re-executes from embedded wait statements



There are two forms of processes in VHDL. One with a sensitivity list, and the other, without a sensitivity list. When we consider a process with a sensitivity list, it gets triggered when there is an event on the signal in the sensitivity list., and statements inside the process are executed sequentially until it reaches the end of the process, where it suspends. This process is retriggered and executed when there is a new event on any signals in the sensitivity list.

The general definition of a process without a sensitivity list is an infinite loop, broken by the use of `wait` statements to force the suspension. If the process doesn't have a sensitivity list, it will run infinitely and causes the simulator to hang. However, this is eliminated by putting a `wait` statement. Now, the process executes all statements sequentially until it encounters the `wait` statement, where it suspends. Managing the process resumes with the next statement when the `wait` condition evaluates true. This is how the process is controlled.

## The Behavior of a Process

```
SL : process (all)
begin
  if (SEL = '1') then
    OP <= A;
  else
    OP <= B;
  end if;
end process SL;
```

```
WS : process
begin
  if (SEL = '1') then
    OP <= A;
  else
    OP <= B;
  end if;

  wait on A, B, SEL;

end process WS;
```

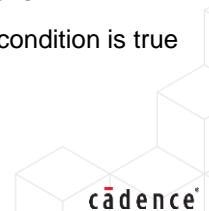
Language defines these two processes as equivalent

The process with a sensitivity list is a specific process style:

- Sensitivity list is an implied wait condition on all signals read by the process
- There can be only one sensitivity list
- Process must not contain wait statements

Process without sensitivity list:

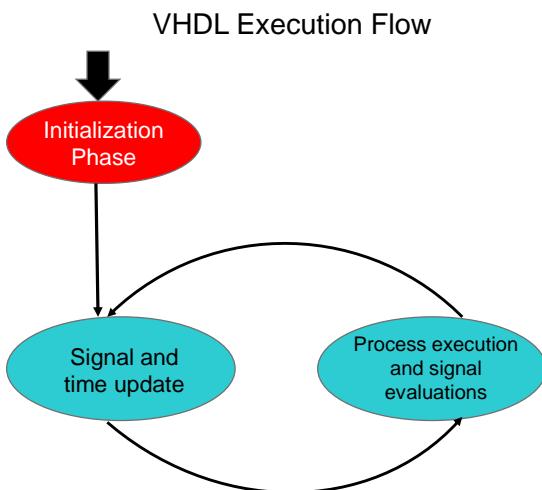
- Infinite loop
- Suspended by one or more wait conditions
- Execution resumes from wait when the condition is true



Let us look into the behavior of the process with a sensitivity list and a process without a sensitivity list but with a `wait` statement. The use of the sensitivity list is a particular process style and behaves like an implied `wait on <signal list>` in the sensitivity list. Hence, instead of writing a process with a sensitivity list, we can write a process without a sensitivity list but a `wait` statement on all signals read by the process at the end of the process. Both are logically equivalent. A process can have more than one `wait` statement, but it can only have one sensitivity list; further, it is not allowed to contain `wait` statements if it has a sensitivity list.

In the code on the slide, both the processes are shown to behave identically. Process WS executes until it encounters a `wait` statement; here, the `wait` statement is on all signals the process reads, including A, B, and SEL. On `wait`, the process suspends and resumes when any of the signals in `wait` changes value. This is similar to the behavior of process SL with a sensitivity list, where the process executes sequentially until the end, where it suspends and is retriggered only when there is an event on any of the signals A, B, or SEL in the sensitivity list.

## VHDL Execution



- VHDL simulation is based on event-driven verification
  - An event is a change in the value of any signal
- In VHDL, no signal assignment is made immediately
  - But is scheduled to be updated in some future time
- Simulator keeps a clock to keep track of the passage of simulation time

The execution of a VHDL model consists of three stages:

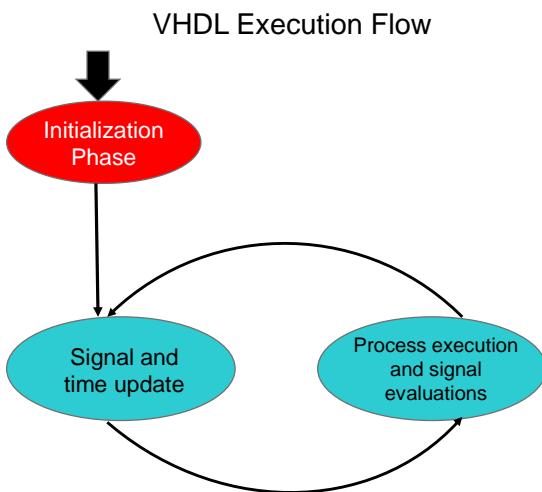
1. Initialization phase
2. Signal Update phase
3. Process Execution phase

Simulation cycles consist of repeated execution of Signal Update and Process Execution phases

VHDL simulations are event-driven. Simulation led by events means the processes simulation is only made when there is an event on any signal in the sensitivity list. An event implies a change in the value of any signal. In VHDL, no signal assignment is made immediately but is scheduled to be updated at some future time.

The simulator keeps a clock to keep track of the passage of simulation time. The execution phase of VHDL consists of three phases. Initialization phase, followed by repeated execution of signal update and process execution phase until the end of the simulation.

## VHDL Execution Initialization Phase



### Initialization phase

- Simulation time set to zero
- Each signal is assigned with initial/default values
  - Leftmost value of the type is assigned
    - E.g., `std_logic` is initialized with '`U`'
- Process instances in design are started
- Some initial values are modeled to set some transactions
  - To get the simulation underway
- Process is suspended on reaching the end of the process or by wait statements

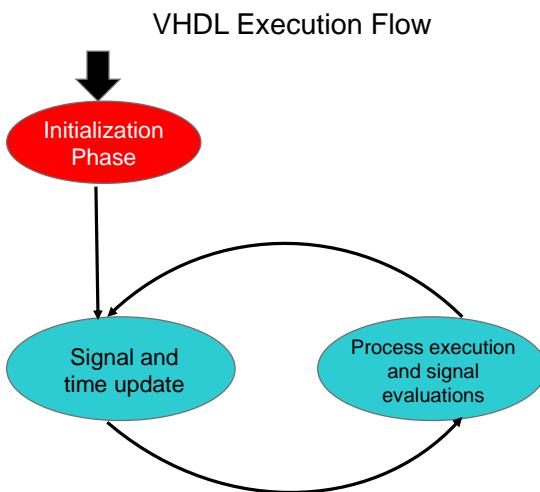
Initialization is now complete



In the initialization phase, the simulation time is set to zero. Each signal is assigned with its initial values declared for the signal or with the default value if no initial value is declared. The default value depends on the type of the signal and is assigned with its leftmost value; for example, `std_logic` is assigned with "`U`". After the objects are initialized, each of the process instances in the design is started and executes the sequential statements in its body. We usually write a model so that at least some of these initial assignments schedule some transactions to get the simulation underway, then suspend by executing a wait statement, which we see later, or by hitting the bottom of the process. When all process instances have been suspended, initialization is complete, and the simulator can start the first simulation cycle.

Triggered processes are executed in each simulation cycle, and signals are updated. This repeats until the end of the simulation.

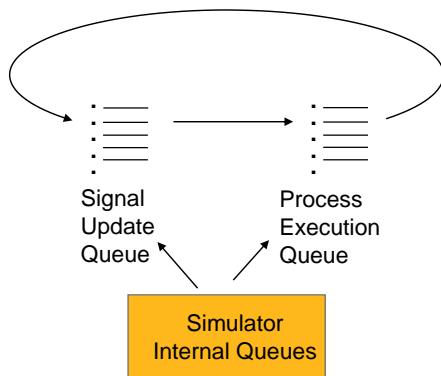
## VHDL Execution Process: Simulation Cycle



- First simulation cycle starts
- Simulation consists of the repetitive execution of the *simulation cycle*, during which:
  - Signals are updated
    - In the signal update phase
  - Processes are executed
    - In the process execution phase
- Simulation time advances
  - Next cycle starts
    - Repetitive execution of signal update and process execution phases

In each simulation cycle, signals are updated in the signal update phase, and triggered processes are executed in the process execution phase. Simulation time is advanced, and the next simulation cycle begins. This repeats until the end of the simulation.

## The Simulation Cycle



VHDL simulator maintains two queues:

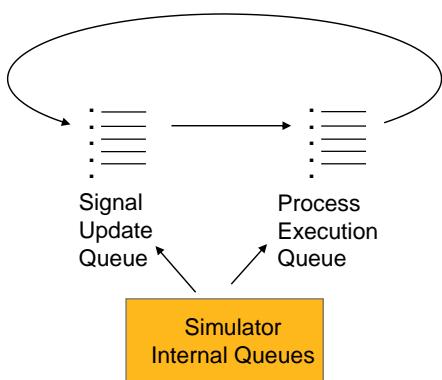
1. Signal update queue
  - Holds the list of transactions scheduled on all signals
  - These are not updated immediately but in some future time
2. Process execution queue
  - Holds the list of processes awakened by events/timeouts



Simulation execution is very precisely defined in VHDL to avoid issues with race conditions and non-determinism, which affect other languages. VHDL Simulation consists of the repetitive execution of the *simulation cycle*, during which processes are executed and nets updated. Let us look at the simulation cycle in more detail.

VHDL simulator maintains two queues, one, to keep track of scheduled transactions on the signal, called signal update queue; and two, a queue to keep track of the processes awakened by events or timeouts, called process execution queue.

## The Simulation Cycle (Signal Update Phase)



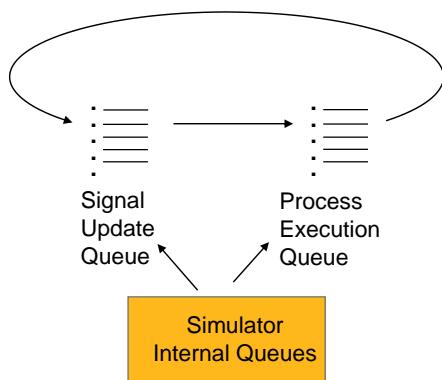
In each simulation cycle:

- Simulation time is advanced to the earliest time at which:
  - A signal assignment is scheduled
  - A process is triggered
  - A timeout occurs
- Signals with new values are updated
- Process sensitivity lists checked
- Triggered processes placed in the process execution queue

The concept is as follows.

Each simulation cycle consists of a signal update phase followed by a process execution phase. First, simulation time is advanced to the earliest time when a scheduled transaction or process triggers or timeout. Signals with new values are updated, which may result in events. Process sensitivity lists are checked to see if any of them is triggered, and triggered processes are placed in the process execution queue.

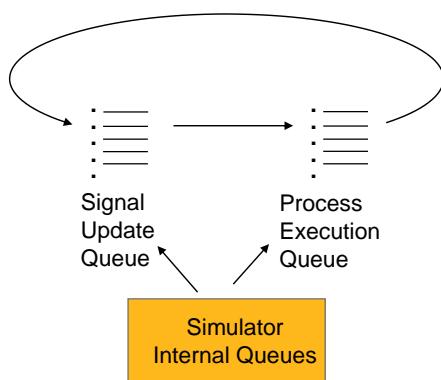
## The Simulation Cycle (Process Execution Phase)



- After all signals in the queue are updated:
  - Processes in the process execution queue are executed until they 'suspend'
  - Any new signal updates are placed in the signal update queue
  - They are not updated immediately but in some scheduled future time
- Simulation time proceeds and the next cycle begins
- This repeats until values become stable
  - When the signal update and process execution queue becomes empty
- Simulation is now complete

When all signals in the queue are updated, all processes in the process execution queue are executed until they 'suspend,' and signals to be updated due to the process execution are placed in the signal update queue. These updates are not made immediately but at some scheduled future time. One cycle is now complete. Simulator time advances, and the next cycle begins. This repeats until values become stable, i.e., until the signal update and process execution queue becomes empty. The simulation is now complete.

## The Simulation Cycle (Delta Cycle)



On completion of a cycle, simulation time is advanced to:

- The earliest scheduled next transaction, or
- The next process triggered by an event, or
- A time-out occurs

Then the next cycle begins

If the simulation time delay is zero:

- The next simulation cycle is performed at the same simulation time as the previous one
- We call such simulation cycle a delta cycle

If there is a non-zero delay:

- The delay value advances simulation time
- The next simulation cycle begins

Note: Delta cycles consumes simulation cycles but not simulation time



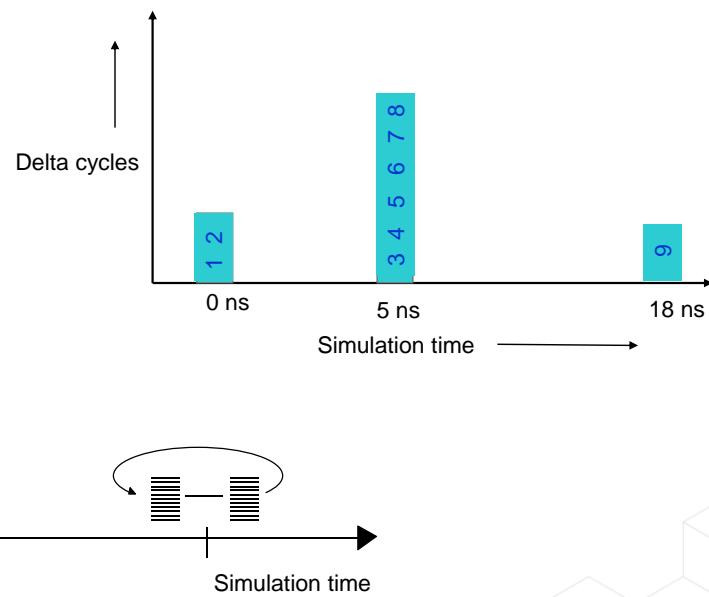
We see that simulation time is advanced to the earliest scheduled next transaction or next process triggered by an event or time-out on completion of a cycle. If this delay is zero and there is an immediate cycle following the current one in a given simulation time, we call such simulation cycle a delta cycle, i.e., a simulation cycle that is performed at the same simulation time as the previous one called a delta cycle.

If there is a delay, simulation time is advanced by the delay value, and the next simulation cycle begins.

Note: Delta cycle consumes simulation cycle but not simulation time.

## Delta Cycle and Simulation Time

- Multiple delta cycles at each point of simulation time
- Time advances when there is no activity to the point where the next activity is scheduled
- This could be due to:
  - Delayed signal assignment  
`A <= B after 5 ns;` -- not used in RTL
  - Suspending a process until a future time  
`wait for 5 ns;` -- not used in RTL



121 © Cadence Design Systems, Inc. All rights reserved.



At each point of simulation time, there can be multiple delta cycles. When no more signals need updating and no more processes need executing, the simulator is free to advance simulation time until the next scheduled activity is found.

A scheduled activity may be assigned a signal at a future point in time. This is done by attaching an `after` clause to a signal assignment. The `after` clause delays the assignment until the simulator has advanced by the required time. After clauses are never used in RTL code as they are non-synthesizable but are used in behavioral testbenches and gate-level timing models.

A scheduled activity may also be executing a process at a future point in time. This is done by executing a `wait for X nanoseconds` statement. The `wait for` statement suspends the process until the simulator has advanced by the required time. Wait for statements are never used in RTL code; they are non-synthesizable but are used in behavioral testbenches and gate-level timing models.

In the graph on the right, we see that, at simulation time 0 ns, there occur two delta cycles, then simulation time proceeds to 5 ns, as there is no activity from 0 ns to 5 ns, at 5 ns, six delta cycles occur, and at 18 ns there is one delta cycle. Note that delta cycles only consume simulation cycles but not simulation time.

## Simulation Cycle 1 (at simulation time t)

```

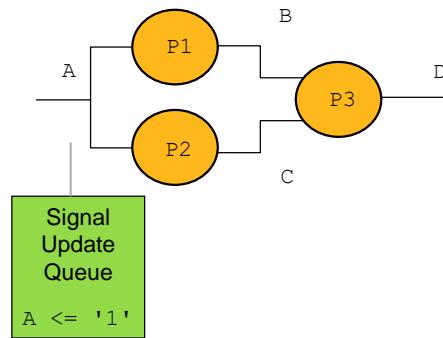
architecture SIM of E is
  -- A is input, D is output
  signal B, C : std_logic;
begin
  P1: process (all)
  begin
    B <= A;
  end process P1;

  P2: process (all)
  begin
    C <= A;
  end process P2;

  P3: process (all)
  begin
    D <= B and C;
  end process P3;
end SIM;

```

122 © Cadence Design Systems, Inc. All rights reserved.



- Assume A, B, C, D are all '0'
- A assigned to '1' at  $t_{ns}$  is placed in the signal update queue

Signal values at start

A :	'0'
B :	'0'
C :	'0'
D :	'0'



We need to look at how delta cycles affect the simulation of your model by looking into a code example.

We will consider an architecture with three processes, P1, P2, and P3. P1 and P2 are sensitive to signal A and represent concurrent parallel logic. P3 is sensitive to signals B and C, which are output from P1 and P2, and represents logic in series with the other two processes.

As the simulation proceeds, we know that your simulator will maintain two lists, a signal update list, and a process execution list, the contents of which will change with simulator time. At a particular time in the simulation, the signal update list will contain all signals assigned a value and are waiting to be updated. The process execution list will contain all processes triggered by a signal event and waiting to be executed.

Let us say that there has been an event scheduled upon signal A – for example, A is currently '0' and is assigned to '1' at the time,  $t$  nanoseconds. A is placed on the signal update list.

This is an example entity for the above architecture:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity E is
  port (A: in std_logic;
        D: out std_logic);
end E;

```

## Simulation Cycle 2 (signal update phase of t+delta cycle)

```

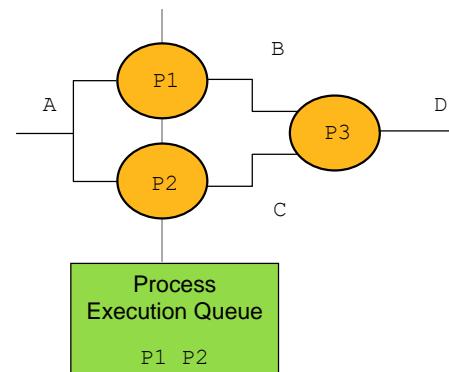
architecture SIM of E is
  signal B, C : std_logic;
begin

  P1: process (all)
  begin
    B <= A;
  end process P1;

  P2: process (all)
  begin
    C <= A;
  end process P2;

  P3: process (all)
  begin
    D <= B and C;
  end process P3;
end SIM;

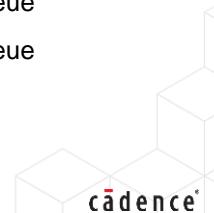
```



- A changes from '0' to '1' at t+delta
- Process P1 placed on the execution queue
- Process P2 placed on the execution queue

Signal  
values  
after update

A :	'1'
B :	'0'
C :	'0'
D :	'0'



123 © Cadence Design Systems, Inc. All rights reserved.

As there is no process to be executed in the process execution queue at t ns, simulation time proceeds by delta, and signals in the signal update queue are updated when the signals are updated. At t + delta, A takes the new value '1'.

A is in the sensitivity list of processes P1 and P2, so both processes are placed on the process execution list. Hence, at t + delta, A equals one, and processes P1 and P2 are in the process execution queue.

## Simulation Cycle 3 (process execution phase of t+delta cycle)

```

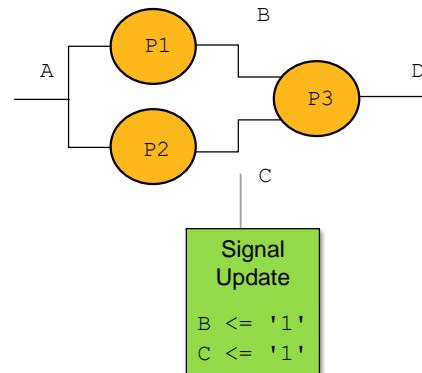
architecture SIM of E is
    signal B, C : std_logic;
begin

    P1: process (all)
    begin
        B <= A;
    end process P1;

    P2: process (all)
    begin
        C <= A;
    end process P2;

    P3: process (all)
    begin
        D <= B and C;
    end process P3;
end SIM;

```



- Processes P1 and P2 execute (random order)
- Updates to C and B are scheduled by P1 and P2
  - Placed in the signal update queue

Signal  
values  
after execution

A :	'1'
B :	'0'
C :	'0'
D :	'0'

124 © Cadence Design Systems, Inc. All rights reserved.



Since all signals in the signal update queue are updated, we move to the second phase of the first delta cycle, the process execution phase. Processes P1 and P2 in the process execution queue are executed in random order. We cannot predict the order of execution of the processes, but this should not matter.

Process P1 assigns the value '`1`' to B. This is a new value for B, so the assignment is placed on the signal update list.

Process P2 assigns the value '`1`' to C. This is a new value for C, so the assignment is placed on the signal update list.

No more processes remain to be executed, so the simulation moves on to the signal update phase of the next delta cycle.

## Simulation Cycle 4 (signal update phase of t+2 delta cycles)

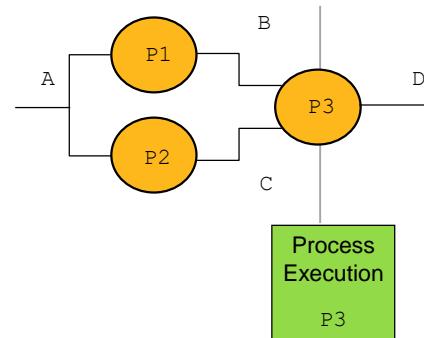
```

architecture SIM of E is
  signal B, C : std_logic;
begin
  P1: process (all)
  begin
    B <= A;
  end process P1;

  P2: process (all)
  begin
    C <= A;
  end process P2;

  P3: process (all)
  begin
    D <= B and C;
  end process P3;
end SIM;

```



- B and C updated to '1' at t+2delta
- Process P3 placed on execution queue

Signal  
values  
after update

A :	'1'
B :	'1'
C :	'1'
D :	'0'



125 © Cadence Design Systems, Inc. All rights reserved.

At t + 2 delta, signals B and C are updated to '1'.

Both B and C are in the sensitivity list of process P3. Therefore, P3 is placed on the process execution list.

No more signals require updating, so simulation moves to the process execution phase of the t + 2 delta cycle.

## Simulation Cycle 5 (process execution phase of t+2 delta cycles)

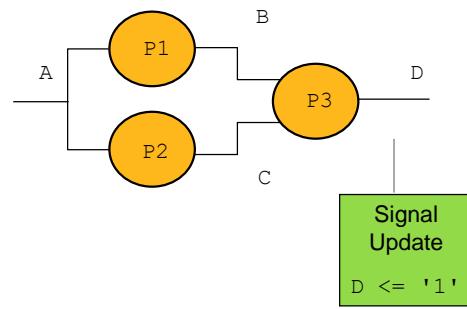
```

architecture SIM of E is
  signal B, C : std_logic;
begin
  P1: process (all)
  begin
    B <= A;
  end process P1;

  P2: process (all)
  begin
    C <= A;
  end process P2;

  P3: process (all)
  begin
    D <= B and C;
  end process P3;
end SIM;

```



- Process P3 executes
- Update to D scheduled

Signal  
values  
after execution

A :	'1'
B :	'1'
C :	'1'
D :	'0'

126 © Cadence Design Systems, Inc. All rights reserved.



When P3 executes, the value '`1`' is assigned to D. This is a new value for D, so the assignment is placed on the signal update list.

No more processes require execution, so simulation moves on to the signal update phase of the next cycle.

## Simulation Cycle 6 (signal update phase of t+3 delta cycles)

```

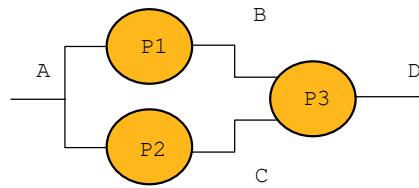
architecture SIM of E is
  signal B, C : std_logic;
begin

  P1: process (all)
  begin
    B <= A;
  end process P1;

  P2: process (all)
  begin
    C <= A;
  end process P2;

  P3: process (all)
  begin
    D <= B and C;
  end process P3;
end SIM;

```



- D updated to '1' at t+3delta
- D is not input to any process
  - No more processes will be executed
  - Process execution and signal update queues are empty
- The simulation cycle completes.

Signal  
values  
after update

A :	'1'
B :	'1'
C :	'1'
D :	'1'



127 © Cadence Design Systems, Inc. All rights reserved.

At t + 3 delta , signal D is updated to '1'.

D is not in the sensitivity list of any process; it is an output from this component.

There are no more processes to be executed, and the model is waiting for another change in input A.

Therefore, the model has taken two delta cycles to reach a “steady state” for a change in A.

## Simulation Cycle Summary

Signal values  
at start

A : '0'
B : '0'
C : '0'
D : '0'

Simulation Time	Process Execution List	Signal Update Queue	Signal Values			
			A	B	C	D
t		A <=1	0	0	0	0
t + delta	P1, P2	B <=1 C <=1	1	0	0	0
t + 2 delta	P3	D <=1	1	1	1	0
t + 3 delta			1	1	1	1

```
architecture SIM of E is
  signal B, C : std_logic;
begin
```

```
  P1: process (A)
  begin
    B <= A;
  end process P1;
```

```
  P2: process (A)
  begin
    C <= A;
  end process P2;
```

```
  P3: process (B, C)
  begin
    D <= B and C;
  end process P3;
end SIM;
```

Here is a summary of the delta cycle activities (described in the previous five slides) required by the architecture SIM to reach a steady state after a value change in input A. We see from the table that, at t nanoseconds, A is assigned to 1 but is updated in the t + 1 delta cycle. A is input for P1 and P2. Hence, they are placed in the process execution queue. After A is updated, P1 and P2 are executed. These processes assign B and C to 1, which is placed in the signal update queue. A second delta, B and C, are updated with 1; this triggers process P3 and is placed in the execution queue. After B and C are updated, no more pending items are in the signal update queue; hence process P3 is executed. P3 assigns D to 1, which is put on the signal update queue. D is updated to 1 in the third delta cycle; since no process is affected by D, the process execution queue becomes empty. The simulation cycle completes since both the queue becomes empty and stable values are reached.

## Simulation Control: Summary Quiz

1. If a process has been suspended by the execution of a `wait until` statement, what will cause it to be reactivated?
2. What is a delta cycle?
3. What causes the suspension of a process?

129 © Cadence Design Systems, Inc. All rights reserved.



Let's take a short quiz on this module. If a process has been suspended by the execution of a `wait until` statement, what will cause it to be reactivated? What is a delta cycle? What causes the suspension of a process?

## Simulation Control: Summary Quiz (Solutions)

1. If a process has been suspended by the execution of a `wait until` statement, what will cause it to be reactivated?
  - A change in value (event) of one of the signals in the wait until condition, and the condition subsequently evaluated to true.
2. What is a delta cycle?
  - A signal update phase followed by a process execution phase.
3. What causes the suspension of a process?
  - The execution of a `wait` statement suspends a process, either implicit (end of a process) or explicit (embedded `wait`).



Here is the solution. A change in the value of one of the signals in the `wait until` condition, and the condition subsequently evaluating to true, will reactivate the process suspended by the execution of `wait until` statement. The delta cycle is nothing but a signal update phase followed by a process execution phase. A process is suspended by executing a `wait` statement, either the implicit end of a process or an explicit embedded `wait`.

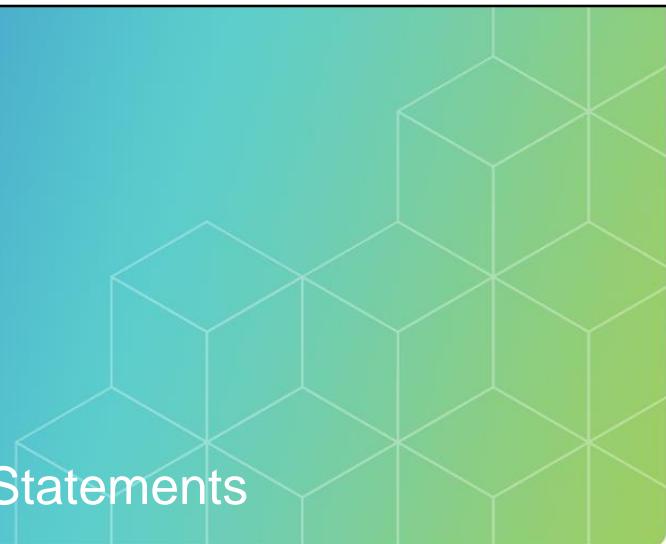
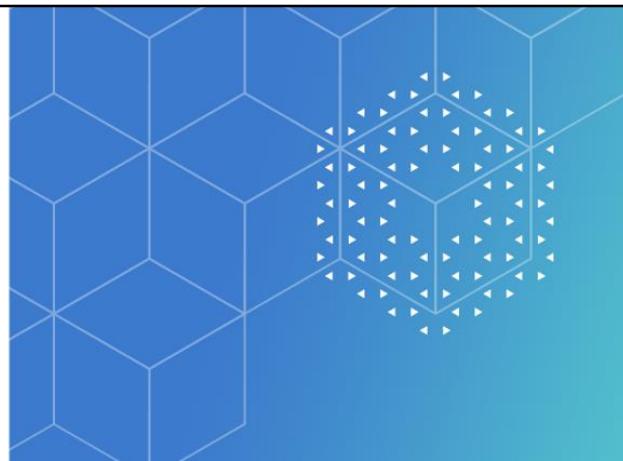
## Lab

### Lab 8-1 Adding a Seven-Segment Display Driver

131 © Cadence Design Systems, Inc. All rights reserved.



You will now have the opportunity to complete a self-paced lab to reinforce the ideas presented in this module.



## Variables and Sequential Statements

**Module** **9**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

We introduced some of the most commonly used concurrent and sequential statements in the previous modules. In this section, we will be looking at loop sequential statements and discuss variables and their usage.

## Module Objectives

In this module, you

- Compare and contrast variables and signals
- Outline variable usage model
- List and use all the array attributes
- Define loop sequential statements, such as
  - For loop
  - While loop
- Describe next and exit statements



By the end of this module, you will be able to compare and contrast variables and signals, outline variable usage models, list and use all the array attributes, and define sequential loop statements, such as for loop, while loop, describe next, and exit statements.

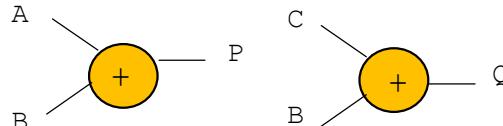
## Variables

```

architecture var of example is
  -- declaration region of the architecture
  signal A, B, C, P, Q : integer;
begin

  VAR: process (all)
    -- declaration region of the process body
    variable J, K : integer;
  begin
    J := A;
    K := B;
    P <= J + K;
    J := C;
    Q <= J + K;
  end process VAR;
...

```



- Declared within the process in the declaration region of the process body before begin
- Can only be used within the process
  - Called its scope
- Retains value between process executions
- Can assign:
  - Signal to variable
  - Variable to signal
- Assignment immediately
  - Like “software”



Beware



134 © Cadence Design Systems, Inc. All rights reserved.

You may have another sort of assignment inside a process other than a signal assignment: the variable assignment. Variables must be declared in the declarations region of the process, before the begin of the process body, not in the declarations region of the architecture. They are visible only inside the process in which they are declared and retain their value while that process is suspended. You can freely assign a signal to a variable and a variable to a signal.

The big difference with variables is that assignments to a variable occur immediately, just as in software.

In the example above, the variable assignments are made to J and K. These variables are immediately updated with the current values of A and B so that P is assigned A+B. In the second variable assignment to J, J is immediately updated with the value of C. Q is therefore assigned C+B. As P and Q are signals, they will be updated after the process suspends.

## Variables Versus Signals

```
...
signal A, B, P : integer;
begin
  PVAR: process (all)
    variable JVAR, KVAR : integer;
  begin
    JVAR := A;
    KVAR := B;
    P <= JVAR + KVAR;
  end process PVAR;
```

```
...
signal A, B, Q : integer;
signal JSIG, KSIG : integer;
begin
  PSIG: process (A, B, JSIG, KSIG)
  begin
    JSIG <= A;
    KSIG <= B;
    Q <= JSIG + KSIG;
  end process PSIG;
```

Variables can be more efficient than signals in combinational logic

- For example, if logic contains serial behavior or intermediate values

If variables are used to store intermediate values, addition is calculated in one pass

If signals are used, the intermediate *signals* must be added to the sensitivity list

- Process then takes several delta cycles to reach a “steady state”

135 © Cadence Design Systems, Inc. All rights reserved.

Process Execution	Signal Update
	A <= 1 B <= 2
PVAR JVAR := 1 KVAR := 2	P <= 3 JSIG <= 1 KSIG <= 2
PSIG	
PSIG	Q <= 3

cadence®

Let us explore some of the differences between using variables and signals. Variables can be more efficient than signals for describing combinational logic, particularly if the combinational logic contains serial behavior and/or uses intermediate or “temporary” storage in the calculation of an algorithm.

In the example on the slide, the left-hand-side process PVAR declares JVAR and KVAR as variables and uses these to temporarily store the values of A and B before assigning the addition to the signal P. Since variable assignment occurs immediately, the value for P is calculated in one pass through the process.

In the right-hand-side process, PSIG, JSIG, and KSIG are declared as signals and used to store A and B temporarily. Since signals are not updated until after the process suspends, we need to add JSIG and KSIG to the sensitivity list of the process. So, when JSIG and KSIG are updated with the values of A and B, the process is re-triggered to assign the correct value to Q. To obtain a value for Q requires two simulation delta cycles before the process PSIG reaches a steady state.

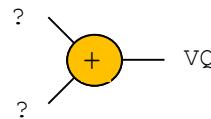
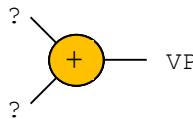
Let us see how this code works, as summarized in the table. Say at simulation time t, A and B changed their values to 1 and 2; this triggers the process PSIG and PVAR, as A and B are in the sensitivity list of both these processes. PSIG and PVAR are placed in the process execution queue and are executed. Process PVAR assigns variables JVAR and KVAR immediately, as they are variables. This is followed by an assignment to P with value three, i.e., 1 plus 2. Still, since P is a signal, its value is not updated immediately but in the next simulation cycle and hence, is placed in the signal update queue. Similarly, process PSIG assigns values for JSIG and KSIG with 1 and 2, which are updated in the next simulation cycle as they are signals. Hence, they are placed on the signal update queue. Since no more processes are to be executed, simulation advances to the signal update phase of the next cycle, the t + delta cycle.

At t + delta, P, JSIG, and KSIG, which are in the signal update queue, are updated with their values 3, 1, and 2.

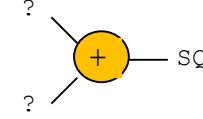
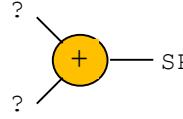
P is the output and hence, doesn't trigger any process. But, the update to JSIG and KSIG triggers process PSIG and is placed in the process execution queue. Since no more signals are updated in the signal update queue, the process execution phase begins, and PSIG is executed. PSIG assigns the value 3 to Q, which is not immediately assigned, as Q is a signal placed in the signal update queue. As no more processes are to be executed, simulation advances to the next cycle, t + 2 deltas. Here, Q is updated to 3, as it is in the signal update queue. Since Q is not in the sensitivity list of any processes and no more updates are pending in the signal update queue, execution stops. Q takes two cycles, unlike P, which takes one cycle to reach a steady state.

## Multiple Assignments

```
signal A, B, C, VP, VQ : integer;  
begin  
PVAR_SUM: process (all)  
variable JVAR, KVAR : integer;  
begin  
    JVAR := A;  
    KVAR := B;  
    VP <= JVAR + KVAR;  
    JVAR := C;  
    VQ <= JVAR + KVAR;  
end process PVAR_SUM;
```



```
signal A, B, C, SP, SQ : integer;  
signal JSIG, KSIG : integer;  
begin  
PSIG_SUM: process (all)  
begin  
    JSIG <= A;  
    KSIG <= B;  
    SP <= JSIG + KSIG;  
    JSIG <= C;  
    SQ <= JSIG + KSIG;  
end process PSIG_SUM;
```



Write down the inputs to these adders  
(in terms of A, B, and C)

Question

In the above code, what are the inputs to these adders?

## Solution: Multiple Assignments

```

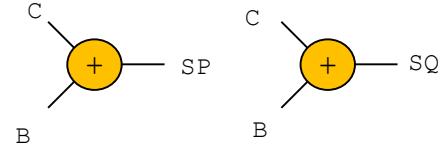
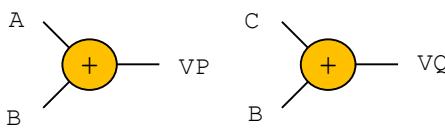
signal A, B, C, VP, VQ : integer;
begin
  PVAR_SUM: process (A, B, C)
    variable JVAR, KVAR : integer;
  begin
    JVAR := A;
    KVAR := B;
    VP <= JVAR + KVAR;
    JVAR := C;
    VQ <= JVAR + KVAR;
  end process PVAR_SUM;

```

```

signal A, B, C, SP, SQ : integer;
signal JSIG, KSIG      : integer;
begin
  PSIG_SUM: process (A, B, C, JSIG, KSIG)
  begin
    JSIG <= A;
    KSIG <= B;
    SP <= JSIG + KSIG;
    JSIG <= C;
    SQ <= JSIG + KSIG;
  end process PSIG_SUM;

```



137 © Cadence Design Systems, Inc. All rights reserved.

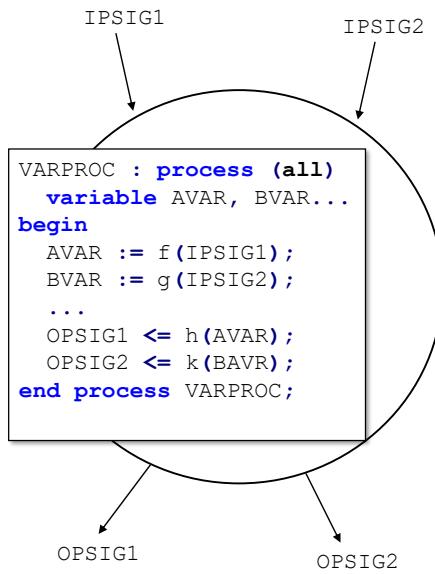


Here is the solution.

In the variable example, when variable JVAR is assigned C, it is immediately updated, making the signal  $VQ \leq C + B$ ;

In the signal example, we have had to add signals JSIG and KSIG to the sensitivity list of the process. Execution of the process takes two delta cycles. On the first cycle, we update JSIG and KSIG. On the second cycle, we update SP and SQ. In the first cycle, the second assignment of C to JSIG overrides the previous assignment to A. Therefore, in the second cycle, JSIG is C, and both SP and SQ are set to  $B + C$ .

## Variable Usage Model



Variables used for algorithms

- Capture input signals in variables
- Perform algorithm using variables
- Assign variables to output signals

A variable cannot be accessed outside of its process

You might be wondering, how are variables used in the real world? Essentially, they are very useful for performing algorithms. To do this, you assign signals to variables, perform the algorithm and then assign your variables to signals to propagate the results of your algorithm throughout your architecture; remember, variables are not visible outside of their process.

In the above code, we see that input signals IPSIG1 and IPSIG2 are assigned to variables AVAR and BVAR, some algorithm is performed on these values, and results are assigned to outputs OPSIG1 and OPSIG2 as shown.

## Loop Statements

Three loop constructs

- for loop
  - Loop for a given number of iterations
- while loop
  - Loop while a condition is true
- loop
  - Infinite loop

Loops can be nested

Label is optional

- Good practice to use, particularly in nested loops

```
[loop_label:]  
for identifier in discrete_range loop  
    -- statements  
end loop [loop_label];  
  
[loop_label:]  
while condition loop  
    -- statements  
end loop [loop_label];  
  
[loop_label:]  
loop  
    -- statements  
end loop [loop_label];
```



Synthesis

for synthesizable with a fixed range  
while generally not synthesizable  
infinite loop not synthesizable

139 © Cadence Design Systems, Inc. All rights reserved.



There are three kinds of loop statements.

A `for` loop defines a loop variable called `identifier` and loops for a given number of iterations defined by the range. The loop variable takes the next value in the range of each iteration.

A `while` loop defines a boolean condition and executes the loop statements for as long as the condition evaluates true. The condition is checked at the start of each loop.

An `infinite loop` iterates around the loop continuously. Other statements must be used to prevent an infinite simulation loop (example – wait statements) and terminate the loop when required. See `exit` on the following slides.

Loops can be nested.

Labels are useful, particularly if you have nested loops.

Note that, for loop is synthesizable with a fixed range, while a loop is generally not synthesizable, an infinite loop is not synthesizable.

## for Loop

```

library IEEE;
use IEEE.std_logic_1164.all;

entity FOR_LOOP is
  port (IP : in std_logic_vector (3 downto 0);
        OP : out std_logic_vector (3 downto 0));
end FOR_LOOP;

architecture REVERSE of FOR_LOOP is
begin
  FL: process (all)
  begin
    for I in 0 to 3 loop
      OP(I) <= IP(3 - I);
    end loop;
  end process FL;
end REVERSE;

```

Loops for a set number of iterations

The loop variable gets values in the range

- Takes on type of range
- Loop variable
- Does not need to be declared
- Can not be assigned *inside* the loop
- Is not visible *outside* the loop
- If the range is dynamic, for loop is not synthesizable



What does this code do?

Question

140 © Cadence Design Systems, Inc. All rights reserved.



A for loop is a section of sequential code that is repeatedly executed a fixed number of times. A for loop has a loop variable with a defined range of values; this range determines the loop's number of repeats or "iterations." In the example on the slide, the loop variable is I, which ranges from 0 to 3, so we will repeat this loop four times. Note that the loop variable is declared as part of the for loop declaration; we don't have to declare it separately. Although we can read the value of I, we are not allowed to assign it to I.

We can use the value of the loop variable within the for loop but not outside; in our example, we use the loop variable I as an index for both the input and output arrays. For loops are accepted by almost all synthesis tools where the number of loop iterations is fixed and known in advance. Where the number of iterations is dynamic, for example, the range is defined using a variable, and the for loop is not synthesizable.

The example on the slide shows that loop variable I is used to index the array OP and IP. The code executes by incrementing the value of I for each loop, i.e., I = 0 for the first iteration, 1 for the second iteration, and so on until four iterations.

The above code reverses the array. When I = 0, OP of 0 equals IP of 3, when I=1, OP of 1 equals IP of 2, when I=2, OP of 2 equals IP of 1 , and when IP = 3, OP of 3 = IP of 0.

## Variable Example

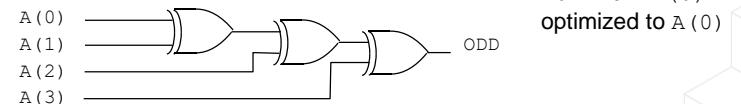
```

library IEEE;
use IEEE.std_logic_1164.all;
entity PARITY is
    port (A : in std_logic_vector(3 downto 0);
          ODD : out std_logic);
end PARITY;

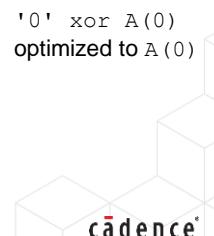
architecture RTL of PARITY is
begin
    PV : process (all)
        variable TMP : std_logic;
    begin
        TMP := '0';
        for I in 0 to 3 loop
            TMP := TMP xor A(I);
        end loop;
        ODD <= TMP;
    end process PV;
end RTL;

```

-- for loop expansion  
 $TMP := '0' \text{ xor } A(0); \text{ when } I = 0$   
 $TMP := TMP \text{ xor } A(1); \text{ when } I = 1$   
 $TMP := TMP \text{ xor } A(2); \text{ when } I = 2$   
 $TMP := TMP \text{ xor } A(3); \text{ when } I = 3$



141 © Cadence Design Systems, Inc. All rights reserved.



We have declared a variable called `TMP`, and we initialize it to '`0`', thus ensuring that our algorithm is independent of the value that `TMP` had when the process was last suspended.

The process then executes a for loop with the loop variable `I` incrementing from 0 to 3. The loop variable is used to index the array `A` to `xor` each element with the variable `TMP`. The expansion of the for loop is shown on the right.

When  $I=0$ , `TMP` equals  $0 \text{ xor } A \text{ of } 0$ ; when  $I=1$ , `TMP` equals  $TM\bar{P} \text{ xor } A \text{ of } 1$ ; here, `TM\bar{P}` is holding the value of  $A \text{ of } 0$  from previous loop execution and hence, becomes,  $A \text{ of } 0 \text{ xor with } A \text{ of } 1$ , and so on.

Finally, having performed the algorithm, the value of `TMP` is assigned to `ODD`.

The architecture synthesized from this code will be a chain of `xor` gates, the first `xor-ing` '`0`' and `A` of `0`, which can be `optimized out`, as we know any value `xor-ed` with `0` gives the same value, the second `xor-ing` the output of the first `xor` gate with `A` of `1`, and so on.

If `TM\bar{P}` were a signal, we would have multiple scheduled assignments to the same target `TM\bar{P}`. If you remember the section on concurrent and sequential assignments, this means that the last signal assignment is the only one that is made.

## Array Attributes

```
signal A : std_logic_vector(3 downto 0);
```

A'left	3
A'right	0
A'high	3
A'low	0
A'length	4
A'range	3 downto 0

```
ATTRIB : process (all)
  variable TMP : std_logic;
begin
  TMP := '0';
  for I in A'low to A'high loop
    TMP := TMP xor A(I);
  end loop;
  ODD <= TMP;
end process ATTRIB;
```

Attributes provide information about an object or type, or components

Extracts information from the declaration and usually return a single value

- Exception: 'range
  - Returns both index bounds *and* direction
  - Used in for loop  
for I in A'range loop
  - Note: 'range is used in for loops instead of 'low to 'high

Code is independent of size or declaration of A

- Helps us to write robust, maintainable code

142 © Cadence Design Systems, Inc. All rights reserved.



Attributes can extract information about objects (such as signals, constants, variables), types, or components. Array attributes are particularly useful since they allow us to create functional code without necessarily knowing an array object's size or index bounds.

An array attribute extracts information from the declaration of the array object.

The various array attributes are listed in the table, 'left gives the index of the leftmost bit, 'right gives the index of the rightmost bit, 'high gives the highest index value the array takes, 'low gives the lowest index value the array can take, 'length, gives the length of the array, 'range returns both index bounds and the direction, so it is ideal to use in for loops. For example, the parity example for loop would be commonly written as for I in the A' range loop. Note: In for loops, we always use 'range instead of 'low to 'high.

We are checking for parity in the example code, as discussed in the previous slide.

'low and 'high are attributes applied to signal A in the for loop. Since A is defined as a std\_logic\_vector(3 downto 0), A'low evaluates to 0 and A'high to 3. Instead of specifying hard-coded values in the for loop, we can specify a range using array attributes. This makes the parity model independent of the size or declaration of signal A. If A changes size or declaration, our parity block will still work. The use of attributes allows us to write robust, maintainable code.

We will see further attributes throughout the course.

## while Loops

```
CLKGEN: process
begin
  while NOW < 10000 ns loop
    CLK <= '1';
    wait for 25 ns;
    CLK <= '0';
    wait for 25 ns;
  end loop;
  -- terminate process
  wait;
end process CLKGEN;
```

```
ENB:
while (ENABLE = '1') loop
  -- sequential statements
end loop ENB;
```

Loops while a condition is true

- Condition checked before every execution of the loop

Sequential statement

The number of iterations is dynamic as it depends on the signal value and is not hard-coded

Not generally synthesizable

Useful in testbenches

- Generation of the clock for a specific time
- NOW function returns the current simulation time



A `while` loop is another sequential statement and, as such, can be used in a process. At the same time, the loop executes when the condition in the `while` statement is true. Condition is first checked, and then the loop is executed. The number of times a `while` loop is executed is dynamic since it often depends upon a signal value. This construct is not generally synthesizable. However, it is a useful construct for testbenches to generate clocks for a specific period.

In the example, we generate a clock with a period of 50 nanoseconds every time we go around the loop. Note how a `while` loop can contain `wait` statements. The function `NOW` is a built-in function that returns the current simulation time. Hence, this process will suspend indefinitely, stopping the clock after we have simulated for 10000 nanoseconds.

## next Loop Statement

```
COUNT := 0;
NO1SA:
for I in A'range loop
  if A(I) = '1' then
    COUNT := COUNT + 1;
  end if;
end loop NO1SA;
```

```
COUNT := 0;
NO1SB:
for I in A'range loop
  if A(I) = '0' then
    next NO1SB;
  else
    COUNT := COUNT + 1;
  end if;
end loop NO1SB;
```

```
COUNT := 0;
NO1SC:
for I in A'range loop
  next NO1SC when A(I) = '0';
  COUNT := COUNT + 1;
end loop NO1SC;
```

Next – sequential statement, used only in loop constructs

- Terminates current iteration and jumps to the beginning of the loop for the next iteration
- Can be used by itself or with a condition
  - Equivalent to an `if` statement in the loop
- Optional loop label
  - Use with labeled loops



next is synthesizable

Synthesis



144 © Cadence Design Systems, Inc. All rights reserved.

`next` is a sequential statement that can only be used inside a `for`, `while`, or infinite loop.

`next` terminates the current loop iteration and jumps to the next iteration. `next` can be used by itself or with a condition.

The example counts the number of 1's in a bit vector by examining each bit, in turn, using a `for` loop, and incrementing `COUNT` if the bit is '`1`'. We can use the following to jump to the next iteration, i.e., to the next bit, when the current bit is '`0`', avoiding incrementing the count.

To aid readability, label your loops, and use this label with the `next` statement.

We see that, in the first code, we loop through the entire range of `A` using `for` loop, check if the bit equals 1; if so, increment the count and go to the next iteration; if not, perform no operation, as no `else` statement following `if`, and go to the next iteration and repeat the process for all iterations specified by the range.

In code 2, we loop through the entire range of `A` using `for` loop. If the bit is zero, we jump to the next iteration without incrementing the count using the `next` statement; if the bit is one, we increment the count and then move on to the next iteration. This repeats for all iterations specified by the range.

Similarly, in code 3, we loop through the entire range of `A` using `for` loop. For each iteration, if the bit is zero, we end the current iteration by using the `next` statement with the condition, then move on to the next iteration; else, if the bit is 1, we increment the count. Typically, all are logically equivalent.

From codes 2 and 3, we observe that `A next with a condition` is equivalent to placing an `if` around the loop statements following the `next`.

## exit Loop Statement

```
ROLTO1A:
for I in 15 downto 0 loop
    DATA := DATA(14 downto 0) & DATA(15);
    if DATA(15) = '1' then
        exit ROLTO1A;
    end if;
end loop ROLTO1A;
```

```
ROLTO1B:
for I in 15 downto 0 loop
    DATA := DATA(14 downto 0) & DATA(15);
    exit ROLTO1B when DATA(15) = '1';
end loop ROLTO1B;
```

exit

- Only allowed in loop constructs
- Jumps to the end of the loop to terminate the loop
- Optional condition
  - Equivalent to an `if` statement in the loop
- Optional loop label
  - Use with labeled loops



exit is synthesizable

Synthesis



145 © Cadence Design Systems, Inc. All rights reserved.

`exit` is allowed only in `for`, `while`, and infinite loop constructs.

`exit` terminates a loop and jumps to the end of the loop. `exit` can be used by itself or with a condition. To aid readability, label your loops when using `exit` statements, and use this label with the `exit`.

The example rotates a 16-bit vector to the left until the MSB is '`1`' or until the entire vector has been rotated if a vector contains only '`0`'s. `exit` is used to terminate the loop when `MSB = '1'`. The second code is logically equivalent to the first code, except that it uses `exit` with conditions instead of placing it around the loop statements following the `exit`.

## exit Loop Statement (continued)

```

OUTER : for I in 0 to 7 loop
    INNER : for J in 0 to 7 loop
        -- statements

        exit INNER when EXIT_INNER = '1';
        exit OUTER when EXIT_OUTER = '1';

        -- statements
    end loop INNER;

end loop OUTER;

```

- By using loop labels, any level of the nested loop can be exited
- Code uses loop labels INNER and OUTER to exit out of the two nested loops
- A high on EXIT\_INNER exits the INNER loop
- A high on EXIT\_OUTER exits the OUTER loop



exit is synthesizable

Synthesis



146 © Cadence Design Systems, Inc. All rights reserved.

Using loop labels, any level of the nested loop can be exited.

The example code contains two nested loops named with loop labels INNER and OUTER. We can exit any of these loops by using exit with condition construct. Here, whenever EXITINNER and EXITOUTER go high, the control exits out of inner and outer loops, respectively.

## Sequential Statements: Summary

1. When is a variable's value updated due to an assignment?
2. What is the scope of a variable?
3. If a signal is declared as:

```
signal B_BUS : std_logic_vector(6 downto 2);
```

then what are the values of:

- B\_BUS'**low**
- B\_BUS'**high**
- B\_BUS'**length**
- B\_BUS'**reverse\_range** (take a guess!)



Let's take a short quiz on this module.

When is a variable's value updated as a result of an assignment? What is the scope of a variable? Write down the values for the following array attributes.

## Sequential Statements: Summary (Solutions)

1. When is a variable's value updated due to an assignment?
  - Immediately
2. What is the scope of a variable?
  - The process in which it is declared
3. If a signal is declared as:

```
signal B_BUS : std_logic_vector(6 downto 2);
```

then what are the values of:

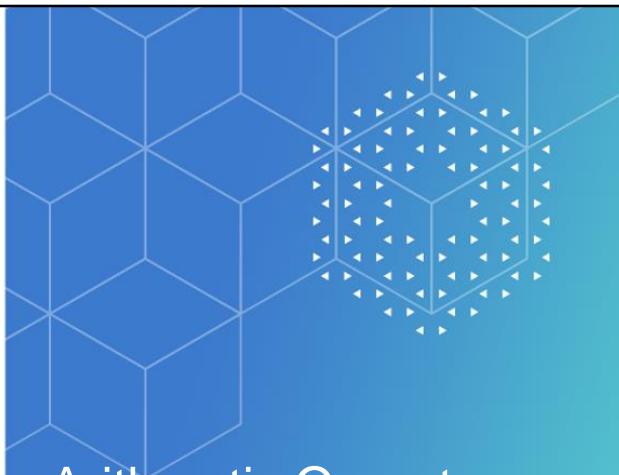
- B\_BUS'low = 2
- B\_BUS'high = 6
- B\_BUS'length = 5
- B\_BUS'reverse\_range = (2 to 6)



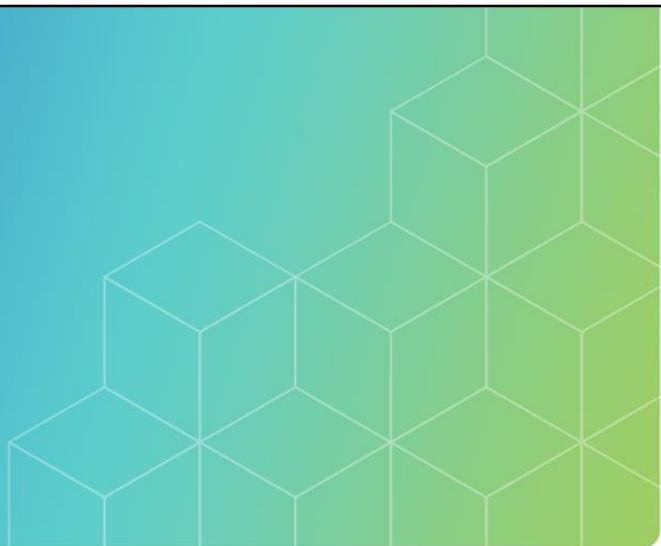
Here is the solution. The variable value is updated immediately due to an assignment.

The scope of the variable is the process in which it is declared.

For array B\_BUS with index 6 downto 2, 'low equals 2, 'high equals 6, 'length equals 5, 'reverse\_range equals 2 to 6.



## Arithmetic Operators



**Module** **10**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

We have already discussed logical and relational operators. In this section, we will be exploring the issues that surround the use of arithmetic operators.

## Module Objectives

In this module, you

- List and use all built-in arithmetic operators
- Define arithmetic operators on signed and unsigned types
- Override the operators for other data types, called operator overloading
- Implement code using functions and procedures
- List some standard packages containing overloaded operators
- List packages of arithmetic operators
- Define and use vector arithmetic



At the end of this module, you will be able to list and use all built-in arithmetic operators and define arithmetic operators on signed. Unsigned types, override the operators for other data types, called operator overloading, implement code using functions and procedures, list some standard packages containing overloaded operators, list packages of arithmetic operators, and define and use vector arithmetic.

## Arithmetic Operators

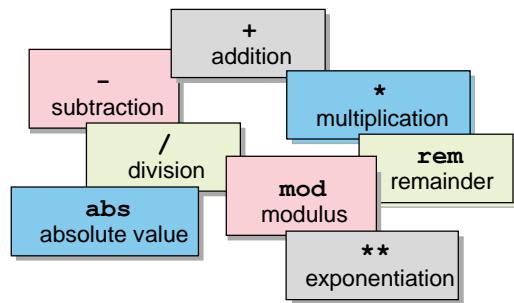
Predefined for

- integer
- real (except mod and rem)
- Physical types (example, time)

Not defined for

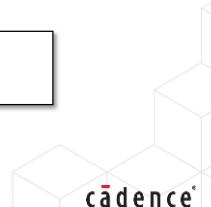
- bit\_vector
- std\_logic\_vector

Generally, operands must be of the same type



```
signal A, B, OP : integer;
OP <= A + B; --adds two integers and returns integer type
```

151 © Cadence Design Systems, Inc. All rights reserved.



The arithmetic operators like addition, multiplication, division, etc., shown on the slide are built-in, or predefined, in VHDL. Unfortunately, they are only defined for integer, real and physical types. They are not defined for the built-in bit\_vector nor the IEEE standard 1164 types such as std\_logic\_vector.

Generally, the operands must be of the same type, and, of course, normal strong typing rules apply in assignment statements. So, if A, B, and OP are defined as integers, VHDL gives you the ability to write OP equals A + B; The operator “+” is used here with two integer operands and returns a value of type integer.

## Adder Example

```

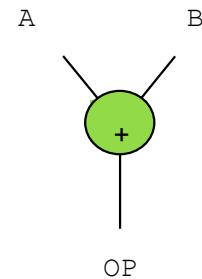
entity ADDINT is
  port( A, B : in  integer range 0 to 7;
        OP   : out integer range 0 to 15);
end ADDINT;

architecture RTL of ADDINT is
begin

  OP <= A + B;

end RTL;

```



The use of constrained integers:

- Help in error-checking
- Give more efficient synthesis results

152 © Cadence Design Systems, Inc. All rights reserved.



The code on the previous slide defined A, B, and OP as unrestricted integers. In practice, it is a good idea to restrict the range of any integer you use, as shown here. Notice, by the way, how the syntax is used to restrict the range of an integer:

```
signal A_INT: integer range 0 to 7;
```

differs from the syntax used to restrict the size of a vector:

```
signal A_BUS: std_logic_vector(3 downto 0);
```

We constrain integers for two main reasons:

1. To give better error checking in simulation. If your adder is only designed for values in the range 0-7, then by constraining the inputs to this range, if a user tries to instantiate your adder with signals outside the range, they will get compilation errors.
2. To give more efficient synthesis results. An unconstrained integer synthesizes to a 32-bit signed vector. Your synthesis tool may be able to optimize this length to something smaller, but by using tightly constrained ranges, we can at least give the synthesis tool a better starting position for optimization.

## Arithmetic of Time

```
-- constant declaration

constant CYCLE : time := 50 ns;

begin
  process
    begin
      wait for 50 ns;
      ...
      wait for CYCLE;
      ...
      wait for 5 * CYCLE;
      ...
      wait for CYCLE * 5.5;
      ...
      wait for CYCLE/2;
      ...
    end process;
```

### Used for

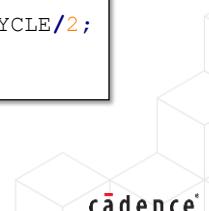
- Testbenches
- Cell delays

### Multiplication/division

- Multiply/divide by integer/real
- Returns type time

```
-- Declare constants for common
-- arithmetic expressions and use
-- in wait to avoid repetition

constant HALFCYCLE : time := CYCLE/2;
...
wait for HALFCYCLE;
```



In testbenches, you often want to operate on values of type `time`. In this example, we define a constant called `CYCLE` of type `time` and value 50 ns.

The syntax for a constant declaration is very similar to that of a signal declaration, except that we need to define a value for the constant, 50 nanoseconds in this case. All references to the constant name then access the value defined in the constant declaration.

Notice how `CYCLE` is multiplied first by an `integer` and then by a `real` number. Later on, it is divided by an `integer`. In all cases, the value returned by the operator is of type `time`.

Note that using arithmetic expressions in a `wait for` statement is inefficient, particularly if the `wait` statement is executed frequently. It would be more efficient to declare constants for common arithmetic expressions and use these in the `wait` statement. This avoids repeated evaluation of the arithmetic expression, for example:

```
constant HALFCYCLE: time:= CYCLE/2;
...
wait for HALFCYCLE;
```

## Operations on std\_logic Array Types

A bit of an issue!

- Logical operators
  - Defined in std\_logic\_1164 package
- Relational
  - By default, different vector lengths give unusual results
- Arithmetic
  - Not defined by default in the language
- std\_logic\_vector
  - How do we represent positive and negative values?

Let's look at some workarounds

- Language background first

154 © Cadence Design Systems, Inc. All rights reserved.



All operations on std\_logic array types or any array type are a bit of an issue!

Logical operators (and, not, nor, etc.) are defined in the std\_logic\_1164 package alongside the declaration of std\_logic and std\_logic\_vector.

We have already seen how the built-in relational operators can return “strange” answers when the operands are of different lengths. This is because VHDL does not recognize that vectors represent numerical values.

Arithmetic operators on vectors are not defined at all.

We need a way to distinguish positive and negative values before we can work with arithmetic or relational operators on vector types.

We will be looking at a workaround for these problems.

## signed and unsigned Types

```
variable A, B, SUM : std_logic_vector(3 downto 0);
...
A := "1001"; -- no numerical representation
B := "0011"; -- no numerical representation
SUM := A + B; -- ?
if A > B then -- true for default operator
...

```

```
variable A, B, SUM : signed(3 downto 0);
...
A := "1001"; -- -7
B := "0011"; -- +3
SUM := A + B; -- "1100" = -4
if A > B then -- should be false
...

```

```
variable A, B, SUM : unsigned(3 downto 0);
...
A := "1001"; -- +9
B := "0011"; -- +3
SUM := A + B; -- "1100" = +12
if A > B then -- should be true
...

```

- Unsigned data
  - Positive values only
  - Binary representation
  - 4-bit vector represents 0 to 15
- Signed data
  - Positive and negative values
  - 2's complement representation
  - 4-bit vector represents -8 to +7
- Arithmetic, relational, and conversion operators behave differently for signed and unsigned data
  - Overcome by using signed and unsigned types, which are arrays of std\_logic



Let us look at the usage of operators concerning signed and unsigned types. One key issue with the use of arithmetic or relational operators on array types is the representation of positive and negative data. Positive or unsigned data is easy to represent – we can use simple binary values, and a 4-bit can be used to represent values between 0 to 15. But the representation of data that can take both positive and negative values or signed data is more difficult. We have several conventions for representing signed data in hardware design, one of the most common being 2's complement. 4-bit represents values between -8 to 7, where MSB represents the sign.

In 2's complement, the value of the Most Significant Bit or MSB of a vector tells us whether the value is positive, when MSB = '0' or negative, when MSB = '1'. To take a positive value (example , 3 = "0011") and convert it to its negative, we invert the vector value and add one ( -3 = "1101").

Arithmetic, relational and conversion operations will behave differently depending on if data is signed or unsigned. Therefore, we need some way of distinguishing between signed and unsigned data and to overload the arithmetic, relational and conversion operators accordingly.

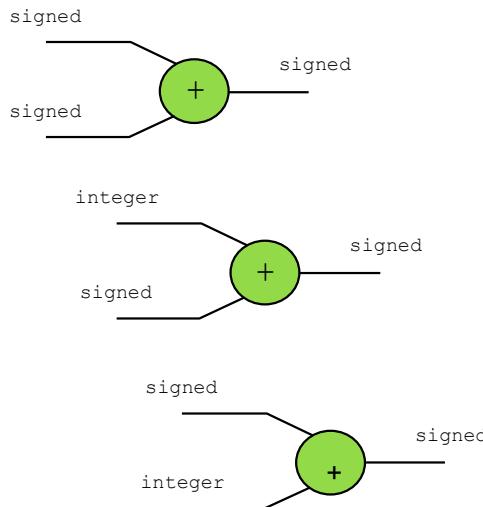
To do this, we declare two new types – signed and unsigned. These have identical type declarations as std\_logic\_vector, i.e., they are both arrays of std\_logic but are treated differently by VHDL operators.

In the code given on slide, we see that in case one, A, B, and SUM are all declared as std\_logic\_vector, which holds array value and has no numeric representation.

In code-2, A, B, and SUM are all declared as signed vectors. Hence, MSB denotes the sign of the vector; if '1', it denotes a negative value, and if '0', it denotes a positive value. Hence, A equals 1 0 0 1 denotes -7 and B equals 0 0 1 1 denotes +3 , hence, it's sum A + B equals -7 + 3=-4. Also, A greater than B yields false.

In code three, for the same binary bits assigned to A and B, they take the value +9 and +3, respectively, as they are declared unsigned types. Hence, their sum is nine, which is different from the previous case. Also, condition A greater than B yields true. From two and three, we see that for the same binary values assigned to A and B, results differ cause of their sign representation.

## Operator Overloading



Operators can be redefined to work with additional types

- *Operator overloading*
- Redefined using functions

Redefine for combinations of

- signed, unsigned, and std\_logic\_vector vector types
- integer types

Called in context, the context has to be unique



Must be synthesizable  
for use in RTL code  
Synthesis



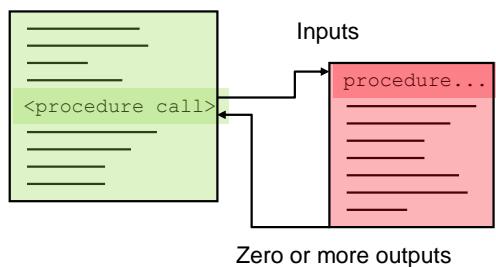
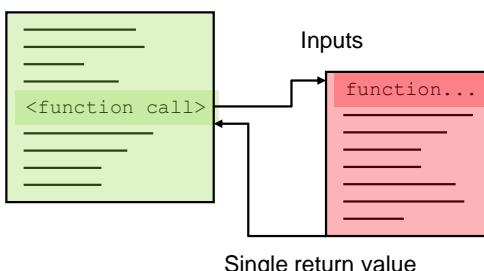
156 © Cadence Design Systems, Inc. All rights reserved.

Operators can be redefined to work with additional data types called operator overloading. Operator overloading is a feature of sophisticated programming languages, including VHDL. It is the ability to reuse a predefined operator and re-define its functionality, so it works with different data types.

As shown, it is redefined for combinations of signed, unsigned, and std\_logic\_vector types and integer types.

The compiler is clever enough to work out exactly which version of an operator is required from the context in which it is called. Contexts have to be unique. Note that the function should be synthesizable for its use in RTL code.

## Functions and Procedures



### Subprograms

- Encapsulate portions of repeated code
- Sequential statements
- Execute in sequence like “software”
- Called from the main body of the code, architecture

### Function

- Multiple inputs, the single return value
- Can only be used as part of an assignment

### Procedure

- Zero or more inputs/outputs
- Is a concurrent or sequential statement



Functions and procedures are two types of subprograms. They are used to encapsulate repeated code which can be called when required. They contain sequential statements that execute in sequence, just like software or the code inside your processes. Both are called from the main body of your code, such as architecture.

A function is defined with input parameters only. You call it by providing values for these parameters, such as a signal or variable values, which are operated upon by the function. The result is a single return value which you can then use in your code.

A procedure works similarly but is defined as having both input and output parameters; you call it by providing values for all its input parameters and telling which internal signals or variables the output computed values should be assigned.

Functions and procedures are described in more detail in the modules.

## Packages of Overloaded Operators

```

package NUMERIC_STD is
  ...
  function "+" (L: UNSIGNED; R: UNSIGNED) return UNSIGNED;    -- 1
  function "+" (L: UNSIGNED; R: NATURAL)   return UNSIGNED;    -- 2
  function "+" (L: NATURAL;  R: UNSIGNED) return UNSIGNED;    -- 3
  function "+" (L: SIGNED;   R: SIGNED)   return SIGNED;      -- 4
  ...
end NUMERIC_STD;    package body NUMERIC_STD is
  ...
  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
    -- code to perform addition of two unsigned vectors and return unsigned vector
    end "+";
  ...
  function "+"(L: UNSIGNED; R: NATURAL) return UNSIGNED is
    -- code to perform addition of an unsigned vector and a natural number,
    -- and return an unsigned vector
    end "+";
end NUMERIC_STD;

```

- Functions that overload operators defined in a package
- Vector arithmetic can be performed by referencing the package



We can reuse existing operators and re-define them to work with array types, like `signed` and `unsigned`. We can also re-define them to work with combinations of array types and integer types. This will allow us to add an `integer` to a `signed` array and an `unsigned` array, as shown. This allows us to describe operations in more flexible ways.

The definitions for the overloaded operations are usually placed in packages so they can be shared amongst many entities/architectures by referencing the package.

Remember, for subprograms; packages can only contain the declaration of the function or procedure. The subprogram body, i.e., code describing the functionality of the subprogram must be placed in the package body.

Note: the function name is the same as the operator to be overloaded, in this case, `+`, but is enclosed in double quotes to indicate this is overloading of an existing operator. This allows us to use the function as an operator, `A + B`, rather than a function call `+ (A, B)`.

See the Functions and Procedures module for more information on subprograms.

## Operator Overloading Example

```

package NUMERIC_STD is
  ...
  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;    -- 1
  function "+"(L: UNSIGNED; R: NATURAL)  return UNSIGNED;    -- 2
  function "+"(L: NATURAL; R: UNSIGNED) return UNSIGNED;    -- 3
  function "+"(L: SIGNED;   R: SIGNED)   return SIGNED;      -- 4
  ...
end NUMERIC_STD;

library IEEE;
use IEEE.NUMERIC_STD.all;
entity OVERLOADED is
  port (A_BUS, B_BUS : in  UNSIGNED(3 downto 0);
        A_INT, B_INT : in  NATURAL range 0 to 15;
        Y_BUS         : out UNSIGNED(3 downto 0);
        Y_INT         : out NATURAL range 0 to 31);
end OVERLOADED;
architecture A of OVERLOADED is
begin
  Y_INT <= A_INT + B_INT;  -- a
  Y_BUS <= A_BUS + B_BUS; -- b
  Y_BUS <= A_BUS + A_INT; -- c
  Y_INT <= A_BUS + B_INT; -- d
end A;

```

159 © Cadence Design Systems, Inc. All rights reserved.

For each expression:

- Is it allowed by default in the language?
- Otherwise, is a single overloaded function available?



Package numeric\_std contains several overloaded function definitions for the operator plus.

Each of these functions will have a unique definition, which is not shown on the slide for reasons of space. These definitions will reside in the package body. If we make this package visible to our model using a use statement, we can write code using these definitions.

Which function definitions are used by calls to the “+” operator used in the four statements?

## Solution: Operator Overloading Example

```

package NUMERIC_STD is
  ...
  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;  -- 1
  function "+"(L: UNSIGNED; R: NATURAL)  return UNSIGNED;  -- 2
  function "+"(L: NATURAL; R: UNSIGNED) return UNSIGNED;  -- 3
  function "+"(L: SIGNED;  R: SIGNED)   return SIGNED;    -- 4
  ...
end NUMERIC_STD;

library IEEE;
use IEEE.NUMERIC_STD.all;
entity OVERLOADED is
  port (A_BUS, B_BUS : in  UNSIGNED(3 downto 0);
        A_INT, B_INT : in  NATURAL range 0 to 15;
        Y_BUS         : out UNSIGNED(3 downto 0);
        Y_INT         : out NATURAL range 0 to 31);
end OVERLOADED;
architecture A of OVERLOADED is
begin
  Y_INT <= A_INT + B_INT;  -- a default integer arithmetic
  Y_BUS <= A_BUS + B_BUS; -- b not default, but matches NUMERIC_STD signature 1
  Y_BUS <= A_BUS + A_INT; -- c not default, but matches NUMERIC_STD signature 2
  Y_INT <= A_BUS + B_INT; -- d illegal: not default, no matching signature
end A;

```

160 © Cadence Design Systems, Inc. All rights reserved.



Package numeric\_std contains several overloaded function definitions for the operator “+.”

Each of these functions will have a unique definition that is not shown on the slide for reasons of space. These definitions will reside in the package body. If we make this package visible to our model using a use statement, we can write code using these definitions.

Which function definitions are used by calls to the “+” operator used in the four statements?

Assignment a is allowed by default in the language (integer arithmetic).

Assignment b is not allowed by default (vector arithmetic). The type signature for the expression is left-hand side (LHS) = unsigned (A\_BUS), right-hand side (RHS) = unsigned (B\_BUS) and return type = unsigned (Y\_BUS). The overloaded function matching this signature is 1.

Assignment c is not allowed by default (includes vectors). Type signature is RHS = unsigned (A\_BUS); LHS = natural (A\_INT); return = unsigned (Y\_BUS) matching function 2.

Assignment d is not allowed by default (includes vectors). Type signature is RHS = unsigned (A\_BUS); LHS = natural (B\_INT); return = natural (Y\_INT). There is no matching function for this type of signature. Therefore, this assignment gives us a compilation error.

## Standard Packages: IEEE 1076.3

numeric\_std

- IEEE standard arithmetic package defines overloaded arithmetic, relational and conversion operators
- Overloaded operators on SIGNED and UNSIGNED and integer types only
  - UNSIGNED for binary arithmetic
  - SIGNED for 2's complement arithmetic
- Also, conversion functions, utilities
- Referenced from the IEEE library

numeric\_bit

- IEEE standard arithmetic package
- Overloaded operators on signed and unsigned operators for **bit-based** arrays
- Rarely used, as bit doesn't hold U, X, Z, etc.
- Both are pre-compiled

The package listed in your *VHDL Reference Guide*

```
-- numeric_std examples
function "+" (L, R: UNSIGNED) return UNSIGNED;
function "+" (L, R: SIGNED) return SIGNED;
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
```

```
-- referencing numeric_std
library IEEE;
use IEEE.numeric_std.all;
```

161 © Cadence Design Systems, Inc. All rights reserved.



numeric\_std is a standard arithmetic package that defines arithmetic, relational and conversion operations for signed and unsigned types, which are arrays of std\_logic, like std\_logic\_vector. Operations are overloaded for combinations of signed, unsigned, and integer types; few are listed on the slide. This package only works with signed, unsigned, or integer-related types.

An IEEE package called numeric\_bit also defines signed and unsigned operations for bit-based arrays like bit\_vector. In practice, the numeric\_bit package is rarely used since the bit type does not include suitable values for un-initialized, high-impedance, unknown, and different strengths.

Both arithmetic packages are referenced from the IEEE library, which is supplied pre-compiled with your simulator; you do not need to compile these packages yourself.

The *VHDL Reference Guide* contains a listing of the IEEE numeric\_std package.

## Vector Arithmetic

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity ADDVEC is
    port (A, B : in unsigned(3 downto 0);
          OROLL : out unsigned(3 downto 0);
          OCARRY : out unsigned(4 downto 0));
end ADDVEC;

architecture RTL of ADDVEC is
begin
    OROLL <= A + B;           -- Adding two 4-bit numbers
                               -- gives rollover on the 4-bit result

    OCARRY <= ('0' & A) + ('0' & B); -- Making inputs 5 bits saves
                                      -- carry in 5-bit result

end RTL;

```

Reference package to allow arithmetic on vectors

- Vectors must be defined as signed or unsigned

Vector widths must be considered

- Size of vectors on either side of the assignment must match

```

-- numeric_std declaration
function "+" (L, R: UNSIGNED) return UNSIGNED;
-- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).

```

162 © Cadence Design Systems, Inc. All rights reserved.



These arithmetic packages allow you to apply arithmetic and relational operators to array types.

numeric\_std only allows operations on signed or unsigned types. Therefore you must either declare all your vector objects as being of these types or use type conversion to convert signed/unsigned to std\_logic\_vector and vice-versa (see next slides).

Note that the overloaded operators have specific rules on vector length (see the package listings in the Reference Guide for details). By default, in numeric\_std, the "+" operator returns a vector of size equal to the longest input (operand). Adding two 4-bit vectors returns a 4-bit result. For example, adding "1111" and "0001" gives a "0000", i.e., the result rolls over.

If we wish to save the full width of the result, we must expand the size of one or both of the input operands to make the result the correct size. In the example above, the unsigned vectors A and B are expanded to 5 bits using concatenation to add a '0' in the most significant bit. This allows the carry to propagate to the extra bit, making our example sum "01111" + "00001" = "10000".

## Arithmetic Package Use

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

architecture BEHAVIORAL of SEQGEN is
  signal A, OP : unsigned(3 downto 0);
begin
  ...
  process (all)
  begin
    if (A >= 9) then      ← Compare vector
      OP <= "0000";       ← Assignment
                           must be vector
    else
      OP <= A + 1;        ← Add vector and integer
    end if;
  end process;
  ...

```

- Defines relational and arithmetic operators
- For combinations of signed, unsigned, and integer types
- For example, can use:
  - Relational operators to compare integer and vector types
  - Arithmetic operators to add integer and vector types
- Assignment types must still match
  - Cannot assign an integer to a vector



Arithmetic packages define relational and arithmetic operators. Note that the packages allow you to mix array and integer types in relational and arithmetic operations examples. They permit you to compare a vector to an integer and add an integer to a vector.

Assignment to a vector must still be done with a vector value, not an integer. To assign an integer to a vector object would require a type conversion function. Some are included in the arithmetic package. The Advanced Data Types module examines type conversions in more detail.

## Mixing std\_logic Array Types

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
architecture BEHAVIORAL of CRT is
    signal ASLV : std_logic_vector(7 downto 0);
    signal BSGN : signed(7 downto 0);
    signal CUSN : unsigned(15 downto 0);
begin
    BSGN <= ASLV; X -- ERROR type mismatch
    BSGN <= signed(ALSV);
    ASLV <= std_logic_vector(BSGN); Convert std_logic_vector to signed and vice-versa
    CUSN <= unsigned(ALSV) and unsigned(BSGN);
end BEHAVIORAL;
Convert signed and std_logic_vector to unsigned

```

- signed, unsigned, and std\_logic\_vector are all arrays of std\_logic and cannot be assigned to each other
- VHDL treats these as *closely-related* types
- Allows simple type conversion  
 $T \leq \langle T \text{ type} \rangle(S);$
- Array sizes must still match



Not applicable to other types  
More details in Module 19 Advanced Data Types

Beware



164 © Cadence Design Systems, Inc. All rights reserved.

Although signed, unsigned, and std\_logic\_vector are all declared as arrays of std\_logic, the strict typing of VHDL does not allow us to assign these types to each other freely.

However, VHDL recognizes that these types are very similar, closely-related in language terms and provides a mechanism to convert between the types: easily

```
target <= <target_type>(source);
```

where target and source are different closely-related types, for example, unsigned, signed, std\_logic\_vector types. The variable assignment could also be used, assuming the target is a variable. Array sizes must still match.

**WARNING:** This form of type conversion cannot be used to convert between other types, for example, integer to std\_logic\_vector or std\_logic\_vector to string. These types are not closely related.

More details on types and type conversion are in the Advanced Data Types module.

## Solutions 1: Arithmetic Operators

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity VECADD is
  port (A, B : in std_logic_vector(3 downto 0);
        OP    : out std_logic_vector(3 downto 0));
end VECADD;

architecture RTL of VECADD is
begin
  OP <= A + B;      
end RTL;

```

- Use signed/unsigned types everywhere

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity VECADD is
  port (A, B : in unsigned(3 downto 0);
        OP    : out unsigned(3 downto 0));
end VECADD;

architecture RTL of VECADD is
begin
  OP <= A + B;      
end RTL;

```

165 © Cadence Design Systems, Inc. All rights reserved.



In the given code, as the `numeric_std` package defines arithmetic operations for signed and unsigned types only, then one easy solution is to use only signed and unsigned types in our code where we need to carry out arithmetic operations.

In the first code, we see that `A`, `B`, and `OP` are all declared as `std_logic_vector`, on which we can't perform any arithmetic operations. This can be overcome by declaring `A` and `B` as unsigned data types, as shown in code 2.

## Solutions 2: Arithmetic Operators

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity VECADD is
    port (A, B : in std_logic_vector(3 downto 0);
          OP    : out std_logic_vector(3 downto 0));
end VECADD;

architecture RTL of VECADD is
begin

P1: process (all)
    variable A_US, B_US, OP_US: unsigned(3 downto 0);
begin

    A_US := unsigned(A);
    B_US := unsigned(B);

    OP_US := A_US + B_US;

    OP <= std_logic_vector(OP_US);

end process P1;
end RTL;

```



- Use signed/unsigned variables
- Convert locally in processes
  - Type marks for closely-related types

166 © Cadence Design Systems, Inc. All rights reserved.



From the example, we see that if we need to use `std_logic_vector` types for ports, then one solution for arithmetic code is to convert the `std_logic_vector` inputs, A and B in this case, in assignment to signed or unsigned type variables, `A_US` and `B_US`; perform the arithmetic operations like addition using variables and then convert back the result, `OP_US` obtained in unsigned format to `std_logic_vector` in assignment to outputs, OP.

## Solutions 3: Arithmetic Operators

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity VECADD is
    port (A, B : in std_logic_vector(3 downto 0);
          OP : out std_logic_vector(3 downto 0));
end VECADD;

architecture RTL of VECADD is
begin

    OP <= std_logic_vector( unsigned(A) + unsigned(B) );  ✓

end RTL;

```

- Convert locally to each operator

167 © Cadence Design Systems, Inc. All rights reserved.



The final option, which is the most common and the most difficult, is to convert `std_logic_vector` signals to the required type local to each arithmetic operator. This works best when you have a few arithmetic operations in your code.

In the code, we see that `A` `B`, which is of `std_logic_vector`, is type converted to `unsigned` and added, the result of which is of `unsigned` type is converted back to `std_log_vector` and is assigned to the output.

## Arithmetic Operators: Summary Quiz

1. Rewrite the additions below so that they will compile using overloaded operators from the `numeric_std` package.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
signal AUS, CUS : unsigned(7 downto 0);
signal AVEC, BVEC, CVEC : std_logic_vector(7 downto 0);
signal BNAT : natural range 0 to 15;
...
AUS <= BNAT + CVEC;

AUS <= BVEC + CUS;

AUS <= BVEC + CVEC;

AVEC <= BVEC + CVEC;
```



Let us take a short quiz on this module. In this code, rewrite the additions above so that they will compile using overloaded operators from the `numeric_std` package.

## Arithmetic Operators: Summary Quiz (Solutions)

Just do type conversions to match the signature styles available in the `numeric_std` package.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

...
signal AUS, CUS : unsigned(7 downto 0);
signal AVEC, BVEC, CVEC : std_logic_vector(7 downto 0);
signal BNAT : natural range 0 to 15;
...
AUS <= BNAT + unsigned(CVEC);

AUS <= unsigned(BVEC) + CUS;

AUS <= unsigned (BVEC) + unsigned (CVEC);

AVEC <= std_logic_vector(unsigned(BVEC) + unsigned (CVEC));
```

169 © Cadence Design Systems, Inc. All rights reserved.



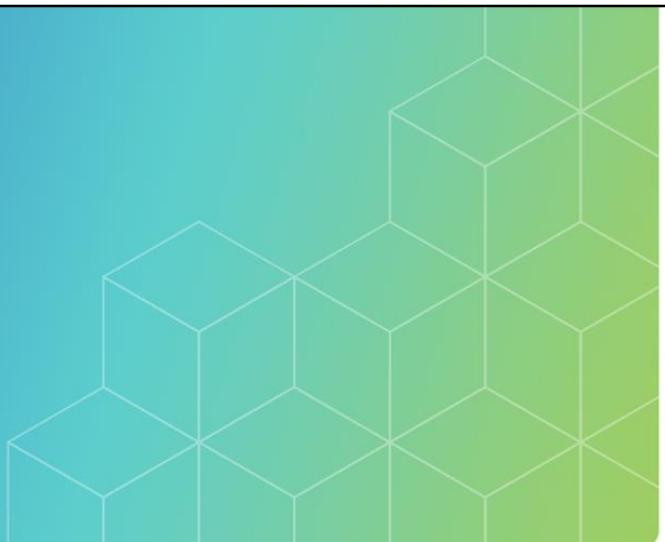
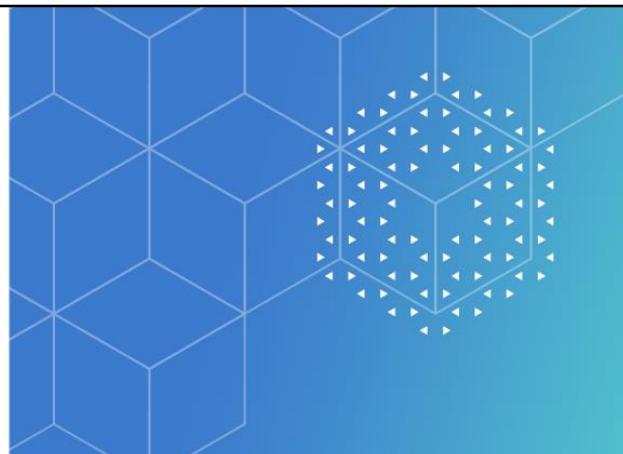
Here is the solution. Just do type conversions as shown on the slide to match the signature styles available in the `numeric_std` package.

```
AUS <= BNAT + unsigned(CVEC);
```

```
AUS <= unsigned(BVEC) + CUS;
```

```
AUS <= unsigned(BVEC) + unsigned(CVEC);
```

```
AVEC <= std_logic_vector( unsigned(BVEC) + unsigned(CVEC) );
```



## VHDL Coding Styles

**Module** **11**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

We have mentioned that there are different coding styles we can employ depending upon the amount of detail we want to describe in our model. In this section, we will look more closely at this subject.

## Module Objectives

In this module, you

- Code, analyze, and debug testbenches in the behavioral model
- State different coding styles in VHDL and locate them in the design flow
- Differentiate between various coding styles and identify their applications
- Write, analyze and debug RTL code for a given design
- Write code in structural/gate-level style



By the end of this module, you will be able to

- Code, analyze, and debug testbenches in the behavioral model
- List different coding styles in VHDL and the place in the design flow where it is used
- Summarize the differences between different coding styles and identify their application
- Write, analyze and debug RTL code for a given design
- Write code in structural/gate-level style

## Behavioral Code

Only addressing function, not implementation

Can use the full scope of language

- No restrictions on data types
- Typically, a few extensive processes
- No restrictions on process style

Behavioral divide by 2

OP  $\leq$  DATA / 2;

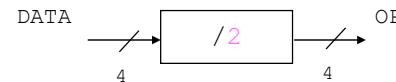
Not required to be synchronous

- Trigger activity on the edge of any signal
- Explicit delays in synchronizing data

Occasionally for initial block/system modeling?

- Create a functional model of the spec to highlight ambiguities and errors
- Experiment with different architectures and partitioning for implementation

Important for testbench design, simulation models, etc.



172 © Cadence Design Systems, Inc. All rights reserved.

For behavioral models, we are only interested in function, not implementation, i.e., what the design must do, not how it will do.

Unlike other coding styles, there are no restrictions for writing behavioral code – all language constructs and data types and any form of the process can be used.

We can use sensitivity lists and wait on constructs or concurrent statements to trigger activity whenever a signal changes value. Therefore, we do not have to make the model synchronous, so we do not need to specify clock schemes, registers, or resets. We can also use explicit wait for statements to synchronize data.

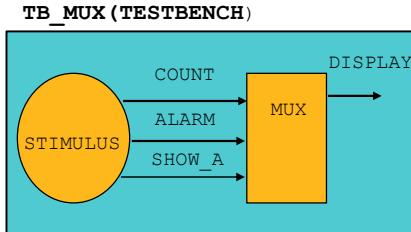
Occasionally, behavioral VHDL modeling is used for system-level design to:

- Create a functional model of the specification to highlight ambiguities and errors.
- Experiment with different architectures and partitioning for implementation.
- Create libraries of test data – both stimuli and expected responses – which can be used to verify RTL models.

Full behavioral-level system modeling with VHDL is rarely used, but behavioral modeling techniques are very important for testbench design, simulation models, etc. In the above example, the code only describes that OP equals data by two and doesn't say anything about how division is performed.

## Simple Testbench Organization (Review)

Used to drive stimulus and verify the design



"Empty" entity

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity TB_MUX is
end TB_MUX;

architecture BENCH of TB_MUX is

component MUX
  port (COUNT : in std_logic_vector(3 downto 0);
        ALARM : in std_logic_vector(3 downto 0);
        SHOW_A : in std_logic;
        DISPLAY : out std_logic_vector(3 downto 0));
end component;

signal TIME_DATA, ALARM_DATA,
       DISPLAY : std_logic_vector(3 downto 0);
signal SHOW_A : std_logic;

begin

  DUT : MUX port map (TIME_DATA, ALARM_DATA,
                       SHOW_A, DISPLAY);

  STIMULUS: process
  ...
  end process STIMULUS;

end BENCH;
  
```

Component declaration for DUT

Signals connecting testbench to DUT

Fully connected DUT component instantiation with a signal for every port

Stimulus process

173 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

The testbench is used to drive stimulus to and verify the DUT. Typically, almost 50% of the time you will spend writing VHDL on a project will be taken up with coding your testbench.

Simple testbenches are useful for small, straightforward designs or for checking out an idea or an unknown piece of VHDL code.

Here are a few important points to note:

- The testbench entity is “empty,” i.e., it has no port list. The testbench is at the top of your design hierarchy and has no external connections.
- The Design Under Test (DUT) must be fully connected to the testbench, with a signal in the component instantiation port map for each port in the declaration. Even if the port is unused (for example, an output port not connected to a process), you should still connect a signal to the port.

Testbenches tend to start simple but quickly become sophisticated.

In the example on the slide, we test the DUT, MUX, which has count, alarm, and SHOW\_A as inputs and DISPLAY as output. Testbench TB\_MUX is built to verify the MUX design. In the testbench, component MUX is declared and instantiated, and each port of the component is connected to a TB signal, which is declared in the architecture of the testbench. Stimulus to the DUT is driven via these signals in STIMULUS. The output of the DUT can then be seen on signal DISPLAY.

## Stimulus Generation

```

constant CYCLE : time := 50 ns;

-- signal with initial value
signal CLK : std_logic := '1';

begin
    -- concurrent clock generator
    CLK <= not CLK after CYCLE;

    process
    begin
        wait for CYCLE;
        ...
        wait for 5 * CYCLE;
        ...
        wait for CYCLE/2;
        ...
    end process;

    wait for CYCLE/2;           -- Instead of this ...
                                -- write this ...

    constant HALFCYCLE : time:= CYCLE/2;
    wait for HALFCYCLE;

```

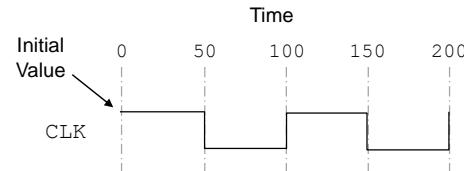
Multiply/divide type time by type integer or real

- Returns type time

Use with `wait for` to create delays between stimulus values

Create assignment delays with "after"

- Concurrent clock generator
- Clock must be initialized
  - Initial value in the declaration



Here we define a constant called CYCLE of type time and value 50 nanoseconds. CYCLE can be multiplied and divided by integers. In all cases, the resulting value is of type time and can be used in a `wait for` statement to insert varying delays between stimuli.

Note: Using arithmetic expressions in a `wait for` statement is inefficient, particularly if the `wait` statement is executed frequently. Repeated evaluation of an arithmetic expression can be avoided by declaring a constant. i.e., instead of writing `wait for CYCLE by 2`, we can declare a constant HALFCYCLE, which is half of the CYCLE value, and then call `wait on` HALFCYCLE wherever required.

An `after` clause introduces a delay to an assignment. We use this in the concurrent clock generator. When CLK changes value, the generator statement is executed, and the next value for CLK is assigned. Without the `after` clause, we would have an infinite loop, but after scheduling the assignment in the future. The `after` is non-synthesizable.

Note that the declaration of the signal CLK includes a value ('1'). This is an initial value for the signal, i.e., CLK will initialize to '1' at the beginning of the simulation. Without this, CLK would initialize the "left-most" value in its type declaration 'U' for `std_logic`. Inverting 'U' with the `not` operator of the clock generator would result in another 'U,' leaving our CLK signal stuck at uninitialized. This code generates a clock signal with a period controlled by the CYCLE value.

## Simple Behavioral Stimulus

```

...
signal TIME_DATA, ALARM_DATA,
       DISPLAY : std_logic_vector(3 downto 0);
signal SHOW_A : std_logic;

constant PERIOD : time := 10 ns;
...

SIMPLE_STIMULUS: process
begin
  TIME_DATA <= "0000";
  ALARM_DATA <= "1111";
  SHOW_A <= '0';
  wait for PERIOD;

  SHOW_A <= '1';
  wait for PERIOD;

  SHOW_A <= '0';
  for I in 0 to 10 loop
    wait for PERIOD;
    TIME_DATA <= std_logic_vector(unsigned(TIME_DATA)+ 1);
  end loop;

  wait; -- suspend process
end process SIMPLE_STIMULUS;

```

175

© Cadence Design Systems, Inc. All rights reserved.

### Simple stimulus

- Send data to design
- No interaction
- Single stimulus processes
  - "In-line" stimulus
  - Simple loops for incrementing stimulus
- As in initial lab exercises



A simple example of stimulus generation is shown on the slide; there is any other interaction. Initially, we drive TIME\_DATA with “0000,” ALARM\_DATA with “1111,” and SHOW\_A with 0; we wait for a PERIOD, which is ten nanoseconds in this case. We then assign SHOW\_A with 0, and we drive incrementing stimulus on TIME\_DATA using for loop every 10 nanoseconds. We execute this code in our initial lab exercises.

## Self-Checking with Assertions

```

signal TIME_DATA, ALARM_DATA,
DISPLAY : std_logic_vector(3 downto 0);
signal SHOW_A : std_logic;
...

SIMPLE_STIMULUS: process
begin
  TIME_DATA <= "0000";
  ALARM_DATA <= "1111";
  SHOW_A <= '0';
  wait for 10 ns;

  assert DISPLAY /= TIME_DATA
    -- message if condition false
    report "incorrect DISPLAY"
    severity ERROR;
...

```

Runtime checks, report an error if the condition fails

Severity levels:

- NOTE
- WARNING
- ERROR (default)
- FAILURE (usually causes the simulation to halt)

Report message can be string type

- Can be built using the concatenation operator



Assertions ignored  
by synthesis tools

Synthesis



No "," after assert  
or report

Beware

176 © Cadence Design Systems, Inc. All rights reserved.



The other function of testbench is verifying the correctness of the design. We can do this either by writing checkers to compare actual values with expected values or by writing assertions. Assertions allow us to add self-checking to our code. It is a compelling way of continuously monitoring the behavior of a system and reporting back to the designer when things go wrong.

They report messages to the simulator interface when the assert condition is FALSE. They are runtime checks.

Useful to feedback information such as:

- Error conditions.
- Test progress reports.
- Warnings associated with the functional operation, e.g., overflow.

The assertion statement can be concurrent or sequential.

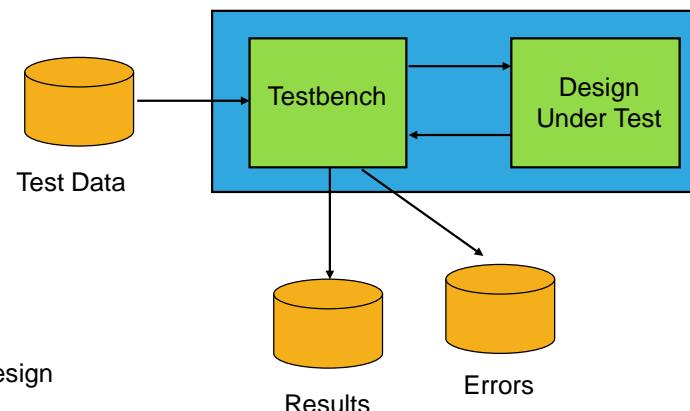
- For the concurrent version, the sensitivity list of the equivalent process is given by signals in the condition.

Severity levels like note, warning, error, and failure can be used to grade messages. The highest severity level, which is the failure, usually forces the simulation to halt, although most simulators have an option to control this. The default severity level is ERROR, which throws an error when there is a failure but doesn't stop the simulator.

The reported message can be any string type – concatenation can be used to build up a meaningful error message.

In the code on the slide, apart from driving stimulus, we continuously monitor if DISPLAY equals TIME\_DATA; if equal, it does nothing; if unequal, it reports an error but doesn't halt the simulation process severity level is ERROR. Assertions are ignored by the synthesis tool. Note there is no comma after assert or report.

## Complex Testbench



Sophisticated testbench

- Models, the environment around design
- Talks to design, for example, bus cycles, handshaking
- Generate results and check response
- Can read data from and write results to external files
- Visualization of data as in later lab exercises



In a sophisticated testbench, you may want to model the environment your design has to interact with. For example, you might want to model a microcontroller's bus cycles that interact with your design, or you may want to evolve your testbench towards a model of your system or PCB. You may want to generate results for automatic comparison with the results of subsequent simulation runs. You may need to check the correctness of the design using assertions.

You might want to read the contents of a file containing, for example, some image data. Perhaps the file has been created by another program or tool, such as a system modeling tool. We will be looking at how to do this in the advanced modules of this VHDL course.

There are many reasons you might want to generate an output file.

You might want a record of any errors or warnings. However, most simulators give you another way of doing this by allowing you to keep a copy of the messages that appear in the simulator transcript window.

You might want to generate a visualization of your output data as we do during the lab exercises.

## Register Transfer Level (RTL) Code

Synthesizer requires code to be written at RTL  
defines an implementation for synthesis

Constructs, style, and design restricted by  
synthesis tool support

- Simple data structures and types

Only two kinds of process

- Combinational or registered

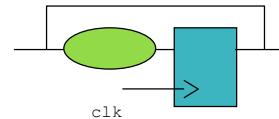
Some design features may need to be instantiated,  
for example, memory

Generally required to be synchronous

- Trigger activities on a clock edge
- Timing is clock cycle accurate

```
process (CLK)
begin
  ...
  ...
end process;
```

Clocked Process



```
process (all)
begin
  ...
  ...
end process;
```

Combinational Process



Processes must conform to specific  
templates to be synthesizable!

178 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Synthesis tools require synthesis code to be written at RTL. It defines the implementation details for synthesis. Synthesis tools do not support all language constructs or data types and place restrictions on the style and design of synthesis code. In particular, all processes must be defined as either inferring combinational or registered logic, and there are strict coding style rules to be followed for each type. In an RTL model, only two types of processes are used: combinational and clocked processes.

We have already seen that all signals read in the process must be in the sensitivity list for combinational processes.

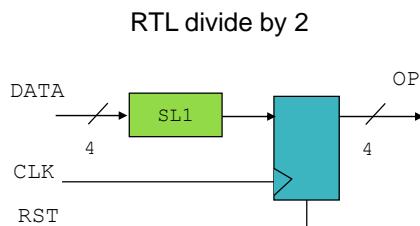
The other type of process is a clocked process. Normally synthesis tools only give you a limited number of ways to write a clocked process. However, there are a very large number of ways to express the same functionality for simulation purposes.

Some design blocks cannot be created by synthesis, for example, RAM, and must be explicitly declared as a component and instantiated as a separate hierarchy level.

Synthesis tools cannot effectively synthesize asynchronous logic because the synthesis methodology is, essentially, a synchronous methodology.

We have to worry about registers, the amount of logic between registers, clocking schemes, and resets.

## Registered Process



```

process (CLK, RST)
begin
    if RST = '1' then
        OP <= (others => '0');
    elsif rising_edge(CLK) then
        OP <= '0' and IP(3 downto 1);
    end if;
end process;
    
```

Registers are inferred on all signal assignments in a clocked process

Must consider resets when describing clocked logic

- Here, RST is asynchronous

Clock edge detected using `rising_edge` function call

- Declared in `std_logic_1164` package
- True when signal has changed from '0' to '1'
- Also `falling_edge` function



A synthesis tool will create a register for any signal which is the target of a signal assignment in a correctly constructed clocked/registered process.

This clocked process is only sensitive to the clock signal CLK and the asynchronous reset signal RST.

In the body of a clocked process is a single `if` statement, although this may contain nested if and case statements as required. The asynchronous reset is checked first. While this is a '1', the register remains in the reset condition. If the reset condition is false, then the simulator checks the state of the clock signal. The condition here is "true" on the rising edge of the clock, so it is only after the rising edge is detected that the assignment of IP to OP is executed. The `rising_edge` function call, defined in the `std_logic_1164` package. It returns true when the clock changes from 0 to 1. Similarly, there is a `falling_edge` function that returns true on detecting a negative edge.

Some synthesis tools do support other coding styles for the clocked process. However, if you stick to the one shown here, you won't go wrong, and you will be writing code that is most portable between synthesis tools.

## Registered Process: Counter

```

architecture RTL of COUNTER is
  signal COUNT : unsigned(3 downto 0);
begin

  process (CLK, RST)
  begin
    if RST = '1' then
      COUNT <= (others => '0');
    elsif rising_edge(CLK) then
      if (COUNT >= 9) then
        COUNT <= (others => '0');
      else
        COUNT <= COUNT + 1;
      end if;
    end if;
  end process;

  Q <= std_logic_vector(COUNT);

end RTL;

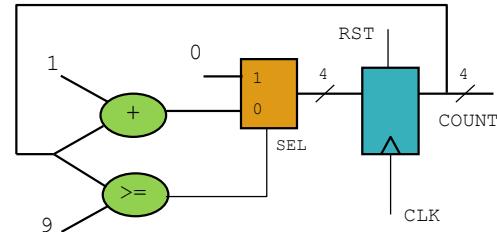
```

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity COUNTER is
port (CLK, RST : in std_logic;
      Q : out std_logic_vector(3 downto 0));
end COUNTER;

```



180 © Cadence Design Systems, Inc. All rights reserved.



A clocked process does not only contain registers. We can include code for any registered logic in a clock process, for example, counter, shift-register, state machine, pipelined arithmetic, etc. The logic code is placed after the edge expression of the clocked process, taking care to add a reset value for the required process output signals. As for clocked processes, we typically use templates to create registered logic. For example, a simple counter in VHDL will usually have the form shown above if the count has reached the maximum value, clear count; otherwise, increment the count by 1.

Similarly, a decrementing counter will have a similar form if the count has reached the minimum value; the count is set to maximum; otherwise, decrement the count by 1.

Previous to VHDL2006, we could not read the output port  $Q$  inside the architecture. Therefore, we use an intermediate signal, COUNT, to calculate the counter value. COUNT is assigned to  $Q$  in a separate concurrent statement. We declare COUNT as a type unsigned to make the comparison and addition operators easier to use, and we convert COUNT from unsigned to std\_logic\_vector in the concurrent assignment to  $Q$ .

VHDL2006 removes the restriction on reading output ports.

To the right, we have the hardware inferred by this code. We see that output equals zero if reset equals zero, irrespective of what values other signals take. The reset is asynchronous in this example, as it is a part of the sensitivity list; output can change when the reset changes and need not wait for the clock edge. If reset equals zero, then the din of the flip-flop is driven by MUX with SEL as the select line. When the count equals 9, the output of the comparator goes high, making SEL high; value 0 on pin 1 of MUX is passed to the din of D-flip-flop; otherwise, when the count is less than nine, the output of the comparator is zero. Hence, the value on pin 0, which is nothing but the increment of the previous count, is passed onto the din of the flip-flop incrementing its output by one. This repeats until the count reaches value nine, and then it rolls over to zero, and the process repeats.

## Structural (Gate-Level) Code

Instantiates and connects components

Component instantiations and signal connections only

- Typically, `std_logic` and `std_logic_vector` types only

Output from synthesis

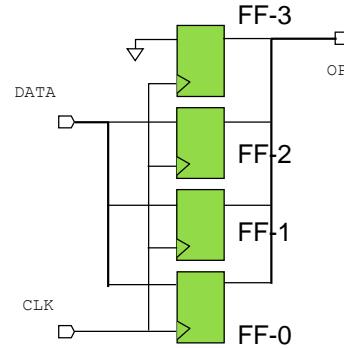
- Netlist of cells/gates from a specific vendor's technology

Also used for top-level hierarchical design

- Integration of functional blocks

Gate-level divided by 2

```
opreg0 : FD1 port map (DATA(1), CLK, OP(0));
opreg1 : FD1 port map (DATA(2), CLK, OP(1));
opreg2 : FD1 port map (DATA(3), CLK, OP(2));
opreg3 : FD1 port map (LOGIC0, CLK, OP(3));
```



181 © Cadence Design Systems, Inc. All rights reserved.



Structural/gate-level VHDL typically contains only component instantiations and signals to connect the component ports. Only very simple types are used for the signals, typically `std_logic` or `std_logic_vector`. In this respect, a structural VHDL netlist can be defined as a text version of a schematic.

Structural code is output by synthesis tools. In this case, the component instantiations will only be basic cells or gates from the vendor technology.

However, a top-level design may also use structural code. In this case, the component instantiations will be the high-level functional blocks of the design.

The example shows how a divide by two logic can be implemented by using four instantiations of D-flip-flop and connecting the ports appropriately.

Here, we see that MSB is assigned to 0, and other bits are shifted to the right by 1-bit, i.e., `DATA(0)` is dropped, flip-flop 0 is input with `DATA(1)`, flip-flop 1 is input with `DATA(2)`, flip-flop 2 is input with `DATA(3)` and flip-flop three which is the MSB is tied to 0. As we see, dividing by two is obtained by performing a logical right shift by one and appending 0 to the MSB.

## VHDL Coding Styles: Summary Quiz

1. What is behavioral modeling used for?
2. Why is the entity of a testbench typically empty?
3. What are the two types of processes used in RTL code?
4. How do we infer registers in RTL code?

182 © Cadence Design Systems, Inc. All rights reserved.



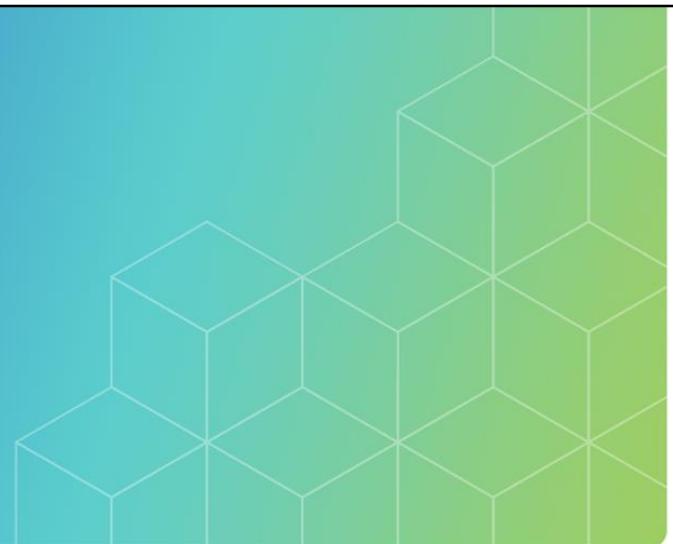
Let's take a short quiz on this module. What is behavioral modeling used for? Why is the entity of a testbench typically empty? What are the two types of processes used in RTL code? How do we infer registers in RTL code?

## VHDL Coding Styles: Summary Quiz (Solutions)

1. What is behavioral modeling used for?
  - Testbenches, simulation models, and occasionally system-level modeling.
2. Why is the entity of a testbench typically empty?
  - It is the top level of the design; it should be completely self-contained and, therefore, will not have any port declarations.
3. What are the two types of processes used in RTL code?
  - Combinational and registered.
4. How do we infer registers in RTL code?
  - Every signal assignment in a correctly constructed registered process will infer registers.



Here is the solution. Behavioral modeling is used for testbenches, simulation models, and occasionally system-level modeling. The testbench entity is typically empty as it is at the top level of the design; it should be completely self-contained and will not have any port declarations. The two types of processes used in RTL code are combinational and registered. Every signal assignment in a correctly constructed registered process will infer registers.



## The Synthesis Process

**Module** **12**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this section, you will look at how synthesis tools work and what they are good at and not so good at doing.

## Module Objectives

In this module, you

- Describe the synthesis process
- Outline the working of a synthesis tool
- Define synthesis-based methodology
- List out the strengths and weaknesses of the synthesis
- State programmable logic device synthesis issues and compare with cell-based ASIC synthesis
- Define language subsets



In this module, you describe the synthesis process, outline how a synthesis tool works, discuss synthesis-based methodology, list synthesis strengths and weaknesses, state programmable logic device synthesis issues and compare with cell-based ASIC synthesis, and define language subsets.

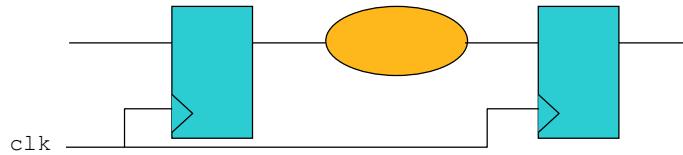
## What Does Synthesis Do?

Infers structure of cells from RTL code

Builds optimized combinational logic

The obvious questions are:

- How much of a design can be synthesized?
- How well does the tool build it?
- Can it be trusted?



186 © Cadence Design Systems, Inc. All rights reserved.



Synthesis automatically transforms your design, expressed in RTL VHDL, into an implementation consisting of a structure of cells. It builds optimized combinational logic.

The obvious questions are:

- How much of your design can be built?
- How well does the tool make it?
- Can it be trusted?

## Synthesis Flow

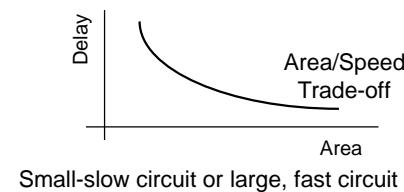
Cells and gates from the technology library were used to build the netlist

VHDL code is the most important input

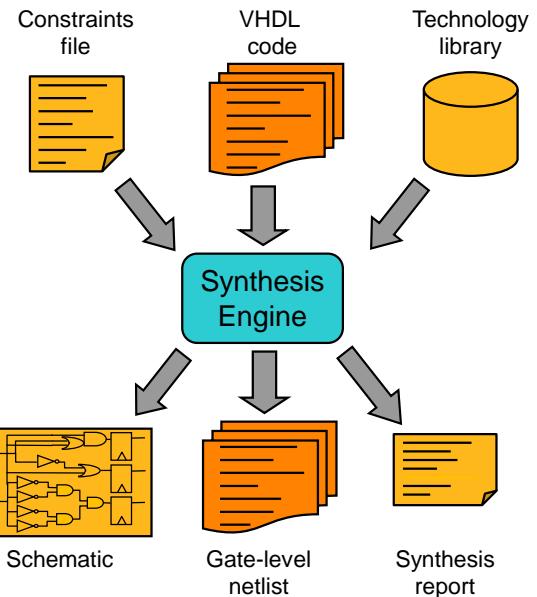
- Quality of results depends mainly on code

Constraints file sets synthesis goals

- Clock speed, area, timing
- Reports success
- Technology library



187 © Cadence Design Systems, Inc. All rights reserved.



A synthesis tool can read your RTL VHDL code and convert it into a gate-level netlist, using gates and cells from a specified target technology library. VHDL code forms the most important input to the synthesizer. The quality of the results mainly depends on the code.

The synthesis tool will try to meet your design constraints for the area and performance of the netlist and will produce reports to tell you how successful it has been.

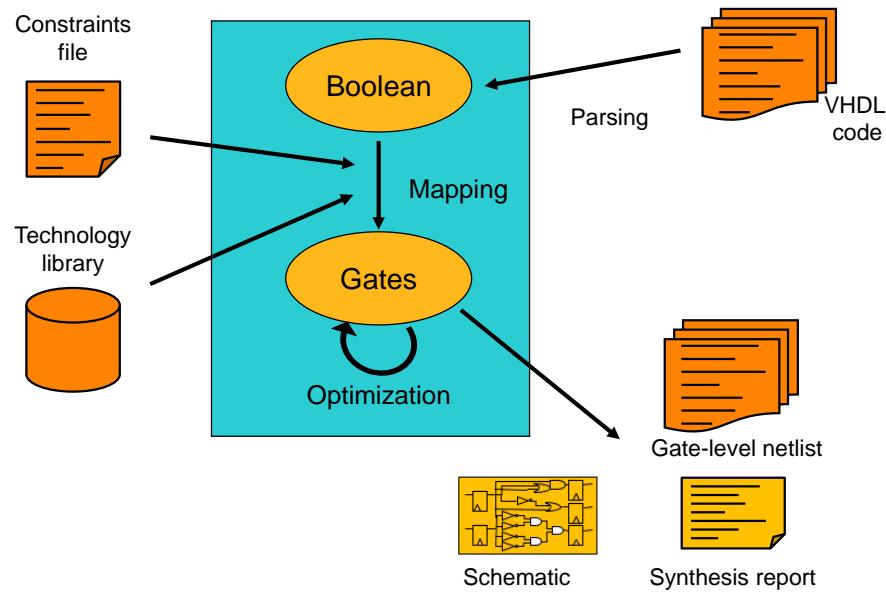
Design constraints may be as simple as a required clock speed specification or include more complex constraints, covering input and output timing requirements, area goals, environmental conditions, etc.

### The Area/Speed Curve

The Area/Speed Curve Envelope is the best that the synthesis engine can accomplish, given the circuit architecture and the area and speed constraints. For example, synthesis can produce a small slow circuit or a large fast circuit, or something in between. The synthesis constraints located in the constraints file drive the synthesis engine to produce the type of circuit required.

The technology library contains details of the area, timing, and characteristics of cells in the target technology library. Sophisticated synthesis tools may also optimize a design for power, requiring a technology library with this information.

## How Synthesis Works



188 © Cadence Design Systems, Inc. All rights reserved.



The synthesis tool first translates VHDL code constructs into an internal boolean representation. The synthesis tool can optimize these equations using boolean algebra and manipulations.

The optimized boolean equations are then mapped to the individual cells and gates from the target silicon technology to produce an initial gate-level netlist.

The synthesis tool can then optimize the gate-level netlist by replacing combinations of simple gates with more complex cells from the technology library if they are available. The synthesis tool also checks if your speed and area design constraints are met. If constraints are not met, the tool may go back to the boolean representation to try different boolean optimizations and alternative cell mapping/optimization.

The synthesis tool will also report on resources used, worst-case timing, synthesis settings, and any warnings issued during the synthesis process. If the constraints cannot be met, the synthesis tool will also report where and how constraints have been violated.

When the optimization is complete, a gate-level description can be written for gate-level simulation and layout.

This is a rather simplified view of how a synthesis tool works but does explain the basic concepts.

## Coding Styles Affect Results

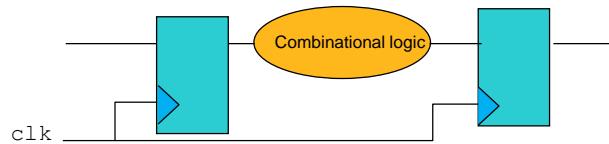
Number of registers

- Combinational logic optimized only
- Not number or placement of registers
- Check code to avoid unintentional registers

Amount of combinational logic between registers

- Pipelining and logic timing must be considered when code written

Be careful about structuring and partitioning your design



189 © Cadence Design Systems, Inc. All rights reserved.



Remember, a synthesis tool does not optimize the registers or latches in your design. You need to be careful how you structure your registered processes only to infer the registers you need. Check synthesis reports carefully to make sure!

It would help if you also were careful about how much combinational logic is placed between adjacent register banks. If you place a 32-bit multiplier between adjacent register banks with a 10 ns clock period, don't be surprised if the synthesis tool struggles to produce a result.

Generally, it would help if you were careful how you partition and structure your design into combinational and registered logic.

## Translation Can Be Literal!

Synthesis tools generate logic directly from the structure of code

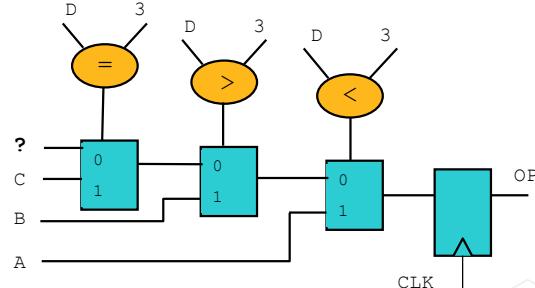
- Here = operator is redundant but *may* still appear in the synthesized netlist

```
process (CLK, RST)
begin
  if RST = '1' then
    OP <= (others => '0');
  elsif rising_edge(CLK) then
    if D < 3 then
      OP <= A;
    elsif D > 3 then
      OP <= B;
    elsif D = 3 then
      OP <= C;
    end if;
  end if;
end process;
```



What would be  
the input at "?"

Question



190 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

Synthesis tools cannot interpret or analyze your code to understand your requirements. Translation of your code to boolean logic is a very literal process. Although synthesis optimization may remove redundant logic from your design, it is better if the designer removes the redundant logic before synthesis.

In this example, the input labeled "?" should be connected to something, but what?

If all the conditional checks in the process are false, then the output OP is not changed. Therefore input ? must be connected to the output OP, and there is a feedback path around the register from the output OP to the input ?.

So even if the path from ? to OP can never be valid; the synthesis tool *may* still make this connection.

## Synthesis Limitations

ASIC design clock tree balancing

- Usually require detailed, accurate net delay information

Complex clocking schemes

- Synthesis tools prefer simple, single synchronous designs

Memory, IO pads, technology-specific cells

- You will probably need to instantiate these by hand

Specific macro-cells

Always design as well as you can

- Although it can analyze hundreds of implementations in the time taken for a designer to analyze one



Synthesis tools are targeted at optimizing combinational logic—large parts of your design will not be combinational logic.

ASIC clock trees are a global, chip-wide problem. To optimize clock skew and propagation delay, accurate net length and delay information is required. This is not available until the layout or place and route. Therefore, using specialist tools, clock tree synthesis is usually a post-logic synthesis operation.

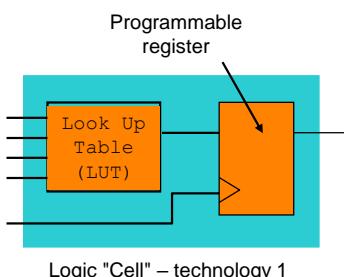
Synthesis tools do not cope well with complex clocking schemes. Typically, each block to be synthesized will be allowed to have one clock, although two-phase clocks or clocking on both edges of the clock is often supported.

You would not want to synthesize large memory blocks since a synthesis tool will build it out of flip-flops. You will need to instantiate a technology-specific RAM from your silicon vendor directly.

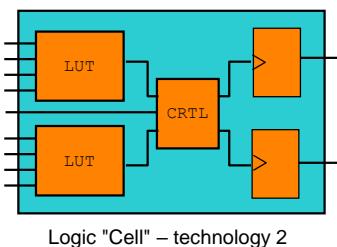
Since synthesis tools are not very good at selecting large cells or macros from a technology library, you may have to instantiate them by hand.

Lastly, synthesis tools are not guaranteed to produce the smallest possible result. They do not always do as good a job as you could, if only you had the time! If you are having problems with a critical path, you could always look at it on a schematic – you may be able to spot an optimization the tool has missed.

## Programmable Logic Device Specific Issues



- Different architectures for different technologies
- Fixed architecture within a specific technology
- Architecture-specific “tricks” for best utilization/speed
- Technology specific/generic code trade-offs
- How your synthesis tool handles your technology



192 © Cadence Design Systems, Inc. All rights reserved.



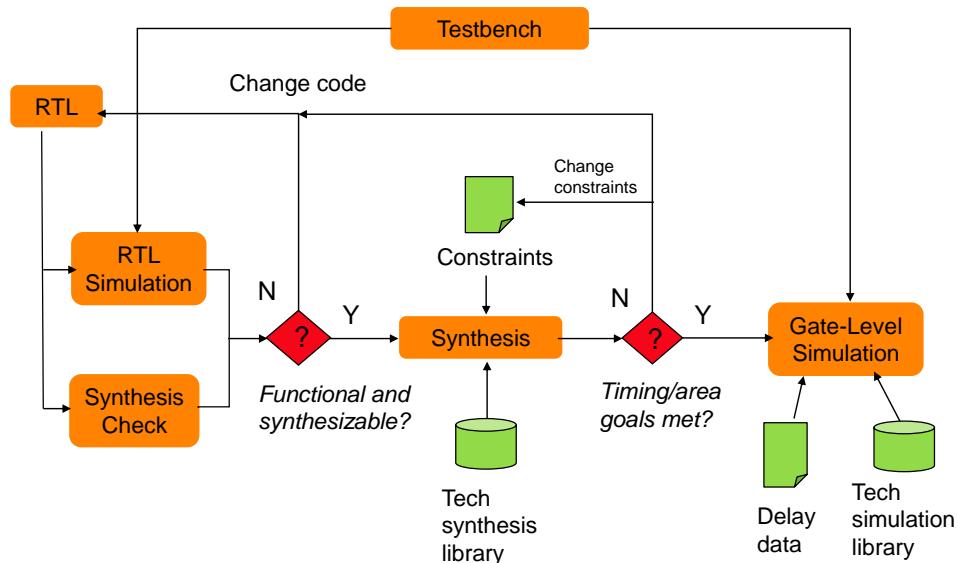
Many of the comments so far have assumed synthesis to an ASIC. With an FPGA, there are some different problems. ASIC synthesis tools all work in the same way, using the same optimization algorithms, and the target technology libraries all have similar cells. With FPGAs, there are different technologies, EEPROM, SRAM, and anti-fuse, and each vendor has different building blocks, which tend to be much larger than the gate-level primitives used in an ASIC.

For a particular FPGA, the architecture is fixed, limiting the usefulness of static timing analysis since routing delays can vary widely – with an ASIC, they are much more predictable. To get a good "fit", the FPGA designer traditionally employs architecture-specific tricks that are difficult to implement in a synthesis tool.

VHDL code may need to use technology and architecture-specific techniques and tricks. These may be expressed with user-defined attributes, comments, or direct instantiation of specific cells. Such methods restrict your code's technology independence and portability but allow the most efficient implementation with a specific technology.

A key issue is how well your synthesis tool handles your chosen technology. To achieve a good result, it will need architecture-specific algorithms.

## Typical Synthesis Methodology



193 © Cadence Design Systems, Inc. All rights reserved.



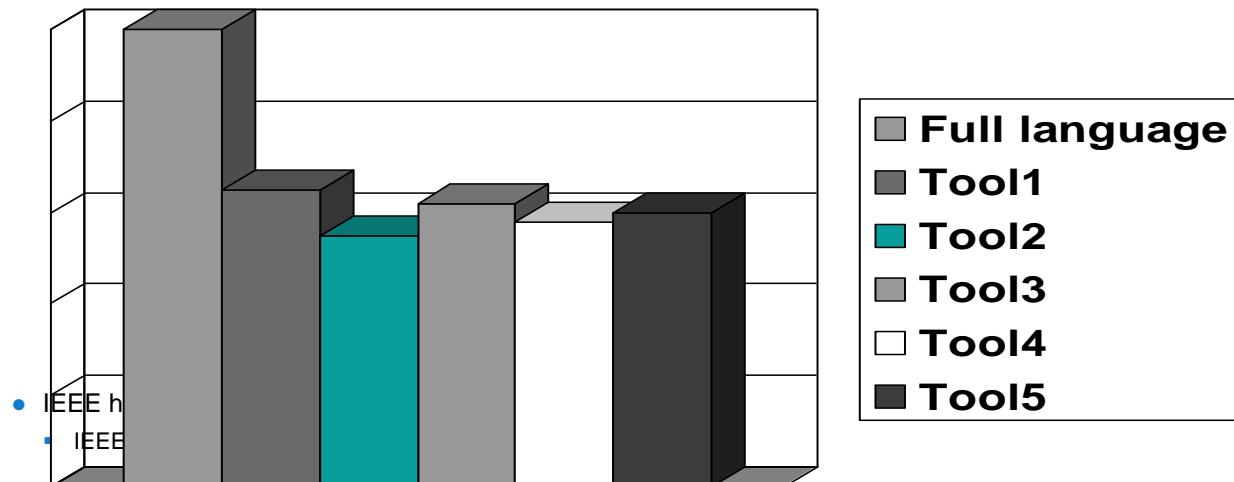
With all synthesis tools, you will be following a very similar methodology to the one shown here. First, select a design block. The size of the block is often a trade-off between how much of your design you want to synthesize at once and the ability of your computer resources to produce a result in a reasonable time! Set up some timing constraints, e.g., clock period, and synthesize.

After synthesis, use the synthesis tool's built-in static timing analyzer to check for timing constraint violations. If satisfied, you would carry out a gate-level simulation to check that you still have the same functionality you started with.

If the results from synthesis are not good (e.g., critical timing violations), there are several options. Check your constraints are accurate and realistic; change some of the synthesis tool switches which affect how the optimization algorithms work. Finally, you could change the VHDL; the easiest thing to do here is to alter the hierarchy by partially flattening it – this again gives the tool more degrees of freedom to meet your constraints. Finally, if nothing else works, you may have to bite the bullet and change the architecture of your design.

## Language Support

### Synthesis Subsets



194 © Cadence Design Systems, Inc. All rights reserved.



The full VHDL language is not synthesizable. As we have seen, we must adopt a specific coding style and use a subset of the language constructs when we write synthesizable RTL code. However, each synthesis tool may support a slightly different coding style and a different subset of language constructs. To address this issue, the IEEE has approved a standard synthesizable subset of VHDL (IEEE1076.6), which defines a minimum coding style and set of constructs that should be supported by every synthesis tool. The RTL code in this course conforms to this standard. You may find that a particular synthesis tool supports additional constructs or styles, but your code may not work with other tools if you use those constructs or styles. If you follow the guidelines, this will give you maximum code portability.

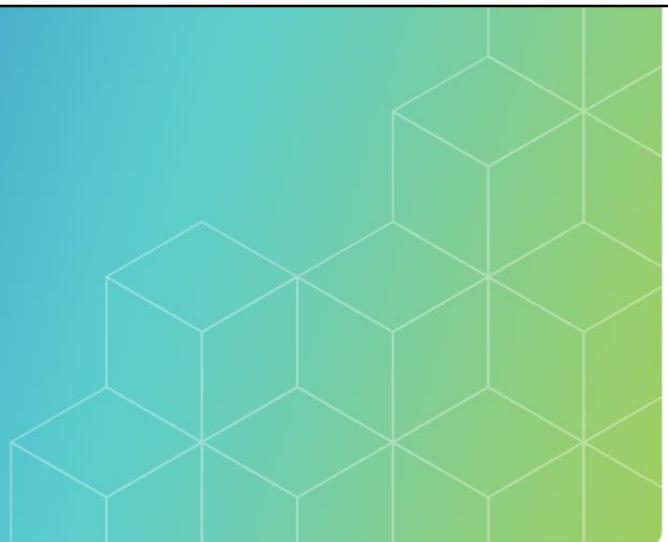
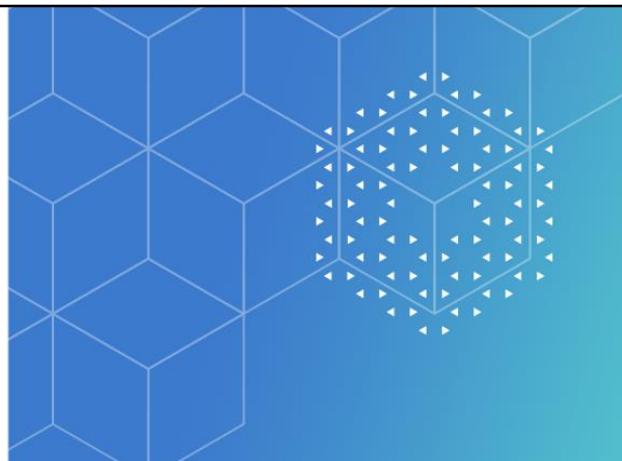
## The Synthesis Process: Summary

- Coding style affects results
- Some design features cannot be synthesized
- Programmable logic design uses architecture-specific “tricks” for best utilization/speed
- Synthesis tools support subsets of language constructs and style
- Always check your synthesis reports

195 © Cadence Design Systems, Inc. All rights reserved.



Let us summarize this module. Coding style affects results. Some design features cannot be synthesized. The programmable logic design uses architecture-specific “tricks” for best utilization/speed. Synthesis tools support subsets of language constructs and style. Always check your synthesis reports.



## Definition of RTL Code

**Module** **13**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we deal with the definition of RTL code, various RTL coding styles, and synthesis rules.

## Module Objectives

In this module, you

- Define RTL and list a few portability issues
- State various RTL coding styles
- Define ways to prevent unintentional latches
- Describe
  - Combinational process
  - Clocked process

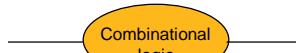


In this module, you will define what RTL is and list a few portability issues, state various RTL coding styles, define ways to prevent unintentional latches, and describe the combinational process and clocked process.

## RTL Style (Review)

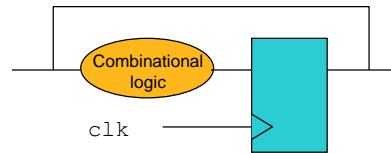
### Combinational Process

```
COMB: process (all)
begin
  ...
  ...
  ...
end process COMB;
```



### Clocked Process

```
REGR: process (CLK)
begin
  ...
  ...
end process REGR;
```



Processes must conform to specific templates to be synthesizable!

Synthesis

198 © Cadence Design Systems, Inc. All rights reserved.



The definition of RTL style is really very simple. You are required to write your code in either:

- A combinational process that defines purely combinational logic.
- A clocked process that defines combinational logic with registers on the output.

By partitioning your code between these two types of processes, you control the architecture of the synthesized design.

Note that processes must conform to specific templates to be synthesized.

## Complete Sensitivity Lists (Review)

- For combinational logic, the sensitivity list must contain all signals read in the process
- Known as a *Complete Sensitivity List*
- Doesn't contain signals which are assigned in the process unless they are also read

```
MUX: process (A, B, SEL)
begin
    if SEL = '1' then
        OP <= A;
    else
        OP <= B;
    end if;
end process MUX;
```



What would the behavior be if `SEL` was missing from the sensitivity list?

Question

199 © Cadence Design Systems, Inc. All rights reserved.



The language allows any signals to be placed on the sensitivity list.

A complete sensitivity list is where all of the signals read in the process are included in the sensitivity list.

Real combinational hardware is sensitive to all of its inputs, so the first rule for a combinational process is that it should have a complete sensitivity list.

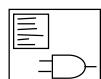
- Note that this does not include the signals assigned within the process unless they are also read.

All synthesis tools require a complete sensitivity list to synthesize combinational logic.

In the given code, on the slide, we see that signals A, B, and SEL are read by the process MUX. Hence, they are placed on the sensitivity list of the process.

We will see the behavior if any signal, say SEL, goes missing in the sensitivity list on the next slide.

## Incomplete Sensitivity List (A Reminder!)

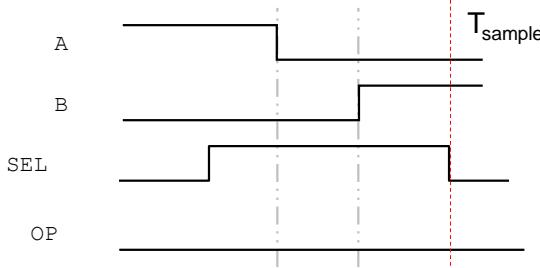


For combinational logic,  
sensitivity list must contain  
all signals read in the process

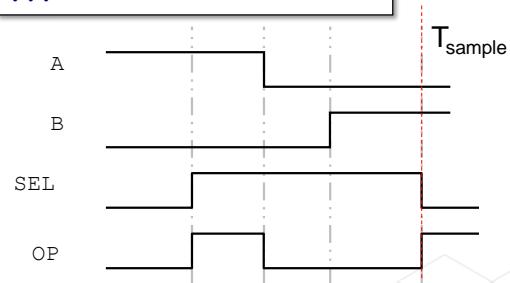
Synthesis

```
MUX: process (A, B, SEL)
begin
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process MUX;
```

MUX: process (A, B)  
...



MUX: process (A, B, SEL)  
...



200 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

Examine the effects of an incomplete list, consider process MUX:

If SEL is missing, the assignment to OP is only made on an event (change of value) on A or B. Changes in the value of SEL do not trigger the process and do not immediately update the output.

Synthesis tools should require a complete sensitivity list to synthesize combinational logic.

Incomplete sensitivity lists can be difficult to debug in simulation: outputs can lag the expected values in some circumstances. In this slide, we see that, in the waveform on the left, OP is evaluated only when A or B changes, changes on SEL don't trigger the process. Whereas in the waveform on the right, which includes SEL in the process sensitivity list, the OP value is evaluated whenever A, B, or SEL changes, as indicated by the vertical lines.

At the time  $T_{sample}$ , the SEL value changes from 1 to 0. This should trigger the process, and OP should be assigned with B, which is one at this instant. This happens in the waveform shown on the right as SEL is contained in the sensitivity list, whereas in the waveform on the right, at  $T_{sample}$   $OP = 0$ , as the process is not triggered due to the incomplete sensitivity list, the value is not updated with B.

## process (all)

- VHDL2008 update
- Used in combinational processes
- Implied process sensitivity list with all inputs
- Safe to use even with procedures
- No performance implications
  - Negligible compiler time to determine the sensitivity list
- Avoids any mistakes of missing signals, renamed or added signals

```
MUX: process (all)
  if SEL = '1' then
    OP <= A;
  else
    OP <= B;
  end if;
end process MUX;
-- compiler expands this to
-- process(A, B, SEL)
-- as A, B, SEL are inputs
```



VHDL2008 introduces `process (all)` for combinational processes. All is equivalent to a sensitivity list with all inputs to the process. It can be used even when the process contains procedures that work on signals. Performance-wise, there is not much change as the compiler takes no extra time to figure out the implied sensitivity list. Process all gives the advantage of avoiding the mistakes that may occur due to accidental missing of any signal listing, renaming, or any addition of a new signal.

In the code on the slide, `process (all)` implies `process (A, B, SEL)` as A, B and SEL are the inputs read by the process MUX.

## Incomplete Assignments

- OP retains its previous value, if CTRL = '0'
- Synthesis tool infers transparent latch
- Can be avoided by ensuring complete assignment to every process output

```

library IEEE;
use IEEE.std_logic_1164.all;

entity INCOMP_IF is
  port (CTRL, A : in std_logic;
        OP       : out std_logic);
end INCOMP_IF;

architecture RTL of INCOMP_IF is
begin

  INCOMP : process (all)
  begin
    if (CTRL = '1') then
      OP <= A;
    end if;
  end process INCOMP;

end RTL;

```



The incomplete assignment is a condition where there is an execution path through a combinational process where an output signal from that process is not updated. In this example, if CTRL equals '1', OP is assigned to A otherwise, and no operation is performed, i.e., the signal OP must retain its previous value.

A synthesis tool will use a transparent latch to implement this in hardware.

This can be avoided by ensuring complete assignment to every process output.

## Preventing Latches

Two ways to prevent latches:

- Use default assignment
- Add else clause

```
architecture RTL of COMPLETE1 is
begin
  FULLIF: process (all)
  begin
    if (CTRL = '1') then
      OP <= A;
    else
      OP <= '0';
    end if;
  end process FULLIF;

end RTL;
```

```
architecture RTL of COMPLETE2 is
begin
  DEF: process (all)
  begin
    OP <= '0';
    if (CTRL = '1') then
      OP <= A;
    end if;
  end process DEF;
end RTL;
```



Which is the best approach for a process with complex, nested if statements?

Question

203 © Cadence Design Systems, Inc. All rights reserved.



There are two ways to prevent a synthesis tool from creating transparent latches:

1. Use an else clause for the if statement.
2. Initialize OP at the top of the process with a default statement.

Which is the best technique in a real design?

With a complex design, it can be difficult to ensure that you have covered all of the branches of complex if and case statements with when others or unconditional else clauses. There initializing the outputs to default values at the top of the process can make our code easier to construct. Making a default assignment to the most common value could simplify our code by only considering exceptions to the common value.

Hence, we have a fundamental rule for the synthesis of combinational logic:

- To prevent inferred transparent latches from a combinational process, you must give default assignments to all signals assigned within the process at the start.

## Rules for Synthesis of Combinational Logic

- Complete sensitivity list
  - Use (all) sensitivity list to ensure completeness
- Default assignments to prevent latches
- Avoid feedback loops



In summary, the rules for the synthesis of combinational logic are:

- Complete sensitivity list.
  - Use (all) sensitivity list to ensure completeness.
- Default assignments to prevent latches.
- Avoid feedback loops (remember?).

## Describing a Rising Clock for Synthesis

Synthesis tools look for code templates for a clocked process

- 'event' makes clock detection edge-sensitive

If CLK is of std\_logic type, then change from any value to logic '1' will trigger the process

- For example, change from 'U' to '1' triggers the process

```
FF1: process
begin
  wait until CLK'event and CLK='1';
  Q <= D;
end process FF1;
```



```
FF2: process
begin
  wait until CLK = '1';
  Q <= D;
end process FF2;
```



```
FF3: process (CLK)
begin
  if (CLK'event and CLK='1') then
    Q <= D;
  end if;
end process FF3;
```



```
FF4: process (CLK)
begin
  if (CLK = '1') then
    Q <= D;
  end if;
end process FF4;
```



205 © Cadence Design Systems, Inc. All rights reserved.



A synthesis tool does not interpret your RTL code and try to understand what you want. A synthesis tool looks for specific code constructs or templates and infers registered logic from these. So, although there are many ways in VHDL to describe a clock edge, only some of these ways are supported by the majority of synthesis tools.

Using those marked with an "X" is not recommended, as some synthesis tools do not recognize these as clocked processes.

For portability, use those marked with a tick (check).

Note: If CLK is a signal of type std\_logic, these processes will trigger when CLK changes from any value to '1', e.g., a change from 'U' to '1' will execute the process.

## Rising Edge with a Function Call

```
library IEEE;
use IEEE.std_logic_1164.all;
...
process
begin
  wait until rising_edge(CLK);

  Q <= D;
end process;
```

- In std\_logic\_1164 package
- Also falling\_edge(CLK)



Use functions for  
clocked processes

Tip

```
-- IEEE.std_logic_1164
function rising_edge (signal CLK : std_ulogic) return boolean is
begin

  if (CLK'event and CLK='1' and CLK'last_value='0') then

    return true;
  else
    return false;
  end if;
end rising_edge ;
```

206 © Cadence Design Systems, Inc. All rights reserved.



The std\_logic\_1164 package (which contains the definitions for std\_logic) also contains functions that describe rising and falling edges.

The function detects clean rising or falling edges by examining the previous value of the clock signal. The function uses the following attributes:

- 'event to make the condition edge sensitive.
- 'last\_value to check the value of the clock signal before the current change.

The function rising\_edge has one input of type std\_ulogic, CLK in this case, and returns a boolean value.

So, for rising edge detection, the function will check that the clock has changed value ('event), i.e., the current value is '1' and the previous value was '0', as shown in the code. If it detects a rising\_edge, it will return true otherwise false. On true, the line wait until rising\_edge(CLK) goes through, and Q is assigned to D.

Therefore, the rising\_edge and falling\_edge functions are actually better than the options mentioned on the previous slide, as they only trigger on clean rising or falling edges. Although if you have 'x' or 'z' values on your clock signals, you have bigger problems than clean edge detection.

## Register Inference in Synthesis

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity COUNTER is
port (CLK : in std_logic;
      Q : out std_logic_vector(3 downto 0));
end COUNTER;

architecture RTL of COUNTER is
signal COUNT : unsigned(3 downto 0);
begin

REGR: process (CLK)
begin
  if rising_edge(CLK) then
    if (COUNT >= 9) then
      COUNT <= (others => '0');
    else
      COUNT <= COUNT + 1;-- + used in unsigned context
    end if;
  end if;
end process REGR;

Q <= std_logic_vector(COUNT);

end RTL;

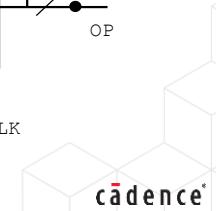
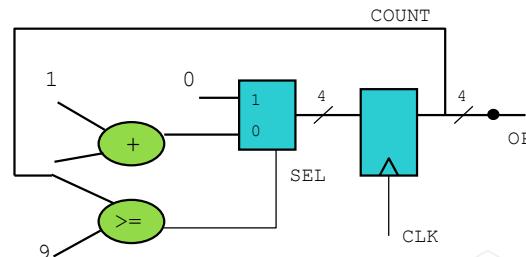
```

- Clocked processes triggered on the clock signal
- Registers are inferred on all signal assignments in clocked processes



What is the problem  
with this counter description?

Question



207 © Cadence Design Systems, Inc. All rights reserved.

This clocked process is sensitive to the clock signal only.

The basic rule about using register inference in a real design is that every signal assigned within a process with a clock edge detection scheme has a register placed on it in the synthesized hardware. In this example, assignment to COUNT generates registers.

This code is an up counter, which counts up to 9 by incrementing count by one on every rising edge of the clock, and rolls over to 0, which repeats. The assignment to COUNT only occurs in the code when there is a rising clock edge; this is why the code is equivalent to the hardware shown. The count is declared to be of the unsigned type; the result is also unsigned to use the arithmetic operator in an unsigned context. Now, we need to type cast to assign this result to Q, which is of std\_logic\_vector type.

The problem with this design is that it lacks a reset to initialize the count value. COUNT will initialize to UUUU, as it is of standard logic type and never increments.

## Resetting Clocked Processes

```
entity COUNTER is
  port (CLK, RESET : in std_logic;
        Q         : out std_logic_vector(3 downto 0);
end COUNTER;
```

```
SYNC_RST: process (CLK)
begin
  if rising_edge(CLK) then
    if RESET = '1' then
      COUNT <= (others => '0');
    elsif (COUNT >= 9) then
      COUNT <= (others => '0');
    else
      COUNT <= COUNT + 1;
    end if;
  end if;
end process SYNC_RST;
```

```
ASYNC_RST: process (CLK, RESET)
begin
  if RESET = '1' then
    COUNT <= (others => '0');
  elsif rising_edge(CLK) then
    if (COUNT >= 9) then
      COUNT <= (others => '0');
    else
      COUNT <= COUNT + 1;
    end if;
  end if;
end process ASYNC_RST;
```

Synchronous Reset	Asynchronous Reset
CLK takes priority over RESET	RESET takes priority over CLK
RESET condition checked <i>after</i> the clock edge	RESET condition checked <i>before</i> the clock edge
	RESET added to sensitivity list

208 © Cadence Design Systems, Inc. All rights reserved.



To infer reset registers, use an `if-elsif` statement to check for the reset condition. There are two types of reset, synchronous and asynchronous.

For synchronously reset registers:

- The test for the reset must come after the clock edge so that the reset is only checked on the clock edge and is implemented as a synchronous reset.

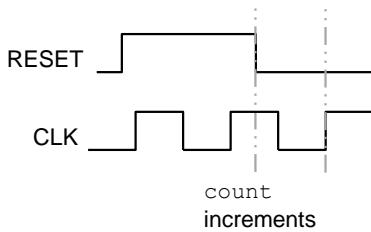
For asynchronously reset registers:

- The test for the reset must come before the clock edge so that the reset takes priority over the clock edge and is implemented as an asynchronous reset. The reset signal must be added to the sensitivity list of the process. Since the reset takes priority over the clock, we must trigger the process on a reset signal event to check the reset condition.

All signals assigned within the process will have registers placed on them in the synthesized hardware and should therefore be assigned values within the reset condition branch.

Note: Resets are level-sensitive; therefore, we check the reset value (`RST = '1'`), not whether the reset has a rising or falling edge.

## Why We Need 'event'



```
ASYNC_RST: process (CLK, RESET)
begin
    if RESET = '1' then
        COUNT <= (others => '0');
    elsif CLK = '1' then
        if (COUNT >= 9) then
            COUNT <= (others => '0');
        else
            COUNT <= COUNT + 1;
        end if;
    end if;
end process ASYNC_RST;
```



Without 'event, clock edge expression is level sensitive

This causes problems, e.g., asynchronously reset registers:

- Falling edge on RESET when CLK high triggers the process
  - Would increment COUNT

Use full clock edge expression in all clocked processes

`rising_edge(CLK)`

`CLK'event and CLK = '1'`



Synthesis usually requires  
full clock edge expression

Synthesis



Although the 'event attribute in the clock edge expression seems to be redundant, it is not redundant in all cases.

The 'event makes the clock expression edge-sensitive. Without the 'event, it is level-sensitive. In an asynchronously reset clocked process, the reset condition is level-sensitive; as long as the reset is high, the process is in reset mode, as the first if checks for reset condition. If the clock expression were also level-sensitive, then if the reset signal goes low when the clock is high, the process would be clocked! The falling edge on the reset signal would trigger the process, the reset condition would be false, but the level-sensitive clock condition would be true, and, in this case, the count signal would be incremented.

Since registers are clock edge-sensitive, we must use edge-sensitive clock expressions to infer registers – for all clocked processes, not just those which are asynchronously reset.

## Clocked Process Rules

```
process
begin
  wait until rising_edge(CLK);
  -- all synchronous actions
end process;
```

```
process(CLK, RST)
begin
  if (RST = '1') then
    -- asynchronous reset actions
  elsif rising_edge(CLK) then
    -- all synchronous actions
  end if;
end process;
```

```
process(CLK)
begin
  if rising_edge(CLK) then
    -- all synchronous actions
  end if;
end process;
```

wait form

- No sensitivity list
- Difficult to use for asynchronous resets

if form

- Sensitivity list only contains clock for synchronous reset
- Clock and reset for the asynchronous reset

All signals assigned to get a register

A rising edge can be described with a function call or event expression

Code must follow these templates



The clocked process can be implemented either using the wait or if form. Wait form has no sensitivity list and is difficult to use with asynchronous reset. The if form has the clock in the sensitivity list and the reset if there is an asynchronous reset. For synchronous, there is the only clock in the sensitivity list. All signals assigned within the process get a register assigned.

Note that the sensitivity list must contain the clock and reset signals so that it executes when one of these signals changes value.

However, it should not contain the other signals used within the process, as an event on any other signal will not cause any part of the process to execute within the code structure shown here.

A rising or falling edge can be detected using a function call or an event expression. Function calls are generally more precise and more readable:

`rising_edge(CLK)` should be used instead of `(CLK'event and CLK = '1')`

`falling_edge(CLK)` should be used instead of `(CLK'event and CLK = '0')`

## Synchronous Feedback

```

library IEEE;
use IEEE.std_logic_1164.all;

entity MUXFF is
port (CLK, RST : in std_logic;
      D, ENB   : in std_logic;
      OP       : out std_logic;
end MUXFF;

architecture RTL of MUXFF is
begin

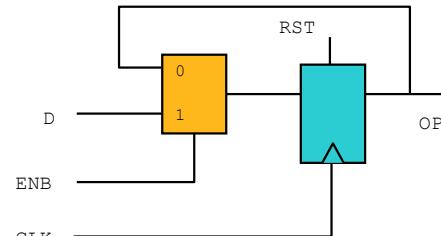
SYNCF: process (CLK, RST)
begin
  if RST = '1' then
    OP <= '0';
  elsif rising_edge(CLK) then
    if ENB = '1' then
      OP <= D;
    end if;
  end if;
end process SYNCF;

end RTL;

```

Incomplete assignment in a *clocked* process implies synchronous feedback

- Implicit read of the output port



The complete assignment only required for *combinational* processes, not *clocked* processes

Synthesis



211 © Cadence Design Systems, Inc. All rights reserved.

Incomplete assignments in combinational logic infer latches in synthesis. An incomplete assignment in a clocked process infers synchronous feedback in synchronous logic. If you do not update the output value, it does not change, and the register inferred by the clocked process stores the value between clock cycles. In the code, if ENB is 1, we assign OP with D; nothing is performed, and the value is stored between clock cycles, inferring synchronous feedback as shown in the hardware.

This is also how you can describe clock enables. If you do not have to operate on every edge of the clock, place the condition after the clock edge expression. This is much safer than creating a new “gated” signal and using this in the process edge expression. Gated clocks cause severe implementation problems in both ASIC and FPGA technologies.

Note that synchronous feedback allows us to implicitly read the value of an output port without needing to declare an intermediate signal. Adding the following lines to the if statement would be a compilation error.

```

else
  OP <= OP; -- error

```

Since OP is an output port, it cannot be read in versions of VHDL before VHDL2008.

## Summary Quiz: Definition of RTL

Find and fix the coding mistakes

```
ONE: process (CLK, RESET)
begin
    if RESET'event and RESET = '1' then
        OP <= '0';
    elsif CLK = '1' then
        OP <= D;
    end if;
end process ONE;
```

```
TWO: process (CLK, RESET)
begin
    if RESET = '1' then
        OP <= '0';
    end if;
    if rising_edge(CLK) then
        OP <= D;
    end if;
end process TWO;
```

```
THREE: process (all)
begin
    ADD <= A;
    if ENABLE = '1' then
        ADD <= A + B;
        SUB <= A - B;
    end if;
end process THREE;
```

```
FOUR: process (CLK, RESET, COUNT)
begin
    if RESET = '1' then
        COUNT <= (others => '0');
    elsif rising_edge(CLK) then
        if COUNT = 10 then
            COUNT <= (others => '0');
        else
            COUNT <= COUNT + 1;
        end if;
    end if;
    OP <= COUNT;
end process FOUR;
```

```
FIVE_A: process (CLK, RESET)
begin
    if RESET = '0' then
        OP1 <= (others => '0');
        OP2 <= (others => '0');
    elsif rising_edge(CLK) then
        OP1 <= D;
    end if;
end process FIVE_A;

FIVE_B: process (OP1)
begin
    OP2 <= OP1 + 5;
end process FIVE_B;
```

212 © Cadence Design Systems, Inc. All rights reserved.



Let's take a short quiz on this module. In the given code snippets, find the mistakes and rectify them.

## Summary Quiz: Definition of RTL (Solutions)

```
ONE: process (CLK, RESET)
Begin

  if RESET'event and RESET = '1' then
    OP <= '0';
  elsif CLK = '1' then
    OP <= D;
  end if;
end process ONE;
```

```
ONE: process (CLK, RESET)
begin
  if RESET = '1' then
    OP <= '0';
  elsif rising_edge(CLK) then
    OP <= D;
  end if;
end process ONE;
```

```
TWO: process (CLK, RESET)
begin
  if RESET = '1' then
    OP <= '0';
  end if;
  if rising_edge(CLK) then
    OP <= D;
  end if;
end process TWO;
```

```
TWO: process (CLK, RESET)
begin
  if RESET = '1' then
    OP <= '0';
  elsif rising_edge(CLK) then
    OP <= D;
  end if;
end process TWO;
```

213 © Cadence Design Systems, Inc. All rights reserved.



Process ONE has an edge expression for the RESET and a level-sensitive expression for CLK. This is the wrong way around.

Process TWO has the RESET and CLK assignments as two separate `if` statements. They need to be part of the same `if` statement, using an `elsif` to link them. Otherwise, if we get a rising edge on CLK while RESET is high, the process would execute the assignment of D to OP.

## Summary Quiz: Definition of RTL (Solutions) (continued)

```
THREE: process (all)
begin
  ADD <= A;
  if ENABLE = '1' then
    ADD <= A + B;
    SUB <= A - B;
  end if;
end process THREE;
```

```
FOUR: process (CLK, RESET, COUNT)
begin
  if RESET = '1' then
    COUNT <= (others => '0');
  elsif rising_edge(CLK) then
    if COUNT = 10 then
      COUNT <= (others => '0');
    else
      COUNT <= COUNT + 1;
    end if;
  end if;
  OP <= COUNT;
end process FOUR;
```

```
THREE: process (all)
begin
  ADD <= A;
  SUB <= A;
  if ENABLE = '1' then
    ADD <= A + B;
    SUB <= A - B;
  end if;
end process THREE;
```

```
FOUR: process (CLK, RESET) --COUNT
begin
  if RESET = '1' then
    COUNT <= (others => '0');
  elsif rising_edge(CLK) then
    if COUNT = 10 then
      COUNT <= (others => '0');
    else
      COUNT <= COUNT + 1;
    end if;
  end if;
end process FOUR;
OP <= COUNT;
```

214 © Cadence Design Systems, Inc. All rights reserved.



Process THREE has a default assignment for ADD but not for SUB. It is possible for a single process to infer combinational logic on one signal and latches on another if there is an incomplete assignment to a signal.

Process FOUR has two problems. Firstly, we have COUNT in the sensitivity list of a registered process, which is redundant. Secondly, the assignment of COUNT to OP is within the registered process, which would infer registers on both COUNT and OP. We may want two sets of registers (output pipelining), but we would expect the reset values to be specified for OP in the reset branch of the `if` statement. We don't have these, so we can assume the assignment to OP should be outside the process.

## Summary Quiz: Definition of RTL (Solutions) (continued)

```
FIVE_A: process (CLK, RESET)
begin
  if RESET = '0' then
    OP1 <= (others => '0');
    OP2 <= (others => '0');
  elsif rising_edge(CLK) then
    OP1 <= D;
  end if;
end process FIVE_A;

FIVE_B: process (OP1)
begin
  OP2 <= OP1 + 5;  --if any bit on OP2 goes '1'
end process FIVE_B; --result goes 'x'
```

```
FIVE_A: process (CLK, RESET)
begin
  if RESET = '0' then
    OP1 <= (others => '0');
    --OP2 <= (others => '0');

  elsif rising_edge(CLK) then
    OP1 <= D;
  end if;
end process FIVE_A;

FIVE_B: process (OP1)
begin
  OP2 <= OP1 + 5;
end process FIVE_B;
```



Processes FIVE have an assignment to OP2 in the reset branch of the registered process FIVE\_A, but no assignment in the clock branch. OP2 is driven in a separate combinational process FIVE\_B. The problem here is we have two drivers on OP2. One driver from FIVE\_A, which drives '0' from the reset detection onwards, and one driver from FIVE\_B, which adds 5 to OP1. The two drivers require a resolution function to determine the final value on OP2. When FIVE\_B drives '0' on a bit of OP2, the result will be '0', but when FIVE\_B drives '1' on a bit of OP2, the result will be 'X' (due to the continual driver of '0' from FIVE\_A. So we have bus contention. As OP2 is only reset in FIVE\_A and not assigned in the clock branch, we can assume the reset assignment is an error and remove it.

## Lab

### Lab 13-1 Alarm Register

216 © Cadence Design Systems, Inc. All rights reserved.



With this knowledge, you will be able to execute the Alarm Register lab. Please refer to the lab book for details.

# Synthesis of Mathematical Operators

**Module** **14**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

This module deals with various mathematical operators and concepts surrounding them.

## Module Objectives

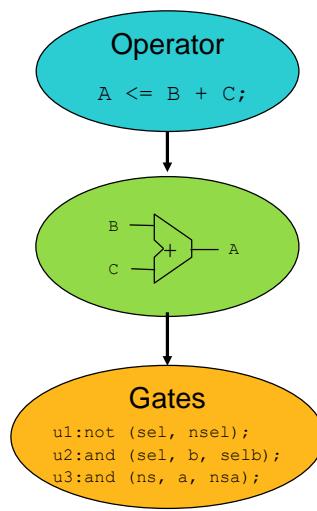
In this module, you

- List the issues associated with both simulation and synthesis of mathematical operators
- Discuss high-level synthesis
- Define resource sharing
- Use intermediate variables to optimize the hardware
- List various architecture optimization techniques
- Perform operations on vectors
- Compare and contrast arithmetic and vector operation
- Describe initialization and reset concepts
- Implement operators using macro-cells



In this module, you list the issues associated with both simulation and synthesis of mathematical operators, discuss high-level synthesis, define resource sharing, use intermediate variables to optimize the hardware, list various architecture optimization techniques, perform operations on vectors, compare and contrast arithmetic and vector operation, describe initialization and reset concepts, implement operators using macro-cells.

## High-Level Synthesis



Operators can have a significant effect on the resulting hardware

Most synthesis tools can identify operators from other structures

- Operators may be isolated from the rest of logic

Operators then optimized separately during synthesis



Operators can have a disproportionate effect on the size and performance of synthesized hardware. A model may contain many lines of signal assignments, if, case, and for loop constructs that synthesize to small areas of random logic. However, a single arithmetical operator in the same model may end up occupying a large proportion of the final silicon and being the chief limitation on the performance of the logic.

Therefore, synthesis tools can isolate operators from other logic to apply separate optimization strategies.

## Optimization of Operators

Synthesizer options for building the operator:

- From discrete gates
- By selecting an implementation from a library of pre-built macro-cells

Isolation of operator allows control of implementation, e.g., for an adder:

- Ripple-carry architecture
- Carry-look-ahead architecture

Implementation options:

- No choice – only one architecture
- Specify architecture in code with comments
- Automatic choice based on speed goals

The designer can optimize the number of operators built:

- Operator sharing



The synthesizer can build the operators in two ways. For example, the synthesis tool can map an 8-bit add operation to a sea of small discrete combinational gates. It can also choose a single 8-bit adder cell from a vendor's library if this will give better results.

Generally, we isolate the operators from the rest of the code to have control over the implementation. For example, an adder can be implemented as a ripple carry or a carry look-ahead architecture. Some synthesis tools may only be able to build one architecture. Usually, a ripple carry. If you require a carry look-ahead architecture, you will need to define this with the appropriate architecture of boolean functions within the VHDL code.

- Other tools may allow you to specify that a specific operation on a specific line of VHDL code must be built with a given architecture.
- Some tools can automatically choose the architecture based on the design's area and speed goals (this can be the most efficient method).

Significant efficiency gains can be obtained using a single arithmetic hardware resource to implement multiple arithmetic operations inferred in the model. This is called operator sharing.

## Mathematical Operators: Resource Sharing Concepts

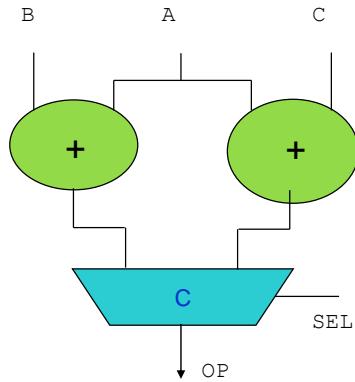
```
if SEL = '1' then  
    OP <= A + B;  
else  
    OP <= A + C;  
end if;
```



How many adders does this code represent?

Question

This is the hardware architecture implied by the code



221 © Cadence Design Systems, Inc. All rights reserved.

Let us look at resource sharing with an example.

In this code, we have two adder operators, where they are mutually exclusive, i.e., only one of the operators may be generating the output OP at any one time, as, if and else both can't be active at the same time.

This code infers the hardware shown on the right, which uses two adders though only one is active at any time. The same logic can be rewritten such that a single adder is used to perform operations with a different set of operands at different times, as shown in the next slide.

## Mathematical Operators: Manual Resource Sharing

```
if SEL = '1' then
    OP <= A + B;
else
    OP <= A + C;
end if;
```

Code:

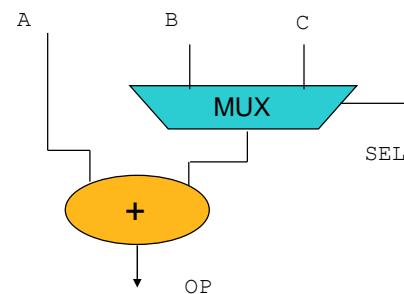
```
variable TMP : integer;
...
if SEL = '1' then
    TMP := B;
else
    TMP := C;
end if;
OP <= A + TMP;
```



How could you re-write this code to use only one + operator?

Question

Hardware Architecture:



222 © Cadence Design Systems, Inc. All rights reserved.

As discussed on the previous slide, the code on the top can be implemented so that it infers only one adder instead of two, as shown in the below code. This code declares a variable TMP assigned with value B or C depending on the value of SEL, thereby inferring a mux. The output of this mux is given as input to the adder along with A. So, if SEL equals '1', TMP is assigned to B, and hence OP equals A + B. If SEL equals '0', TMP is assigned with C, and hence, OP equals A plus C. Which of the two architectures is the fastest depends on the critical path.

If SEL to OP is critical, then the one with two adders is the fastest!

Some synthesis tools can perform automatic resource sharing for the area only.

- These would not have chosen the one with two adders if the critical path was SEL → OP.

Some can also do resource sharing for speed.

## Use Intermediate Variables

```
P <= A + B + C;
Q <= A + B + D;
R <= C + D;
```



```
P <= A + B + C;
Q <= D + A + B;
R <= C + D;
```



```
P <= A + B + C;
Q <= B + A + D;
R <= C + D;
```



```
T := A + B;
P <= T + C;
Q <= T + D;
R <= C + D;
```

Re-code to save one adder

- Some tools perform this optimization automatically
- Also, it depends on the order and position of operands in an expression



Automatic resource sharing  
may be dependent on operand  
order and position

Beware



Use intermediate terms to  
force resource sharing

Tip



Some synthesis tools will explicitly build an operator in the hardware for each operator specified in the code.

Other synthesis tools can automatically perform arithmetic optimization, but this depends on the order and position of the operands in the expression.

For example, tools may be able to identify  $A + B$  as a common sub-term for  $P$  and  $Q$  only if:

- $A + B$  appears in the same position in the expression for  $P$  and  $Q$ 
  - i.e.,  $Q <= D + A + B$ ; may prevent sub-term sharing
- $A + B$  appears in the same order in the expression for  $P$  and  $Q$ 
  - i.e.,  $Q <= B + A + D$ ; may prevent sub-term sharing

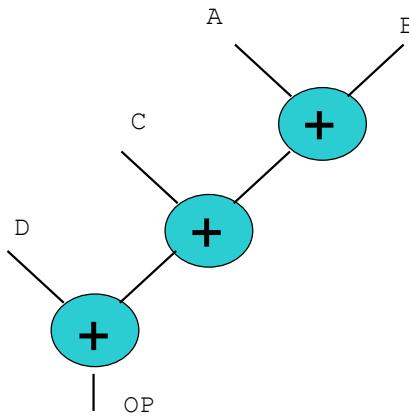
Since the rules are strict, it is better to use intermediate variables to force resource sharing. Intermediate variables are also more efficient for simulation by removing the recalculation of arithmetic expressions.

In the code on the slide, we see that a variable  $T$  is declared and is assigned with a common sub-term expression,  $A + B$  in this case. The result is stored in  $T$ , which is given as input to the adders containing the common sub-term,  $P$  and  $Q$ . This removes the need for additional adders, making the code more efficient.

## Mathematical Optimization

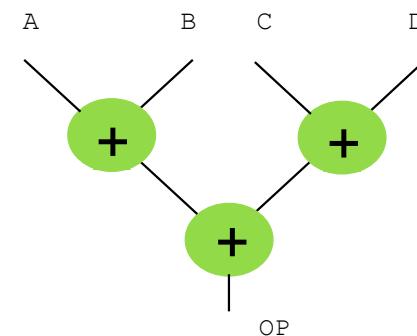
Chain (serial) structure

```
OP <= A + B + C + D;
```



Tree (parallel) structure

```
OP <= (A + B) + (C + D);
```



224 © Cadence Design Systems, Inc. All rights reserved.



In most synthesis tools, you will need to use parentheses to generate these different structures. Some synthesis tools may change such an architecture automatically based on the constraints. In the example on the slide, we see that the use of parenthesis for addition infers adders in a tree structure, as against chain structure inferred by addition without any parenthesis.

## Operations on std\_logic Arrays (Review)

A bit of an issue!

Logical

- Defined in std\_logic\_1164 package

Relational

- Different vector lengths

Arithmetic

- Not defined by default in the language

std\_logic\_vector

- How do we represent positive and negative values?

225 © Cadence Design Systems, Inc. All rights reserved.



As discussed in the previous modules, all operations on array types are a bit of an issue! We have already seen how the built-in relational operators can return “strange” answers when the operands are of different lengths. This is because VHDL does not recognize that vectors represent numerical values.

Arithmetic operators on vectors are not defined at all.

Before working with arithmetic operators on vector types, we need a way to distinguish positive and negative values.

We will be looking at a workaround for these problems.

## signed and unsigned Types (Review)

```

variable A, B, SUM : std_logic_vector(3 downto 0);
...
A := "1001"; -- no numerical representation
B := "0011"; -- no numerical representation
SUM := A + B; -- ?
if A > B then -- true for default operator
  ...

variable A, B, SUM : signed(3 downto 0);
...
A := "1001"; -- -7
B := "0011"; -- +3
SUM := A + B; -- "1100" = -4
if A > B then -- should be false
  ...
A, B, SUM : unsigned(3 downto 0);
...
A := "1001"; -- +9
B := "0011"; -- +3
SUM := A + B; -- "1100" = +12
if A > B then -- should be true...
  ...

```

- unsigned
  - Positive values only
  - Binary
  - 4-bit vector represents 0 to 15
- signed
  - Positive and negative values
  - 2's complement
  - MSB denotes the sign
    - If 0, positive
    - If 1, negative
  - 4-bit vector represents -8 to +7
- Identical types to std\_logic\_vector
  - Arrays of std\_logic

226 © Cadence Design Systems, Inc. All rights reserved.



One key issue with using arithmetic or relational operators on array types is the representation of positive and negative data. Positive /unsigned data is easy to represent. We can use simple binary values; we can use 4-bits to represent data from 0 to 15. But the representation of data that can take both positive and negative values, i.e., signed data, is more difficult. In a signed case, the MSB bit represents the sign information; hence, 4-bit can represent data from -7 to +8. We have several conventions for representing signed data in hardware design, one of the most common being 2's complement.

In 2's complement, the value of the Most Significant Bit of a vector tells us whether the value is positive when MSB equals '0' or negative when MSB = '1'. To take a positive value and convert it to its negative, we invert the vector value and add one.

Arithmetic, relational and conversion operations will behave differently depending on if data is signed or unsigned. Therefore, we need some way of distinguishing between signed and unsigned data and to overload the arithmetic, relational and conversion operators accordingly.

To do this, we declare two new types – signed and unsigned. These have identical type declarations as std\_logic\_vector, as they are both arrays of std\_logic but are treated differently by VHDL operators.

In the code on the slide, we see that for given values of A and B, the operation A + B gives different results for different type declaration. For the values, A equals 1 0 0 1 and B equals 0 0 1 1; if A and B are declared as std\_logic\_vector, addition is not defined as the values are not interpreted as numerical representation. If A and B are declared as signed, A + B implies -7 + 3, which yields the result as -4 since, here, MSB denotes the sign of the number. If A and B are declared unsigned, then A + B implies 9 + 3, which yields 12 as the answer.

## Arithmetic Package Use (Clocked)

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

...
signal COUNT : unsigned(3 downto 0);
...
process (CLK, RESET)
begin
    if RESET = '1' then
        COUNT <= (others => '0');
    elsif rising_edge(CLK) then
        if COUNT = 9 then
            COUNT <= (OTHERS => '0');
        else
            COUNT <= COUNT + 1;
        end if;
    end if;
end process;
OP <= COUNT;
...

```

Compare vector and integer  
Assignment type *must* be vector  
Add vector and integer

Allows arithmetic on vectors, integers, and combinations

- Vectors must be defined as `signed` or `unsigned`
- Or use type conversion

The package allows mixing integer and array types

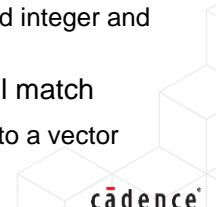
- In relational and arithmetic operators

For example, you can use

- Relational operators to compare integer and vector
- Arithmetic operators to add integer and vector

Assignment types must still match

- Cannot assign an integer to a vector



Accessing arithmetic packages allows you to apply arithmetic and relational operators to array types. `numeric_std` only allows operations on `signed` or `unsigned` types. Therefore you must either declare all your vector objects as being of these types or use type conversion to convert `signed/unsigned` to `std_logic_vector` and vice-versa (see next slide).

Note that the packages allow you to mix array and `integer` types in relational and arithmetic operations, e.g., permitting you to compare a vector to an integer and add an integer to a vector. However, the assignment types must match and cannot assign an integer to a vector.

In VHDL, prior to VHDL2008, you could not read the value of an output port. Therefore, as in this example, we must use a local signal for the counter. This allows us to compare the count to the maximum value and increment the count. A concurrent signal assignment is used to update the output `OP` from the local `COUNT`. If we tried using the output directly in the counter, e.g., `OP` equals `OP` plus 1, we would get compilation errors for attempting to read `OP`.

## Mixing std\_logic Array Types (Review)

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
architecture BEHAVIORAL of CRT is
    signal A : std_logic_vector(7 downto 0);
    signal B : signed(7 downto 0);
    signal C : unsigned(15 downto 0);
begin
    B <= A;      X  ROR type mismatch
    B <= signed(A);
    A <= std_logic_vector(B);
    C <= unsigned(A) and unsigned(B);
end BEHAVIORAL;

```

Convert std\_logic\_vector to signed and vice-versa

Convert signed and std\_logic\_vector to unsigned

- signed, unsigned, and std\_logic\_vector are all arrays of std\_logic
- Can't freely assign to each other
- VHDL treats these as *closely-related types*
  - Allows simple type conversion  
 $T \leq \langle T \text{ type} \rangle(S);$
- Array sizes must still match



Beware

Not applicable to other types.  
More details in Module 19  
Advanced Data Types



228 © Cadence Design Systems, Inc. All rights reserved.

Although signed, unsigned, and std\_logic\_vector are all declared as arrays of std\_logic, the strict typing of VHDL does not allow us to assign these types to each other freely.

However, VHDL recognizes that these types are very similar (closely-related in language terms) and provides a mechanism to convert between the types: easily

```
target <= <target_type>(source);
```

where target and source are different closely-related types (unsigned, signed, std\_logic\_vector). The variable assignment could also be used (assuming the target is a variable).

**WARNING:** This form of type conversion cannot be used to convert between other types, e.g., integer to std\_logic\_vector or std\_logic\_vector to string. These types are not closely related.

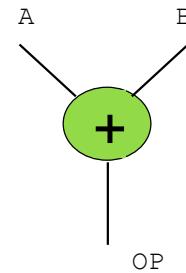
More details on types and type conversion are in the Advanced Data Types module.

We see that B can't be assigned with A as a types mismatch in the code. However, A can be typecast to signed and then assigned to B, as B is of the type signed. Similarly, B can be type converted to std\_logic\_vector and then assigned to A, as A is of the type std\_logic\_vector. Also, since C is of unsigned type, we typecast B and C to unsigned and then perform AND operation.

## Arithmetic with Integers (Review)

```
entity ADD is
    port( A, B : in integer range 0 to 7;
          OP   : out integer range 0 to 15);
end ADD;

architecture INT_ARITH of ADD is
begin
    OP <= A + B;
end INT_ARITH;
```



What is the benefit of having the integer range:

- a. Generally?
- b. For synthesis?

Arithmetic operators are defined by default for integer types in VHDL. Using integer-type signals can make arithmetic much easier.

The integer ranges will be checked for out-of-range assignments at analysis and simulation time. The use of the range on integers improves the efficiency of the hardware inferred.

## Integer Versus Vector-Based Arithmetic

```

library IEEE;
use IEEE.std_logic_1164.all;

entity INTADD is
  port(A, B : in integer range 0 to 7;
       OP : out integer range 0 to 15);
end INTADD;

architecture RTL of INTADD is
begin
  OP <= A + B;
end RTL;

```

- Simple and simulates faster
- Type ranges can be mixed
- Range maximum 32 bits signed
- No un-initialized values
  - Possible issues in/after synthesis
- Logical/bit-wise operations difficult

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity VECADD is
  port(A, B : in unsigned(2 downto 0);
       OP : out unsigned(3 downto 0));
end VECADD;

architecture RTL of VECADD is
begin
  OP <= ('0' & A) + ('0' & B);
end RTL;

```

- Requires arithmetic package numeric\_std
- Arrays must be the same size
- No maximum vector size
- Bit-wise/logical operations easier
- Explicit un-initialized value 'U'
  - Fewer issues in/after synthesis



Arithmetic can be carried out in several ways in VHDL.

Arithmetic using integer types is simple. No additional packages are required. Integer objects can be declared with different ranges and freely mixed in arithmetical expressions. However, integer types are limited to a maximum of 32-bits signed – i.e., range -2,147,483,648 to 2,147,483,647. Integers cannot represent un-initialized, unknown, or high-impedance values, limiting their usefulness for clocked or tri-state/bi-directional logic. A synthesis tool will convert the integer types to vectors, so integer ranges must be carefully selected (powers of 2 usually). There may be type conversion issues in gate-level simulation. Logical/bit-wise operations are difficult to implement with integer types.

Arithmetic using vector types requires the use of an arithmetic package called numeric\_std. This can lead to type conversion issues with mixing signed/unsigned and std\_logic\_vector types. Array sizes must be the same on both sides of an assignment. The vector size is not limited. To save the result of adding two 3-bit numbers requires a 4-bit number. Since the target object has to be 4 bits, the operands must be expanded to 4-bits also, here using concatenation. For example, this is easy for addition but less straightforward for multiplication. The maximum length of a vector is 2,147,483,647, so there are no problems modeling 64 or 256-bit vectors. Since vector types are not converted during synthesis, there are fewer issues in gate-level simulation after synthesis. Bit-wise/logical operators are easier to perform with this data type.

## Initialization and Higher-Level Types

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity INTEGER_COUNTER is
  port (CLK : in std_logic;
        OP : out unsigned(3 downto 0);
end INTEGER_COUNTER;

architecture RTL of INTEGER_COUNTER is

  signal COUNT : integer range 0 to 15;
begin
  process (CLK)
  begin
    if rising_edge(CLK) then
      if (COUNT >= 9) then
        COUNT <= 0;
      else
        COUNT <= COUNT + 1;
      end if;
    end if;
  end process;

  OP <= to_unsigned(COUNT, OP'length);
end RTL;

```

231 © Cadence Design Systems, Inc. All rights reserved.

What is the starting value for COUNT?

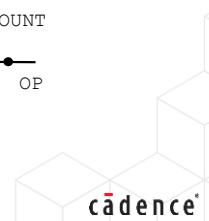
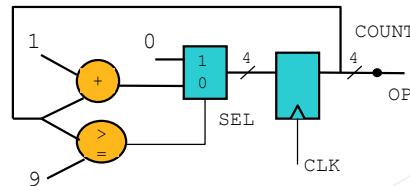
- Zero for type integer

RTL

- Initial value is 0
- Missing reset not detected in simulation

Gates

- COUNT mapped to std\_logic\_vector by synthesis
- Initial value is UUUU which propagates through design
- Missing reset only detected in gate-level simulation



The code on the slide is an up counter discussed in previous modules. The hardware inferred by this code is shown on the right. Remember signals initialize to the leftmost value in their type declaration unless a specific initial value is assigned.

Common problems in the simulation of synthesized hardware are due to registers that are not reset properly.

This can be missed during RTL simulation because a registered signal may be of a type whose initial value is legitimate, e.g., 0 for an integer type, so COUNT initializes to 0 in RTL simulation and counts correctly up to 9.

Synthesis maps COUNT to a 4-bit std\_logic\_vector, which initializes to "UUUU" since 'U' is the leftmost value of std\_logic.

Gate-level simulation libraries always propagate un-initialized values. Therefore, COUNT will remain at "UUUU" throughout gate-level simulation in the absence of an explicit reset, highlighting the initialization problem.

Next, we see how we could fix the problem.

## Initial Values

```

library IEEE;
use IEEE.std_logic_1164.all;

entity INTEGER_COUNT is
  port (CLK : in std_logic;
        OP  : out unsigned(3 downto 0));
end INTEGER_COUNT;

architecture RTL of INTEGER_COUNT is

  signal COUNT : integer range 0 to 15 := 7;
begin
  process (CLK)
  begin
    if rising_edge(CLK) then
      if (COUNT >= 9) then
        COUNT <= 0;
      else
        COUNT <= COUNT + 1;
      end if;
    end if;
  end process;
  OP <= to_unsigned(COUNT, OP'length);
end RTL;

```

- Initial values hide initialization problems in RTL simulation
- In synthesis, the initial value is ignored
  - With no explicit reset, COUNT would initialize to "UUUU" which propagates through design
  - Missing reset only detected in gate-level simulation



Question

What does this initial value mean:

- For simulation?
- For synthesis?



232 © Cadence Design Systems, Inc. All rights reserved.

In a language like VHDL, a signal can be given an initial value when it is declared.

Initial values are only a simulation concept and should not be used in synthesizable code since they can hide initialization problems with your logic.

For this code, the initial value sets COUNT to 7 at the start of the simulation, and the RTL simulation would work.

In synthesis, the initial value is ignored, and COUNT would be synthesized to a `std_logic_vector` which, due to the lack of reset, would remain at "UUUU."

So, using the initial value in RTL code has hidden an initialization problem that is not detected until gate-level simulation.

Initial values are helpful in testbench and behavioral modeling.

## Resets for Synthesizable Code

```
library IEEE;
use IEEE.std_logic_1164.all;

entity INTEGER_COUNT is
  port (CLK, RESET : in std_logic;
        OP        : out unsigned(3 downto 0));
end INTEGER_COUNT;
```

- You should explicitly reset synthesizable registered processes
  - To avoid initialization problems
- Add reset port to
  - Entity
  - Component declarations
  - Map in signal instantiations
- Some rules
  - Synchronous or asynchronous
  - Internally or externally generated
  - Reset all or reset key registers, etc.

```
architecture RTL of INTEGER_COUNT is
  signal COUNT : integer range 0 to 15;
begin
  process (CLK, RST)
  begin
    if RESET = '1' then
      COUNT <= 0;
    elsif rising_edge(CLK) then
      if (COUNT >= 9) then
        COUNT <= 0;
      else
        COUNT <= COUNT + 1;
      end if;
    end if;
  end process;
  OP <= to_unsigned(COUNT, OP'length);
end RTL;
```

233 © Cadence Design Systems, Inc. All rights reserved.



To remove the initialization problem in gate-level code, you must add an explicit reset to the registered process to get the design to a known state.

- This means adding a reset port to the entity and related component declarations and mapping this to an appropriate reset signal in component instantiations.

Your company may have design rules for the use of resets, e.g.,

- Synchronous or asynchronous
- Internally generated or externally generated
- Reset all registers or reset key registers, etc.

In the code, we see that we explicitly initialize the count value of the UP-counter to zero on reset. Hence, even if the signal is of `std_logic_vector` type, it is initialized to zero as output by the synthesizer. Otherwise, it would remain at U and propagate, causing initialization problems.

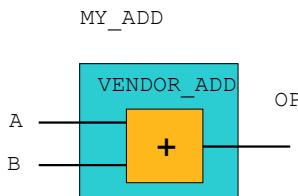
## Instantiation of Macro-Cells

```
OP <= A + B;
```

Or

```
U1 : VENDOR_ADD port map(A,B,OP);
```

```
U1 : MY_ADD port map(A,B,OP);
```



When?

- Tool cannot infer macro-cell from library
- Optimized cell available

Create your own cell names and use a hierarchy

Issues

- Code less readable
- Not technology independent

Also, for

- Memory blocks
- Embedded microprocessor
- Third-party IP



If a synthesis tool cannot generate a vendor's macro from a statement such as  $C \leq A+B$ , it is necessary to make an instance of the macro to obtain the required optimal results.

While this makes the code technology-independent, it does give more control over the resulting synthesized design.

This could also be necessary with tools when the module generation capabilities do not give the desired results, such as when the resulting design is too large or too slow.

To make the design more technology-independent, it is possible to instantiate the vendor's macro within your cell name and instantiate your technology-independent cell name in the actual design. In this case, it is easier to change to new technology later, as you are only required to modify the definition of the user-defined cell rather than change every instantiation of the macro.

## Synthesis of Mathematical Operators: Quiz

1. In what two ways can mathematical operators be built by a synthesis tool?
2. What are the two disadvantages of directly instantiating macro-cells instead of using mathematical operators?
3. On what types does the IEEE numeric\_std package allow mathematical operations?
4. What are the advantages of using integer types for arithmetic operations?



Let us take a short quiz on this module. In what two ways may mathematical operators be built by a synthesis tool? What are the two disadvantages of directly instantiating macro-cells instead of using mathematical operators? On what types does the IEEE numeric\_std package allow mathematical operations? What are the advantages of using integer types for arithmetic operations?

## Synthesis of Mathematical Operators: Quiz (Solution)

1. In what two ways can mathematical operators be built by a synthesis tool?
  - Either built from basic gates or mapped to a pre-built macro-cell.
2. What are the two disadvantages of directly instantiating macro-cells instead of using mathematical operators?
  - The code may be less readable.
  - The code becomes technology-specific.
3. On what types does the IEEE numeric\_std package allow mathematical operations?
  - Signed, unsigned, and integer (natural) only.
4. What are the advantages of using integer types for arithmetic operations?
  - Simple and faster with the ability to mix ranges.



Here is the solution. Synthesis tool builds mathematical operator either from basic gates or maps it to a pre-built macro cell.

The disadvantages of directly instantiating macro-cells instead of using mathematical operators are that:

- The code may be less readable.
- The code becomes technology-specific.

The IEEE numeric\_std package only allows mathematical operations on signed, unsigned, and integer types.

The advantage of using integer types for arithmetic operations is that it is simple and faster with the ability to mix ranges.

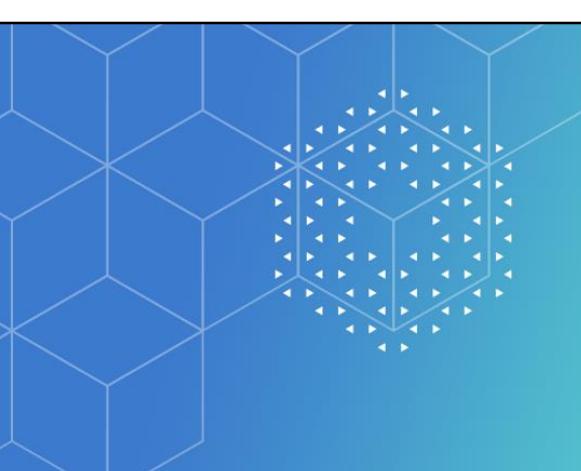
## Lab

### Lab 14-1 Pulse Generator

237 © Cadence Design Systems, Inc. All rights reserved.



With this knowledge, you will be able to execute the Pulse Generator lab. Please refer to the lab book for details.



## Finite State Machine (FSM) Design and Synthesis

**Module** **15**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we look at the design of RTL using FSMs and the synthesis issues surrounding them.

## Module Objectives

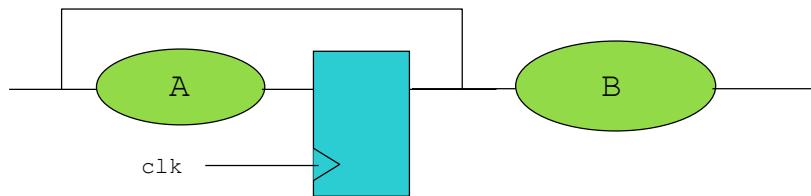
In this module, you

- Design Finite State Machines (FSMs) for synthesis
- Define how VHDL constructs relate to hardware and illustrate how coding style will affect synthesis results
- Describe controlling registers
- Define state machines
- Describe state encoding
- List synthesis directives



*This page does not contain notes.*

## Inferring Registers



```
process (CLK)
begin
  if rising_edge(CLK) then
    -- logic in blocks A and B
  end if;
end process;
```



Does this code describe this logic?

Question

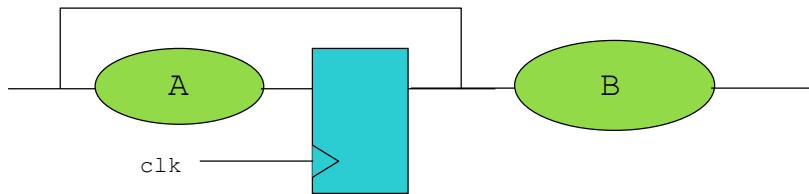
240 © Cadence Design Systems, Inc. All rights reserved.



When writing your RTL code, it is important to correctly partition the functionality between processes to control the location of registers in the design.

Let us look at an example. The code on the slide doesn't infer the hardware shown on the slide, as the code has logic B registered within the process, whereas the hardware contains B as combinational logic.

## How Should It Be Written?



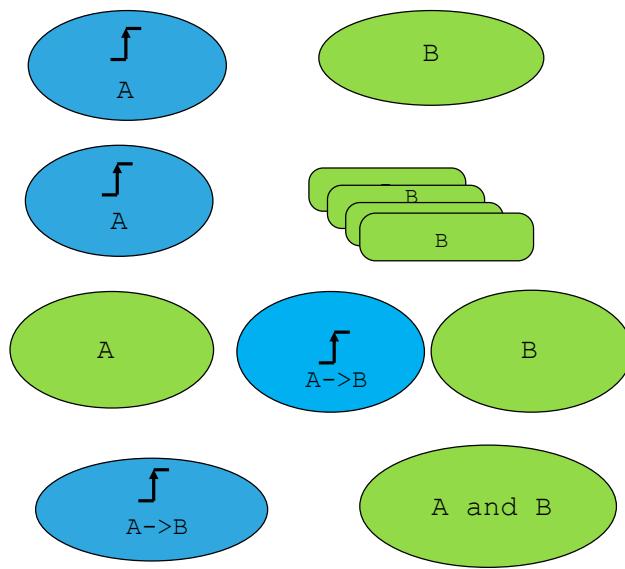
How should the code be structured?  
(There are at least three options!)

Question

Let us determine how the code should be structured to infer the hardware shown on the slide.

- There are at least three options, which are discussed in the next slide.

## FSM Design: How Should It Be Written?



### Option 1

- Registered process A
- Combinational process B

### Option 2

- Registered process A
- Concurrent statements B

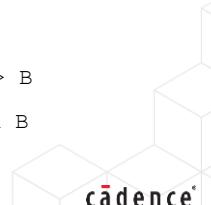
### Option 3

- Combinational process A
- Registered process signals  $A \rightarrow B$
- Combinational process B

### Option 4

- Registered process signals  $A \rightarrow B$
- Combinational processes A and B

242 © Cadence Design Systems, Inc. All rights reserved.



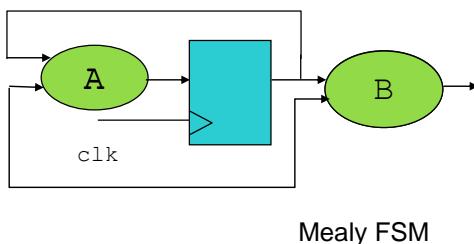
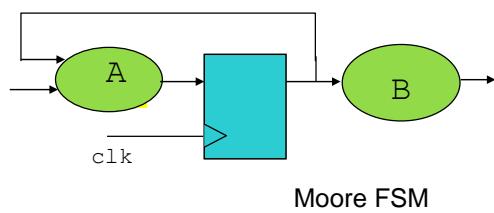
This slide shows how the code can be structured to infer the hardware shown on the previous slide. Option 1 is fairly straightforward. All the outputs from the combinational logic A are registered. Therefore, the combinational logic and registers can be placed in one registered process.

Option 2 is a variant of Option 1. Combinational logic can be described in a combinational process or as several concurrent statements (typically one statement per output) for any options here.

Option 3, logic A is placed in a separate combinational process. Registers are inferred in a separate process  $A \rightarrow B$ , which contains signal assignments. Extra signals must carry information from the combinational logic process A to the registered process  $A \rightarrow B$ . Although this option requires additional signals and processes, it is easier to see which signals are registered and how many registers should be inferred in synthesis.

Option 4 is a progression from Option 3, where the separate combinational processes for A and B are merged into one. This may affect the readability of the design, particularly if A and B are unrelated (have no functionality, inputs, or outputs in common). It is probably not a good partition of the design.

## Finite State Machine (FSM) Review



FSM is a control path logic structure:

- Has the number of predefined states
  - Transition defined by current state and current input
- It has three components:
- State register
    - Stores current state
  - Next state decode logic (A)
    - Decides next state based on current state and inputs
  - Output decode logic (B)
    - Decodes state (or state and inputs) to produce outputs

Outputs from the FSM can be a function of:

- Current state only (Moore)
- Current state and current inputs (Mealy)



We will examine an FSM design as an example of structuring RTL processes. First, we will look at some FSM backgrounds.

An FSM is a control path logic structure. It is clocked (sequential) and has several predefined states that it can be in. The transition between the states is determined by combining the current state and the current inputs to the FSM (examined at a clock edge). Any FSM has three components: a State register that stores the current state; two, next state decode logic that decides the next state based on the current state and inputs; and three, output decode logic that decodes the state to produce output.

The FSM usually has several output status signals that can control logic (e.g., multiplexors).

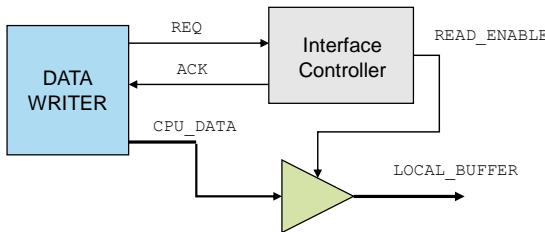
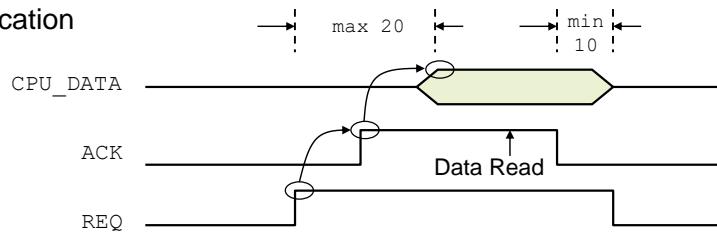
FSM can be of two types, Moore and Mealy, based on how the output is derived.

For a Moore-type FSM, the outputs are derived from the current state only.

For a Mealy-type FSM, the outputs are derived from the current state and the current inputs.

## FSM Design: Bus Interface Controller

Specification



**Controller must:**

- Detect **REQ** high
- Respond with **ACK** high
- Initiate read-off bus at the correct time
  - Generate enable for buffer/mux

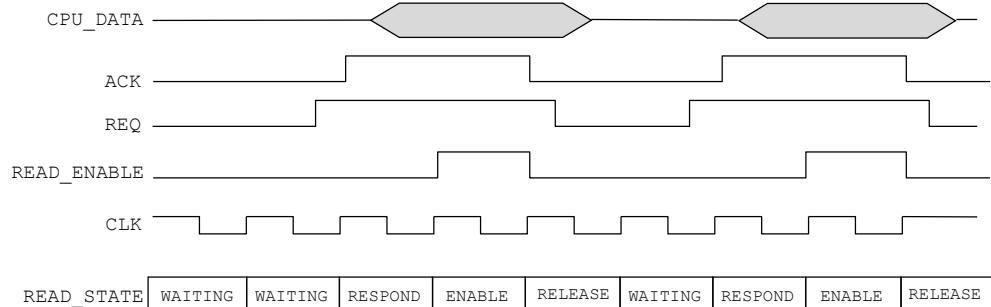
Let us consider the example of the FSM design of the bus controller interface. The specification for a Bus Interface Controller is shown in the waveform diagram.

**REQ** is an external signal which is asserted true just before some data is put onto the bus. The controller will assert the **ACK** line in response and, after at least 20 ns, initiate a read from **CPU\_DATA** into a local buffer.

## Controller: State Allocation

On the rising edge of the clock:

- Sample input and current state
- Determine the next state
- Assign outputs



Beware

245 © Cadence Design Systems, Inc. All rights reserved.



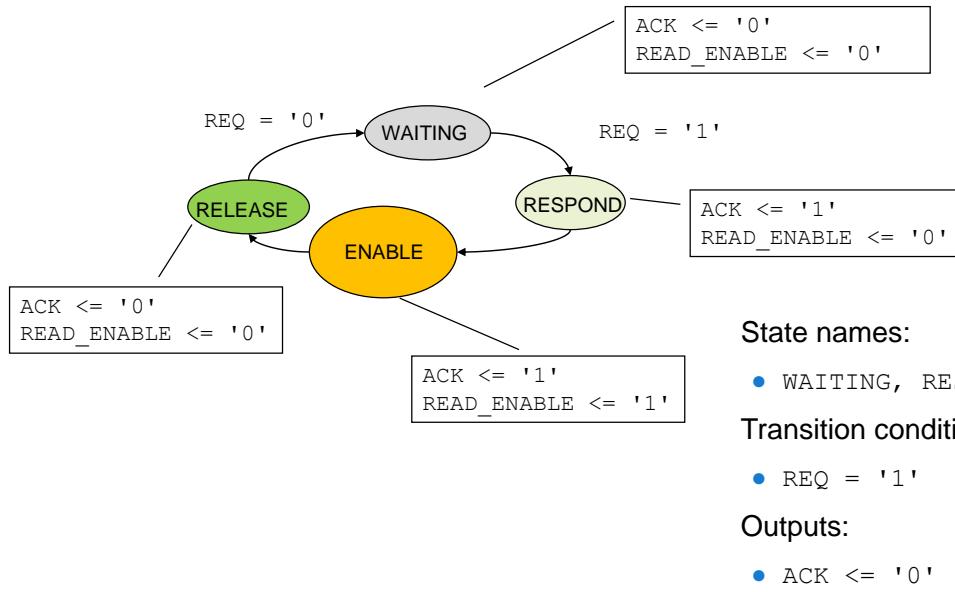
To write an RTL model of the controller, we need to design a state machine.

In this slide, we have added a clock signal and a new signal called `READ_STATE`, which has as many values as the number of states we need; in this case, we need four states: `WAITING`, `RESPOND`, `ENABLE`, and `RELEASE`. This signal will be used to tell us what state we are in. We have allocated the states across the waveform diagram.

<code>WAITING</code>	Waiting for <code>REQ</code> to be '1'
<code>RESPOND</code>	set <code>ACK</code> and wait for 20 ns
<code>ENABLE</code>	set <code>READ_ENABLE</code> and wait for 20 ns
<code>RELEASE</code>	Clear <code>ACK</code>

Note that we only change the state on a clock edge. The clock period, in this case, should not be less than 20 nanoseconds.

## Controller: State Diagram



246 © Cadence Design Systems, Inc. All rights reserved.



FSMs can be specified using a State Diagram. In this abstract example, the FSM has four states.

Input signal conditions dictate how we transition between the different states. Since FSMs are synchronous, these input conditions are checked on the relevant edge of a clock signal. For example, if the input signal `REQ` equals '1' on the clock edge when in-state `WAITING`, the FSM moves to state `RESPOND`. If `REQ` is not '1', the FSM does not change the state. If there is no transition condition, e.g., transition from `RESPOND` to `ENABLE`, the transition occurs on the next clock cycle.

The state diagram also shows the output signal values for each state. For example, in-state `RESPOND`, the output `ACK` is '1', and `READ_ENABLE` is '0'. When the FSM transitions to state `ENABLE` on the next clock edge, both the outputs become '1'.

## FSM: Clocked Process

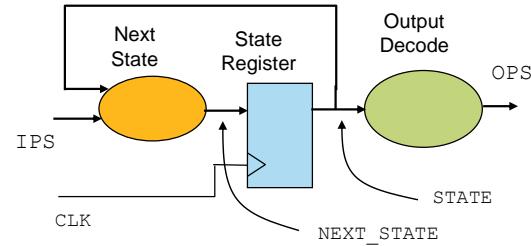
```

entity FSM is
port (REQ      : in std_logic;
      CLK, RESET : in std_logic;
      ACK      : out std_logic;
      READ_ENABLE : out std_logic);
end FSM;

architecture RTL of FSM is
  type STATE_T is (WAITING, RESPOND,
                    ENABLE, RELEASE);
  signal STATE, NEXT_STATE: STATE_T;
begin

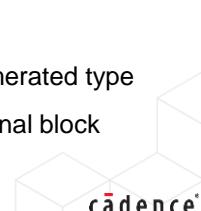
  -- clocked State Register
  STATE_REG: process (CLK, RESET)
  begin
    if RESET = '1' then
      STATE <= WAITING;
    elsif rising_edge(CLK) then
      STATE <= NEXT_STATE;
    end if;
  end process STATE_REG;
...

```



- Partition FSM into three processes:
  - Combinational Next State logic
  - Clocked State Register with reset
  - Combinational Output Decode logic
- State names declared with an enumerated type
- A separate process for each functional block

247 © Cadence Design Systems, Inc. All rights reserved.



An FSM is composed of three functional blocks:

- Next State combinational logic determines the next state from the current state and inputs.
- A State Register to store the current state.
- Output Decode combinational logic to derive the outputs from the current state (Moore FSM).

The state vector is defined using an enumerated type, with an enumerated value for each state in the machine.

Here, we will structure the FSM using a separate process for each FSM functional block.

This slide shows the State Register block implemented with the clocked process STATE\_REG. Note we need a signal NEXT\_STATE to pass the next state from the combinational Next State process to clocked State Register process.

## FSM: Combinational Processes

```
-- generate next state
NEXTSTATE: process(all)
begin
  NEXT_STATE <= STATE;
  case STATE is
    when WAITING =>
      if REQ = '1' then
        NEXT_STATE <= RESPOND;
      end if;
    when RESPOND =>
      NEXT_STATE <= ENABLE;
    when ENABLE =>
      NEXT_STATE <= RELEASE;
    when RELEASE =>
      if REQ = '0' then
        NEXT_STATE <= WAITING;
      end if;
    end case;
  end process NEXTSTATE;
```

```
-- decode outputs from state
OP_DECODE: process(all)
begin
  -- default assignments to avoid latches
  ACK <= '0';
  READ_ENABLE <= '0';
  case STATE is
    when RESPOND =>
      ACK <= '1';
    when ENABLE =>
      ACK <= '1';
      READ_ENABLE <= '1';
    when others =>
      null;
  end case;
end process OP_DECODE;
```

- Combinational Next State logic
- Combinational Output Decode logic

248 © Cadence Design Systems, Inc. All rights reserved.



This slide describes the next state and output decode combinational process. The Next State logic is described using a case statement, where we have a branch for each state of the FSM. The rules of the case statement help us by forcing us to consider functionality for every value of the state, i.e., every state in the FSM.

We use if statements to check transition conditions for each state. Note that the order in which we check the transition conditions is important. Transition conditions in an `if-elsif` statement are checked in the order they appear.

In this example, the Output Decode logic is implemented in a single combinational process. We use default assignments to avoid inferring latches. The default assignment values are those for state WAITING and RELEASE. Hence, we do not have an explicit branch for these – they are covered in the when others branch.

The combinational processes don't generate transparent latches because of the default assignment to NEXT\_STATE at the top of the NEXT STATE process and READ\_ENABLE and ACK in the output decode processes.

## FSM: Alternative Process Structure

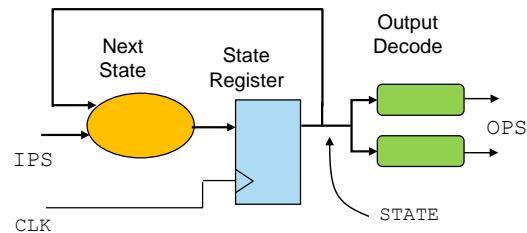
```

FSM2 : process(RESET, CLK)
begin
  if RESET = '1' then
    STATE <= WAITING;
  elsif rising_edge(CLK) then

    case STATE is
      when WAITING =>
        if REQ = '1' then
          STATE <= RESPOND;
        end if;
      when RESPOND =>
        STATE <= ENABLE;
      when ENABLE =>
        STATE <= RELEASE;
      when RELEASE =>
        if REQ = '0' then
          STATE <= WAITING;
        end if;
    end case;

    end if; -- RESET = '1'
  end process FSM2;

```



```

-- concurrent output decode statements
with STATE select
  ACK <= '1' when RESPOND | ENABLE ,
  '0' when others;

READ_ENABLE <= '1' when STATE = ENABLE else
  '0';

```

- Next state and state register processes can be merged

249 © Cadence Design Systems, Inc. All rights reserved.



This slide shows an alternative way of describing this state machine, with a single clocked process defining the next state logic and the flip-flop.

- Note how in this case, we do not need a NEXT\_STATE signal.

We read the current STATE and FSM inputs on the rising edge of the clock to determine if STATE will change on this clock edge. If it does, we assign a new value. If not, STATE is not updated and so remains unchanged. Remember, we do not need default assignments in clocked processes – we are using synchronous feedback with STATE (see the slide Synchronous Feedback in the Definition of RTL Code module).

If all the outputs of a combinational logic block feed straight into a register, then the combinational logic can be merged into the registered process.

The output decode process is implemented as a pair of concurrent statements – selected signal assignments. A separate concurrent statement for each output can be more efficient and readable than implementing the decode in a single process.

## State Encoding

### Default encoding

- $\log_2(n)$  registers, where n-number of states
- Binary sequence
  - WAITING "00"
  - RESPOND "01"
  - ENABLE "10"
  - RELEASE "11"

### Other encodings may be desirable

- Tool specific
- Not always portable
- Not always available

```

type T_STATE is (WAITING, RESPOND,
                  ENABLE, RELEASE);
signal STATE : T_STATE;
...
case STATE is
    when WAITING =>
        ...
    when RESPOND =>
        ...
    when ENABLE =>
        ...
    when RELEASE =>
        ...
end case;

```



We have looked at how state machines can be defined using an enumerated type for the state vector. Let's now consider how we can control the encoding of the state vector in the synthesized implementation.

If an enumerated type is used, the default encoding applied by most tools is to have the minimum number of registers ( $\log_2(n)$ , where n is the number of states) and to encode the enumerated values in a binary sequence from the left-hand side of the type definition, as shown in the example here.

- Some synthesis tools offer a tool-specific way to override this and specify your encoding within the VHDL code.
- There is no standard way to do this, so this method is not very portable.

## Safe State Machines

```

type T_STATE is (IDLE, VALID, ACTIVE);
signal STATE : T_STATE;
...
case STATE is
when IDLE =>
    ...
when VALID =>
    ...
when ACTIVE =>
    ...
end case;

```

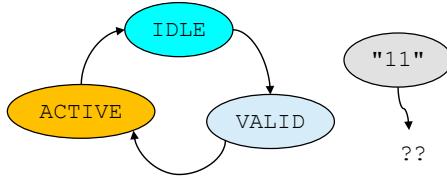
If the number of states is not a power of 2, additional states may be inferred in the synthesis

For example, three states are implemented in two flip-flops

- Binary sequence
  - IDLE "00"
  - VALID "01"
  - ACTIVE "10"

Additional state possible

- Illegal state "11"
- Behavior of state not defined in VHDL



251 © Cadence Design Systems, Inc. All rights reserved.



If the number of states is not a power of 2, additional states may be inferred in the synthesis. For example, a three-state machine, shown in this example, will be implemented with at least two registers. This means that the registers could enter at least one other state, in which the behavior is not defined, and could lead to the state machine locking up.

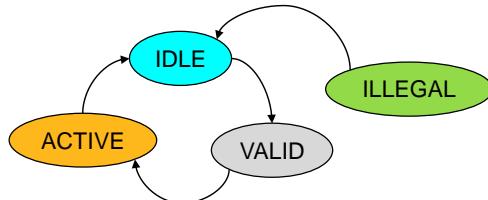
We will now look at methods to allow this illegal behavior to be specified within the VHDL so that the state machine becomes “safe.”

## Specifying Illegal States

```
type T_STATE is (IDLE, VALID, ACTIVE, ILLEGAL);
signal STATE : T_STATE;

...
case STATE is
  when IDLE =>
    ...
  when VALID =>
    ...
  when ACTIVE =>
    ...
  when ILLEGAL =>
    ...
end case;
```

- Can now simulate illegal state
- Can get tedious if a lot of illegal states
- No longer safe if a tool-specific option is used



252 © Cadence Design Systems, Inc. All rights reserved.



One method to specify all illegal states in VHDL is to define them in the enumerated type definition. This method works well, except that:

- It can be tedious if there are a lot of illegal states (e.g., in a 33-state machine).
- It is no longer safe if a tool-specific option is used to change the number of registers used to build the state vector.

## Hand Specifying Encoding in VHDL

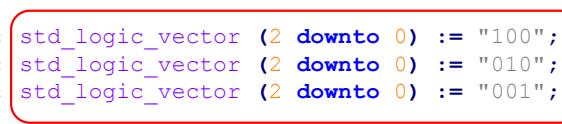
- You can control encoding and illegal conditions from VHDL
- Probably best to use synthesis tool options to optimize encoding

```

signal STATE : std_logic_vector (2 downto 0);

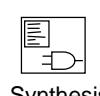
constant IDLE   : std_logic_vector (2 downto 0) := "100";
constant VALID  : std_logic_vector (2 downto 0) := "010";
constant ACTIVE : std_logic_vector (2 downto 0) := "001";
...
case STATE is
  when IDLE =>
    ...
  when VALID =>
    ...
  when ACTIVE =>
    ...
  when others =>
    ...
end case;

```



**Beware**

May not synthesize to a true "one-hot" state encoding



Synthesis

253 © Cadence Design Systems, Inc. All rights reserved.



To define the state vector, you can hand specify the number of registers and their encoding in the VHDL code using a vector rather than an enumerated type.

This method gives the most control over the VHDL code.

Note: With some synthesis tools, this may not give a true one-hot encoding – output decode logic may be created to check other bits are ‘0’ and the ‘one-hot’ bit is ‘1’. True ‘one-hot’ decoding examines a single bit to determine if the FSM is in a particular state. There will usually be a tool-specific method of forcing true one-hot state decoding.

Since state encoding is a key part of FSM optimization in synthesis, your synthesis tool should have several options for specifying the state encoding (Gray code, one-hot, binary, Johnson, etc.). Most synthesis tools will also have an automatic setting that allows the tool to analyze your FSM and the target technology and automatically choose the most efficient encoding.

This automatic option is usually the best choice for synthesizing FSMs.

## Synthesis Directives

```
-- synthesis translate_off
...
-- synthesis translate_on
```

```
-- pragma synthesis_off
...
-- pragma synthesis_on
```

Tool-specific directives

```
--RTL_SYNTHESIS OFF
...
-- RTL_SYNTHESIS ON
```

IEEE1076.6 directives

- Synthesis operation can be controlled via directives
- Can be embedded in code as comments
  - Ignored by simulation
  - Interpreted by synthesis compiler
- For example, synthesis on/off
  - Used to hide non-synthesizable code from synthesis compiler
    - Debug, timing checks, etc.
- Directives are usually tool-specific
- IEEE1076.6 Standard for VHDL RTL Synthesis defines:
  - RTL\_SYNTHESIS OFF
  - RTL\_SYNTHESIS ON
- Refer to tool documentation
- Directives hidden in comments may be overlooked and cause problems in synthesis



Synthesis operation can be controlled via synthesis directives. There are many synthesis directives. It is embedded as comments in the code, ignored by simulation, and interpreted by the synthesis compiler. For example, the synthesis on/off directive hides non-synthesizable code such as debug, timing checks, etc., from the synthesis compiler. Most are tool-specific, although some tool vendors support directives from other companies. Vendor-specific directives may have the tool or company name in the comment.

IEEE 1076.6 standard defines two directives (called “meta-comments”) only. It would help if you used these in preference to the tool-specific directives for turning synthesis on and off.

You will need to refer to your tool documentation to understand which specific directives are supported by your tools.

Other directives can control:

- Implementation of arithmetic operators
- FSM state encoding
- Selection of gates and cells from the technology library

Warning: It may be better design practice to place synthesis control directives other than simple synthesis on/off into separate scripts, where they can be more easily reviewed, edited, and maintained. Directives hidden in VHDL comments may be overlooked and can cause problems in synthesis.

## Module Summary

Development of any design in RTL code requires:

- Careful structuring of combinational and registered processes
- A well-defined clocking and reset scheme
- Careful checking of synthesis reports



Let's summarize this module. When writing VHDL for synthesis, you need to think about the hardware you intend to create. The design of RTL requires careful structuring of combinational and registered processes, a well-defined clock and reset scheme and careful checking of synthesis reports.

Keep the things we discussed in mind, but do not become too obsessed about them: often, they are things you can change when synthesis does not give you the expected results.

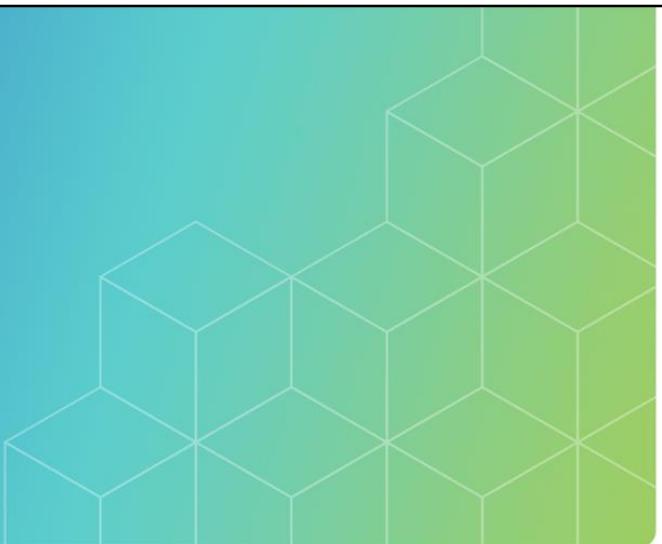
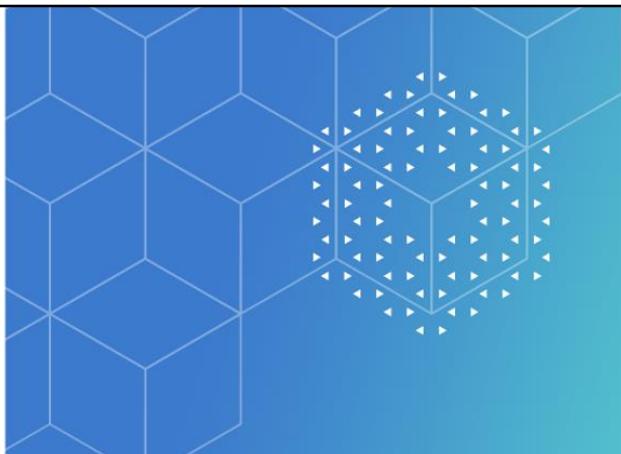
## Lab

### Lab 15-1 Alarm Clock Controller

256 © Cadence Design Systems, Inc. All rights reserved.



You will now have the opportunity to complete a self-paced lab to reinforce the ideas presented in this module.



## Synthesis Coding Styles

**Module** **16**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we discuss the concept of hardware synthesis and various issues surrounding them.

## Module Objectives

In this module, you

- Describe some of the more advanced coding style issues which affect synthesis results
- Define `if` and `case` synthesis
- Describe synthesis of variables
- Implement tri-state drivers
- Type convert higher data types on ports
- Outline how to manage hierarchy



*This page does not contain notes.*

## if Synthesis

```

library IEEE;
use IEEE.std_logic_1164.all;

entity IF_EXAMPLE is
    port (A, B, C, D : in std_logic_vector(3 downto 0);
          OP        : out std_logic_vector(3 downto 0));
end IF_EXAMPLE;

architecture RTL of IF_EXAMPLE is
begin
    process (all)
    begin
        if (D = "0000") then
            OP <= A;
        elsif (D <= "0101") then
            OP <= B;
        else
            OP <= C;
        end if;
    end process;
end RTL;

```



Draw the architecture of the hardware that this represents

Question



259 © Cadence Design Systems, Inc. All rights reserved.

For the given sample code on the slide, spend a few minutes working out architecture in terms of multiplexers and comparators.

It is important to understand what architecture of hardware would be synthesized from this.

Hence it is possible to influence the architecture of the synthesized gate structure from the style of writing the VHDL.

This can influence the critical paths within the gate-level design.

## Synthesis Coding Styles: if Synthesis: Solution

```

library IEEE;
use IEEE.std_logic_1164.all;

entity IF_EXAMPLE is
    port (A, B, C, D : in std_logic_vector(3 downto 0);
          OP        : out std_logic_vector(3 downto 0));
end IF_EXAMPLE;

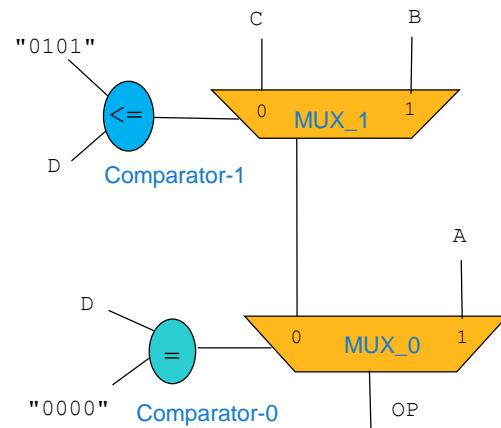
architecture RTL of IF_EXAMPLE is
begin
    process (all)
    begin
        if (D = "0000") then
            OP <= A;
        elsif (D <= "0101") then
            OP <= B;
        else
            OP <= C;
        end if;
    end process;
end RTL;

```



Draw the architecture of the hardware that this represents

Question



The hardware inferred by the given code is shown on the right. In the code, we see that if D equals 0 0 0 0, then OP is assigned with A. To implement this, we need a comparator to make the comparison; also, we know that if logic infers priority logic, hence a mux, with the output of the comparator as a select line. If D equals 0 0 0 0, the output of the comparator goes high, which is given as a select line to the MUX\_0, which routes input on pin-1, i.e., A to the output, OP. If the comparison fails, then input on pin-0, which is driven by the else if logic is routed to output, OP.

In the else, if block, we use a comparator to see if the D value is 0 1 0 1 as in the code. The output of this comparator is given as a select line to MUX\_1. If the comparator output is true, then the value on pin-1, i.e., B, is ported to the output of MUX\_1. Otherwise, the value on pin-0, which is driven by the else logic, C in this case, is routed to the output of MUX\_1.

## Example: Clocked Process

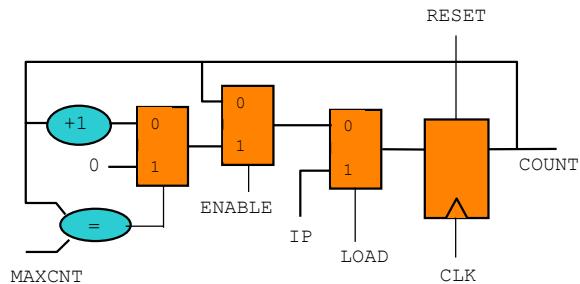
```

...
signal CLK, RESET, ENABLE, LOAD : std_logic;
signal IP, COUNT : unsigned(3 downto 0);
constant MAXCNT : natural := 9;
...
process (CLK, RESET)
begin
    if RESET = '1' then
        COUNT <= "0000";
    elsif rising_edge(CLK) then
        if LOAD = '1' then
            COUNT <= IP;
        elsif ENABLE = '1' then
            if COUNT = MAXCNT then
                COUNT <= "0000";
            else
                COUNT <= COUNT + 1;
            end if;
        end if;
    end if;
end process;
...

```

Up counter with:

- Asynchronous RESET
- Synchronous LOAD
- Count ENABLE



261 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

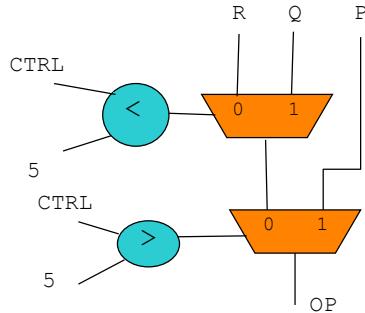
Let us look at how hardware is inferred for a clocked process.

Remember:

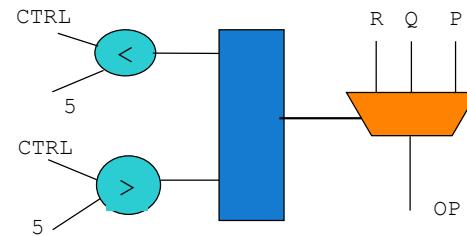
- When structuring clocked processes, asynchronous reset behavior is placed before the clock edge expression, and synchronous clocked behavior is placed after.
- When describing conditional clocked behavior, we use an `if-else-if` construct with the highest priority condition checked first.
- The example on the slide infers on the up counter with asynchronous reset. On reset, the count value is updated with 0; else count is driven by the `if-else-if` construct. We see that the first load is checked; if it is 1, whatever value is on IP is loaded onto the count. If load equals 0, then we see if the counter is enabled or not; if enable equals 0, then count retains its previous value via the feedback path. If enable is high, the value of count is checked; if it is equal to max-count, then count is updated with 0; else, the count is incremented by 1 on every clock edge. On reaching, max count, it is made to roll over to 0 and then increment. We see that reset takes the highest priority overload, which in turn takes priority over enable.

## if Versus case Architecture

```
if (CTRL > 5) then
    OP <= P;
elsif (CTRL < 5) then
    OP <= Q;
else
    OP <= R;
end if;
```



```
case CTRL is
when "000" | "001" | "010" |
    "011" | "100" =>
    OP <= Q;
when "101" =>
    OP <= R;
when others =>
    OP <= P;
end case;
```



262 © Cadence Design Systems, Inc. All rights reserved.



We have looked at the architecture from the nested IF statements. Now let's look at the architecture which results from a case statement.

The architecture created from a case statement is more parallel, consisting of a large selector driven by a large number of comparators. If the case statement tests a range of values ("000", "001", "010", "011", "100"), then this may result in a hardware architecture with two comparators ( $CTRL \geq 0$  and  $CTRL \leq 4$ ), or a single comparator ( $CTRL < 5$ ) if the synthesis tool can understand that  $CTRL$  is only positive.

All synthesis tools will further optimize these logic structures. However, the structure which results from synthesis often depends on the starting position, and so the choice of statements can have an effect on the final result.

The correct choice of statement can also have a significant effect on FPGA design, where one structure may fit more naturally into an FPGA's internal architecture than another one.

## Decoding with Lookup Tables

```

architecture USE_CASE of DECODER is
begin
  process (all)
  begin
    case D_IN is
      when 0 to 4 =>
        D_OUT <= 255;
      when 5 to 9 =>
        D_OUT <= 250;
      when 10 to 14 =>
        D_OUT <= 245;
      when 15 to 19 =>
        D_OUT <= 240;
      when 20 to 24 =>
        D_OUT <= 235;
      when 25 to 29 =>
        D_OUT <= 230;
      when 30 to 34 =>
        D_OUT <= 225;
      when 35 to 39 =>
        D_OUT <= 220;
      ...
    end case;
  end process;
end USE_CASE;

```

```

entity DECODER is
port (D_IN : in integer range 0 to 255;
      D_OUT : out integer range 0 to 255);
end DECODER;

```

```

architecture USE_LUT of DECODER is
type T_LUT_ARRAY is array (0 to 255) of
  integer range 0 to 255;
constant LUT_ARRAY : T_LUT_ARRAY :=
  (255, 255, 255, 255, 255,
   250, 250, 250, 250,
   245, 245, 245, 245, 245,
   240, 240, 240, 240, 240,
   235, 235, 235, 235, 235,
   230, 230, 230, 230, 230,
   225, 225, 225, 225, 225,
   220, 220, 220, 220, 220,
   ....);
begin
  D_OUT <= LUT_ARRAY(D_IN);
end USE_LUT;

```

263 © Cadence Design Systems, Inc. All rights reserved.



The case statement initially maps to a huge MUX with two comparators on each of the 255 select lines. Each comparator has half of its inputs tied high or low. This can cause a synthesis tool to work very hard to break down all of the logic.

A simpler way to describe this for a synthesis tool is to use a lookup table, as shown in the example on the right of this slide. Often this gives better results with the synthesis in less time.

The lookup table declares an array with a separate element for every value of the input DIN. The corresponding value of DOUT is then placed in each element. We can then obtain the right DOUT by using DIN as an address to the lookup table array.

Lookup tables are an excellent method for describing arbitrary coding/decoding operations.

This example has been deliberately simplified by using integer types instead of vectors for DIN and DOUT. The case example would be significantly more complex using vector types as the case ranges (35 to 39) cannot be used with vectors. With vectors, you would have to list each value in the range. In fact, if DIN was a vector, it would probably be easiest to convert DIN to an integer before using the case statement as it stands, rather than trying to list all the possible values in each branch of the case.

This example shows how much you sometimes need to think about not only the hardware you want but also the hardware structures that you are describing with your code.

## Variables in Clocked Process

```
library IEEE;
use IEEE.std_logic_1164.all;
entity FB is
  port (IN1 : in std_logic;
        CLK : in std_logic;
        OUT : out std_logic);
end FB;
```

```
FIRST: process
  variable TMP : std_logic;
begin
  wait until rising_edge(CLK);
  TMP := not IN1;
  OUT <= not TMP;
end process FIRST;
```

?

How many registers  
are inferred in synthesis?  
Question

```
SECOND: process
  variable TMP : std_logic;
begin
  wait until rising_edge(CLK);
  OUT <= not TMP;
  TMP := not IN1;
end process SECOND;
```

264 © Cadence Design Systems, Inc. All rights reserved.



Let's first look at how variables are synthesized in a clocked process.

There are two very similar processes here:

- In process FIRST, the variable is assigned and then read.
- In process SECOND, the variable is read and then assigned.

How many flip-flops are synthesized in each of these designs?

HINT: The golden rule is that a register is inferred on a variable in a clocked process if the variable is ever read before it is assigned.

Process FIRST infers one register. The variable is written first and then read, so it synthesizes down to a simple net.

In process SECOND, however, the variable is read first, then assigned. When the variable is read, we must be reading the variable value from the last time the process was executed. This would be the last rising edge of the clock. Therefore, synthesis must infer a register on the variable to store the value from the last clock edge. The signal assignment to OUT also infers a register, so we find two registers inferred by process SECOND.

## Register Inference and Variables



How many registers are inferred?

Question

```
process (CLK)
  variable VAR_REG : std_logic_vector(7 downto 0);
begin
  if rising_edge(CLK) then
    VAR_REG(0) := D;
    for I in 6 downto 0 loop
      VAR_REG(I+1) := VAR_REG(I);
    end loop;
    Q <= VAR_REG(7);
  end if;
end process;
```

```
if rising_edge(CLK) then
  VAR_REG(0) := D;
  VAR_REG(7) := VAR_REG(6);
  VAR_REG(6) := VAR_REG(5);
  VAR_REG(5) := VAR_REG(4);
  VAR_REG(4) := VAR_REG(3);
  VAR_REG(3) := VAR_REG(2);
  VAR_REG(2) := VAR_REG(1);
  VAR_REG(1) := VAR_REG(0);
  Q <= VAR_REG(7);
end if;
```

- A register inferred on a variable if the variable is ever read before it is assigned

265 © Cadence Design Systems, Inc. All rights reserved.



In the execution of a process, if a variable is read *before* it is assigned, then the value read must be the value that was assigned to the variable in the previous execution of the process. For a registered process, this is the value assigned to the variable on the previous clock edge. Hence the synthesis tool creates a register to store the value of the variable between clock cycles.

In the example above, we get seven registers inferred. Elements 6 to 1 of VAR\_REG are all read before being assigned, and so infer registers. Element 0 is written first, then read, so it doesn't infer a register. Finally, the signal assignment to Q infers a register, as always.

Reading a variable before assigning a value in a combinational process will either result in a latch being inferred (for the same reason as in a registered process) or will be reported by the synthesis tool as a syntax error.

This example is to illustrate the synthesis of variables, not necessarily to illustrate how to model shift registers. A better way to describe a shift register could be to use a slice of an array and concatenation in a single signal assignment.

```
signal VAR_REG : std_logic_vector(6 downto 0);
...
process (CLOCK)
begin
  if rising_edge(CLK) then
    VAR_REG <= VAR_REG(5 downto 0) and D;
  end if;
end process;
```

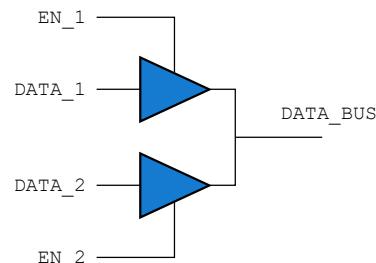
## Tri-state Inference: Assignment of 'Z'

```
library IEEE;
use IEEE.std_logic_1164.all;
entity TRI_STATE_DRIVERS is
  port(DATA_1, DATA_2 : in std_logic;
       EN_1, EN_2      : in std_logic;
       DATA_BUS        : out std_logic);
end TRI_STATE_DRIVERS;
```

```
architecture PROCESSES of TRI_STATE_DRIVERS is
begin
  process (all)
  begin
    if EN_1 = '1' then
      DATA_BUS <= DATA_1;
    else
      DATA_BUS <= 'Z';
    end if;
  end process;

  DATA_BUS <= DATA_2 when EN_2 = '1' else 'Z';

end PROCESSES;
```



- Conditional assignment of signal to 'Z' infers a tri-state controller
- Tri-state signal must be a resolved type

266 © Cadence Design Systems, Inc. All rights reserved.



This slide shows the type of code that represents the behavior of tri-state drivers and the code that most synthesis tools require to infer tri-state drivers in the design.

A tri-state controller is inferred from a specific form of if statement.

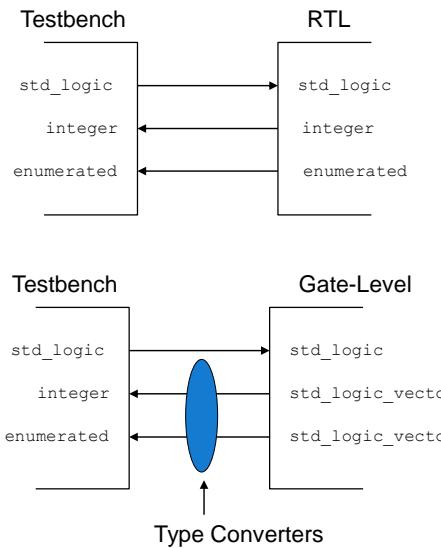
The if statement condition examines the enable signal of the tri-state controller. If the enable is active, then a value is driven onto the tri-state signal. Otherwise, an unconditional else is used to assign the tri-state to the std\_logic high-impedance value 'Z' (others => 'Z' can be used for tri-state buses).

Since, in these examples, we have processes containing only an if statement which has a single target and an unconditional else, we can use concurrent statements – conditional signal assignments – instead of the processes.

Note that concurrent statements are much more succinct in this situation!

## "High-Level" Port Data Types

- "High-level" RTL types synthesize to `std_logic` vector gate-level types
- Ports using "high-level" types require "type converters" to integrate netlist with the original RTL testbench
  - Not all synthesis tools can create these!
- Good guidelines:
  - Only use high-level types on ports of "minor" blocks
  - Only use `std_logic`-based types on ports of major blocks



267 © Cadence Design Systems, Inc. All rights reserved.



If you have integers and enumerated types on ports, the synthesis will need to write out conversion functions to convert these types to and from the individual vector signals in the gate-level design.

Most synthesis tools are not able to write out such conversion functions. Because of this, designers often use only simple `std_logic` and `std_logic_vector` types (including arithmetic package types `signed` and `unsigned`) on the ports of their designs for synthesis.

Check netlist formats and limitations carefully.

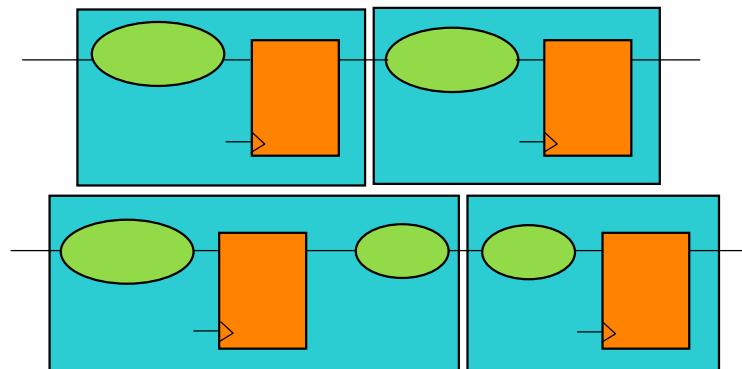
Type conversion is covered in more detail in the Advanced Data Types module.

## Hierarchy: Registering All Outputs

If the output is registered:

- Arrival time of each signal is well-defined
- Hence, easy to set constraints on each block

The registered output provides a complete clock period to implement combinational logic at the second block



268 © Cadence Design Systems, Inc. All rights reserved.



You will find the synthesis process is easier to manage if you try to have registers on the outputs of each of the hierarchical blocks in your design. This is because:

- All combinational logic “clouds” are in the same hierarchical block and can be fully optimized for area and timing.
- It is easier to set the constraints on each block, as the arrival time of signals at each input and output are well defined.

With registered outputs, the synthesis tool has almost a complete clock period to implement the combinational logic at the input of the second block.

With un-registered outputs, the designer must specify what proportion of the clock period is available to implement each combinational block. This is done by specifying input and output delays for each block. These delays must be realistic and reflect the comparative performance of the logic blocks.

## Synthesis Coding Styles: Summary

Lots of issues

- if and case synthesis
- Synthesis of variables
- Register inference
- Tri-states
- Integers and enumerated types

Think hardware!



In this module, we looked at the hardware inferred by if and case statements; we saw how variables are synthesized and issues surrounding them, inference of registers in clocked processes, discussed the implementation of tri-states, and finally, saw how higher types like integers and enumerated types are converted to std\_logic using type converters by the synthesis tool. The main goal of this module was to make the user think in terms of the hardware inferred by the code that one writes.

## Lab

### Lab 16-1 Full Counting Mechanism for the Alarm Clock

270 © Cadence Design Systems, Inc. All rights reserved.



With this knowledge, you will be able to execute the lab Full Counting Mechanism for the Alarm Clock. Please refer to the lab book for details.

## Functions and Procedures

**Module** **17**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

This module is in two parts, one explaining the syntax and use of functions and procedures, two followed by some examples of their use.

## Module Objectives

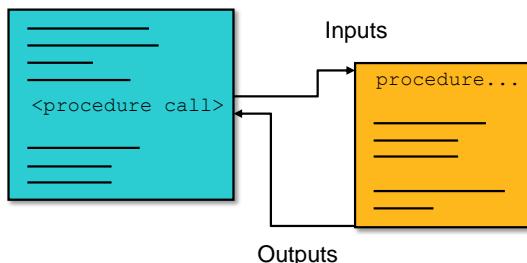
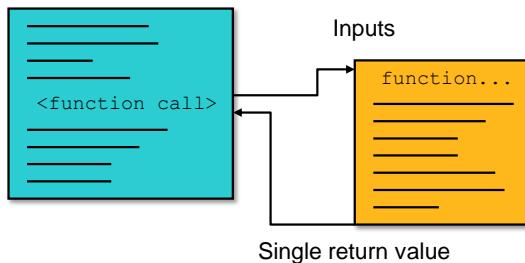
In this module, you

- List and define the types of subprograms
- Describe subprogram concepts
- Define functions and procedures
- Implement subprogram overloading capabilities
- Describe resolution functions



*This page does not contain notes.*

## Functions and Procedures



### Subprograms

- Encapsulate portions of repeated code
- Two types: functions and procedures
- Contain sequential statements
- Execute in sequence like “software”
- Called from within
  - A process as a sequential statement
  - An architecture as a concurrent statement

### Function

- Multiple inputs, the single return value
- Can only be used as part of an expression

### Procedure

- Zero or more inputs/outputs
- Is a concurrent or sequential statement



273 © Cadence Design Systems, Inc. All rights reserved.

Subprograms encapsulate portions of repeated code. Function and procedure are two types of subprograms. They contain sequential statements that execute in sequence, just like software or the code inside your processes.

Functions and procedures can be used, or called, from within processes as sequential statements or within architectures as concurrent statements.

A function is defined with input parameters only. You call it by providing values for these parameters, such as a signal or variable values, which are operated upon by the function. The result is a single return value used as part of an assignment. A function can only be used as part of an expression, usually on the right-hand side of an assignment.

A procedure works similarly but is defined as having input and output parameters; you call it by providing values for all its input parameters and by telling it what internal signals or variables the output computed values should be assigned to.

## Example: Function

```

Function name           Formal parameter name
Formal parameter type
function PARITY_FUNC (FP : std_logic_vector) return std_logic is
  variable TMP : std_logic;           Local variable
begin
  TMP := '0';
  for J in FP'range loop
    TMP := TMP xor FP(J);
  end loop;
  return TMP;           Returned value
end PARITY_FUNC;

```

*Returned type*

*Function exits when return executed*

274 © Cadence Design Systems, Inc. All rights reserved.



An example function is shown on the slide. We use a “formal” parameter FP in the function declaration. It is important to know that this is the "formal" parameter as this term often appears in error messages.

Only input (`in`) parameters are allowed for functions. Functions only return a single value whose type is defined in the function declaration.

The value is returned, and the function terminates when the `return` statement is executed.

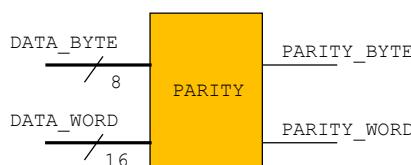
Note: Input parameter FP does not have a range specified here.

- Specification of the range is optional – we will see why omitting it is useful.
- This does mean we have to use attributes ('range) in the function declaration because we don't know the size of FP.

Note: Function can declare local variables, like a process.

- BUT variable values are not remembered between calls to the function.
  - In fact, they do not exist when the function is not executing.
  - Can be difficult to trace/monitor during simulation.

## Function Call



```

function PARITY_FUNC (FP : std_logic_vector)
    return std_logic is
begin
    variaable TMP : std_logic;
    begin
        TMP := '0';

        for J in FP'range loop
            TMP := TMP xor FP(J);
        end loop;
        return TMP;
    end PARITY_FUNC;

entity PARITY is
    port( DATA_BYTE      : in std_logic_vector(7 downto 0);
          DATA_WORD      : in std_logic_vector(15 downto 0);
          PARITY_BYTE    : out std_logic;
          PARITY_WORD    : out std_logic);
end PARITY;

architecture FUNC of PARITY is
    -- function declaration
begin
    PARITY_BYTE <= PARITY_FUNC(DATA_BYTE);
    PARITY_WORD <= PARITY_FUNC(DATA_WORD);
end FUNC;

```

Actual parameter      Function used in expression

275 © Cadence Design Systems, Inc. All rights reserved.



So, let's now look at how this function would be called.

A function call is always used as part of an expression, i.e., usually on the right-hand side of an assignment statement to a signal or variable.

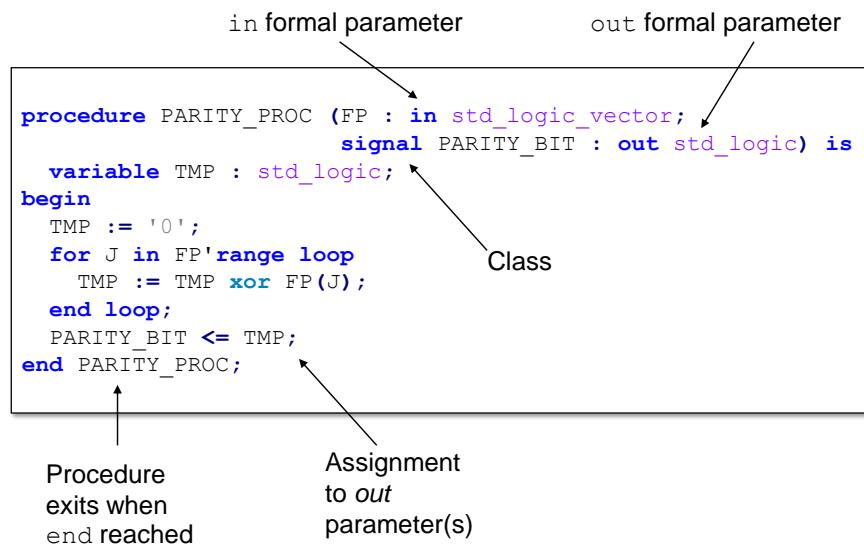
Function calls can be used in either concurrent or sequential statements. Here, we are calling functions within architecture as a concurrent statement.

The actual parameter used in the function call is mapped to the formal parameter of the definition, and the returned value is, in this example, assigned directly to a signal, parity-byte, and parity-word. Note that the types must match.

Since we left FP unconstrained, we can use the same function with actual parameters of different lengths, bytes, and words in this case. The size of FP depends on the size of the actual parameter mapped to it. This is why we used the FP tick range in the function declaration to give us the range of the actual parameter for each specific function call.

In the example on the slide, function PARITY\_FUNCTION calculates and returns the parity of the input data by performing xor over all the bits of the input data.

## Example: Procedure



276 © Cadence Design Systems, Inc. All rights reserved.



Here is the same example using a procedure instead of a function to calculate parity.

Procedures can have any number of output as well as input parameters.

- Use procedures when you want to return multiple values.

In the formal parameter list, we need to identify input and output parameters by specifying a mode (`in`, `out`, or `inout`) for each parameter.

For the `PARITY_BIT` output parameter, we have also defined the class of the parameter as a signal – we will see why shortly.

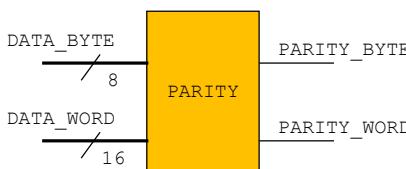
To return values, assignments are made to the output formal parameters.

- Signal assignment in this case because we have specified the class of `PARITY_BIT` as a signal.

A procedure terminates when the `end PARITY_PROC` line is reached.

As in the function example, we have an unconstrained input parameter and local variables.

## Procedure Call



```
procedure PARITY_PROC (FP : in std_logic_vector;
                      signal PARITY_BIT : out std_logic) is
begin
  variable TMP : std_logic;
  TMP := '0';
  for J in FP'range loop
    TMP := TMP xor FP(J);
  end loop;
  PARITY_BIT <= TMP;
end PARITY_PROC;
```

```
entity PARITY is
  port( DATA_BYTE   : in std_logic_vector(7 downto 0);
        DATA_WORD   : in std_logic_vector(15 downto 0);
        PARITY_BYTE : out std_logic;
        PARITY_WORD : out std_logic);
end PARITY;

architecture PROCEDURES of PARITY is
  -- procedure declaration
begin
  PARITY_PROC (DATA_BYTE, PARITY_BYTE);
  PARITY_PROC (DATA_WORD, PARITY_WORD);
end PROCEDURES;
```

-- Parameter named association  
PARITY\_PROC(FP => DATA\_WORD,  
 PARITY\_BIT => PARITY\_WORD);

Actual parameters

Procedure used  
as concurrent statement

277 © Cadence Design Systems, Inc. All rights reserved.



Let us see how a procedure is called. A procedure call is always a statement, either concurrent or sequential.

- These examples are concurrent.

In these examples, actual parameters used in the procedure calls are mapped to the formal parameters of the declaration on a positional basis, first actual to first formal, etc.

- Types must match.

Since we left FP unconstrained, we can use the same procedure with actual parameters of different lengths.

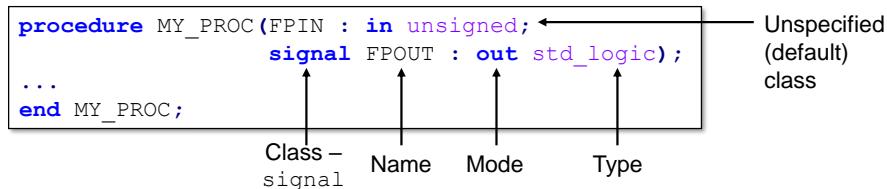
Note: The formal output parameter PARITY\_BIT is mapped to signals (actually entity ports) in both procedure calls.

Since we want to use this procedure to calculate values for signals, the class of the PARITY\_BIT parameter needs to be signal.

Named association could also be used to map actual and formal parameters, just as in a component port map:

```
PARITY_PROC (FP => DATA_WORD, PARITY_BIT => PARITY_WORD);
```

## Subprogram Parameter Properties



Subprogram formal parameters have four properties:

- Name
- Mode (direction)
  - in (read), out (write), or inout (read/write)
- Type
- Class
  - Defines how formal parameter is used within the subprogram
  - Defines what actual parameter can be associated with formal

Three classes for parameters

- Signal, constant, variable

278 © Cadence Design Systems, Inc. All rights reserved.



Formal subprogram parameters have four properties:

Name – used within the subprogram definition to access a specific parameter.

Mode (direction)

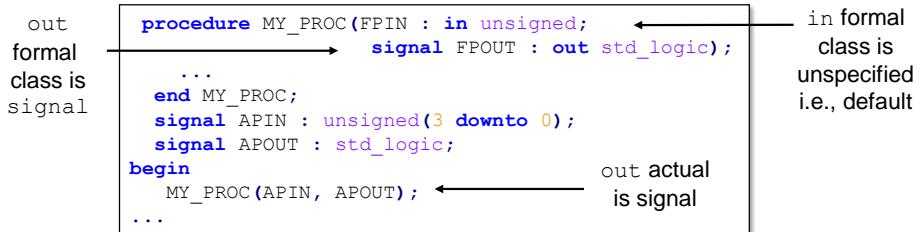
- in mode parameters are read-only within the subprogram
- out mode parameters are write-only within the subprogram
- inout mode parameters are read/write within the subprogram

Type – the formal parameter type must match that of the actual parameter.

Class

- All parameters have a class – signal, constant or variable.
- The formal parameter class determines what actual parameter can be associated with the formal parameter. It also determines what information is passed into the subprogram with the actual parameter – e.g., some signal waveform information is only passed into the subprogram for a formal parameter of class signal – to use an attribute like 'event on a parameter, requires the parameter to be of class signal, or the subprogram cannot determine the attribute value.

## Parameter Class and Mode



Formal parameters can be either a `signal`, `variable` or `constant` class

`in` parameter

- Formal is class `constant` by default
  - Actual can be `constant`, `signal` or `variable`, or an expression
  - Most flexible class and most commonly used

`out/inout` parameter

- Formal is a class `variable` by default
- Must match the class of formal and actual
  - Class `signal` for a signal actual parameter
  - Class `variable` for a variable actual parameter

279 © Cadence Design Systems, Inc. All rights reserved.



Formal parameters can be either `signal`, `variable` or `constant` class.

For input, the mode `in` parameters, the default class is `constant` – this is the most flexible and commonly used.

- The actual parameter associated with a constant formal parameter can be a signal, variable or constant object or expression (e.g., `A` and `B`).
- `signal` class inputs are only needed when using certain signal attributes on parameters (e.g., `'event`).
- `variable` class inputs are only needed for access variables (beyond the scope of this course) and (in VHDL'87) file objects.

For output or inout (mode `out/inout`) parameters, the default class is `variable`.

- For a procedure, the class of the formal `out` parameter must match the class of the corresponding actual when it is called.
- In this example, `FPOUT` must be declared as a class `signal` in the parameter list because the actual parameter `APOUT` is a `signal` object. If we were writing to a `variable`, the formal parameter could be left as default.

Guidelines: Usually, `in` parameters are left as default mode `constant` (most flexible) and `out` parameters declared as `signal` (we typically use subprograms to write to signals).

## Defining Subprograms

```
-- sub-program within an architecture
architecture FUNCTIONS of PARITY is

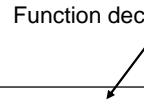
    function PARITY_FUNC (FP : std_logic_vector)
        return std_logic is
            variable TMP : std_ulogic;
    begin
        TMP := '0';
        for J in FP'range loop
            TMP := TMP xor FP(J);
        end loop;
        return TMP;
    end PARITY_FUNC;
begin
    ...
end FUNCTIONS;
```

Process (not shown)

Architecture

Package

- Declaration in package
- Definition in the package body



```
package P_FUNCS is

    function PARITY_FUNC (FP : std_logic_vector)
        return std_logic;

end P_FUNCS;

package body P_FUNCS is

    function PARITY_FUNC (FP : std_logic_vector)
        return std_logic is
        variable TMP : std_ulogic;
    begin
        TMP := '0';
        for J in FP'range loop
            TMP := TMP xor FP(J);
        end loop;
        return TMP;
    end PARITY_FUNC;
end P_FUNCS;
```

280 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Subprograms can be defined in three places:

Process (not shown here)

A subprogram is only available in that specific process.

Architecture

A subprogram is only available in that specific architecture.

Package

Subprograms can be made visible to any design unit via a use clause.

Subprogram definition – containing name, parameter list, and return type (for a function) – must be placed in the package.

Complete subprogram declaration (including the first line) must be placed in the package body.

## Subprogram Overloading

```
package SINEPACK is
    function SINE (L : integer)          return real;
    function SINE (L : real)            return real;
    function SINE (L : std_logic_vector) return real;

end SINEPACK;
package body SINEPACK is
    ...
end SINEPACK;
```

```
use work.SINEPACK.all;
entity OVERLOADED is
    port ( A_BUS : in std_logic_vector (3 downto 0);
           B_INT  : in integer;
           C_REAL : in real;
           A,B,C  :out real);
end OVERLOADED;
architecture DEMO of OVERLOADED is
begin

    A <= SINE (A_BUS);
    B <= SINE (B_INT);
    C <= SINE (C_REAL);

end DEMO;
```

Subprogram overloading:

- Multiple declarations with the same name but different types
- Call mapped to implementation via type signature

Any subprogram can be overloaded

281 © Cadence Design Systems, Inc. All rights reserved.



We have already looked at the concept of overloading for arithmetical operators in the Arithmetic Operators module. Overloading can be applied to any type of subprogram. Subprogram declared multiple times with the same name but with different types is called subprogram overloading. Each call is mapped to a specific implementation via type signature. In this example, we have defined a function called SINE three times, with each definition having a different data type for the input. Any subprogram can be overloaded.

The more general term for overloading is “subprogram overloading.”

## Type Qualification

```

library IEEE;
use IEEE.std_logic_arith.all;
entity QUALIFY is
  port (A, B : in UNSIGNED(3 downto 0);
        EQL : out boolean);
end QUALIFY;

architecture RTL of QUALIFY is
begin
  process (all)
  begin
    EQL <= (A <= B);
    -- This is legal
    -- L:UNSIGNED R:UNSIGNED

    EQL <= (A <= "1001");
    -- Compilation error: ambiguity
    -- R:UNSIGNED? R:SIGNED?

    EQL <= (A <= UNSIGNED'("1001"));
    -- This is legal

  end process;
end RTL

```



```

package numeric_std is

type UNSIGNED is array (NATURAL range <>) of std_logic;
type SIGNED is array (NATURAL range <>) of std_logic;
...
function "<=" (L: UNSIGNED; R: UNSIGNED) return BOOLEAN;

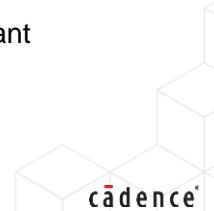
function "<=" (L: UNSIGNED; R: SIGNED) return BOOLEAN;
...

```

Qualify type of an object to remove ambiguity

- Particularly literals

Alternatively, define literal as constant



282 © Cadence Design Systems, Inc. All rights reserved.

Multiple overloading can lead to situations where it is unclear which overloaded definition should be used.

EQL <= (A <= B);

A and B are unsigned types, so the compiler knows the first overloaded function must be used.

EQL <= (A <= "1001");

The compiler does not know the type of the literal “1001” – it could be a 2's complement signed or a binary unsigned. Either the first or the second function could be used. Since the compiler doesn't know which to use, it will report an error.

EQL <= (A <= UNSIGNED'("1001"));

We can use a type name, followed by ' and the literal value in brackets, to identify the type of a literal – this is called type qualification.

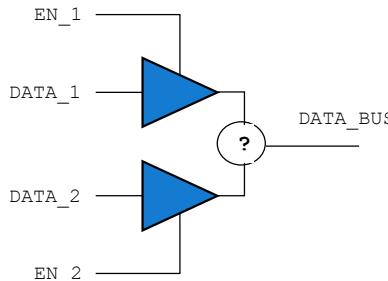
Here, by identifying “1001” as type unsigned, the compiler will now use the first function.

## Resolved Types (Review)

```

signal DATA_1, DATA_2 : std_logic;
signal EN_1, EN_2      : std_logic;
signal DATA_BUS        : std_logic;
...
DATA_BUS <= DATA_1 when EN_1 = '1'
else 'Z';
DATA_BUS <= DATA_2 when EN_2 = '1'
else 'Z';

```



All predefined types are unresolved

- Multiple assignments to the same target give compile errors

Can define *resolved* types

- Multiple assignments allowed signals of resolved types
- Resolution functions* work out value on resolved signal
- Essential for modeling real hardware, e.g., tri-state buses

Only resolved types you are likely to use:

- std\_logic, std\_logic\_vector, signed, unsigned



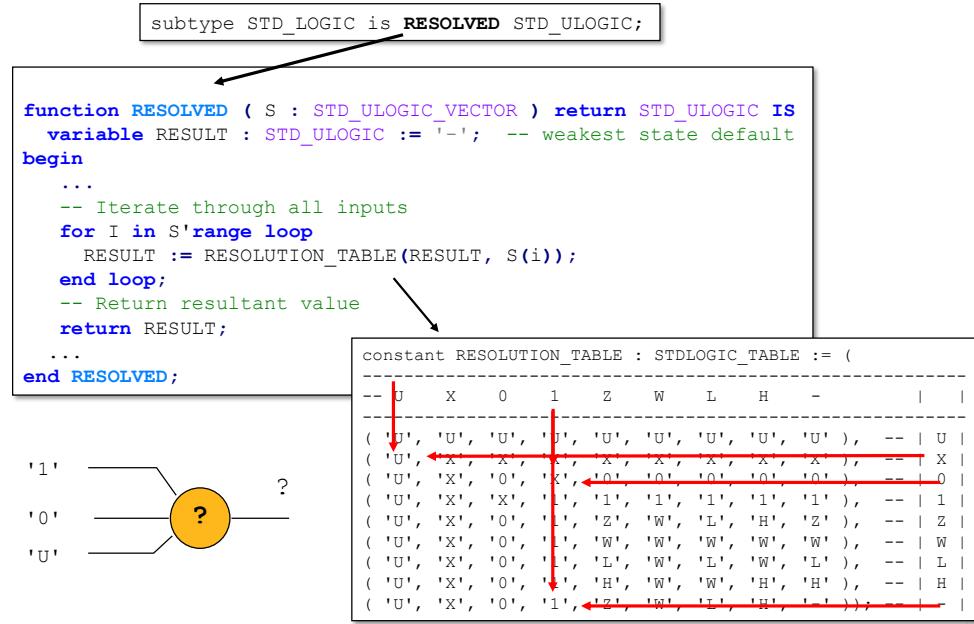
When you connect the outputs of two gates together in hardware, you get problems; the same is true in VHDL. Two signal assignments to the same signal would normally give a compiler error in VHDL because it is not possible to work out what the value on the signal should be.

VHDL does allow you to drive the same signal with two different signal assignment statements in one special case; when the signal is of a resolved type. The basic idea here is that when you make two or more signal assignments to a signal of a resolved type, a special function called a resolution function is called up to sort out what the value on the resolved signal should be.

Resolved types are essential for modeling tri-state buffers. Std\_logic, std\_logic\_vector, signed and unsigned are the only resolved types we are likely to use.

In the example on the slide, if both EN\_1 and EN\_2 are high at the same time, DATA\_BUS is assigned with both DATA\_1 AND DATA\_2, which value would be picked by the DATA\_BUS is decided by the resolution function, as DATA\_BUS is of resolved type.

## std\_logic Definition



284 © Cadence Design Systems, Inc. All rights reserved.

**cadence®**

We will illustrate how the std\_logic resolution function works. This code has been cut and pasted directly from the std\_logic\_1164 package.

The simulator creates a vector of all the driving values on a signal and calls the resolution function

- The example here would create a vector of length 3 with values '1', '0', and 'U'.
  - Note: The order of values cannot be determined.

The function initializes RESULT to '-' (weakest std\_logic value). It then compares this to the first driving value in the vector using the RESOLUTION\_TABLE lookup table. Comparing '-' and '1' gives a new RESULT of '1'. The function then indexes through all the other driving values in the vector, using the lookup table to derive a final value on the signal.

The calculation is as follows:

$$\begin{array}{lll} \text{drive} & '1' & '0' & 'U' \\ \text{RESULT} & '-' & = '1' & = 'X' & = 'U' \end{array}$$

Understanding the operation of the resolution function is useful for debugging bus contention. For example, a common problem is a signal switching between '1' and 'X'. This is due to a process continually driving '1' onto the signal, usually in a reset condition (perhaps a synchronous process has been copied, but the signal names are not correctly updated). When the signal normally switches to '0', this conflicts with the extra driving value of '1', and the resolution function returns 'X'.

## Summary Quiz: Functions and Procedures

1. Which type of subprogram:
  - a. Can return multiple values?
  - b. Can return only one value?
2. Which type of subprogram call:
  - a. Is an expression?
  - b. Is a statement?
3. What is a:
  - a. Formal parameter?
  - b. Actual parameter?
4. What is the definition of subprogram overloading?
5. When is a signal required to be of a resolved type?

285 © Cadence Design Systems, Inc. All rights reserved.



Let's take a short quiz on this module.

Which type of subprogram:

- Can return multiple values?
- Can return only one value?

Which type of subprogram call:

- Is an expression?
- Is a statement?

What is a formal parameter and an actual parameter?

What is the definition of subprogram overloading?

When is a signal required to be of a resolved type?

## Summary Quiz: Functions and Procedures (Solutions)

1. Which type of subprogram:
  - a. Can return multiple values?
    - Procedure
  - b. Can return only one value?
    - Function
2. Which type of subprogram call:
  - a. Is an expression?
    - Function
  - b. Is a statement?
    - Procedure
3. What is a:
  - a. Formal parameter
    - Parameters used in the declaration of the subprogram
  - b. Actual parameter
    - Parameter used in the subprogram call

286 © Cadence Design Systems, Inc. All rights reserved.



Here are the solutions.

The procedure can return multiple values, whereas the function can return only a single value.

A function subprogram call is an expression, whereas a procedure subprogram call is a statement.  
Parameters used in the declaration of the subprogram are formal parameters.

The parameter used in the subprogram call is called actual parameters.

## Summary Quiz: Functions and Procedures (Solutions) (continued)

4. What is the definition of subprogram overloading?

- The declaration of multiple subprograms with the same name but with different parameter types. The subprogram call can be made with the common name, and the compiler works out which implementation to use based on the types of the actual parameters.

5. When is a signal required to be of a resolved type?

- When there are multiple drivers on the signal.



The declaration of multiple subprograms with the same name but with different parameter types. The subprogram call can be made with the common name, and the compiler works out which implementation to use based on the types of the actual parameters. This is called subprogram overloading. When there are multiple drivers on the signal, the signal has to be of resolved type.

# Advanced Concurrent VHDL

**Module** **18**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we will look at some of the advanced concurrent and structural constructs available in VHDL.

## Module Objectives

In this module, you

- Make the implementations more flexible using generics
- Implement structures using generate statements
- List and describe various port modes
- Describe blocks



*This page does not contain notes.*

## Generics

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

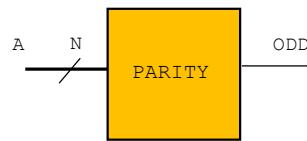
entity PARITY is
  generic (N : integer);
  port (A : in unsigned(N-1 downto 0);
        ODD : out std_logic);
end PARITY;

architecture RTL of PARITY is
begin
  process (all)
    variable TMP : std_logic;
  begin
    TMP := '0';
    for I in 0 to N-1 loop
      TMP := TMP xor A(I);
    end loop;
    ODD <= TMP;
  end process;
end RTL;

```

Generic declaration      Generic use      Generic use

290 © Cadence Design Systems, Inc. All rights reserved.



Allows instance-specific information for components

- Components can be made scalable



Generics of types other than an integer may not be supported

Synthesis



Generics allow us to pass different data into different instantiations of the same component.

- Often used for port width or timing data.

With integer generics, you can make parameterized, synthesizable modules.

- In this example, by specifying a generic N and using it in the declaration of the size of input port A, we now have a scalable parity component that can be instantiated with different bus widths.

Generics can be of any type and for any purpose.

- For example, address an instance of a network controller.

Synthesis tools may not support types other than an integer.

- Generic integer data can be converted inside the model to other types if required.

## Passing Values Through Generics

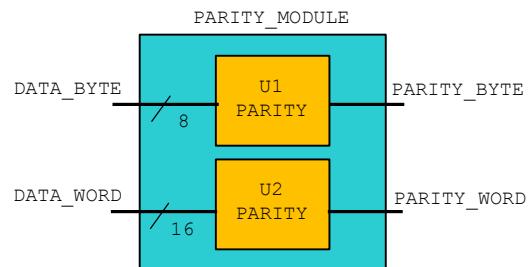
```

...
entity PARITY_MODULE is
  port ( DATA_BYTE : in unsigned(7 downto 0);
         DATA_WORD : in unsigned(15 downto 0);
         PARITY_BYTE :out std_logic;
         PARITY_WORD :out std_logic);
end PARITY_MODULE;

architecture GENERICS of PARITY_MODULE is
  component PARITY
    generic (N : integer);
    port (A : in unsigned(N-1 downto 0);
          ODD : out std_logic);
  end component;
begin
  Generic map           No semi-colon
  U1 : PARITY
    generic map (N => DATA_WORD'length)
    port map (A => DATA_WORD,
              ODD => PARITY_WORD);
  U2 : PARITY
    generic map (N => DATA_BYTE'length)
    port map (A => DATA_BYTE,
              ODD => PARITY_BYTE);

end GENERICS;

```



```

library IEEE;
use IEEE.std_Logic_1164.all;
use IEEE.numeric_std.all;

entity PARITY is
  generic (N : integer);
  port ( A : in unsigned (N-1 downto 0);
         ODD : out std_logic);
end PARITY;

architecture RTL of PARITY is
  ...

```

291 © Cadence Design Systems, Inc. All rights reserved.



Generic maps are used to map values to each instantiation, as port maps are used to connect ports and signals for each instantiation.

- Note there is no “;” between the generic map and port map.
- The code in the slide, PARITY\_MODULE, declares and instantiates two parity components, U1 and U2.

The generic value N of the component is mapped to widths 16 and 8 using generic maps. Here, U1 is a parity block of input width 16, and U2 is a parity block of width 8.

- Both instantiations are linked to a single model.

Generics can also be passed through configurations (see the Application of Configurations module). You don't need to re-compile the entity when you change the generic.

## Generic Values

```
library IEEE;
use IEEE.Std_Loic_1164.all;
entity DELAY_GENERIC is
  generic ( DELAY : time := 1.0 ns);
  port ( A, B : in std_logic;
         OP : out std_logic);
end DELAY_GENERIC;
architecture CELL of DELAY_GENERIC is
begin
  OP <= A and B after DELAY;
end CELL;
```

Generics can pass delayed information to specific instances

All generics must have a value, either from:

- Generic map
  - Instantiation or configuration
- Default value
  - Entity or component declaration

```
...
component DELAY_GENERIC
  generic ( DELAY : time := 1.0 ns);
  port ( A, B : in std_logic;
         OP : out std_logic);
end component;
...
U1: DELAY_GENERIC generic map (1.2 ns) port map (N1, N2, N10);
U2: DELAY_GENERIC generic map (0.9 ns) port map (N3, N4, N11);
U3: DELAY_GENERIC port map (N5, N6, N12);
...
```

292 © Cadence Design Systems, Inc. All rights reserved.



Similarly, we can use generics to pass instance-specific timing data into a component instantiation.

This is a simple example: real cell models may have complex pin-to-pin delays, etc.

All generics must have a value – it is an error to leave a generic unknown. Generics are assigned values in generic maps in component instantiations or configurations (see the Application of Configurations module).

Generics can also be given default values in an entity (when the generic is declared) or a component declaration. If the generic is given a default value in the entity, it must also be given a default value (not necessarily the same value) in the component declaration.

In the example on the slide, we see that the delay\_generic model has a generic delay initialized to 1 nanosecond in the port declaration. Generic delay is also assigned a default value in the component declaration. Hence, the value to delay generic may or may not be passed in the component instantiation. Here, U1 and U2 have their generic delay values overridden with different values, while U3 picks up the default delay value from the component declaration as there is no new value passed during instantiation.

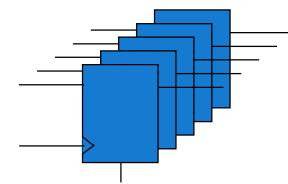
## generate Statement

```

library IEEE;
use IEEE.std_logic_1164.all;
entity REG_BANK is
    generic (DEPTH : integer);
    port (DIN : in std_logic_vector( DEPTH-1 downto 0 );
          CLK, RESET : in std_logic;
          DOUT : out std_logic_vector(DEPTH-1 downto 0));
end REG_BANK;

architecture STRUCTURAL of REG_BANK is
    component REG
        port(D,CLK,RESET : in std_logic;
             Q           : out std_logic);
    end component;
begin
    GEN_REG:
    for I in 0 to DEPTH-1 generate
        REGX : REG port map (DIN(I), CLK, RESET, DOUT(I));
    end generate GEN_REG;
end STRUCTURAL;

```



293 © Cadence Design Systems, Inc. All rights reserved.

The generate statement allows a variable number of concurrent statements to be created. In this example, an array of registers are created by placing a component instantiation inside the generate statement. Here the number of registers, and the width of the bus, are given by the generic DEPTH.

Syntax point to note:

- The generate label must be present.
- Use of the label at the end of generate is optional.

Note: A generate statement can contain any concurrent statement and can even contain processes!

## if...generate

`if...generate` can be used within a `for..generate` to allow irregularity

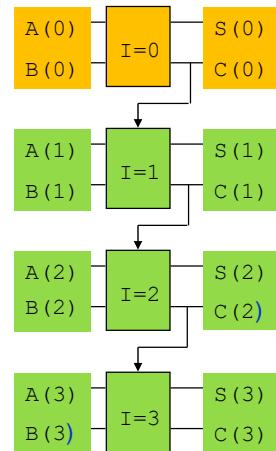
Also standalone for conditional structures

Makes generate more flexible

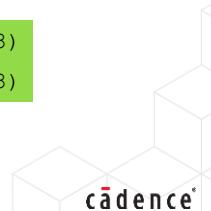
- For example, ripple-carry adder with no carry-in

```
GEN_ADD: for I in 0 to 3 generate
    LOWER_BIT: if I = 0 generate
        U0: HALFADD port map (A(I),B(I),S(I),C(I));
    end generate LOWER_BIT;

    UPPER_BITS: if I > 0 generate
        UX: FULLADD port map (A(I), B(I), C(I-1), S(I), C(I));
    end generate UPPER_BITS;
end generate GEN_ADD;
```



294 © Cadence Design Systems, Inc. All rights reserved.



An example of `if...generate` shows how irregular structures can be described.

In this example, for the case when  $I = 0$  (LSBs of A, B, and S), we instantiate a half-adder. For all other values of  $I$ , we instantiate a full adder and map the carry between the instantiations.

Note: The `if...generate` statement can be used standalone to create a structure based on the value of a constant (for example).

- The value must be known at compilation time!

There is no `else` allowed with an `if...generate`. For mutually exclusive conditions (as in the example above), you must create two `if...generate` statements with different conditions.

## Port Modes

vhdl2006

```

entity COUNTER is
port (CLK, RST : in std_logic;
      OP: out unsigned(3 downto 0));
end COUNTER;

architecture RTL of COUNTER is
begin
  process (CLK, RST)
  begin
    if RST = '1' then
      OP <= (others => '0');
    elsif RISING_EDGE(CLK);
       if (OP = 9) then
        OP <= (others => '0');
       else
        OP <= OP + 1;
      end if;
    end if;
  end process;
end RTL;

```

VHDL  
06  
Update

out port  
can be read

“Mode” means the direction

Five mode ports in VHDL:

<b>in</b>	Port may be read but may not be assigned
<b>out</b>	Port may be assigned Originally could not be read back (This limitation was removed in VHDL2006)
<b>inout</b>	Bi-directional port
<b>buffer</b>	Like output port but may be read back
<b>linkage</b>	Obsolete

295 © Cadence Design Systems, Inc. All rights reserved.



There are five port modes (direction) in VHDL.

Ports of mode `in` can be read inside the entity/architecture but can not be assigned.

Ports of mode `out` can be assigned inside the entity/architecture but historically could not be read. This limitation was removed in the VHDL2006 release.

Ports of mode `inout` are true bi-directional ports and should only be used for such.

Ports of mode `buffer` are like output ports but can be read from within the entity/architecture.

Historically this port mode provided a means of reading an `out` port simply by changing the mode from `out` to `buffer`. However, buffer ports have additional issues which prevent their widespread adoption, as we will see.

The final port mode is `linkage`, but this mode is obsolete.

Buffer mode ports seem to allow us to get around the problem of reading output ports, but there is an issue.

## Reading an Out Mode Port

```

entity COUNTER is
port (CLK, RST : in std_logic;
      OP : out unsigned(3 downto 0));
end COUNTER;

architecture RTL of COUNTER is
signal ICOUNT : unsigned(3 downto 0);
begin
process (CLK, RST)
begin
  if RST = '1' then
    ICOUNT <= (others => '0');
  elsif RISING_EDGE(CLK);
    if (ICOUNT = 9) then
      ICOUNT <= (others => '0');
    else
      ICOUNT <= ICOUNT + 1;
    end if;
  end if;
end process;  Read/write local signal
OP <= ICOUNT; Assign to output
end RTL;

```

296 © Cadence Design Systems, Inc. All rights reserved.

Before VHDL2006, out mode ports could not be read

One solution was to use buffer ports

However, port modes propagate

- A buffer port can only be directly connected to another buffer port
- Leads to problems when the component is instantiated
- Limitation removed in VHDL2001, but buffer ports are not common

The traditional solution was to use an intermediate signal

- Read/write local signal internally
- Assign to out port



There is a potential problem with buffer ports.

When a port of mode buffer is connected directly to a port in the next level of the hierarchy up, then this port to which it is connected also needs to be of mode buffer.

This issue can also propagate down the hierarchy, as well as up.

This usability issue with buffer ports meant they were not widely used, and a different solution, using intermediate signals, was commonly adopted to work around the out port issue. A local (intermediate) signal (e.g., ICOUNT) is declared in the entity/architecture. This signal can be read and written without restriction inside the component and then is assigned to the port OP in a concurrent statement. This allows OP to be defined as an out port and avoids the port mode propagation issue of buffers.

In VHDL2001, the port mode propagation issue has been fixed, and buffer port modes no longer propagate, meaning that buffer ports can be connected directly to ports of mode out.

However, in VHDL2006, output ports can be read from within architecture, so using buffer ports is again virtually redundant.

## Blocks

```

architecture BEH of BLOCK_DEMO is
begin

  -- concurrent statements

  LABEL_1 : block
    -- declarations
  begin
    -- concurrent statements
  end block LABEL_1;

  -- concurrent statements

  LABEL_2 : block
    -- declarations
  begin
    -- concurrent statements
  end block LABEL_2;

  -- concurrent statements

end BEH;

```

- Subdivision of architecture
- It can have local declarations
- Allows grouping of concurrent statements for readability
- No effect on simulation or synthesis
- Synthesis tools have varying support of blocks

297 © Cadence Design Systems, Inc. All rights reserved.



The concept of a block is that it is a subdivision of an architecture that contains concurrent statements and can have its own local declarations.

- It is purely a readability feature and does not affect the simulation.

There are various forms of the block, but none of them allow you to do any more than can be done with the process.

Synthesis tools all accept processes but have varying support of blocks.

- Blocks, where supported, do not affect the synthesized netlist.

There is a type of block called a guarded block. Its usage is now obsolete, but it does use VHDL keywords BUS and REGISTER, which prohibit you from using these words as object names.

## Summary Quiz: Advanced Concurrent VHDL

1. What is the difference between a `generic` and a `constant`?
2. How is the value of a `generic` assigned for a specific instance?
3. What is the difference between the `for..generate` statement and the `for` loop?
4. What is the difference between a port of mode `out` and a port of mode `buffer`?
  - a. Before VHDL2006?
  - b. After VHDL2006?



Let us take a short quiz on this module. What is the difference between a generic and a constant? How is the value of a generic assigned for a specific instance?

What is the difference between the `for..generate` statement and the `for` loop? What is the difference between a port of mode `out` and a port of mode `buffer`? Before VHDL2006? After VHDL2006?

## Summary Quiz: Advanced Concurrent VHDL: Solutions

1. What is the difference between a `generic` and a `constant`?
  - A generic value can be changed on each component instantiation, whereas a constant value can only be changed by editing the code.
2. How is the value of a `generic` assigned for a specific instance?
  - A generic value is assigned in a generic map.
3. What is the difference between the `for..generate` statement and the `for` loop?
  - A `for..generate` is a concurrent statement (used outside a process) and can only contain concurrent statements. A `for` loop is a sequential statement (used only inside a process) and can only contain sequential statements.
4. What is the difference between a port of mode `out` and a port of mode `buffer`?
  - a. Before VHDL2006?
    - Are VHDL2006, an `out` port could not be read inside the architecture, but a `buffer` port could.
  - b. After VHDL2006?
    - After VHDL2006, an `out` port could be read inside the architecture, making a `buffer` port redundant.

299 © Cadence Design Systems, Inc. All rights reserved.



### Question 1

A generic value can be changed on each component instantiation, whereas a constant value can only be changed by editing the code.

### Question 2

A generic value is assigned in a generic map.

### Question 3

A `for..generate` is a concurrent statement (used outside a process) and can only contain concurrent statements. A `for` loop is a sequential statement (used only inside a process) and can only contain sequential statements.

### Question 4

- a. Before VHDL2006, an `out` port could not be read inside the architecture, but a `buffer` port could.
- b. After VHDL2006, an `out` port could be read inside the architecture, making a `buffer` port redundant.

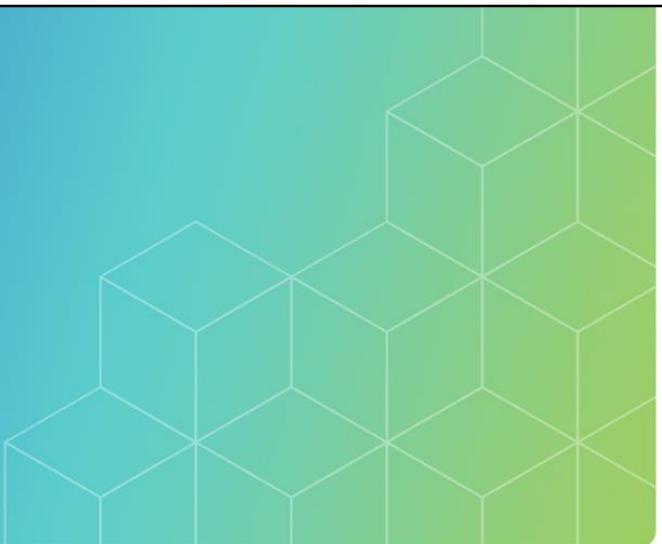
## Lab

### Lab 18-1 Time and Alarm Adjustment

300 © Cadence Design Systems, Inc. All rights reserved.



With this knowledge, you will be able to execute the Time and Alarm Adjustment lab. Please refer to the lab book for details.



## Advanced Data Types

**Module** **19**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we look at more sophisticated data structures and the issues in converting between different types.

## Module Objectives

In this module, you

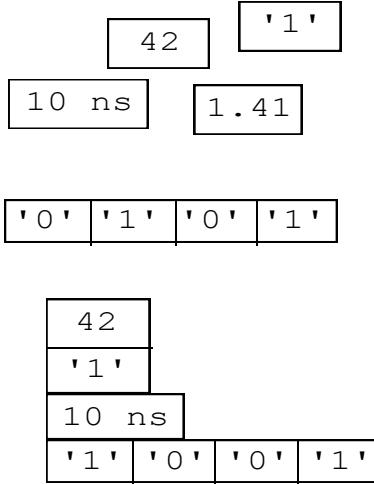
- List and define type classifications
- Define subtypes
- Declare and access arrays and array elements
- Define alias and state their use
- Use various built-in type conversion functions
- Implement your own user-defined type conversion function



*This page does not contain notes.*

## Type Classifications

Data types fall into four distinct categories:



Scalar	<ul style="list-style-type: none"> <li>Object holds one value at a time</li> <li>Integer, real, bit, enumerated, physical</li> </ul>
Composite	<ul style="list-style-type: none"> <li>Object holds more than one value</li> <li>Array: All values of the same type</li> <li>Record: Values of different types</li> </ul>
File	<ul style="list-style-type: none"> <li>Used for reading and writing information to and from external files</li> </ul>
Access	<ul style="list-style-type: none"> <li>Allows dynamic structure to be created</li> </ul>



While we have discussed many different data types, they fall into four distinct categories:

- Scalar – contains a single discrete value. It can be an integer, real, bit, enumerated, or physical type.
- Composite – a collection of values, all of the same type for an array or of different types for a record.
- File – used for reading and writing information to and from external files.
- Access – like a pointer in ‘C,’ allowing dynamic data structures to be created.

We have not yet examined File or Access types; file types are discussed in the testbench modules. Access types like C pointer types are covered in advanced VHDL training.

## Subtype

```

subtype SLV8 is std_logic_vector(7 downto 0);
signal SUB_VEC : SLV8;

signal STD_VEC : std_logic_vector(7 downto 0);
...
SUB_VEC <= STD_VEC;  ✓

subtype T_INT4 is integer range 0 to 15;
subtype T_INT3 is integer range 0 to 7;
...
variable INT4 : T_INT4;
variable INT3 : T_INT3;
...
INT4 := 7;  ✓
INT3 := INT4;  ✓
INT4 := INT4 + 1; -- 8  ✓
INT3 := INT4; -- runtime error!  ✗

-- pre-defined declarations from std package
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;

```

- Names a subset of an existing type
- Still the same type
- Can freely assign between base type and subtypes
- Range checked on assignment
  - Run time error if the value exceeds the range of the target
  - Some predefined subtypes exist in standard packages
- Constraining a type in a declaration is same as creating a sub-type
  - These two examples are the same:

```

-- constrained declaration type...
signal A : unsigned(7 downto 0);
signal INT: integer range 0 to 8;

```

```

-- ... same as subtype
subtype USV8 is unsigned(7 downto 0);
subtype INT_T is integer range 0 to 8;
signal A : USV8;
signal INT : INT_T;

```

304 © Cadence Design Systems, Inc. All rights reserved.



The concept of a subtype is that it is a subset of an existing type.

By declaring SLV8 as a subtype of `std_logic_vector`, it is still a `std_logic_vector` type, so we can freely assign `STD_VEC` to `SUB_VEC`.

VHDL allows you to declare subtypes of an integer with different ranges and to assign these subtypes to each other freely. The simulator will perform range checking for every subtype assignment to check that the value assigned is in the correct type range. In the example above, when `INT4` is assigned to `INT3`, the simulator checks that the current value of `INT4` is an allowable value for `INT3`. When `INT4` is incremented beyond the range of `INT3`, the simulator produces a run-time range constraint error. There are some predefined subtypes declared in standard packages.

Constraining a type in an object declaration is the same as creating a subtype. For example, signal A and INT of constrained unsigned and integer types can be declared as sub-types with the same constraints.

For example:

```

signal A : unsigned(7 downto 0);
signal INT: integer range 0 to 8;

```

is the same as:

```

subtype USV8 is unsigned(7 downto 0);
subtype INT_T is integer range 0 to 8;
signal A : USV8;
signal INT : INT_T;

```

## Subtype Uses

```

subtype T_INT3 is integer range 0 to 7;
subtype T_INT4 is integer range 0 to 15;
...
signal A, B : T_INT3 ;
signal OP : T_INT4 ;
...
OP <= A + B;

```

```

signal DATA : unsigned(31 downto 0);

subtype HIGHBYTE is integer range 31 downto 24;
subtype LOWBYTE is integer range 7 downto 0;
...
DATA(LOWBYTE) <= "10101010"; ← Slices

DATA <= (HIGHBYTE => '0',
          LOWBYTE => '1',
          others => 'Z'); ← Aggregate choices

for I in LOWBYTE loop ← Loop ranges
    DATA(I) <= DATA(LOWBYTE'high - I); ← Array indexing
end loop;

```

Subtypes are used to define constrained vector and integer types that are used many times

- It is a single declaration with a meaningful name
- Used with object declarations
- Aids:
  - Readability
  - Error checking
  - Synthesis efficiency

To define ranges or subranges for arrays

- Meaningful name
- Easy access to fields in data



305 © Cadence Design Systems, Inc. All rights reserved.

Where the same constrained type is used many times, a subtype can declare the constrained type once with a suitable name, which can be used for all the object declarations, aiding readability. Similarly, for a large number of constrained types, subtypes can make code more readable.

Subtypes can also be used to define ranges or subranges of array types. Integer subtypes can be declared with appropriate ranges, and these subtypes are used to select parts of an array in a slice; for loop range or aggregate choice. The attributes 'high, 'low, 'left, 'right, and 'length can also be applied to the subtypes to extract information for indexing individual elements of an array. The for loop example above reverses the bits in the lowest byte of the data array only.

## Array Declarations and Indices

```
-- constrained integer declaration
subtype INT_DEC is integer range 0 to 9;

-- user-defined array type declaration
type T_4DIGIT is array (3 downto 0) of INT_DEC;

-- signal declaration with array type name
signal DIGITS : T_4DIGIT;
```

```
subtype INT_DEC is integer range 0 to 9;

-- enumerated type
type T_DIGITS is (MS_HOUR, LS_HOUR, MS_MIN, LS_MIN);

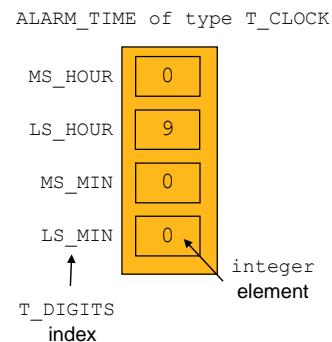
-- array declaration using enumerated indices
type T_CLOCK is array (T_DIGITS) of INT_DEC;

signal ALARM_TIME : T_CLOCK := (1,2,0,0);
...
ALARM_TIME(MS_HOUR) <= 0;
ALARM_TIME(MS_HOUR to LS_HOUR) <= (0,9);
```

Declare array type first, then an object of that type

Array index can be of any discrete type

- More readable data structures



306 © Cadence Design Systems, Inc. All rights reserved.



User-defined array types must always be declared with a `type` declaration statement, which names the array type. Then objects are created using the array type name.

While all the examples we have looked at so far use an index of type `integer`, the index can be of any discrete data type.

This technique can help build more meaningful data structures by clarifying the function of each element in the array.

In the example, the enumerated type `T_DIGITS` is used to index the array `T_CLOCK`. This gives the array four elements with the indices `MS_HOUR`, `LS_HOUR`, `MS_MIN`, and `LS_MIN`. The content of each element is of type `INT_DEC`, i.e., an integer in the range 0 to 9. Any object of the array type can be indexed using the element name. Array slices can also be used.

Lookup tables can easily be built using this technique.

## Unconstrained Array Declarations

```

subtype INT_DEC is integer range 0 to 9;

-- unconstrained declaration
type T_UARR is array (NATURAL range <>) of INT_DEC;

-- unconstrained objects not allowed
signal BADOBJECT : T_UARR;  X

-- different sized objects of same type
signal TWODIGIT : T_UARR(1 downto 0);
signal FOURDIGIT : T_UARR(3 downto 0);  ✓

...
FOURDIGIT <= TWODIGIT & TWODIGIT;

```

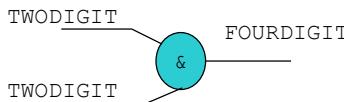
Array declarations can also be unconstrained

Unconstrained arrays must be sized in an object declaration

Predefined types are usually unconstrained

Allows different-sized objects of the same type

- Easier manipulation of data structures



```

-- std_logic_1164 declaration
type std_logic_vector IS array ( NATURAL range <> ) of std_logic;

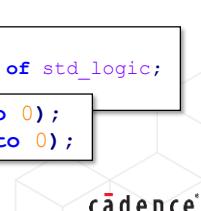
```

```

signal BYTE : std_logic_vector(7 downto 0);
signal WORD : std_logic_vector(15 downto 0);

```

307 © Cadence Design Systems, Inc. All rights reserved.



User-defined array types must always be declared with a type declaration statement.

When an array data type is declared, it can be in one of two forms:

- Constrained array, where the size of the array is defined in the type definition.
- Unconstrained array, where the specification of the size of the array can be deferred until we declare an object of that type.

We must declare the index type and maximum size of the unconstrained array type. (NATURAL range <>) tells us the array is unconstrained but must have an index of type NATURAL and a maximum size of integer'high.

- Natural is defined in the standard package as integers in the range 0 to integer'high.

Unconstrained type definitions allow us to create different-sized arrays of the same type. Being of the same kind, data manipulation is easier; e.g., in the example above, we could easily assign TWODIGIT to locations in FOURDIGIT using the appropriate array slicing:

```
FOURDIGIT(3 downto 2) <= TWODIGIT;
```

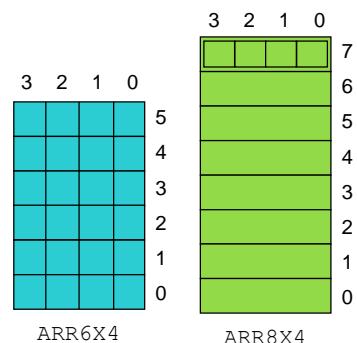
If FOURDIGIT and TWODIGIT had been declared using different constrained types, the above assignment would not be so easy.

## Multi-Dimensional Arrays

```

...
type T_A_OF_A is array (7 downto 0) of std_logic_vector(3 downto 0);
type T_2D_A is array (5 downto 0, 3 downto 0) of std_logic;
signal ARR8X4 : T_A_OF_A;
signal ARR6X4 : T_2D_A;
begin
  ARR8X4(7)(3) <= '1'; -- array of array
  ARR6X4(5,3) <= '1'; -- 2 D array
...

```



Two options with different access forms:

- Array of arrays
  - Separate indices (outer) (inner)
- Two (or more) dimensional array
  - One set of indices (row, column)

An array of arrays supported  
2D arrays are rarely  
Synthesis



308 © Cadence Design Systems, Inc. All rights reserved.

Multi-dimensional arrays can be declared in one of two ways:

- An array of an array.
- Two (or more) dimensional arrays.

An array of arrays is an array where every element is, in itself, an array. In this example, the array T\_A\_OF\_A is an array of elements, where each element is an array of std\_logic\_vector(3 down to 0).

- Synthesis tools can interpret this as a collection of busses.
- To index an individual element in the example above, ARR8X4 (7) selects element 7 of the outer length of eight arrays, and then the second address (3) accesses bit 3 of the contents of element 7.

A two (or more) dimension array is a matrix-type structure.

- This structure has no direct representation in hardware.
- Simple matrix “row, column...” addressing is used to access an individual element.

An array of arrays is synthesized as a collection of buses. For some FPGA synthesis tools, an array of arrays can be used to infer RAM directly.

Multi-dimensional arrays are rarely synthesizable.

## Multi-Dimensional Array Aggregates

```

...
type T_A_OF_A is array (7 downto 0) of std_logic_vector(3 downto 0);
type T_2D_A is array (5 downto 0, 3 downto 0) of std_logic;
signal ARR8X4 : T_A_OF_A;
signal ARR6X4 : T_2D_A;
...
signal NIB0,NIB1,NIB2,NIB3,
      NIB4,NIB5,NIB6,NIB7 : std_logic_vector(3 downto 0);

begin
...
ARR8X4 <= (NIB0,NIB1,NIB2,NIB3,NIB4,NIB5,NIB6,NIB7);
ARR6x4 <= (NIB0,NIB1,NIB2,NIB3,NIB4,NIB5);

ARR8X4 <= (others => NIB7);
ARR6x4 <= (others => NIB5);

ARR8X4 <= (others => (others => '0'));
ARR6x4 <= (others => (others => '1'));
...

```

309 © Cadence Design Systems, Inc. All rights reserved.

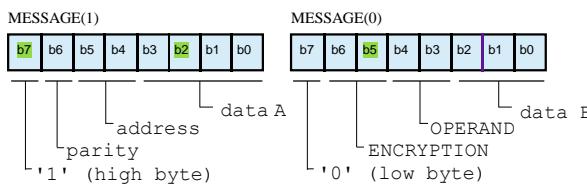


Aggregates can be used to make assignments to all elements of multi-dimensional arrays.

- Can be named or positional and can have the other choice.
- The code example shows that every 4-bit element of the array 8X4 and array 6X4 is assigned with a 4-bit std\_logic\_vector using aggregates. It can also be assigned values using others, assigning all elements with the value specified via others.

The method of assigning all elements of either structure is the same and consists of an aggregate within an aggregate.

## Alias



- An alternative name for an existing object
- Often used for array slices or complex data structures

```
type PACKET is array (1 downto 0) of std_logic_vector(7 downto 0);
signal MESSAGE: PACKET;
signal DATA : std_logic_vector(3 downto 0);
alias ENCRYPTION is MESSAGE(0)(6 downto 5);
alias OPERAND   is MESSAGE(0)(4 downto 3);
alias A         is MESSAGE(1)(3 downto 0);
...
DATA <= A;
```



Supported by some synthesis tools

Synthesis

310 © Cadence Design Systems, Inc. All rights reserved.



Alias does not define or create a new object – it just provides an alternative name for an existing object. This can help with the readability of code for complex data structures.

From VHDL'93, aliases can generally be used for types, functions, etc., instead of just objects.

Before VHDL'93, the type of an alias had to be specified. From VHDL'93, this was made optional. In the code example, elements of complex user-defined data type packets are given alternative names like the message, and encryption aids readability.

For example, a lower nibble of the first element of the message is given the name ‘A’ and so on.

## Type Conversion Functions

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...

signal INT256 : integer range 0 to 255;
signal USN_VEC : unsigned(7 downto 0);
...
INT256 <= USN_VEC; 
USN_VEC <= INT256; 
...
```

- VHDL is strongly typed
- Types must match in assignments
  - Different types are not compatible, even if they have the same values
- Some type conversion functions are provided in packages

```
-- conversion functions from package numeric_std
function to_integer (ARG: UNSIGNED) return NATURAL;
function to_unsigned (ARG, SIZE: NATURAL) return UNSIGNED;
```

INT256 <= to_integer(USN_VEC);	<input checked="" type="checkbox"/>
USN_VEC <= to_unsigned(INT256, USN_VEC'length);	<input checked="" type="checkbox"/>



Remember – the types on either side of an assignment must match exactly – none of the assignments shown here are valid because the types do not match.

We cannot assign `USN_VEC` to `INT256` since the source object is an array and the target is an integer. Likewise, we cannot assign `INT256` to `USN_VEC` since the source object is an integer and the target is an array.

The VHDL packages `std_logic_1164` and `numeric_std` do contain some predefined type conversion functions which we can use.

For the assignment of `USN_VEC` to `INT256`, we can use the `to_integer` conversion function from the `numeric_std` package. Note that the return type of the conversion function is a natural number – remember, natural is a subset or subtype of integer, covering positive integers and zero.

For the assignment of `INT256` to `USN_VEC`, we can use the `to_unsigned` type conversion function from `numeric_std`. This function has two input parameters – the integer value to be converted and an integer specifying the length of the resulting vector – this must match the length of the target of the assignment. Using the '`length`' attribute is ideal for this.

`numeric_std` package is listed in the *VHDL Reference Guide*.

## Closely Related Types

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
signal INT256 : integer range 0 to 255;
signal STD_VEC : std_logic_vector(7 downto 0);
signal USN_VEC : unsigned(7 downto 0);
...
USN_VEC <= STD_VEC; 
USN_VEC <= unsigned(STD_VEC); 
INT256 <= STD_VEC; 
INT256 <= to_integer(unsigned(STD_VEC)); 

```

Allows simple type conversion

$T \leq <T\ type>(S);$

Array sizes must match

Closely related types

- Integers and reals
- std\_logic\_vector, signed and unsigned
- Arrays of same:
  - Length
  - Index type
  - Element type

```
-- declaration from package numeric_std
type UNSIGNED is array (natural range <>) of std_logic;
```

```
-- declaration from package std_logic_1164
type std_logic_vector is array (natural range <>) of std_logic;
```

312 © Cadence Design Systems, Inc. All rights reserved.



Although STD\_VEC and USN\_VEC are of the same size with elements of the same type, they have all been declared with separate type declarations, and so the language regards them as different types.

However, because the arrays are of the same length (8), with elements of the same type (std\_logic), and use integers to index the elements, VHDL defines them as “closely related types.”

For conversion between closely related types only, we can type convert by simply using the name of the target type:

```

target <= <target_type>(source);
USN_VEC <= unsigned(STD_VEC);

```

However, we can't type convert STD\_VEC to INT\_256 as they are not closely related.

Closely related type conversion allows us to convert between the following predefined VHDL types:

- std\_logic\_vector, unsigned, signed
- Integer and real types are also defined as closely related.

Note this feature also allows us to convert STD\_VEC into an unsigned type to use the to\_integer type conversion from numeric\_std.

## Type Conversion in Vector Arithmetic

- `std_logic_vector`, `signed` and `unsigned` types are closely related
- Convert `std_logic_vector` to use overloaded functions

```
-- numeric_std
-- Id: A.3
function "+" (L, R: UNSIGNED) return UNSIGNED;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
-- signal declarations
signal AVEC, BVEC, CVEC : std_logic_vector(7 downto 0);
...
AVEC <= std_logic_vector(unsigned(BVEC) + unsigned(CVEC));
```

Convert return type to target

Match operand types to function

313 © Cadence Design Systems, Inc. All rights reserved.



Note that by the rules of “closely related types,” the `signed` and `unsigned` types used by arithmetic packages are closely related to `std_logic_vector`.

Therefore to use the package's arithmetic functions with `std_logic_vector` types, we can use implicit conversions to match our object and function types.

In the example, `unsigned (BVEC)` and `unsigned (CVEC)` convert the `std_logic_vector` signals `BVEC` and `CVEC` to type `unsigned`. This allows us to use the overloaded function `A.3` from the `numeric_std` package to add `BVEC` to `CVEC`. The function returns an `unsigned` type, so we need another conversion to convert to `std_logic_vector` to match the assignment target `AVEC`.

The `numeric_std` package is listed in the *VHDL Reference Guide*.

## User-Defined Type Conversion Functions

```

library IEEE;
use IEEE.std_logic_1164.all;
...
signal ASIG : std_logic;
signal CHAR : character

function TO_CHAR (VEC : std_logic) return character is
    variable VCHAR : character;
begin
    case VEC is
        when 'U'      => VCHAR := 'U';
        when 'X'      => VCHAR := 'X';
        when '0' | 'L' => VCHAR := '0';
        when '1' | 'H' => VCHAR := '1';
        when 'Z' | 'W' => VCHAR := 'Z';
        when '-'      => VCHAR := '-';
    end case;
    return VCHAR;
end TO_CHAR;
...
CHAR <= ASIG;           
CHAR <= TO_CHAR(ASIG); 
...

```

You may have to define your own type conversion functions

- When types are not closely related
- When no predefined conversions

Writing flexible conversions can be tricky

Define common conversions in packages for sharing amongst projects and design teams



314 © Cadence Design Systems, Inc. All rights reserved.

If types are not closely related and no predefined type conversions exist, you may need to find your own type conversions. Since VHDL does not predefine many useful type conversions (e.g., `std_logic` to `character`), packages containing these functions may be found in internet code repositories and freely or commercially available code. Your company may already have such packages available.

If packages of such functions are not available, you can write your own. In this example, function `TO_CHAR` takes a `std_logic` bit and converts it into a `character` using a `case` statement to convert each `std_logic` value into its equivalent character.

## Summary Quiz: Advanced Data Types

1. What is the difference between a scalar and a composite type?
2. How do you reference a single element of:
  - a. A 2-dimensional array?
  - b. An array of an array?
3. Which of these assignments are legal VHDL?

```
subtype SLV4 is std_logic_vector(3 downto 0);
signal A : SLV4;
signal B : bit_vector(3 downto 0);
signal C : std_logic_vector(3 downto 0);
...
A <= B;
B <= C;
A <= C;
```

315 © Cadence Design Systems, Inc. All rights reserved.



Let's take a short quiz on this module.

What is the difference between a scalar and a composite type?

How do you reference a single element of a 2-dimensional array? An array of an array?

Which of these assignments are legal VHDL?

## Summary Quiz: Advanced Data Types (Solution)

1. What is the difference between a scalar and a composite type?
  - A scalar type has a single value, whereas a composite type, such as an array or record, has a collection of values.
2. How do you reference a single element of:
  - a. 2-dimensional array?
    - Two indices in the same brackets, for example (row, column)
  - b. An array of an array?
    - Two indices in separate brackets, for example (outer) (inner)

316 © Cadence Design Systems, Inc. All rights reserved.



Here is the solution.

### Question 1

A scalar type has a single value, whereas a composite type, such as an array or record, has a collection of values.

### Question 2

- a. Two indices in the same brackets, for example (row, column)
- b. Two indices in separate brackets, for example (outer) (inner)

## Summary Quiz: Advanced Data Types (Solution) (continued)

3. Which of these assignments are legal VHDL?

```

subtype SLV4 is std_logic_vector(3 downto 0);
signal A : SLV4;
signal B : bit_vector(3 downto 0);
signal C : std_logic_vector(3 downto 0);
...
A <= B; --incorrect
B <= C; --incorrect
A <= C; --correct

```

- A <= B; -- incorrect – source is a bit vector, and the target is a subtype of std\_logic\_vector. Even though the bit values are a subset of std\_logic, they are still two different types, and the assignment is illegal.
- B <= C; -- incorrect – source is a std\_logic\_vector, target is a bit\_vector
- A <= C; -- correct – source is a std\_logic\_vector, target is a subtype of std\_logic\_vector



### Question 3

- A <= B; -- incorrect – source is a bit vector, and the target is a subtype of std\_logic\_vector. Even though the bit values are a subset of std\_logic, they are still two different types, and the assignment is illegal.
- B <= C; -- incorrect – source is a std\_logic\_vector, target is a bit\_vector.
- A <= C; -- correct – source is a std\_logic\_vector, target is a subtype of std\_logic\_vector.

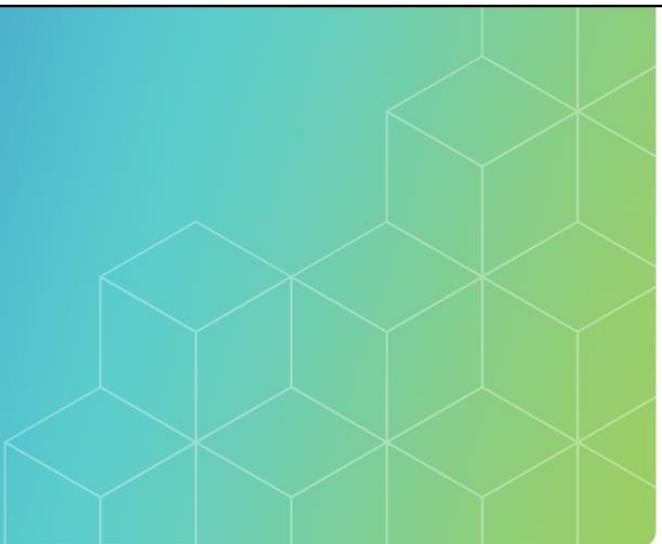
## Lab

### Lab 19-1 Putting It All Together

318 © Cadence Design Systems, Inc. All rights reserved.



With this knowledge, you will be able to execute the following lab. Please refer lab book for more details.



## Testbench Coding Styles

**Module** **20**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we look at the various testbench coding styles and their pros and cons.

## Module Objectives

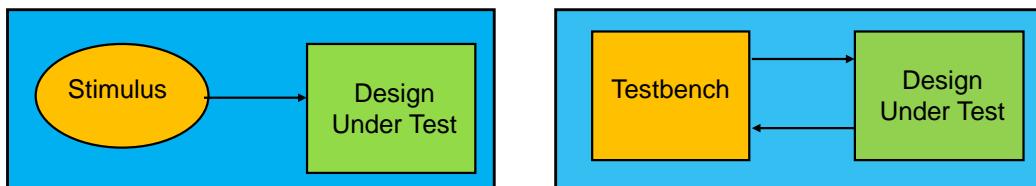
In this module, you

- List and describe different coding styles which are commonly used to write testbenches
- Define testbench configurations
- Define assertions
- List various ways of driving stimulus
- Implement and describe the various TextIO operations



*This page does not contain notes.*

## Testbench Organization



### Simple Testbench

Small number of processes to create a stimulus

Send data to design

No interaction

Manual checking

As in initial lab exercises

### Sophisticated Testbench

Multiple versions of stimulus for different situations

Models environment around design

Talks to design, for example, bus cycles; handshaking

Possibilities of self-checking

As in later lab exercises



In these diagrams, a square is a hierarchical block; an oval is a collection of processes.

The top testbench architecture is as we have used in the initial lab exercises.

- The stimulus could be a separate level of hierarchy.
  - This could be useful if you have multiple versions of the stimulus for different situations.
- It could also have a separate hierarchical block monitoring the results.

Most people on real projects tend to use some form of “intelligent” or “smart” testbench, where the stimulus generated is a function of the response of the model (bottom testbench architecture).

- That is, bus cycles, handshake mechanisms, etc.

The sophistication of your testbench is only limited by your time and imagination.

The following few slides look at some different ways of generating stimulus.

## Assertions (Review)

```
architecture BEH of TESTBENCH is
  signal ADDRESS : unsigned(7 downto 0);
  constant RESERVED : unsigned(7 downto 0)
    := "11110000";
  ...
begin
  -- concurrent assertion
  assert ADDRESS/=RESERVED
    -- message when condition false
    report "Address has RESERVED value"
    severity ERROR;
  ...
end BEH;
```

Assertions are used to validate the behavior of the design

Feedback:

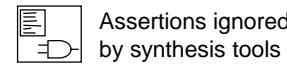
- Error condition
- Test progress reports
- Warnings associated with the functional operation
  - Example, overflows

Reports message when boolean condition is *false*

Assertions can be concurrent or sequential

Severity levels grade messages:

- NOTE
- WARNING
- ERROR (default)
- FAILURE (usually causes the simulation to halt)



Synthesis



No , after assert  
or report

Beware

322 © Cadence Design Systems, Inc. All rights reserved.



Assertions are a very powerful way of continuously monitoring the behavior of a system and reporting back to the designer when things go wrong.

Assertions allow us to report messages to the simulator interface from inside our code.

Useful to feedback information such as:

- Error conditions
- Test progress reports
- Warnings associated with the functional operation, e.g., overflow

The assertion statement contains a conditional clause that will report a message when the boolean condition is false.

The assertion statement can be concurrent or sequential.

- For the concurrent version, the sensitivity list of the equivalent process is given by signals in the condition.

Severity levels can be used to grade messages. The highest severity level usually forces the simulation to halt, although most simulators have an option to control this.

The reported message can be any string type – concatenation can be used to build up a meaningful error message.

In the code on the slide, an assertion checks to see that the address doesn't take any reserved values. If the condition fails, it reports an error.

## Assertion and Report Statements

```

entity TESTREP is
end TESTREP;
architecture TEST of TESTREP is
-- declaration for HIER
begin
  U1 : HIER port map (...);
end TEST;

entity HIER is
  port (...);
end HIER;
architecture RTL of HIER is
begin
  P1: process(all)
    variable ADDR : integer range 0 to 128;
  begin
    ...
    report "Access violation " & ADDR'path_name &
      " -Address: " & integer'image(ADDR)
        severity error;
    end process P1;
end RTL;

```

Error: Access violation :testrep:u1:p1:addr -Address: 128

report can be used as a standalone, sequential statement

- Equivalent to assert FALSE report
- Optional severity clause

The assertion report may only give the name of the design unit

Use attributes for more meaningful reports

'path\_name  
'instance\_name  
'image  
    'image doesn't work with array or record types

323 © Cadence Design Systems, Inc. All rights reserved.



From VHDL'93, the reported clause of an assertion can be used as a standalone sequential statement (used inside a process only).

- Equivalent to assert FALSE report "..."
- E.g.,
  - report "entering RAM test";

From VHDL'93, there are also extra attributes that can make constructing meaningful report messages easier:

- 'path\_name creates a string of the hierarchical path from the testbench to the attribute object
  - :testrep:u1:p1:addr
- 'instance\_name is similar but includes architecture and configuration information
  - :testrep(test):u1@hier(rtl):p1:addr
- 'image converts any scalar value to its equivalent string
  - Warning! The image does not work with an array or record type.

## “In Line” Stimulus (Review from Labs)

```

architecture IN_LINE of TESTBENCH is
  signal A_BUS : std_logic_vector(7 downto 0);
  signal RESET, ACK : std_logic;
  -- declaration of UUT
begin
  -- instance of UUT
  IN_LINE: process
  begin
    A_BUS <= (others => '0');
    RESET <= '0';
    wait for 10 ns;

    RESET <= '1';
    wait for 7 ns;

    A_BUS <= "00000001";
    RESET <= '0';
    wait on ACK;

    A_BUS <= "01100001";
    ...
    wait;
  end process IN_LINE;
end IN_LINE;

```

324 © Cadence Design Systems, Inc. All rights reserved.

Signals are listed only when value changes  
Complex timing relationships easy to describe

- wait for different time values
- wait on handshake signals, etc.
- wait combinations

This type of testbench becomes very large for complex stimuli

- Hence, not efficient



It is very common when first writing a design just to write a simple sequence of stimuli over time using a single process. The stimulus is described using a series of signal assignments separated by wait statements.

This is what you have used in the initial lab exercises.

This is not very efficient to write for a very large amount of stimulus. However, having separate wait statements between each signal assignment allows us to describe complex timing relationships and handshaking protocols.

Note the testbench entity is usually “empty,” i.e., it has no ports.

Testbench entity for the above example is:

```

entity TESTBENCH is
end TESTBENCH;

```

In the code on the slide, we first apply the reset value onto signal A by making the reset high and then drive the first stimulus by making the reset low; the next stimulus is driven only by receiving an ACK. Wait on ACK to implement the handshaking between the DUT and the testbench. We see that any complex timing relationships can be implemented using this coding style, but it becomes very large and inefficient for complex stimuli.

## Stimulus from Loops

```

library IEEE;
use IEEE.std_logic_1164.all;
-- numeric_std has to_unsigned conversion function
use IEEE.numeric_std.all;

architecture USE_LOOP of TESTBENCH is
    signal A_BUS : unsigned(7 downto 0);

    -- declaration of UUT
begin
    -- instance of UUT
    LOOP_STIM: process
    begin
        for I in 0 to 4 loop
            A_BUS <= to_unsigned(I, A_BUS'length);
            wait for 10 ns;
        end loop;
        wait;
    end process LOOP_STIM;
end USE_LOOP;

```

- Easy to implement
- Compact description
- Best for stimulus:
  - Regular values
  - Set time period

325 © Cadence Design Systems, Inc. All rights reserved.



Here we use a for loop to create an incrementing set of values on a bus.

The loop determines the number of stimuli to be applied (5), and the loop variable is also used as the actual stimuli data.

However, the loop variable is an integer type (because loop limits 0 to 4 are integers), and the stimulus signal A\_BUS is of type unsigned. Therefore, we need a type conversion function to convert from integer to unsigned. This style of coding is easy to implement and compact. It is best suited when the stimulus has regular values and a set time period between them.

In this example, we use a function from the IEEE 1076.3 arithmetic package, numeric\_std. The function requires two parameters: the integer value to be converted and the size of the unsigned array to be returned. Here, we use the attribute length to match the size of the returned array to the size of the ABUS.

The wait statement inside the loop gives us a ten ns interval between each stimulus value. Therefore, the loop applies the values 0 to 4; as unsigned 8-bit vectors; at intervals of 10 ns to A\_BUS.

## Stimulus from Array Constant

```

architecture USE_ARRAY of TESTBENCH is

signal A_BUS : std_logic_vector(7 downto 0);

type T_DATA is array (0 to 4) of std_logic_vector(7 downto 0);
constant DATA : T_DATA := ("01101010",
                           "10110101",
                           "01011010",
                           "10101101",
                           "01010110");

-- declaration of UUT
begin
  -- instance of UUT
  ARRAY_STIM: process
  begin

    for I in DATA'range loop
      A_BUS <= DATA(I);
      wait for 10 ns;
    end loop;

    wait;
  end process ARRAY_STIM;
end USE_ARRAY;

```

326 © Cadence Design Systems, Inc. All rights reserved.

Easy to enter

Compact description

Single location to change stimulus values

Best for stimulus with:

- Irregular values
- Set time period



If we wish to apply irregular data values at set time intervals, we can list the data values in a table and drive values onto signals by picking up values from this table via a for a loop.

This can be done by defining an array constant, where each element is an array or record of the data to be applied to the model.

Here T\_DATA is a 5-element array, where each element is an 8-bit std\_logic\_vector. The constant DATA is declared as type T\_DATA, and the stimulus values are defined.

We then use a for loop to apply the data at intervals of 10 ns. The attribute 'range is used to set the loop variable limits – this gives us (0 to 4) for the loop limits.

The loop variable then indexes each element of the DATA constant in turn.

This style of coding is easy to implement. By declaring our stimuli data in one place, we can easily change the data by editing the constant declaration. This style is best suited when the stimulus has irregular values and a set time period.

Note: The time period does not have to be a set value – both data and time delays could be indexed from a constant array. The array element would be a record type containing both data and a time field.

## Use of Subprograms

```

procedure MEM_WRITE (DATA: in T_DATA; ADDR: in T_ADDR) is
begin
    -- Memory write procedure
end MEM_WRITE;

procedure MEM_READ (DATA: out T_DATA; ADDR: in T_ADDR) is
begin
    -- Memory read procedure
end MEM_READ;

```

Testbench is behavioral

Use software style structure

- Functions and procedures

Gets large quickly

```

procedure MEM_INIT() is
begin
    MEM_WRITE("1000", "00");
    MEM_WRITE("0100", "01");
    MEM_WRITE("0010", "10");
    MEM_WRITE("0001", "11");
end MEM_INIT;

```

```

procedure MEM_TEST() is
    variable VALUE_READ: T_DATA;
begin
    MEM_INIT();
    MEM_READ(VALUE_READ, "01");
    if VALUE_READ /= "0100" then
        ...
end MEM_TEST;

```

327 © Cadence Design Systems, Inc. All rights reserved.

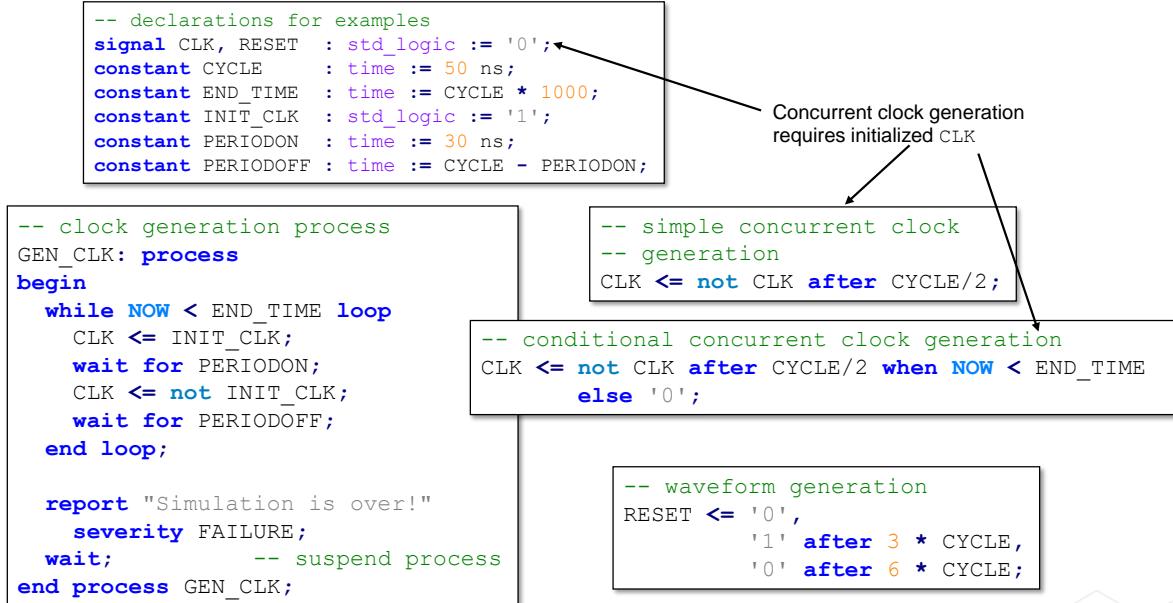


A testbench uses behavioral coding styles and does not need to have a structure based upon entities and architectures. Instead, the structure can be created using functions and procedures, as shown in this example.

Functions and procedures have the advantage that they are easier to declare and reference, and they can be called from both concurrent and sequential parts of your code. This style gets too large for large stimuli.

In the example on the slide, we have procedures MEM\_WRITE to write to a specified memory location with a given data, MEM\_READ to read the value from a given address location, and MEM\_INIT to initialize the memory by writing the initial values into a series of the address location. In the MEM\_TEST procedure, we first initialize the memory locations by calling the procedure MEM\_INIT; next, values from the written memory locations are read by calling the MEM\_READ procedure and are compared against the expected value which is the value written previously into the memory via MEM\_INIT procedure. For large stimuli, this becomes tedious.

## Creating Clocks and Resets



328 © Cadence Design Systems, Inc. All rights reserved.



The clock can be created sequentially using a process or concurrently.

All the declarations for the cycle, period-on, period-off, etc., are made in one place for readability and ease of modification. Period-on and period-off determine the duty cycle of the clock generated. The cycle is the time period of the clock. End-time is used to specify the end of simulation time. The clock generation process uses a `while` loop to generate the clock for a fixed number of cycles with a duty cycle of 60%. CLK is driven high for ON-period and low for period-off, which repeats until simulation time is less than the end time. Remember, NOW returns the current simulation time. The process uses an assertion to stop the simulation after the specified number of clock cycles. An unconditional `wait` is also used to suspend the process indefinitely.

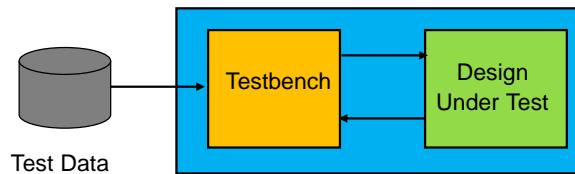
Simple concurrent clock generation uses an accompanying statement which creates a continuous waveform on the CLK signal. When CLK changes, the information is triggered, inverts the current value, and assigns it back to CLK after a delay of 25 ns. The conditional concurrent clock generation also uses a conditional signal assignment to generate the clock for a fixed number of cycles. Else drives 0.

- Note we initialize CLK when we declare the signal; otherwise, inverting an un-initialized value, 'U,' just gives us another un-initialized value.

Multiple `after` clauses can generate a complete waveform on a signal. The waveform generation example assigns '0' to RESET when the statement is executed; schedules RESET to '1' three cycles (150 ns) in the future and then RESET to '0' six cycles in the future.

- The `after` delays must increase as they are evaluated, and the waveform is generated in one operation. It is an error to include diminishing delays.

## Stimulus from a File



Capability to read and write ASCII text files known as TextIO

Utilities are available in a package called TextIO in the library `std`

Data could be created by another program or tool

- For example, image data for image processing system

Advantages

- Stimulus easily changed
- Less analysis/compile time

Disadvantage

- Overhead if a set of tests is very varied

329 © Cadence Design Systems, Inc. All rights reserved.



VHDL has a powerful capability to read and write ASCII text files: known as TextIO.

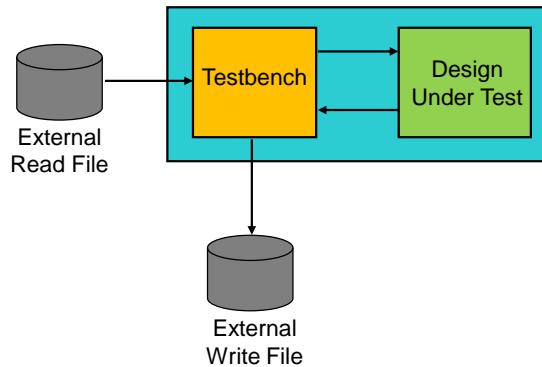
This is a feature of the language definition, and the various utilities required are available in a package called `textio` in the library `std`. This text file could be created by any other program or tool, for example, image data for the image processing system.

Using TextIO, the stimulus can be easily changed by editing the stimulus file without compilation overhead. Fairly sophisticated testbenches can be designed to read instructions or pseudo-code from a file, allowing complex stimuli to be generated by the testbench internally in an efficient high-level manner.

However, there can be a significant overhead in designing such a testbench. If the required set of tests is very varied, then a file-based testbench may not be the most efficient. Also, if the stimulus is directly coded into the testbench, the designer can use the full range of VHDL constructs, but this will not be the case if a pseudo-code is being used.

It may be worth using the TextIO and direct coding approaches for different sets of tests.

## TextIO Capabilities



Standard package to handle file input and output

- TextIO

TextIO can read/write predefined types

- bit, bit\_vector
- boolean
- character, string
- integer, real
- time

Other types must be converted to or from character or strings

Non-standard package `std_logic_textio` able to read/write `std_logic_vector`

Accessed via `use IEEE.std_logic_textio.all`



TextIO allows you to read and write the predefined data types: other types must be converted to or from characters or strings, which can be processed using the existing procedures.

Simulators usually supply a non-standard TextIO package that supports `std_logic_vector` types, e.g., `std_logic_textio`. Accessed via:

```
library IEEE;
use IEEE.std_logic_textio.all;
```

This package also contains `HREAD` and `HWRITE` procedures that allow `std_logic_vector` arrays to be read or written as hex values.

There are not many good references in books, but the Language Reference Manual (LRM) is very readable on this.

The `textio` and `std_logic_textio` packages are listed in the *VHDL Reference Guide*.

## Reading from a File

```

use std.textio.all;
architecture READFILE of TESTBENCH is
  signal A_INT, B_INT : integer;
  -- UUT component declaration
begin
  -- instance of UUT

  STIM_READ: process
    file IN_FILE : text open READ_MODE is "stimulus.vec";
    variable L  : line;
    variable INT : integer;
  begin
    while not endfile(IN_FILE) loop
      readline(IN_FILE, L); ← Read from input
      read(L, INT);           file into line variable
      A_INT <= INT;
      read(L, INT);
      B_INT <= INT;
      wait for 10 ns;
    end loop;
    wait;
  end process STIM_READ;
end READFILE;

```

Reference TextIO Package

Line variable

Read the first integer into INT

Read the second integer into INT

Read file declaration

Read from input file into line variable

stimulus.vec

331 © Cadence Design Systems, Inc. All rights reserved.



Declare use std.textio.all to access text read-write procedures.

The file declaration declares a file object (`IN_FILE`) of type `text` associated with the external file `stimulus`, `vec`, opened in `READ_MODE`. Note the name of the external file is a VHDL string and must be enclosed in double-quotes.

Reading a file consists of reading a line from the file into memory and then extracting the data from this line for use in the model.

The TextIO package defines a data type called `line`, representing a line of text from the file. So we need to declare a variable of this type.

Use a `readline` procedure to read a complete line from the file into the `line` variable.

Then use `read` procedures to read individual data from the `line` into a local variable. The local variable can then be assigned to a signal. Here, we see that the entire line is read into the `line` variable `L` using the `readline` procedure, and then, the first element of the line, i.e., integer '1', is read into `INT` and is assigned to variable `A_INT`. Then, the second element of the line, i.e., integer 45, is read into variable `int` and is set to variable `B_INT`.

- The local variable type determines how the data is read from the line.
- Reading into an integer uses white space to distinguish individual values.

## Line Variables in Read Operations

`readline` copies next line into line variable

`read` copies data item into read variable

- Both delete data after copying
- Amount copied and deleted depends on the type of the variable
- Both advance sequentially through the file and line data

Read file

1	45
23	
32	104
45	54

Code	Line Variable	INT						
<code>readline(IN_FILE, L);</code>	<table border="1"><tr><td>1</td><td></td><td>4</td><td>5</td></tr></table>	1		4	5	-		
1		4	5					
<code>read(L, INT);</code>	<table border="1"><tr><td>4</td><td>5</td></tr></table>	4	5	1				
4	5							
<code>read(L, INT);</code>	<EMPTY>	45						
<code>readline(IN_FILE, L);</code>	<table border="1"><tr><td>2</td><td>3</td></tr></table>	2	3	-				
2	3							
<code>read(L, INT);</code>	<EMPTY>	23						
<code>read(L, INT);</code>	<b>ERROR</b>	23						
<code>readline(IN_FILE, L);</code>	<table border="1"><tr><td>3</td><td>2</td><td></td><td>1</td><td>0</td><td>4</td></tr></table>	3	2		1	0	4	-
3	2		1	0	4			
<code>readline(IN_FILE, L);</code>	<table border="1"><tr><td>4</td><td>5</td><td></td><td>5</td><td>4</td></tr></table>	4	5		5	4	-	
4	5		5	4				

Runtime error if read from empty line

Current line contents lost on new readline

Reading a line from a file and reading a data item from a line are “one-shot” operations. Every time a `readline` is called on a file, the next line of the file is read into the `line` variable.

Each time a `read` is called on a `line` variable, a certain amount of data is copied to the `read` variable, which is deleted from the `line` variable contents. The amount of data copied and deleted depends on the type of the variable used in the `read` procedure.

`readline` and `read` advance sequentially through the file and line data.

Structuring your file access is important. It is an error to call the `read` procedure on an empty `line` variable. If you call a `readline` when the `line` variable is not empty, the data left in the `line` variable is deleted.

From the table, we see that, on the first `readline`, the first line of the file is copied onto `line` variable `L` when `read` is called, and the first integer ‘1’ separated by whitespace is copied onto the variable `int` and is deleted from `line` variable `L`. When the next `read` is called, the second integer ‘45’ is copied onto `INT`. It is deleted from the `line` variable `L`, which is now empty. On the next `readline`, the file’s second line is copied onto the `line` variable, `L`. A `read` following this copies the first integer onto `INT`, deleting the same from `line` variable `L`, which is now empty. Again, if the `read` is called, it results in a runtime error as `L` is empty. The next `readline` is called twice without copying the contents of the line into the local variable after the first call, which results in the loss of line 3 contents.

## Writing to a File

Reference  
TextIO  
Package

Line  
variable L

Write the value  
of A\_INT into L

Write the value  
of B\_INT into L

```
use std.textio.all;
architecture WRITEFILE of TESTBENCH is
    signal A_INT, B_INT : integer;
    -- UUT component declaration

begin
    -- instance of UUT

    DATA_WRITE: process (A_INT, B_INT)
        file OUT_FILE : text open WRITE_MODE is "results.txt";
        variable L : line;
        constant SPACE : character := ' ';
        begin
            write(L, A_INT);
            write(L, SPACE);           -- Write space character
                                         -- to separate integers
            write(L, B_INT);

            writeline(OUT_FILE, L);   -- Write from line
                                         -- variable into
                                         -- output file
        end process DATA_WRITE;

    end WRITEFILE;
```

1	45
23	21
32	1
45	54
104	110

results.txt

Write file  
declaration

333 © Cadence Design Systems, Inc. All rights reserved.



Declare `use std.textio.all` to access file write procedures. The file declaration declares a file object (`OUT_FILE`) of type `text`, associated with the external file `results.txt`, opened in `WRITE_MODE`. Note the name of the external file is a VHDL string and must be enclosed in double-quotes.

Writing to a file consists of writing a data line into memory and then writing the line to the file.

Again, line data types from the TextIO package are used to assemble a line of text for the file. So we need to declare a variable of this type.

Use `write` procedures to write individual data to the line. By default, there is no formatting. Here we have to write space to separate the two values.

Use a `writeline` procedure to write the line to the file. The procedure also appends newline characters to the data and clears the line variable.

Beware – the simulator will delete the written file contents without warning before outputting data.

Note that there is a third file mode `APPEND_MODE` which appends data to the end of the external file so that the existing contents (if any) are not deleted.

Messages can also be written to the simulator transcript window or log file by writing to the predefined file `output`.

## Line Variables in Write Operations

`write` copies object value into line variable

Writes all predefined types

Type convert user-defined types to predefined ones before writing

No data formatting performed

`writeline` copies line variable into file

- Adds CR/LF
- Deletes data from line

Constant SPACE used → to separate values

Writing empty line → gives blank line

Write file

4523  
32 104

Code	INT	Line Variable
<code>write(L, INT);</code>	45	4 5
<code>write(L, INT);</code>	23	4 5 2 3
<code>writeline(OUT_FILE, L);</code>	-	<EMPTY>
<code>write(L, INT);</code>	32	3 2
<code>write(L, SPACE);</code>	32	3 2
<code>write(L, INT);</code>	104	3 2 1 0 4
<code>writeline(OUT_FILE, L);</code>	-	<EMPTY>
<code>writeline(OUT_FILE, L);</code>	-	<EMPTY>

334 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The `write` procedure takes the current object value, converts it to a string, and copies it to the line variable. The object can be a signal, variable, constant, literal, or expression as long as the type of the object is known, and it is a predefined type. Type qualification may be required to make the type of a literal or expression known. Type conversion can convert user-defined types to predefined types for writing. See the Advanced Data Types module.

By default, no data formatting is performed. So, while a read operation into an integer uses white space to delimit the value, a write operation of an integer does not add white space after the value.

The `writeline` procedure copies the current contents of the line variable to the file, adds a carriage return/line feed (depending on the hardware platform) to the end of the line, and deletes the current line variable contents. Before a write, calling the `writeline` procedure again means that the line variable will be empty. Only the carriage return/line feed will be written to the file, resulting in a blank line in the output.

The table shows that 45 is written onto the line variable, followed by 23 without space. When `writeline` is called, the value 4523 is copied onto the write file while deleting the line variable contents, thereby making it empty. Next, value 32 is written onto the line variable, followed by white space. Value 104 is written after this. `Writeline` writes the contents of the line variable, i.e., 32 space 104, onto the write file while deleting line variable contents. The next `writeline` call will write a blank line with only carriage return/line feed onto the write file.

## VHDL'87 File Declaration

```

use std.textio.all;
architecture TEXTIO_87_STYLE of TESTBENCH is
...
FILE_87: process
  variable VECTOR : line;
  file VECTOR_FILE : text is in "stimulus.vec";
begin
  ...
end process FILE_87;
end TEXTIO_87_STYLE;

```

vhdl'87

Two possible modes:  
in  
out

```

use std.textio.all;
architecture TEXTIO_93_STYLE of TESTBENCH is
...
FILE_93: process
  variable VECTOR : line;
  file VECTOR_FILE : text open READ_MODE is "stimulus.vec";
begin
  ...
end process FILE_93;
end TEXTIO_93_STYLE;

```

vhdl'93

Three possible modes:  
READ\_MODE  
WRITE\_MODE  
APPEND\_MODE

335 © Cadence Design Systems, Inc. All rights reserved.



There were significant changes to file declarations between the VHDL'87 and VHDL'93 standards.

- Some simulators automatically convert VHDL'87 file declarations into VHDL'93, so this syntax may still be seen in legacy code.

Only file declarations changed as shown. Besides the declarations, file access is the same for both standards.

VHDL'93 has three modes:

- READ\_MODE – opens the file at the first line and reads each line in sequence.
- WRITE\_MODE – clears file contents and writes lines in sequence to the file.
- APPEND\_MODE – keeps current file contents and writes lines sequentially to the end of the file.

VHDL'87 has two modes:

- in – opens the file at the first line and reads each line in sequence.
- out – clears any current file contents and writes lines in sequence to the file.

## Opening and Closing Files

Files are opened when the file object declaration is elaborated

- Normally, at the start of the simulation

Files are closed when the file variable no longer exists

- Normally, at the end of the simulation

If a file is declared inside a subprogram, it will be opened and closed every time it is called

From VHDL'93, there are also file open and close *sequential* calls:

```
-- declare file variable but do not open file
file VECTOR_FILE : text;
...

-- open file stimulus.vec in write mode
FILE_OPEN(VECTOR_FILE, "stimulus.vec", WRITE_MODE);
...

FILE_CLOSE(VECTOR_FILE); -- close file

-- open file stimulus.vec in read mode
FILE_OPEN(VECTOR_FILE, "stimulus.vec", READ_MODE);
```

336 © Cadence Design Systems, Inc. All rights reserved.



Files are typically opened when the simulator is invoked and closed when terminated.

However, if we declare file objects inside procedures, the file is opened when the procedure is called and closed when it is exited. This gives us some control over file operations.

- E.g., to read lines from a file more than once or to create filenames for write files.

From VHDL'93, we can use procedure calls to open and close files dynamically. This can allow us to open a file in read mode, close the file, open it in append mode to write data, and close and reopen it in read mode to continue reading data.

For multiple file open/close operations on the same file object, `file_open` can use an additional parameter of type `file_open_status` (defined in `TextIO`). You can declare a variable of type `file_open_status` and make the variable the first parameter in a `file_open` call. After the `file_open` has been completed, the variable can be checked for successful completion. If the value is `open_ok`, the file open was successful and can be accessed. If not `open_ok`, there has been an access violation (file object already in use or file cannot be accessed).

## Module Summary

Testbenches soon get sophisticated

- Use behavioral level code

Variety of methods to generate stimulus

- Loop
- Array constants
- In-line
- TextIO
  - Read
  - Write



In this module, we saw that the testbenches are implemented in behavioral modeling. There are various ways to generate a stimulus to the DUT like a loop, array constants, in-line stimulus, and TextIO.

## Testbench Applications

**Module** **21**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

This module discusses various ways of applying stimulus to the DUT and generating simulation output.

## Module Objectives

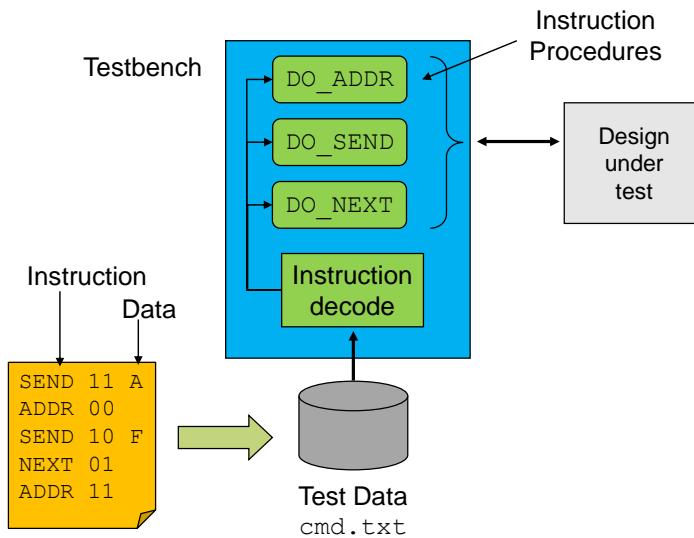
In this module, you

- Implement different ways of applying testbenches to specific design situations
- Use FileIO to apply pseudo-code-based testbenches
- List various ways of generating simulation output
- Implement testbench to produce visualized output



*This page does not contain notes.*

## Script-Driven Testbench



User-defined instructions specify

- Sequence of test operations
- Data associated with operations

Testbench interprets instructions

Performs predefined operations

- Using specified data

Benefits

- Higher level of abstraction
- Human-readable, easier than other methods
- Sequence of tests and data can be edited easily
- Tests and testbench are modular
  - Can easily add additional tests

Disadvantage

- Written manually, hence error-prone



340 © Cadence Design Systems, Inc. All rights reserved.

It is possible to define a set of instructions by which the operation of a testbench can be controlled. This collection of user-defined instructions is sometimes called “Pseudo-code,” and a testbench usually reads these instructions from a text file, interprets the text of each instruction, and performs predefined operations using specified data. This type of testbench is also known as a script-driven testbench.

We will see an example of this in the following slides.

The benefits of a script-driven testbench are many, such as:

- Higher level of abstraction for creating test sequences and data.
- The text file is “human-readable” – easier to understand than in-line, looped, or arrayed stimulus.
- The sequence of tests and data can be easily changed by editing the external file.
- Tests and testbench are modular – additional tests can be easily added.

One issue is that user-defined instructions are more likely to be written by hand and therefore contain errors than a file of stimulus data created by, for example, a graphics package. Therefore, our testbench needs to carefully check the instruction data it is trying to read.

## Example 1: Script-Driven Testbench

```

INST_DECODE: process
    file INFILE           : text open READ_MODE IS "cmd.txt";
    variable L              : line;
    variable DATA_CHAR, SP_CHAR : character;
    variable ADDRESS          : string(1 to 2);
    variable INST             : string(1 to 4);
begin
    while not ENDFILE(INFILE) loop

        readline(INFILE, L);           -- read line
        assert (L'length=9 or L'length=7) -- instruction length check
            report "Instruction Error (line length)"
            severity FAILURE;

        read (L,INST);                -- read instruction as string of four characters

        read (L,SP_CHAR);            -- read separator
        assert (SP_CHAR=' ')         -- instruction separator check
            report "Instruction Error (separator)" & SP_CHAR
            severity FAILURE;

        read (L,ADDRESS);            -- read address
        -- assertion test
...

```

cmd.txt

ADDR 01  
 SEND 10 F  
 NEXT 01  
 ...

341 © Cadence Design Systems, Inc. All rights reserved.



In this example, there are three forms of instruction allowed in the input text file: ADDR XX, NEXT XX, and SEND XX Y, where XX is a hex address, and Y is a hex number.

The testbench needs to read different data types at other times and be very robust – to report errors in the instruction data.

The approach adopted is to read instructions as characters or strings, check and interpret the information and then convert as required to the data types needed for the stimulus.

The testbench structure is based around a while loop, reading each line from the input file and then performing the following operations:

- Check line length (7 characters for ADDR/NEXT commands, 9 for SEND commands)
- Read instruction as a string of four characters
- Read and check the instruction separator
- Read XX address as a string

## Example 2: Script-Driven Testbench

```

...
  case INST is
    when "ADDR" =>                                -- ADDER instruction
      DO_ADDR(ADDRESS);                            -- addr operation

    when "SEND" =>                                -- SEND instruction
      read (L,SP_CHAR);                           -- read separator
      read (L,DATA_CHAR);                         -- read data
      DO_SEND(ADDRESS, DATA_CHAR); -- send operation

    when "NEXT" =>                                -- NEXT instruction
      DO_NEXT(ADDRESS);                           -- next operation

    when others =>                                -- report invalid instruction
      report "Invalid Instruction Skipped : " & INST
      severity ERROR;

  end case;
  -- wait until ready to get next data
end loop;

SIM_FINISHED <= TRUE;                          -- simulation end flag to stop clock
wait;
end process INST_DECODE;

```

cmd.txt

ADDR 01  
SEND 10 F  
NEXT 01  
...



342 © Cadence Design Systems, Inc. All rights reserved.

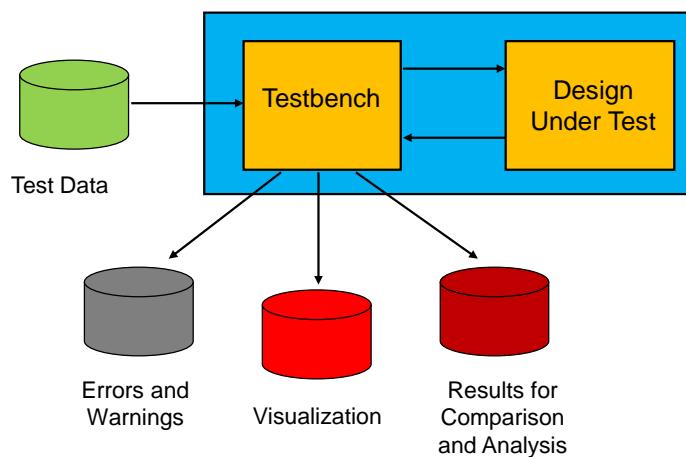
After reading the instruction and address as per the previous slide, the next step is to analyze the instruction and carry out the required function.

- For SEND instructions, read the separator and data as characters; check as required; then call procedure to operate.
- Report invalid instructions.

This example assumes that the conversion of the ADDRESS string (hexadecimal value) to `std_logic_vector` is carried out inside the procedures.

When all the files' instructions have been processed, the `SIM_FINISHED` signal is true. This signal can be used to stop other testbench processes such as clock generators, data monitors, etc.

## Outputs from Simulation (Review)



343 © Cadence Design Systems, Inc. All rights reserved.

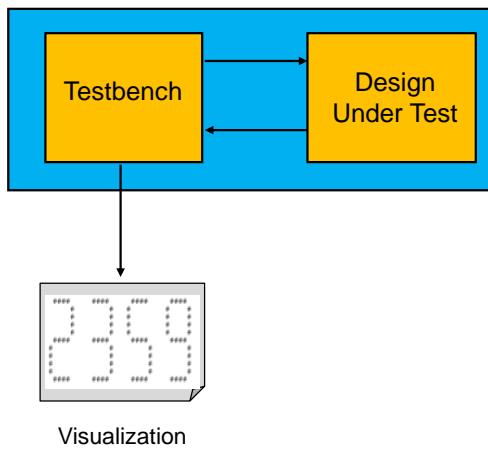


Several different types of output may be generated from a testbench.

FileIO is just as important for capturing data from the simulation as well as applying stimulus. This captured data can be compared and analyzed with expected values to check the behavior of the design.

Assertion statements can also report information such as errors and warnings on simulation. Outputs can also be visualized graphically. The following slides discuss these in more detail.

## Visualized Output



Analyzing binary data for errors can be difficult

Graphical display of data may aid debugging

- This is called visualization
- Useful for output data such as:
  - Displays
  - Image processing

Data can be formatted for visualization in other tools

- Alarm Clock Viewer
- Graphics tools



Binary data is not necessarily the most appropriate method of displaying simulation output, either in a waveform display or output file. Sometimes it can be better if we could have a graphical display of our output: the ability to do this is known as visualization.

In the Alarm Clock labs, it is hard to verify if the correct segments of the seven-segment display are being lit just by looking at the binary data on the output signal.

Visualizing the output by generating output text files from your simulation is possible. The seven-segment display is represented by a two-dimensional array of characters, with either a blank space or a # character being set in each location to make up the pattern of the seven segments. This two-dimensional array of characters can be output to a file each time the output signal changes value.

## Example 1: Visualized Output

```

procedure DISP1 (signal DISPLAY : in std_logic_vector(6 downto 0)) is
    file OUTFILE : text open WRITE_MODE is "clock.txt";
    variable L : line;
    constant SEG_CHAR : character := '#';
    constant SEG_ARR : string(4 downto 1) := (others => SEG_CHAR); -- #####
    -- output is an array of strings
    type T_DIGIT_ARRAY is array (1 to 11) of string (1 to 9);
    variable DIGIT_ARRAY : T_DIGIT_ARRAY;
begin
    -- clear output initially
    DIGIT_ARRAY := (others=>(others=> ' '));
    if DISPLAY(1) = '1' then
        -- Set array elements for segment 1
        DIGIT_ARRAY(2)(7) := SEG_CHAR;
        DIGIT_ARRAY(3)(7) := SEG_CHAR;
        DIGIT_ARRAY(4)(7) := SEG_CHAR;
        DIGIT_ARRAY(5)(7) := SEG_CHAR;
    end if;
    ... -- continued next slide

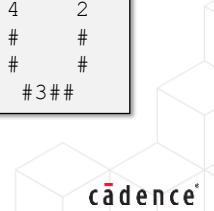
```

```

-- 123456789
-- 1 #0###
-- 2 #
-- 3 5 1
-- 4 #
-- 5 #
-- 6 #6###
-- 7 #
-- 8 4 2
-- 9 #
-- 10 #
-- 11 #3###

```

345 © Cadence Design Systems, Inc. All rights reserved.



This is part of the DISP1 procedure used in the Alarm Clock lab exercises to display one of the Alarm Clock seven-segment display digits. We write the digit onto the output file clock.txt. Hence we open the file in the write mode.

The procedure uses an array of arrays to construct the digit. All the digits can be built using seven segments. Each lit segment of the display is represented by a row or column of 4 # characters. Unlit segments and unused locations are left blank.

The procedure first sets every element to a blank character to clear the display.

Then each bit in the DISPLAY signal is checked, and if it is set, the four corresponding array locations are assigned to #.

So if bit 1 of DISPLAY is set, we set the top right column of the display to lit the first segment.

## Example 2: Visualized Output

```

procedure DISPL1 (signal DISPLAY : in std_logic_vector(6 downto 0)) is
  constant SEG_CHAR : character := '#';
  constant SEG_ARR : string(4 downto 1) := (others => SEG_CHAR); -- "#####
-- other declarations from previous slide
-- bits 2 - 4 omitted

if DISPLAY(5) = '1' then
  -- Set array elements for segment 5
  DIGIT_ARRAY(2)(2) := SEG_CHAR;
  DIGIT_ARRAY(3)(2) := SEG_CHAR;
  DIGIT_ARRAY(4)(2) := SEG_CHAR;
  DIGIT_ARRAY(5)(2) := SEG_CHAR;
end if;

if DISPLAY(6) = '1' then
  -- Set array elements for segment 6
  DIGIT_ARRAY(6)(3 to 6) := SEG_ARR;
end if;

-- now write output array to the file line by line
for I in DIGIT_ARRAY'range loop
  write(L, DIGIT_ARRAY(I));
  writeline(OUTFILE, L);
end loop;
end DISPL1;

```

```

-- 123456789
-- 1 #0## #
-- 2 #      #
-- 3 5      1
-- 4 #      #
-- 5 #      #
-- 6 #6## #
-- 7 #      #
-- 8 4      2
-- 9 #      #
-- 10 #     #
-- 11 #3## #

```

346 © Cadence Design Systems, Inc. All rights reserved.



And if bit 5 of DISPLAY is set, we set the top left column of the display.

And if bit 6 of DISPLAY is set, we set the middle row of the display and so on.

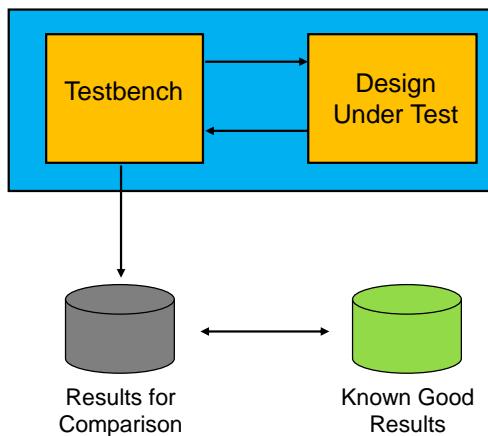
When we have checked all DISPLAY bits, we write the array out to the file, one row at a time, using the for loop.

Note: Since we declared the output file object inside the procedure, we re-recreate the write file every time the procedure is called. We call the procedure a concurrent procedure call as follows:

```
DISPL1(DISPLAY);
```

So every time DISPLAY changes, the procedure is called; the file is re-declared; `clock.txt` is cleared, and the visual representation of DISPLAY is written to the file. Hence the file only contains the current model of DISPLAY.

## Results for Comparison



Compare results for different simulations

Essential for large designs

- Studying waveforms is very tedious!

Output data must be time-independent if comparing between different abstraction levels

- Behavioral/RTL
- RTL/gates
- Data type or timing differences must be allowed for

Results for analysis contain more detail



It is efficient to have expected results for a given design and stimulus to automatically compare the actual versus the desired results. This is typically useful for large designs as it is not feasible to verify the design by looking into the waveforms. Known good simulation results can be very useful to check the results for:

- Simulation of lower abstraction level designs, e.g., behavioral level results, can be used to verify RTL simulations.
  - Data type or timing differences must be allowed.
- Simulation of modified or debugged designs (regression testing) to check functional enhancements or debugging has not introduced further errors.

## Module Summary

### Input methods

- Pseudo-code
- Processor code

### Output methods

- Errors and warnings
- Visualization output
- Results for analysis and comparison



Let us summarize this module. We looked at various methods of driving the stimulus to the DUT, such as pseudo-code and processor code. We also went through various output methods such as errors and warnings using assertions, visualization of output using FileIO, and behavioral methods of generating expected results to compare and analyze against the actual results.

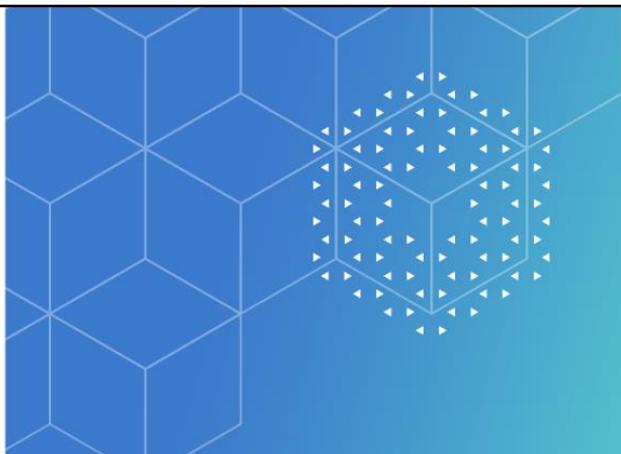
## Lab

### Lab 21-1 Script-Driven Testbench

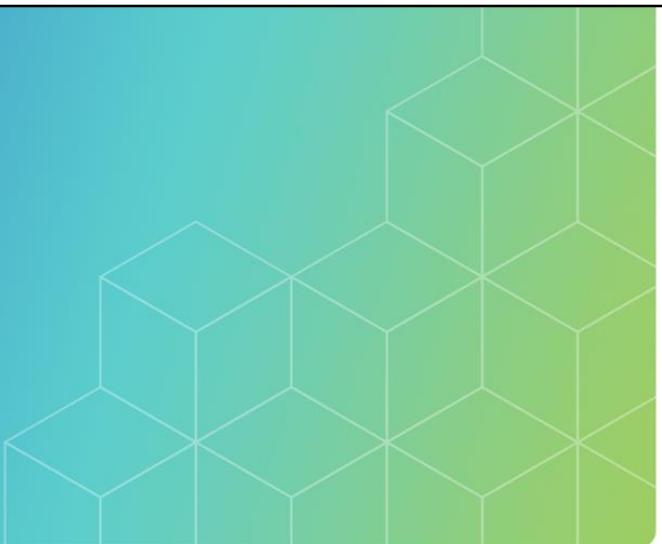
349 © Cadence Design Systems, Inc. All rights reserved.



You will now have the opportunity to complete a self-paced lab to reinforce the ideas presented in this module.



## Gate-Level Simulation



**Module** **22**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

In this module, we look into the gate-level simulation and other timing-related concepts.

## Module Objectives

In this module, you

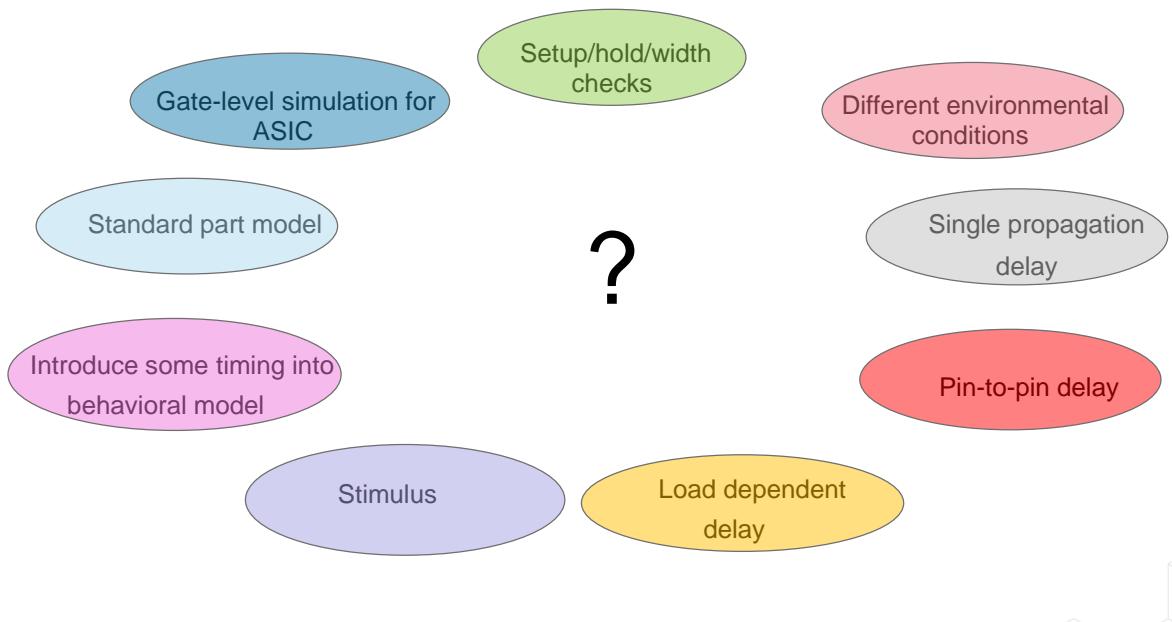
- Describe how timing is handled in VHDL
- List implications for the library user and vendor
- State different timing model requirements
- Define VITAL gate-level simulation standard
- Describe gate-level simulation without VITAL

351 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*

## What to Model and Why



352 © Cadence Design Systems, Inc. All rights reserved.



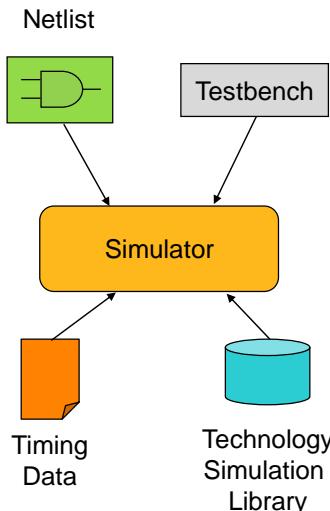
To perform effective gate-level timing simulation, technology cell models (usually vendor supplied) must incorporate some propagation delay information. Also, track delays may need to be backannotated to the netlist.

A testbench will need to model timing to apply the stimulus. The simulation environment may also use behavioral and/or standard part models, which may include timing behavior (e.g., for the bus or interface timing).

Timing checks will be required for gate-level simulation and to check any timing critical or asynchronous behavior at RTL or Behavioral level.

Delay calculation can be very complex. Simple timing models may use a single propagation delay for every gate in a netlist but ignore interconnect delay. More complex delay models consider best case/worst case/typical delay variation; single “lumped” delays for interconnecting – giving a set delay for the whole net, or complex load-dependent delays giving different delays for each individual path in the net. Delays may be scaled by environmental conditions (process, temperature voltage).

## Gate-Level Simulation



### Purpose

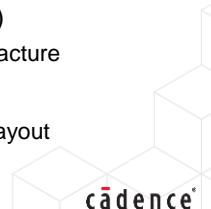
- Verifies function (and sometimes timing) of the gate-level netlist
- Output of synthesis tools may be inaccurate

### Requirements

- Netlist output from the synthesis
- Testbench
- Simulation models of technology cells
- Timing data

### When

- After Layout/Place and Route (P&R)
  - Check netlist before prototype manufacture
- Before layout for ASIC
  - Check netlist before high-cost ASIC layout



353 © Cadence Design Systems, Inc. All rights reserved.

It is important to verify the design netlist to check functionality and timing.

As we have seen from previous modules, synthesis tools cannot interpret or analyze your RTL design to understand your requirements. They produce a very literal translation of your code. Therefore, a synthesis tool can produce a netlist with different functionality from your RTL code. Also, delay models used by synthesis tools can be inaccurate (particularly for large deep-submicron designs). More accurate delay information may be available, e.g., from vendor-specific delay estimators or from Place and Route (P&R), to check design timing during a gate-level simulation. It requires Netlist generated by synthesis, testbench to drive stimulus, simulation models of technology cells, and timing data to perform gate-level simulation.

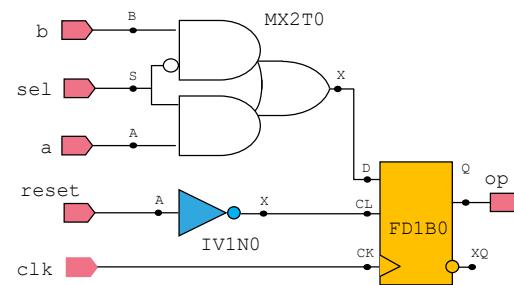
Gate-level simulation is always carried out after P&R as a final check on design functionality and timing. Simulation is also carried out before P&R for ASIC designs. ASIC P&R is an expensive and time-consuming process, so it is essential to check the netlist before beginning the process. Otherwise, errors can be costly!

## Gate-Level Netlist

```

process (CLK, RESET)
begin
  if RESET = '1' then
    OP <= '0';
  elsif RISING_EDGE(CLK) then
    if SEL = '1' then
      OP <= A;
    else
      OP <= B;
    end if;
  end if;
end process;

```



Synthesis

```

ix9 : FD1B0 port map ( Q=>op, XQ=>OPEN, D=>nx6, CK=>clk, CL=>nx48);
ix7 : MX2T0 port map ( X=>nx6, A=>a, B=>b, S=>sel);
ix49 : IV1N0 port map ( X=>nx48, A=>reset);
...

```

Need simulation models of  
components for gate-level simulation

354 © Cadence Design Systems, Inc. All rights reserved.



The slide shows a piece of RTL code translated to netlist by the synthesis tool. The corresponding circuit diagram is shown on the right. The netlist is a series of component instantiations of technology-specific cells linked by signals. We need functional models of these cells for gate-level simulation. These models, usually provided by the silicon vendors, should also include basic timing behavior.

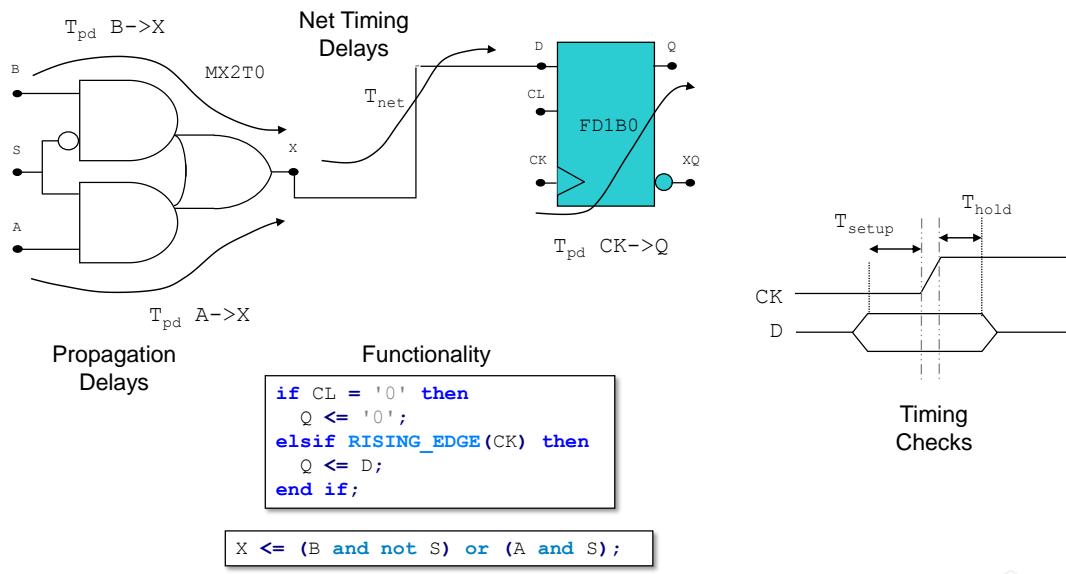
Synthesis and P and R tools can write netlists in VHDL or Verilog. For data interchange, e.g., transfer a design netlist from synthesis to P and R, other formats like EDIF (Electronic Data Interchange Format) may be used.

For VHDL netlists, the instantiated technology components must have corresponding component declarations. These are usually supplied by the technology vendor in a package for convenience. The component declarations are made visible to the netlist by referencing the package.

In this example, our RTL registered single-bit mux has been implemented using an FD1B0 cell for the register and an MX2T0 complex cell for the mux itself. The FD1B0 register has an active-low clear pin, so we need an inverter IV1N0 for our reset. Note that the synthesis tool has produced a netlist with lowercase signal names. This is no problem for VHDL (case insensitive) but can be a problem for other HDLs.

Note: We can leave output ports unconnected by mapping the port to the VHDL keyword OPEN in the port map (e.g., XQ in the instantiation ix9: FD1B0). Input ports cannot be left unconnected.

## Simulation Model Requirements



355 © Cadence Design Systems, Inc. All rights reserved.

cadence®

We require simulation models for the technology-specific cells referenced by our netlist. The vendors usually provide these as a “library” of behavioral level models. The models must include functional behavior for each technology cell, but they must also support timing behavior. Each cell may have several timing paths (a route through the cell from a given input to a given output). Each timing path may have a different propagation delay (timing delay from an input change to an update on the output denoted by  $T_{PD}$ ). The simulation library must be able to model these delays.

Each net (signal connection) in the netlist may also have a delay,  $T_{ent}$ . In Deep Sub-Micron (DSM) technologies, the net delay becomes significant with respect to cell propagation delay and must be accounted for in timing analysis.

In the example on the slide, we see a propagation delay,  $T_{PD}$  associated with the and gates and flip-flop, i.e., from A to X, B to X, and CK to Q, which is a delay from input to output. Also, there is a net timing delay associated with the interconnect; here, the delay is from X to D.

Finally, it is not good modeling timing if we do not include checks to verify that the timing is correct. Typically models of sequential cells will have set up and hold timing checks to check that data inputs do not change within a certain time of the clock edge.

## Simple Gate-Level Model

```

architecture TIMING of FD1B0 is
  constant TSETUP : time := 1 ns;
  constant TCKQ   : time := 0.7 ns;
begin
  DFF: process (CK, CL)
  begin
    if CL = '0' then
      Q <= '0';
      XQ <= '1';
    elsif RISING_EDGE(CK) then
      if D'stable(TSETUP) then
        Q <= D after TCKQ;
        XQ <= not D after TCKQ;
      else
        -- setup timing violation
        report "Setup Violation...";
        Q <= 'X';
        XQ <= 'X';
      end if;
    end if;
  end process DFF;
end TIMING;

```

356 © Cadence Design Systems, Inc. All rights reserved.

```

entity FD1B0 is
  port (D, CLK, CL : in std_logic;
        Q, XQ       : out std_logic);
end FD1B0;

```

'stable is a signal attribute

- Returns true if D stable for previous TSETUP time
- else report statement writes a message to the simulator
- Only one simple method of introducing delays
  - after clause



This slide looks at the simulation model of D flip-flop library cell FD1B0. Timing delays are modeled using an after clause attached to a signal assignment. This schedules a value to be assigned to the target signal after the specified time.

It is also possible to perform all standard timing checks with VHDL.

This slide shows an example of a setup check. On a rising edge of CK, the signal attribute 'stable' is used to check whether the data input D has changed within a period of time before the clock edge specified by TSETUP.

- D'stable(T) returns true if signal D was stable (i.e., has not changed value) over the last T time units.

If D has changed within the set-up time, we highlight the timing violation with a report statement and drive the outputs with 'X.' This writes a message to the simulator transcript window.

In this example, timing delays are specified as constants. In a real simulation library, timing delays could be different for each instantiation of a cell. VHDL has the generic construct to apply instance-specific data to each instantiation of a component. Generics are covered in the Advanced Concurrent VHDL module.

## Timing Delays

Accurate delay modeling can be complicated

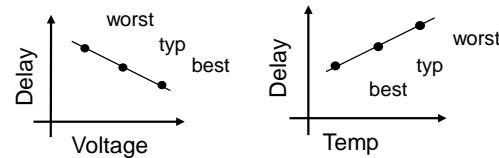
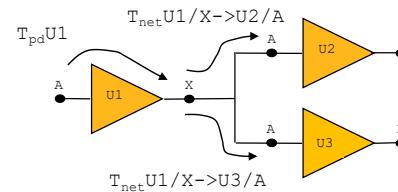
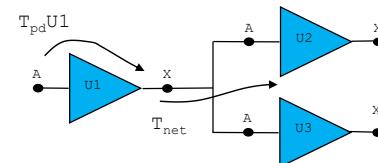
- Single/unit delay
  - Relatively straightforward
- Pin-to-pin timing
  - More accurate
  - Complex and slow

Delays further scaled by

- Environmental conditions
- Net loading

Solution

- Calculate delays externally
- "Backannotate" into a netlist



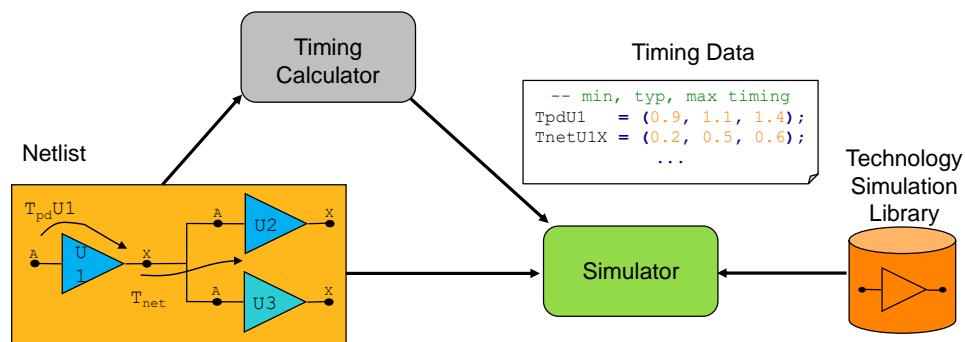
357 © Cadence Design Systems, Inc. All rights reserved.

There are many different methods of modeling and analyzing timing delays in a digital circuit. Simple or unit delay modeling applies a single delay to each net in the netlist. So the delay from pin U1/X to U2/A would be the same as that from U1/X to U3/A. However, on the actual silicon, these two paths may be of different lengths and have different delays. Pin-to-pin delay modeling calculates different delays for the paths U1/X->U2/A and U1/X->U3/A. This is more accurate but is more complex to calculate and slower to simulate.

Delays are also affected by environmental conditions (voltage, temperature, and process variations) and net loading. Some timing models compare the driving strength on the net to the load on the net to refine timing.

The solution to this complexity is to use external tools which analyze a netlist (with knowledge of the technology) and extract timing information. This data is then “backannotated” onto the netlist.

## Backannotation



Several timing values calculated

- Maximum, typical, minimum

Timing data is specific to netlist

- If netlist updated, timing data must be re-calculated

The data format must be compatible with:

- Netlist
- Simulator
- Technology simulation library



A timing calculator reads the VHDL netlist from synthesis. The calculator estimates gate propagation and interconnect delay and scales the delays for environmental conditions. The timing data is passed to the simulator to be annotated to the netlist.

Timing calculators can be used both pre-layout to estimate timing information (ASIC design methodologies require pre-layout timing analysis) and after layout, using more detailed information output from layout tools (e.g., exact net length) to calculate accurate timing.

Timing data usually includes a number of timing values for each timing path. These represent variations in timing due to environmental conditions and tolerances. Timing must be analyzed with both maximum and minimum delays for accuracy.

Timing data is always specific to the netlist from which it was generated. Updating the netlist may remove or add nets to the design, causing backannotation with old data to fail. Data must be re-calculated for every netlist change, no matter how minor.

For successful backannotation, each delay must be associated with the correct net/cell in simulation, and timing checks performed accurately. Therefore, the timing data format must be compatible with the original netlist, the simulator, and the technology simulation library.

## VITAL

VITAL provides a standard for VHDL gate-level simulation:

- VHDL initiative towards ASIC libraries

Defines:

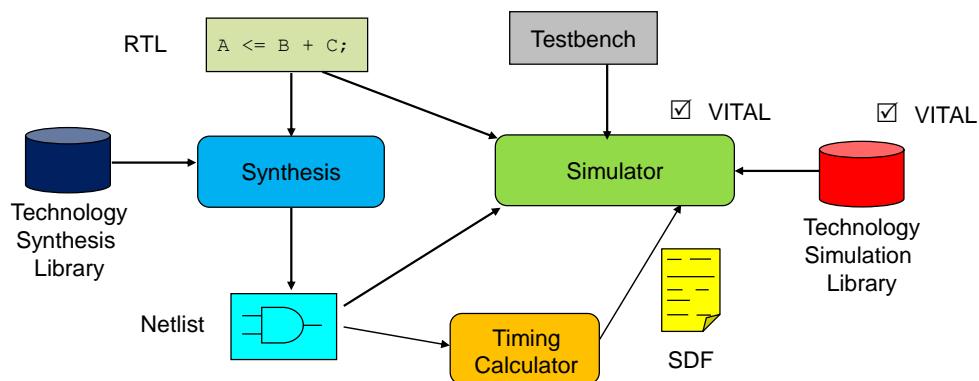
- Format for timing data
  - Standard Delay Format (SDF)
- Method of backannotating timing data
  - Through VHDL generics
    - Instance specific parameters
- Building blocks for technology simulation libraries
  - VITAL\_PRIMITIVES package
  - Can be accelerated in the simulator for performance improvements
- Standard timing checks
  - VITAL\_TIMING package
- Guidelines and conventions for gate-level modeling



Gate-level simulation models can be complex, even though they can be written in behavioral VHDL. By restricting the description of functional and timing behavior to a defined set of modeling primitives, the simulation vendors can “hardwire” (or “accelerate”) these primitives for faster gate-level simulation. VITAL provides the standard for VHDL gate-level simulation. It defines the timing data in a particular file format known as standard delay format. The data provided in the SDF is used to backannotate timing data into the netlist and is explicitly modeled for each instance using generics. The vendors provide certain primitives in a package that acts as building blocks for technology simulation libraries. This accelerates the simulator resulting in performance improvements. Various standard timing checks can be performed by using the VITAL timing package. VITAL also defines guidelines and conventions for gate-level modeling. Writing VITAL compliant simulation library models is a specialized task usually performed by ASIC or FPGA technology vendors. It is beyond the scope of this course.

Generics are covered in the Advanced Concurrent VHDL module.

## VITAL Gate-Level Simulation



### Ideal solution

- Same testbench for RTL and gate-level

### Requires

- VITAL compliant simulator
- VITAL compliant simulation library
- SDF timing data

360 © Cadence Design Systems, Inc. All rights reserved.



Ideally, we want to use just one simulator for both RTL and gate-level simulation.

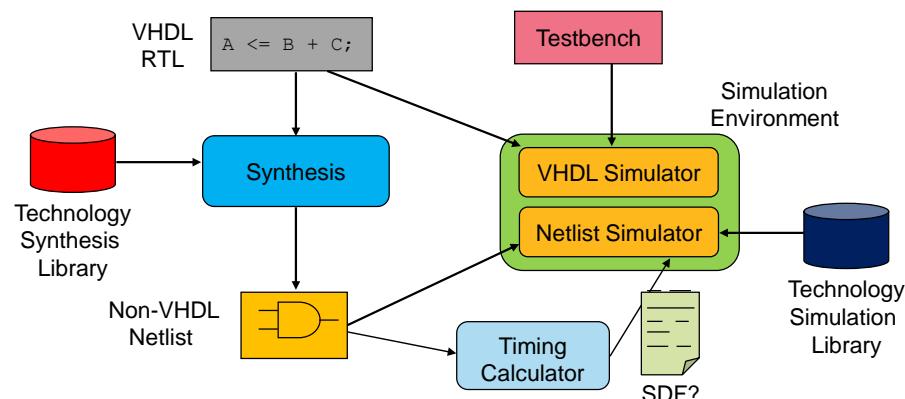
The VITAL gate-level simulation requires both the simulation libraries and the simulator to support the VITAL standard. The timing data has to be in the standard delay format.

- Simulation libraries are usually available from your Silicon vendor.
- Check if the libraries are available for your chosen technology.

This allows us our ideal solution of performing both RTL and gate-level simulation in VHDL.

From the diagram, we see that the RTL and the netlist generated from the synthesis of this RTL use the same simulator and testbench for simulation. To perform timing analysis, timing data is calculated and provided to the simulator in the standard delay format. The simulator uses building blocks from the technology simulation library supplied by the vendor.

## Non-VITAL Gate-Level Simulation



When VITAL libraries are unavailable

Match netlist to available libraries

- For example, Verilog libraries

Two separate simulators

- Co-simulation

Disadvantage

- Need two simulators and one integrated environment

One simulator running two formats

- Co-execution

Some simulators execute both VHDL and Verilog together

- By converting into single format

VITAL libraries may not be readily available for certain technologies (particularly new ASIC technologies). However, simulation libraries may be available in another format (e.g., Verilog). Many EDA companies allow you to link a VHDL simulator directly with another gate-level (e.g., Verilog) simulator. Both simulators would be run in parallel for gate-level simulation: the VHDL simulator for the testbench and the gate-level simulator for the netlist. Vectors are passed back and forth via a dedicated simulator backplane. This is called co-simulation.

However, this usually required three pieces of separately licensed EDA software – two simulators and the integration environment. Therefore, this method tends to be expensive.

For designers familiar with performing gate-level simulation with Verilog, newer simulation technologies allow co-execution. Some simulators can read both VHDL and Verilog models and execute them together. These tools work by compiling both languages down to a single machine code format, which the simulator then runs. So we can use our original VHDL testbench and simulation models to verify a Verilog netlist, using Verilog gate-level simulation libraries with one simulation tool.

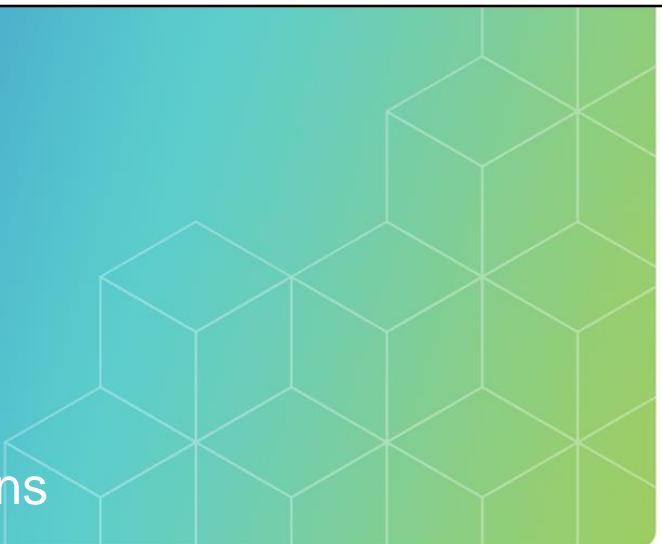
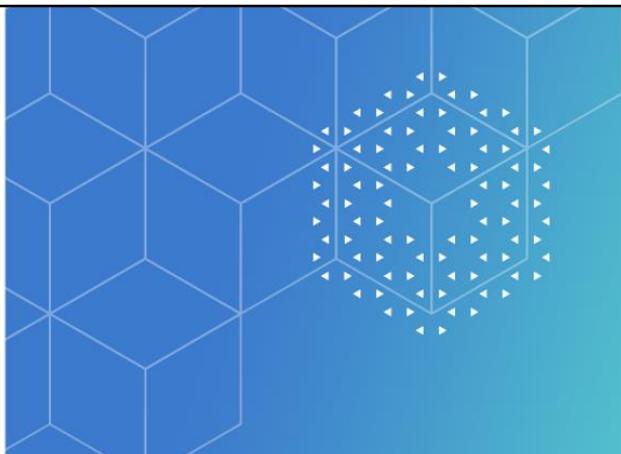
## Module Summary

- Timing modeling can be complex
- Timing specification is flexible in VHDL
- VITAL provides a standard for gate-level simulation in VHDL
- Co-simulation and co-execution provide alternatives

362 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Application of Configurations

**Module** **23**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

This module describes the configuration mechanism and its applications.

## Module Objectives

In this module, you

- Describe the configuration overview
- State the capabilities offered by configurations, and their application to real design situations
- Define entity and port mapping in the configuration
- List different forms of configuration
- Analyze the application issues for each of these forms



*This page does not contain notes.*

## Representing Hierarchy

```

library IEEE;
use IEEE.std_logic_1164.all;

entity FULLADD is
  port ( X, Y, CIN : in std_logic;
         SUM, COUT : out std_logic );
end FULLADD;

architecture STRUCTURAL of FULLADD is

  component HALFADD
    port( A, B      : in std_logic;
          SUM, CARRY : out std_logic );
  end component;

  signal N_SUM, N_CARRY1, N_CARRY2 : std_logic;
begin

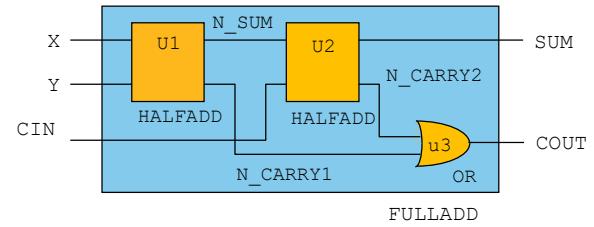
  U1: HALFADD port map (X, Y, N_SUM, N_CARRY1);

  U2: HALFADD port map (N_SUM, CIN, SUM, N_CARRY2);

  COUT <= N_CARRY1 or N_CARRY2;

end STRUCTURAL;

```



- The component statement is a template (“socket”)
- The configuration makes the link

365 © Cadence Design Systems, Inc. All rights reserved.



To build up a hierarchy in VHDL, we need to declare components and instantiate them to provide the “sockets” into which we plug “chips.” In this case, the “chip” is an entity AND a single architecture of that entity that defines the interface and behavior. The VHDL construct that “plugs” the entity/architecture into the component “socket” is the configuration statement. The code on the slide depicts a full adder built using two half-adders, as shown on the right. The architecture of the full adder contains the component declaration and instantiation of the half-adder, which forms the socket. Which entity/architecture pair of half adder is plugged into this socket is determined by the configuration statement which we see in the coming slides.

## Default Configurations

```
...
component HALFADD
  port( A, B : in std_logic;
        SUM, CARRY: out std_logic );
end component;
...
```

default configuration

```
entity HALFADD is
  port ( A, B : in std_logic;
         SUM, CARRY : out std_logic );
end HALFADD;
```



Use copy/paste to ensure port lists match

Tip

Component declaration/instantiation creates a "socket"

- Not a direct connection to an entity

Which entity and architecture is to be plugged is defined by the configuration statement

If certain rules are followed, component declaration and an existing entity can be automatically linked

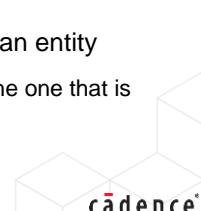
- Called default configuration

Rules for a default configuration

- Entity and component names **must** match
- Port names/types **must** match (can be declared in a different order)

If more than one architecture exists for an entity

- The last architecture to be compiled is the one that is used



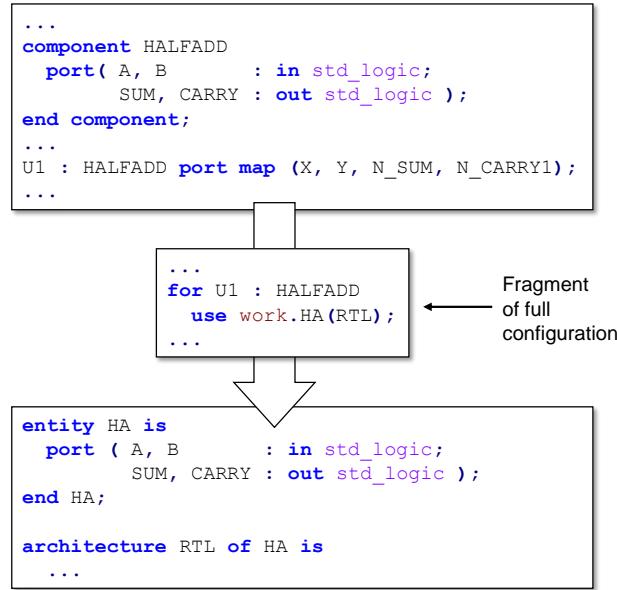
As we saw in the previous slide, the VHDL construct that “plugs” the entity/architecture into the component “socket” is the configuration statement.

The configuration statement defines which entity and the architecture of that entity will be used to plug into a particular component socket. You can think of it as a “parts list” for the design.

When the names and port lists of the component and the entity are identical, we don't need to write an explicit configuration. The simulator can link the component and entity automatically. This is called a default configuration. If you are using a default configuration but have more than one architecture for an entity, the last architecture to be compiled is the one that is used.

Default configurations are the only configuration method generally supported by synthesis tools.

## Explicit Configurations



Default configurations are sometimes unsuitable:

- Different entity/architecture for each component instance
- Component/entity names may differ
- Component/entity port lists may differ

Configuration used to explicitly link hierarchy:

- Selects specific entity/architecture pair for each component instance

Explicit configuration is generally not supported by the synthesis tool

Consider as a “build list” for hierarchy



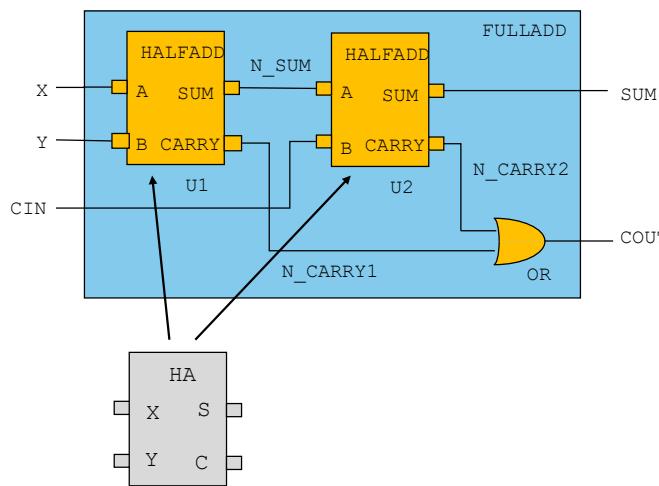
367 © Cadence Design Systems, Inc. All rights reserved.

Default configurations are sometimes not suitable. If we need to map different instances of a component to different entities, or the names or port lists do not match, then we would need to explicitly write a configuration. Configurations are used to explicitly link the hierarchy; it selects specific entity/architecture pair for each component instance.

Explicit configurations are not generally supported by synthesis tools. Configurations are very useful in simulation, however, allowing you to quickly “build” various views of your design for simulation. You can consider it as a build list for hierarchy.

The full configuration is not shown here, but the fragment shows how we could link a specific entity and architecture to every instantiation of the component HALFADD. This statement specifies to link entity-architecture pair HA-RTL to instance U1 of HALFADD.

## Configuration Capabilities



Link models from different sources

- Entity mapping
- Port mapping
- Generic mapping

```
entity HA is
  port ( X, Y : in std_logic;
         S, C : out std_logic );
end HA;
architecture TIMED of HA is
begin
  S <= X xor Y after 5 ns;
  C <= X and Y after 5 ns;
end TIMED;
```



If we need to map different instances of a component to different entities, or if the names or port lists do not match, then we would need to explicitly write a configuration.

Mapping capabilities enable models from different designers, using different names, types, etc., to be used. For example, each of the two instances of the half-adder can be mapped to different entities using different names, types, etc., using configuration.

## Configuration Example

`for all` configures all instantiations

Instantiations can be separately configured:

```
for U1 : HALFADD use entity work.E1(AA);
for U2 : HALFADD use entity work.E2(AB);
for others: HALFADD use entity work.E2(BB);
```

```
entity FULLADD is
  port ( X, Y, CIN : in std_logic;
         SUM, COUT : out std_logic );
end FULLADD;

architecture STRUCTURAL of FULLADD is

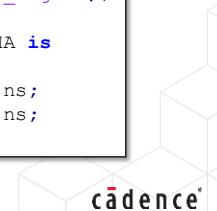
  component HALFADD
    port( A, B : in std_logic;
          SUM, CARRY : out std_logic );
  end component;

  signal N_SUM, N_CARRY1, N_CARRY2 : std_logic;
begin
  U1: HALFADD port map (X, Y, N_SUM, N_CARRY1);
  U2: HALFADD port map (N_SUM, CIN, SUM, N_CARRY2);
  COUT <= N_CARRY1 or N_CARRY2;
end STRUCTURAL;
```

```
configuration CFG_FULLADD_DELAY of FULLADD is
  for STRUCTURAL
    for all : HALFADD
      use entity work.HA(TIMED)
        port map (X => A, Y => B,
                  S => SUM, C => CARRY);
    end for;
  end for;
end CFG_FULLADD_DELAY;
```

```
entity HA is
  port ( X, Y : in std_logic;
         S, C : out std_logic );
end HA;
architecture TIMED of HA is
begin
  S <= X xor Y after 5 ns;
  C <= X and Y after 5 ns;
end TIMED;
```

369 © Cadence Design Systems, Inc. All rights reserved.



The configuration `CFG_FULLADD_DELAY` is associated with the entity `FULLADD` and configures the architecture `STRUCTURAL`. It tells us:

- For all instantiations of the component `HALFADD`, use the entity named `HA`, with the architecture `TIMED`, from the library `work`.
- Map port `X` on the entity `HA` to port `A` on `HALFADD`, `HA` port `Y` to `HALFADD` port `B`, etc.
  - Note: named port mapping is more readable than positional

The first `end for;` matches `for all`, the second `end for;` matches `for structural`

- Use indentation to keep track!

Note that we could configure the individual instantiations of `HALFADD` to different entity/architectures using separate clauses, e.g.:

```
for U1:HALFADD use entity work.E1(AA) ...
for U2:HALFADD use entity work.E2(AB) ...
```

We can also use an `others` clause, which is useful for multiple instantiations:

```
for U1:HALFADD use entity work.E1(AA) ...
for others:HALFADD use entity work.E2(AB) ...
```

## Configuration Mapping

```
architecture TEST of CONF_MAP is
component COMPMAP
    port ( A : in unsigned(7 downto 0);
           OP:out unsigned(7 downto 0) );
end component;

signal STD_IP : unsigned(7 downto 0);
signal STD_OP : unsigned(7 downto 0);

begin
    U3 : COMPMAP port map (STD_IP, STD_OP);
    ...

```

```
entity COMPGEN is
    generic (LEN : integer);
    port (AUS : in std_logic_vector(LEN -1 downto 0);
          OPUS: out std_logic_vector(LEN -1 downto 0));
end COMPGEN;
architecture RTL of COMPGEN is
    ...

```

```
configuration CONFMAP_CFG of CONF_MAP is
for TEST
    for U3 : COMPMAP use entity work.COMPGEN(RTL)
        generic map (LEN => 8)
        port map (AUS => std_logic_vector(A), unsigned(OPUS) => OP);
    end for;
end for;
end CONFMAP_CFG;
```

Type conversion

370 © Cadence Design Systems, Inc. All rights reserved.

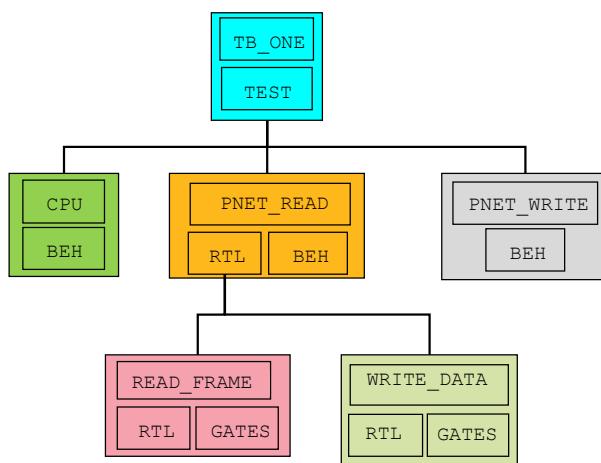
**cadence®**

Configurations are quite powerful and are becoming increasingly important for large complex system simulations.

In the example on the slide, we need a component COMPMAP with input and output ports of type `unsigned` and length 8. We have a model COMPGEN with generic-sized ports, but of the wrong type (`std_logic_vector`). Using configuration, we can specify the correct generic value and perform type conversion on the ports in the port map.

We could create several configurations, building a different simulation “view” of our system.

## Configuration Organization



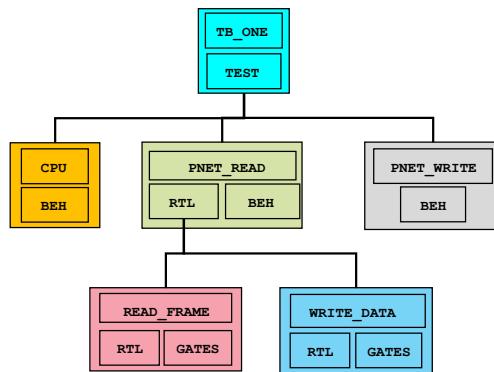
Three different major organizations of configuration:

1. One top-level configuration references all components/entities
2. Configuration references other configurations
3. Architecture configuration

There are three different major organizations of configuration possible; one top-level configuration references all components/entities, two configuration references other configurations and three architecture configurations.

We will look at these in the following few slides using the example shown in the diagram. Each block in the diagram contains two rows, the first one denotes the entity name, and the second row denotes the architecture associated with it.

## 1. Single Top-Level Configuration



```

library MYLIB;
configuration CFG_TEST_BEH of TB_ONE is
for TEST

for all: PNET_READ
use entity MYLIB.PNET_READ(RTL);
for RTL
for all: READ_FRAME
use entity MYLIB.READ_FRAME(RTL);
end for;
for all: WRITE_DATA
use entity MYLIB.WRITE_DATA(RTL);
end for;
end for;

for all: PNET_WRITE
use entity MYLIB.PNET_WRITE(BEH);
end for;

for all: CPU
use entity MYLIB.CPU(BEH);
end for;

end for;
end CFG_TEST_BEH;
  
```

372 © Cadence Design Systems, Inc. All rights reserved.



The first style of configuration organization is the single top-level configuration. In the example, the architecture RTL of entity PNET\_READ contains further hierarchy (READ\_FRAME and WRITE\_DATA components), which requires configuring.

Therefore, once we have specified using entity PNET\_READ and architecture RTL for all instantiations of the component PNET\_READ, we must then configure the details in architecture RTL. We enclose the configuration of these components with, `for RTL`, and `end for`; these are statements to signify that they are part of the architecture RTL hierarchy. This is followed by the configurations for the remaining entities, PNET\_WRITE and CPU.

This configuration method has the benefit of having the specification of how the whole hierarchy is built in one place.

Note that all the components are configured with design units from the library MYLIB referenced by the library statement.

- If our hierarchical models are in different libraries, we would need additional library statements to reference the other libraries.

## 2. Referencing Other Configurations

```

library MYLIB;
configuration CFG_TEST_BEH of TB_ONE is
  for TEST

    for all:PNET_READ
      use configuration MYLIB.CFG_PNET_READ_RTL;

    end for;

    for all:PNET_WRITE
      use entity MYLIB.PNET_WRITE(BEH);
    end for;

    for all:CPU
      use entity MYLIB.CPU(BEH);
    end for;

  end for;
end CFG_TEST_BEH;

```

```

library MYLIB;
configuration CFG_PNET_READ_RTL of PNET_READ is
  for RTL

    for all:READ_FRAME
      use entity MYLIB.READ_FRAME(RTL);

    end for;

    for all:WRITE_DATA
      use entity MYLIB.WRITE_DATA(RTL);
    end for;

  end for;
end CFG_PNET_READ_RTL;

```

373 © Cadence Design Systems, Inc. All rights reserved.

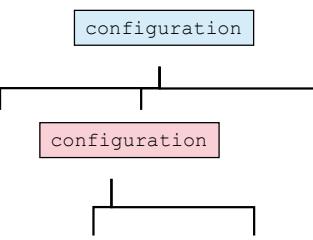
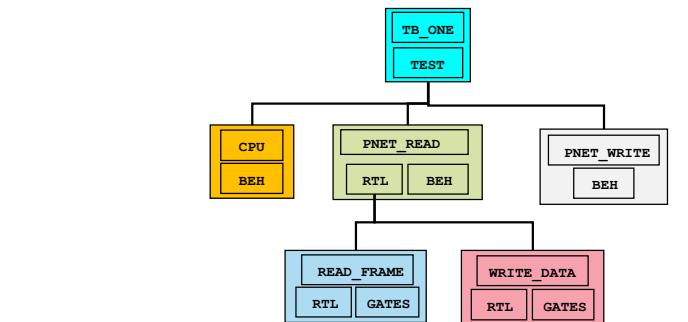


In style two, we see that we can reference other configuration design units in a configuration statement.

The configuration of the architecture RTL of PNET\_READ has been created in a new configuration design unit CFG\_PNET\_READ\_RTL. In this design unit, we specify the entity-architecture pair used by the READ\_FRAME and WRITE\_DATA component socket. We can now reference this configuration for PNET\_READ in the original configuration with the use configuration. This is followed by configurations for PNET\_WRITE and CPU component instances.

Note that we specify the library into which the configuration or entity/architecture design units have been compiled.

### 3. Architecture Configurations



```

entity TB_ONE is
end TB_ONE;

architecture TEST of TB_ONE is
  -- component declarations
  for ALL: CPU      use entity work.CPU(BEH);
  for ALL: PNET_READ use entity work.PNET_READ(BEH);
  for ALL: PNET_WRITE use entity work.PNET_WRITE(BEH);
begin
  -- instances of CPU, PNET_READ, PNET_WRITE
end TEST;
  
```

374 © Cadence Design Systems, Inc. All rights reserved.



#### Disadvantage

Architecture needs to be edited and re-compiled to change the configuration information

In style three, we can also add configuration information to an architecture.

The architecture configuration can be used in each hierarchical block that instantiates components. Note that the architecture must still contain declarations for each component. Port and generic maps can be added to the architecture configuration also, e.g., for the FULLADD design seen previously:

```

architecture STRUCTURAL of FULLADD is
  component HALFADD
    port(A, B : in std_logic;
         SUM, CARRY : out std_logic);
  end component;
  for all : HALFADD use entity work.HA(TIMED)
    port map (X => A, Y => B, S => SUM, C => CARRY);
  ...
  
```

This method means that the configuration design unit is not required, and the simulation can be invoked on the top-level entity.

Note that the architecture must be edited and recompiled to change the configuration information.

## Configuration Recommendations

Used to control simulation builds

Just using default

- Easy to use but restrictive
- Most commonly used on the first project

Top-level configuration

- Commonly used
- Flexibility

Architecture configurations

- Less flexible
- More compilation overhead

Referencing other configurations

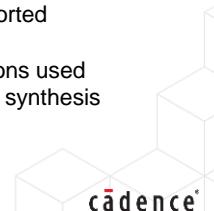
- One top-level with several at the next level
- Large designs with several engineers



Synthesis

Explicit configurations generally not supported

Default configurations used when compiling for synthesis



375 © Cadence Design Systems, Inc. All rights reserved.

Let us look at the pros and cons of different configuration organizations.

Default configurations are easy to use but restrictive.

The top-level configuration is the most common and provides the most flexibility.

Architecture configuration:

- Less flexible.
- More compilation overhead – changing the hierarchy requires recompilation of the architecture.

For large designs with several engineers working on separate hierarchical blocks, it can be helpful to have separate configurations for each block referenced by a top-level configuration. Each engineer can then maintain an up-to-date configuration describing their block, which can easily be referenced by other engineers wishing to simulate the whole design.

Having a separate configuration for each entity is not very common because this leads to many design unit files, causing design management problems.

## Direct Instantiation

- Can make direct instantiations of entities
- Do not need component declarations/configurations
- Architecture name is optional
- Architecture needs to be edited and re-compiled for any change in design configuration
- Can also make instantiations of configurations
- Lose flexibility of components

```

entity FULLADD is
  port ( A, B, CIN : in std_logic;
         SUM, CARRY : out std_logic);
end FULLADD;

architecture STRUCTURAL of FULLADD is
  signal N_SUM, N_CARRY1, N_CARRY2 : std_logic;
begin
  u1: entity work.HA(BEH) port map (A, B, N_SUM, N_CARRY1);
  u2: entity work.HA(BEH) port map (N_SUM, CIN, SUM, N_CARRY2);
  ...
end STRUCTURAL;

```

```

entity HA is
  port ( X, Y : in std_logic;
         S, C : out std_logic);
end HA;

architecture TIMED of HA is
begin
  S <= X xor Y after 5 ns;
  C <= X and Y after 5 ns;
end TIMED;

```

376 © Cadence Design Systems, Inc. All rights reserved.



From VHDL'93, configuration information can be included in a component instantiation. No component declaration is required in the architecture.

Instead of instantiating a component declared in the architecture, we can directly instantiate an entity, entity/architecture pair, or configuration.

Since these are direct instantiations built into the architecture, the architecture will need to be edited and recompiled to modify the design configuration.

Note that positional port mapping is best to use in direct instantiations. Compilation errors may be seen if a named association is used since the compiler may not be able to see declarations for the entity ports X, Y, S, and C. Here, we see that the full adder design directly instantiates half-adder along with the configuration information. Also, no half-adder component is declared before instantiation.

## Configuration Summary

A configuration specifies how to link together the hierarchy

Allows selection of:

- Specific entities/architectures
- Library design units to be taken from

You may choose to use the default if:

- Names match
- Only one entity/architecture to choose from
- Only one library to choose them from
- Design to be synthesized

Otherwise, you should be specific:

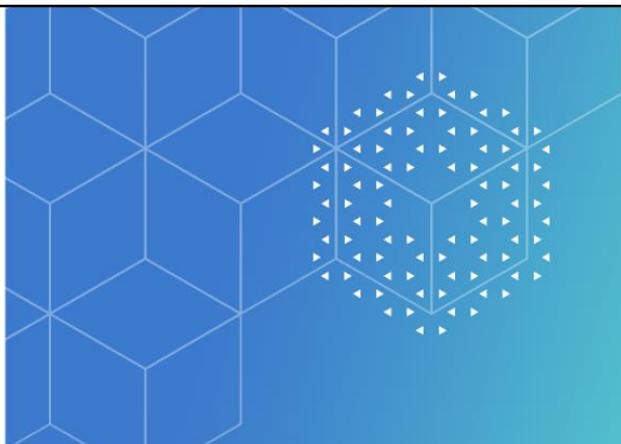
- Allows linking models from different designers, from various libraries, using other names, etc.
- Used to construct simulation environments to verify the design



Let us summarize this module. A configuration specifies how to link the hierarchy. It allows the selection of entity/architectures and the library design units to be taken from. We can use default configurations under the restrictions of name matching, only one entity/architecture to choose from, only one library to choose from, and design hierarchies for synthesis. Synthesis tools do not widely support configurations.

Specific configurations allow us to link models from different designers, from various libraries, using other names, containing generics, etc.

Default configurations are generally used for design hierarchies intended for synthesis, and specific configurations are used to construct simulation environments to verify the design.



## Design Organization and Management

**Module** **24**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

This module discusses concepts related to designing organization and management.

## Module Objectives

In this module, you

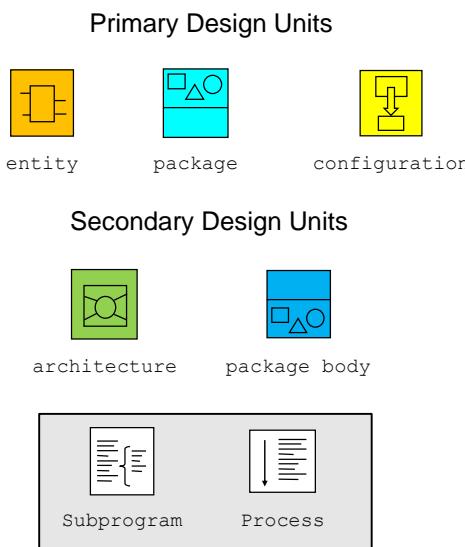
- List the issues that affect the organization of your design
- Define design units
- Describe effective use of packages within design teams
- Define namespaces
- State and describe compilation order
- Describe design partitioning across files



In this section, we look at the language issues that affect design management and then explore ways in which these issues can be applied to the organization of your design files.

Topics that we will cover include the definition of design units and the rules for visibility within them, the rules about compilation order, and finally, how these issues affect the management of files.

## Design Units



### Design unit

- Basic building block of VHDL
- Individually compiled into libraries

Five design units classified into primary and secondary

- Entity, package, and configuration are primary design units
- Architecture and package body form secondary units
- Secondary units are explicitly associated with a primary unit

Subprograms and processes are not design units

- They don't build a hierarchy
- Not compiled separately



380 © Cadence Design Systems, Inc. All rights reserved.

Design units are the basic building blocks of a VHDL design.

- When a file containing multiple design units is compiled, the individual design units are extracted from the file and separately compiled into the working library.

There are only five design units in VHDL which are those shown here.

They are split into two categories: primary and secondary.

- Architectures and package bodies are secondary design units because they must be associated with a primary design unit – an entity or package.

Remember, a package itself contains the declarations. The package body must contain the details of any functions or procedures declared in the package.

Subprograms and processes don't build up the hierarchy in synthesizable VHDL description and are not compiled separately. Hence, they are not design units.

## Packages and Package Bodies

```
package DEMO_PACK is
    -- constants
    -- data types
    -- components
    -- subprogram declarations
end DEMO_PACK;
```

```
package body DEMO_PACK is
    -- data types
    -- constant values
    -- subprogram bodies
end DEMO_PACK;
```

```
library PACKS;
use PACKS.DEMO_PACK.all;
...
```

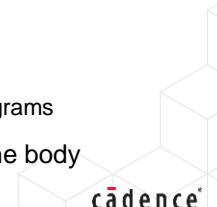
### Package

- Useful declarations
  - Types, constants, components, and subprograms
- Subprograms
  - Declared in package
  - Definition in the package body
- Only visible when referenced
- Referenced by `use` clauses

### Package body

- Associated with package
- Contains
  - Definitions of subprograms in a package
  - Contains local types, constants, subprograms
- Local declarations only visible within the body

381 © Cadence Design Systems, Inc. All rights reserved.



Packages and package bodies are split into separate design units to help manage complexity. The package contains only declarations of types, constants, components, and subprograms.

Subprogram declaration is placed in the package, which is made visible to the user, while subprogram implementation details are hidden in the body, concealed from the user. Implementation can be changed without affecting the users (as long as the declaration remains unchanged).

Packages must be explicitly referenced with `use` clauses to make their contents visible to a specific entity/architecture pair.

The package body is associated with a package and contains definitions of subprograms, local types, and constants:

- Functions and procedures in packages must be declared in the package, but functional code associated with subprograms is placed in the package body.
- Subprogram functional code can use local type and constant declarations from the package body and local subprograms defined in the package body.
- Items declared in the package body are for local use only and are not visible to the package or to any design unit which references the package.

## Example: Package Visibility

```
package P1 is
...
type MY_TYPE1 is ...
type MY_TYPE3 is ...
procedure MY_PROC1 (...);
...
end P1;
```

```
package body P1 is
...
type MY_TYPE2 is ...
procedure MY_PROC2 (...) is
begin
...
end MY_PROC2;

procedure MY_PROC1 (...) is
begin
...
end MY_PROC1;
```

```
package P2 is
...
type MY_TYPE3 is ...
constant CONST1 ...
...
end P2;
```

```
library PACKS;
use PACKS.P1.all;
use PACKS.P2.all;

entity E is
...
end E;
architecture BEH of E is
signal MY_SIG1:MY_TYPE1; 
signal MY_SIG2:MY_TYPE2; 
signal MY_SIG3:MY_TYPE3; 
begin
...
MY_PROC1(...); 
MY_PROC2(...); 
...
X <= A when X = CONST1 else 
B;
...
end BEH;
```

382 © Cadence Design Systems, Inc. All rights reserved.



Let's look at some of the rules related to package visibility. The example on the slide contains two packages, P1 and P2, which are referenced by the design unit on the right using the use clauses. The example demonstrates various legal and illegal package content access made by the design unit. Declaration of MY\_SIG1 is allowed because MY\_TYPE1 was declared in package P1 and is referenced before the entity. The declaration of MY\_SIG2 is illegal because MY\_TYPE2 is declared only in the package body P1 and not in the package declaration, so it is not visible to any design unit referencing package P1. The declaration of MY\_SIG3 is illegal because the type MY\_TYPE3 is declared in two packages, both of which are referenced before the entity. Because these are "homonyms" (have the same names) as one another, the simulator cannot determine which to use, so it makes neither of them visible.

Similarly, The call to procedure MY\_PROC1 is also allowed because it was declared in package P1 and defined in package body P1, and this package was then referenced using the use clause before the entity. The call to procedure MY\_PROC2 is invalid because MY\_PROC2 is not declared in the package but only in the package body.

Note: The subprogram package declaration consists of the first line of the subprogram definition, giving the name and parameter list.

## Scope of Library and Use Clauses

Need library clauses here

```
library PACKS;
use PACKS.PCON.all;

entity E1 is
...
end E1;

architecture RTL of E1 is
...
begin
...
end RTL;
```

No library clauses needed here

```
library PACKS;
use PACKS.PCON.all;
entity E2 is
...
end E2;
```

```
architecture RTL of E2 is
...
begin
...
end BEH;
```

Library clause makes library visible

Use clause makes specific library contents visible

The scope is:

- Design unit that immediately follows the use clauses
- All related secondary design units
  - File independent

Library and use clauses must be repeated for multiple entities within a file

383 © Cadence Design Systems, Inc. All rights reserved.



In order to reference objects within a library, it is first necessary to define a logical name to reference that library in the VHDL code. This is done by the library clause.

- Here, we declare that the name PACKS refers to a library.

To make objects within a library visible, use the use statement.

- Here, we are making all objects contained within the package PCON (which should be compiled in the library PACKS) visible to the code that follows.
- Remember, the simulator defines the mapping of the logical library name PACKS to a specific directory that contains the compiled code for PCON.

A library clause on its own simply declares a logical name to reference a library; it does not make items within that library visible.

Library and use clauses only apply to the design unit, which immediately follows in the source file.

Any object made visible to an entity is then automatically made visible to all of its architectures, even if they are defined in different files. Hence, no need for a use clause before architecture.

Library and use clauses must be repeated for multiple entities in a file.

## Recommendations: Packages

Share definitions

- Types
- Subtypes
- Constants
- Common subprograms

```

...
package MASK_TYPES is
    constant PERIOD : time := 20 ns;
    constant LINE_DATA_WIDTH : natural := 5;
    constant LINE_LENGTH : natural := 2**LINE_DATA_WIDTH;
    subtype FILT_DATA is integer range -32768 to 32767;
    subtype SLV8 is std_logic_vector(7 downto 0);
...
```

```

package PROCS is
    procedure MAJOR ..... -- declaration
end PROCS;
```

```

package body PROCS is
    procedure ELEMENT1 ...
    ...
end ELEMENT1;

procedure ELEMENT2...
...
end ELEMENT2;

procedure MAJOR ..... -- definition
    ELEMENT1(...);
    ELEMENT2(...);
end MAJOR;
end PROCS;
```

The structure of the package and the package body allows for complexity management

Only shared subprograms declared in the package

- Complexities of implementation hidden in the body

384 © Cadence Design Systems, Inc. All rights reserved.



Let us look at some of the recommendations related to packages. Careful and intelligent use of packages can make design easier by allowing us to reuse existing types, constants, and common subprograms in other areas of our design or other design projects.

The structure of the package and the package body allows simple complexity management through information hiding. Shared subprograms are declared in the package, where they can be easily browsed by users (see standard package listings in the appendices). The implementation of these subprograms, together with the definitions of types, constants, and subprograms required by the implementation, are hidden in the package body.

## Namespaces

```
use work.P.all;
architecture A ....
  signal I : ...;
begin
  I <= ...;
```

```
A1 : process...
  variable I : ...
begin
  if I = ...
    for I in ...loop
    ...
    end loop;
```

```
end process A1;
```

```
P2 : process...
  variable I: ...
begin
  ...
end process A2;
```

```
end A;
```

```
package P ...
  constant I : ...;
```

```
use work.P.all;
architecture B ...
  signal I ...;
begin
  process B1 ...
  begin
    if I = ...
      ...
    end if;
    ...
    if work.P.I = ...
      ...
    end if;
    ...
  end process B1;
```

385 © Cadence Design Systems, Inc. All rights reserved.

Code is broken down into namespaces

Objects only accessible in the namespace where defined

- Architecture
- Process
- Function/procedure
- Loop (for loop variable)

Multiple objects of the same name allowed

- Only if declared in different namespaces
- Local name overrides

Pathname to package object can help resolve



Objects are only visible within the area where they are declared, e.g., an architecture, process, function/procedure, or a loop. This is known as the object's namespace. This allows different objects within different namespaces to have the same name.

Where namespaces “nest,” e.g., a loop statement within a process or a process within architecture, the most local name takes priority.

In the example above, a reference to the object I within the loop statement in process A1 picks up the loop variable. A reference to I outside the loop picks up the variable I declared in the process.

Reference to I in process P2 picks up the local variable, and reference to I outside the processes in architecture A picks up the signal declaration.

The constant declaration I in package P will be masked by the signal declaration and so never seen in architecture A.

In architecture B, the declaration of signal I would normally mask the constant I declared in the package P, but we can use an absolute pathname to reference package objects using the syntax library.package.object.

Note: You should not create code to take advantage of namespace features; instead, use meaningful, unique names for objects.

## Compilation Order Rules

Rule 1: Primary before secondary

- This means compile:
  - Entity before architecture
  - Package before package body
- Rule 1 only applies if design units are in separate files

Rule 2: Design the unit before it is referenced

- This means compile:
  - Package before entity/architecture which references package
  - Configuration after entity/architecture

These dependencies also apply for recompilation

- Architecture if entity recompiled (only if in separate files)
- Package body if package recompiled (only if in separate files)
- Entity/architecture referencing recompiled package
- Configuration referencing recompiled entity/architecture



There are two basic rules regarding compilation order:

- A primary design unit must be compiled before its secondary because the secondary depends on the information in the primary. Therefore, you must compile:
  - Entity file before architecture file.
  - Package file before body file. This rule applies if design units are in separate files.
- A design unit must be compiled before any unit that references it. Therefore, you must analyze:
  - Package file before entity/architecture file(s) which reference the package.
  - Configuration file after entity/architecture files(s) referenced by configuration.

These compilation dependencies can cause problems with recompiling your design, particularly if your design units are in separate files.

For example, you find an error in an entity during the simulation and edit the entity file. If both design units are in the same file, the architecture will be recompiled with the entity. If the entity and architecture are in different files, you cannot recompile just the entity and then re-elaborate the design – the compiler will report errors. Since the architecture depends on the entity, you must recompile the architecture file after the entity file.

## Deferred Constants

```
pcon_p.vhd
package PCON is
    constant ROWS : integer := 32;
end PCON;
```

```
pdef_p.vhd
package PDEF is
    constant ROWS : integer;
end PDEF;
```

```
pdef_b.vhd
package body PDEF is
    constant ROWS : integer := 32;
end PDEF;
```



Deferred constants are generally supported

Synthesis

Changing a package can cause much recompilation

- Every unit which references the package

Deferred constants useful

- Name/type declared in package
- Value declared in the body (separate file)
- Saves a lot of compilation time

Changing value means the only body needs recompilation

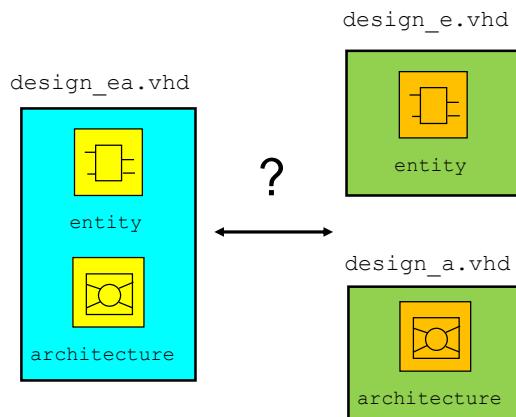
387 © Cadence Design Systems, Inc. All rights reserved.



As we saw from the compilation example, if we make changes to a package, we need to recompile every design unit which references that package. However, if we modify a package body, we do not need to recompile any other design units. It is possible to declare constants in a package header and give it a value in the body. This is called a “deferred constant.”

This can save a lot of compilation time. In the example on the slide, process PCON declares a constant named ROWS of type integer with value 32. If for some reason, this value is changed, then all design units referencing this package need to be re-compiled. This is overcome in the package PDEF, where constant ROWS of type integer is only declared in the package but is assigned a value in the package body. If we have to change the value, only the package body needs to be re-compiled, which saves a lot of time.

## Design Units and Files



Package separate from the body

- Deferred constants
- Subprograms

Separate files for entity/architecture?

- Less analysis, multiple architectures
- More files

Configurations

- Separate files



Now we understand the rules about compilation order, we can see how this will influence the separation of design units between files.

There is an advantage in separating a package from its body, as seen from the previous slides.

The issues in separating entities and architectures into different files are less clear.

- Only one architecture and it is better to have just one file.
- More than one architecture, and it is important to keep them separate.

## Summary Quiz: Design Organization

1. What are the three primary design units?
2. What are the two rules of compilation order?
3. What is the scope of a `use` statement?



*This page does not contain notes.*

## Summary Quiz: Design Organization (Solutions)

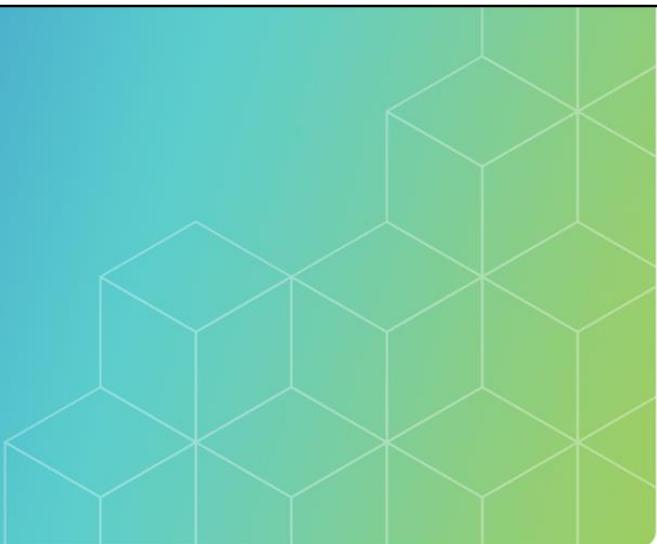
1. What are the three primary design units?
  - Entity, package, and configuration
2. What are the two rules of compilation order?
  - Primary before secondary (if they are in separate files)
  - Compile a design unit (e.g., package) before it is referenced
3. What is the scope of a `use` statement?
  - The design unit which immediately follows and all related secondary units



*This page does not contain notes.*



## Course Conclusions



**Module** **25**

Revision

1.0

Version

VHDL 9.0

Estimated Time:

- Lecture
- Lab

**cadence®**

*This page does not contain notes.*

## Summary

This course covered 85% of the language parts used 99% of the time!

This includes:

- Basic language syntax
- RTL synthesis coding guidelines
- Advanced VHDL language details
- Advanced application issues, including:
  - Synthesis coding issues
  - Building sophisticated testbenches
  - Design organization and management
- Practical use with both simulation and synthesis tools

Several months “hands-on” work will make you experienced



Over five days, the course covers about 85% of language: the parts used 99% of the time!

## Recommended Resources

- Standards
  - IEEE Std.1076 <https://www.ieee.org/>
- Advocacy groups
  - Accellera <https://www.accellera.org/>
- Cadence Support, online help and online training
  - <http://support.cadence.com/>
- Suggested reference books
  - The Designers Guide to VHDL – Peter Ashenden
  - VHDL Answers to Frequently Asked Questions – Ben Cohen

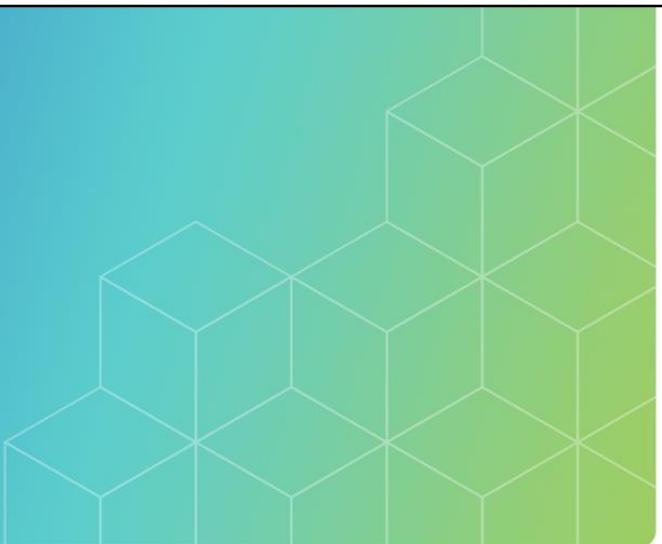
393 © Cadence Design Systems, Inc. All rights reserved.



*This page does not contain notes.*



## Next Steps



**Module** **26**

Revision **1.0**

Version **1.0**

Estimated time:

- Lecture **5 minutes**
- Lab **NA**

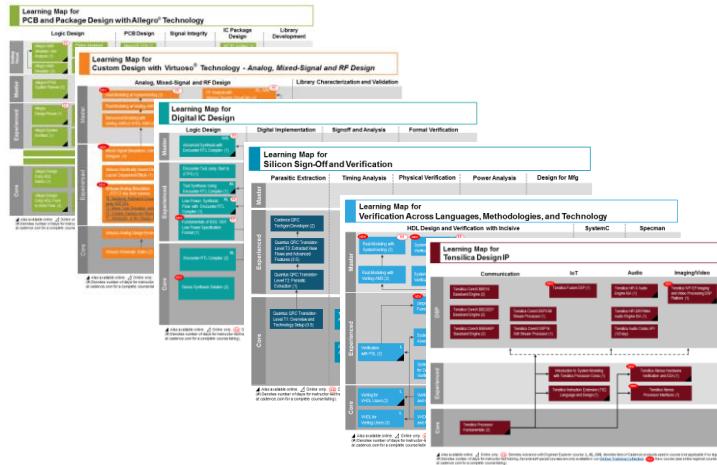
**cadence®**

*This page does not contain notes.*

## Learning Maps

Cadence® Training Services learning maps provide a comprehensive visual overview of the learning opportunities for Cadence customers.

Click [here](#) to see all our courses in each technology area and the recommended order in which to take them.



395 © Cadence Design Systems, Inc. All rights reserved.



Go here to view the learning maps:

[http://www.cadence.com/Training/Pages/learning\\_maps.aspx](http://www.cadence.com/Training/Pages/learning_maps.aspx)

## Cadence Learning and Support

The screenshot shows the Cadence Learning and Support website. At the top, there's a navigation bar with links for Cases, Tools, IP, Resources, Learning, Software, My Support, and Contribute Content. To the right are icons for notifications and user profile. Below the navigation is a search bar with a magnifying glass icon and a placeholder 'Start your search here...'. A large play button icon is overlaid on the center of the page. The background features a dark green and blue circuit board design. At the bottom, there are six categories with icons: Installation & Licensing (gear), Product Manuals (book), Training Courses (document), What's New (lightbulb), Troubleshooting Information (wrench), and Video Library (play). A banner at the bottom states: "Cadence Support now includes over 2000 product/language/methodology videos ("Training Bytes")!".

All Content  Start your search here...

View History

Installation & Licensing

*Cadence Support* now includes over 2000 product/language/methodology videos ("Training Bytes")!

396 © Cadence Design Systems, Inc. All rights reserved.



Click [here](#) to view the demo of COS.

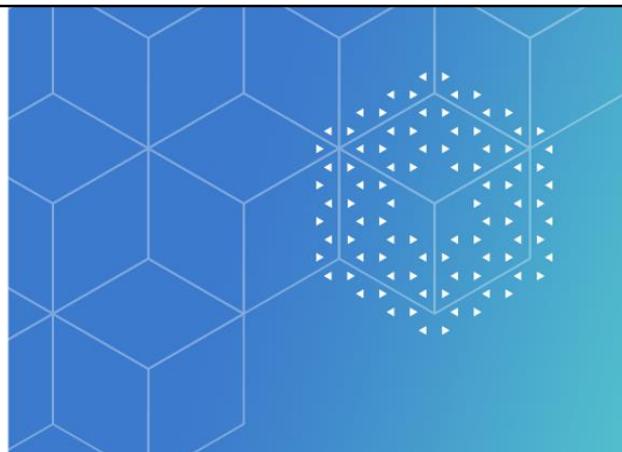
## Wrap Up

- Complete Post Assessment, if provided
- Complete the Course Evaluation
- Get a Certificate of Course Completion

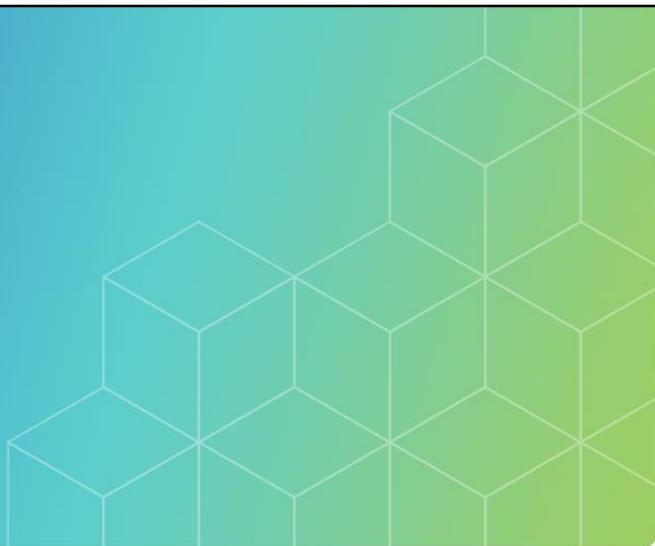
# Thank you!



*This page does not contain notes.*



VHDL200X



## Appendix

Revision

A

Version

1.0

Estimated Time:

VHDL 9.0

- Lecture
- Lab

**cadence®**

This module discusses the new features proposed for the VHDL language and describes some of the VHDL2006 enhancements.

## Appendix Objectives

In this module, you

- Describe VHDL200x and look at selected proposals for the language
- Define VHDL200x
- State the motivation behind VHDL200x
- List VHDL2006/VHDL2008 enhancements
- State longer-term proposals



*This page does not contain notes.*

## What Is VHDL200x?

A new release schedule for VHDL *and* a roadmap for future development

Two stages

- Regular updates addressing urgent requirements
  - VHDL2006, VHDL2008
- Longer-term plans covering the subsequent revisions of the language

Offering significant enhancements to the language *and* new packages

Work initially undertaken by several IEEE working groups

- Now by Accellera, based upon IEEE WG work

Milestones

- IEEE 1076-2008 (VHDL2008) standardized in 2009
- Continued work on a future release of Accellera VHDL LRM

Support for VHDL2006/2008 constructs is growing!

400 © Cadence Design Systems, Inc. All rights reserved.



VHDL200x is an ongoing effort. It is not a single standard release. It is a significant effort to enhance VHDL to meet tomorrow's design and verification needs. The VHDL2006 and VHDL2008 standards are a mixture of urgent enhancements and useful constructs that can be easily implementable now and lay the groundwork for future language enhancements.

The IEEE initially set up working groups to create the new 1076 LRM. The EDA standards body, Accellera, is now conducting this work. Significant changes were released in the 2009 standard. Required enhancements are continuously monitored for future updates. Support for all VHDL2006/2008 constructs by the tools is growing.

## Motivation for Change

VHDL93/02 met design and verification challenges for a decade, however

- Design complexity was growing very fast
- Verification complexity growing exponentially
- Increased competition from SystemVerilog, SystemC

New methodologies required

- Constrained random testing
- Assertion-Based Verification (ABV)
- Higher levels of abstraction (Object-Oriented Design)
- Transaction-based verification

Greater efficiency is required in

- Simulation runtimes
- Design/Testbench creation and maintainability

Also, the opportunity to fix outstanding issues in VHDL

- Syntax, operators, packages, constructs



VHDL has stood the test of time and has been a good solution to design and verification needs for over a decade. However, design complexity is growing very fast, verification complexity is growing exponentially, and there is tough competition from other languages such as SystemVerilog and SystemC. The new methodologies (particularly in verification) are difficult to implement using the current language standard. So in order to remain effective, the language must be enhanced to support these methodologies (randomization, Assertion-Based Verification, Object-Oriented Design, etc.). At the same time, there is an opportunity to fix some of the outstanding issues with the language to make it easier and simpler to use.

## VHDL2006: New Operators

Logical reduction operators

Combined scalar/array operations

"Matching" relational operators

- ?=, ?/=, ?<=, ?<, ?>=, ?>
- Return std\_logic values
  - Including 'U' and 'X'
- Inequality and equality allow std\_logic.'-' as a wildcard

```
-- vhdl2002
PAR <= VA(0) xor VA(1) xor VA(2);
-- vhdl2006
PAR <= xor VA;
```

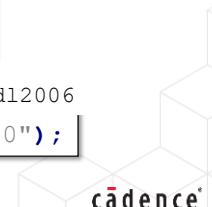
```
-- vhdl2002: VB is array, B is scalar
VA <= (VB(0) and B, VB(1) and B, VB(2) and B);
-- vhdl2006
VA <= VB and B;
```

```
ASLV := "10101010"; -- vhdl2006
MASK := "-0-0-0-0";
S := ASLV ?= MASK; -- '1'
```

```
if (ADDR = "0010") then
  VALID <= '1';
else
  VALID <= '0';
end if;
```

```
VALID <= (ADDR ?= "0010"); -- vhdl2006
```

402 © Cadence Design Systems, Inc. All rights reserved.



Let us look at the new operators added in VHDL2006. The bitwise logical operations in VHDL2002 were replaced with logical reduction operators in VHDL2006.

As shown on the slide, scalar and array operations were combined in VHDL2006.

Reduction and combined scalar/array operators are straightforward.

The multi-line code for matching relational operators in VHDL2002 was replaced with a single assignment statement in VHDL2006.

“Matching” relational operators serve several purposes:

- To simplify conditional code by allowing relational operators to return std\_logic values for direct assignment to an object instead of Booleans.
- To allow (in)equality operators to interpret std\_logic don’t care value '-' as a wildcard.

Matching relational operators can also return 'U' or 'X' std\_logic values. E.g., for the matching equality operator ?=, comparison to '-' returns '1' (true); otherwise, comparison to 'U' returns 'U', otherwise, comparison to 'X' or 'Z' returns 'X' (see Language Reference manual for full truth tables for these operators).

## VHDL2006: Conditional Statements

Simplified conditional expressions

- std\_logic '1' = BOOLEAN TRUE

Conditional and selected signal assignments  
are concurrent statements only in VHDL2002

In VHDL2006, they are allowed as sequential  
statements

- Can also assign to variables

```
vhdl2002
if (CS1 = '1') and (CS2 = '0') and (ADDR = 10) then
  ...
...
```

```
vhdl2006
if (CS1 and not CS2 and ADDR = 10) then
  ...
...
```

```
vhdl2006
process (all)
  variable LOCALVAR : std_logic;
begin
  OP1 <= '0' when STATE = INITIAL else '1';

  with STATE select
    OP2 <= '0' when INITIAL,
    '1' when GO | STOP,
    'X' when others;

  LOCALVAR := '1' when STATE = GO else '0';
  ...
end process;
```

403 © Cadence Design Systems, Inc. All rights reserved.

Conditional statements are simplified in VHDL2006 than in the previous version. The simplified conditional expressions allow the std\_logic '1' value to be interpreted as boolean TRUE in a complex conditional expression.

The VHDL2002 concurrent conditional and selected assignments are now allowed in VHDL2006 to be sequential statements (i.e., to be placed in a process or subprogram). As such, both statements can now be used to assign variables as well as signals.

## VHDL2006: Case Enhancements

New "matching case" construct

- `case? ... end case?;`

Allows `std_logic.'-'` as wildcard value in case choice expressions

Restrictions

- Only allowed in choice expression
- No overlapping between values
- bit, boolean, `std_logic` types or vectors only

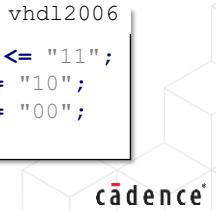
Predefined operators/functions allowed in case choice or select expressions

- Locally static expressions
- No overlapping between values

```
vhdl2002
case IP is
  when "100" | "101" | "110" | "111" => OP <= "11";
  when "010" | "011" => OP <= "10";
  when "001" | "000" => OP <= "01";
  when others => OP <= "00";
end case;
```

```
vhdl2006
case? IP is
  when "1--" => OP <= "11";
  when "01-" => OP <= "10";
  when "00-" => OP <= "01";
  when others => OP <= "00";
end case?;
```

```
vhdl2006
case A xor B is
  when CONST1 and "10" => OP <= "11";
  when "0000" => OP <= "10";
  when others => OP <= "00";
end case;
```



A VHDL2006 `case?` statement can use `std_logic` don't care values (' - ') in case choice expressions as a true wildcard, matching any other `std_logic` value. This can greatly simplify case statements, particularly for priority encoder structures, by removing the requirement for the case choice expression to explicitly list every value for a branch.

However, the standard VHDL case restrictions still apply, specifically that there can be no overlapping values between case choices and that every value of the case select expression should be covered in the case choices.

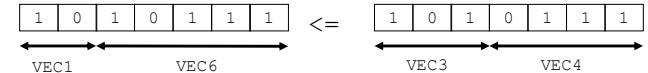
In VHDL2006, there is also more freedom for constructing values in case select expressions. Predefined operators and functions can be used. Again the usual case rules apply – the case select expressions must be locally static (i.e., calculable by only compiling the design unit containing the case statement) and cannot overlap.

## VHDL2006: Assignment Enhancements

Slices allowed in array aggregates

Aggregates are allowed on both sides of the assignment in the same statement

```
signal VEC1: signed(1 downto 0);
signal VEC3: signed(2 downto 0);
signal VEC4: signed(3 downto 0);
signal VEC6: signed(5 downto 0);
...
(VEC1, VEC6) <= (VEC3, VEC4);
```



```
-- Declarations for both examples
signal OP1, OP2, RES : unsigned(7 downto 0);
signal CARRY : std_logic;
```

```
process (all)
  variable TEMP : unsigned(8 downto 0);
begin
  TEMP := ('0' and OP1) + ('0' and OP2);
  (CARRY, RES) <= TEMP;
end process;
```

vhdl2002

```
process (all)
begin
  (CARRY, RES) <= ('0' and OP1) + ('0' and OP2);
end process;
```

vhdl2006

405 © Cadence Design Systems, Inc. All rights reserved.

VHDL2006 allows the slice of an array to be included in an aggregate expression and also allows aggregates to be on both sides of the same assignment.

VHDL2002 prohibits slices in aggregates and also prohibits aggregates on both sides of an array assignment.

## VHDL2006: Sized and Signed Literals

Array literals specified using hex or octal

- A single hex literal is always of length 4
- Hence, hex assignment made to arrays with length equals multiple of four

In VHDL2006, sized literals added before the base specifier

- Truncate or pad (by default with 0) literal to a set number of bits
  - Match the length of the target array
- $6X^{\prime}f\prime = "001111"$
- Works with X and Z values too
- $6X^{\prime}XX\prime = "XXXXXX"$

Decimal D notation

- $6D^{\prime}27\prime = "011011"$

Signed/Unsigned literals

- Unsigned qualifier U pads literals with 0 (default)
- $6UX^{\prime}C\prime = "001100"$
- Signed qualifier S pads literals with Most Significant (Sign) Bit
- $6SX^{\prime}C\prime = "111100"$

```
vhdl2002
process
  variable myvar : unsigned(5 downto 0);
begin
  myvar := X"27" -- length mismatch 6 <= 8
...

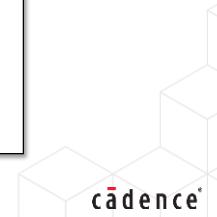
```

```
vhdl2006
process
  variable myvar : unsigned(5 downto 0);
begin
  myvar := 6X"27" -- length match 6 <= 6
...

```

X"7"	= "0111"
X"ZZ"	= "ZZZZZZZZ"
3X"7"	= "111"
9X"F"	= "000001111"
8D"17"	= "00010001"
9SX"F"	= "111111111"

406 © Cadence Design Systems, Inc. All rights reserved.



In VHDL2002, array literals can be expressed using hex or octal notation. However, a single hex literal is always 4 bits. Rules requiring array sizes to be the same on both sides of an assignment limit the use of literal hex assignments to arrays with a length equal to a multiple of 4. On the slide, we see that the 6-bit variable myvar cannot be assigned with an 8-bit hexadecimal value 27.

In VHDL2006, an optional size integer can be added before the hex or octal base specifier. The literal value is then truncated or padded (by default with 0 in the MSB) to the length specified by the size (which must match the length of the target array). The literal value can include X and Z values. Now hex literals can be used to assign arrays of any size. This allows the 6-bit variable myvar to be assigned with an 8-bit hexadecimal value 27 by adding sized literal six in front of the hexadecimal value.

In addition, a decimal specifier is also defined for VHDL2006 literals.

The default padding of literal values with 0 causes problems for signed (2's complement) literals. Therefore, an additional signed specifier can be placed between the size and base specifier to indicate that any padding of the literal value must be via sign extension rather than with 0. Unsigned qualifier 'U' pads literals with zeros, signed qualifier S pads literals with MSB, sign preserved.

## VHDL2006: Verification Features

`to_string` defined for every type

```
report "VEC : " & to_string(VEC);
```

- Also, `to_hstring`, `to_ostring` for bit array types

```
A <= <<signal .top.ul.my_sig : std_logic>>;  
alias hiersig =  
<<signal .top.ul.my_sig : std_logic>>;
```

Hierarchical references

- Syntax
- `<<class pathname : type>>`
- Class is one of signal, constant or variable

```
process  
begin  
  if INJECT_ERROR = '1' then  
    hiersig <= force "1001";  
  end if;  
  wait until (CYCLE_OVER = '1');  
  hiersig <= release;  
  ...
```

force and release statements

- A means to override existing values in VHDL code

Procedure `hread` and `hwrite` added to `textio`

```
procedure HREAD  
(L: inout LINE; VALUE: out BIT_VECTOR);
```

407 © Cadence Design Systems, Inc. All rights reserved.



`to_string` is an implicitly defined conversion function for all pre-and user-defined types. Note that, unlike the `'image` attribute, `to_string` works with array as well as scalar types.

`to_hstring` and `to_ostring` create string values in hex or octal form, respectively, for bit array types only, e.g., for `std_logic_vector`, `signed`, `unsigned` types.

Hierarchical references are not allowed in VHDL2002. Every simulator has a proprietary solution to this, but this makes code non-portable, and workarounds to make the code portable are not easy.

Hierarchical references can be aliased for ease of use:

```
alias hiersig = <<signal .top.ul.my_sig : std_logic>>;
```

If the hierarchical signal is a vector, the type clause of the reference does not have to be constrained:

```
alias hiervec = <<signal .top.ul.my_vec : signed>>;
```

In the hierarchical reference pathname, a `^` character navigates one level up in the design hierarchy.

Hex `read` and `hex write` procedure calls are available from the proprietary `std_logic_textio` package in VHDL2002. In VHDL2006, they are added to the `std_textio` package.

## VHDL2006: Stop and Finish

Stopping simulation in VHDL2002

- Requires inelegant use of assertion of severity failure

- With unconditional waits used as backups

`stop` and `finish` procedure calls

- In package `env` from library `std`
  - Must be referenced!
- More elegant means of ending the simulation
- `stop` – like a breakpoint
  - Simulation can continue
- `finish` – simulator will exit
- Optional status integer may be useful for simulation scripts

```
package ENV is
  procedure STOP ( STATUS: INTEGER );
  procedure FINISH ( STATUS: INTEGER );
  ...
end package ENV;
```

```
use std.env.all;
...
if SIM_END = '1' then
  stop(0);
end if;
...
```

408 © Cadence Design Systems, Inc. All rights reserved.



Stopping simulation runs in VHDL2002 usually requires inelegant use of assertions of severity failure, with unconditional waits used as a backup.

In VHDL2006, two special procedures, `stop` and `finish`, are defined to control simulation. `Stop` and `finish` are declared in an additional package named `env` from the library `std`. This package must be explicitly referenced to use the procedures. This is for backward compatibility with designs that might already use `finish`/`stop` as object names.

`Stop` acts as a breakpoint by pausing simulation when executed. The simulation then reverts to user control and can be continued or terminated as required. `Finish` terminates the simulation and forces the simulator to exit. Continuation is not possible.

Both procedures have optional status integers, which may be interrogated by simulation scripts (implementation dependent).

## VHDL2006: Unconstrained Arrays

Current VHDL restrictions:

- Array types with only a single unconstrained dimension
- No unconstrained record array elements

VHDL2006 adds:

- Unconstrained, multi-dimensional array types
- Unconstrained record types

A partial definition of sizes allowed:

- Dimensions can be defined in different places

```
type UAA is array (natural range <>) of unsigned;
signal MEM1 : UAA(D downto 0)(W downto 0);
```

```
type TWOVEC is record
    ONE, TWO : unsigned;
end record;

signal V8_4 :
    TWOVEC(ONE(15 downto 0), TWO(7 downto 0));
```

```
type UAA is
    array(natural range <>) of unsigned;
type FIFO_T is record
    DATA      : UAA;
    EMPTY, FULL : boolean;
    INDEX     : natural;
end record;

signal FIFO1 :
    FIFO_T(DATA(15 downto 0)(7 downto 0));
```

409 © Cadence Design Systems, Inc. All rights reserved.



Multidimensional arrays can only have one unconstrained dimension. This can be limiting because we cannot declare a type that is a generic FIFO with variable depth and width. This is inconvenient because we can write code that will operate on types independent of array dimensions using array attributes. VHDL2006 allows unconstrained multi-dimensional arrays and records. Obviously, the size must be defined at some point before simulation. VHDL2006 allows the partial definition of sizes, i.e., we can give a final size for each dimension in different places.

## VHDL2006: Automatic Sensitivity List

Automatic sensitivity list for combinational processes

- Using keyword `all`

```
vhdl2002
P1: process (SEL, A, B, C, D)
begin
  case SEL is
    when "00" => OP <= A;
    when "01" => OP <= B;
    when "10" => OP <= C;
    when others => OP <= D;
  end case;
end process P1;
```

```
vhdl2006
P1: process (all)
begin
  case SEL is
    when "00" => OP <= A;
    when "01" => OP <= B;
    when "10" => OP <= C;
    when others => OP <= D;
  end case;
end process P1;
```

410 © Cadence Design Systems, Inc. All rights reserved.



The use of keyword `all` as a replacement for a process sensitivity list makes the process sensitive to all signals read.

This feature ensures greater compatibility between the simulator and the synthesis tool.

If the sensitivity list were incomplete, there would be a mismatch in RTL and gate-level simulation because the simulator will implement behavior as defined by the LRM. Still, a synthesized tool may assume completeness (with a warning).

## VHDL2006: Assertions

VHDL2006 includes the IEEE 1850 PSL LRM, by reference

- Property Specification Language for creating assertions

Implications

- PSL units become VHDL primary design units
- PSL declarations are allowed in any declarative VHDL region
  - Properties, sequences, default clocks, etc.
- PSL directives are allowed in any statement region
  - assert, cover, assume
- PSL declarations and directives are uncommented and omit the PSL keyword
- VHDL2006 can read the value of an output port!

```
architecture TEST of ABV is
  property CLIMIT is always
    (unsigned(ICOUNT) <= 10) @falling_edge(CLK);
begin
  A1 : assert CLIMIT;
  ...

```

411 © Cadence Design Systems, Inc. All rights reserved.



VHDL2006 includes PSL in the VHDL LRM by reference. Any tool compliant with 1076 must also be compliant with the PSL LRM 1850. PSL units become VHDL primary design units, and PSL declarations such as properties, sequences, and default clocks are allowed in any declarative VHDL region. PSL directives such as assert, cover, and assume are allowed in any statement region

PSL assertions, including cover and assume directives, can now be directly embedded into VHDL code with the need for commenting or the PSL keyword.

Only the VHDL flavor of PSL is supported.

Future standard revisions may allow VHDL to directly refer to PSL endpoints, sequences, etc., without that object being declared in VHDL. The way in which this will be achieved is not yet finalized due to backward compatibility issues. The VHPI (standardized C interface) may be the mechanism used.

## VHDL2006: Package and Type Generics

Packages and subprograms can contain generic clauses

Types can be passed via generics

- New generic qualifier `type`

Instantiated as a primary design unit

- Passed via the generic map
- Separate, named package/subprogram created via a new statement
- Type can also be instantiated in any declarative region where a type is allowed

```
genstack_p.vhd
package GENERIC_STACK is
  generic (type PAYLOAD_T);

  type ELEMENT;
  type POINTER is access ELEMENT;

  type ELEMENT is record
    PAYLOAD : PAYLOAD_T;
    NEXT_DATA : POINTER;
  end record ELEMENT;
  ...
  
```

```
t1stack_p.vhd
use work.TYPE1_DEC.all;

package TYPE1_STACK is new work.GENERIC_STACK
  generic map (PAYLOAD_T => TYPE1);
  
```

```
mydesign_ea.vhd
use work.TYPE1_STACK.all;
...
TEMP := new TYPE1_STACK.ELEMENT;
  
```

412 © Cadence Design Systems, Inc. All rights reserved.

VHDL2006 adds generic clauses to both packages and subprograms, creating *generic packages* and *generic subprograms*. The generic clause allows type declarations to be passed into the package or subprogram to create, for example, a generic stack structure that can be used with any data type. Generic packages and subprograms cannot be used directly – they must be instantiated with a generic map to define values for every generic. The instantiation uses the keyword `new` to define a new, separate declared object with a unique name.

Generic packages can be instantiated as a primary design unit, as in the example above. In order to use local types and objects, they can also be instantiated in any declarative region – i.e., wherever a type can be defined.

## VHDL2006: Context

New primary design unit

- Effectively, a bundle of library and package references
- Declared and compiled separately
- Referenced via keyword context

This allows for more effective management of packages in large design hierarchies

```
context PROJECT1_CONTEXT is
    use STD.textio.all;
    library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
end
```

```
library MYLIB;
context MYLIB.PROJECT1_CONTEXT;

entity ONE is
    ...
```

413 © Cadence Design Systems, Inc. All rights reserved.



VHDL2006 introduced the context feature. Context allows several libraries and package references to be bundled into a single, new primary design unit (declared and compiled separately) and then referenced where required using the keyword context.

This allows for more effective management of packages in large design hierarchies.

## VHDL2006: Fixed-Point Math Package

Synthesisable fixed-point maths package

```
package fixed_pkg is
    type ufixed is array (INTEGER range <>) of STD_LOGIC; -- unsigned
    type sfixed is array (INTEGER range <>) of STD_LOGIC; -- signed
    ...
    use ieee.fixed_pkg.all;
    ...
    signal US_ARRAY: ufixed(4 downto -3);
    signal S_ARRAY: sfixed(3 downto -4);
```

US_ARRAY								
Power	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>-1</sup>	2 <sup>-2</sup>	2 <sup>-3</sup>
Index	4	3	2	1	0	-1	-2	-3
Value	1	0	0	0	1	•	1	1

Implied Binary Point

Value shown = 17.75

S_ARRAY								
Power	-2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>-1</sup>	2 <sup>-2</sup>	2 <sup>-3</sup>	2 <sup>-4</sup>
Index	3	2	1	0	-1	-2	-3	-4
Value	1	0	1	0	•	0	1	0

Implied Binary Point

Value shown = -5.25



Typically, users have previously defined this kind of functionality with their own packages. VHDL2006 provides a standard implementation that is intended to be synthesizable. A binary point is implied between index 0 and -1. The arithmetic, relational and shift operations have been overloaded for these types. Conversion functions are also defined in this package.

A different version of the fixed-point math package is defined in the VHDL2006 standard named `fixed_generic_pkg` and uses package generics for more user control over rounding and overflow in arithmetic operations.

## VHDL2006: Floating-Point Math Package

Float32 – 32-bit IEEE 754 single-precision floating-point

Float64 – 64-bit IEEE 754 double-precision floating-point

Float128 – 128-bit IEEE 854 extended precision floating-point

```
use ieee.float_pkg.all;
...
subtype float32 is float (8 downto -23);
...
signal MY_FP_32: float32;
```

Index	8	7	6	5	4	3	2	1	0	-1	-2	-3	.....	-21	-22	-23	MY_FP_32
Purpose	S	E	E	E	E	E	E	E	E	F	F	F	.....	F	F	F	

S – Sign

E – Exponent – Biased at 127

F – Fraction – Biased at  $2^{24}$

Number value is given by =

$S * (2^{** (E-127)}) * (1.0 + (F / 2^{24}))$



Implied '.' between indices 0 and -1



Floating-point operations are more complex, requiring more hardware (by approximately a factor of three) than fixed-point operations. Before using Floating-point types and operations, ensure that you actually require the dynamic range and precision that floating point gives. In many cases, a fixed-point is an adequate and simpler solution.

The arithmetic, relational and shift operations have been overloaded for these types. Conversion functions are also defined in this package. Floating-point numbers can be 32-bit, 64-bit, or 128-bit depending on the range and precision. The example on the slide shows a 32-bit floating-point number with 8-bits to depict the whole part, 24-bit to depict the decimal part, and 1-bit to represent the sign. The numeric value of this representation is computed using the formula shown on the slide.

## VHDL200x: Verification Proposals

### Verification data structures

- Dynamic and associative arrays
- Predefined queues, FIFOs, stacks
- Mailboxes
  - Multi-type FIFOs with the synchronization control
- Semaphores
  - Key-based access/arbitration control

### Randomization

- Random generation of stimulus
- Constraints to control values generated
- Weighting to control the probability of values

### Object-Oriented/Class-based data structures

- With randomization

416 © Cadence Design Systems, Inc. All rights reserved.



Directed-Random tests are a vital tool in the Verification engineer's toolbox and need to be supported. Many proposals are made to enable this, such as verification data structures such as dynamic and associative arrays, predefined queues, FIFOs and stacks, and mailboxes with multi-type FIFOs for synchronization control and semaphores. Proposals are made for constructs that enable random generation of stimulus, constraints to control the values generated, and weighting to control the probability of values. Object-oriented/class-based structures are another enhancement that can be added to attain a higher level of abstraction. These are some of the proposed features to be added to future releases.

## Object Oriented Type Proposal

Based on protected types:

- VHDL2002 shared variables

Type declaration includes open properties and method definitions

Type body for method declarations and protected properties

Constructors can be explicitly defined with new

Other class types can inherit

Class types can be used:

- Statically as a type of a declared object
- Dynamically through assignment to a pointer

```
type CFIFO is class
    variable DATA : FIFO_ARR; -- open
    new OOFIFO (INIT : natural); -- constructor
    procedure PUSH (INT_IN : in integer);
end class CFIFO;

type CFIFO is class body
    variable EMPTY : boolean; -- protected
    variable FULL : boolean; -- protected
    procedure PUSH (INT_IN : in integer) is
        begin
        ...
    end class body CFIFO;

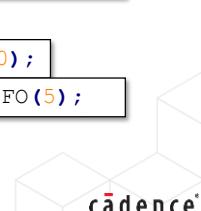
-- inherit new type from CFIFO
type SUBFIFO is new CFIFO class
    ...

```

```
variable FIFO1 : CFIFO(0);
```

```
pointer := new CFIFO(5);
```

417 © Cadence Design Systems, Inc. All rights reserved.



Object-Oriented/class-based designs are a powerful method of developing complex, reusable data sets for verification.

VHDL2002 already has a similar mechanism for handling shared variables. The current proposal is to extend this mechanism to allow true object-oriented design, with inheritance, polymorphism, dynamic class instantiation, static class members, etc. The type construct allows for method definition and open properties. The type body allows for method declaration and protected properties. The example on the slide defines a class CFIFO containing variable DATA, procedure PUSH, and the new constructor. Two new variables, EMPTY and FULL, are added to the class body. Class CFIFO is constructed by calling the new constructor and passing an integer to it; the variable pointer contains the handle. Other class types can inherit from this class. Here, class SUBFIFO inherits from class CFIFO, as shown. Class types can be used statically as a type of a declared object or dynamically through assignment to a pointer.

## Summary

- VHDL2006 and VHDL2008 are IEEE-approved standards
- Some vendors are already supporting most features
- More support will only come with clear, prioritized customer requests
- Still plenty of work to do!



*This page does not contain notes.*



cadence®

© Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, and the other Cadence marks found at <https://www.cadence.com/go/trademarks> are trademarks or registered trademarks of Cadence Design Systems, Inc. Accellera and SystemC are trademarks of Accellera Systems Initiative Inc. All Arm products are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All MIPI specifications are registered trademarks or service marks owned by MIPI Alliance. All PCI-SIG specifications are registered trademarks or trademarks of PCI-SIG. All other trademarks are the property of their respective owners.

*This page does not contain notes.*