Seminar

Designing with VHDL

Workbook

Lecturer

Nir Balulu

Designing with VHDL

lang-vhdl-2019.1-wkbp-rev1

Designing with VHDL 2019.1

© Copyright 2019 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, UltraScale, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.

DISCLAIMER

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps. All other trademarks are the property of their respective owners.

Table of Contents

Course Agenda **1-1**

Course Objectives 1-1

Course Parameters 1-2

Agenda Day One 1-3

Agenda Day Two 1-3

Agenda Day Three 1-4

Review of FPGA Architecture (1) 1-4

Review of FPGA Architecture (2) 1-6

Review of Xilinx Tool Flow 1-6

Waking Up 1-8

Appendix 1-9

Documentation for VHDL 1-9

Latest Product Information 1-10

Introduction to VHDL **2-1**

Objectives 2-1

Historical Background of VHDL 2-2

VHDL History and Usage 2-3

VHDL as a Language (1) 2-4

VHDL as a Language (2) 2-4

Compilation 2-5

Compilation Path 2-6

Coding Styles 2-6

HDL Description Levels 2-7

VHDL and FPGAs 2-8

© Copyright 2019 Xilinx

Setting up the Vivado Design Suite to Use VHDL-2008 2-9
Summary 2-9

VHDL Design Units 3-1

Objectives 3-1
VHDL Design Units 3-1
Typical VHDL "Module" 3-2
Typical VHDL "Module" Example 3-3
Library Declarations 3-4
Package Declarations 3-5
Entity Declaration 3-6
Port Declarations 3-7
Architecture Declarations 3-8
The "Entity" 3-9
Configuration Declarations 3-10
Module Partitioning Guidelines 3-11
Summary 3-12

VHDL Objects, Keywords, Identifiers 4-1

Objectives 4-1
Keyword and Identifier 4-2
Identifiers Exercise 4-3
Expressions and Literals 4-4
Data Objects in VHDL 4-5
VHDL Object Classes 4-6
VHDL Comments 4-6
Five Levels of Commenting 4-7
Header Comment Example 4-8
Apply Your Knowledge 4-8
Summary 4-9

© Copyright 2019 Xilinx

Scalar Data Types **5-1**

Objectives 5-1

Data Types 5-2

Scalar Data Types 5-3

bit and boolean 5-4

std_ulogic and std_logic 5-5

std_logic versus std_ulogic 5-6

Signal Resolution 5-7

integer and real 5-8

What Can and Cannot be Done with Integers and Reals 5-9

Data Type Conversion 5-10

VHDL Type Casting and Conversion Functions 5-11

std_logic_vector to/from integer 5-12

Types and Subtypes 5-13

Subtypes 5-14

Characters and Strings 5-15

Physical 5-16

Enumerated Types 5-17

Enumerated Types – Caution! 5-18

Summary 5-19

Composite Data Types **6-1**

Objectives 6-1

Composite Data Types 6-2

STD_LOGIC_VECTOR and SIGNED/UNSIGNED 6-3

What CAN and CANNOT Be Done with std_logic 6-4

Arrays 6-5

Array Assignments 6-6

Array Assignment Notation 6-7

© Copyright 2019 Xilinx

Array Slices 6-8
Signals and Ports with Example 6-9
Records 6-10
Strings 6-11
Array Aggregates 6-12
Record Aggregates 6-13
Creating Pseudo 2-D Arrays 6-13
Creating Arrays of Records 6-14
Assigning Values to Arrays 6-15
Higher Order Arrays 6-16
Higher Order Array Example 6-17
Apply Your Knowledge 6-20
One Possible Solution 6-21
Summary 6-21

VHDL Operators 7-1

Objectives 7-1
Operators in VHDL 7-2
Logical Operators 7-3
Logical Operations on Arrays 7-4
Relational Operators 7-5
Arithmetic Operators 7-6
Arithmetic Operations on Arrays 7-7
Array Arithmetic 7-8
Miscellaneous Operators - abs, **, rem 7-8
Shift Operators 7-9
Shift Operator Examples 7-10
Concatenation 7-11
Shift Register Using Concatenation 7-12

© Copyright 2019 Xilinx

Grouping Operators 7-12
New Conditional Operator 7-13
Operator Precedence 7-13
Summary 7-14

Concurrency in VHDL 8-1

Objectives 8-1
Concurrency and Delta Cycles 8-1
What is Concurrency? 8-2
Signals 8-4
Signal Assignment 8-5
Constants 8-6
Transactions 8-7
Events 8-8
Transaction versus Event 8-9
Delta Cycles 8-10
Scheduling Events 8-10
Scheduling Events Example 8-11
Modeling Concurrency 8-11
Apply Your Knowledge 8-12
Summary 8-14

Concurrent Assignments 9-1

Objectives 9-1
Types of Concurrent Constructs 9-2
Why Conditional Assignments? 9-3
"when" Form 9-4
"with/select" Form 9-6
About Instantiation 9-7
Creating the Component Declaration 9-8

© Copyright 2019 Xilinx

Instantiation Port Mappings 9-9
Instantiation Example 9-10
Timed Conditional Assignments 9-11
Force and Release 9-12
Concurrent Looping 9-12
Conditional Generate Example 9-13
Example of a Variable-Width Counter Using Generics 9-14
Generate Statement – Conditional Form 9-15
Generate Statement – Loop Form 9-16
Generics 9-17
Using Generics 9-18
Loop Generate Example 9-19
Top-Level Generics 9-19
Summary 9-20

Processes and Variables 10-1

Objectives 10-1
Processes are Concurrent 10-2
VHDL Connectivity Model 10-3
Process 10-3
Process Syntax Basics 10-5
Process Sensitivity List 10-6
Process Sensitivity List Examples 10-7
Process Labels 10-8
Synchronous Processes 10-9
Inside Process versus Outside Process 10-10
Variables 10-11
Variable Scope 10-12
Using Variables 10-13

© Copyright 2019 Xilinx

Variables in Clocked Processes 10-14
Safe Variable Usage in a Clocked Process 10-15
Variable Usage Guidelines 10-16
Variables versus Signals 10-17
Apply Your Knowledge 10-19
Summary 10-20

Conditional Statements in VHDL: if/else, case 11-1

Objectives 11-1
Process Control Structures 11-2
if/else Statements 11-2
Process if versus Concurrent if 11-3
if/elsif Example and Rules 11-4
Clock Enable Inferred 11-5
Avoiding CE Inference 11-6
Overlapping Conditions 11-7
Selected Signal Assignment 11-8
Case Statement 11-9
Case Statement Ranges 11-10
Case Examples and Rules 11-11
Default Assignments 11-12
Apply Your Knowledge 11-13
Answer 11-14
Specifying Ranges – Apply Your Knowledge 11-15
Specifying Ranges 11-15
Summary 11-16

Sequential Looping Statements 12-1

Objectives 12-1
Looping Structures – Hardware versus Software 12-2

© Copyright 2019 Xilinx

Sequential Looping 12-2
Loop Statements 12-3
Using Loop Statements 12-4
Deliberate Infinite Loops 12-5
Accidental Infinite Loops 12-6
Example of Accidental Infinite Loop 12-6
for loop 12-7
while Loop 12-9
Nested Loops 12-10
Apply Your Knowledge 12-10
Summary 12-11

Delays in VHDL: Wait Statements 13-1

Objectives 13-1
Delays in VHDL 13-1
Inertial Delay 13-2
Transport Delay 13-2
General Form of the Wait Statement 13-3
Suspending Processes 13-4
Synthesized Wait Statement Example 13-5
Wait 13-5
Wait For 13-6
Wait On 13-7
Wait Until 13-8
Summary 13-9

Introduction to the VHDL Testbench 14-1

Objectives 14-1
What is a Testbench? 14-1
Testbench 14-2

© Copyright 2019 Xilinx

Application of a Testbench 14-2
Simulation I/O Flexibility 14-3
Testbench Consistency 14-4
Behavioral Coding 14-5
Verification at Each Level 14-6
Apply Your Knowledge 14-6
Components of a Testbench 14-7
Instantiating DUT in Testbench 14-8
Creating Clock Signals 14-9
Stimulus Generation 14-10
Other Clock Creation Approaches 14-11
Initializing Signals 14-12
Apply Your Knowledge 14-12
Summary 14-13

VHDL Assert Statements 15-1

Objectives 15-1
Assertions 15-1
Assertion Syntax 15-3
Other Uses of Assertions 15-3
Report Clause 15-4
Uses of Severity 15-4
Assertion Statements 15-5
Other Validation Techniques 15-6
Example of a Self-Checking Testbench 15-7
Summary 15-8

Attributes 16-1

Objectives 16-1
Attributes 16-2

© Copyright 2019 Xilinx

Using "Tic" (Predefined) Attributes 16-3
Range Attributes 16-4
Commonly Used Attributes 16-4
Examples of Some Predefined Attributes 16-5
Typical Uses of Attributes 16-6
Revisiting Multiple Dimension Arrays with Attributes 16-6
Signal Attributes (1) 16-7
Signal Attributes (2) 16-7
Signal Attributes (3) 16-8
Common Subprogram Usage 16-8
User-Defined Attributes 16-9
Commonly Used User (Vendor) Attributes 16-10
Apply Your Knowledge 16-11
Answer 16-12
Summary 16-12

VHDL Subprograms 17-1

Objectives 17-1
Software versus Hardware 17-2
What are Subprograms? 17-3
Using Subprograms 17-4
Functions versus Procedures 17-4
Apply Your Knowledge 17-5
Summary 17-5

VHDL Functions 18-1

Objectives 18-1
Function Overview 18-1
Function Structure 18-2
Input Parameter List 18-2

© Copyright 2019 Xilinx

Passing Parameters 18-3
Function Example 18-3
Logic Created 18-4
Math and Conversion Functions 18-4
Pure and Impure Function 18-5
Function Overloading 18-6
Using an Overloaded Operator 18-7
Building an Overloading Function 18-7
Example of Building an Overloaded Function 18-8
Apply Your Knowledge 18-9
Answer 18-10
Summary 18-10

VHDL Procedures 19-1

Objectives 19-1
Procedure Overview 19-1
Procedure Structure 19-2
Procedure Parameters 19-3
Passing Parameters 19-3
Associating Parameters 19-4
Parameter Classes 19-4
Default Parameter Class 19-5
Default Parameter Class 19-5
Example Procedure 19-6
Explicit Input Parameter Class 19-6
Defining Output Parameters 19-7
Declaring a Procedure in an Architecture 19-8
Calling the Procedure 19-9
Call with Explicit Signal Class 19-10

© Copyright 2019 Xilinx

Summary 19-10

VHDL Libraries and Packages 20-1

Objectives 20-1
Libraries 20-2
Library Structure 20-3
xil_defaultlib Library 20-4
Creating a Library in the Vivado IDE 20-5
Developing Packages in Libraries 20-6
Accessing the Library and its Packages 20-7
Packages 20-8
Syntax 20-8
When to Create a Package 20-9
Creating a Package 20-10
Declaring a Procedure or Function in a Package 20-11
Calling the "Packaged" Procedure 20-12
VHDL Predefined Packages 20-13
IEEE Predefined Packages 20-13
Using Packages 20-14
Using Components with the Same Name in Packages 20-15
Summary 20-16

Interacting with the Simulation 21-1

Objectives 21-1
Verification Plan 21-2
Types of Verification 21-3
Interfacing with the Simulation Environment 21-3
Simulation I/O Flexibility 21-4
Text I/O in VHDL 21-4
Utility of Text I/O in the Real World 21-5

© Copyright 2019 Xilinx

TextIO and Its Basic Console Constructs 21-6
File Access 21-7
Reading Data from a File 21-8
Writing Data to a File 21-9
Writing Data to the Console 21-10
Formatting the Output 21-11
Apply Your Knowledge 21-12
Summary 21-13

Finite State Machine Overview 22-1

Objectives 22-1
FSM Overview 22-2
FSM Types 22-2
FSM State Diagram 22-3
Encoding Methods (1) 22-4
Encoding Methods (2) 22-4
Enumerated Data Types 22-5
Using Constants 22-6
Using Enumerated Data Types 22-7
Unreachable States 22-8
Summary 22-8

Mealy Finite State Machine 23-1

Objectives 23-1
Mealy Machine 23-1
Mealy Machine State Diagram 23-2
Assigning Mealy Outputs 23-3
Registered Mealy Outputs 23-4

© Copyright 2019 Xilinx

Moore Finite State Machine **24-1**

Objectives 24-1

Moore Machine 24-1

Moore Machine State Diagram 24-2

Assigning Moore Outputs 24-2

Registered Moore Outputs 24-3

FSM Coding Guidelines **25-1**

Objectives 25-1

FSM Coding 25-2

Process Blocks 25-3

Initialization 25-3

Single-Process Block 25-4

Single-Process FSM 25-5

Multi-Process FSM 25-6

FSM Outputs 25-7

Concurrent Operations 25-8

Modelling Next-State Logic 25-9

FSM Coding Example – UART Receiver Control 25-10

Diagram 25-11

genNextState (1) 25-12

genNextState (2) 25-12

oversample_counter (1) 25-13

oversample_counter (2) 25-13

bitTracker 25-14

outCtrl 25-14

ferrGen 25-15

Apply Your Knowledge 25-15

Summary 25-16

© Copyright 2019 Xilinx

Vivado Simulator and Race Conditions in VHDL **26-1**

Objectives 26-1

Vivado Simulator 26-2

Accessing the Vivado Simulator in the Vivado IDE 26-3

Simulation Settings 26-4

Running the Vivado Simulator 26-5

Selecting Signals 26-5

Simulator Simulation Controls 26-6

Simulator View Controls 26-6

Debugging with Vivado Simulator 26-7

What is a Race Condition? 26-8

Race Conditions in VHDL 26-8

Race Condition in VHDL Example 26-9

Steps to Avoid Race Conditions 26-9

Summary 26-10

Writing a Good Testbench **27-1**

Objectives 27-1

Review of the Testbench 27-1

Principle 27-2

Top Level 27-3

Data Generator 27-3

Data Formatter 27-4

Delivery Delay 27-4

Data Checker 27-5

Unit Under Test 27-5

Overview of UART_led_tb.vhd 27-6

Understanding the Delivery Delay Example (1) 27-7

Understanding the Delivery Delay Example (2) 27-7

© Copyright 2019 Xilinx

Understanding the Delivery Delay Example (3) 27-8
Understanding the Delivery Delay Example (4) 27-9
Summary 27-9

Targeting Xilinx FPGAs 28-1

Objectives שגיאה! הסימניה אינה מוגדרת.
Device-Level Challenges שגיאה! הסימניה אינה מוגדרת.
7 Series FPGA Architecture שגיאה! הסימניה אינה מוגדרת.
UltraScale Architecture Layout שגיאה! הסימניה אינה מוגדרת.
Inference vs. Instantiation שגיאה! הסימניה אינה מוגדרת.
Simple Coding Steps Yield 3x Performance שגיאה! הסימניה אינה מוגדרת.
When to Code Constraints שגיאה! הסימניה אינה מוגדרת.
Synchronous Design Techniques שגיאה! הסימניה אינה מוגדרת.
Metastability שגיאה! הסימניה אינה מוגדרת.
Timing Closure שגיאה! הסימניה אינה מוגדרת.
Control Sets שגיאה! הסימניה אינה מוגדרת.
Use Embedded Blocks שגיאה! הסימניה אינה מוגדרת.
Pipelining for Performance and Power שגיאה! הסימניה אינה מוגדרת.
State Machine Encoding שגיאה! הסימניה אינה מוגדרת.
Why Synthesis Options? שגיאה! הסימניה אינה מוגדרת.
Synthesis Settings שגיאה! הסימניה אינה מוגדרת.
Attributes שגיאה! הסימניה אינה מוגדרת.
Apply Your Knowledge שגיאה! הסימניה אינה מוגדרת.
Summary שגיאה! הסימניה אינה מוגדרת.

Course Summary 29-1

Objectives Achieved? 29-1
Topics 29-2
Topics 29-3
Topics 29-3

© Copyright 2019 Xilinx

Follow Xilinx 29-4

Appendix: Glossary 30-1

Appendix: Glossary 30-1

Appendix 31-1

Additional Xilinx Courses 31-1

© Copyright 2019 Xilinx

Course Agenda

Slide 1-1:

2019.1

This module covers the agenda for the course. 10 minutes.

Slide 1-2:

Course Objectives

After completing this course, you will be able to:

- Implement the VHDL portion of coding for synthesis
  - Identify the differences between behavioral and structural coding styles
  - Distinguish coding for synthesis versus coding for simulation
  - Use scalar and composite data types to represent information
  - Use concurrent and sequential control structure to regulate information flow
  - Implement common VHDL constructs
    - Finite State Machines (FSMs)
    - RAM/ROM data structures
- Simulate a basic VHDL design
  - Write a VHDL testbench and identify simulation-only constructs
- Identify and implement coding best practices
  - Optimize VHDL code to target specific silicon resources within the Xilinx FPGA
- Create and manage designs within the Vivado Design Suite environment

© Copyright 2019 Xilinx

Slide 1-3:

Course Parameters

- Prerequisites
  - Basic digital design concepts
    - Flip-flops and logic gates
  - Basic understanding of clocked (synchronous) designing

- Course length
  - Three days

- Additional class building upon concepts provided in this class
  - *Advanced VHDL*

Basic VHDL knowledge (covered in this course)

- Entity/architecture declarations
- IF/THEN statements
- CASE statements
- Clocked processes
- Component instantiation

Basic Xilinx FPGA architecture (you should know the meaning of the following terms):

- Configurable Logic Block (CLB)
- Input/Output Block (IOB)
- Distributed RAM
- Block RAM
- Global clock buffer (BUFG)
- Carry logic
- Look-Up Table (LUT)
- Programmable interconnect (routing)

An introduction to FGPA architecture is covered in this module and a slightly more detailed overview of FPGA architecture is included in the “Targeting Xilinx FPGAs” module.

© Copyright 2019 Xilinx

Slide 1-4:

Agenda Day One

- Introduction to VHDL {Lecture} - 10 min.
- VHDL Design Units {Lecture, Lab} - 55 min.
- VHDL Objects, Keywords, Identifiers {Lecture} - 10 min.
- Scalar Data Types {Lecture} - 15 min.
- Composite Data Types {Lecture} - 15 min.
- VHDL Operators {Lecture} - 15 min.
- Concurrency in VHDL {Lecture} - 15 min.
- Concurrent Assignments {Lecture, Lab} - 80 min.
- Processes and Variables {Lecture, Demo, Lab} - 95 min.

Slide 1-5:

Agenda Day Two

- Conditional Statements in VHDL: if/else, case {Lecture, Lab} - 75 min.
- Sequential Looping Statements {Lecture, Lab} - 70 min.
- Delays in VHDL: Wait Statements {Lecture} - 10 min.
- Introduction to the VHDL Testbench {Lecture, Lab} - 65 min.
- VHDL Assert Statements {Lecture} - 10 min.
- VHDL Attributes {Lecture} - 15 min.
- VHDL Subprograms {Lecture} - 10 min.
- VHDL Functions {Lecture, Lab} - 60 min.
- VHDL Procedures {Lecture} - 15 min.

© Copyright 2019 Xilinx

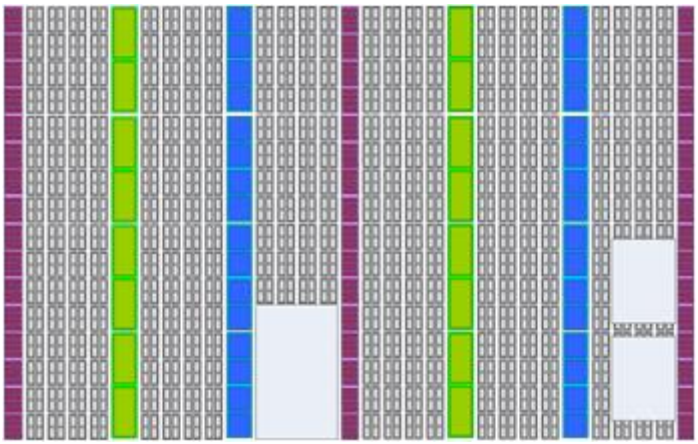
Slide 1-6:

Agenda Day Three

- VHDL Libraries and Packages {Lecture, Lab} - 65 min.
- Interacting with the Simulation {Lecture} - 15 min.
- Finite State Machine Overview {Lecture} - 10 min.
- Mealy Finite State Machine {Lecture} - 10 min.
- Moore Finite State Machine {Lecture, Lab} - 70 min.
- FSM Coding Guidelines {Lecture} - 20 min.
- Vivado Simulator and Race Conditions in VHDL {Lecture} - 15 min.
- Writing a Good Testbench {Lecture, Lab} - 75 min.
- Targeting Xilinx FPGAs {Lecture, Lab} - 80 min.

Slide 1-7:

Review of FPGA Architecture (1)



- Basic building block – the Configurable Logic Block (CLB)
  - Comprised of two or more logic slices*
  - Each slice contains two or more
    - Combinatorial logic Look-Up Tables (LUTs)
    - Flip-flop
    - Interconnecting resources (multiplexers)
  - LUTs can behave as
    - Combinatorial elements (n-inputs producing one output)
    - Synchronous shift registers
    - Memory (single port, dual port, RAM, ROM)

© Copyright 2019 Xilinx

*FPGA families will differ by both the number of resources within a CLB and the number of slices. For the most part, VHDL insulates the designer from the deep intricacies of these architectural details; however, in rare cases, the choice of coding style can significantly impact how these resources are used.

This and the next few slides will provides the rudimentary understanding of what an FPGA looks like on the inside. Having a basic grasp of the architecture will lead to a better understanding of coding styles and choices.

Terms to be familiar with:

- Configurable Logic Block (CLB) – Collection of both decision-making logic and simple memory.
- Input/Output Block (IOB) – Provides necessary support to interface to the outside world. May include additional flip-flops and logic for specialized interfaces such as DDR and serial.
- Distributed RAM – One of the many functions of the LUT.
- Block RAM – Larger repositories of memory. These are dual-port meaning that information can be accessed from two points.
- Global Clock Buffer (BUFG) – Part of a larger system of clock generation, distribution, and control.
- Carry logic – Special routing mechanism to accelerate math and wide logic functions.
- Look-Up Table (LUT) – Provides combinatorial logic solutions and can be used as small memories or shift registers.
- Programmable interconnect (routing) – How it all gets tied together. The largest and most numerous resource in an FPGA.

© Copyright 2019 Xilinx

Slide 1-8:

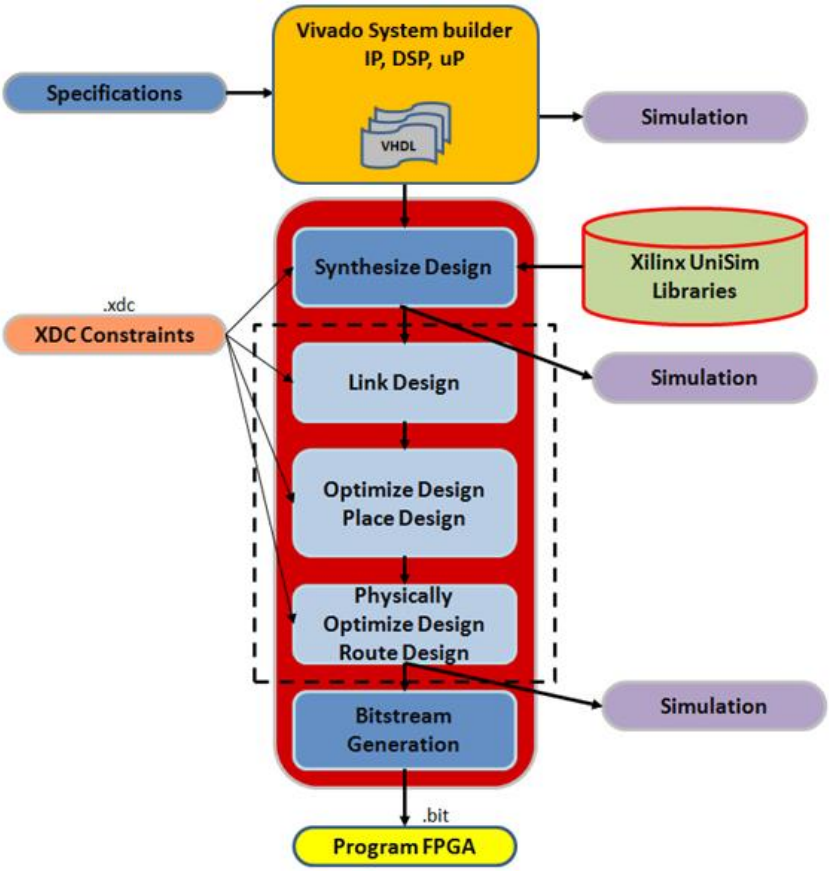
Review of FPGA Architecture (2)

- Specialized silicon resources
  - Clocking resources (DCM, PLL, and BUFG, for example)
  - Dual-port block RAM, memory interface
  - Math accelerators (DSP48/E)
  - Communications (TEMAC, high-speed serial, and PCI Express® core, for example)
  - Embedded processors
- Routing resources (switches and long lines, for example)

Slide 1-9:

Review of Xilinx Tool Flow

- Design entry: description of the design
- Synthesis: process of turning the description into a netlist
- Implementation: conversion of netlist into physical resources and connection
- Bitstream generation: creating an FPGA downloadable bitstream



© Copyright 2019 Xilinx

Design Entry

The flow begins with the development of the source code. Most often this is done with a text editor. The Vivado Design Suite provides an environment to configure, implement, verify, and integrate IP. IP can include logic, embedded processors, digital signal processing (DSP) modules, or C-based DSP algorithm designs. Custom IP is packaged following the IP-XACT protocol and then made available through the Vivado IP catalog. The IP catalog provides quick access to the IP for configuration, instantiation, and validation of IP. Xilinx IP utilizes the AXI4 interconnect standard to enable faster system-level integration. Existing IP can be used in the design in either RTL or netlist format.

The Vivado IP integrator environment enables you to stitch together various IP using the AMBA AXI4 interconnect protocol. You can interactively configure and connect IP using a block diagram style interface and easily connect entire interfaces by drawing DRC-correct connections similar to a schematic. These IP block designs are then packaged and treated as a single design source. Block designs can be used in a design project or shared among other projects.

Synthesis

The next step in the process is synthesis, which is the process of converting each source file into a netlist. A netlist is basically a list of connections among hierarchical modules and primitive components.

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Vivado synthesis is timing-driven and optimized for memory usage and performance. Vivado synthesis supports SystemVerilog as well as mixed VHDL and Verilog languages. The tool supports Xilinx Design Constraints (XDC), which is based on the industry-standard Synopsys Design Constraints (SDC).

Implementation

Netlist linking is done in `link_design`. Here if you do have third-party EDF, the files can be read into the Vivado IDE.

`opt_design`, `power_opt_design`, and `place_design` are the packing and placement functions. `power_opt_design` is optional, however.

phys_opt_design is also optional. The significant thing is that all of data is stored in one data model.

© Copyright 2019 Xilinx

You can run that entire flow in memory and never write anything files out when using the non-project batch design flow. But in any flow there is no data translation, no data model. There are huge advantages to that in terms of speed. Checkpoints can be saved at any stage of the design flow when using the non-project batch design flow. A checkpoint is basically a snapshot of the design in time. All of these commands can be scripted so the flow is very flexible.

The single data model in memory allows you to do a lot of things, such as cross-probing and fixing timing at the end of the implementation without having to reimplement again.

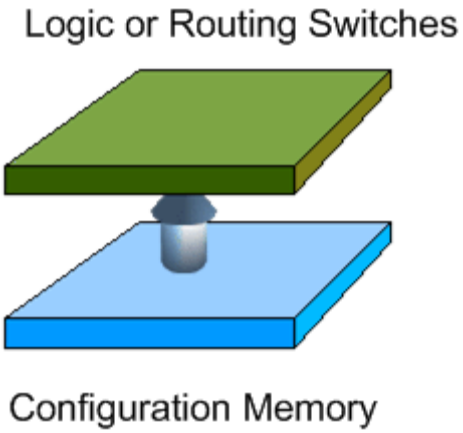
Bitstream Generation

Finally, the output of the implementation tools is fed into a bitstream generator, which produces the bit file that is loaded into the FPGA via the Vivado Hardware Manager (which will be used in this course).

Slide 1-10:

Waking Up

- FPGAs are SRAM-based devices
  - This implies that when power is removed, the FPGA loses its configuration
- The configuration is the collection of settings used to direct signals at switch junctures, set equations that perform logic, and define the behavior of silicon resources
- FPGAs are typically programmed from an external source
  - Configuration PROMs
  - Microprocessors
- Each FPGA has a startup module which
  - 3-states the outputs during configuration
  - Applies a global sync set/reset
  - Controls the PROM when mastering



© Copyright 2019 Xilinx

This slide sets the stage for making the distinction between Global Set Reset (GSR), which only occurs after a configuration, versus a user reset and a user applied reset.

The initial values that will be introduced in this course are only valid immediately post-configuration, whereas the user reset should drive signals and variables to a known state. This requires a different technique that will be covered later.

Slide 1-11:

Appendix

- Note that this course also includes the following appendixes
  - Appendix: "Glossary"

Slide 1-12:

Documentation for VHDL

- Websites
  - IEEE VHDL Language Reference Manual
  - VHDL specification: ieeexplore.ieee.org/document/4772740/?arnumber=4772740
- Generic books
  - HDL Chip design by Douglas Smith
  - The Designers Guide to VHDL by Peter Ashendon
- VHDL guidelines for use with Xilinx tools
  - *Vivado Design Suite User Guide - Synthesis (UG901)*
  - *XST User Guide (UG627)*

© Copyright 2019 Xilinx

Slide 1-13:

Latest Product Information

Please visit the following resources for the most current information on the Xilinx devices described in this course.

- User design information can be found in user guides
- IP core information can be found in product guides
- Characteristics, such as timing, performance, etc., are listed in data sheets
- For design and software issues or bugs, see www.xilinx.com > Support
- DocNav provides easy access to these documents

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Introduction to VHDL

Slide 2-1:

2019.1

This module discusses the history of the VHDL language and provides an overview of the different features of VHDL. 10 minutes.

Slide 2-2:

Objectives

After completing this module, you will be able to:

- Describe the original intent of VHDL and how this intent has influenced the design of the language
- Define key terms and concepts in relationship to VHDL
- Describe the different coding styles that are available in VHDL

© Copyright 2019 Xilinx

Slide 2-3:

Historical Background of VHDL

- 1981 – United States Department of Defense initiated the requirements for the **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit (VHSIC) program
  - Needed to describe the behavior of designs for both documentation and Simulation
  - Initiative known as **V**HSIC **H**ardware **D**escription **L**anguage (VHDL)
- 1983 – Intermetrics, IBM, and Texas Instruments begin development
  - Released in 1985
- 1984 – Xilinx produces the first FPGA*
- IEEE formalized the language into a standard
  - IEEE 1076-1987
  - Updated in 1993, 2000, and again in 2002**
  - Current standard is IEEE1076-2008***

*This is important because VHDL pre-dates FPGAs. VHDL did not really see acceptance in programmable logic until more than a decade later.

**The compilers and synthesizers in this class accept the IEEE 1076-2000 standard; therefore, all the examples provided in this class will confirm to this standard. Older versions of the tools may not fully support the capabilities demonstrated in this class.

***The Vivado® synthesis and Vivado simulator features support VHDL-2008 constructs. For more information on the supported VHDL-2008 constructs, please refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900) for supported constructs in the Vivado simulator feature, and *Vivado Design Suite User Guide: Synthesis* (UG901) for supported constructs in the Vivado synthesis feature.

© Copyright 2019 Xilinx

Slide 2-4:

VHDL History and Usage

- VHDL is, first and foremost, a tool for hardware modeling and documentation
  - For simulation as opposed to synthesis
  - Top-Down flow provides for better abstraction
- The IEEE 1076 standard is exhaustive with respect to modeling, but it defines only broad parameters for synthesis
- The result: A given hardware module does not necessarily lend itself to a consistent and universal gate-level implementation across various tools and target technologies

Broad parameters for synthesis means that synthesis tool manufacturers can create a netlist (a file that contains basic logical elements and how they are interconnected) in any way they wish; that is, for a given source code file, many different netlists can be created.

By contrast, Verilog began as a low-level, switched-gate design language for ASICs and evolved as a “bottom-up” language with superior low-level support. VHDL implements the VITAL library which now provides an equivalent amount of low level support.

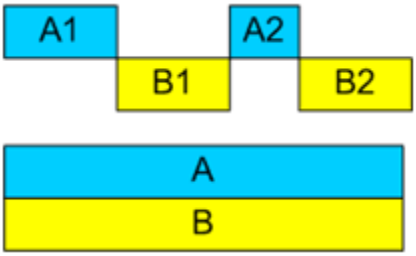
Much of the work being done on VHDL outside of the IEEE is performed by ACCELLERA.ORG, an industry consortium.

© Copyright 2019 Xilinx

Slide 2-5:

VHDL as a Language (1)

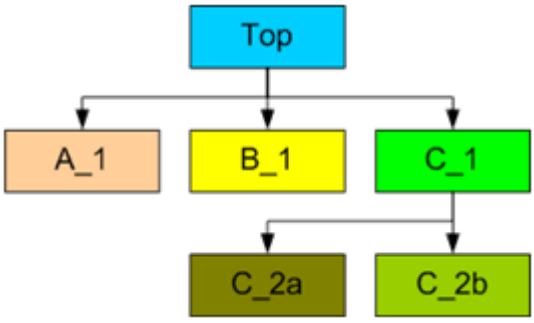
- Case insensitive
- Strong typing
  - Every signal, component, and function must be defined before it can be used
  - Size and type of target signal or variable must match exactly the left-hand side
- Concurrency
  - Software describes a single sequential flow
    - Multi-threading gives the appearance of concurrency
  - VHDL describes truly concurrent operations
    - Concurrent statements, instantiations, processes



Slide 2-6:

VHDL as a Language (2)

- Hierarchy and partitioning
  - VHDL supports and encourages the use of hierarchy
- Analysis order
  - VHDL is a single-pass compiler; everything must be defined before it can be used
  - The Xilinx tool automatically orders files properly for analysis



© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 2-7:

Compilation

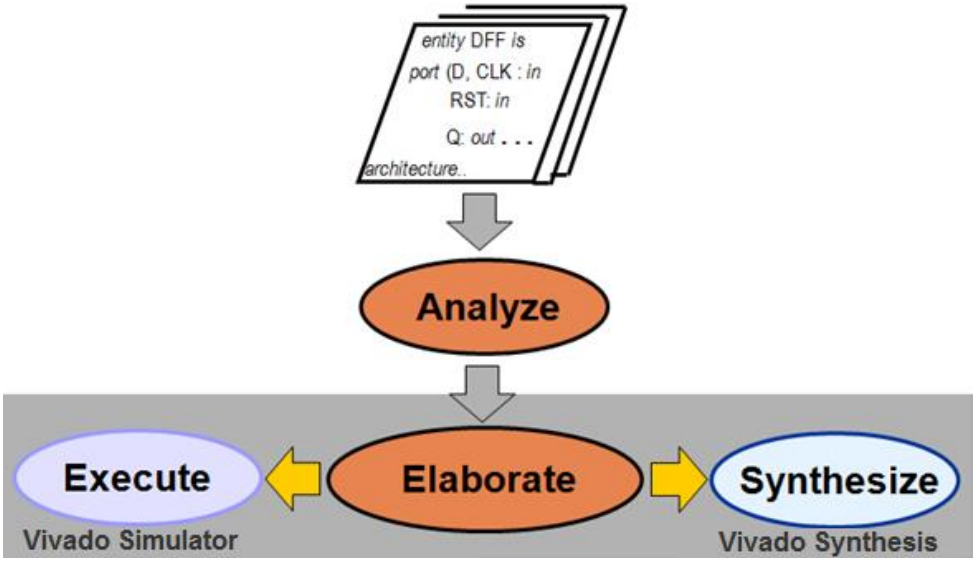
- Four levels of processing can take place for a VHDL hardware model
  - **Analysis**
    - The design unit is checked for syntax errors
    - Once finalized, it is stored in the *xil_default* library
  - **Elaboration**
    - The design hierarchy is fleshed out, starting from the top
    - A unique copy of each submodule instance is created
    - "Generics" expanded at this stage
  - **Synthesis**
    - A netlist description of the logic is generated either in an industry-standard or vendor-specific format
  - **Execution** (simulation only)
    - The model is simulated in discrete time steps
    - This is driven primarily by events on signals that trigger related processes

Note that today's compilers no longer list these as specific menu options. Nonetheless, this is the VHDL requirement for any given module.

© Copyright 2019 Xilinx

Slide 2-8:

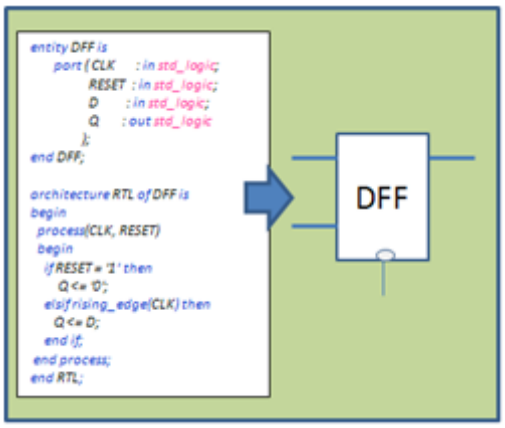
Compilation Path



Slide 2-9:

Coding Styles

- Coding styles
  - Behavioral coding – Use of higher levels of abstraction (algorithm), such as loops and conditional assignments
  - Register Transfer Logic (RTL) – Description of how data is passed from flop to flop and through combinatorial logic
  - Structural coding – Connecting instantiations



At this level of coding there are three forms – behavioral, RTL, and structural. The concept of inference forms the basis of behavioral coding while instantiations are performed to achieve structural coding.

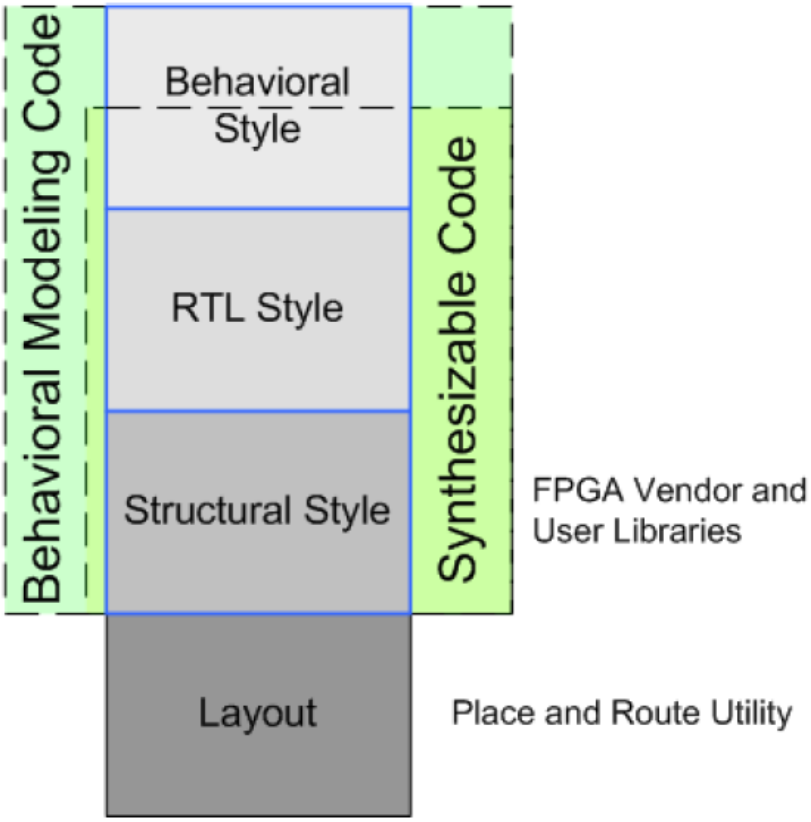
© Copyright 2019 Xilinx

It is important to note that these three styles can be mixed freely within a single VHDL component (or module). For example, RTL is commonly mixed with structural coding to provide “glue” between various component instantiations.

Slide 2-10:

HDL Description Levels

```
sum <= a + b after 3 ns;

x1: while (sum > 15) loop
    ...
end loop x1;

sum <= a + b;

inst_1: mux21 PORT MAP
        (a0 => inp1,
         a1 => inp2,
         s => sel,
         y => temp1);
```



Hardware model: Generally, but not always, unsynthesizable. Hardware modeling describes *what* happens, not *how* it happens. In this fashion, the behavior of complex systems can be described with relative ease and without knowing the gate level details of how the internal working operate.

Synthesizable code: This code is converted into a netlist by the synthesis tool and passed on to the implementation tools to ultimately become a bitstream that is downloaded into the FPGA.

FPGA vendor library: Contains a list of “black-box” primitives (that is, architectural resources) that can be instantiated by the user.

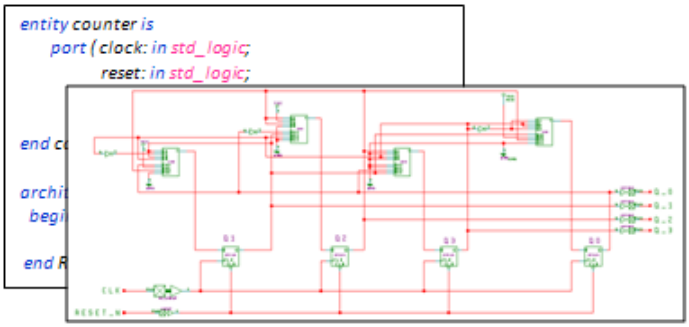
Place & Route utility: A collection of applications that stitches all of the netlists together (as netlists can be created from sources, third-party vendors).

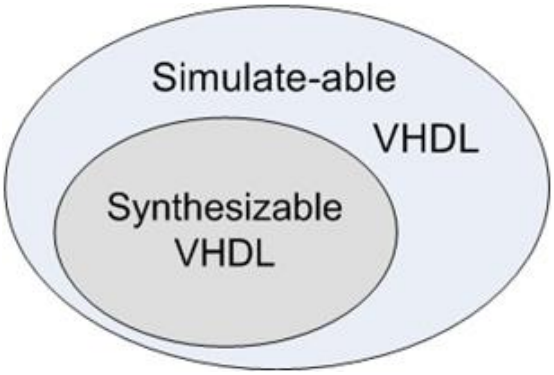
© Copyright 2019 Xilinx

Slide 2-11:

VHDL and FPGAs

- FPGAs added a new twist to the original intent of VHDL: synthesis
- VHDL was designed to document and simulate circuit designs
- Synthesis versus simulation
  - **Synthesis** is the process of converting the VHDL code to a collection of FPGA-specific primitives and connections
    - Must be physically realizable; therefore, there are limits as to what can be done
  - **Simulation** runs on a computer; therefore, many additional abstractions and capabilities exist in the simulation environment than exist for synthesis
    - Example: "wait for 12.031 ns;"





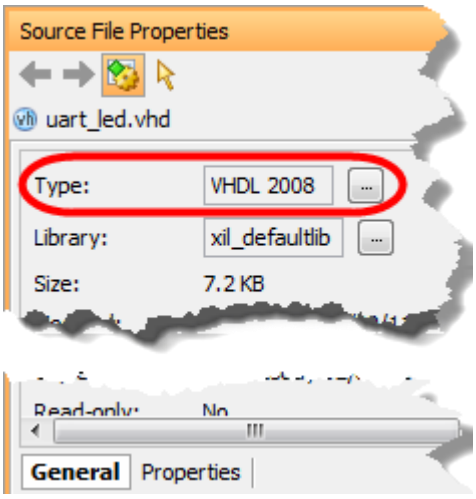
The “wait for 12.031 ns;” example is a legitimate statement and will execute properly in simulation. To realize this in hardware would require a clock accurate to better than 1 THz. This is just not achievable in today’s commercial technology. This type of command is available only within the confines of a simulator.

© Copyright 2019 Xilinx

Slide 2-12:

Setting up the Vivado Design Suite to Use VHDL-2008

- There are several ways to enable VHDL-2008 in the Vivado Design Suite
  - Set VHDL-2008 from within the GUI
    - Set File Type to VHDL-2008 using the Source File Properties window
  - Use the Tcl Console
    - Enter the following Tcl command
    - `set_property FILE_TYPE {VHDL 2008} [get_files <file_name>.vhd]`



When working with the Vivado Design Suite in non-project mode, use the `read_vhdl -vhdl2008` Tcl command to read VHDL-2008 files into the design.

Slide 2-13:

Summary

- VHDL was created by the US Department of Defense and remains the preferred HDL for most military, military contracts, and high-reliability systems
- VHDL is case insensitive, strongly typed, concurrent language
- VHDL supports data flow, structural and behavioral style of coding

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Design Units

Slide 3-1:

2019.1

This module provides an overview of typical VHDL code, covering design units such as libraries, packages, entities, architectures, and configuration. 10 minutes.

Slide 3-2:

Objectives

After completing this module, you will be able to:

- Describe the design units that are available in VHDL code
- Define the library and packages and how they are declared
- Explain entity and architecture syntax and declarations

Slide 3-3:

VHDL Design Units

- Library
- Package
- Entity
- Architecture
- Configuration

VHDL is composed of design units

These are the basic building blocks of which all source code, behavioral, RTL, or structural code is comprised

Some design units are primary, while others are secondary (dependent). *No secondary design unit can exist on its own*

All design units are ultimately compiled into a “library.”

© Copyright 2019 Xilinx

Slide 3-4:

Typical VHDL "Module"

- Is a member of a design hierarchy
- Performs a single "idea" or task
- Generally small, often instantiating lower-level modules
- Every module should have the following defined for it
  - Its interface (covered by the entity)
  - Its behavior (covered by the architecture)
  - Its target performance (covered by constraint file and synthesis options)
- Every project should have the following defined for it
  - Its performance
  - Its connection to the outside world
    - These are covered in the Xilinx Design Constraint (XDC) file

A VHDL module consists of four basic components: library, entity, architecture, and configuration.

© Copyright 2019 Xilinx

Slide 3-5:

Typical VHDL "Module" Example

- Most VHDL modules will follow a simple pattern
  - Library declarations
  - Entity
  - Architecture
- Entities describe the interface between the outside world and the behavior described in the architecture
  - Entities can legally include
    - Generics
    - Constants
    - Ports
    - Functions and procedures

Typical VHDL module structure/file

```
--Library Declarations
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--Entity Declaration
entity this_module is
        [<generic declarations>]
        [<port declarations>]
end entity this_module;
--Architecture Declaration
architecture BEH of this_module is
     <architecture declaration area>
  begin
     <architecture function
                 description>
  end architecture BEH;
```

Constants, functions, and procedures may also exist within an architecture or process/process body.

VHDL allows infinite architectures associated to an entity if the names are unique in the target library. Vivado synthesis will allow multiple (two or more) architectures if they are in different files and are analyzed separately into the target library, but make sure that the required architecture is enabled.

© Copyright 2019 Xilinx

Slide 3-6:

Library Declarations

- Libraries are repositories for commonly used items

- Libraries are created by
  - IEEE — Includes widely used standards such as std_logic/std_logic_vector
  - Vendors — To support their IP
    - Xilinx provides UNISIM and SIMPRIM libraries for simulation and synthesis of low-level architectural primitives
  - Designers — For convenience and code reuse

- How to incorporate a library into a design – two-step process
  - Declare the library
  - Specify which part of the library is to be used
    - "ALL" means to include the entire contents of a package
      - Finer granularity is achieved by replacing all with the specific item within the library/package

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

Libraries are comprised of one or more packages. Each package may have one or more member.

The “use” statement is used to first identify which package within the library is being selected, then either a specific member of the package or all members of the package.

© Copyright 2019 Xilinx

Slide 3-7:

Package Declarations

- Packages are constructs that help collect and organize functions, procedures, types, subtypes, and constants

- The syntax for the package is very similar to that of the entity and architecture

```
package <package name> is
   <package declarative items>
end [package] <package name>;
```

- The package body follows the same pattern and must share the same name as the package to be properly associated

```
package body <package name> is
   <package body declarative items>
end [package body] <package name>;
```

Syntax:

- <package name> - any legal identifier. This name must not be the same as any other package name that is included in the same library that this package resides in. Identical package names may reside in other libraries.
- <package declarative items> - any combination of type, subtype, constant, function, or procedure declarations. Any function or procedure declarations must be defined in a package body. If no functions or procedures are present, then the package body is not required.

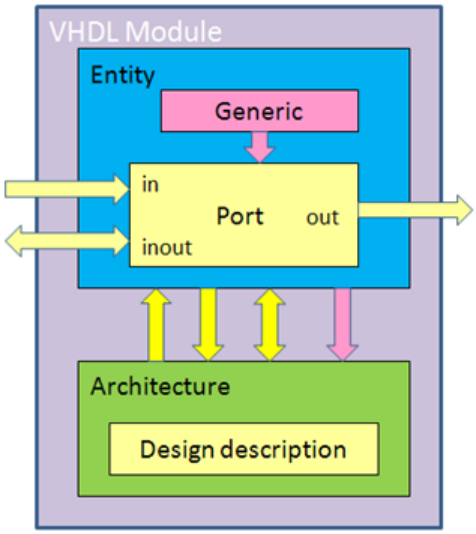
© Copyright 2019 Xilinx

Slide 3-8:

Entity Declaration

- The entity statement is a primary unit
- It describes the interface between the behavior of the module and the "calling" module or outside world
- Generic – List of default values typically used to control the width of ports or internal signals
- Ports – List of input, output, and bidirectional signals
  - Only the signals listed in the port are available to the architecture
- Syntax

```
entity <entity name> is
    <generic declarative area>
    <port declarative area>
  end [entity] [<entity name>];
```



Syntax:

- <entity name> - a unique and legal identifier.
- <generic declarative area> - region where generics are defined.
- <port declarative area> - region where ports are defined.

© Copyright 2019 Xilinx

Slide 3-9:

Port Declarations

- Ports provide the mechanism to pass signals into and out of an entity
  - These signals are the only view that the entity and corresponding architecture have of the outside world
  - In – Signals arriving into the module (can be used or "read" but not assigned)
  - Out – Signals produced by the module (can be assigned, but not read)
  - Inout – Bidirectional signals that can be used or modified*
  - Buffered – Like out, but can be read internal to the module (rarely used)

- Syntax

```
port ( <port name> : <dir> <type> [:= <expression>];
                          . . .
       <port name> : <dir> <type> [:= <expression>] );
```

- Example

```
port( clk      : in     std_logic;
      reset    : in     std_logic;
      data_bus : inout  std_logic_vector(3 downto 0);
      result   : out    std_logic
     );
```

*An inout direction requires the module to manage the bidirectionality. It is important to know that there is no true 3-state buffer within the FPGA—it is all modeled with 2-state logic. The only “true” 3-state buffers exist in the I/O blocks. You must be careful to resolve multiple devices attempting to write to this signal.

Syntax:

- <port name> - any legal and unique identifier
- <dir> - in, out, inout, buffer
- <type> - any legal type. Typically, std_logic and std_logic_vector (covered in data types)
- <expression> - any valid expression representing a legal initial condition of the specified type

© Copyright 2019 Xilinx

Indentation and use of white space is at your discretion. However, writing cleanly written and formatted code that makes understanding and maintenance much easier is highly recommended.

In VHDL-2008, the restriction on reading out-mode ports is also removed.

Slide 3-10:

Architecture Declarations

- The architecture describes the behavior of the module
- It can be coded
  - Structurally
    - Uses "instantiations" to explicitly describe modules or silicon resources
  - As RTL
    - Describes action as combinations of logic and registers
  - As behavioral
    - Uses higher level abstractions such as "loop" and "if" to describe actions
  - Any mix of the above
- Syntax

```
architecture <architecture name> of <entity name> is
    <architecture declarative area>
  begin
    <architecture functional area>
  end [architecture] [<architecture name>];
```

Syntax:

- <architecture name> - any legal and unique identifier. This names the architecture, not the design (which is the entity name). The name is only for reference and used when specifying full paths to other tools or when multiple architectures are used.

  Typical names are “BEH”, “Behavioral”, “STRUCT”, “Structural”, “RTL”, “Testbench.” The name has no bearing on what any of the tools do or how the code is used—it is for the designer’s benefit only.

© Copyright 2019 Xilinx

- <entity name> any legal and previously defined entity name identifier. This is how the architecture ties to the entity.
- <architecture declarative area> - this region is used to describe what is being used in the architecture, not how it is used. Typically, signal, constant, component, types, subtypes, functions, and procedures are described here.
- <architecture functional area> - this region is used to describe the behavior of the module.

Slide 3-11:

The "Entity"

- An entity can have zero or more architectures associated with it
  - Advanced designs will use a "configuration" statement to resolve the confusion caused by multiple architectures
  - The VHDL standard does not specify a maximum number of architectures per entity

Typical VHDL module structure/file

```
--Library Declarations
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--Entity Declaration
entity this_module is
        [<generic declarations>]
        [<port declarations>]
end entity this_module;
--Architecture Declaration
architecture BEH of this_module is
    <architecture declaration area>
  begin
    <architecture function
              description>
  end architecture BEH;
```


© Copyright 2019 Xilinx

Slide 3-12:

Configuration Declarations

- Specifies the binding of one architecture body with the entity
- Specifies the bindings of components used in the selected architecture body to the entity
- Entity can have any number of different configurations
- Syntax

```
configuration <configuration name> of <entity name> is
        <declarations>
        <block configuration>
end [configuration] [configuration name];
```

- The allowed block configurations are

```
for <component_instance_name> : <component_name>
        --use clause
        use entity <library_name>.<entity_name>(<architecture_name>)
end for;
```

Syntax:

- <configuration name> - any legal and unique identifier. The name is only for reference and used when specifying full paths to other tools or when multiple configurations are used.
- <entity name> - any legal and previously defined entity name identifier. This is how the architecture ties to the entity.
- <declarations> - this region is used to describe which architecture is to be associated with the entity.

© Copyright 2019 Xilinx

Slide 3-13:

Module Partitioning Guidelines

- How big should a module (entity/architecture) be?
  - Large enough to contain a single “thought” or “idea”

- Key considerations when partitioning a design for VHDL design entry
  - Register outputs at module boundaries
    - Consistent with RTL coding structure
  - Minimize the number of clocks per block
    - Important for timing constraints and optimization
  - Maintain critical signals within a block
    - Eliminates need for optimization across hierarchical boundaries
  - Keep blocks small enough for rapid verification
    - Quick, relatively simple testbench for each submodule

Example of a “thought” or “idea”: This module represents an RS-232 UART, this architecture encompasses all the code to successfully implement a debouncer circuit, or this design represents a 15 tap FIR filter, for example.

Generally, if it can be expressed as a single, simple sentence, it is the right size. This does not exclude the use of hierarchies. Stating that “This module represents an RS-232 UART” can be built of even more primitive user modules such as “Baud Rate Generator,” “Receiver,” and “Transmitter”.

Partitioning, in this context, refers to separating and segmenting the overall design into logical modules.

© Copyright 2019 Xilinx

Slide 3-14:

Summary

- VHDL design units consists of library, package, entity, architecture, and configuration
- Entity is a primary unit, it can have zero or more architectures associated with it
- Architecture describes the behavior of the code
- Configuration describes entity and architecture bindings

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Objects, Keywords, Identifiers

Slide 4-1:

2019.1

This module discusses the data objects that are available in the VHDL language. It also covers keywords and identifiers and how to comment a piece of code in VHDL. 10 minutes.

Slide 4-2:

Objectives

After completing this module, you will be able to:

- Use appropriate keywords in VHDL
- Describe identifiers and the rules for writing identifiers
- Comment the piece of VHDL code

© Copyright 2019 Xilinx

Slide 4-3:

Keyword and Identifier

- Keywords are reserved words for language constructs and are used to denote a particular function to the synthesis/compilation tool
- Identifiers are "nouns" that are used to describe various constructs within VHDL
  - Signal, variable names
  - Function, procedure, constants
- An identifier cannot be a "keyword"
- VHDL is case insensitive
  - "IDenTifier" is the same as "identifier" is the same as "IDENTIFIER"
- Rules
  - Must start with a letter
  - May contain letters, numbers, and underscores
  - No contiguous underscores
  - The underscore may not end an identifier
  - Length is specified by the provider of the synthesis or compilation tool
    - Infinite length is supported by the standard

The full list of keywords is provided in the VHDL *Language Reference Manual*. More conveniently, most context-sensitive editors, such as the one provided by Xilinx in the Vivado® Design Suite tool set will highlight or otherwise identify keywords.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 4-4:

Identifiers Exercise

- Exercise: Determine whether each of the following identifiers is valid in VHDL
  - thisisjumbledup
  - 100Hz
  - Hz_100__000
  - senseCharacter_
  - C_style_identifier
  - javaStyleIdentifier
  - Hz-100_000_000

- thisisjumbledup: Even though this identifier is hard to read and not a recommended coding style, it is, nonetheless, legal.
- 100Hz: Illegal due to the identifier beginning with a number.
- Hz_100__000: Illegal due to two sequential underscores.
- senseCharacter_: Illegal due to the identifier ending with an underscore.
- C_style_identifier: Many VHDL designers choose to follow the C style of writing identifiers using the underscore character. This is usually a very good choice as some of the tools in the tool flow will change the capitalization on identifiers.
- javaStyleIdentifier: A few VHDL designers choose to follow the Java, or mixed capitalization, style. This is not illegal, but may lead to some difficulty reading some tool reports.
- Hz-100_000_000: Illegal due to the use of an illegal character (hyphen).

© Copyright 2019 Xilinx

Slide 4-5:

Expressions and Literals

- An expression comprises of operator and operands
- Data objects form the operands of an expression and their value is used by operators in VHDL

```
Y<= A +( B - C);   -- Simple expression with operands A, B & C, and operators '+' & '-'
M<=Y;              -- Can also be a single identifier
sig_hold<= func_or(a,b); -- or function call
```

- A literal is a constant valued operand

```
A<= '1';                -- Single bit binary literal (bit or std_logic)
Y<= "10000100";         -- Bit string literal where Y is [7:0] bus
                        -- (Note: Use " " for strings)
CH<='A';                -- Single character literal
STR<="VHDL CLASS";      -- Character string literal

INT<= 35;               -- Integer literal
State<= IDLE;           -- Enumerated literals
```

Remember that VHDL is strongly typed:
Data type of LHS should match RHS.

The above examples illustrate unconditional signal assignments where the value on the Right Hand Side (RHS) of the assignment symbol (<=) is applied to the object on the Left Hand Side (LHS).

VHDL operands can legally include literals, identifiers, qualified expressions, Index and slice names, function calls, aggregates, records, and record fields.

In the examples of literals in the illustration above, the **bolded** text represents the various literal values. These values are shown in a simple, unconditional signal assignment statement. It is assumed that:

- A is of type bit or std_logic,
- Y is of type bit_vector or std_logic_vector,
- CH is of type character
- STR is of type string (note that the string definition must be defined with a range 1 to 10
- INT is of type integer
- State is of a user-defined enumerated type

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 4-6:

Data Objects in VHDL

- Data objects hold a value of a specific data type

- VHDL has four classes of data objects
  - Constants - have a single value for a given instance and cannot be changed
  - Variables - have a single value and are updated using a variable assignment in a process or function
  - Signals - have a single value and are updated using a signal assignment
  - Files - provide sequential access to files

```
signal A : std_logic;
constant  DATA_BUS_WIDTH:  integer:= 16;
variable count_temp: unsigned(2 downto 0);
file dataout : text open read_mode is "Stordat.txt";
```

VHDL data objects are used to pass values from one point to another.

Each data object can be assigned to one of a collection of finite values.

Constants can only hold one specific value of a specific data type and cannot be changed once declared.

Variables can hold any value of a specific data type and are normally used for temporary values inside a process or function.

Signals hold a list of current and future values to be assigned that are to appear on the signal.

File refers to system file which contains a value of specified data type and the values are read or written sequentially.

© Copyright 2019 Xilinx

Slide 4-7:

VHDL Object Classes

- Four object classes exist in VHDL in which certain qualities of the data object are identified
  - Scalar
    - Single-value object, defined indexes, ordered
  - Composite
    - Group objects, similar or different types
  - Access
    - Defines pointers to objects
  - File
    - Sequence of objects of a given type

Slide 4-8:

VHDL Comments

- Comments can greatly enhance the readability of the code, provide useful documentation, and make the intent of the designer clear
- Rules
  - VHDL comments always begin with a "--" and end with a new line
    - Implies that there are no multi-line comments such as /* and */ from C/C++ in VHDL prior to VHDL 2008
  - A comment may begin anywhere in the line
  - A comment always ends with the new line character
  - Any printable character may be included in a comment

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 4-9:

Five Levels of Commenting

- External documents
  - Any documentation not included, but cited, in the source module
- File-level (header) comments
  - Introduces the module with file name and module contents
  - Describes the inputs and outputs to/from this module
  - Describes what the module does
- Section or code group comments
  - Explains the purpose of a group of code
- In-line comments
  - Pinpoints the exact purpose of a statement of code
- Identifier-level comments
  - Good naming practices makes the code more readable, less error prone

- External documents:
  - Detailed design document is the primary source
  - Other likely sources - Websites, Textbooks
- File header:
  - Introduces the module with file name and module contents, project, and company
  - Describes what the module does
  - Lists the authors
  - Describes the inputs and outputs to/from this module
  - Includes a revision history
  - Includes a "known issues" list
  - Identifies special conditions required by this module
- Section or code group comments:
  - Explains the purpose of a block of code

© Copyright 2019 Xilinx

  - Multiple lines of comments may be required to support one line or block of code
- In-line comments:
  - Pinpoints the exact purpose of a statement of code
  - Do not include a comment that is redundant with the line of code it supports
    - Counter = counter + 1; // increment counter by one
- Identifier-level comments:
  - Good naming practices makes the code more readable, less error prone
  - Use meaningful names so that the code can be understood without a data dictionary
  - Not like this - C5h1 := x002_a + tyz(3).n4; (VHD)

Slide 4-10:

Header Comment Example

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer: Bill
--
-- Create Date:    11:11:08 06/04/2009
-- Design Name:    Reset bridge for wave_gen module
-- Module Name:    reset_bridge - Behavioral
-- Project Name:   wave_gen
-- Target Devices: xc7k325tffg900-2
-- Tool versions:  2013.2
-- Description:
--  This module is a specialized synchronizing circuit for resets.
--  The input is an asynchronous, active-high reset, and the output is a synchronized version
--  that asserts asynchronously (hence combinatorially from the asserting edge of the input),
--  but deasserts synchronously.
--
-- Dependencies:   None
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
```

Slide 4-11:

Apply Your Knowledge

1) What are the five levels of commenting?

2) How does one add a comment to source code?

© Copyright 2019 Xilinx

1) What are the five levels of commenting?

External documentation, file header comment, block or group comment, in-line comment, identifier choice.

2) How does one add a comment to source code?

Comments are begun with the double dash or hyphen “--” and ends with the new line character.

Slide 4-12:

Summary

- VHDL has four classes of data objects: signal, constant, variable, and file
- Identifiers are "nouns" that are used to describe various constructs within VHDL
  - An identifier cannot be a "keyword"
- VHDL comments always begin with a "--" and end with a new line

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Scalar Data Types

Slide 5-1:

2019.1

Data types form one of the cornerstones of any language. In this module, both intrinsic and commonly used data types are covered. Basic operations and initialization techniques for these data types are also included. 15 minutes.

Slide 5-2:

Objectives

After completing this module, you will be able to:

- Use appropriate data types when declaring ports and signals
- List legal values for standard data types
- Create scalar data types

© Copyright 2019 Xilinx

Slide 5-3:

Data Types

- A wide range of available data types provides flexibility in hardware modeling and built-in error checking to ensure signal compatibility in large, complex models
  - Typing rules must be obeyed!

- Data types are very important in VHDL
  - A given data type allows only values within its range to be applied
  - Each object (signal, variable, constant, or port) must have its type defined when declared

- VHDL is a strongly typed language
  - Connected signals must be of the same type
  - Strong typing enables the use of operator and function "overloading"

- VHDL has a limited number of "built-in" data types
  - Libraries extend the number of data types

A data type in VHDL is a name that has been associated with a set of values and a set of operations.

In the overall VHDL language, there are a large number of data types, but this module will focus on those commonly used for synthesis, including FPGA development.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 5-4:

Scalar Data Types

- Scalar data types are single values

- This class covers
  - bit
  - boolean
  - std_logic
  - integer
  - real
  - character
  - Physical concepts and amounts
  - Enumerated types for immediate recognition

bit - {0,1} – A built-in data type which has been supplanted by std_logic types.

boolean – { false, true } – Boolean expressions are frequently used in behavioral modeling.

std_logic and std_ulogic – { ‘0’, ‘1’, ‘L’, ‘H’, ‘U’, ‘X’, ‘Z’, ‘-’, ‘W’ } – std_logic (and std_logic_vector) are the most commonly used data types for both synthesis and simulation, although not all values of std_logic are synthesizable (such as ‘L’, ‘H’, and ‘W’).

integer – { natural } – Integers are most frequently associated with mathematical computation (integers can be used to represent fixed-point real numbers), indices into arrays and matrices, and loop values. Integers can be both simulated and synthesized.

real – { natural } – Real values are generally used in simulation for computation. A recently approved IEEE library supports single precision floating point.

character – { control, numbers, symbols, alpha …} – Synthesizable and can be used to handle character data (that is, strings) for human interface devices.

Physical concepts and amounts – { time, … } – The time type is the only naturally supported physical type. Extensions for other units are available through custom or IEEE libraries. These are not synthesizable.

Enumerated types – { user defined } – Enumerated types are synthesizable and most frequently used with state machines. Enumerated type enable the synthesis tool to select the most appropriate method to represent these values in hardware.

© Copyright 2019 Xilinx

Slide 5-5:

bit and boolean

- Concise for modeling logic, but it does not adequately model hardware
  - Use std_logic/std_logic_vector instead

type **bit** is (**'0', '1'**) ;

```
architecture Behavioral of mux is
    signal  A, B, SEL, Z : bit;
  begin
    Z <= A when (SEL = '1') else B;
  end Behavioral;
```

- Useful for modeling at a more abstract level

type **boolean** is (**false, true**) ;

SEL ='1' and F >= G
both yield boolean result

The type declarations shown are taken from the library STD and the package Standard. These are predefined and implicit for every VHDL model and, therefore, no explicit library declaration is ever required.

The bit data type is generally not used anymore for synthesis. It has been essentially replaced by std_logic. Bit data type does execute more quickly in simulation as there are only two states rather than nine. If you choose to use this type, be certain that there is no possibility that any other state may occur as it will be resolved to a '0' or '1' and the simulation may not match reality.

The boolean type is synthesizable; however, it is more frequently seen in testbenches.

You can try all the codes given in the slides.

© Copyright 2019 Xilinx

Slide 5-6:

std_ulogic and std_logic

- std_ulogic was developed from the Multi-Value Logic (MVL) system and provides for more detailed hardware modeling than bit
  - Supports different signal strengths, don't-care conditions, and 3-state drivers
  - Defined in package std_logic_1164

- std_logic is the "resolved" form of std_ulogic and is more commonly used

```
type  std_ulogic  is  ( 'U',      -- Uninitialized
                        'X',      -- Forcing Unknown
                        '0',      -- Forcing Zero
                        '1',      -- Forcing One
                        'Z',      -- High Impedance
                        'W',      -- Weak Unknown
                        'L',      -- Weak Zero
                        'H',      -- Weak One
                        '-'       -- Don't Care
                      ) ;
```

Recall: type **bit** is limited to ('0', '1')

	U	X	0	1	Z	W	L	H	D
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
D	U	X	X	X	X	X	X	X	X

VHDL is case insensitive. Although this is true for identifiers and keywords, the std_logic and std_ulogic types are actually enumerated types which must match exactly. Therefore, when std_logic/std_ulogic is used, the values ('Z', 'U', 'W', etc.) must always be capitalized, as that is how the enumeration types are defined.

The 'u' in std_ulogic means unresolved. This indicates that a function, at some point after the initial compilation takes place, returns to "resolve" or has each signal take on a final meaning.

This topic cluster will generally deal with resolved types only and will use std_logic/std_logic_vector to represent all signals unless otherwise indicated.

© Copyright 2019 Xilinx

Slide 5-7:

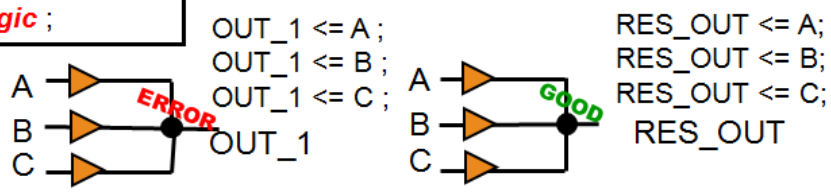
std_logic versus std_ulogic

- Both contain the same set of possible values
  - The difference is in implementation
  - The u in ulogic means unresolved

- If you want to drive two or more signals to a common output of type std_logic, some form of a resolution function must be used

- std_ulogic offers no such capability, but it does provide a built-in means of error checking for inadvertent multiple drivers

```
signal A,B,C,RES_OUT : std_logic ;
signal OUT_1 : std_ulogic ;
```



- RES_OUT (std_logic) will make it through compile but often will not make it through implementation (multiple drivers)
  - Takes longer to find error

The expression on the right is permitted in simulation, with the help of predefined resolution table available in the std_logic_1164 package. A synthesis compiler targeting Xilinx FPGAs would not allow you to model a ‘wired and’ or ‘or’ because this is not supported in the chip-level technology.

© Copyright 2019 Xilinx

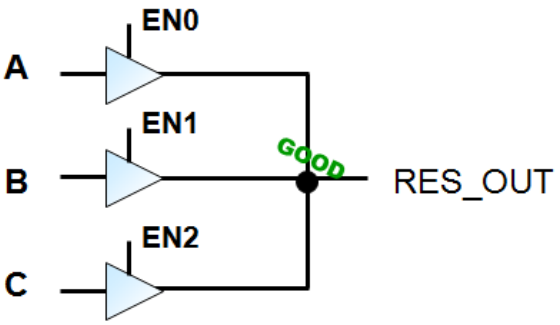
Slide 5-8:

Signal Resolution

- A given output may not have multiple wire-or drivers
  - Several methods of modeling 3-state buffers exist; one is using a conditional signal assignment and data type std_logic

```
signal A,B,C, RES_OUT : std_logic ;
```

```
RES_OUT <= A when EN0 = '1' else 'Z' ;
RES_OUT <= B when EN1 = '1' else 'Z' ;
RES_OUT <= C when EN2 = '1' else 'Z' ;
```



This rule depends on the technology. Within a Xilinx FPGA and most devices, wired-or and wire-and drivers are not allowed.

At the board level, however, signals can be combined via open-drain/open-collector outputs—in which case, the std_logic weak high (H) and weak low (L) could be used to model those drivers.

In both cases, the synthesis tool must recognize that capability and allow those statements for compilation.

Internal to a current FPGA (within the fabric, not the I/O banks) there is no such thing as 3-state. The concept of 3-state is represented using regular logic because this is easier to shrink in silicon and performs faster.

© Copyright 2019 Xilinx

Slide 5-9:

integer and real

- The integer type allows for flexible, intuitive quantities and values
  - Specifying the range of any integer has significant impact during synthesis
  - Without specifying a range, the compiler defaults to the maximum range
  - The standard specifies a minimum 32-bit implementation, but no maximum*

type **integer** is range . . .	*signal* A : *integer* *range 0 to 7;* *signal* B : *integer* *range 15 downto 0 ;*

- The real type allows you to use floating point values in the range of +1e38 to -1e38
  - Declare reals with the intended range of real values
  - Real values are generally not synthesizable**
    - Can be used in synthesizable code and resolved during synthesis for constant use in synthesizable code

type **real** is range . . .	*type* CAPACITY *is range -25.0 to 25.0 ;* *signal* SIG_1 : CAPACITY := 3.0 ;

*This implies that the compiler manufacture may implement a larger range for an integer. This introduces inconsistencies among the various tools. The implication for synthesis is that more fabric (and hence more power) may be consumed than is necessary. For simulation, out-of-bound errors may not be found.

**IEEE has released a single-precision, floating-point library for synthesis.

The total range of the type integer is somewhat compiler dependent; however, it must encompass the minimum range -2E31+1 to 2E31-1, which implies a 32-bit value.

Notice that the real data type is not used like integer. Use the real data type to create your own data types, as shown in the example.

For effective modeling, reals may be used to scale and offset physical data types, or used for purely mathematical processes. Reals can be used with physical constants (such as Time or Voltage) within simulation environments that support mixed signal modeling.

© Copyright 2019 Xilinx

Reals can be used in synthesizable code if the results that they produce are resolvable during synthesis. For example, if a memory block size is specified as an integer and the number of bits is required for the synthesizable code, then a *real* function to compute the log of the size divided by log 2 will produce the minimum number of bits required to represent the integer size.

Slide 5-10:

What Can and Cannot be Done with Integers and Reals

- All forms of comparison are legal
  - >, <, >=, <=, =, /=
  - **Caution:** Testing real numbers for equality is not reliable
    - Recall that a real number (floating point) may not be exact
    - The actual implementation of reals varies from vendor to vendor
  - Only test like types (integer against integer, real against real)
    - Use conversion or casts to make the types the same

- Basic math functions*
  - +, -, *, /, mod, rem
    - Be aware that some of these operations can cause problems during synthesis, but not during simulation

*NUMERIC_STD provides the appropriate support for signed and unsigned.

© Copyright 2019 Xilinx

Slide 5-11:

Data Type Conversion

- VHDL is a strongly typed language
  - By its nature, assigning one type to another is illegal
  - Requires a conversion mechanism to perform the change
- Three types of conversion processes
  - "casting"
  - Conversion function
  - ‘image/‘value attributes
- Common conversions
  - std_logic_vector to integer
  - integer to std_logic_vector
- Frequently used for indexes into arrays; math with "real-world" data

Conversion from one type to another is an important step here.

© Copyright 2019 Xilinx

Slide 5-12:

VHDL Type Casting and Conversion Functions

- Type casting
  - Type casting is used to move between the std_logic_vector type and the signed and unsigned types
  - std_logic_vector and signed/unsigned types are both defined with a specific bit width
  - signed (ex1) – where ex1 is a bit vector of the std_logic_vector type
  - std_logic_vector (ex2) – where ex2 is a bit vector of the signed or unsigned type

- Conversion function
  - Functions are used to move between signed and unsigned types and the integer type
  - Integers do not define a bit width
  - to_signed (ex3, ex2'length) – where ex3 is an integer and ex2 is a bit vector of the signed or unsigned type
  - to_integer (ex4) – where ex4 is a bit vector of the signed or unsigned type

A type cast between std_logic_vector and signed/unsigned can be used as long as the origin and destination signals have the same bit width.

Integers do not have a set bit width, which is why the conversion function from integer to signed/unsigned includes a specification of the intended bit width.

© Copyright 2019 Xilinx

Slide 5-13:

std_logic_vector to/from integer

- std_logic_vector → integer
  - First, the std_logic_vector must be cast to a signed or unsigned type
    - This process describes how the bits are to be used
  - Next, the signed or unsigned type should be converted to an integer

```
integer_value <= to_integer ( unsigned ( slv_value ) );
```

- integer → std_logic_vector
  - First, the integer must be converted to a signed or unsigned type
    - The length of the final signed/unsigned must also be specified (shown as 'n' below)
  - Next, the signed or unsigned type should be cast to a std_logic_vector

```
slv_value <= std_logic_vector ( to_unsigned ( integer_value, n ) );
```

Consider: a std_logic_vector should be many std_logics.

Technically:

- type UNSIGNED is array (NATURAL range <>) of STD_LOGIC
- type SIGNED is array (NATURAL range <>) of STD_LOGIC

This means that signed and unsigned are also many std_logics. So std_logic_vector and signed/unsigned are pretty much the same (other than signed/unsigned cannot have a negative index). This allows them to be cast as long as the std_logic_vector does not have a negative index.

Integers are something completely different. While some may think that integers are also just bunches of bits (which they are in the computer) but not in the VHDL environment.

An integer is defined almost like a literal.

- type INTEGER is range -2147483647 to 2147483647;

This makes it impossible to be cast, but must be converted.

© Copyright 2019 Xilinx

Slide 5-14:

Types and Subtypes

- Types describe a class of data representation
  - STD defines a specific collection of types such as integer, real, bit, etc.
  - New types can be created using enumeration, arrays, records, etc.

- Subtypes provide a mechanism for limiting the range of a type
  - Can be used in simulation to do boundary checking
  - Boundary checking in hardware does not exist - subtypes only reduce the number of bits required to describe a signal or variable
  - Consequently, may reduce the number of unnecessary warnings

```
type mem_array is array (integer range 0 to 1023) of std_logic_vector(15 downto 0);
```

creates a new type

```
subtype ROM_MEMORY_RANGE is integer range 0 to 255;
```

modifies an existing type

mem_array becomes a full-blown type which a new signal or variable may be defined as. ROM_MEMORY_RANGE limits the integer (by default, a minimum of 32 bits) to 8 bits.

Simulation tools will often report when a value exceeds the range of a subtype. Synthesis tools will limit the number of bits but simply wrap around if the ROM_MEMORY_RANGE is exceeded.

© Copyright 2019 Xilinx

Slide 5-15:

Subtypes

- Basic syntax: subtype <new subtype name> is <type or subtype name>;
  - Refines the use of a previously defined type or subtype by further limiting its scope

- Example: You need to steer an aircraft in azimuth and elevation (integer resolution is adequate). Define the appropriate subtypes
  - subtype azimuth_range is integer range 0 to 360;
  - subtype elevation_range is integer range -90 to 90;

- Question: Which of the following are legal?

Given:

```
signal azimuth   : azimuth_range;
signal elevation : elevation_range;
```

- azimuth <=45;
- elevation <= -89.9;
- elevation <= 30;
- azimuth <= elevation;

- azimuth <= 45; // VALID
- elevation <= -89.9; // ILLEGAL – real number assigned to subtype of integer
- elevation <= 30; // VALID
- azimuth <= elevation; // ILLEGAL – elevation of different subtype

The syntax provided here covers the most frequently used form of the subtype:

- <new subtype name> - any name in the form of a legal identifier.
- <type or subtype name> - any previously defined type or previously defined subtype.

Example explanation: The azimuth_range subtype limits the range of integer from (minimally) -2^32 to +2^32-1 to 0 to 360. The elevation_range subtype limits the range for elevation to -90 to 90.

© Copyright 2019 Xilinx

From a good coding perspective, it will help identify specific quantities, avoiding certain types of calculation errors, and thus making the code more reliable.

In simulation, this will help uncover errors due to out-of-bound values being assigned to any signal or variable defined by these subtypes. During synthesis, this will help reduce the amount of logic built saving both power and logic resources.

Slide 5-16:

Characters and Strings

- Updated in VHDL-93 to include all items in the ISO 8-bit set
  - Typically, characters are used to create strings
  - Characters types are synthesizable

- Strings are arrays of characters

```
type character is ( nul, sol, stx, etx, eot, enq, ack, bel,
                    . . .
                    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                    . . . ) ;
```

```
constant MY_CHAR: character := 'Q' ;
```

Note that most characters are written in single quotes.

Strings are enclosed within double quotes.

It is important to note that once the string size is fixed (defined), it cannot be changed even if a new string value is smaller than the defined size—the remaining space will contain the remnants of the previous value.

Example for a string -

type String is array (positive range <>) of character;

constant msg: String (1 to 20) := "setup time";

© Copyright 2019 Xilinx

Slide 5-17:

Physical

- Physical types are used to quantify real-world physical concepts and amounts such as mass, length, time
  - Time is the only pre-defined physical type
  - Other physical types may be hand-rolled or explicitly loaded from IEEE libraries
  - Physical types are generally not synthesizable

- A physical type must be defined in terms of its primary unit
  - Any secondary units must be multiples of the primary
  - Units must be of integer type (not real)

```
type time is range -2147483647 to 2147483647
units
fs;
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
. . .
end units ;
```

time is the only pre-defined physical type in VHDL. It can be used to model cell delays and other time-based parameters.

```
constant TPD : time := 3ns ;
. . .
Z <= A after TPD ;
```

Standards in the IEEE also support voltage, current, and a number of other physical types. The IEEE libraries must be explicitly loaded. The specific packages include: ENERGY_SYSTEMS, ELECTRICAL_SYSTEMS, MECHANICAL_SYSTEMS, THERMAL_SYSTEMS, FLUIDIC_SYSTEMS, and RADIANT_SYSTEMS. These packages may not have yet been ratified.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 5-18:

Enumerated Types

- Offers the most flexibility in abstract hardware modeling
  - User-defined enumerated types use values that are immediately recognizable and intuitively relevant to the operation of the model
  - Makes code more readable when describing state machines and complex systems

- Syntax: type <new type name> is (<list of items>);

```
type MY_STATE is ( RST, LOAD, FETCH, STOR, SHIFT ) ;
. . .
signal STATE, NEXT_STATE : MY_STATE;
. . .
```

```
case (STATE) is
  when LOAD => . . .
    if COND_A and COND_B then
      NEXT_STATE <= FETCH ;
    else
      NEXT_STATE <= STOR ;
          . . .
```

Syntax:

- <new type name> - any legal identifier.
- <list of items> - a comma separated list of items having the form of any legal identifier.

By default, enumerated values are sequentially encoded from left to right within the parentheses. In this example, this would result in type MY_STATE having values of “000”, “001”, “010”, “011”, “100” for RST, LOAD, FETCH, STOR, and SHIFT, respectively.

Different synthesis tools may apply a different encoding scheme based on technology-specific optimization algorithms or other proprietary factors.

Although VHDL is a case-insensitive language, when enumerated types are enclosed in single quotes, as they are in std_logic, they ARE case sensitive.

© Copyright 2019 Xilinx

Slide 5-19:

Enumerated Types – Caution!

- When is VHDL case sensitive?
  - Only when using Enumerated type that are enclosed by *single quotes*
  - Most common example: std_logic/std_logic_vector

- Example: Define states for two state machines—a UART transmitter and receiver
  - Consider

```
type rx_states is (IDLE, START, DATA, PARITY, STOP);
type tx_states is (IDLE, START, DATA, PARITY, STOP);
signal rx_state : rx_states := IDLE;
signal tx_state : tx_states := IDLE;
```

  - **Caution:** Although the state names are identical, the following is illegal
    - rx_state <= tx_state;
    - Why is this?

Even though the state names are identical, “IDLE” as defined in rx_states is not the same as “IDLE” as defined in tx_states. These states may be coded differently during synthesis, but more to the point, they are considered to be completely different types.

For ease of understanding, consider the states list in rx_states to be rx_states.IDLE, rx_states.START, etc. Similarly, consider the states in tx_states to be tx_states.IDLE, tx_states.START, etc.

Now it becomes clear that rx_state has a completely different set of values than tx_state.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 5-20:

Summary

- VHDL is a strongly typed language
  - VHDL supports a wide range of data types
- std_logic is the "resolved" form of std_ulogic and it is more commonly used
- Three types of conversion processes
  - "casting", conversion functions, and ‘image/‘value attributes
- VHDL supports enumerated data type, which makes code more readable when describing state machines and complex systems

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Composite Data Types

Slide 6-1:

2019.1

Data types form one of the cornerstones of any language. In this module, the composite data types (arrays and records) are covered. Basic operations and initialization techniques for these data types are also included. 15 minutes.

Slide 6-2:

Objectives

After completing this module, you will be able to:

- Describe what composite data types are
- Create composite data types (array and record)
- Declare one-dimensional and two-dimensional arrays

© Copyright 2019 Xilinx

Slide 6-3:

Composite Data Types

- Composite data types are groups of elements in the form of an array or record
  - bit_vector, std_logic_vector, and string are all pre-defined composite types

```
signal A_WORD : std_logic_vector (3 downto 0) := "0011" ;
```

- This creates four elements of type std_logic grouped together in an array
  - There is no pre-defined LSB or MSB interpretation; therefore, this value is not automatically read by the compiler as 3
  - Note the use of double quotes ("0011") for any bit_vector, std_logic_vector, or string object and the use of single quotes ('1') for bit, std_logic, and character

VHDL does not infer that the higher index is the MSB (3 in this example). VHDL simply indexes the vector from left to right.

Choosing and using one indexing method consistently is good design practice—for example, '(3 downto 0)' or '(0 to 3)'. The most popular is the '(3 downto 0)' method because this is similar to the schematic bus representation My_BusA[3:0].

However, VHDL does infer the MSB/LSB for the subtypes signed and unsigned. Although these are special versions of the std_logic_vector, the arithmetic subprograms that take in those subtypes are written such that the MSB is considered and treated as a sign bit.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 6-4:

STD_LOGIC_VECTOR and SIGNED/UNSIGNED

- STD_LOGIC_VECTOR is an array of type STD_LOGIC
  - Defined in STD_LOGIC_1164
  - Part of the IEEE library
- STD_LOGIC_VECTORs have no positional meaning
  - "0110" is NOT 6, it's just "0110"
- SIGNED and UNSIGNED are also arrays of type STD_LOGIC
  - These DO have positional meanings
  - Leftmost bit is the sign bit when using SIGNED
  - "0110" is interpreted as 6
- Use UNSIGNED when the vector needs math
  - Such as adding 2 BCD digits
- Use STD_LOGIC_VECTOR when only a pattern is needed
  - Such as driving LED segments - the value means nothing

```
entity BCD_add_and_disply is
    port ( clk          : in  STD_LOGIC;
           BCD_A_in     : in  UNSIGNED (3 downto 0);
           BCD_B_in     : in  UNSIGNED (3 downto 0);
           LED_7seg_out : out STD_LOGIC_VECTOR (6 downto 0)
          );
end entity BCD_add_and_disply;
```


© Copyright 2019 Xilinx

Slide 6-5:

What CAN and CANNOT Be Done with std_logic

- std_logic can be used in logical operations
  - Not – inversion
    - sl_A <= not sl_B;
  - Two-input logic (AND, NAND, OR, NOR, XOR, XNOR)
    - sl_C <= sl_A XNOR sl_B;
  - Comparisons of equality and inequality*
    - If (sl_A = sl_B) then …

- Math cannot be performed on std_logic/std_logic_vectors
  - std_logic_vectors hold patterns whose positions are irrelevant

*Recall that std_logic has nine different potential values. Comparing std_logic will compare the position of the enumerated type against the position of the other item's enumerated type. Most likely, this expected outcome will not be realized, therefore, it only makes sense to test for equality or inequality.

NUMERIC_STD library contains appropriate overloading functions to support the various comparisons and shifting functions. This is not part of the VHDL language per se; however, it is common practice to use this *standardized* library, hence its inclusion here.

The signed and unsigned are special cases of std_logic_vector.

© Copyright 2019 Xilinx

Slide 6-6:

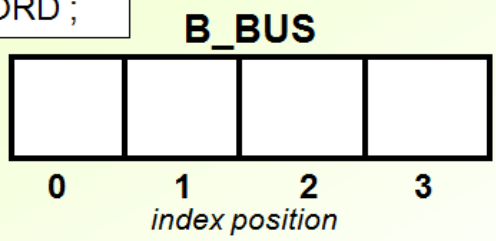
Arrays

- Arrays are groups of elements, all of the same type
- Syntax: type <new type name> is array (<range>) of <data type>;

Example 1:

type WORD *is* ***array (0 to 3) of*** *std_logic ;*

signal **B_BUS** : WORD ;



What are the possible values for each element of the array in each case?

Example 2:

type DATA *is array (0 to 3)* ***of integer range 0 to 9*** *;*

signal **B_BUS** : DATA ;

If B_BUS is of type WORD, the possible values for each element are ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, or ‘-’. Therefore, there are 9^4 (6561) possibilities versus 16 if the type were “bit”.

When declaring arrays of integers, remember to specify the integer range to avoid the default (minimum) 32-bit implementation.

The <new type name> is any valid identifier.

The <range> specifies the minimum and maximum values of the indices via the “to” or “downto” notation. Typically arrays are specified in an upward (to) direction, whereas ranges for std_logic_vectors are usually specified in the downward direction.

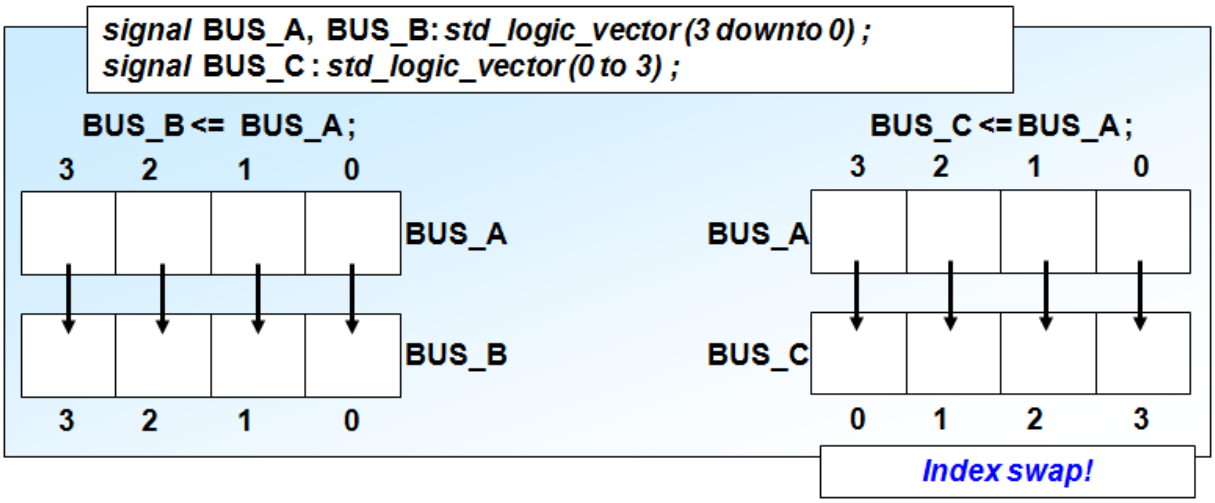
The <data type> may be an “intrinsic” data type such as integer or bit, an extension provided by a library such as std_logic, or a user-defined type.

© Copyright 2019 Xilinx

Slide 6-7:

Array Assignments

- When assigning one array to another
  1. The arrays must be the same type
  2. The arrays must be the same length
  3. The assignment is positional, from left to right



Positional assignment means that the arrays are lined up as they were declared (either '(3 downto 0)' or '(0 to 3)'). The positions in the array are then matched and the assignment is made. This causes a bit swap, although the data is still aligned from left to right, as shown in the second example. While there is nothing wrong with bit swapping in this fashion, it should be done deliberately, not as the result of an accidental change in index ordering.

The best way to avoid this problem is to use a consistent convention when declaring arrays.

Note that the convention to notate vectors typically uses downto so that during a conversion the left-most bits appear to be the most significant. When you are defining an array, the opposite is true, and the to notation is typically used.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 6-8:

Array Assignment Notation

- To simplify array assignments—and enhance readability—you can designate a hexadecimal or octal base
  - Underscores can also be used to further enhance readability

Example 1:

```
signal DATA_WORD : std_logic_vector (11 downto 0) ;
```

```
DATA_WORD <= X"A6F" ;
DATA_WORD <= "101001101111" ;
DATA_WORD <= O"5157" ;
DATA_WORD <= B"1010_0110_1111" ;
```

Example 2:

```
signal DATA_WORD : std_logic_vector (10 downto 0) ;
```

```
DATA_WORD <= B"100_0110_1111" ;
DATA_WORD <= X"46F" ;
```

ERROR!

Most compilers allow underscores to enhance readability. Vivado® synthesis only allows underscores when the base is explicitly defined via hex, octal, or binary notation, as shown here.

Notice that decimal is not listed here. Decimal does not align to power of two boundaries, which is what is required here. If the vector is not a convenient power of 2 using 3 (for octal) or 4 (for hex), hex or octal notation may not be used.

For example if the vector were 6 downto 0, then the only way to represent this is in binary or concatenate a single octal and single hex character together.

Example 2 illustrates that the size of the vector must be an integer multiple of the base. In order to use the hexadecimal base, the length of the vector must be evenly divisible by 4. Octal is the same, except that the length of the vector must be evenly divisible by 3. So, even though there is a '0' in the leftmost position in X"46F", it is too long by 1 std_logic.

© Copyright 2019 Xilinx

Slide 6-9:

Array Slices

- Any group of contiguous elements within an array can be referenced as a slice
  - The remaining elements are unaffected by the assignment

```
signal A_VEC, B_VEC : std_logic_vector (7 downto 0) ;
signal Z_VEC : std_logic_vector (15 downto 0) ;
signal A_BUT, B_BUT, C_BIT, D_BIT : std_logic ;
```

```
Z_VEC (15 downto 8) <= A_VEC ;
B_VEC <= Z_VEC (12 downto 5) ;
A_VEC (1 downto 0) <= C_BIT & D_BIT ;
    . . .
Z_VEC (5 downto 1) <= B_VEC (1 to 5 ) ; ERROR
```

The direction (ascending or descending) of the slice must be consistent with the direction of the array as it was originally declared

The last example is wrong because the vector B_VEC was defined in descending order (7 down to 0). It cannot be “sliced” in ascending order (1 to 5).

© Copyright 2019 Xilinx

Slide 6-10:

Signals and Ports with Example

- Data type and width must match on signal and port assignments

```
entity REG_4 is
   port (reset : in  std_logic;
         clk   : in  std_logic;
         d_in  : in  std_logic_vector (3 downto 0);
         cntrl : in  std_logic_vector (1 downto 0);
         q     : out std_logic_vector (3 downto 0)
        );
end entity REG_4;
```

```
signal A : std_logic;
signal B : bit;
signal C : std_logic;
signal D : integer;
signal E : std_logic_vector(3 downto 0);
```

A <= C;	GOOD
A <= cntrl(1);	GOOD
E <= d_in + 1;	DEPENDS
A <= B;	ERROR
D <= E;	ERROR
B <= D;	ERROR
Q <= cntrl;	ERROR

A <= C – valid since both A and C are of type std_logic.

A <= cntrl(1) – This notation pulls out a single bit (second from the right) and assigns it to A. Because cntrl is an array of std_logic types, one element from this array is a std_logic hence this is a valid assignment.

E <= C + 1 is strictly an error as C is a std_logic type and 1 is an integer. There is, however, a library (IEEE.STD_LOGIC_UNSIGNED) that is frequently used which contains an overload function that converts the std_logic_vector into an integer, performs the operation, and returns a std_logic_vector.

A <= B – A can represent all the information specified in B (and 7 more values on top of that). But because they are not exactly equal, this is an invalid assignment. A simple function already exists in the std_logic_1164 library which will perform this conversion.

D <= E – Although E may contain a pattern which represents an integer, it is just that—a pattern without meaning. E must be converted to an integer (which gives positional meaning to the pattern) in order to be assigned to an integer.

© Copyright 2019 Xilinx

B <= D – A conversion function is required to make this assignment because an integer (with many possible values) is being converted to a bit. How is this conversion to be done? Is only the least significant digit converted to a '1' or '0', or is any non-zero value converted to a '1' – the ambiguity is blatant; hence, this assignment is invalid.

Q <= cntrl - Even though Q and CNTRL are both of type std_logic_vector, Q is 4-bits wide and CNTRL is only 2-bits wide; therefore the error is in size, not type.

It is also worth noting that D is an unbounded integer. This means that the synthesis or simulation tools will typically set the range of the integer to ±2^32 bits (technically 2^32-1 down to -2^32). This is a minimum implementation required by the language; however, this range can be larger. In general, this is a poor practice.

During simulation the range is unchecked which invites an undetected error (that is, if the valid range of D was 0 to 1023 and the D were set to -3 or 1024 then the simulation tools would flag an error and invite the designer to review the code to see if there was a problem with the algorithm that produced this value, or that the range was too limiting).

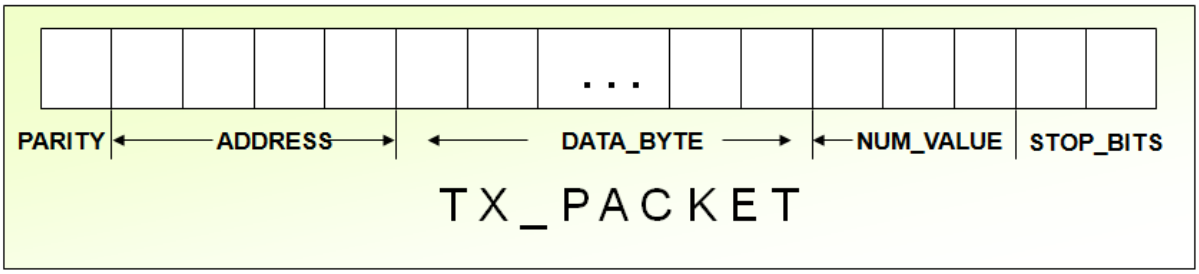
During synthesis, this oversight will likely result in a more fabric (and power) hungry implementation.

Slide 6-11:

Records

```
type PACKET is record
  PARITY      : bit;
  ADDRESS     : std_logic_vector ( 0 to 3 );
  DATA_BYTE   : std_logic_vector ( 7 downto 0 );
  NUM_VALUE   : integer range 0 to 6;
  STOP_BITS   : bit_vector (1 downto 0);
end record ;
. . .
signal  TX_PACKET, RX_PACKET  : PACKET;
```

A ***record*** is a group of elements that may be of *different* types.



© Copyright 2019 Xilinx

To access individual parts of a record, use the <signal_name>.<field name> syntax.

Also, although this example shows using data types bit and std_logic in the same design, this is only done to illustrate that different types can be combined in a single record. Using both is not good coding practice.

For example, the individual parts of the signal TX_PACKET are:

- TX_PACKET.PARITY
- TX_PACKET.ADDRESS
- TX_PACKET.DATA_BYTE
- TX_PACKET.NUM_VALUE
- TX_PACKET.STOP_BITS

Slide 6-12:

Strings

- Strings are arrays of characters
  - Range always positive and non-zero and less than the largest integer
  - Once a string is defined, its size is fixed
  - String constants must match the size exactly
- Strings and characters can be concatenated using the '&' operator

```
constant WARNING1: string ( 1 to 27 ) := "Unexpected Outputs Detected" ;
```

```
write(output," Unexpected Outputs..." );
write(output, WARNING1 & '!');
```

A *string* is a user-defined array of data type character

```
signal str: string (1 to 15) := "too short";
results in the following error:
ERROR:HDLCompiler:410 - "testTypes_tb.vhd" Line 67. Expression
has 9 elements ; expected 15
```

Strings are very helpful during simulation and debugging to display user-defined text, messages, warnings, or errors and are often used with the assert statement.

Note that the Xilinx Simulation tool generates an error if the string size and the declared size are mismatched.

© Copyright 2019 Xilinx

Slide 6-13:

Array Aggregates

- Aggregates are a convenient means for grouping both scalar and composite data types for assignment

```
signal H_BYTE, L_BYTE: std_logic_vector (0 to 7);
signal Q_OUT : std_logic_vector (31 downto 0);
signal A, B, C, D  : std_logic;
signal WORD : std_logic_vector (3 downto 0);
```

```
(A,B,C,D) <= WORD;
WORD  <= ( 2 => '1', 3 => D, others => '0' ) ;
Q_OUT  <=  ( others => '0' ) ;
WORD <=  (A, B, C, D ) ;
H_BYTE  <= ( 7|6|0|1 => '1', 2 to 5 => '0' ) ;
```

Only scalar data variables are allowed on the left-side aggregates

The total number of elements on both sides of any assignment must match; "others" can be used as a default assignment, regardless of the array size

Aggregates are enclosed in parentheses and separated by commas. When making aggregate assignments to bit_vectors or std_logic_vectors, you can assign by index, as shown in the third example above.

When combined with explicit associations, *others* => must be last in the parenthesis.

© Copyright 2019 Xilinx

Slide 6-14:

Record Aggregates

- Aggregates are a convenient means for grouping both scalar and composite data types for assignment

```
type D_WORD is record
      UPPER : std_logic_vector (7 downto 0) ;
      LOWER : std_logic_vector (7 downto 0) ;
end record ;
signal DATA_WORD : D_WORD ;
signal H_BYTE, L_BYTE: std_logic_vector (7 downto 0);
signal TX_PACKET, RX_PACKET : OPCODE; --defined earlier
```

```
DATA_WORD <= ( H_BYTE, L_BYTE) ;
TX_PACKET  <= ( '1','0011','11101010',5,"10" ) ;
DATA_WORD <= ( LOWER => L_BYTE, UPPER=> H_BYTE) ;
TX_PACKET.ADDRESS  <= ( "0011" ) ;
TX_PACKET.ADDRESS(2)  <= '0' ;
DATA_WORD <= ( LOWER | UPPER=> H_BYTE);
DATA_WORD <= ( others => H_BYTE);
```

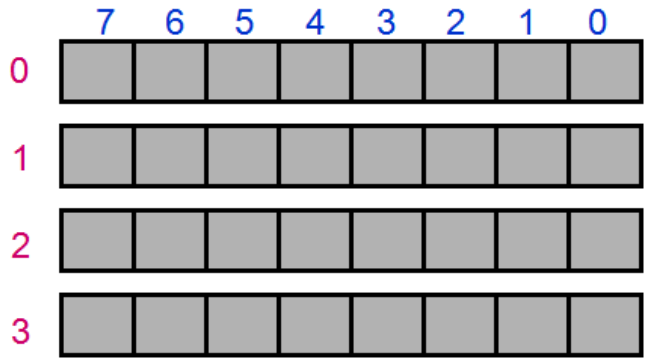
Only records can accept aggregate of arrays

Slide 6-15:

Creating Pseudo 2-D Arrays

- A convenient way of modeling memory structures is creating a pseudo two-dimensional array structure
  - Effectively an array of arrays (or records)
  - Synthesis tools understand this construct as a way to represent memory

```
type MEM_ARRAY is array (0 to 3) of std_logic_vector (7 downto 0);
signal MY_MEM : MEM_ARRAY ;
```



© Copyright 2019 Xilinx

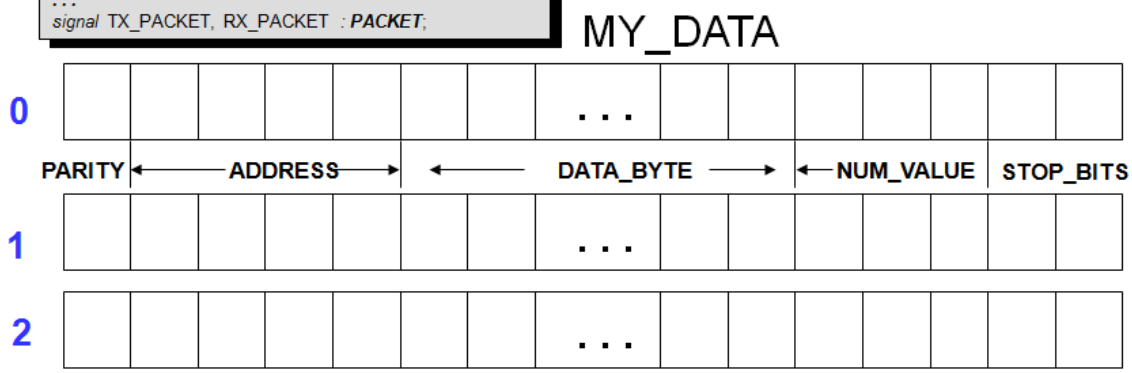
Slide 6-16:

Creating Arrays of Records

- For packet-handling applications, an array of records can be useful
  - The ability to combine different data types affords great flexibility

```
type PACKET is record
  PARITY  : bit;
  ADDRESS : std_logic_vector ( 0 to 3 );
  DATA_BYTE : std_logic_vector ( 7 downto 0 );
  NUM_VALUE : integer range 0 to 6;
  STOP_BITS : bit_vector (1 downto 0);
end record ;
. . .
signal TX_PACKET, RX_PACKET : PACKET;
```

```
type DATA_ARRAY is array ( 0 to 2 ) of PACKET ;
signal MY_DATA : DATA_ARRAY ;
```



type **PACKET** ***is*** ***record***

PARITY : *bit;*

ADDRESS : *std_logic_vector (0 to 3);*

DATA_BYTE : *std_logic_vector (7 downto 0);*

NUM_VALUE : *integer range 0 to 6;*

STOP_BITS : *bit_vector (1 downto 0);*

end record ;

. . .

signal TX_PACKET, RX_PACKET : **PACKET**;

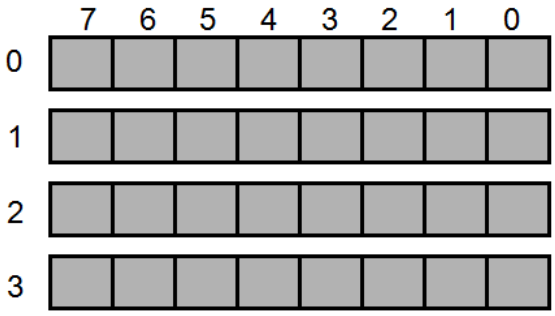
© Copyright 2019 Xilinx

Slide 6-17:

Assigning Values to Arrays

- For most memory applications, the read and write address vector is converted to integer, referencing a row within the array
  - Individual bits may be selected from within this row

```
type MEM_ARRAY is array ( 0 to 3 ) of std_logic_vector ( 7 downto 0 );
signal MY_MEM : MEM_ARRAY ;

signal R_ADDR, W_ADDR : std_logic_vector (1 downto 0 ) ;
```




```
MY_MEM (to_integer(unsigned(W_ADDR)))<= DATA_IN ;
. . .
D_OUT <= MY_MEM (to_integer(unsigned(R_ADDR)));
```

For Xilinx on-chip memory applications, the write address bus is always registered. When inferring block RAM resources, the read address bus is also registered.

© Copyright 2019 Xilinx

Slide 6-18:

Higher Order Arrays

- True multi-dimensional arrays can be constructed by creating a user-defined type of the desired size

```
type  MATRIX_A  is array(1 to 3, 1 to 16)  of  std_logic ;
type  MATRIX_B  is array(1 to 3, 1 to 3, 1 to 3) of integer range -1024 to 1023;
```

- Initialization is performed in the same way as records

```
signal MATRIX_3x16   : MATRIX_A := (others=>(others=>'0'));
signal MATRIX_3x3x3 : MATRIX_B := (others=>(others=>(others=>0)));
```

- Accessing any given point is done with the following notation

```
MATRIX_3x16 (3, 15)   <= '1';
value <= MATRIX_3x3x3(2,1,3);
```

Note the (others=>(others=>'0')) notation. This is the "recursive" form of the aggregate assignment. Of course, the designer may choose to initialize individual rows, or even individual elements. The looping construct, covered later in this course, is another commonly used mechanism for initializing arrays.

Arrays can also be loaded from text or binary files.

© Copyright 2019 Xilinx

Slide 6-19:

Higher Order Array Example

- Simple example to multiply an array by an integer scalar

```
mmult: process
    type  tMATRIX_3x3x3  is array(1 to 3, 1 to 3, 1 to 3) of integer range -1024 to 1023;
    --variable matrix3D : tMATRIX_3x3x3 := (others=>(others=>(others=>0)));
    variable matrix3D : tMATRIX_2x3x4 := ((( 1, 2, 3, 4),( 5, 6, 7, 8),( 9,10,11,12)),
                                          ((13,14,15,16),(17,18,19,20),(21,22,23,24)));
    constant k : integer := 7;
  begin
    -- sparsely populate the matrix
    matrix3D(1,2,3) := 4;   matrix3D(1,3,1) := -10;
    matrix3D(2,2,2) := 100; matrix3D(2,3,4) := -50;

    -- loop through the matrix and multiply each element by a scalar
    rowLoop: for row in 1 to 3 loop
      colLoop: for col in 1 to 3 loop
        dthLoop: for depth in 1 to 3 loop
          report integer'image(row) & "," & integer'image(col) & "," &
                 integer'image(depth) & " * " & integer'image(k) & " = ";
           matrix3D (row,col,depth) := k * matrix3D (row,col,depth);
          report "   " & integer'image(matrix3D (row,col,depth));
        end loop dthLoop;
      end loop colLoop;
    end loop rowLoop;
    wait;
  end process mmult;
```

This type of looping can also be used to initialize a matrix to values that can be described as a function or constant.

Notice that there are two types of initialization demonstrated here: default initialization, which is shown here as the commented out “initialize all elements to 0” and the explicitly initialized form. Notice that the rightmost index is “deepest” in the initialization sequence.

Matrices can also be initialized element by element as shown beneath the comment “sparsely populate matrix”.

The looping structure can also be used to initialize a matrix if it is easier to express the positions within the matrix as constants or as a function.

A more generalized version which can automatically handle changes in the matrix size without having to manually adjust the loop ranges is provided in the Attributes module.

The ***report*** statement is essentially a “print to console” command. Details on this statement are provided later in the course. This statement was used here so that the result of the operation was visible on the simulation console and is provided below.

© Copyright 2019 Xilinx

The (formatted and modified) output from the simulator when running this example is:

```
Note: 1,1,1 * 7 =
Note:       7
Note: 1,1,2 * 7 =
Note:       14
Note: 1,1,3 * 7 =
Note:       21
Note: 1,1,4 * 7 =
Note:       28
Note: 1,2,1 * 7 =
Note:       35
Note: 1,2,2 * 7 =
Note:       42
Note: 1,2,3 * 7 =
Note:       28
Note: 1,2,4 * 7 =
Note:       56
Note: 1,3,1 * 7 =
Note:       -70
Note: 1,3,2 * 7 =
Note:       70
Note: 1,3,3 * 7 =
Note:       77
Note: 1,3,4 * 7 =
Note:       84
Note: 2,1,1 * 7 =
Note:       91
Note: 2,1,2 * 7 =
Note:       98
```


© Copyright 2019 Xilinx

Note: 2,1,3 * 7 =

Note: 105

Note: 2,1,4 * 7 =

Note: 112

Note: 2,2,1 * 7 =

Note: 119

Note: 2,2,2 * 7 =

Note: 700

Note: 2,2,3 * 7 =

Note: 133

Note: 2,2,4 * 7 =

Note: 140

Note: 2,3,1 * 7 =

Note: 147

Note: 2,3,2 * 7 =

Note: 154

Note: 2,3,3 * 7 =

Note: 161

Note: 2,3,4 * 7 =

Note: -350

© Copyright 2019 Xilinx

Slide 6-20:

Apply Your Knowledge

1) Write the HDL source code that would produce the memory structure initialized to the values shown
   - Label the structure as MY_ROM2 of type ROM_ARRAY
   - Use named association for the memory base (word) element
   - Use "others" notation wherever possible; consider a single assignment to the last two words
   - Use hexadecimal notation for the first two assignments only

1	0	0	0
1	1	1	0
0	0	0	0
0	0	1	1
1	1	1	1
1	1	0	0
0	0	0	0
0	0	0	0

© Copyright 2019 Xilinx

Slide 6-21:

One Possible Solution

1) Write the HDL source code that would produce the memory structure initialized to the values shown

	3	2	1	0
0	1	0	0	0
1	1	1	1	0
2	0	0	0	0
3	0	0	1	1
4	1	1	1	1
5	1	1	0	0
6	0	0	0	0
7	0	0	0	0

```
type ROM_ARRY is array (0 to 7) of std_logic_vector (3 downto 0 ) ;
constant MY_ROM2 : ROM_ARRY := (
                0 => X"8",
                1 => X"E",
                2 => (others => '0'),
                3 => "0011",
                4 => ( others=>'1'),
                5 => B"11_00",
                others => (others => '0') );
```

Slide 6-22:

Summary

- VHDL provides scalar and composite data types
- Memory structures are often created from arrays of std_logic_vectors
- Multidimensional arrays can be created of any type
- Record is a group of elements with different data types

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

VHDL Operators

Slide 7-1:

2019.1

In this module, all types of VHDL operators are covered. 15 minutes.

Slide 7-2:

Objectives

After completing this module, you will be able to:

- Analyze all the operators that are available in VHDL
- Use these operators in VHDL code

© Copyright 2019 Xilinx

Slide 7-3:

Operators in VHDL

- Seven groups of predefined VHDL operators

Group	VHDL Operator	Return Type
Binary logical	and, or, nand, nor, xor, xnor, not	Same type
Relational	=, /= , < , <=, >, >=	Boolean
Shift	sll, srl, sla, sra, rol, ror	Same type
Adding	+, -, & (concatenation)	Same type
Unary	+ , - , not	Same type
Multiplying	*, /, mod, rem	Same type
Miscellaneous	abs, ** (exponent)	Same type

- These groups are arranged in increasing priority when parentheses are not used

- More operators are available through additional library support

VHDL operators are intrinsically defined for type bit and bit vectors operations only, and type bit and bit vector can only hold values ‘0’ or ‘1’.

Recall from the Data Types section that std_logic can take on one of the following nine possible values {‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W,’ ‘L’, ‘H’, ‘-’}. Included within the IEEE’s std_logic_1164 package are a number of “overloading” functions which operate identically to the table above; however, there is NO support for shifting. Typically, coding constructs are used to shift std_logic and std_logic_vector values.

Return type is normally the same data type of the operands except for the relation operators, which returns a boolean value.

Shift operators apply to the bit_vector type only.

VHDL-2008 supports relational operators that return `std_logic` types.

With the new types, code that needed to be written as:

```
if x = y then
out1 <= ‘1’;
else
out1 <= ‘0’;
end if;
```


© Copyright 2019 Xilinx

Can now be written as:

```
out1 <= x ?=y
```

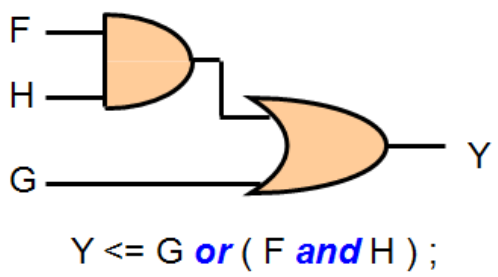
The Vivado® Design Suite supports the following relational operators that return the `std_logic` type.

- ?= (x equal to y)
- ?/= (x not equal to y)
- ?< (x less than y)
- ?<= (x less than or equal to y)
- ?> (x greater than y)
- ?>= (x greater than or equal to y)

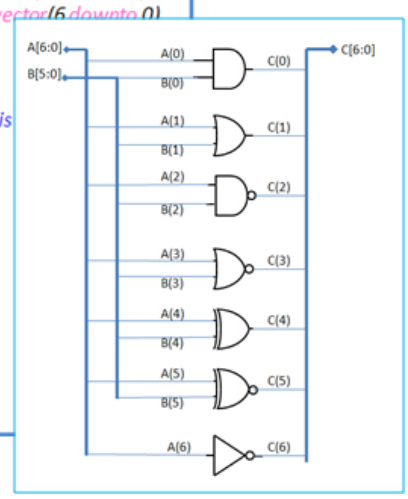
Slide 7-4:

Logical Operators

- Logical operators are predefined for bit, boolean, and bit_vector within VHDL
- std_logic_vector data types are supported in the IEEE.std_logic_1164 library with appropriate overloading language operators




```
entity LOGIC is
  port ( A  :in std_logic_vector(6 downto 0);
         B :in std_logic_vector(5 downto 0);
         C :out std_logic_vector(6 downto 0)
         );
end  LOGIC;

architecture RTL of LOGIC is
begin

C(0) <=A(0) and B(0);
C(1) <=A(1) or B(1);
C(2) <=A(2) nand B(2);
C(3) <=A(3) nor B(3);
C(4) <=A(4) xor B(4);
C(5) <=A(5) nxor B(5);
C(6) <=not A(6);

end RTL
```



Note that the use of parentheses is absolutely essential to specify operator precedence. Only the 'not' operator has an inherently higher precedence among logical operators.

Without parentheses, the second example resolves to G 'or'ed with F, and then 'and'ed with H.

© Copyright 2019 Xilinx

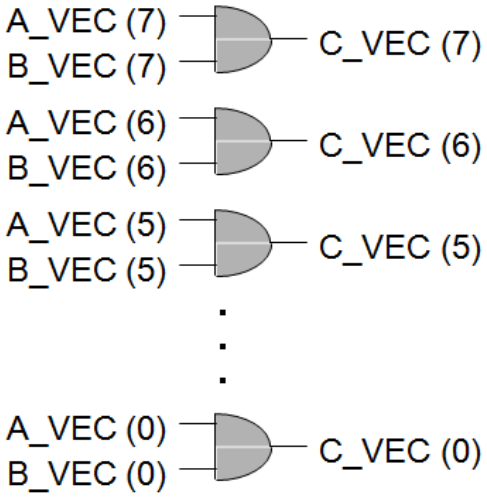
Recall from the previous slide, that std_logic_vector data types are supported in the IEEE.std_1164 library with appropriate overloading language operators.

This enables logical operations to be performed on std_logic_vectors with the addition of std_logic_1164 library.

Slide 7-5:

Logical Operations on Arrays

```
signal A_VEC, B_VEC, C_VEC : bit_vector ( 7 downto 0 ) ;
```




```
C_VEC <= A_VEC and B_VEC ;
```

Rules for use on arrays

1. Arrays must be the same type
2. Arrays must be the same length
3. Operation applied to positional elements within array are left to right

In this example, if A_VEC, B_VEC, or C_VEC had been declared in an ascending fashion—for example, bit_vector (0 to 7)—the assignment would still match the array elements from left to right.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 7-6:

Relational Operators

- Relational operators are used to compare operands
- Return type of the expression is boolean

Symbol	Operation	Number of Operands
=	Equal to	2
/=	Not equal to	2
>	Greater than	2
>=	Greater than or equal to	2
<	Less than	2
<=	Less than or equal to	2

VHDL-2008 has a new set of predefined matching relational operators (“?=”, “?/=”, “?<”, “?<=”, “?>”, and “?>=”) that return bit or std_logic results.

This allows the rewriting the assignment as:

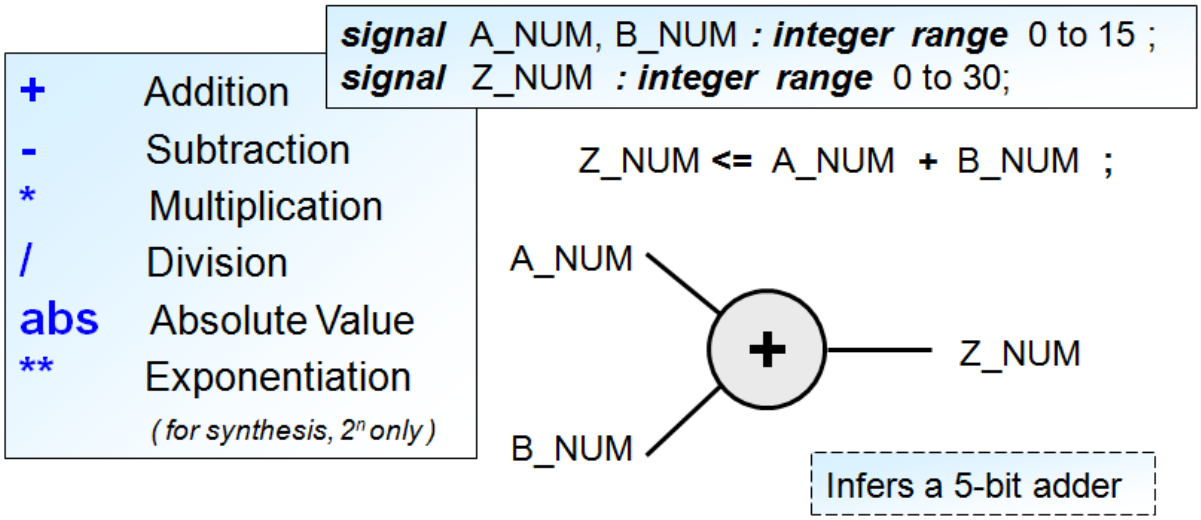
y <= A ?= B;

© Copyright 2019 Xilinx

Slide 7-7:

Arithmetic Operators

- Arithmetic operators are predefined for integer, real, and physical data types
- As noted earlier, arithmetic operators are not predefined for arrays



If bit_vector or std_logic_vector is used, the equivalent expression might be:

Z_Num <= (('0' & A_num) + ('0' & B_num));

where each operand is zero extended by using the concatenation operator so that each array in the expression has the same length.

Note that for division, synthesis support is limited to where the divisor is a constant and a power of 2. In that case, the operator is implemented in a shifter; otherwise, a compiler error message is issued. The following code segment demonstrates this requirement:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity MY_DIVIDE is
   port ( D_IN : in unsigned ( 7 downto 0 ) ; D_OUT : out unsigned ( 7 downto 0));
end entity MY_DIVIDE;
architecture RTL of MY_DIVIDE is
```


© Copyright 2019 Xilinx

```
begin
   D_OUT <= D_IN / 2 ;
end architecture RTL;
```

Slide 7-8:

Arithmetic Operations on Arrays

- Performing arithmetic operations on arrays requires functions supplied by IEEE or the synthesis tool vendor
  - Such functions are normally included in an arithmetic package of some type
    - Package may require compilation into the library
    - Some tools pre-compile arithmetic packages into their own internal library
  - The packages NUMERIC_STD are defined within the IEEE library to enhance source-code portability across tools

```
package NUMERIC_STD is
  constant CopyRightNotice: STRING
      := "Copyright 1995 IEEE. All rights reserved.";
  --============================================================================
  -- Numeric array type definitions
  --============================================================================
  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;


  function "+" (L, R: UNSIGNED) return UNSIGNED;
  function "+" (L, R: SIGNED) return SIGNED;
  function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

- This snippet of code shows how the "+" operator is overloaded so that it may be used with UNSIGNED, SIGNED, etc.

The concept of operator/subprogram overloading is discussed in more detail in Functions topic cluster.

© Copyright 2019 Xilinx

Slide 7-9:

Array Arithmetic

- If the appropriate functions are available and made visible within the module (via the *use* clause), the compiler automatically passes the arguments to the subprogram and returns the result
  - Otherwise, the compiler will inform you that the operators are undefined for the given data types

```
signal  A_VEC : std_logic_vector ( 7 downto 0 ) := "11001001" ;
signal  B_VEC : std_logic_vector ( 7 downto 0 ) := "11100100 " ;
signal  Z_VEC : std_logic_vector ( 8 downto 0 ) ;
signal  D_INT  :  integer  range ( 0 to 9 ) ;
```

```
Z_VEC <= A_VEC + B_VEC ;

Z_VEC <= A_VEC + D_INT ;
```

Slide 7-10:

Miscellaneous Operators - abs, **, rem

- abs operator
  - Returns the absolute value of a signed vector
- ** operator
  - Exponentiation: 2 ** 4 = 16
- rem operator
  - Returns remainder after division operation
  - Result takes the sign as that of the first operand
  - 7 rem 4 = 3
  - -7 rem 4 = -3
  - 7 rem -4 = 3
  - -7 rem -4 = -3

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 7-11:

Shift Operators

- Shift operators are only predefined for the data type *bit_vector*
  - sll - Logical left shift – vacated bit is zero filled
  - srl - Logical right shift – vacated bit is zero filled
  - sla - Arithmetic left shift – vacated bit is copied (sign extension)
  - sra - Arithmetic right shift – vacated bit is copied (sign extension)
  - rol - Rotate left logical – right-most bit is rotated to vacated bit
  - ror - Rotate right logical – right-most bit is rotated to vacated bit
- Separate definitions for data type STD_LOGIC_VECTOR in STD_LOGIC_1164, NUMERIC_STD and NUMERIC_STD_UNSIGNED

```
A<="1001010";
B<=A sll 2;      0101000
B<=A srl 3;      0001001
B<=A sla 2;      0101000
B<=A sra 3;      1111001
B<=A rol 2;      0101010
B<=A ror 3;      0101001
B<=A ror -4;     0101001
B<=A sla -2;     1110010
```

Shift operators were added in the 1993 VHDL language revision. These were later found to be flawed and replaced with functions of the same name.

- Shift operators are defined for data type UNSIGNED and SIGNED in NUMERIC_STD
  - SHIFT_LEFT -- vacated bit is zero filled
  - SHIFT_RIGHT -- vacated position is zero filled for UNSIGNED (logical), MSB filled for SIGNED (arithmetic)
  - ROTATE_LEFT -- right-most position is rotated to vacated position
  - ROTATE_RIGHT -- left-most position is rotated to vacant position

© Copyright 2019 Xilinx

Slide 7-12:

Shift Operator Examples

Given:

```
signal uSource : unsigned(7 downto 0) := X"C6";      -- "11000110"
signal sSource : signed(7 downto 0) := X"C6";
signal uResult : unsigned (7 downto 0) := (others=>'U');
signal sResult : signed (7 downto 0) := (others=>'U');
```

uResult <= SHIFT_RIGHT(uSource,2);	Yields...	X"31" or "00110001"
sResult <= SHIFT_RIGHT(sSource,2);		X"F1" or "11110001"
uResult <= ROTATE_RIGHT(uSource,2);		X"B1" or "10110001"
sResult <= ROTATE_RIGHT(sSource,2);		X"B1" or "10110001"
uResult <= SHIFT_LEFT(uSource,2);		X"18" or "10001000"
sResult <= SHIFT_LEFT(sSource,2);		X"18" or "10001000"

Note that when the shift value is a negative number, this effectively reverses the direction of the operation.

Notice that it does not make sense to sign extend an unsigned number; therefore, SHIFT_RIGHT for unsigned values is a logical shift right and SHIFT_RIGHT for signed values is an arithmetic shift right.

The shift and rotate functions will only shift a positive integer number of positions.

© Copyright 2019 Xilinx

Slide 7-13:

Concatenation

- The concatenation operator "**&**" allows flexible grouping of scalars and arrays into larger arrays
  - Assignments are made via positional association

```
signal  A_VEC, B_VEC : std_logic_vector ( 7 downto 0 ) ;
signal  Z_VEC : std_logic_vector ( 15 downto 0 ) ;
signal  A_BIT, B_BIT, C_BIT, D_BIT : std_logic ;
signal  X_VEC : std_logic_vector ( 2 downto 0 ) ;
signal  Y_VEC : std_logic_vector ( 8 downto 0 ) ;
```

```
Z_VEC  <=  A_VEC & B_VEC ;

X_VEC  <=  A_BIT & B_BIT & C_BIT ;

Y_VEC  <=  B_VEC & D_BIT ;
```

The concatenation operator is not as flexible as the “aggregate” assignment, but it does allow you to group arrays when assigning to another array.

In addition, the concatenation is supported in all tools, whereas the use of aggregates may not be fully supported in some tools.

© Copyright 2019 Xilinx

Slide 7-14:

Shift Register Using Concatenation

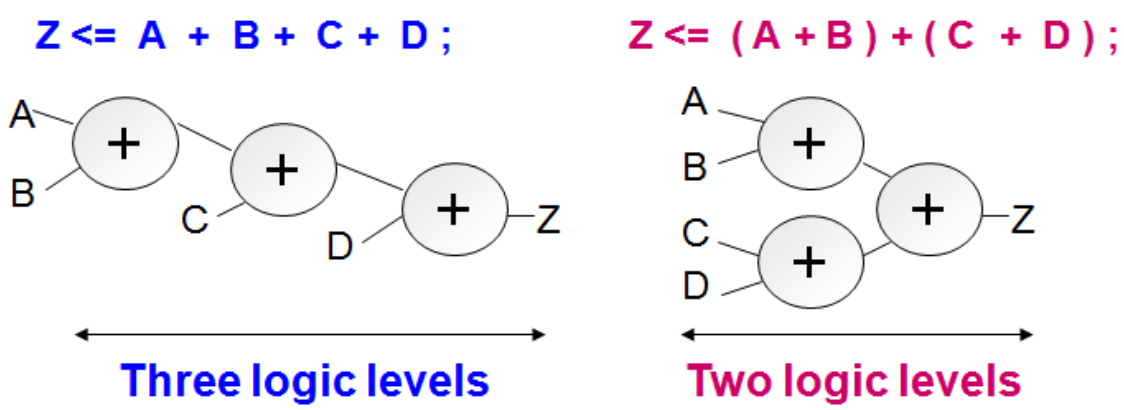
- Concatenation operators can be used to model standard shift registers

```
entity SHIFT_TEST is
   port ( CLK    : in std_logic;
          D_IN   : in std_logic;
          D_OUT  : out std_logic
        ) ;
end SHIFT_TEST;

architecture RTL of SHIFT_TEST is
  signal INT : std_logic_vector ( 7 downto 0 );
begin

 process ( CLK )
   begin
      if ( rising_edge(CLK) ) then
        INT <= INT(6 downto 0) & D_IN ;
     end if;
    end process;
  D_OUT <= INT(7) ;
end architecture RTL ;
```

Slide 7-15:

Grouping Operators

- Using parentheses to group operators in a given expression can help to guide some aspects of logic synthesis while enhancing the readability of the code



This is especially important when the target technology is look-up table (LUT) based. Each added level of logic incurs additional block and routing delays

© Copyright 2019 Xilinx

By default, the elaboration process starts with the target (output) and then parses to the right.

Almost all operators are modeled as two-input functions (that is, two operands around a given operator). This leads to the three levels. Most FPGA synthesis tools will attempt to optimize this, but using parentheses directly controls the implementation and enhances source-code portability.

In the given example, the optimization of the logic levels will occur at the elaboration stage. In the synthesis stage, the optimization may occur, based on the number and the size of inputs.

Slide 7-16:

New Conditional Operator

- VHDL-2008 provides a new conditional operator, "**??**"
- Allows treating an expression to produce a bit or std_ulogic value as a condition
  - Converts from a bit or std_logic value to a boolean value
- Conditional operator converts '1' and 'H' to to true and rest all to false
- Example
  - *if ?? A and B then …*

Slide 7-17:

Operator Precedence

- **, abs, not
- Multiply, divide, modulus (* / %), rem
- Unary – including reduction (+ - &)
- Shift (sll, srl, sla, sra, rol, ror)
- Relational (< <= > >=)
- Equality (= /=)
- Logical (and, or, nand, nor, xor, xnor)

Always use parentheses.

Proper use of parentheses also enhances readability and guides synthesis.

© Copyright 2019 Xilinx

Slide 7-18:

Summary

- VHDL provides different operators: Logical, arithmetic, relational, shift, concatenation operator, etc.
- Operator precedence is also discussed

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Concurrency in VHDL

Slide 8-1:

2019.1

Concurrent statements form the basis for the RTL coding style. In this module, you will see what signals are and how they help in achieving the concurrency. 15 minutes.

Slide 8-2:

Objectives

After completing this module, you will be able to:

- Describe what a signal is and how it behaves in a concurrent use
- Describe how to define a constant and how it behaves
- Define the event and transaction
- Define the concept of delta cycles

Slide 8-3:

Concurrency and Delta Cycles

- VHDL was originally designed as a documentation and simulation language
  - FPGAs were not even invented at the time VHDL came into being
  - Synthesizing VHDL for FPGAs was almost an afterthought
- Concurrent behavior modeling had to be performed on a sequential machine (computer)
  - Required some creative virtual time management
  - Delta cycles are "timeless" intervals that do not exist in real time but are necessary in simulation

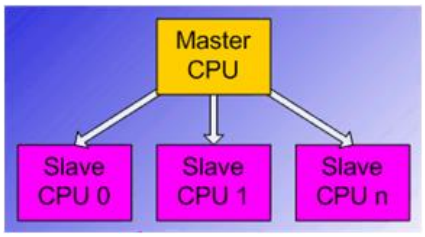
© Copyright 2019 Xilinx

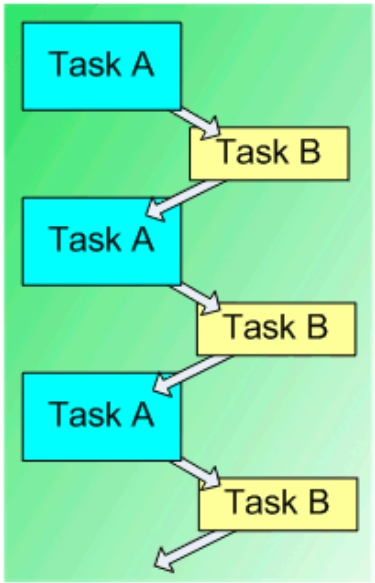
Slide 8-4:

What is Concurrency?

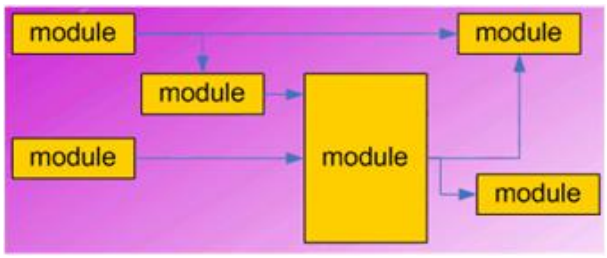
- Concurrency is the "execution" of multiple tasks simultaneously

- Software view
  - Software describes a path through a fixed architecture
  - Multiple threads - *appearance* of concurrency
  - Multiple processors - actual concurrency





- FPGA view
  - HDL describes the architecture itself
    - The is no "execution" per se
  - The fabric of the FPGA can be configured to build multiple, simultaneously running tasks



Multi-threading: This is the process of making it appear as through multiple tasks are running concurrently. Small amounts of time are relegated to various tasks in a cyclical fashion so that the “thread’s” response to a user (or other application) appears to be running simultaneously and without significant latency.

Important ramifications:

- If a task requires 1 minute to complete in a system where no other tasks (including operating system) are present, then it will take just over 2 minutes to run with another task of equal priority.

© Copyright 2019 Xilinx

- The reason it is “just over” instead of “exactly” is that there is overhead incurred by the operating system to “swap” the tasks. (Hence, the slight vertical separation between Task A and Task B). The smaller the time allocation per task, the higher the overhead penalty of swapping tasks. There is a point at which the duration in which the task is allowed to run approaches the time it takes to swap tasks. This condition is known as “thrashing” and brings the system to a non-operating state.
- A multiple CPU system is generally run in one of two ways: either a master CPU splits a single thread and distributes pieces of the application to the various slave CPUs and is responsible for coordinating data movement (typically seen in some “super” computers), or the master CPU manages the OS and coordinates the individual tasks running on each slave CPU.

Regardless of which method is used, the architecture is FIXED and cannot be changed, all that changes is the path (as controlled by the software) through the fixed architecture.

Keep in mind that software “runs” on a processor. HDL is the program and architecture.

To be technically correct, the execution of HDL is simulation. After synthesis there is no more HDL—just a netlist, so there is no run-time state, just the resultant hardware.

This can be a confusing point for software engineers. A loop in software runs over time. A loop in hardware produces gates.

Consider an old-fashion 7400 series logic circuit. Just because the critical data flow may not be through one chip (perhaps it is disabled), the chip does not cease to exist while it is not used — it is always there.

The same is true for the synthesized HDL—gates do not spontaneously burst into existence when needed and blink out when unused (a discussion of partial reconfiguration is outside the scope of this class).

© Copyright 2019 Xilinx

Slide 8-5:

Signals

- Signals give concurrent behavior to VHDL as the assignment is scheduled and assigned at the end of current simulation time

- Definition/declaration in
  - Components - between the **architecture** statement and the **begin** statement

- Syntax

```
signal <signal name> : <signal type> [ := <initial value(s)>] ;
```

- Examples
  - Signal uninit : real;
  - Signal xyz : std_logic_vector(3 downto 0) := (others=>'U');
  - Signal iValue : integer range 0 to 90 := 0;
  - Signal packet : examplePacket := (others=>(others=>'0'));

Syntax:

- <signal name> - any legal identifier.
- <signal type> - any previously defined type, whether intrinsic to the language, defined in a library, or user created.
- <initial value(s)> - the value that the signal is set to upon first power up. Check with the manufacturer of the synthesis tools to verify that this is supported.

Important: The initial value(s) are applied upon the FPGA coming out of configuration, NOT upon reset.

Best design practice:

- Because initial values only exist when the FPGA comes out of configuration, a good practice is to set these values to "undefined" or another easily recognizable value.
- During simulation, these undefined values will persist and enable the designer to validate the behavior of the reset mechanisms. The value of the undefined usually synthesizes to 0.

© Copyright 2019 Xilinx

Explanation of examples:

- uninit is defined as a real over the full set of values that real supports and its value is unknown at the beginning of simulation.
- xyz is defined as a std_logic_vector of four elements in width, each element being set to ‘U’ upon reset in simulation or (likely, but not guaranteed) ‘0’ in synthesis.
- iValue is defined as an integer between 0 and 90 inclusive and is initialized to 0.
- packet is defined to be an examplePacket. There are several assumptions that go along with this example. The first is that examplePacket is defined in one of the user’s libraries, packages, or declaration region of the module that this statement is contained in. The other assumption is that every element within examplePacket is a std_logic_vector.

Slide 8-6:

Signal Assignment

- Both conditional and unconditional signal assignments use the "<=" operator
  - This is used to distinguish between a test for equality ‘=’ and variable assignment " := " operators
- Unconditional signal assignments are used to
  - Make simple connections
  - Perform logical or arithmetic operations
- Left-hand side type and size must match the result on the right-hand side

```
    signal a, b, d : std_logic_vector(15 downto 0);
    signal c : integer range 0 to 65535;
begin
    c <= to_integer(unsigned(a) +  unsigned(b));
    d <= a and b;
```

  - Use type casting and conversion functions as necessary

Recall that the IEEE.STD_LOGIC_1164 library contains the definitions for std_logic and std_logic_vector, overloading functions so that basic logic functions, conversions to bit/bit_vectors, and resolution functions can be applied to these types.

IEEE.NUMERIC_STD extends the capabilities to include the signed/unsigned types, conversion functions, and basic mathematical operations.

© Copyright 2019 Xilinx

Extra points given to the student that recognizes that when two 16-bit numbers are added together, it could produce a 17-bit result. The solution to this is to increase the range of c to 131070.

Slide 8-7:

Constants

- Definition/declaration
  - Constants are declared between the "architecture" line and the "begin" statement
  - Constants can also be declared within the entity block or within a process
- Syntax

```
constant <constant name> : <constant type> [ := <value(s)> ] ;
```

- Examples
  - Constant MAX_VELOCITY : real := 65.5;
  - Constant always_low : std_logic_vector(3 downto 0) := (others=>'0');
  - Constant X_SCALE : integer := 2;
  - Constant packet : examplePacket := (field1=>3, field2=>X"35",(others=>'0'));

Syntax:

- <constant name> - any legal identifier.
- <constant type> - any previously defined type, whether intrinsic to the language, defined in a library, or user created.
- <value(s)> - the value that the constant is permanently set to.

Best design practice:

- Many designers and software programmers distinguish between constants and signals or variables by using all capital letters for constants. As VHDL is case insensitive, this is strictly an ease of reading concern.
- Many companies have established coding style rules. It usually does not matter what they are as long as everyone follows them.

Sometimes constants are declared in the package without initialization and value is assigned to these constants in the package body. These type of constants are deferred constants.

© Copyright 2019 Xilinx

```
package deferred is
        constant num : integer;
end deferred;
package body deferred is
        constant num : integer := 10;
end deferred
```

Slide 8-8:

Transactions

- The signal assignment statement Z <= A causes a transaction to be scheduled

```
process ( A, B, G, M )
begin
        Z <= A or B ;
        F <= G and M ;
          . . .
end process ;
```

- Specifically, the current value of A is read and or-ed with the current value of B, then scheduled to be driven onto Z when the process suspends

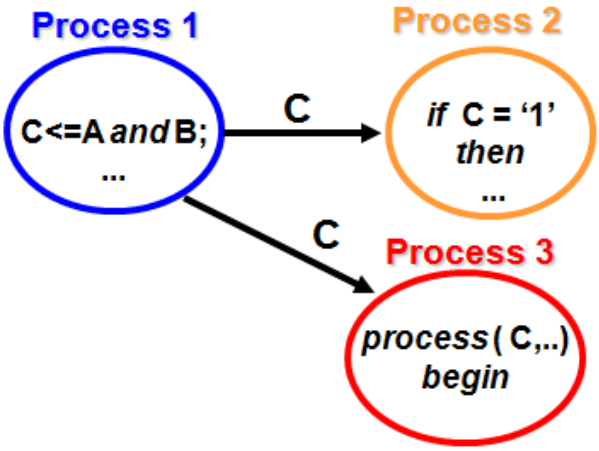
© Copyright 2019 Xilinx

Slide 8-9:

Events

- If the value of C or F is actually changed as a result of the assignment (transaction), an event occurs on that signal
- All signal assignments cause a transaction to be scheduled, but not every transaction results in an event on the target signal
- Only an event on a given signal will cause a process to trigger if that signal is included in its sensitivity list

```
process ( A, B, G, M )
begin
        C <= A or B ;
        F <= G and M ;
          . . .
end process ;
```



© Copyright 2019 Xilinx

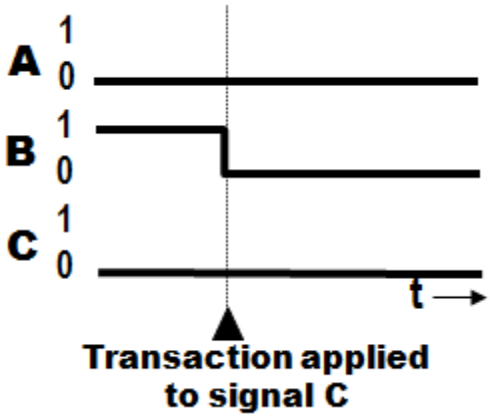
Slide 8-10:

Transaction versus Event

- Although every signal assignment within a process causes a scheduled transaction, not every transaction leads to an event on that signal

```
process (A, B)
   begin
         C <= A and B ;
. . .
end process ;
```

- For example, in a 2-input and gate
  - 1. Signal A is already low
  - 2. Signal B goes low
  - 3. Process triggers on signal B event
  - 4. Expression re-evaluates
  - 5. Transaction scheduled, logic 0 on C
  - 6. Process suspends
  - 7. Transaction applied to C
  - 8. Value of node C does not change
  - 9. Summary, transaction on C, no event



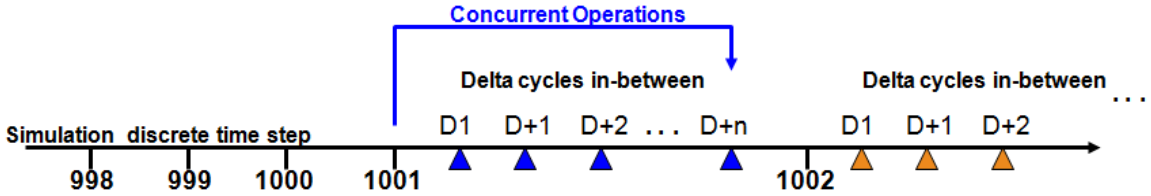
Transaction is basically signal assignment, whereas the event is nothing but a transaction where signal value changes.

© Copyright 2019 Xilinx

Slide 8-11:

Delta Cycles

- A VHDL simulator is event-driven
  - At any single point of discrete simulation time
    - 1. All processes execute until they suspend
    - 2. Signals are updated
    - 3. Events on those signals cause more processes to resume execution
- This is referred to as a delta cycle
- In this manner, processes and other concurrent operations are seen to take place at the same point in discrete simulation time



For simulation, describe how the value of a signal is updated as the result of an assignment:

1) When a signal in the sensitivity list changes (event), it triggers the process to be evaluated.
2) Each assigned signal has a transaction scheduled.
3) At the finish of the delta cycle (when the process suspends), each transaction occurs, causing an event if the value changes.

Slide 8-12:

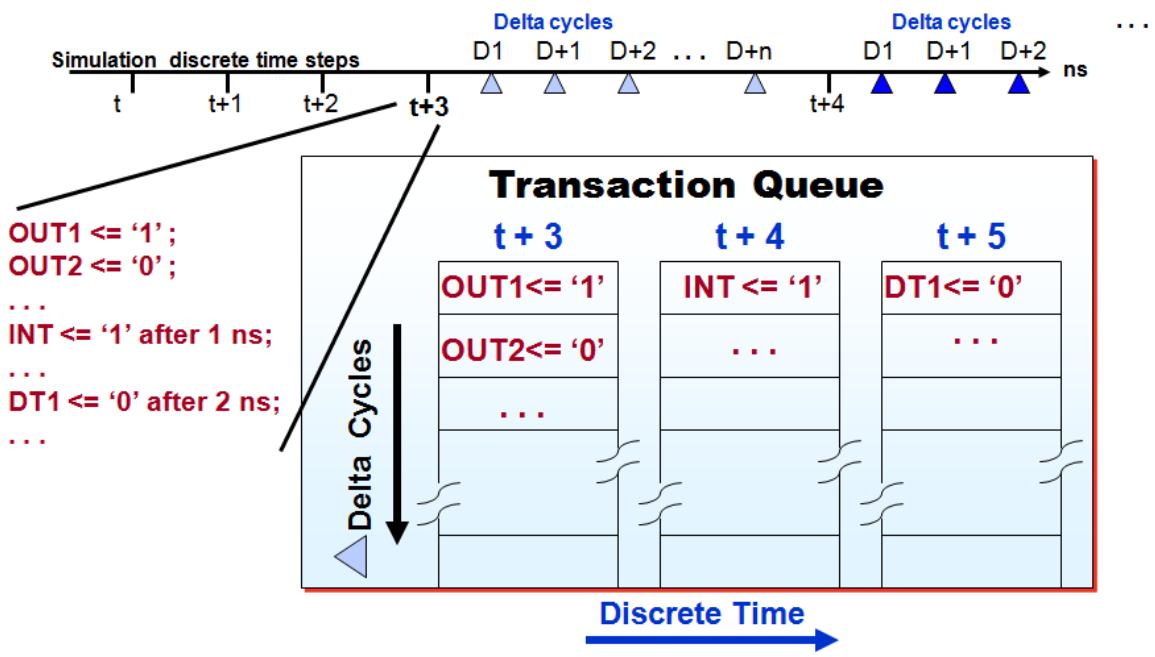
Scheduling Events

- The event scheduler is the heart of the HDL behavioral environment
- Each transaction is scheduled at its appropriate discrete time
- Discrete time advances only when no more transactions are scheduled at the current time

© Copyright 2019 Xilinx

Slide 8-13:

Scheduling Events Example



Slide 8-14:

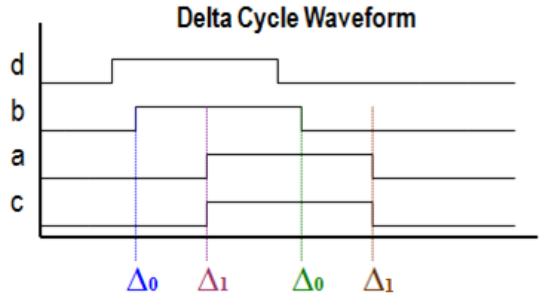
Modeling Concurrency

```
process (b, d)
begin
  a <= b;
  b <= d;
end process;

process (b)
begin
  c <= b;
end process;
```

Delta Cycles and Current Value - Next Value Queue

d=0	a		b		c	
	CV	NV	CV	NV	CV	NV
d=1, delta 0	0		0	1	0	
	0		1		0	
b=1, d=1, delta 1	0	1	1		0	1
	1		1		1	
d=0, delta 0	1		1	0	1	
	1		0		1	
b=0, d=0, delta 1	1	0	0		1	0
	0		0		0	

Delta Cycle Waveform



© Copyright 2019 Xilinx

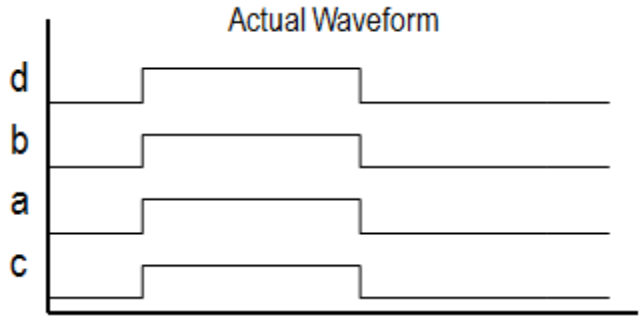
Describe how concurrency is modeled:

1) At time T, an event on a signal that is in a sensitivity list creates a delta cycle.
2) At the completion of the delta cycle, all scheduled transactions on signals occur.
3) At time T, again, if an event occurs on a signal in a sensitivity list, it creates a new delta cycle (time has not changed).
4) Steps 2 and 3 are repeated until there are no more events that can cause a delta cycle at time T.
5) Time moves forward to the next scheduled transaction.

For example:

clk <= not clk after 10 ns;

Every 10 ns, a new delta cycle is created.



Slide 8-15:

Apply Your Knowledge

1) True or false: Signals defined in the entity are visible only in the corresponding architecture

2) True or false: Signals defined in the architecture are only visible within that architecture

3) How are signals synthesized; that is, what do they become?

4) If you needed to send a string to an output device based on the status of one of four buttons being pressed, how might you code this?

© Copyright 2019 Xilinx

A

1) True or false: Signals defined in the entity are visible only in the corresponding architecture.

True. Only the "in" signals within the entity are "seen" by the architecture. Conversely, only the "out" signals within the entity can be "seen" by a higher level module.

2) True or false: Signals defined in the architecture are only visible within that architecture.

True. If the designer wants to make a signal inside the architecture visible outside the module, the signal must be placed in the port statement within the entity. There are no mechanisms to circumvent the hierarchy.

3) How are signals synthesized; that is, what do they become?

It depends how they are used. If they are used for connection (renaming) or combinatorial logic (including arithmetic), then they become wires connecting one LUT to another.

It is interesting to note that some signals will be absent from the netlist (optimized away) as logic is collapsed into a more concise or efficient form. There are mechanisms available to force the preservation of these signals; however, this runs the risk of producing less optimized logic.

If a signal is associated with a clock, then a register can be produced. As with the previous case, there is always the risk that a signal may be optimized away to improve performance or real-estate usage.

Be certain to read the synthesis report because it indicates what signals have been removed. One unintentional source of signal removal is an unused output. Typically, synthesis engines will work from the outputs to the inputs and if there are any inputs or logic connected to inputs that are not used to produce an output, it will be removed.

4) If you needed to send a string to an output device based on the status of one of four buttons being pressed, how might you code this?

Because there is only a single button pressed, a good way to code this would be to use a with/select construct.

For example:

Assume buttonPressed is a std_logic_vector of width 4 and message is a string of length 8.

buttonPressed <= signalFromButton4 & signalFromButton3 & signalFromButton2 & signalFromButton1;

© Copyright 2019 Xilinx

```
with (buttonPressed) select
    message <= “button 1” when(1),
                                “button 2” when(2),
                                “button 3” when(4),
                                “button 4” when(8),
                                “—none—” when others;
```

Slide 8-16:

Summary

- Concurrency is executing multiple tasks simultaneously
- Signals give concurrent behavior to VHDL code
- Event and transaction
  - Transaction is nothing but signal assignment and Event is a transaction with the change in value
- Delta cycles are "timeless" intervals that do not exist in real time but are necessary in simulation

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Concurrent Assignments

Slide 9-1:

2019.1

Concurrent statements form the basis for the RTL coding style. In this module, both conditional and unconditional assignments are covered. 20 minutes.

Slide 9-2:

Objectives

After completing this module, you will be able to:

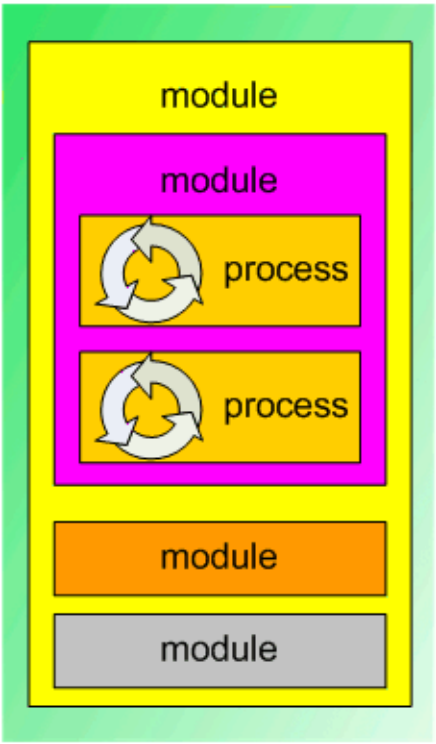
- Define the seven types of concurrent statements
- Show how to make unconditional and conditional assignments
- Use the generate statement in your code

© Copyright 2019 Xilinx

Slide 9-3:

Types of Concurrent Constructs

- Assignment (logical, arithmetic)
  - Uses the <= construct
  - Left side and right side types and sizes must match
  - Using assignment statements is considered RTL coding because it describes the movement through logic between registers
- Processes
  - Provides a mechanism for encapsulating sequential statements
  - Forms the basis for behavioral coding
- Instantiations
  - Basis for hierarchical structure
  - Using only instantiations and assignments is termed structural coding
- Generate statements



All the items on this slide exist and operate concurrently.

All four of these items can be freely mixed within a module. That is, a module can contain a number of conditional and unconditional assignment in which logic or math is performed, zero or more processes, and any number of instantiations.

© Copyright 2019 Xilinx

Slide 9-4:

Why Conditional Assignments?

- Sometimes a signal should be assigned different values at different times or in various circumstances

- All relational operations have return type as boolean
  - Operators are predefined for most data types
  - Objects being compared must be of the same (or overloaded) types

```
=     Equality
/=    Inequality
<     Less than
<=    Less than or equal
>     Greater than
>=    Greater than or equal
```

- Two basic forms of conditional assignment
  - *when*
  - *with/select*

The VHDL-2008 standard provides relational operators that return bit or std_logic types.

© Copyright 2019 Xilinx

Slide 9-5:

"when" Form

- Use the *when* form when the assignment conditions are based on various inputs

- Syntax

```
<target signal> <= <waveform> when (<condition>)
                   [else <waveform> when (<condition>)]
                   [else <waveform> when (<condition>)] … ;
```

- Example 1

```
target <= a and b when (c = '1');
```

- Example 2

```
target <= a and b when (c = '1') else d;
```

- Example 3

```
target <= a and b when (c = '1') else
          y when ((c = '0') and (d = '1')) else
          X"41";
```

Syntax:

- <target signal> - any legal identifier.
- <waveform> - any signal or legal combination of signals (such as a logical or arithmetic equation).
- <condition> - any boolean value or equation that evaluates to a boolean value.

Note to software engineers: The *when* form is most similar to the *if-then-elseif-endif*" structure.

Assume that the signals specified in the examples are all std_logic, except Example 3 where a, b, and y are std_logic_vectors.

Example 1 demonstrates the simplest form of the *when* statement. In this example, when c is equal to '1' then the target receives the logical 'and'ing of a and b.

© Copyright 2019 Xilinx

Important note: This reflects poor coding style because it infers a latch. The discussion about why latches are bad is really a side issue, but suffice it to say that the timing analysis tools have a very difficult time handling latches and should be avoided in FPGA designs. The latch is created because there is not an *else* condition. That is, when c = '1' then target gets a and b. But what happens when c /= '1'? In this situation, the previous value of target must be stored, and, because there is no clock function, it is a latch.

Best design practice:

- When using the *when* statement, always have a default *else* clause.
- Example 2: As with the previous example, when c is '1' then target gets the result of a and b. When c is not '1' then target get the value of d.

  Recall that std_logic is comprised of nine different values. Therefore, during simulation, you can test if c is 3-state or undefined, which does not have an equivalent in synthesis. Therefore, it is a better practice to test for inequality than equality for such conditions.
- Example 3: Building on the previous examples, additional comparisons are provided. It is useful to note that the <condition> clause does not need to always contain a single signal. Any statement that evaluates to a boolean can be used here. This is an important point to contrast with the way *with/select* form which is shown next.

© Copyright 2019 Xilinx

Slide 9-6:

"with/select" Form

- Use the *with/select* form when the assignment is only predicated on multiple values of one signal

- Syntax

```
with <signal> select
   <target> <= <waveform> when <value of signal>,
               <waveform> when <value of signal>,
               <waveform> when others;
```

- Similar to *when* but the signal selection is limited to one value

- Example

```
with (quadrant ) select
   maxAngle <=  90 when (1),
               180 when (2),
               270 when (3),
               360 when (4),
                 0 when others;
```

Syntax:

- <target signal> - any legal identifier.
- <waveform> - any signal or legal combination of signals (such as a logical or arithmetic equation).
- <value of signal> - any legal value of the type of target signal.

Note to software engineers: This is roughly the concurrent statement equivalent of a case or switch statement.

© Copyright 2019 Xilinx

Slide 9-7:

About Instantiation

- Instantiations are structural in nature — they define how modules connect
- Instantiations form the basis of hierarchy — lower level modules are "instantiated"
- Modules *must* be defined before they can be used
  - Once defined, modules can be instantiated any number of times

```
architecture STRUCT of TOP is
   <signal declarations>
   ...
   component <module name>
      generic (<generic declaration list>);
      port (<port declaration list>);
   end component <module name>

 begin

   <label>: <module name>
      generic map (<generic mapping list>)
      port map (<port mapping list>);

   <label>: <module name>
      generic map (<generic mapping list>)
      port map (<port mapping list>);

end architecture STRUCT;
```

- Declaration syntax shown in the figure on the right in gold
- Use syntax shown in the figure on the right in green

Most tools will create both the declaration and instantiation templates so that they can be cut-and-pasted into a source file.

Since VHDL 93, an alternate form has been available. Because this form specifies the module name and the architecture associated with that module's entity, a component declaration is not required.

An example of this form:

```
regCh1: entity xil_defaultlib.register8(Behavioral)
        PORT MAP(
                clock => clock,
                reset => reset,
                enable => Channel_1_enable,
                data_in => Channel_1_data,
                data_out => Channel_1_registered_data
        );
```


© Copyright 2019 Xilinx

Slide 9-8:

Creating the Component Declaration

- The component declaration identifies all of the information that can be passed in and out of the module

- They are nearly identical to the entity clause

```
component uart_tx
  generic( CLOCK_RATE    : integer := 50_000_000;
           BAUD_RATE     : integer :=    115_200);
  port ( clk_tx          : in  STD_LOGIC;
         rst_clk_tx      : in  STD_LOGIC;
         char_fifo_empty : in  STD_LOGIC;
         char_fifo_dout  : in  STD_LOGIC_VECTOR (7 downto 0);
         char_fifo_rd_en : out STD_LOGIC;
         txd_tx          : out STD_LOGIC);
end component uart_tx;
```

Generics can be viewed as values that are constants, but may differ between instantiations.

Many designers will place the component declarations into a package and include that package in every module (packages are covered near the end of this course). This reduces clutter (and the overall length of a module) and, should a change be required, only one declaration needs to be modified—the one in the package.

© Copyright 2019 Xilinx

Slide 9-9:

Instantiation Port Mappings

- Port mappings associate signals used inside the module with signals at the current level of hierarchy

- There are two ways to associate these signals
  - Unnamed association uses the position (order) of the signals to map
  - Named association uses names of the signals to map
    - Uses the => symbol to make the association
    - This symbol does NOT imply directionality
    - Port names (belonging to the module) are on the left; signals at the current level of hierarchy are on the right

```
uart_tx_ctl_i0: uart_tx_ctl
     port map (clk_tx          => system_clock,
               rst_clk_tx      => system_reset,
               baud_x16_en     => baud_x16_en,
               char_fifo_empty => char_fifo_empty,
               char_fifo_rd_en => char_fifo_rd_en,
               char_fifo_dout  => open,
               txd_tx          => txd_tx);
```

Unnamed association is prone to errors. The time it takes to identify, locate, and fix one error is considerably greater than the time it takes just to create a named association.

Named associations allow for easier debugging, documentation, and maintenance. Best practices calls for the use of named associations.

Inexperienced VHDL users are often confused by the association operator (=>). This operator establishes a connection between two, possibly different, named signals. It does not imply a direction.

If a signal is not connected, best design practice calls for the signal to still be listed in the port map. Use the keyword *open* to indicate that the port is not connected.

© Copyright 2019 Xilinx

Slide 9-10:

Instantiation Example

```
architecture Behavioral of uart_tx is
    signal baud_x16_en : std_logic := 'U';
    [component declaration for uart_baud_gen]
    [component declaration for uart_tx_ctl]
  begin
  uart_baud_gen_tx_i0: uart_baud_gen
        generic map (CLOCK_RATE   => CLOCK_RATE,
                     BAUD_RATE    => BAUD_RATE)
        port map    (rst          => rst_clk_tx,
                     clk          => clk_tx,
                     baud_x16_en => baud_x16_en);

  uart_tx_ctl_i0: uart_tx_ctl
        port map (clk_tx          => clk_tx,
                  rst_clk_tx      => rst_clk_tx,
                  baud_x16_en     => baud_x16_en,
                  char_fifo_empty => char_fifo_empty,
                  char_fifo_rd_en => char_fifo_rd_en,
                  char_fifo_dout  => char_fifo_dout,
                  txd_tx          => txd_tx);

    end Behavioral;
```

```
entity uart_baud_gen is
  generic (CLOCK_RATE   : integer := 100_000_000;
           BAUD_RATE    : integer :=     115_200);
  port ( rst            : in  STD_LOGIC;
         clk            : in  STD_LOGIC;
         baud_x16_en    : out STD_LOGIC);
end entity  uart_baud_gen;
```

```
entity uart_tx_ctl is
  port ( clk_tx          : in  STD_LOGIC;
         rst_clk_tx      : in  STD_LOGIC;
         baud_x16_en     : in  STD_LOGIC;
         char_fifo_empty: in  STD_LOGIC;
         char_fifo_rd_en: out STD_LOGIC;
         char_fifo_dout : in  STD_LOGIC_VECTOR(7 downto 0);
         txd_tx          : out STD_LOGIC
         );
end entity uart_tx_ctl;
```

[component declaration for uart_baud_gen], [component declaration for uart_tx_ctl] represent the full declaration for these components. They have been omitted due to size limitations. This example was taken from the wave_gen design which is available in the lab file set.

Important: The generic map portion is only required when the default values defined in either the declaration or the entity are to be overridden.

Note the use of named association.

© Copyright 2019 Xilinx

Slide 9-11:

Timed Conditional Assignments

- Timed conditional assignments are not synthesizable
- Used in testbenches to provide various values to signals as a function of time
- Syntax

```
<signal name> <= <expression> after <time specification> [,
                 <expression> after <time specification>,
                   […]    ]
```

- Examples
  - reset <= ‘1’, ‘0’ after 1 us;
  - sl_x <= ‘Z’ after 500 ns;
  - int_y <= 1 after 10 ns, 5 after 20 ns, 33 after 50 ns;
  - clk <= not clk after 5 ns;

Syntax:

- <signal name> - any defined, legal identifier.
- <expression> - any expression that evaluates to the same type as <signal name>.
- <time specification> - any legal time construct.

Examples:

- reset <= ‘1’, ‘0’ after 1 us; – reset takes on the value of ‘1’ immediately and is held for 1 us when it changes to a ‘0’.
- sl_x <= ‘Z’ after 500 ns; – sl_x is undefined or at its specified initial value until 500 ns into the simulation where it becomes 3-stated.
- int_y <= 1 after 10 ns, 5 after 20 ns, 33 after 50 ns; – Undefined, or at its specified initial value until 10 ns when it becomes a 1. 10 ns later it changes to a 5, and 30 ns later it changes to 33 and remains.

The previous examples use STATIC values. This causes the statement to evaluate once during its lifetime. The following statement changes every 5 ns and therefore resets its “timer”.

© Copyright 2019 Xilinx

clk <= not clk after 5 ns; – clk takes on the inverse of its initial value after 5 ns, where it takes on the inverse of its current value (“flipping”). This process repeats until the end of simulation resulting in a 10-ns or 100-MHz 50% duty cycle waveform.

Note: If clk is not properly initialized to a ‘1’ or ‘0’, it may fail to oscillate.

Slide 9-12:

Force and Release

- For verification, it is convenient to be able to "override" the value of a signal
  - Mainly used for error injection
- VHDL-2008 allows you to override the value of a signal and also allows you to remove the overridden value
- force – Allows the overriding of the value of a signal
  - V <= force in '0'; -- force effective value
  - V <= force out '1'; -- force driving value
- release – Allows the removal of the overridden value of a signal
  - V <= release in; -- release effective value
  - V <= release out; -- release driving value

Slide 9-13:

Concurrent Looping

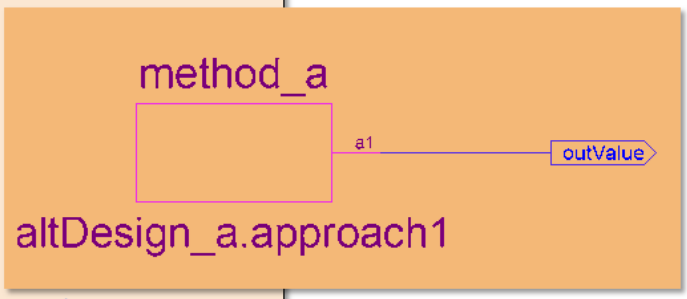
- Sequential looping is preferable as a control structure
  - Has limited ability to create logic
  - Most commonly used in testbenches to build stimulus
- Concurrent looping structures are used primarily for instantiating multiple copies of components
- The generate statement has two forms
  - Loop form – For creating multiple copies of a collection of components
  - Conditional form – For creating or not creating a collection of components

© Copyright 2019 Xilinx

Slide 9-14:

Conditional Generate Example

```
entity y is
  port ( outValue : out std_logic);
end entity y;

architecture Behavioral of y is
    component method_a is
      port (a1 : out std_logic);
    end component method_a;

    component method_b is
      port (b1 : out std_logic);
    end component method_b;

    type iType is (T1, T2);
    constant testModule : iType := T1;

  begin
    altDesign_a: if (testModule = T1) generate
      approach1: method_a port map (a1 => outValue);
    end generate altDesign_a;

    altDesign_b: if (testModule = T2) generate
      approach2: method_b port map (b1 => outValue);
    end generate altDesign_b;
  end Behavioral;
```



Notice that the generation of the design is determined by a constant during elaboration.

Note that both port a1 and b1 are tied to outValue. Under normal circumstances, this would produce a “multi-source” error, the VHDL equivalent of a short circuit. It does not occur here because only one of these modules will ultimately drive outValue.

© Copyright 2019 Xilinx

Slide 9-15:

Example of a Variable-Width Counter Using Generics

```
entity variCounter is
  Generic (width : integer := 7);
  Port ( clk      : in  STD_LOGIC;
         reset    : in  STD_LOGIC;
         enable   : in  STD_LOGIC;
         countOut : out  STD_LOGIC_VECTOR (WIDTH-1 downto 0)
       );
end variCounter;
```

```
architecture Behavioral of variCounter is
    constant max_slv   : std_logic_vector(WIDTH-1 downto 0) := (others=>'1');
    constant max_value : integer := to_integer(unsigned(max_slv));
  begin
    doCounter: process (clk)
        variable internal_count : integer range 0 to max_value-1 := 0;
      begin
        if rising_edge(clk) then
          if (reset = '1') then
            countOut <= (others=>'0');
            internal_count := 0;
          else
            if (enable = '1') then
               -- see comment in the notes section
              if (internal_count = max_value-1) then internal_count := 0;
              else                                  internal_count := internal_count + 1;
              end if;
            end if;
            countOut <= std_logic_vector(to_unsigned(internal_count,WIDTH));
          end if;
        end if;
      end process doCounter;
  end Behavioral;
```

Note the coding technique for the constants in the architecture (which could just as easily be put in the process).

The trick is to find the largest value that the integer can take on during the operation of the circuit. With max_slv being set to a std_logic_vector and initializing it to all ones, the maximum value has been created. Thus, the following line creates the maximum value that the integer can have for the specified width. This technique avoids the use of exponentiation.

*Is this block of code really needed? Isn't it sufficient to just have "internal_count := internal_count + 1;"?

Yes and no. To be proper and run in simulation then yes, a reset on reaching the maximum value is required; however, in synthesis, this would roll over naturally due to max_value always being a power of 2.

Synthesis will only generate WIDTH-1 number of flops which will automatically roll over one clock after the maximum count is reached. This can cause a simulation mismatch and should be watched for carefully.

© Copyright 2019 Xilinx

Slide 9-16:

Generate Statement – Conditional Form

- Used to implement one group of modules over another
  - Useful for supporting multiple architectures

- Syntax

```
[<generate label>:] if <condition> generate
   <concurrent statements>
end generate [<generate label>];
```

- Example
  - Two algorithms have been written to implement a specific function
  - Each needs to be tested within the framework of the other FPGA modules
  - Both modules will be evaluated for function, resource utilization, and performance
  - The generate statement provides a straightforward mechanism for testing

Syntax:

- <generate label> - any unique, legal identifier. If the trailing <generate label> is used, then it must match the initial <generate label>.
- <concurrent statements> - any number of legal concurrent statements.

VHDL-2003 does not support the “else” statement; however, it is supported in VHDL-2008. Regardless, the same functionality can be achieved by building another if-generate statement with the negative condition.

The Vivado® synthesis tool supports the VHDL-2008 construct `if-else-if generate` statement.

© Copyright 2019 Xilinx

Slide 9-17:

Generate Statement – Loop Form

- Used to repeatedly implement a group of modules or components
  - Useful for supporting replicating logic such as I/O buffers

- Syntax

```
[<generate label>:] for <identifier> in <range> generate
   <concurrent statements>
end generate [<generate label>];
```

- Example
  - Special I/O buffers need to be instantiated for creating a wide bus
  - Multiple copies of components need to be instantiated

Syntax:

- <generate label> - any unique, legal identifier. If the trailing <generate label> is used, then it must match the initial <generate label>.
- <concurrent statements> - any number of legal concurrent statements.
- <identifier> - any legal identifier. The identifier should not be otherwise declared and is only valid within the generate loop.
- <range> any legal range.

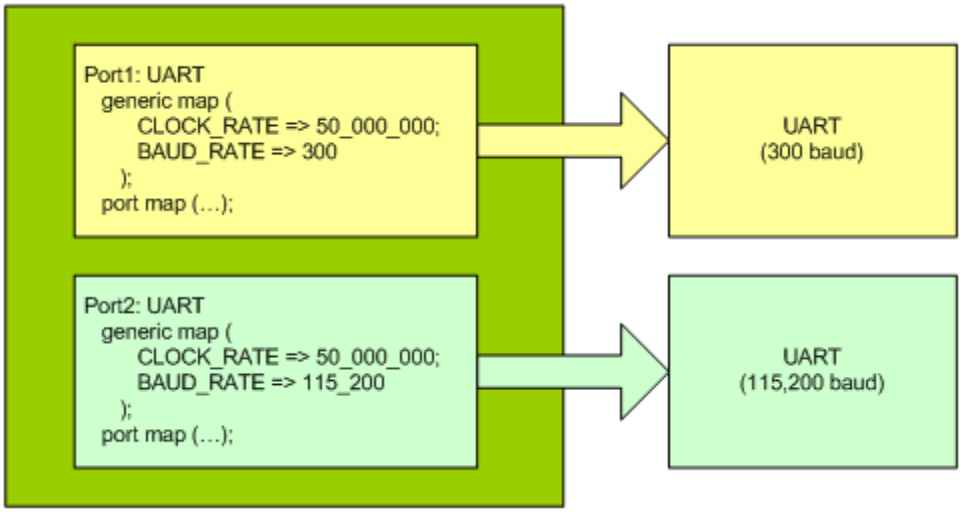
© Copyright 2019 Xilinx

Slide 9-18:

Generics

- Generics are a convenient means for passing different, but fixed values to components

- For example, if a design supports multiple RS-232 serial ports and one port is designated to run at 300 baud and the other at 115200 baud, then the UART code can be written based on the desired baud rate (and thus remain unchanged between instantiations), but generate the proper code to support each baud rate



© Copyright 2019 Xilinx

Slide 9-19:

Using Generics

- The component must have the generics declared in the entity
  - Default values may be provided

- Syntax (partial entity syntax)

```
entity <entity name> is
  [<generic declaration>;]
  [<port declaration>;]
end [entity [<entity name>]];
```

- Generics must be called out in the component declaration
  - Defaults values can be provided here and override values in the component
  - Multiple generic values are separated by commas

- Generics without default values must be mapped in the instantiation of the component

```
component <entity name> is
  [<generic declaration>;]
  [<port declaration>;]
component [entity [<entity name>]];
```

```
component Ex1 is
  generic (X : integer := 7);
   . . .
component entity Ex1;
```

Note that the entity syntax contains far more capability.

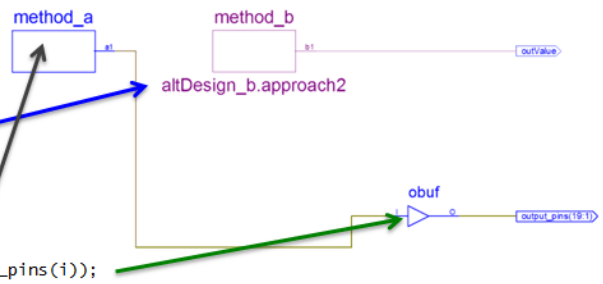
Default values for generics can be provided in both the component definition and within the component module. The default in the component definition overrides that listed in the component module.

© Copyright 2019 Xilinx

Slide 9-20:

Loop Generate Example

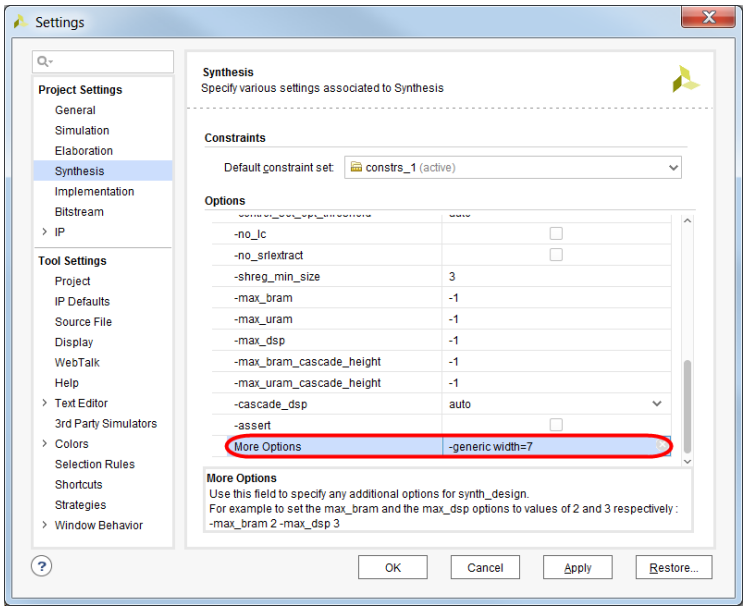
```
 . . . Snip . . .

entity y is
   generic (number_of_pins : integer := 20);
   port ( outValue    : out std_logic;
          output_pins : out std_logic_vector(number_of_pins-1 downto 0) );
end y;

architecture Behavioral of y is

    . . . Snip . . .
  type iType is (T1, T2);
  constant testModule : iType := T2;
  signal intermediate : std_logic_vector(number_of_pins-1 downto 0) := (others=>'U');

begin

  altDesign_a: if (testModule = T1) generate
    approach1: method_a port map (a1 => outValue);
  end generate altDesign_a;

  altDesign_b: if (testModule = T2) generate
    approach2: method_b port map (b1 => outValue);
  end generate altDesign_b;

  manyUnits: for i in 0 to number_of_pins-1 generate
    unitsA: method_a port map (a1 => intermediate(i));
    buf:    obuf port map (I=>intermediate(i), O=>output_pins(i));
  end generate manyUnits;

end Behavioral;
```



Slide 9-21:

Top-Level Generics

- Overriding generics are specified when the component is instantiated
- Synthesis options can pass generics to the top level



© Copyright 2019 Xilinx

Another popular mechanism is to set a parameter in a package. For more information, refer to the *Vivado Design Suite User Guide: Synthesis* (UG901).

Slide 9-22:

Summary

- Four forms of concurrent assignments
  - Conditional/unconditional, instantiation, process, and generic
- Conditional assignment using the *when* construct is analogous to a software *if-then-elseif-else-endif* structure
- Conditional assignment using the *with/select* construct closely matches the *case/switch* statements in software
- Generate is a concurrent looping statement and is used primarily for instantiating multiple copies of components

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Processes and Variables

Slide 10-1:

2019.1

This module introduces sequential programming techniques for a concurrent language. Variables are also discussed here. 20 minutes.

Slide 10-2:

Objectives

After completing this module, you will be able to:

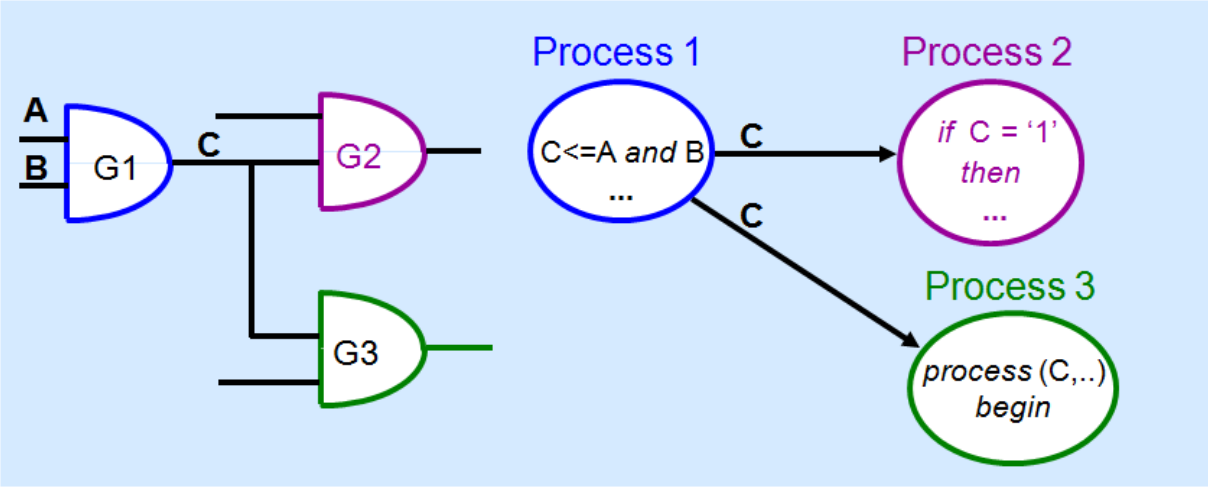
- Describe what a process is and why it is beneficial
- Explain the purpose and proper implementation of a process sensitivity list
- Show how to define variables and compare and contrast them to signals

© Copyright 2019 Xilinx

Slide 10-3:

Processes are Concurrent

- When modeling hardware, the concept of concurrency is essential
  - Any logic change on signal C, the output of gate G1, is seen at the input of G2 and G3 concurrently



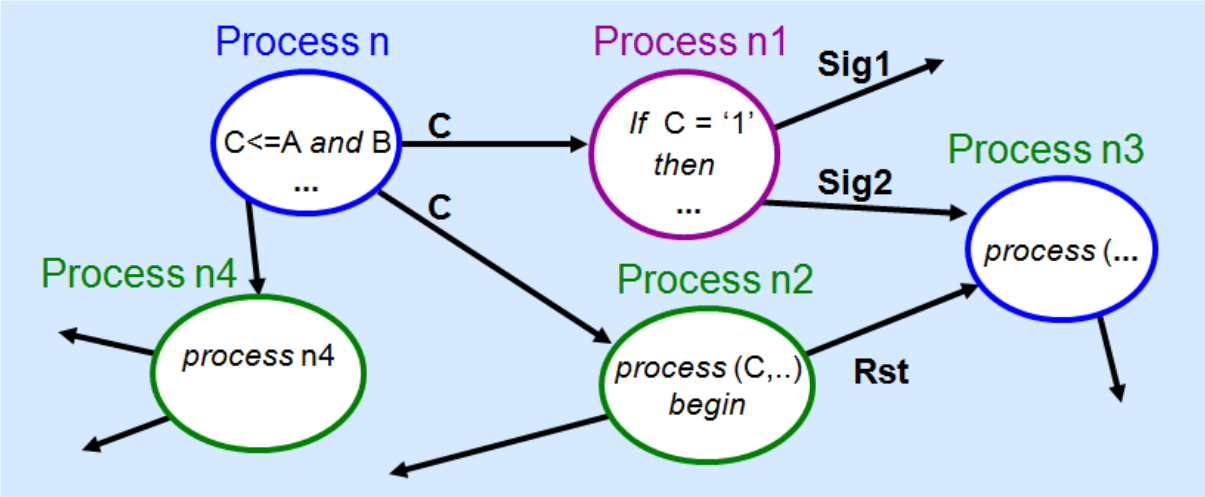
- Remember that other VHDL constructs such as conditional, non-conditional, generate, and other statements exist concurrently with processes

© Copyright 2019 Xilinx

Slide 10-4:

VHDL Connectivity Model

- In RTL coding, only signals carry values from one process to another
  - Concurrent processes, connected by signals, are sometimes referred to as the VHDL Connectivity Model



Slide 10-5:

Process

- A process contains statements that are executed sequentially
  - The order of the statements impacts the final result
- Processes are "executed" simultaneously with other concurrent statements
- Each process has a means for being triggered
  - By changes on specific signals as expressed in a wait condition
  - All signals on the "right-hand" side of an assignment must be listed for non-clocked processes

Syntax

[<label>:] *process* (<sensitivity list>)

<process declarative area>

begin

<process behavior area>

end process [<label>];

```
architecture RTL of MY_AND2 is
  begin
    . . .
    and2: process (A, B)
      begin
        C <= A and B ;
    end process ;
    . . .
  end architecture RTL;
```


© Copyright 2019 Xilinx

Syntax:

- <label> - any legal identifier. Labels can be useful as documenting what the process does as well as helping to locate and view items of interest during simulation and in reading synthesis reports.
- <sensitivity list> - a list of signals of which if any change is detected in any of them, the process will release from its suspension and make another pass through the sequential statements in the <process behavior area>.
- <process declarative area> - where variables, signals, constants, types, subtypes, functions, and procedures may be declared. This region is similar to the architecture declarative area in that anything defined in this area is considered to be “local” to the process and not accessible to any other process or statement.
- <process behavior area> - where the function of the process is described.
- The final (optional) label can only exist if the process was given a label prior to the process keyword. These labels must match.

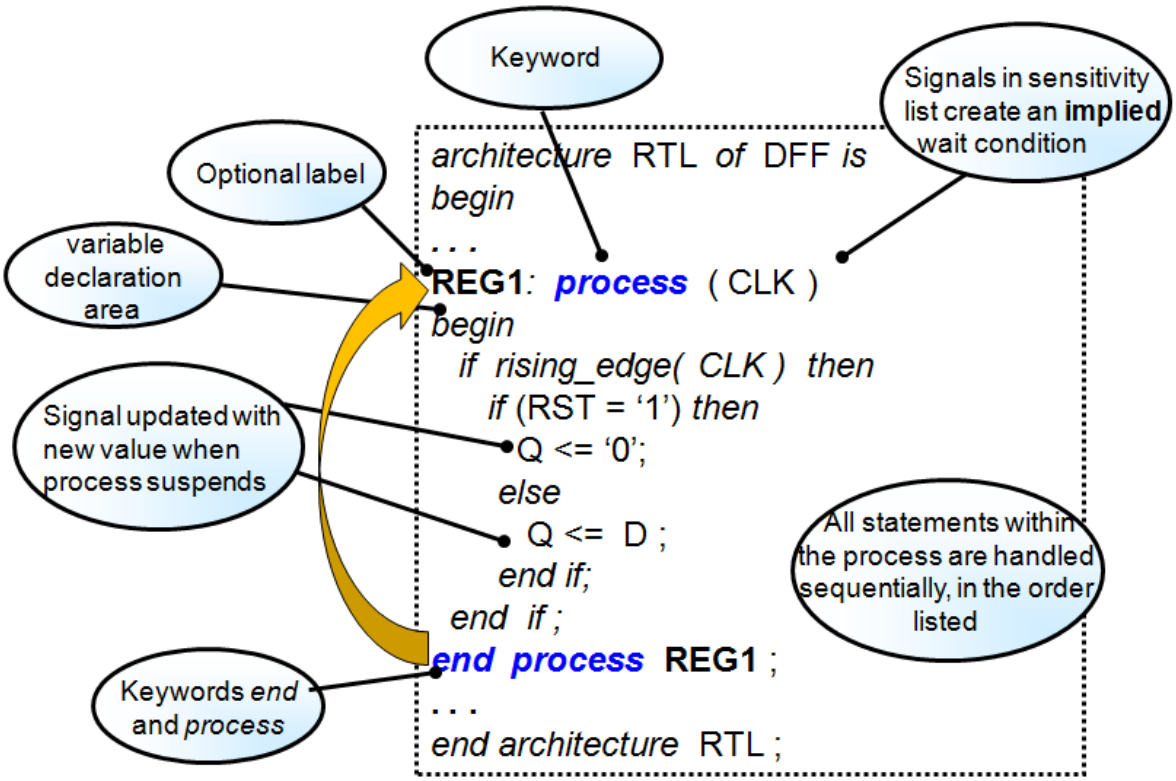
Processes can only exist inside of an architecture. Any non-clocked logic can also be modeled outside the process—in this case, using the same C <= A and B; construct within the architecture.

The process illustrated on this slide is known as an asynchronous process because there is no association with a clock. Although this type of process can be useful, one should limit their use as it tends to lead to low-performing designs as it may add additional layers of logic between registers.

© Copyright 2019 Xilinx

Slide 10-6:

Process Syntax Basics



The arrow reminds you that a process is inherently a self-looping statement. This also means that the end process keyword does not suspend the process.

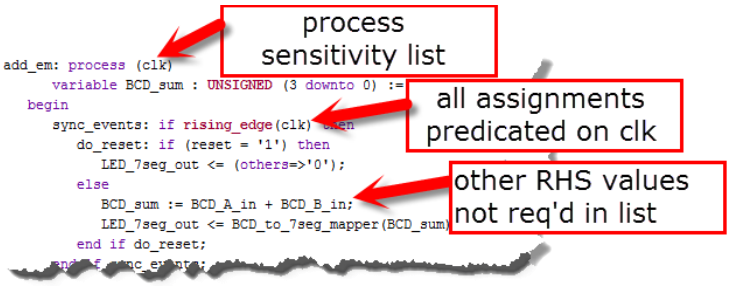
The illustration shows where the variable declaration area is; however, for this simple design, no variables are defined.

© Copyright 2019 Xilinx

Slide 10-7:

Process Sensitivity List

- The process sensitivity list is used to indicate when the process should be evaluated
  - Required for simulation to execute properly
  - Optional for synthesis (although warnings will be generated)
- If any signal in the process sensitivity list changes, then the process will be evaluated
- Asynchronous processes require that any signal on the right-hand side of any equation in the process be included in the process sensitivity list
- Synchronous processes only require the clock signal to be present in the process sensitivity list



A signal that is used in the body of a process but not listed in the process sensitivity list will (or should) generate a warning message.

If this warning is ignored, there is a risk of running into a “simulation mismatch,” which means that the simulation will not execute this process every time it is supposed to, and the results will not match the synthesized netlist.

What is the purpose of the process sensitivity list?

An historically-based answer: In simulation, CPU cycles are expensive, so why would or should a process be evaluated if nothing in it has changed? The process sensitivity list provides a mechanism for accelerating simulation time.

During synthesis the process is converted into silicon resources that always exist. Therefore, the process sensitivity list is essentially ignored.

© Copyright 2019 Xilinx

When listing items in the sensitivity list of a process statement for combinational logic, it was up to the designer to make sure all the items read by the process statement were listed. If any were missed, there would be Warning messages and possible latches inferred in the design.

With VHDL-2008 you can use the process (all) statement, which looks for all the inputs to the process and then creates the logic.

Slide 10-8:

Process Sensitivity List Examples

```
asynch1: process (???)
   variable X, Y : std_logic;
 begin
   X := not (A xor B);
   Y := X nor C;
   Z <= X and Y;
 end process asynch1;
```

asynch1: process (A, B, C) …

```
synch2: process (???)
   variable X, Y : std_logic;
 begin
   if (reset = '1') then
     Z <= '0';
   if rising_edge(clk) then
         X := not (A xor B);
         Y := X nor C;
         Z <= X and Y;
   end if;
 end process synch2;
```

synch2: process (clk, reset) …

Assume that clk, reset, and Z are signals defined outside the process for both processes.

asynch1 example:

- If signal 'A' changes, then it may cause a change in X. This requires that it be included in the Process Sensitivity List.
- The same is true of signal 'B'.
- Because signal 'C' may cause Y to change, regardless of the behavior of A or B, it too is required to be mentioned in the process sensitivity list.
- Neither X nor Y are signals; therefore, they do NOT belong in the process sensitivity list. Even if they were signals (which could not be driven external to this process as it would cause a "multi-source" (short) error), they would not need to be included in the list as they are dependent on the signals already included in the process sensitivity list.

© Copyright 2019 Xilinx

synch2 example:

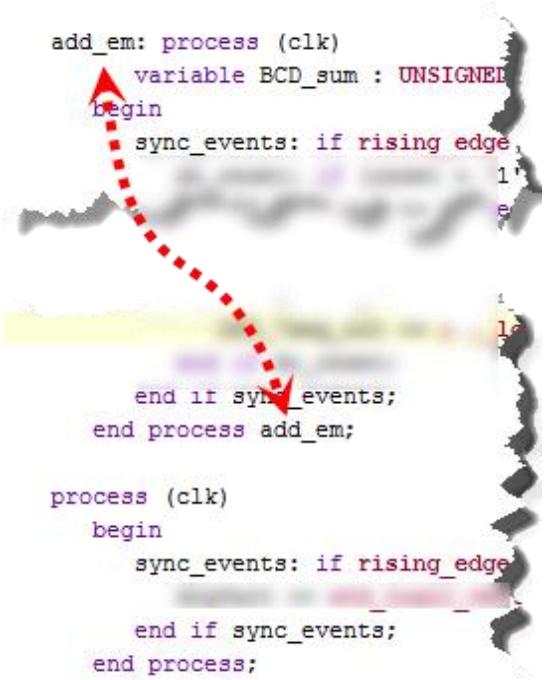
- Reset is an asynchronous signal and signal ‘Z’ may change as a result of a change in reset; therefore, it must be included in the list.
- If a rising edge transition is seen on signal clk, then it is possible that X, Y, and Z may need to be evaluated; therefore, it too must be included in the process sensitivity list.
- A, B, and C are no longer required in the process sensitivity list as a change in their state has no effect on Z unless the clk is rising. Therefore, it is sufficient to include only the clk and reset signals in the process sensitivity list.

synch2 is an example of an asynchronous reset. This is NOT the preferred coding style, but is presented here as an example of a mixed asynchronous/synchronous process example.

Slide 10-9:

Process Labels

- While process labels are optional, it is a "Best Practice" to intelligently name the process
  - Helps to identify where resources are consumed, where debugging target are, etc.
  - Otherwise the process is named by the synthesis/simulation tools and it is difficult to trace back to the source
  - Label names at the beginning and end of a process help identify which process is which when looking at lengthy code



© Copyright 2019 Xilinx

Slide 10-10:

Synchronous Processes

- Synchronous processes have a number of advantages over asynchronous processes
  - Only the clock signal is required in the process sensitivity list
  - Generally leads to higher performing design
    - Increases latency, but reduces levels of logic
    - Flip-flops built into Xilinx architecture along with combinatorial resources
    - Using them is essentially "free"

- Best design practice
  - Resets should be avoided as much as possible
    - When absolutely required they should be active high and synchronous
    - This matches how the FPGA is architected and results in better performance and resource utilization

- Best design practice
  - Name each process meaningfully
    - Process names are seen in both simulation and in the synthesis report

In the original 7400 series logic, resets were active low as this meshed better with the open gate technology of the day. Early FPGA libraries kept this active low configuration as engineers were used to it and, depending on the specific architecture, may not have introduced any inefficiencies.

Current FPGAs are designed with active high sets/resets in the flops, so providing an active low reset actually causes additional logic to be used.

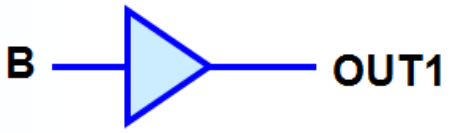
It is true that synchronous processes always use more flip-flops than asynchronous processes; however, the way the FPGA is constructed, if the LUT is used in a LUT/FF pair, the flip-flop cannot be used for anything else, so coding in an asynchronous fashion does not actually reduce the number of LUT/FF pairs used. Simply put, for this purpose, flip-flops are free.

© Copyright 2019 Xilinx

Slide 10-11:

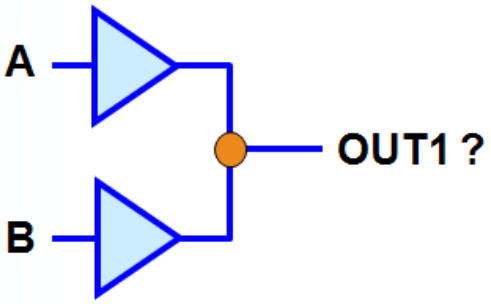
Inside Process versus Outside Process

```
architecture ...
process ( )
begin
        OUT1 <= A ;
        OUT1 <= B ;
            . . .
 end  process ;
end  architecture ;
```



The last assignment takes effect

```
architecture  . . .
      begin
        OUT1 <= A ;
        OUT1 <= B ;
. . .
end  architecture ;
```



Which driver is applied? Some form of resolution is required

Within a process (top example), the assignments to OUT1 are made sequentially, which means that only the last assignment will be performed. All previous assignments to OUT1 are overridden.

In the second example, the assignments to OUT1 are not inside a process; therefore, they are handled concurrently. This means that both assignments are considered to be simultaneous, and some sort of resolution function (or the default, as defined in the std_logic_1164 package) must be defined to determine which value will drive the signal OUT1.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 10-12:

Variables

- Variables are "place-holders" for signals
  - Often used to break up large, complex equations into smaller, clearer statements
  - May not exist in the final design, therefore, difficult/impossible to "see" with tools
    - Current simulation tools enable limited viewing of variables
- Variables take on their assignments IMMEDIATELY
  - Whereas signals take on their new values when the process suspends
- Variables must be declared before use
  - All declarations within a process are defined between the end of the "process" statement and the begin statement for that process
  - Variables are only visible within the function, process, or procedure that they are declared in
- Syntax for a variable declaration
  - variable <variable name> : <variable type> [:= <initial value>];
- Syntax for assignment
  - <variable name> := <waveform>;

Syntax:

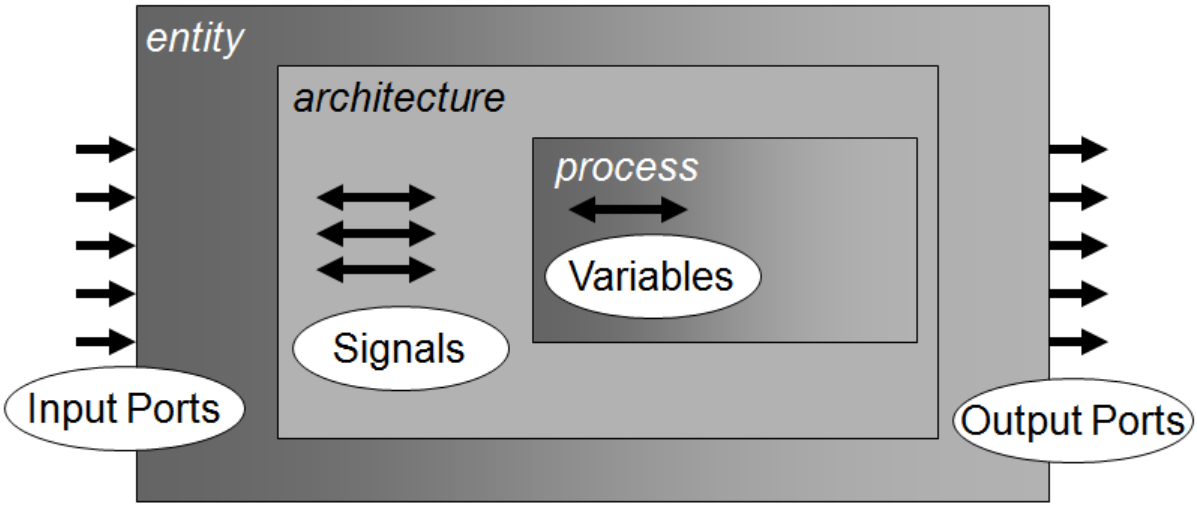
- <variable name> - a legal identifier.
- <variable type> - any previously defined type or subtype.
- <initial value> - the value of this variable when the FPGA comes out of configuration (not when it is reset).

© Copyright 2019 Xilinx

Slide 10-13:

Variable Scope

- Variables are declared within the process and are local to that process
- They cannot be used to directly carry values from one process to another



However, a common practice would be to assign the value of the variable to a signal so that its value can be carried outside of the process.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 10-14:

Using Variables

- Any variable that is read before its value is defined will infer a register in hardware if it is used in a clocked process (acceptable practice)
  - Variables must be used carefully within clocked processes
  - A read before being assigned in combinatorial logic results in a latch*

```
oops: process (a, b, c)
    variable x : std_logic;
  begin
    if (a = '1') then
      x := b and c;
    end if;
  end process oops;
```

```
noops: process (a, b, c)
    variable x : std_logic;
  begin
    if (a = '1') then
      x := b and c;
    else
      x := '0';
    end if;
  end process noops;
```

- Variables are commonly used and highly recommended for intermediate combinatorial logic
- They speed up simulation run time because there are fewer delta cycles required to update outputs
- Any variable assigned to another variable or signal must be of the same data type and length

In this case, the term intermediate specifically means within the process.

*Inadvertent latch inference should be avoided at all costs. Because a latch only latches the presented value when the enable signal is dropped (unlike a register which captures the value when an edge of the clock arrives), it is nearly impossible to properly determine the timing through this device which makes latches a highly undesirable feature in FPGAs.

The "oops" process shows how to inadvertently create a latch. When a is '1', then x is assigned the logical 'and'ing of b and c; however, if a is not '1', then there is no behavior explicitly defined for x, so the compiler will preserve the last value of x. Because the only way to "capture" the value of x is when a is '1' (with no association to a clock), the compiler infers a latch.

The "noops" process demonstrates one way to avoid inadvertent latch inference. This is not exactly the same behavior as in oops as when a is not '1', then x is zeroed rather than preserved. Generally speaking, processes should be clocked—this approach avoids many coding pitfalls.

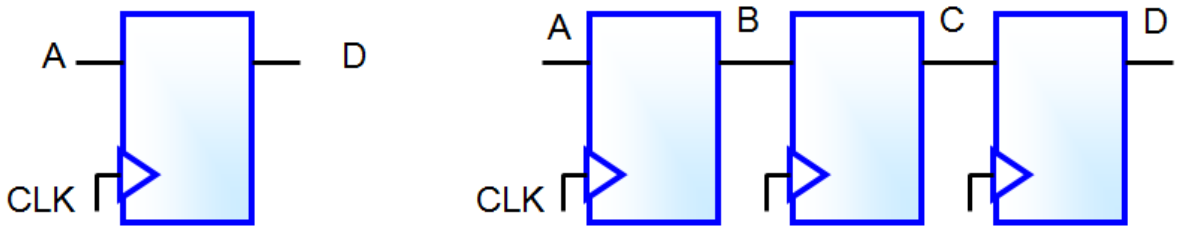
© Copyright 2019 Xilinx

Slide 10-15:

Variables in Clocked Processes

```
process ( CLK )
 variable  B, C  : bit := '1' ;
 begin
   if rising_edge ( CLK ) then
   B := A ;
   C := B ;
   D <= C ;
  end if ;

end process ;
```

```
process ( CLK )
 variable  B, C  :  bit := '1' ;
 begin
   if rising_edge ( CLK ) then
   D <= C ;
   C := B ;
   B := A ;
  end if ;

end process ;
```



Here you see why variables must be used carefully in clocked processes. The code in each example looks very similar, but the inferred logic is quite different.

If B, C, and D were signals, the two processes would be identical (both would create the circuit shown on the right). But, because B, C, and D are variables, the order of the assignments becomes important.

On the left, each variable is assigned before it is read. Because variable assignments are immediate, the net effect of the process is that D gets A on the rising edge of CLK.

On the right, each variable is read before its value is assigned. This creates multiple registers (one for each variable).

Note also that in both cases, the value of the variable is passed out of the process by assigning it to a signal.

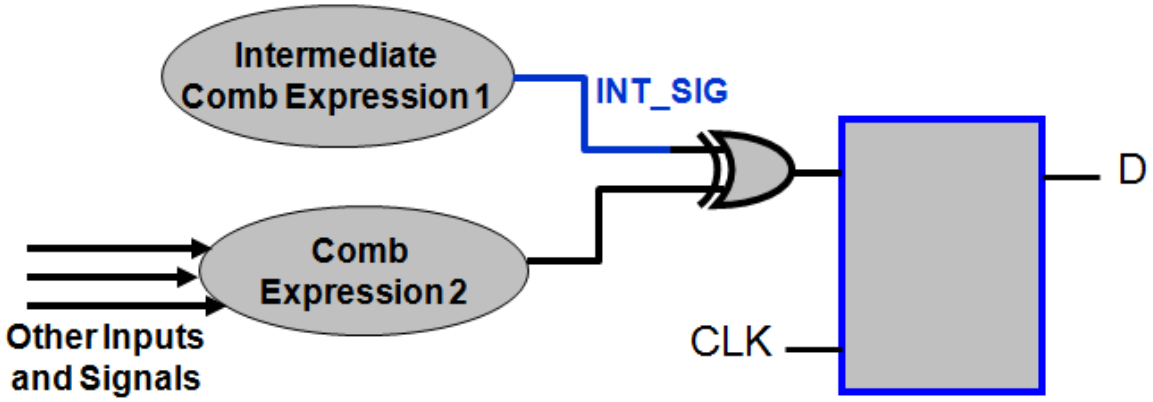
© Copyright 2019 Xilinx

Slide 10-16:

Safe Variable Usage in a Clocked Process

- Advantages
  - Readability
  - Faster simulation

```
process (CLK)
 variable INT_SIG : std_logic := '1' ;
 begin
   if (CLK'event and CLK ='1') then
   INT_SIG := < comb expression1 > ;
   D <= < comb expression2 > xor INT_SIG ;
   end if ;
end process ;
```



Readability is enhanced in part because if both comb expression1 and comb expression2 are combined, the HDL source code is more difficult to read and debug.

Alternatively, if comb expression1 is modeled as a signal, it must be placed in a separate process to ensure that it is updated before it is used in the assignment to signal D.

Note that you should make certain to assign to the variable before it is used in an expression to avoid having additional registers inferred.

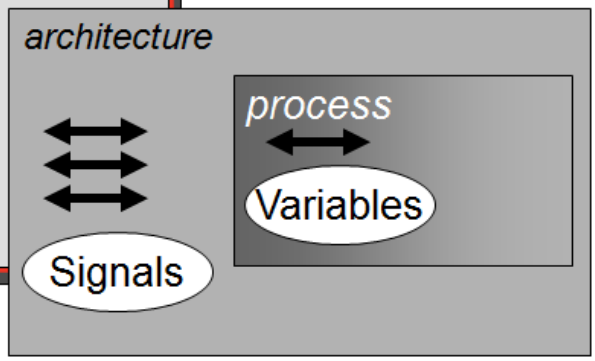
Once again, the value of the variable is passed out of the process by assigning it to a signal in the expression.

© Copyright 2019 Xilinx

Slide 10-17:

Variable Usage Guidelines

```
process (CLK)
 variable INT_SIG : std_logic := '1' ;
 begin
   if rising_edge(CLK) then
   INT_SIG := < comb expression1 > ;
   D <= < comb expression2 > xor INT_SIG ;
   end if ;
end process ;
```



1) Variables can be used for intermediate (local to process) combinatorial logic
2) Always assign to a variable before using it in an expression, unless you want to infer storage (register)
3) Within a clocked process, pay close attention to rule 2 above!
4) Variables can make source code more concise and enhance readability

It is worth mentioning that many less-experienced VHDL designers simply opt to avoid variable usage completely. However, this usually reflects an overall lack of understanding—variables are a powerful language feature in both behavioral and RTL coding.

Following the simple guidelines outlined here, you can fully leverage the advantages that variables offer in your VHDL source code.

© Copyright 2019 Xilinx

Slide 10-18:

Variables versus Signals

```
 1  doCount200: process ( clk )
 2     variable internal_count : integer range 0 to 255 := 0;
 3   begin
 4     if rising_edge(clk) then
 5        if (reset = '1') then
 6           internal_count := 0;
 7           limit_reached  <= '0';
 8           external_count <= (others=>'0');
 9        else
10           limit_reached <= '0';
11           internal_count := internal_count + 1;
12           if (internal_count = 200) then
13              limit_reached <= '1';
14              internal_count := 0;
15           end if;
16           external_count <=
17             std_logic_vector(to_unsigned(internal_count,8));
18        end if;
19     end if;
20   end process doCount200;
```

This simple process illustrates how to create a "count to 200 and recycle" counter. Both the value of the counter (provided via a signal named "external_count" which must be declared in the architecture declaration area) and a signal which indicates that the count of 200 has been reached (which must also be declared in the architecture declaration area with the name "limit_reached").

This process is synchronous with the reset applied synchronous to the clock.

Line 1 – the process suspends until a change is seen on signal clk. The process comes out of suspension on BOTH the rising and falling edges of the clock, although line 4 shows that actual changes will occur only on the rising edge.

Line 4 – The function rising_edge() is provided in IEEE. STD_LOGIC_1164. This statement returns boolean "true" when the signal in question changes from a logic low state to a logic high state since the last simulation time cycle. In synthesis, this places flip-flops on all signals that this process drives.

© Copyright 2019 Xilinx

Line 10 – The limit_reached signal is ALWAYS marked to 0; however it does not take on this value here. If it turns out that internal_count reaches 200, then limit_reached is marked to 1. The internal representation of limit_reached (during simulation) does not occur until the process suspends to wait for the next CHANGE in clk. Notice that the change is detected, not the rising or falling edge.

Line 11 – internal_count is incremented by one and this change takes place IMMEDIATELY.

Line 12 – Because internal_count was just updated it can be used to compare against 200 to determine if the final value was reached. If internal_count were a signal (or external_count used instead) then the LAST value of the count would be available on this clock cycle, not the current count. The difference between using a signal and a variable, in this example, is that a signal would offset (or delay) the value of the count and the limit_reached signal by one clock.

Line 13 – Here "limit_reached" is marked to a '1', but, as before, it does not take on its value until the process suspends.

Line 14 – internal_count is IMMEDIATELY set to 0. Any reference to internal_count after this point would see internal_count as 0, not 200.

Line 17 – Although std_logic_vectors can be incremented using an overloaded '+' operation, this module uses the more correct process of doing math with integers rather than std_logic_vectors. Because the world outside this process uses a std_logic_vector and all the computations and comparisons internal to this process are integers, a conversion is required. Notice that the integer is converted into an 8-bit unsigned value (basically a std_logic_vector with positional understanding), then cast as a std_logic_vector and assigned to external_count.

© Copyright 2019 Xilinx

Slide 10-19:

Apply Your Knowledge

```
asynch1: process (???)
   variable X, Y : std_logic;
 begin
   X := not (A xor B);
   Y := X nor C;
   Z <= X and Y;
 end process asynch1;
```

asynch1: process (???) …

```
synch2: process (???)
   variable X, Y : std_logic;
 begin
   if (reset = '1') then
     Z <= '0';
elsif rising_edge(clk) then
         X := not (A xor B);
         Y := X nor C;
         Z <= X and Y;
   end if;
 end process synch2;
```

synch2: process (???) …

Assume that clk, reset, and Z are signals defined outside the process for both processes.

Exercise: Find the sensitivity list for asynch1: process and sync2: process.

asynch1 example:

- If signal A changes, then it may cause a change in X. This requires that it be included in the process sensitivity list.
- The same is true of signal B.
- Because signal C may cause Y to change, regardless of the behavior of A or B, it too is required to be mentioned in the process sensitivity list.
- Neither X nor Y are signals; therefore, they do NOT belong in the process sensitivity list. Even if they were signals (which could not be driven external to this process as it would cause a "multi-source" (short) error), they would not need to be included in the list as they are dependent on the signals already included in the process sensitivity list.

synch2 example:

- Reset is an asynchronous signal and signal Z may change as a result of a change in reset; therefore, it must be included in the list.
- If a rising edge transition is seen on signal clk, then it is possible that X, Y, and Z may need to be evaluated; therefore, it too must be included in the process sensitivity list.

© Copyright 2019 Xilinx

- A, B, and C are no longer required in the process sensitivity list as a change in their state has no effect on Z unless clk is rising. Therefore, it is sufficient to include only the clk and reset signals in the process sensitivity list.

synch2 is an example of an asynchronous reset. This is NOT the preferred coding style, but is presented here as an example of a mixed asynchronous/synchronous process example.

Slide 10-20:

Summary

- Processes provide a convenient mechanism for inserting sequential instructions
  - Some logic is more easily understood or created with sequential instructions
- Processes limit scope
  - Any signal, variable, constant, or function defined within a process is not visible outside the process
- Every process requires a mechanism for suspension
  - The process sensitivity statement provides this for synthesis
  - Multiple forms of the *wait* statement are available for simulation

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Conditional Statements in VHDL: if/else, case

Slide 11-1:

2019.1

This module introduces sequential programming techniques for a concurrent language. Conditional statements such as the *if/else* and *case* statements are also discussed and compared with their concurrent counterparts. 15 minutes.

Slide 11-2:

Objectives

After completing this module, you will be able to:

- Enumerate control structures within a process
- Use *if/else* and *case* statements in your code

© Copyright 2019 Xilinx

Slide 11-3:

Process Control Structures

- Control structures within a process guide the flow of logic
- Control structures are very similar to traditional software
  - *if/elsif/else/end if*
  - *case*
  - *for* loop
  - *while* loop
- Equivalent structures exist in both processes and the concurrent realm

```
architecture RTL of ENTITY_1 is
      . . .
   begin
      concurrent statements ;
         . . .
       process
         begin
            case ( SEL_A ) is
               when…
                  . . .
               end case ;
          end process ;
         . . .
      process
         begin
            if ( SEL_B = "00" ) then
               . . .
            else . . .
            end if ;
       end process ;
        . . .
end architecture RTL ;
```

Slide 11-4:

if/else Statements

- The *if/else* statement allows operations to be performed based on certain conditions and has three basic forms

```
process
begin
if (boolean expression) then
sequential statements;
end if ;
```

```
process
begin
if (boolean expression) then
sequential statements ;
else
sequential statements ;
end if ;
```

```
process
begin
if (boolean expression 1) then
sequential statements ;
elsif (boolean expression 2) then
sequential statements ;
elsif (boolean expression 3) then
sequential statements ;
else
sequential statements ;
end if ;
```


© Copyright 2019 Xilinx

All if statements should include an *else* clause; otherwise, a latch is generally inferred unless placed in a clocked process which will infer a register.

The VHDL *elsif* works the same as *else if*. However, you can have multiple *elsif* statements and close with a single *end if*. VHDL fully supports the traditional *else if*.

Process if versus Concurrent if

- Equivalent forms

```
architecture ...
 begin
   process ( A,B, C, Sel )
    begin
     if     (Sel = "00" )  then
       Z <= A ;
     elsif (Sel = "10")   then
       Z <= B ;
     elsif (Sel = "11")   then
       Z <= C ;
     else
     Z <= 'X' ;
    end  if ;
   end  process ;
end  architecture ;
```

```
architecture ...
 begin
   Z <= A  when  Sel = "00"  else
        B  when  Sel = "10"  else
        C  when  Sel = "11"  else
        'X' ;
end architecture ;
```

- Both statements produce the same results in simulation and synthesis
  - Why would one be used rather than the other?

The std_logic assignment of X must be uppercase.

These examples describe asynchronous, combinatorial logic. There are certainly times when asynchronous logic is necessary, and even beneficial (such as when implementing a multiple process Moore finite state machine); however, as a general rule, it should be kept to a bare minimum.

Why would one be used rather than another?

- Process form:
  - Synchronous version of the multiplexer (or whatever logic is being implemented).
  - Additional logic to combine with the results of the if statement (concurrent form).
  - Compact size for "point" solutions.

© Copyright 2019 Xilinx

Slide 11-5:

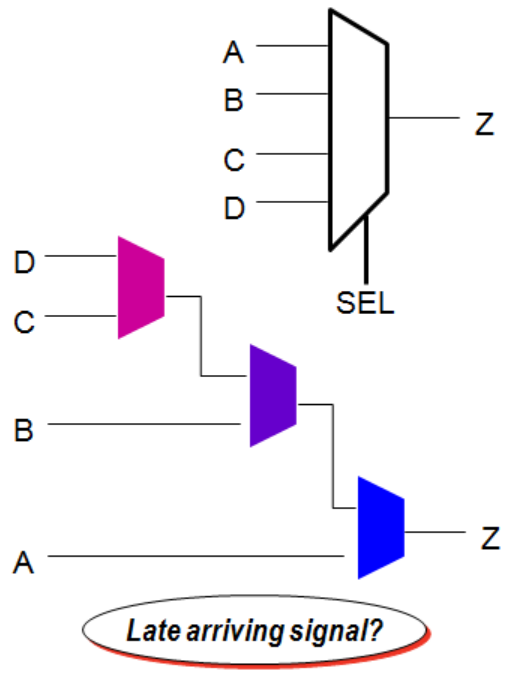
if/elsif Example and Rules

1) The first condition found to be true will be executed

2) Conditions can overlap

3) The first condition of an *if/elsif* has priority

```
process (A, B, C, D, SEL )
begin
if      (SEL = "00")  then
            Z <= A ;
elsif  (SEL = "01")  then
            Z <= B ;
elsif  (SEL = "10")  then
            Z <= C ;
elsif  (SEL = "11")  then
            Z <= D ;
end if;
end process ;
```



Because the *if/elsif* statement implies a priority among the conditions, synthesis tools may create the cascaded circuit shown on the bottom, instead of a wide multiplexer as shown on the top.

Implied priority can be useful, especially if you have a late-arriving signal. For example, if A is a late-arriving signal, the cascaded structure ensures that the late signal will experience the shortest delay through to the Z output. While true, this is a dangerous way to design. A better solution is to add a pipeline stage which would add a clock worth of latency, but remove the concern about late arriving signals.

Another, perhaps easier and better solution, is to register the output. In this way, the tools will compute the maximum delay through all of the paths and indicate if it will meet timing or not.

If you do not intend to create a cascaded circuit, make sure that the conditions do not overlap. This should be sufficient to avoid priority encoding by most FPGA-centric synthesis tools. For greater portability, you might consider using a *case* statement (described next).

© Copyright 2019 Xilinx

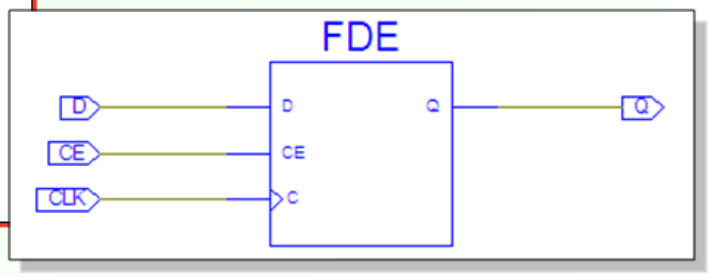
Note that the VHDL standard has no influence on how the synthesis vendor implements this. With today's technology, this could be implemented as either multiplexers or LUTs.

Slide 11-6:

Clock Enable Inferred

- The following example shows how to infer the CE resource

```
process ( CLK )
begin
 if  rising_edge ( CLK ) then
  if  ( CE = '1' )  then
    Q <= D ;
  end if ;
 end  if ;
end process;
```



Note that an *else* clause is not necessary

The *else* clause is not necessary because this is a clocked process, and therefore models a storage element.

Note: Make sure that you have a higher width of data; the synthesis tool may not infer FDE for a smaller data width.

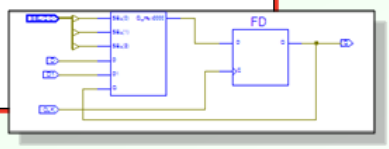
© Copyright 2019 Xilinx

Slide 11-7:

Avoiding CE Inference

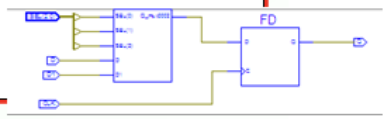
- The following example shows how to avoid CE inference

```
process (CLK) begin
if rising_edge (CLK) then
  if (SEL = "000") then
        Z <= A ;
  elsif (SEL = "010") then
        Z <= B ;
  end if ;
end if ;
end process ;
```



Creates CE (clock enable) from signal SEL for register driving output Z

```
process (CLK) begin
if rising_edge (CLK) then
  Z <= '0' --default
  if (SEL = "000") then
        Z <= A ;
  elsif (SEL = "010") then
        Z <= B ;
  end if ;
end if ;
end process ;
```



Default assignment to Z prevents CE inference

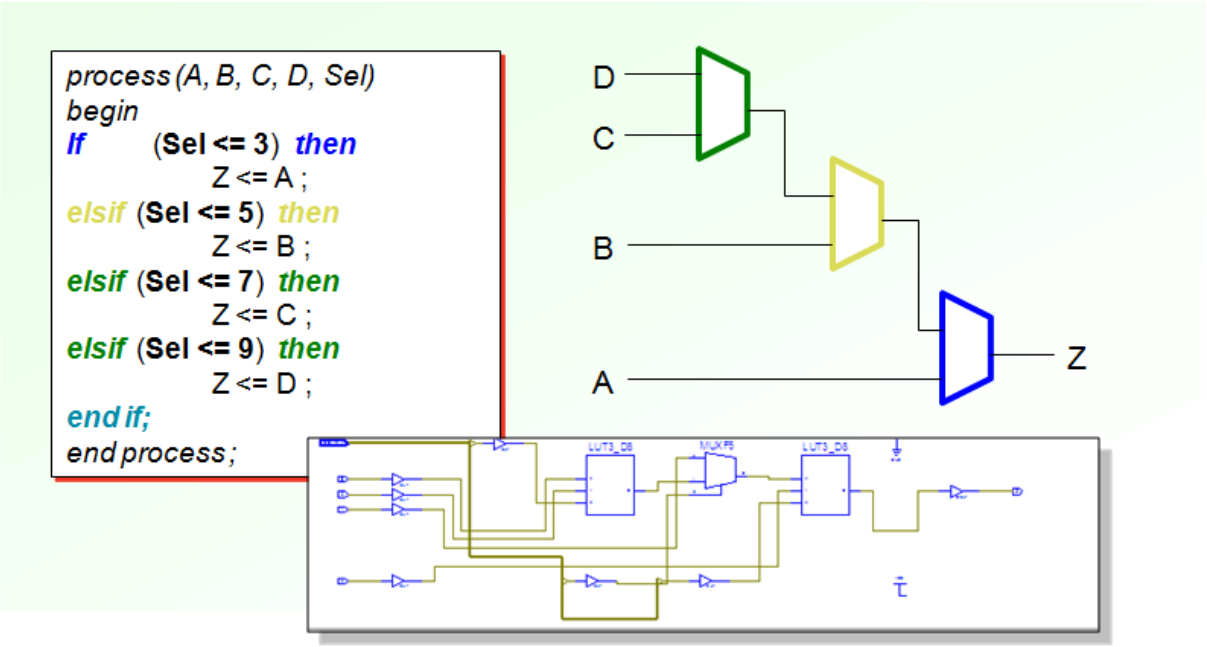
This assumes that if SEL is neither "000" nor "010," then the result should be '0' rather than leaving the flip-flop unchanged.

© Copyright 2019 Xilinx

Slide 11-8:

Overlapping Conditions

- If conditions overlap, an *if/elseif* statement is necessary



Here is an example of overlapping conditions. Clearly, any value of Sel that is less than 3 will also be less than 5. The *if/elsif* statement can handle this because only the first true condition will be executed.

This code could be written as a *case* statement, but it would require more specific, non-overlapping conditions:

```
case (Sel) is
when 0 to 3 => Z <= A;
when 4 | 5 => Z <= B;
when 6 | 7 => Z <= C;
when 8 | 9 => Z <= D;
end case;
```


© Copyright 2019 Xilinx

Slide 11-9:

Selected Signal Assignment

- A selected signal assignment is a concurrent form of the *case* statement

- The same rules exist as for the *case* statement
  1. All conditions must be specified
  2. No conditions may overlap

```
architecture...
 begin
   process (A,B,C, SEL)
   begin
      case (SEL) is
      when "00" => Z <= A;
      when "10" => Z <= B;
      when "11" => Z <= C;
      when others => Z <= 'X';
    end case;
  end process;
end architecture;
```

```
architecture...
  begin
    with SEL select
      Z <=  A  when "00",
            B  when "10",
            C  when "11",
            'X' when others;
end architecture;
```

The std_logic assignment of **X** must be uppercase.

VHDL-2008 provides a way to deal with explicit "don't care" assignments. When using the `select?` statement, the Vivado® tool now evaluates explicit don't cares, for example:

```
process(clk) begin
   if clk'event and clk='1' then
      with my_reg select?
           out1 <= in1 when "11--",
           in2 when "000-",
           in3 when "1111",
           in4 when others;
   end if;
end process;
```

Note: For this statement to work, the signal in question must be assigned an explicit don't care.

© Copyright 2019 Xilinx

Slide 11-10:

Case Statement

- The *case* statement allows operations to be performed based on the value of the selector expression
- Equivalent to the concurrent statement *with/select*
- Syntax

```
case ( <selector expression> ) is
  when <target> =>
            <sequential statements >;
  . . .
  when others =>
            <sequential statements >;
end case ;
```

- Usage
  - Ranges may not be used as <targets>
  - Only valid within a process/function/procedure
  - "when others=>" used to catch conditions not explicitly identified
  - By default, many synthesis tools will optimize away the "others" portion when used to build a finite state machine
    - Must be manually turned on via the synthesis tools option
    - Done to improve performance

Syntax:

- <selector expression> - any predefined legal expression composed of either signals, variables, or both
- <target> - any explicitly defined value whose type matches <selector expression>
- <sequential statements> - any single or multiple statements that are legal within a process

Note that although the example shows parentheses around the selector expression, this is not required and may be flagged as a warning in the Mentor Graphics ModelSim simulator.

© Copyright 2019 Xilinx

You can also avoid **when others** if the *case* statement is fully enumerated. That is, all possible combinations or values have been accounted for and explicitly listed as a <target> of a *when* clause.

For example, if a *case* statement is used to test the value of a std_logic, then a *when others* condition is required when only the '0' and '1' condition of the <selector expression> is identified. This is because a std_logic has nine different values and each must be explicitly declared in order to avoid using the *when others* clause.

VHDL-2008 provides a way to deal with explicit don't care assignments. When using *case?*, the Vivado synthesis feature now evaluates explicit don't cares in the code.

Slide 11-11:

Case Statement Ranges

- Sometimes multiple conditions can be satisfied with a single behavior of a *when* clause

- Often these can be a range such as values between 1-10, or 'A'-'F'
  - Type must be ordered, such as integer or character
  - For example, when 0 to 7 =>
  - For example, when 'A' to 'F' =>

- Or specific patterns such as "010001" and "101110"
  - Almost any type may be used
    - For example, when "010001" | "101110"

Be careful when using "real" type as different systems may represent real values differently. It is best to use *if* statements when testing real values against ranges. Testing real values for equality or inequality may lead to problems due to how they are represented internally.

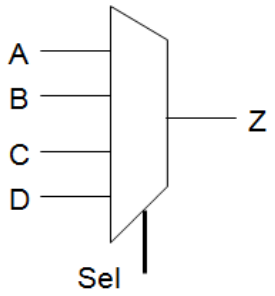
© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 11-12:

Case Examples and Rules

- All possible values of the selector expression must be specified
  - When "others" may be used to catch unspecified selectors
  - Unspecified selection may infer storage
- No conditions can overlap
- All range specifications must be of a discrete type
- *case* statements are preferable for LUT-based architectures
  - Synthesis tools generally produce a multiplexer



Assume bit type for all signals...

```
process (A, B, C, D, SEL)
begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when "11" => Z <= D ;
end case ;
. . .
end process ;
```

Is this sufficient for std_logic?

Remember that signals of type std_logic can have values other than 0 and 1, which means that most *case* statements will need to have a *when others* clause. This example assumes that SEL is a std_logic_vector(1 down to 0). This means that there are 9^2 possible combinations. An "others" clause is required to catch the 88 undefined conditions such as "-U" or "ZW".

Since synthesis will not generate logic to test for any condition other than '0' or '1', the above may be acceptable; however, this may lead to a simulation mismatch (that is, simulation generating different output than the hardware). The *when others* statement will generally be optimized out of synthesis if it does not contain the '0' or '1' values, so there is little if any impact on the logic resources.

© Copyright 2019 Xilinx

VHDL-2008 provides a new form of the *case* statement that uses the predefined "?=" operator to compare choice values with the expression value. It deals with the explicit don't care assignments.

A question mark symbol is included after the keyword *case*, as follows:

```
case? expression is
when "0---" =>        Covers all options from 0x0 to 0x7
...
end case?;
```

Slide 11-13:

Default Assignments

- Any conditional expression, *if/else* or *case*, should have the default explicitly stated to avoid compiler ambiguity and inferred latches

- However, there are two equally important considerations for the default
  - Simulation: Visibility, should not obscure or mask problems
  - Synthesis: Optimization, no additional logic, decreased performance

```
process ( A, B, C, D, SEL ) begin
if     ( SEL = "00" ) then Z <= A ;
elsif  ( SEL = "01" ) then Z <= B ;
elsif  ( SEL = "10" ) then Z <= C ;
elsif  ( SEL = "11" ) then Z <= D ;
end if ;
end process ;
```

```
process ( A, B, C, D, SEL ) begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when "11" => Z <= D ;
end case ;
end process ;
```

  - Given this, which example on the next slide best meets both objectives?

© Copyright 2019 Xilinx

Slide 11-14:

Apply Your Knowledge

4) Which default assignment for the *if/else* or *case* conditional statement best satisfies both objectives? Discuss the simulation/synthesis implications of each

EX.1

```
process ( A, B, C, D, SEL ) begin
if     ( SEL = "00" ) then Z <= A ;
elsif  ( SEL = "01" ) then Z <= B ;
elsif  ( SEL = "10" ) then Z <= C ;
else                       Z <= D ;
end if ;
end process ;
```

EX.2

```
process ( A, B, C, D, SEL ) begin
if     ( SEL = "00" ) then Z <= A ;
elsif  ( SEL = "01" ) then Z <= B ;
elsif  ( SEL = "10" ) then Z <= C ;
elsif  ( SEL = "11" ) then Z <= D ;
else                       Z <= 'X';
end if ;
end process ;
```

EX.3

```
process ( A, B, C, D, SEL ) begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when "11" => Z <= D ;
  when others => Z <= 'X' ;
end case ;
end process ;
```

EX.4

```
process ( A, B, C, D, SEL ) begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when others => Z <= D;
end case ;
end process ;
```

When would you use a *case* statement instead of an *if/elsif/else* statement? *if* statements are better at comparing ranges, while *case* statements are good for explicitly listing individual situations where there is no possibility of overlap.

© Copyright 2019 Xilinx

Slide 11-15:

Answer

5) Which default assignment for the *if/else* or *case* conditional statement best satisfies both objectives? Discuss the simulation/synthesis implications of each

EX.1

```
process ( A, B, C, D, SEL ) begin
if     ( SEL = "00" ) then Z <= A ;
elsif  ( SEL = "01" ) then Z <= B ;
elsif  ( SEL = "10" ) then Z <= C ;
else                       Z <= D ;
end if ;
end process ;
```

EX.2

```
process ( A, B, C, D, SEL ) begin
if     ( SEL = "00" ) then Z <= A ;
elsif  ( SEL = "01" ) then Z <= B ;
elsif  ( SEL = "10" ) then Z <= C ;
elsif  ( SEL = "11" ) then Z <= D ;
else                       Z <= 'X';
end if ;
end process ;
```

EX.3

```
process ( A, B, C, D, SEL ) begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when "11" => Z <= D ;
  when others => Z <= 'X' ;
end case ;
end process ;
```

EX.4

```
process ( A, B, C, D, SEL ) begin
case SEL is
  when "00" => Z <= A ;
  when "01" => Z <= B ;
  when "10" => Z <= C ;
  when others => Z <= D;
end case ;
end process ;
```

Example 1: Not so good. Most designers tend to make the last valid choice the default; however, this decreases visibility during simulation by masking problems on the SEL signal or the logic that drives it. It is okay for synthesis, not so good for simulation.

Example 2: Best overall for *if/else*. The std_logic value X is readily visible in simulation and alerts you to problems with the SEL signal. At the same time, synthesis will ignore this and no additional logic is created. It is good for synthesis and good for simulation.

Example 3: Best overall for case. The value std_logic value X is readily visible in simulation and alerts you to problems with the SEL signal. At the same time, synthesis will ignore this and no additional logic is created. It is good for synthesis and good for simulation.

Example 4: Not so good. This is a common approach, making the last valid choice the default. However, in simulation it masks problems on the SEL signal by always passing valid data through. It is okay for synthesis, not so good for simulation.

© Copyright 2019 Xilinx

Slide 11-16:

Specifying Ranges – Apply Your Knowledge

- If a range is to be specified as a condition of a *case* or *if/else* statement, it must be of a discrete type

```
process   (...)
  begin
    case  ( selector expression )  is
      when  0 to 7            => sequential statements ;   ?
      when  4.3 to 7.7        => sequential statements ;   ?
      when  "1000" to "1010"  => sequential statements ;   ?
      when  "1000" | "1010"   => sequential statements ;   ?

      . . .
      when others             => sequential statements ;   ?
    end case;
end process ;
```

Are the ranges specified in the above code correct?

- when 0 to 7
- when 4.3 to 7.7
- when "1000" to "1010"
- when others

Slide 11-17:

Specifying Ranges

- If a range is to be specified as a condition of a *case* or *if/else* statement, it must be of a discrete type

```
process   (...)
  begin
    case  ( selector expression )  is
      when  0 to 7            => sequential statements ;   OK
      when  4.3 to 7.7        => sequential statements ;   ERROR
      when  "1000" to "1010"  => sequential statements ;   ERROR
      when  "1000" | "1010"   => sequential statements ;   OK

      . . .
      when others             => sequential statements ;   OK
    end case;
end process ;
```


© Copyright 2019 Xilinx

when 4.3 to 7.7 is not allowed because real numbers are not discrete.

when "1000" to "1010" is not allowed because signals of type bit_vector and std_logic_vector are not interpreted as actual numbers.

The last example basically uses the pipe '|' as a separator (or) between possible choices for that branch of the case statement.

A general rule about dealing with reals: Never test for equality.

When an "exact" value is specified, the environment attempts to find the nearest real number, but, because of the limited number of bits, the "exact" value will not necessarily match.

Slide 11-18:

Summary

- Basic control structures that were discussed
  - *if/elsif/else/end if*
  - *case*

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Sequential Looping Statements

Slide 12-1:

2019.1

This module introduces the concept of looping, both within the simulation environment to produce stimulus and within the synthesis environment to produce logic. 10 minutes.

Slide 12-2:

Objectives

After completing this module, you will be able to:

- Describe the available looping structures and where they can be used
- Apply best practices in synthesizable code

© Copyright 2019 Xilinx

Slide 12-3:

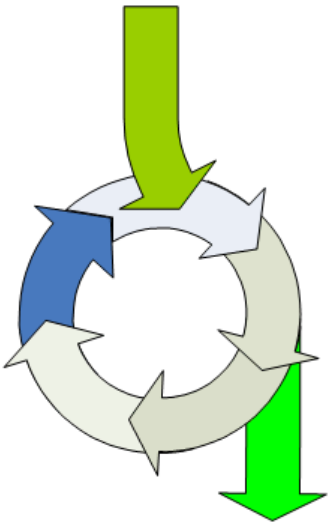
Looping Structures – Hardware versus Software

- Software
  - An inherent time cost when looping
    - Loops are evaluated in "run time"
    - Overhead due to PC jumps, comparing against limits, updating the index
  - Used to reduce the size of memory footprint
    - Secondary benefit is reducing code size for ease of coding and maintenance
- Hardware
  - Loops consume resources in synthesis
    - No run-time cost as there is no "run time" in synthesis, only existence
    - Logic is replicated per iteration of loop
  - Used to reduce tedious and redundant logic coding
  - Loops have similar trades-offs in the simulation environment, because they are "software"

Slide 12-4:

Sequential Looping

- Sequential looping structures can only exist within processes and subprograms
- Loop statements can be constructed for any repetitive operation
- Three basic sequential forms
  - Infinite loop (*loop*)
  - *for* loop
  - *while* loop
- No *do/until* structure exists
  - Can be modeled using *loop* and *exit*



© Copyright 2019 Xilinx

Slide 12-5:

Loop Statements

- There are different forms, each having a different method of control
  - The loop variable index ("i" in this example) is not separately declared, cannot be assigned to, and is not visible outside the loop
  - It is treated as a constant
  - With each loop iteration, it successively assumes the discrete values indicated in the range from left to right

```
process ( A, B_BUS )
 begin
  for i in 7 downto 0 loop
    C_BUS (i) <= A and B_BUS (i) ;
  end loop ;
end process ;
```

The loop variable index (i) can only be seen inside the loop. For example, a nested or inner loop can read the value of the outer loop.

For those with software backgrounds, it may be worth noting that the loop is “unrolled” during synthesis and does not actually loop during run time.

VHDL also supports while…loop statements in simulation.

There is also the reasonably obscure “infinite loop.” This loop has no iteration scheme. It will be executed continuously until it encounters an exit or next statement.

```
[loop_label :] loop
  sequential statements
  [next   [label] [when condition];
  [exit   [label] [when condition];
end loop [loop_label];
```

Loops can be used in both simulation and in synthesis. When used in synthesis, the loop parameters must be static and will generate logic. Simulation does not place such restrictions on the loop parameters.

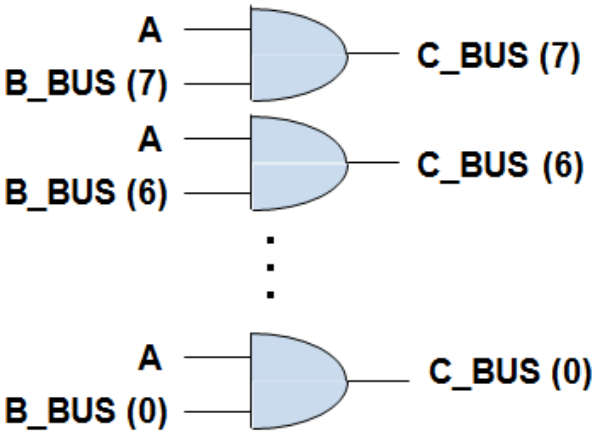
© Copyright 2019 Xilinx

Slide 12-6:

Using Loop Statements

- For synthesis, a loop is *unrolled*
  - Logic resulting from each loop iteration is synthesized
- For simulation, loop statements provide a very flexible method for behavioral modeling
- Loop statements are also used extensively in subprograms (functions and procedures)

```
process ( A, B_BUS )
 begin
  for i in 7 downto 0 loop
    C_BUS (i) <= A and B_BUS (i) ;
  end loop ;
end process ;
```



© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 12-7:

Deliberate Infinite Loops

- Processes form a type of infinite loop
  - But they are re-evaluated prior to entering the next cycle via
    - The process sensitivity list or other *wait* construct
- Deliberate infinite loop syntax

```
[<loop label>:] loop
        <sequential statements>
    end loop [<loop label>];
```

- Not quite so infinite when an "*exit*" statement is contained in the loop
  - Works in a similar fashion to a software break
  - Forces exit from the loop
- Exit syntax

```
[<exit label>:] exit [<loop label>] [when <condition>];
```

- Next statements skip the remainder of the loop

```
[<next label>:] next [<loop label>] [when <condition>];
```

Syntax:

- <loop label> - any unique, legal identifier. If the loop label is specified, then the opening loop label must match the closing loop label.
- <next label>, <exit label> - any unique, legal identifier.
- <condition> - any legal construct which produces a boolean result.

Loops can exist on a single line. The syntax is shown on multiple lines because this is the way it is usually coded.

Some infinite loops are desirable, such as clock generation and other constant periodic functions. Other constant periodic functions can include pseudo-noise generators or pseudo-random number generators.

Signal-over-power lines require a 60-Hz analog sine wave in which digital information is super-imposed. While data may not always be present, the 60-Hz carrier must always be generated during the life of the simulation.

It should be noted that deliberate infinite loops are only intended for the simulation environment.

© Copyright 2019 Xilinx

Slide 12-8:

Accidental Infinite Loops

- "Timeless" infinite loops are problematic
  - Looping without regard to external conditions or time
    - Will cause a delta-cycle loop in zero simulation time (but infinite real time)
  - Due to not having a wait statement in the control flow of a process

- Will cause some simulators to lock up
  - Depending on how the infinite loop is entered, it may be possible to single step to the point of lock up
  - Many simulators have a user selectable "delta" cycle limit option to avoid lock-up

Bottom line: Every control path in a process must have some form of *wait* statement.

Slide 12-9:

Example of Accidental Infinite Loop

```
oopsLoop: process
    variable new_value : integer range 0 to 256 := 0;
    variable textOut   : line;
    file fp : text;
begin
    file_open(fp, "c:\training\loop_log.txt", WRITE_MODE);
    signal_to_drive <= 0;
    wait for 1 us;
    while (device_is_active) loop
        new_value := new_value + 1;
        if (new_value > 255) then
            new_value := 0;
        end if;
        signal_to_drive <= new_value;
        write(textOut,"time now is " & time'image(now) &
              " and the value count is " & integer'image(new_value));
        report("time now is " & time'image(now) &
              " and the value count is " & integer'image(new_value));
        writeline(fp,textOut);
    end loop;
    wait for 10 us;
    file_close(fp);
end process oopsLoop;
```

AOL.txt | loop_log.txt

```
time now is 1000000 ps and the value count is 1
time now is 1000000 ps and the value count is 2
time now is 1000000 ps and the value count is 3
time now is 1000000 ps and the value count is 4
time now is 1000000 ps and the value count is 5
time now is 1000000 ps and the value count is 6
time now is 1000000 ps and the value count is 7
time now is 1000000 ps and the value count is 8
time now is 1000000 ps and the value count is 9
```


© Copyright 2019 Xilinx

Assume device_is_active is a boolean signal which is true at this instant in time.

Notice that once the *while* loop is entered, it can be exited when device_is_active becomes false; however, there is no “simulation time” delay. That is, as soon as new_value has been incremented and displayed, the loop forces the next increment and display without any simulation time passing.

This is witnessed in the console output as the wait statement just prior to the while loop beginning allows 1 us of simulation time to occur. Then, time ceases to advance although new_value continually cycles through its 0 to 255 count.

Once device_is_active evaluates to false, this loop is exited and forced to delay 10 us of simulation time (post while loop) plus an additional 1 us of simulation time prior to being re-evaluated.

Depending on the simulator, this could cause other concurrent statements to halt operation, thus never seeing devices_is_active deassert.

The solution is to put a wait statement inside the control path (*while* loop).

Slide 12-10:

for loop

- Used when exact number of iterations is known
  - Although it can be exited early with an exit statement
- Loop variable
  - Intrinsically declared to be of proper type
  - Only valid inside the loop
- Syntax

```
[<loop label>:] for <identifier> in <range> loop
   <sequential statements>
end loop [<loop label>];
```

- Example

```
forLoopEx: process
    variable textOut: line;
  begin
    for j in 0 to 5 loop
      write(textOut, "the value of j is "
            & integer'image(j));
      writeline(OUTPUT,textOut);
    end loop;
    wait;
  end process forLoopEx;
```

```
Console
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
the value of j is 0
the value of j is 1
the value of j is 2
the value of j is 3
the value of j is 4
the value of j is 5
Finished circuit initialization process.
ISim>
```


© Copyright 2019 Xilinx

Syntax:

- <loop label> - any unique, legal identifier. Loop labels must match, if used.
- <identifier> - any unique, legal identifier. Is intrinsically declared.
- <sequential statement> - any collection of legal sequential statements including other loops.

Example:

Note that the range is completely covered 0 through 5 inclusive. This is a difference compared to the way most software programmers write code. An equivalent C or Java for statement would be: for (j = 0; j<= 5; j++) { … }.

Also note the final *wait* statement. Without this *wait*, this process becomes an "accidental" infinite loop.

Additional example:

```
doAND: process (a, b)
        begin
           for i in 0 to width-1 loop
              q(i) <= a AND b(i);
        end loop;
end process doAND;
```

This example ANDs a against all the members of std_logic_vector b.

© Copyright 2019 Xilinx

Slide 12-11:

while Loop

- *while* loops are typically used when the final number of iterations is not known
- Entered when a condition set is met
  - Unlike a "loop" with an exit statement where the loop is always entered, but exits when a condition set is met
- Syntax

```
[<loop label>:] while <condition> loop
   <sequential statements>
end loop [<loop label>];
```

- Example

```
makeClk: process
  begin
    genClk: while not system_reset loop
      clk <= '0'; wait for 5 ns;
      clk <= '1'; wait for 5 ns;
    end loop genClk;
  end process;
```

Syntax:

- <loop label> - any unique, legal identifier. Loop labels must match, if used.
- <condition> - any legal statement which evaluates to a boolean.
- <sequential statement> - any collection of legal sequential statements including other loops.

© Copyright 2019 Xilinx

Slide 12-12:

Nested Loops

- Loops can be nested
  - Good design practice calls for using loop labels
    - Especially when next and exit statements are used so that the statement is applied to the proper loop
  - Loops do not need to be of the same type to be nested
  - *while* loops can reside in *for* loops which may reside in loops, etc.

```
nesting: process (a, b)
  begin
    iloop: for i in 0 to width-1 loop
      jloop: for j in 0 to 15 loop
        q(i,j) <= i * j;
      end loop jloop;
    end loop iloop;
  end process nesting;
```

"width" could be a constant or variable (in a simulation as it would need to be static for synthesis). Although width is a commonly used generic name, it does not have to be as shown in this example.

Also note the use of labels, which help identify where each loop ends.

Slide 12-13:

Apply Your Knowledge

1) Under what conditions would you use a
   - *for* loop
   - *while* loop
   - *loop* with exit

2) When would you code a loop as a concurrent statement rather than in a process?

© Copyright 2019 Xilinx

1) Under what conditions would you use a
   - *for* loop – When the number of iterations is known and finite.
   - *while* loop – When the number of iterations is not known.
   - *loop* with exit – Similar to a *while* loop, but more closely resembles an *until* loop. *while* loops may not execute depending on how the condition evaluates whereas a *loop with exit* will execute once and subsequent passes through the loop are controlled by the *exit* statement.

2) When would you code a loop as a concurrent statement rather than in a process?
   Concurrent looping is generally used for instantiating multiple copies of a component. This capability does not exist within a process.

Slide 12-14:

Summary

- Loop structures have a sequential form and are written inside processes
- Available loops include infinite loop, *for* loop, *while* loop, and more

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Delays in VHDL: Wait Statements

Slide 13-1:

2019.1

This module covers the *wait* statement and how it controls the execution of the *process* statement. 10 minutes.

Slide 13-2:

Objectives

After completing this module, you will be able to:

- Describe delay types available in VHDL
- Define the four versions of the *wait* statement
- List the key concepts for good coding style with respect to synchronous processes

Slide 13-3:

Delays in VHDL

- Signals are assigned values using the signal assignment statement
  - This signal assignment can be delayed, or performed after a certain amount of time
- Two types of delay models available in VHDL
  - Inertial delay
    - Models the delays found in the switching circuits
  - Transport delay
    - Represents pure propagation delays

© Copyright 2019 Xilinx

Slide 13-4:

Inertial Delay

- Inertial delay model
  - Default delay model in digital circuits
  - Used to model the switching circuits
  - Value will be assigned to signal after specified time
  - Input must be stable in order to get recognized, otherwise will be ignored

- Syntax

```
y <= a after <time>;

y <= 5 after 10 ns;
y <= inertial 5 after 10 ns;
```

Slide 13-5:

Transport Delay

- Transport delay model
  - Used to represent propagation delays or routing delays
  - Keyword *transport* is required

- Syntax

```
y <= transport a after <time>;

y <= transport 5 after 10 ns;
```


© Copyright 2019 Xilinx

Slide 13-6:

General Form of the Wait Statement

- Although there are four commonly used forms of the *wait* statement, Standard defines the *wait* statement as

```
[ <label> : ] wait [ <sensitivity_clause> ] [ <condition_clause> ] [ <timeout_clause> ] ;
```

- The <sensitivity_clause>, if present, enables the action in the <condition_clause>, if present
  - If neither the <sensitivity_clause> nor the <condition_clause> is met by the time specified in <timeout_clause>, then the *wait* is exited

- This implies that more complex wait structures can be created, such as

```
wait until test_passed = '1' for 10 us;
```

  - and

```
stayHere: wait on A, B until C = '0' for clock_period*100;
```

Syntax:

- <label> is a legal label identifier.
- <sensitivity_clause> is an expression that lists signals to monitor for changes (on <sensitivity_list>).
- <condition _clause> is an expression that evaluates to a boolean (until <condition>).
- <timeout_clause> is an expression that evaluates to a time (for <time_expression>).

Example 1:

This statement suspends the process until test_passed attains the value of '1', if the condition has not been met in 10 micro-seconds, then the *wait* exits.

In the absence of an explicit sensitivity clause, there will be an implied sensitivity list on test_passed. This means that if test_passed is already at '1', then this *wait* statement will hold the process in suspension until the timeout clause is reached.

Example 2:

This statement waits for a change on A or B then evaluates C against '0'.

© Copyright 2019 Xilinx

C is only tested against ‘0’ when there is an event on A or B. If C should change to 0 when neither A nor B has an event, C will not be re-evaluated and the *wait* state will remain “active” until the timeout.

If none of these conditions are met by 100 clock periods (which must be defined in terms of time earlier in the testbench), then the *wait* statement is exited. If C is included in the sensitivity list, then the *until* expression will be evaluated when C changes state.

Slide 13-7:

Suspending Processes

- *Every* process must have some means of triggering that is indirectly the means of suspension
  - This forms the *wait* condition for that process
- For behavioral modeling, the explicit *wait* statement is often used
  - There are four "basic" forms of the *wait* statement

wait on...	An event on given signal	***wait on*** **A, B ;**
wait until...	A specific condition	***wait until*** **CLK = ‘1’ ;**
wait for ...	A specified time amount	***wait for*** **10** ***ns*** **;**
wait	Indefinite suspension	***wait*** **;**

- Explicit *wait* statements are generally the purview of simulation
  - Some synthesis tools allow the explicit *wait* statement in lieu of the implied *wait* statement of the sensitivity list

© Copyright 2019 Xilinx

Slide 13-8:

Synthesized Wait Statement Example

- When carefully executed, *wait* statements can be synthesized
- Use with caution as this is not a common practice and may cause confusion to the reader
  - In some cases, coding in this style will make the point easier to understand

```
-- A different style of inferring a register
-- This make an implied state machine that alternates between
-- clocking d into q and q2
altAsgn: process
   begin
      wait until clock = '1';
      q        <= d;
      wait until clock = '1';
      q2       <= d;
   end process altAsgn;
```

Slide 13-9:

Wait

- Waits forever
- Syntax
  - *wait;*
- Typically used to suspend (in this case halt) a process until the end of simulation

© Copyright 2019 Xilinx

Slide 13-10:

Wait For

- The most commonly used form of wait
- Delays for a specific amount of time as measured from the present time
- Also known as the "timeout"
  - Useful when combined with other *wait* statements
- Syntax

```
wait for <time description>;
```

- Typical uses
  - Delaying a fixed amount of time: wait for 100 us;
  - Delaying a fixed number of clock cycles: wait for CLOCK_PERIOD * 25;
  - Introducing jitter: wait for CLOCK_PERIOD * (1 + rand_number_minus1_to_1 / 100.0);

Syntax:

- <time description> - any legal construct which evaluates to the physical type of time.

Typical use descriptions:

- wait for 100 us; – This construct suspends the execution within a process for 100 microseconds.
- wait for CLOCK_PERIOD * 25; – This construct assumes that CLOCK_PERIOD is defined as a time. The result is to suspend the process for 25 times whatever the clock period is. This technique is useful when the exact clock period has not been decided upon.

  constant CLOCK_PERIOD : time := 10 ns; -- defines a 100-MHz clock
- wait for CLOCK_PERIOD * (1.0 + rand_number_-1_to_1 / 100000.0); – this is one mechanism for generating jitter.

  Assuming a function which produces a real number between -1 and 1, this function will produce a 10-ppm jitter.

© Copyright 2019 Xilinx

Some prefer to use a pseudo-random function generator using linear feedback shift registers (LFSRs). This provides for a repeatable jitter and guarantees both a maximum and minimum value of a given time.

Notice how time can be multiplied by an integer (from the previous example) and a real.

Slide 13-11:

Wait On

- The process sensitivity list is a form of an implied wait condition
  - If a *wait* statement is used inside a process, the process may NOT have a sensitivity list
- Processing is suspended until any change is detected on the listed signals
- Syntax
  - [<label>:] wait on <signal list>;

```
Wait on new_data;
```

- Example of two equivalent processes

```
ex1: process ( x, y, z )
begin
   -- sequential statements
process ex1;
```

```
ex2: process
begin
   -- sequential statements
   wait on x, y, z;
end process ex1;
```

- This construct is typically used to wait for a specific event to occur whose timing may be asynchronous
  - For example, waiting for a change in a signal before proceeding

Syntax:

- <label> - any legal identifier.
- <signal list> - a comma separated list of previously defined signals.

© Copyright 2019 Xilinx

Slide 13-12:

Wait Until

- Processing is suspended until a SPECIFIC change is detected
  - The condition must evaluate to a boolean

- Syntax

```
[<label>:] wait until <condition>;
```

- This construct is typically used to wait for a specific event to occur whose timing may be asynchronous

```
ex3: process
Begin
   -- stimulus initialization
   wait until reset = '0';
   -- stimulus generation
end process ex1;
```

- The *until* has a built-in *wait on* statement
  - Implies that the condition must be met after the process suspends

Syntax:

- <label> - any legal identifier.
- <condition> - any construct that returns a boolean.

Note that the condition can be an equation with multiple signals of various types. The only requirement is that it evaluates to a boolean.

In this example, "asynchronous" is meant to mean "it will occur at some unknown time", but not necessarily asynchronous to a clock.

wait until X = '1'; is equivalent to saying "wait for an event on X, then when X changes, is X = '1'?, if so, then resume from wait."

In implementation a sensitivity list is built from the <condition> then effectively prepended to the *until* statement.

© Copyright 2019 Xilinx

Slide 13-13:

Summary

- Four types of *wait* statements
  - *wait*
  - *wait for*
  - *wait on*
  - *wait until*

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Introduction to the VHDL Testbench

Slide 14-1:

2019.1

This module introduces the concept of the VHDL testbench. 20 minutes.

Slide 14-2:

Objectives

After completing this module, you will be able to:

- Define a testbench
- Identify the basic components of a testbench
- Write a simple testbench

Slide 14-3:

What is a Testbench?

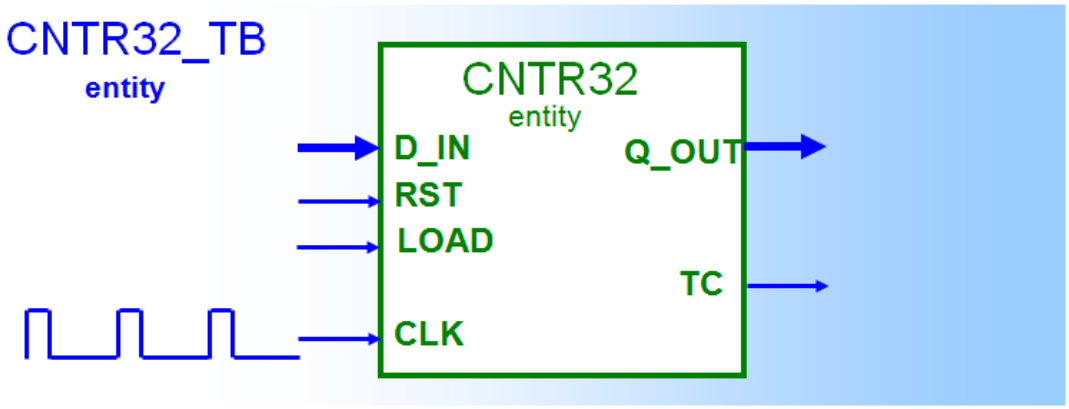
- VHDL can be used as a test language that performs verification of the design under test (DUT)
- A key advantage of HDLs is the ability to verify the design before implementation
- To simulate the designs with stimulus you need to produce a testbench
- A testbench is a specification in VHDL that plays the role of a complete simulation environment for the DUT

© Copyright 2019 Xilinx

Slide 14-4:

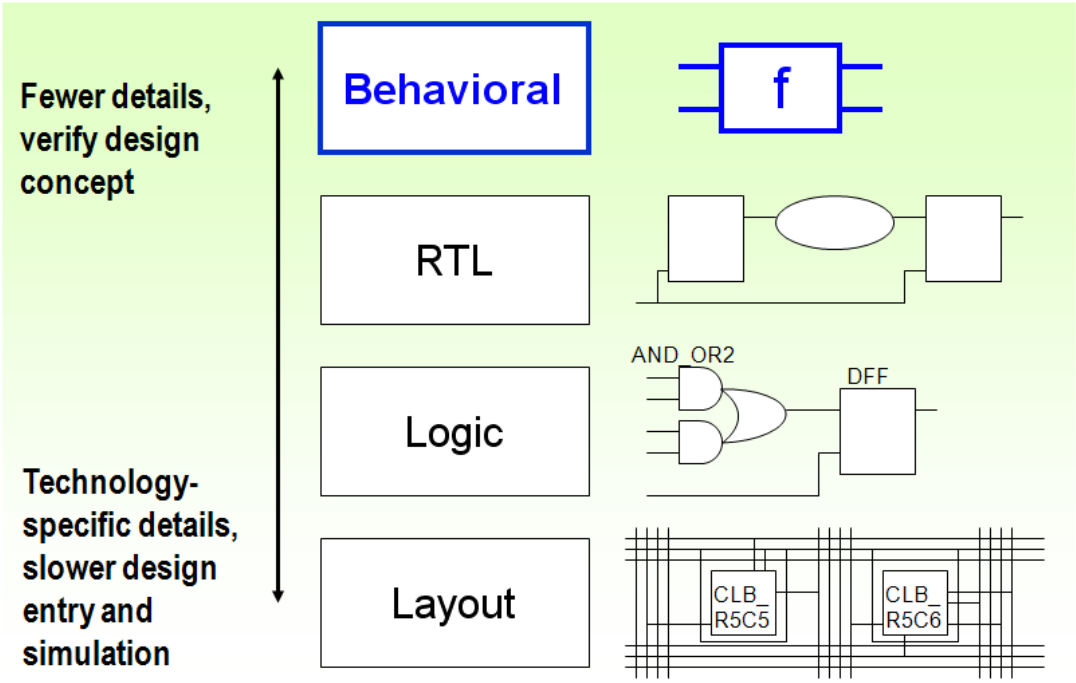
Testbench

- Testbenches form the cornerstone of functional validation
- A VHDL testbench is a virtual *test bed*
  - An upper-level hierarchical VHDL structure that applies input stimulus to a unit under test (UUT) and monitors the output to verify functionality



Slide 14-5:

Application of a Testbench



© Copyright 2019 Xilinx

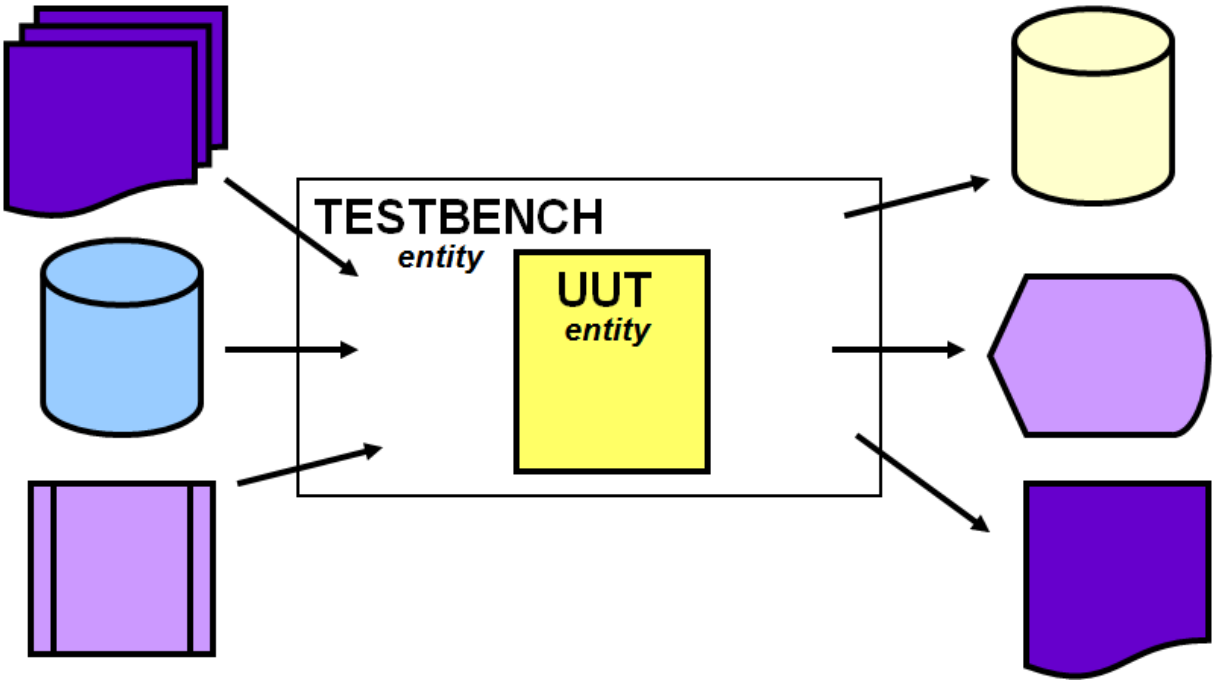
A testbench is a purely behavioral concept. Nothing within the testbench itself can be synthesized.

Synthesizable code must be at the RTL level or lower. This is required to actually generate the logic.

Slide 14-6:

Simulation I/O Flexibility

- VHDL provides considerable flexibility for declaring and accessing files, along with various formats of data inputs and outputs



- Text-based data operations are specified in the textio package

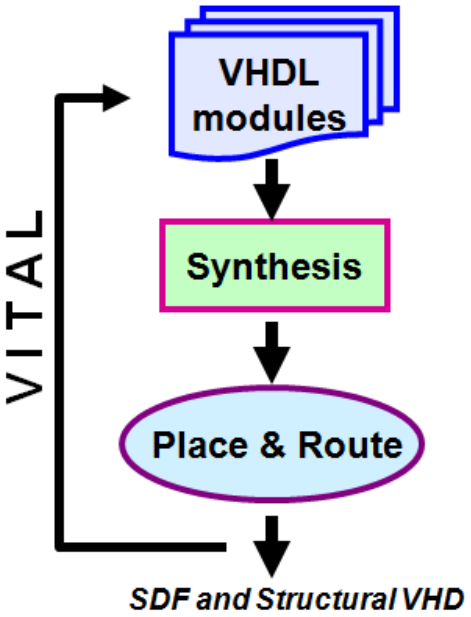
Given that VHDL is first and foremost a means for hardware modeling, the language contains various resources for formatting, reading, storing, dynamically allocating, comparing, and writing simulation data. This includes input stimulus and output results.

© Copyright 2019 Xilinx

Slide 14-7:

Testbench Consistency

- Ideally, the same testbench should be applied at each stage of verification



1. Behavioral/RTL simulation: Execute RTL source code

2. Post-synthesis VHDL simulation: Execute structural VHD file

3. VHDL timing simulation: Execute post-layout structural VHD and SDF file

An important goal is to maintain as much consistency as possible throughout the design verification process. Creating a testbench that is robust enough to be applied at each stage of verification helps achieve this objective.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 14-8:

Behavioral Coding

- A testbench has advantages that are not available in RTL-level code or actual hardware
  - Values stored in signal
    - ***signal RST: std_logic := '0' ;***
  - Model bus input and output
    - ***D_IN <= X"FFFF0000" ;***
  - No need to consider propagation delay
  - Model timing as necessary
    - ***Z <= A and B after 2 ns ;***
  - Arbitrary sequencing of events
    - ***RST <= '1' after 25 ns, '0' after 50 ns ;***

This shows some basic examples of statements that would be legal for simulation (or a behavioral hardware model) but not allowed in synthesis. These statements enhance your flexibility within the simulation environment, allowing the executable source code to be more abstract.

For example, you can store the value in the signal itself, meaning that detailed description and operation of the register or latch is not required at this stage.

The second example also implies that no actual driver has to be modeled for the specific value being driven on to signal.

The 'X' notation specifies that data is in hexadecimal format.

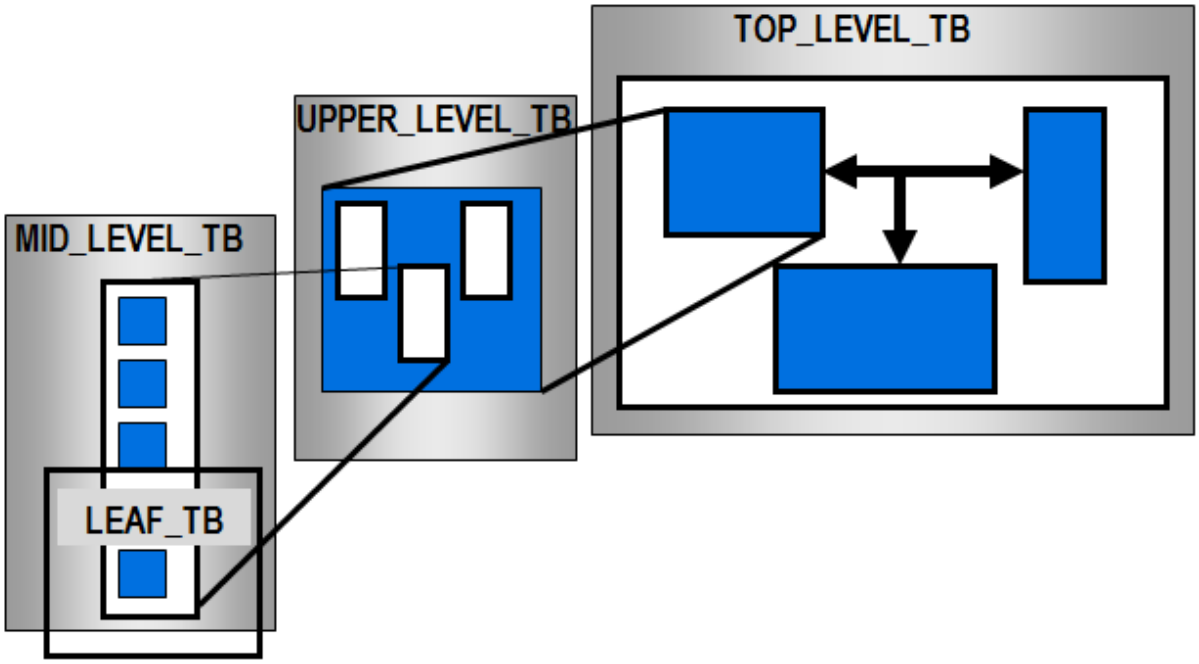
Note that the syntax used to initialize a signal value for simulation can also be used within Vivado® synthesis to define a given register value at the end of the Xilinx FPGA configuration (power-up value) process.

© Copyright 2019 Xilinx

Slide 14-9:

Verification at Each Level

- Good design practice is to design from the top down and build from the bottom up
- Each hierarchical stage can be independently validated with its own testbench



In general, verifying each leaf-level module makes it much easier to diagnose interface and related problems at later stages as the design hierarchy starts to form.

Slide 14-10:

Apply Your Knowledge

1) Can an RTL model be simulated in a simulation environment?

1) Can an RTL model be simulated in a simulation environment?

Yes. The simulation environment is comprised of the full set of capabilities of VHDL whereas synthesis is limited to what can be physically realized within the FPGA.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 14-11:

Components of a Testbench

- Entity, which defines a testbench top-level structure
  - A testbench usually does not have ports
- Internal signals, which will drive the stimuli into the UUT and monitor the response from the UUT
  - Signal to drive and monitor
- UUT instantiation
- Stimuli generation
  - Write statements to create stimulus
  - Usually implemented as a series of processes
- Response monitoring and comparing
  - Self-testing statements that will report values, error, and warnings
  - Text I/O routines in VHDL

A testbench can also have other components at the same level of hierarchy as the UUT.

For example, if you are modeling an FPGA with external memory, then the FPGA is described in the UUT, and another module would be constructed to model the memory and the connections between the two. The other stimulus would be described within the testbench.

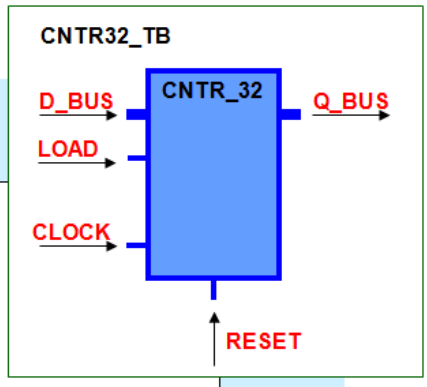
© Copyright 2019 Xilinx

Slide 14-12:

Instantiating DUT in Testbench

- A simple testbench for the counter can be constructed as follows

```
entity CNTR32_TB is
end entity CNTR32_TB ;
```




```
architecture  TEST of  CNTR32_TB  is
 -- Component declaration
component CNTR32
   port ( D_IN : in std_logic_vector (31 downto 0 );
          CLK, RST, LOAD : in  std_logic;
          Q_OUT : out  std_logic_vector (31 downto 0 );
end component  ;
 -- declare inputs, outputs and initialize them
signal  D_BUS, Q_BUS : std_logic_vector (31 downto 0 ) := (others=>'U');
signal  CLOCK, RESET, LOAD : std_logic := 'U';
 -- Instantiate Design Under Test (DUT)
begin
   UUT : CNTR32  port map ( D_IN=>D_BUS, CLK=>CLOCK, RST=>RESET,
                            LOAD=>LOAD, Q_OUT=>Q_BUS );
( continued )
```

In simulation, default initialization for type *std_logic* is U and 0 for type bit

- Entity and architecture declaration
- Signal declaration
- Instantiation of DUT into testbench
- Stimulus generation

Note the explicit initialization of the signals to 'U'. Most simulators use this as a default. If the designer wanted to have this design exit from power-on reset in a predefined stated, this is where the values would be placed.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 14-13:

Creating Clock Signals

- To create a free-running clock for a testbench
  - Declare the clock signal and initialize it to either 1 or 0

    ***signal* CLK_SIG: *std_logic* := '1' ;**

  - Create a concurrent assignment that inverts the clock signal at the appropriate interval for the intended frequency

    CLK_SIG <= *not* CLK_SIG *after* 5 ns ;

- If the clock will be other than a 50/50 duty cycle, use separate invert and wait statements within a process

This example creates a clock with a 10-ns period and 50% duty cycle. Note that this is the concurrent form of the clock stimulus example shown previously using a process.

© Copyright 2019 Xilinx

Slide 14-14:

Stimulus Generation

- The most important component of the testbench is the process that actually provides the stimulus
- Best design practice
  - Provide a separate process for groups of signals, for example
    - One process to create a clock
    - One process to create a reset
    - One or more processes to build stimulus
    - One or more processes to validate outputs

```
(continued)
 STIMULUS : process
   begin
      RESET <= '1';
      wait for  100 ns;

      RESET    <= '0';
      D_BUS <= X"FFFF0000" ;
      CLOCK  <= '1';
      wait for 25 ns;

      D_BUS <= X"00000001" ;
      CLOCK  <= '0';
      LOAD <= '1' :
      wait  for 25 ns;

      CLOCK  <= '1';
      wait ;
   end  process  STIMULUS ;
end  architecture  TEST ;
```

This is an example of how *not* to build a testbench. In this example, the designer attempts to combine all stimulus into a single process. Execution within the process is sequential, and this makes it more difficult to model the individual inputs, such as CLOCK, RESET, LOAD, and D_BUS.

Overall, this is not a good example of declaring input stimulus because all inputs are handled in the same process block. Over the next few slides, you will see how to separate these for greater flexibility and ease of use.

Wait constructs are used in process procedure. A *wait* statement causes the process to suspend.

The different forms of wait statements are:

- Wait for specific time
- Wait on sensitivity list
- Wait until condition
- Wait

© Copyright 2019 Xilinx

The explicit *wait for* allows you to arbitrarily sequence events during simulation. The process suspends at each wait condition.

The explicit *wait* is used to stop the process from looping back to the top.

Wait constructs are an excellent way for describing timing specifications. They can only be used for simulation purposes, as they are definitely not synthesizable.

Slide 14-15:

Other Clock Creation Approaches

- The example on the left shows a non-50/50 duty-cycle clock; the example on the right includes the clock signal initialization

```
architecture TEST of  CNTR32 is
  signal CLK_SIG : std_logic ;
  constant PERIOD : time := 10 ns ;
begin

  CLK_STIM: process
    begin
      CLK_SIG <= '1' ;
       wait for PERIOD * 0.4 ;
      CLK_SIG <= '0' ;
       wait for PERIOD * 0.6 ;
   end process CLK_STIM ;

(continued)
```

```
architecture TEST of  CNTR32 is
  signal CLK_SIG  : std_logic ;
  constant PERIOD : time := 10 ns ;
begin

  CLK_STIM2: process
    begin
      if  (CLK_SIG = 'U') then
         CLK_SIG <= '1' ;
         wait for PERIOD /2 ;
      end if;
      CLK_SIG <= not CLK_SIG ;
      wait for PERIOD /2 ;
    end process CLK_STIM2 ;

(continued)
```


© Copyright 2019 Xilinx

Slide 14-16:

Initializing Signals

- Any signal can be initialized by using a concurrent assignment or process

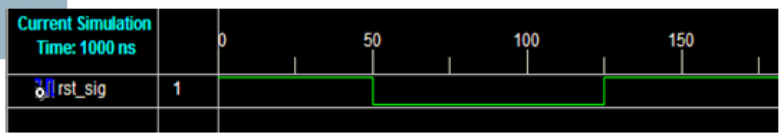
```
RST_SIG <= '1', '0' after 50 ns, '1' after 125 ns ;
```

Concurrent Statement; Time is *Absolute*

```
INIT: process
begin
   RST_SIG <= '1';
      wait for 50 ns ;
   RST_SIG <= '0' ;
      wait for 75 ns ;
   RST_SIG <= '1' ;
      wait ;
end process INIT ;
```

Sequential Statement; Time is *Relative*

In both cases, the signal RST_sig is initialized to 1 at time zero, then 0 at time 50 ns; 75 ns later, it is deasserted for the duration of the simulation



The explicit *wait* statement at the bottom of the process causes the process to suspend indefinitely, it does not loop back to the top. The process will only re-execute if and when the simulation is restarted. This RST signal was modeled as active low.

A similar approach can be used for the LOAD signal or CE, if present.

Slide 14-17:

Apply Your Knowledge

1) List the basic elements of a typical testbench

2) Which of the following declarations would not be part of a testbench?
   - Signal(s)
   - Component(s)
   - Port(s)
   - Architecture

3) How can a stimulus process be made to repeat throughout simulation?

© Copyright 2019 Xilinx

1) List the basic elements of a typical testbench.
   - Top-level entity
   - UUT
   - Signals for inputs and outputs
   - Input stimulus

2) Which of the following declarations would not be part of a testbench?
   - Signal(s)
   - Component(s)
   - ***Port(s)***
   - Architecture

3) How can a stimulus process be made to repeat throughout simulation?
   - By not using the unconditional wait statement within the process block.

Slide 14-18:

Summary

- A VHDL testbench is used to verify functionality
- Testbenches cannot be synthesized
- Testbenches utilize the full range of the VHDL language
- VHDL models hardware by using concurrent operations
- The same testbench can be applied at each stage of verification

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

VHDL Assert Statements

Slide 15-1:

2019.1

This module describes the concept of VHDL assertions. 10 minutes.

Slide 15-2:

Objectives

After completing this module, you will be able to:

- Define a VHDL assertion and its syntax
- Write an *assert* statement

Slide 15-3:

Assertions

- The assertion statement is used to report mismatches between expected data and simulated data
- Assertions are non-synthesizable constructs
- Syntax
  - [<label>:] *assert* <condition> [*report* <expression>] [*severity* <expression>];
- Example

```
ackRcvd:  assert c = ACK
          report "Failed to Acknowledge"
          severity FAILURE
```


© Copyright 2019 Xilinx

Syntax:

- <label> - any unique, legal identifier.
- <condition> - any expression which evaluates to a boolean result. In the case of assertions, when the <condition> evaluates false, then report and severity (when present) are processed.
- <expression> - any legal expression that results in a string. Expression for the report portion can include string literals, function that return strings, or concatenation operators, for example. The expression for the severity must be one of the severity enumerated types to properly interface with the simulation tool. Legal severity expressions are defined in STD.STANDARD and can be one of the following four values: NOTE, WARNING, ERROR, or FAILURE.

Assertions can be used by some synthesis tools in synthesizable code to report certain types of configuration errors; however, they do not generate any logic.

In the absence of the report option, the message “Assertion violation” usually appears in the user console.

Example:

Assume that c is a character type that is received from some external device. If this assertion statement is placed where either an acknowledgment or a negative acknowledgment is expected, then the report and severity cases will be executed if c is NOT equal to the acknowledgment character. Note that ACK is a predefined character value from the STD.STANDARD package.

The message “Failed to Acknowledge” will display on the Tcl console of the Vivado® Design Suite (or equivalent display in other tools).

Severity FAILURE is interpreted by most simulation tools to halt simulation.

For synthesis:

VHDL assertions can be enabled by going to **Settings** > **Project Settings** > **Synthesis** and enabling it under the **Options** category in the Vivado Design Suite.

VHDL assertions can also be enabled with the following Tcl command:

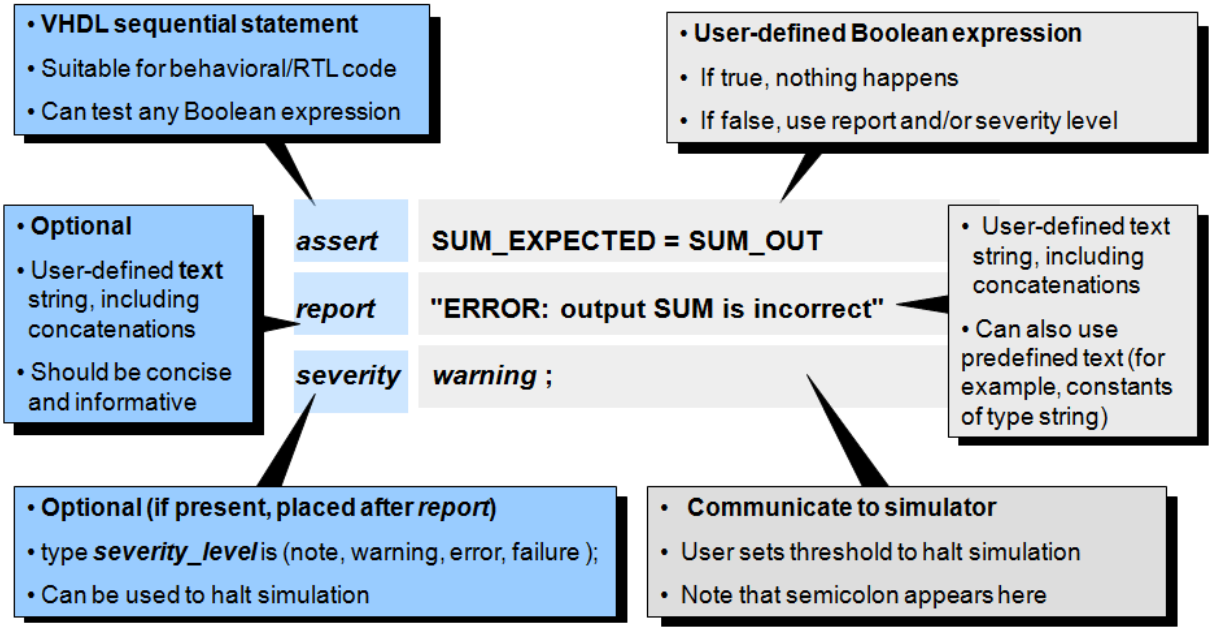
set_param synth.elaboration.rodinMoreOptions {rt::set_parameter ignoreVhdlAssertStmts false}

A severity level of failure will stop the synthesis flow and produce an error.

© Copyright 2019 Xilinx

Slide 15-4:

Assertion Syntax



Slide 15-5:

Other Uses of Assertions

- Quick way to always print a message without using TextIO

```
report "this message will always display in the console";
```

- The current simulation time is always available through the function "now"
  - Defined in STD.STANDARD
  - Returns the value of the current time of type time

- Stop the simulation at a designated time

```
assert now < endSimulationAt
report "simulation ending normally…"
severity failure;
```


© Copyright 2019 Xilinx

Slide 15-6:

Report Clause

- The *report* clause can be used by itself
  - No *assert* or *severity* required
  - Write strings directly to the simulation console

- Example

```
assert false
report "this message will always display in the console"
severity note;
```

  - Can be replaced by

```
report "this message will always display in the console";
```

Slide 15-7:

Uses of Severity

- Severity indicates the degree of the failed condition

- "Note" – Typically used to pass information messages to the user
  - Status of what is occurring during the testing procedure

- "Warning" – Often used when an unusual or unexpected situation occurs
  - Not serious enough to stop the simulation

- "Error" – Commonly used to indicate a known illegal or invalid condition
  - Also known as a recoverable error

- "Failure" – Used when an unrecoverable or never allowable situation has arisen

- If a severity clause is not present, the default is "error"

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 15-8:

Assertion Statements

- An *assert* statement is used to report information
- Assertion statements can be used to test conditions during simulation without creating additional logic
  - They can also be used to halt simulation
- Assertions are triggered by false conditions

```
TEST_FULL_ADD : process
  begin
  for i  in DATA_FRAME'range loop

  SUM_EXPECTED <= DATA_FRAME.EXP;
  SUM_OUT <= ALU_OUT ;
  . . .
 wait for 100 ns;

 assert  SUM_EXPECTED = SUM_OUT
 report "ERROR: output SUM is incorrect"
 severity warning ;

 end loop;
 wait;
end process ;
end architecture TEST ;
```

```
# ** Warning: ERROR: output SUM is incorrect
#    Time: 200 ns  Iteration: 0  Instance: /full_add_full_add_tb_vhd_tb
# ** Warning: ERROR: output C_Out is incorrect
#    Time: 200 ns  Iteration: 0  Instance: /full_add_full_add_tb_vhd_tb
```

An assertion is similar to an *if* statement. As long as *if* is used within a testbench, no additional logic is created.

The string VHDL data type is very helpful during simulation and debugging to display user-defined text, messages, warning, or errors. Note that the index range for string is declared as VHDL subtype positive and, therefore, starts with integer 1.

signal Warning1: string (1 to 30) := " Unexpected Outputs Detected " ; --declared within the architecture

variable Warning2: string (1 to 30) := " System Unstable, Aborting Now" ; --declared within the process

constant Warning3: string (1 to 21) := " Entering FSM State2" ; --declared within the package or architecture_

© Copyright 2019 Xilinx

That is:

```
process ( A_sig , B_sig, C_sig )
        begin
                if    (A_sig and B_sig ) /= '1' then
                        report Warning1;
                elsif ( A_sig and C_sig ) = '1'    then
                        report Warning2 & " Problem Mod2";
        end if ;
end process ;
```

Slide 15-9:

Other Validation Techniques

- Assertions are powerful statements
  - Can display messages to the user console
  - Can possess limited control over the simulation tool

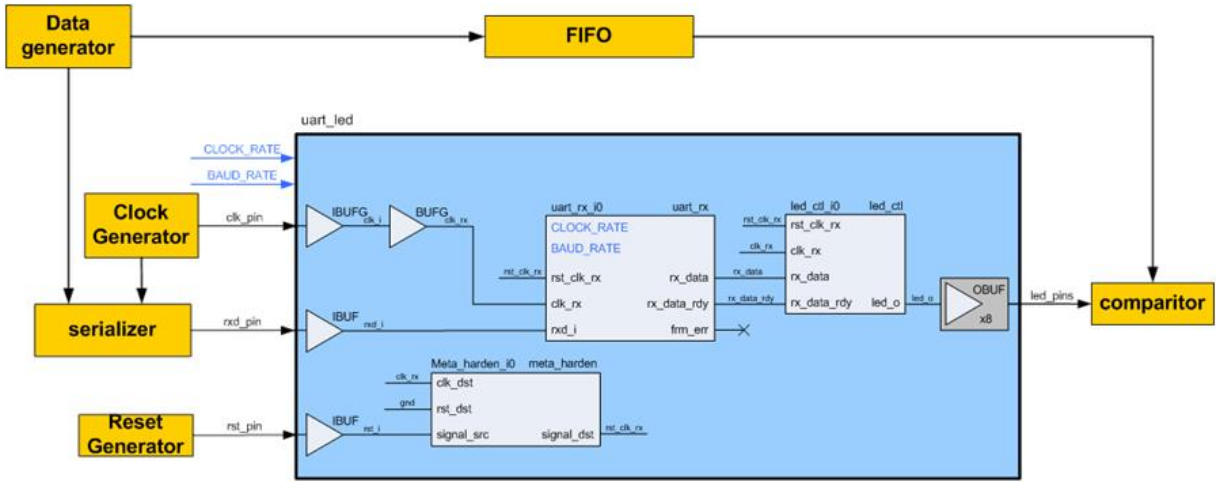
- There are, however, more powerful techniques for validating responses
  - A testbench is not limited to just the testbench and the UUT; it can also contain other modules that can
    - Model devices internal to the FPGA, such as processors and I/O
    - Model devices external to the FPGA, such as memory and A/D/A converters
    - Validate device performance
  - Remember that there is no requirement that *any* module evaluated in the simulation environment can be synthesized
    - Although the UUT, which usually represents an FPGA is synthesizable

Other higher level verification techniques are becoming popular, including Universal Verification Methodology (UVM), SystemVerilog, and SystemC.

© Copyright 2019 Xilinx

Slide 15-10:

Example of a Self-Checking Testbench



The orange boxes represent modules within the testbench that are not to be synthesized. Only the blue box represents the contents of the FPGA.

Each orange box represents a separate process or module.

The UUT is a simple circuit that receives a character via the RS-232 connection and latches it upon successful reception to drive eight LEDs.The test data is programmed into the data generator, which issues a series of characters to the serializer and the FIFO. This data is 8-bit parallel.The serializer converts this 8-bit data into a data stream whose timing and protocol is RS-232 compliant. This data stream is pumped into the UUT. The FIFO receives the data from the data generator and buffers it in time until the UUT produces the received data signal.

When the UUT has completely processed the data, it presents it in a parallel form to the LEDs, in this case connected to a comparator. The comparator checks the data from the FIFO against the data from the UUT and reports success or failure.

In this fashion, a test set can be applied to the UUT and automatically verifies with a minimum amount of coding. This type of testbench can be applied to any level of simulation (behavioral, netlist, post-place and route). The data set only needs to be changed in one place, making it quick and reasonably error free to try different test sets. Best of all, the results are text based and no manual waveform tracing is required.

These text-based testbenches are called self-checking testbenches. It checks the expected results against actual results at run time, rather that at the end of simulation. It is very useful to track errors when the design fails and shortens debug time.

© Copyright 2019 Xilinx

Slide 15-11:

Summary

- An assertion statement is used to report mismatches between expected data and simulated data
- Assertions are triggered by false conditions

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Attributes

Slide 16-1:

2019.1

This module describes attributes, the primary mechanism for extracting information about objects and injecting direction to downstream tools. Both predefined and user-defined attributes are discussed. 15 minutes.

Slide 16-2:

Objectives

After completing this module, you will be able to:

- Enumerate the three classes of "tic" attributes
- Use the signal attributes in the code
- Demonstrate how to locate and employ synthesis and implementation attributes

© Copyright 2019 Xilinx

Slide 16-3:

Attributes

- Attributes fall into two categories
  - Predefined or "tic" attributes which access information relevant to a specific type
  - Attribute specifications which "decorate" a named item
- Attributes are a convenient way of extracting information about an item
- Attributes make code reuse much easier
- Tic attributes may be thought of a functions with no argument that return a quality of the item it is attached to
- Many tic attributes are predefined for the specific of type, array object, or signal
- Decoration attributes have meanings assigned outside VHDL
  - User defined
  - But VHDL provides this as a means of communicating constraints to the tools

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 16-4:

Using "Tic" (Predefined) Attributes

- Generally used to extract information about an item

- Can be applied to
  - Types (T) – provides information on the specified type
  - Arrays (A) – provides basic functionality on the array
  - Signals (S) – provides information and basic functionality for signal types
  - Value or Entity (E) – provides information regarding the entity

- Syntax
  - <item>’<attribute name>

- Example
  - slv_arriving_data’length – returns the number of bits in the slv

Syntax:

- <item> - any legal and previously defined type, array, or signal name.
- <attribute name> - the name of a predefined attribute.

The focus will be on the first three types. The ‘E’ class of attributes is covered in the *Advanced VHDL* course.

© Copyright 2019 Xilinx

Slide 16-5:

Range Attributes

- VHDL provides attributes for composite data types. The following range attributes make subprograms more versatile and portable

```
signal  H_BYTE : std_logic_vector ( 7 downto 0 ) ;
signal  WORD : std_logic_vector ( 0 to 3 ) ;
```

range ‘left	H_BYTE ‘left returns 7,
range ‘right	H_BYTE ‘right returns 0,
range ‘low	H_BYTE ‘low returns 0,
range ‘high	H_BYTE ‘range returns 7 downto 0,
range ‘range	WORD ‘left returns 0,
range ‘length	WORD ‘low returns 0,
range ‘ascending	WORD ‘length returns 4,
range ‘reverse_range	WORD ‘ascending returns true,
. . .	H-BYTE ‘reverse_range returns 0 to 7,

Slide 16-6:

Commonly Used Attributes

- `left/`right – Returns the left-most or right-most value in the range
- `high/`low – Returns the largest or smallest value in the range (regardless of order)
- `image/`value – Produces a string of the value of the item / converts from string to type (limited capabilities)
- `pos/`val – Position within a list of the specified type the value at the specified position
- `succ/`pred – The next or previous value in the list
- `rightof/`leftof – The item to the right or left of the specified item within the list
- `range/`reverse_range – The range or reversed range of an array

© Copyright 2019 Xilinx

The IEEE Standard VHDL 1076 contains a full list of predefined attributes in section 14.1.

Slide 16-7:

Examples of Some Predefined Attributes

- Given

```
type coin is (penny, nickel, dime, quarter, half_dollar, dollar);
subtype coin_reversed is coin range dollar downto penny;
```

- coin'pos(dime) is 2
- coin'val(3) is quarter
- coin'rightof(quarter), coin'succ(quarter) is half_dollar
- coin'leftof(quarter), coin'pred(quarter) is dime
- coin_reverse'rightof(quarter) is dime, coin_reverse'succ(quarter) is dime
- coin_reverse'leftof(quarter) is half_dollar, coin_reverse'pred(quarter) is half_dollar

© Copyright 2019 Xilinx

Slide 16-8:

Typical Uses of Attributes

- When determining the size of a vector

```
variable vectorSize : integer := someVector'length;
```

- When looping through each member of a vector

```
touchEachPosition: for i in someVector'range loop
```

- When checking if one vector can fit inside another

```
if ((smallerVector'low >=largerVector'low) and
    (smallerVector'high <=largerVector'high)) then …
```

Slide 16-9:

Revisiting Multiple Dimension Arrays with Attributes

```
mmult: process
    type  tMATRIX_3x3x3  is array(1 to 3, 1 to 3, 1 to 3) of integer range -1024 to 1023;
    --variable matrix3D : tMATRIX_2x3x4 := (others=>(others=>(others=>0)));
    variable matrix3D : tMATRIX_2x3x4 := ((( 1, 2, 3, 4),( 5, 6, 7, 8),( 9,10,11,12)),
                                          ((13,14,15,16),(17,18,19,20),(21,22,23,24)));
    constant k : integer := 7;
  begin
    -- sparsely populate the matrix
    matrix3D(1,2,3) :=   4; matrix3D(1,3,1) := -10;
    matrix3D(2,2,2) := 100; matrix3D(1,2,3) := -50;

    -- loop through the matrix and multiply each element by a scalar
    rowLoop: for row in tMATRIX_3x3x3'range(1) loop
      colLoop: for col in tMATRIX_3x3x3'range(2) loop
        dthLoop: for depth in tMATRIX_3x3x3'range(3) loop
          report integer'image(row) & "," & integer'image(col) & "," &
                 integer'image(depth) & " * " & integer'image(k) & " = ";
           matrix3D(row,col,depth) := k * matrix3D(row,col,depth);
          report "   " & integer'image(matrix3D(row,col,depth));
        end loop dthLoop;
      end loop colLoop;
    end loop rowLoop;
    wait;
  end process mmult;
```

This example is identical to the example shown in the Composite Data Types" module, except that these loops automatically adjust for any change in the matrix size.

© Copyright 2019 Xilinx

Slide 16-10:

Signal Attributes (1)

- Signal attributes help in extracting more information about a signal
- Two main signal attributes
  - `event
    - Returns true if there is an event on the signal or else false
    - Example

```
if (clk`event and clk = '1') then
        op <= d;
```

  - `active
    - Returns true if there is a transaction on the signal or else false
    - Example

```
assert (not data`active)
report "Data Assignment has taken place"
```

Slide 16-11:

Signal Attributes (2)

- Signal attributes
  - `last_event
    - Returns the time interval since the last event on the signal
  - `last_active
    - Returns the time interval since the last transaction on the signal

- Check the setup time of your circuit

```
process (clk)
   variable c : time := 0 ns;
   begin
      if ( clk'event and clk = '1' ) then
         C:=S'last_event;
                        assert(C >= 3 ns)
                        report "Setup Violation"
                        severity warning;
                  end if;
   end process;
```


© Copyright 2019 Xilinx

Slide 16-12:

Signal Attributes (3)

- Other signal attributes
  - `delayed (T)
    - Returns a signal delayed by specified time T
  - `stable (T)
    - Returns true if there has been no event on the signal for time T
  - `quiet (T)
    - Returns true if there has been no transaction on the signal for time T

Slide 16-13:

Common Subprogram Usage

- Detecting the rising edge of a clock is a two-step process
  - First detect that an event on the clock has occurred, then
  - Check if the new value of clock is now high, which implies a rising edge
- The VHDL function rising_edge located in std_logic_1164 provides a convenient "shorthand" for this commonly used operation
  - Note the explicit signal designation for the formal parameter

```
package  std_logic_1164   is
      . . .
  function  rising_edge ( signal CLK : std_logic ) return  boolean;
    begin
      if ( CLK'event and CLK= '1' and CLK'last_ value = '0' )  then
        return  true ;
      else
        return  false ;
      end if ;
end  rising edge ;
. . .
```


© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 16-14:

User-Defined Attributes

- Used to pass information from the source code to various tools
- Can be applied broadly across a design, limited to a module, or target a single signal
- Two-step use
  - First, as always, the attribute must be defined: *attribute* <item name> : <type mark>;
  - Next, the quality can be applied to the attribute: *attribute* <item name> *of* <target name> : <entity class> *is* <expression>
- XST and Vivado® synthesis only support a <type mark> of type *string*
- Can be defined in the entity or architecture
  - If defined in the entity, then the attribute can be used within the entity and the architecture

Syntax:

- <item name> - any legal identifier. This must be the first use of the identifier otherwise there will be a conflict as to how that identifier is being used.
- <type mark> - any legal, predefined type. Most commonly, this is a string.
- <target name> - any legal, predefined identifier. This is the thing that the specified attribute will be applied to.
- <entity class> - any name of a type of VHDL structure. Most commonly the entity class is “signal”, or “entity”. The current set of choices is: entity, architecture, configuration, package, procedure, function, type, subtype, constant, signal, variable, file, component, label, literal, units, group, property, or sequence.
- <expression> - any statically evaluate-able equation or literal that is of type <type mark>.

© Copyright 2019 Xilinx

Slide 16-15:

Commonly Used User (Vendor) Attributes

- Attributes that guide the tools on an entity level
  - Predefined attribute RAM_STYLE : string;
  - Attribute RAM_STYLE of <signal>: signal is "block";

- Attributes that guide the tools on a signal level
  - Keep – Used to preserve a signal through synthesis so that it can be seen using tools such as the Vivado logic analyzer
    - Attribute keep : string;
    - Attribute keep of some_signal : signal is "true";

KEEP attributes

For example, if a signal is an output of a 2-bit AND gate, and it drives another AND gate, the KEEP attribute can be used to prevent that signal from being merged into a larger LUT that encompasses both AND gates.

KEEP is also commonly used in conjunction with timing constraints. If there is a timing constraint on a signal that would normally be optimized, KEEP prevents that and allows the correct timing rules to be used.

© Copyright 2019 Xilinx

Slide 16-16:

Apply Your Knowledge

4) Using either the Notes section or the tools, rewrite this procedure to handle data of any length by using the VHDL range attributes

```
procedure  BIT_REV_32  ( IN_VEC  : in  std_logic_vector ;
                         OUT_VEC : out std_logic_vector ) is
begin

   for  i in 31 downto 0 loop

     OUT_VEC(i) :=  IN_VEC(31-i);

   end loop;

end  procedure  BIT_REV_32 ;
```

Workspace

© Copyright 2019 Xilinx

Slide 16-17:

Answer

5) Using either the Notes section or the tools, rewrite this procedure to handle data of any length by using the VHDL range attributes

```
procedure  BIT_REVERSE  ( IN_VEC  : in  std_logic_vector ;
                          OUT_VEC : out std_logic_vector ) is
  begin

    for i in IN_VEC'range loop

      OUT_VEC (i) :=  IN_VEC ((IN_VEC'length-1) - i);

    end loop;

  end  procedure  BIT_REVERSE ;
```

Note that you went one step further in renaming the procedure. To enhance source-code readability—and to compensate for the inevitable reduction in readability that results from extensive subprogram usage—the subprogram label should be as descriptive as possible. Now that your procedure can handle data of any length, the label was changed to reflect the broader capability.

Note also that the additional set of parentheses in the expression enhances readability, but is not required.

Final point: If you were to build a simple entity/architecture to test this in simulation, explicitly declaring OUT_VEC as a signal would allow you to call the procedure concurrently without placing it within a process because there is no requirement to declare a variable.

Slide 16-18:

Summary

- VHDL provides a collection of predefined attributes
  - Provides information about a signal or other object
  - Commonly used to "generalize" code for wider application and code reuse
- VHDL also provides a mechanism for user-defined attributes
  - Typically used to pass information from the source code to the synthesis and implementation tools

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Subprograms

Slide 17-1:

2019.1

This module covers the use of subprograms in verification and RTL code to model functional blocks. 10 minutes.

Slide 17-2:

Objectives

After completing this module, you will be able to:

- Describe the use of subprograms in VHDL coding
- Differentiate between the functions and procedures

© Copyright 2019 Xilinx

Slide 17-3:

Software versus Hardware

- Functions are used in software and hardware to reduce the overall size of the code, gain granularity (hierarchy), and reuse code

- Embedded programmers are taught to use functions or procedures sparingly because there is run-time overhead
  - Calling a function requires space on the stack which runs the risk of overflowing
  - There is a time overhead in pushing data onto the stack and transferring control to the function. Additional time is consumed when information is popped from the stack and another jump must be executed

- Embedded programmers opt for functions as they reduce the amount of code that must be written and the overall code "footprint" (when memory is limited)

- Functions in hardware have no "run time" per se
  - Functions are expanded into hardware during synthesis
  - Functions take up silicon space, not time

Strangely enough, hardware and software functions are opposites. Where software costs time to save space, hardware costs space to save time.

Software can save time by allowing the use of macros or copying of blocks of code into a module. Hardware can save time by allowing the reuse of fabric and time slicing when pushing data into this region.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 17-4:

What are Subprograms?

- Coding a module may sometimes call for encapsulating functionality within the module
  - This functionality may be reused several time within the module
  - Encapsulating done for clarity
- In both behavioral and RTL-level coding, such functionality can be defined in a subprogram
- The subprogram can be called as necessary throughout the design structure
- Subprograms can be called both outside and inside a process
- VHDL offers two types of subprograms — functions and procedures

Another approach to creating functionality is to put it in a subprogram.

Subprograms are not global to a design, they are local to a module. They can be defined in multiple modules using an include file.

They are primarily a way of encapsulating functionality in an easy to understand place so that they can be called a number of times in a given module.

© Copyright 2019 Xilinx

Slide 17-5:

Using Subprograms

- Most large designs use common functional blocks repeatedly
- In both behavioral and RTL VHDL code, commonly used functions can be defined within a subprogram
  - The subprogram can then be called as necessary throughout the design
  - Provides flexibility and modularity to the code
  - Reduces the volume of coding necessary in a large design
- VHDL offers two types of subprograms: functions and procedures
- These functions and procedures are often stored in packages for use in other designs

Slide 17-6:

Functions versus Procedures

- Functions and procedures differ in their usage and capabilities
- Functions are used as an operator in an expression
  - Their parameters (arguments) can only be inputs (not outputs or inouts)
  - They must return a single result
  - Functions may be called as traditional functions with an arguments list
    - X <= f(a,b,c)
  - Or as operators, such as overloading a '+' sign to add two new types together
- Procedures can be used as a sequential or concurrent statement
  - Their arguments can be inputs, outputs, or inouts
  - They contain sequential statements that produce an effect

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Functions are more limited than procedures.

Functions can only be used as an operator inside of an expression; procedures can stand alone as a complete statement.

Function arguments must be of type input (or internal signals).

Functions must return a single result; procedures can affect several signals in the design.

Slide 17-7:

Apply Your Knowledge

1) True or false: VHDL subprograms can be used in both RTL and simulation environments

2) What are the two types of subprograms available in VHDL?

1) True or false: VHDL subprograms can be used in both RTL and simulation environments.

   True. Subprograms can be quite useful in either environment.

2) What are the two types of subprograms available in VHDL?
   - Functions
   - Procedures

Slide 17-8:

Summary

- Subprograms can be used to enhance code flexibility and modularity
- Two types of VHDL subprograms
  - Functions
  - Procedures

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Functions

Slide 18-1:

2019.1

Functions form an integral part of both reusable code and highly maintainable code. In this module, these common constructs are compared against their software counterparts. The concept of overloading is also covered with several examples. 15 minutes.

Slide 18-2:

Objectives

After completing this module, you will be able to:

- Write functions in your code
- Use function overloading in the code

Slide 18-3:

Function Overview

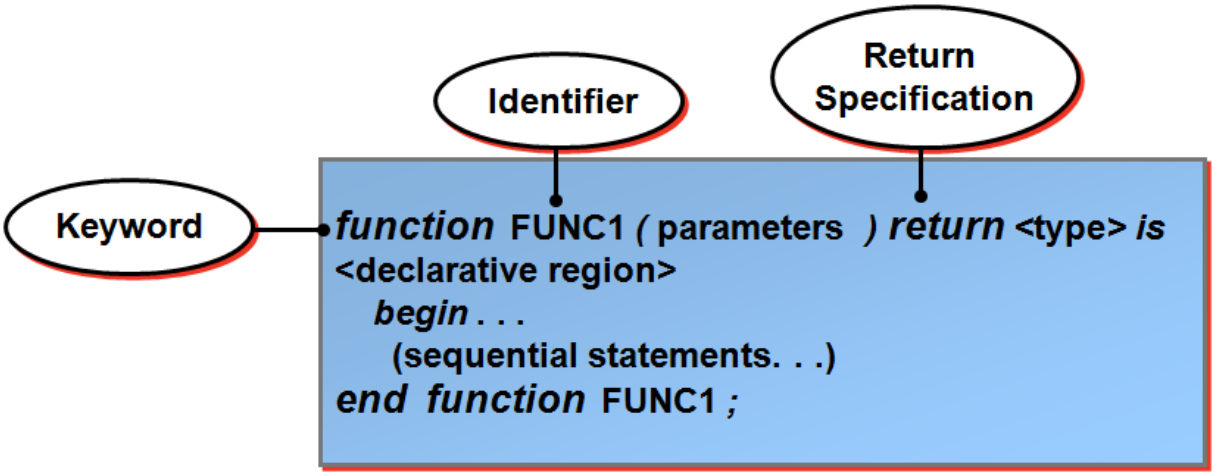
- A function is a type of subprogram
- Avoids code repetition as the function can be called multiple times
- A function only supports input ports; it does not support output or inout ports
  - Should contain at least on input argument
- A function returns only a single value
- A function does not support timing delays
  - Executes in zero simulation time

© Copyright 2019 Xilinx

Slide 18-4:

Function Structure

- Function parameters are inputs only and the return type must be specified
- The function is an operand in a given expression



Slide 18-5:

Input Parameter List

- Input list contains any number of parameters including zero
- Parameter if not defined considered as IN by default
- Input list objects can be signal and constant
  - Constant is the default type

```
function arith_logic_fun ( signal a, b : in std_logic;
                                  c, d : integer := 5;
                           constant  d : integer;
                                  sel : bit ) return integer is
```


© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 18-6:

Passing Parameters

- Rules that should be followed while passing parameters
  - Can be any number of parameters
  - Can only have inputs
  - Signals can only be connected to signals
  - Constants can be connected to any other objects
  - Variables are not allowed

Slide 18-7:

Function Example

- This function is declared within the architecture and is used to calculate the parity for a given bit_vector

```
architecture RTL of Mod1 is
. . .
function PAR_GEN ( BV: in bit_vector ) return bit is
variable PAR : bit := '0';
 begin - - function
  for I in BV'range loop
  PAR := PAR xor BV (I) ;
 end loop ;
return PAR ;
end function PAR_GEN ;
begin - - architecture
 DATA_FRAME <= D_WORD & PAR_GEN ( D_WORD ) ;
. . .
end architecture RTL;
```

Concurrent Call

Note that when a subprogram is declared within the architecture, or process, the declaration and body remain together (as compared to separation that occurs within the package and package body).

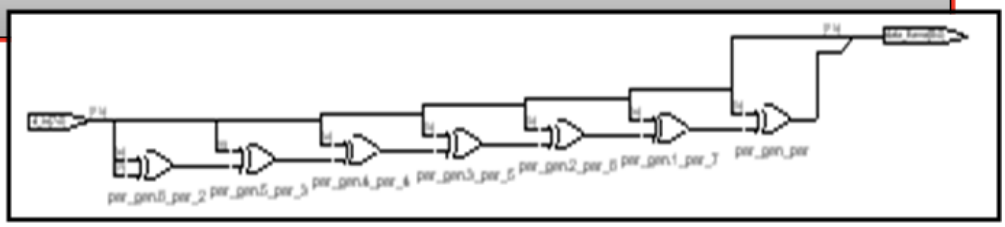
© Copyright 2019 Xilinx

Slide 18-8:

Logic Created

- This previous function described a recursive *XOR* operation
- Each time it is called, it will produce an *XOR* tree whose depth is dependent on the length of the array

```
begin   - - architecture

 DATA_FRAME <=  D_WORD & PAR_GEN ( D_WORD ) ;
. . .
end  architecture RTL;
```



Slide 18-9:

Math and Conversion Functions

- VHDL does not predefine arithmetic operations for array structures (for example, bit_vector, std_logic_vector)
- VHDL tools include packages that contain arithmetic functions to facilitate these operations for certain data types such as UNSIGNED and SIGNED
- The VHDL template provided by the Xilinx tools includes the NUMERIC_STD package

```
package NUMERIC_STD is
  constant CopyRightNotice: STRING
      := "Copyright 1995 IEEE. All rights reserved.";

  --=============================================================
  -- Numeric array type definitions
  --=============================================================

  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;

  --=============================================================
  -- Arithmetic Operators:
  --=============================================================

  -- Id: A.1
  function "abs" (ARG: SIGNED) return SIGNED;
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0).
  -- Result: Returns the absolute value of a SIGNED vector ARG.

  -- Id: A.2
  function "-" (ARG: SIGNED) return SIGNED;
  -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0).
  -- Result: Returns the value of the unary minus operation on a
  --         SIGNED vector ARG.

  --=============================================================

  -- Id: A.3
  function "+" (L, R: UNSIGNED) return UNSIGNED;
  -- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two UNSIGNED vectors that may be of different lengths.
```

Making Package Visible

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;
```

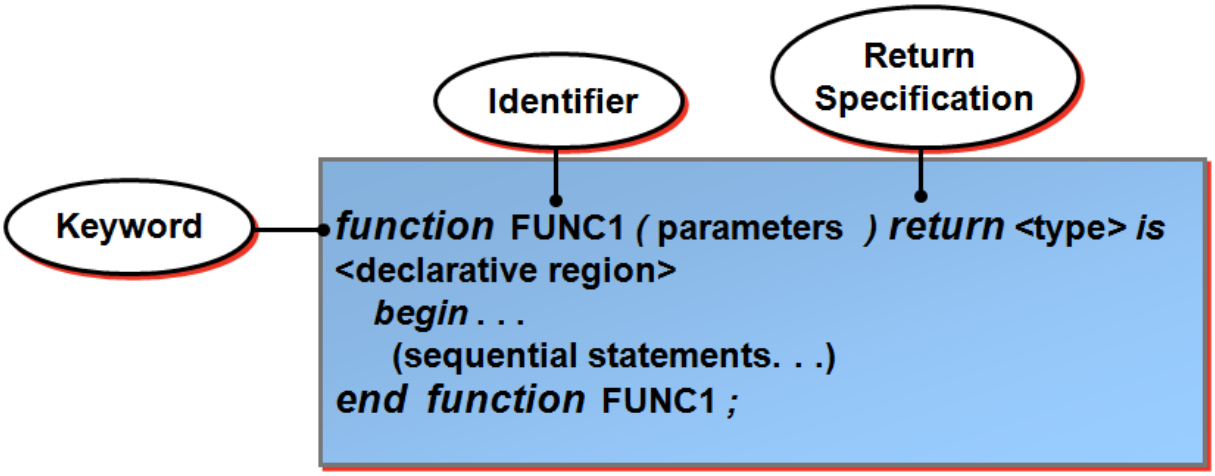

© Copyright 2019 Xilinx

You will find this IEEE packages under the Xilinx Vivado installed directory *C:\Xilinx\Vivado\2019.1\data\vhdl\src\ieee\distributable*.

Slide 18-10:

Pure and Impure Function

- Pure function returns the same values each time with the same set of inputs
  - If nothing is specified, function is considered as pure by default
- Impure function returns different values with the same set of inputs
- Example of impure function



© Copyright 2019 Xilinx

Slide 18-11:

Function Overloading

- Because most intended operations are similar, VHDL allows the same call name to be used for multiple functions and procedures

- To differentiate between subprograms, the type and/or number of formal parameters must be different

```
package array_arith is
  function "+" (L, R: UNSIGNED) return UNSIGNED;
  function "+" (L, R: SIGNED) return SIGNED;
  function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
  ...
  ...
  ...
```

- This technique is referred to as overloading

Both the terms *operator overloading* and *subprogram overloading* are used. The latter is obviously a broader description.

© Copyright 2019 Xilinx

Slide 18-12:

Using an Overloaded Operator

- If the functions are present and visible within the design unit, the compiler will automatically pass the parameters and return the result
  - Otherwise, it will indicate that the operation is undefined

```
architecture RTL of Mod_1 is
. . .
signal A_INT : integer range natural ;
signal B_BUS, C_BUS, D_BUS : bit_vector ( 3 downto 0 );
. . .
begin
B_BUS <= C_BUS + D_BUS ;
A_INT <= B_BUS + C_BUS ;
B_BUS <= C_BUS + A_INT ;
. . .
```

"+" Operator Calls Function

Slide 18-13:

Building an Overloading Function

- Assume that you have an enumerate type, such as

```
type color is (BLACK, BROWN, RED, ORANGE, YELLOW,
               GREEN, BLUE, VIOLET, GRAY, WHITE);
```

- And you want to be able to add two enumerated types together
  - colorValue := BLACK + WHITE;
  - colorValue := RED + YELLOW;

- How can this be done?

© Copyright 2019 Xilinx

Slide 18-14:

Example of Building an Overloaded Function

- Create an overload function for type color
  - Accepts two colors as "arguments", returns one color as a result
  - Function must determine what the proper return value is

```
function '+' is (x : in color; y : in color) return color is
   begin
      arg1color: case (x) is
         when BLACK =>
            blackAndArg2: case (y) is
               when BLACK => return BLACK;          -- black + black = black
               . . .
               when WHITE => return GRAY;           -- black + white = gray
            end case blackAndArg2;
         when BROWN =>
            . . .
         when RED =>
            redAndArg2: case (y) is
               . . .
               when YELLOW => return ORANGE;        -- red + yellow = orange
               . . .
            end case redAndArg2;
         . . .
         when WHITE =>
            . . .
      end case arg1color;
   end function '+';
```

You will realize that there must be some logical limit to how many colors can be reasonably supported.

For example, if red and yellow is orange, what is orange and yellow? The answer must be one of the enumerated types. In order for this to be complete, there will be a nearly infinite number of solutions.

An easier approach to this is to simply limit the number of types to primary and secondary (and if the designer is ambitious, tertiary) colors. If two tertiary colors are added together, then the first argument could be returned thus limiting further levels of color mixing.

Of course, converting colors to RGB or other numeric equivalent would also be an easy mechanism to perform the same type of capability.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 18-15:

Apply Your Knowledge

1) Group exercise: Write a function to convert the following std_logic_vector to an integer

0	1	1	0	1	0	0	1

- Hints
  - 1. For loop
  - 2. If statement
  - 3. Exponential operator (**)
  - 4. Addition operator (+)

- If written correctly, the function should return a result of 105

- Create a testbench that proves the subprogram is written correctly

Workspace

© Copyright 2019 Xilinx

Slide 18-16:

Answer

1) Write a function to convert the following std_logic_vector to an integer

0	1	1	0	1	0	0	1

```
function VEC2INT ( VEC : in std_logic_vector) return integer is
   variable RESULT : integer := 0 ;
begin
 -- assume bit 0 is LSB
 for i in VEC'range loop
    if VEC(i) = '1' then
     RESULT := RESULT + (2**i) ;
    end if ;
  end loop ;
 return RESULT ;
end VEC2INT;
```

```
entity VEC2INT_TB is
end  VEC2INT_TB ;

architecture TEST of VECT2INT_TB is
 function VEC2INT . . .- - declare subprogram
 signal  IN_VEC : std_logic_vector ( 7 downto 0) := "01101001" ;
 signal  OUTPUT : integer range 0 to 255 ;
. . .
begin
  OUTPUT <= VEC2INT ( IN_VEC ) ;   --call subprogram
end architecture TEST ;
```

Slide 18-17:

Summary

- Functions form an integral part of reusable code
- Common functions include arithmetic and conversion operations
- Overloading allows the same operator name to reference multiple functions

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Procedures

Slide 19-1:

2019.1

Procedures form an integral part of both reusable code and highly maintainable code. In this module, these common constructs are compared against their software counterparts. 15 minutes.

Slide 19-2:

Objectives

After completing this module, you will be able to:

- Explain procedures and their syntax
- Write procedures in your code

Slide 19-3:

Procedure Overview

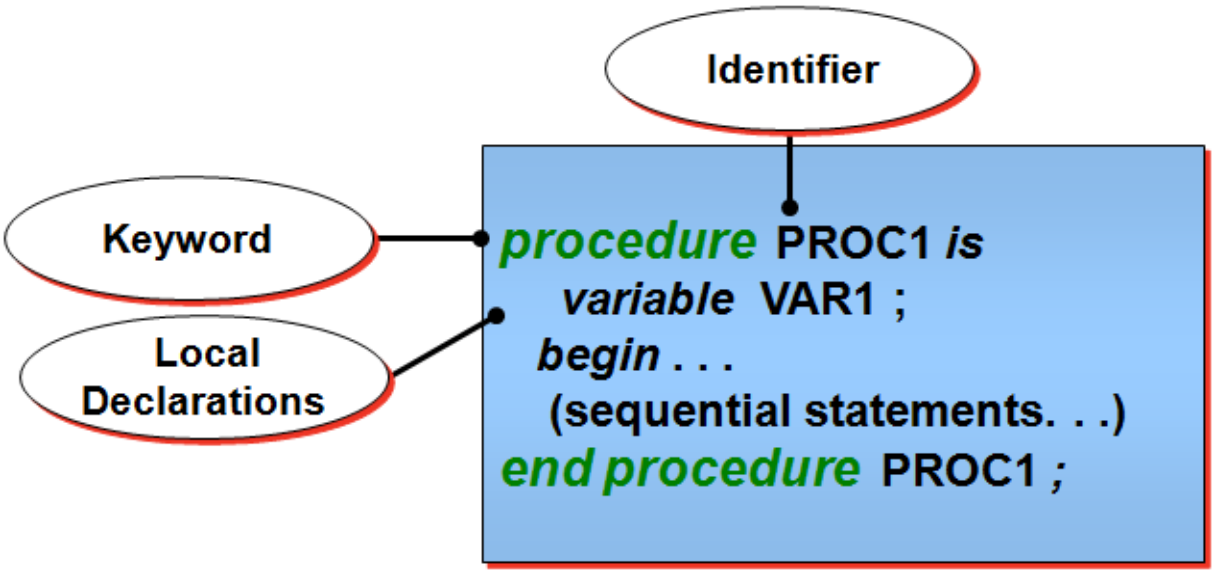
- A procedure is a type of subprogram
- Avoids code repetition by calling the procedure again and again
- Procedures can be declared with or without arguments
  - Procedures can have input, output, and inout ports
- Procedures may or may not include timing delays
  - Procedures execute in non-zero simulation time

© Copyright 2019 Xilinx

Slide 19-4:

Procedure Structure

- A procedure can be declared with or without arguments



- If arguments are listed, they allow greater utilization throughout the design

- A procedure that has no parameters can be called simply by using its identifier, such as PROC1 ;

Procedures without arguments are typically used to run sequences of events. For example, if a design runs three devices, a procedure called “MoveThing” might initiate a series of calls to other procedures.

This is most commonly done in testbenches where a procedure might write message to the console or drive specific signals.

If a function or procedure has no arguments, it interacts with the rest of the architecture it is instantiated in only by directly using signals in that architecture. VHDL scope rules give it visibility of those signals.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 19-5:

Procedure Parameters

- Procedure arguments are similar to port declarations—they can be inputs, outputs, or bidirectional
  - Mode *in* can be read, but not changed; treated as constant
  - Mode *out* cannot be read, only assigned to; copied back to caller
  - Mode *inout* can be read and assigned to; copied back to caller

```
procedure PROC2 ( IN1: in <type> ;
                  OUT1: out <type> ;
                  BI_DIR1: inout <type> ) is
variable VAR2 ;
  begin . . .
   (sequential statements. . .)
end procedure PROC2 ;
```

Formal Parameters

Slide 19-6:

Passing Parameters

- If a procedure has arguments, it is called using its identifier and the actual parameters, which then map to the formal parameters

- That is, PROC2 (D_IN, D_OUT, D_IO);

```
procedure PROC2 ( IN1: in <type> ;
                  OUT1: out <type> ;
                  BI_DIR1: inout <type> ) is
variable VAR2 ;
  begin . . .
   (sequential statements. . .)
end procedure PROC2 ;
```

- As in instantiations, this is not recommended
  - Use named associations

© Copyright 2019 Xilinx

Slide 19-7:

Associating Parameters

- As with the port map declaration, the call expression can explicitly bind the formal and actual parameters via named association
- That is, PROC2 (IN1 => D_IN, OUT1 => D_OUT, BI_DIR1 => D_IO);

Formal → Actual

```
procedure PROC2 ( IN1:  in  <type> ;
                  OUT1:  out  <type> ;
                  BI_DIR1:  inout <type> ) is
variable VAR2 ;
  begin . . .
   (sequential statements. . .)
end procedure PROC2 ;
```

Slide 19-8:

Parameter Classes

- In addition to mode and data type, the formal and actual parameters must also be of the same class
- VHDL has four classes of objects
  - Constant
  - Variable
  - Signal
  - File

```
procedure  PROC2 ( <pclass> IN1    :in    <type> ;
                   <pclass> OUT1   :out   <type> ;
                   <pclass> BI_DIR1:inout <type> ) is
     variable  VAR2 ;
   begin . . .
    (sequential statements. . .)
end procedure  PROC2 ;
```

- Procedure parameter classes can use default, explicit constant, variable, or signal declaration

© Copyright 2019 Xilinx

Slide 19-9:

Default Parameter Class

- A formal parameter of mode
  - *in* will default to *constant*
  - *out* will default to *variable*
  - *inout* will default to *variable*

```
procedure  PROC2 ( IN1     : in    <type> ;
                   OUT1    : out   <type> ;
                   BI_DIR1 : inout <type> ) is
      variable  VAR2 ;
   begin . . .
     (sequential statements. . .)
end procedure  PROC2 ;
```

Slide 19-10:

Default Parameter Class

- The most important consequence is that formal and actual must be consistent
  - Therefore, in the previous example, the actual object associated with OUT1 or BI_DIR must be declared as a variable

```
procedure  PROC2 ( IN1     : in    <type> ;
                   OUT1    : out   <type> ;
                   BI_DIR1 : inout <type> ) is
      variable  VAR2 ;
   begin . . .
     (sequential statements. . .)
end procedure  PROC2 ;
```

That is, PROC2 (IN1 → D_IN, OUT1 → D_OUT, BI_DIR1 → D_IO);

Must be Variable

© Copyright 2019 Xilinx

Slide 19-11:

Example Procedure

- The following procedure is used to reverse the order of elements for a std_logic_vector of unspecified length

```
procedure  SLV_REVERSE ( SLV_IN   : in   std_logic_vector ;
                         SLV_OUT  : out  std_logic_vector ) is
 begin

   for i in IN_VEC'length-1 downto 0 loop

      SLV_OUT(i) := SLV_IN (IN_VEC'length-i);

   end loop;

end  procedure SLV_REVERSE ;
```

What assumptions are made here? For one, it assumes that IN_VEC and OUT_VEC are of equal length. Do you see any others?

Could this be written another way, perhaps using the ‘range attribute?

Slide 19-12:

Explicit Input Parameter Class

- Actually specifying the class of input parameters declared in the subprogram is often helpful
  - Can simplify usage and expand the capabilities of the subprogram

```
procedure PROC2 ( signal IN1: in <type> ;
                  OUT1: out <type> ;
                  BI_DIR1: inout <type> ) is
variable VAR2 ;
  begin . . .
   (sequential statements. . .)
end procedure PROC2 ;
```

In this example, the keyword ***signal*** before the parameter IN1 declaration changes it from a constant to a signal class

  - The difference in use is that the actual signal is passed to the subprogram, rather than a “snapshot”—as in the case of the default constant
    - This allows VHDL signal attributes such as ‘event to be applied

© Copyright 2019 Xilinx

It is also important to point out that if an input is constant and the procedure has any wait statements in it, it will only see the initial value of the input parameter at the time the procedure was called.

Slide 19-13:

Defining Output Parameters

- Actually specifying the class of output parameters declared in the subprogram is often helpful
  - Can simplify usage and enhance the readability of the source code

```
procedure PROC2 ( IN1: in <type> ;
                  signal OUT1: out <type> ;
                  BI_DIR1: inout <type> ) is
variable VAR2 ;
  begin . . .
   (sequential statements. . .)
end procedure PROC2 ;
```

In this example, the keyword ***signal*** before the parameter OUT1 declaration changes it from a variable to a signal class

  - The difference in use is that you are not required to declare a variable within the process so that it can be passed into the subprogram and then assigned to an output signal—as in the previous example

© Copyright 2019 Xilinx

Slide 19-14:

Declaring a Procedure in an Architecture

- If a *procedure* or *function* is declared within the *architecture*, it is entered in its entirety between the architecture's *is* and *begin* statements

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

entity exampleProcedure is
   <port declaration>
end entity exampleProcedure;

architecture behavioral of exampleProcedure is

        procedure SLV_REVERSE (SLV_IN  : in  std_logic_vector;
                               SLV_OUT : out std_logic_vector) is
          begin
             for i in SLV_IN'length-1 downto 0  loop
               SLV_OUT(i) :=  SLV_IN(SLV_IN'length-1 - i);
             end loop;
         end  procedure SLV_REVERSE;
   begin

         <concurrent statements>

   end architecture exampleProcedure;
```

Procedure Body*

Note that the subprogram declaration is repeated in the package body. In general, this must match the declaration in the package.

Exceptions include the base specification for formal parameters—that is, octal versus hex or binary.

*There is no need for an explicit procedure or function definition as is required for packages.

Procedures and functions can also be declared in an entity statement following the port declaration.

© Copyright 2019 Xilinx

Slide 19-15:

Calling the Procedure

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

entity MOD1 is
   port ( IN_BUS : in std_logic_vector (31 downto 0);
                . . .
          OUT_BUS : out std_logic_vector (31 downto 0)
        );
end  MOD1 ;

architecture RTL of MOD1 is
    procedure SLV_REVERSE (SLV_IN : std_logic_vector,
                           SLV_OUT : std_logic_vector) is
    begin
       . . .
    end procedure SLV_REVERSE;
  begin
    process ( IN_BUS )
      variable TEMP_BUS : std_logic_vector (31 downto 0);
     begin
        SLV_REVERSE ( IN_BUS, TEMP_BUS )
        OUT_BUS <= TEMP_BUS;
  end process;
end architecture RTL;
```

Sequential Call

In this call, the actual parameters in the call are positionally associated to the formal parameters in the procedure and must be of the same class*

*Note the use of the variable TEMP_BUS. The variable is declared within the process, passed as an argument into the procedure, and then copied to the output signal.

Although VHDL supports positional association when using functions, procedures, and instantiations, it is NOT considered good coding practice. Named association is much preferred as it is explicit and less prone to coding errors.

The above “call” should read as: SLV_REVERSE(SLV_IN=>IN_BUS, SLV_OUT=>TEMP_BUS);

© Copyright 2019 Xilinx

Slide 19-16:

Call with Explicit Signal Class

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

entity MOD1 is
port ( IN_BUS : in std_logic_vector (31 downto 0);
       . . .
       OUT_BUS : out std_logic_vector (31 downto 0)
            );
end  MOD1 ;

architecture RTL of MOD1 is
begin
  process   ( IN_BUS )
    begin
            BIT_REV_32 ( IN_BUS, OUT_BUS );
  end process;
end architecture RTL;
```

```
package my_pack is
…

  procedure BIT_REV_32(signal SLV_IN : std_logic_vector,
                       signal SLV_OUT : std_logic_vector) is
    begin
        for i in SLV_IN'length-1 downto 0  loop
           OUT_VEC(i) :=  IN_VEC(SLV_IN'length-1 - i);
        end loop;
   end procedure BIT_REV_32;

end package body my_pack;
```

my_pack.vhd

Having declared the formal parameter OUT1 as a *signal*, you can pass the output port OUT_BUS (signal) directly

Sequential Call*

*Procedures can be called concurrently as well.

The code shows the modified example of the earlier source code.

Note that having both versions of this subprogram stored in the same package is possible. The concept of overloading means that the compiler would choose between the versions based on the exact nature of the parameters passed in. That is, the distinguishing factor would be whether a signal or variable is passed as an argument to the subprogram formal parameter OUT1.

Slide 19-17:

Summary

- A procedure can be declared with or without arguments
- Procedure parameter classes can use default, explicit constant, variable, or signal declaration
- A procedure is declared within the *architecture*, it is entered in its entirety between the architecture's *is* and *begin* statements

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

VHDL Libraries and Packages

Slide 20-1:

2019.1

This module demonstrates how libraries are declared and used. It also talks about the packages and how they can be built and reused. Commonly used packages from the STD and IEEE libraries are discussed. 20 minutes.

Slide 20-2:

Objectives

After completing this module, you will be able to:

- Enumerate the frequently used standard libraries and identify the relevant contents of each
- Explain how to create packages and libraries
- Describe the contents of a package and a library

© Copyright 2019 Xilinx

Slide 20-3:

Libraries

- Libraries are compiled collections of packages
- Several libraries always exist in a VHDL project
  - STD includes the STANDARD and TEXTIO packages
  - WORK includes all the compiled elements for a design*
- Additional libraries can be imported into the workspace
  - IEEE contains the STD_LOGIC_1164 and NUMERIC_STD packages
  - Identifying the IEEE library enables access to all provided IEEE packages
- You can create libraries
  - Managed through the Vivado® IDE
  - Implemented as sub-directories under the main working directory
  - You can import as many or as few packages into the libraries as you want

* For designs that are created using the Vivado Design Suite, "xil_defaultlib" is the default library for the VHDL files and includes all the compiled elements for a design.

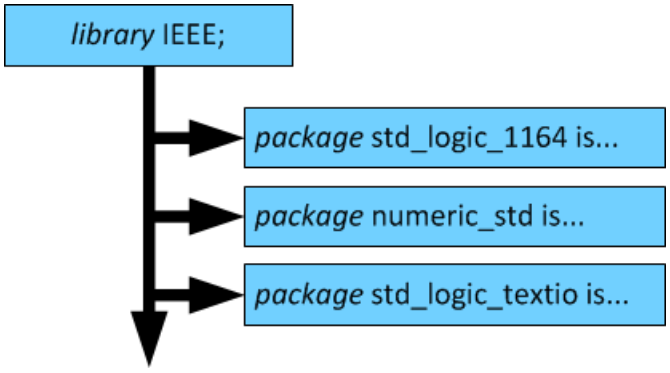
© Copyright 2019 Xilinx

Slide 20-4:

Library Structure

- All VHDL objects (design units) ultimately reside in libraries
  - By definition, a library is a collection of analyzed design units
  - The STD library is implicit and therefore available to every design unit

- Libraries and packages comprise the VHDL design management structure
  - Somewhat analogous to directories and subdirectories



Commonly used packages for RTL source code:

- std_logic_1164: Contains all type and operator specifications for data types std_logic and std_ulogic.
- std_logic_numeric: A set of arithmetic, conversion, and comparison functions for SIGNED, UNSIGNED, SMALL_INT, INTEGER, STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR.
- std_logic_textio: This package overloads the standard TEXTIO procedures, READ and WRITE. Use for reading and writing to external files during simulation.
- textio: This package overloads the standard TEXTIO procedures, READ and WRITE for types bit, boolean, integer, string, or time, for example.

© Copyright 2019 Xilinx

Slide 20-5:

xil_defaultlib Library

- This is the default subdirectory or physical location used to store all compiled design units unless otherwise indicated
- Does not require explicit declaration
- Each simulator or synthesis tool maintains such a structure
  - Note that design unit labels, not filenames, are visible in the library

Design Unit	Identifier
entity	**HALF_ADD**
entity	DFF
package	MY_CNTRS. . .
architecture	**RTL**
architecture	STRUCTURAL

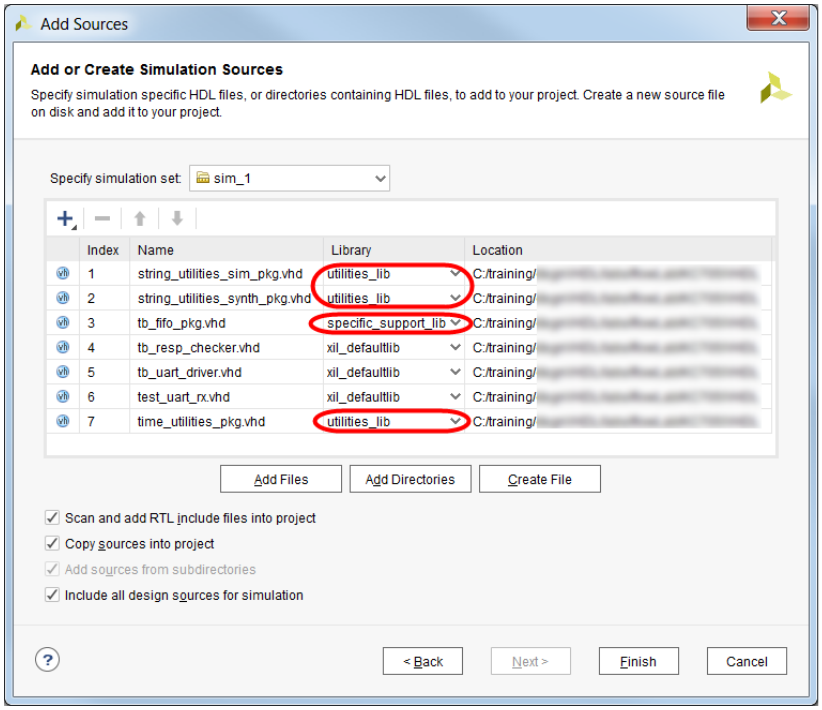
Example of xil_defaultlib library contents

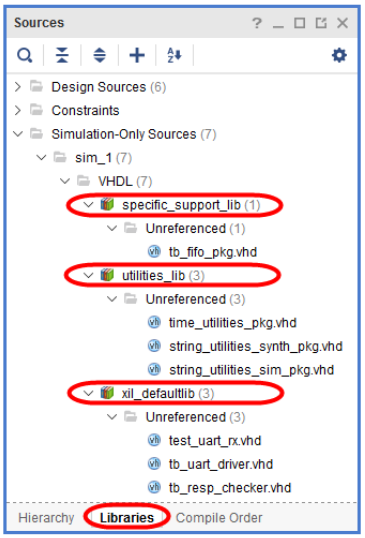
Secondary (dependent) design units are referenced to their primary counterpart

© Copyright 2019 Xilinx

Slide 20-6:

Creating a Library in the Vivado IDE





When a file is added to the project, you have the option of importing the file into the default xil_defaultlib library, a previously defined library, or a new library.

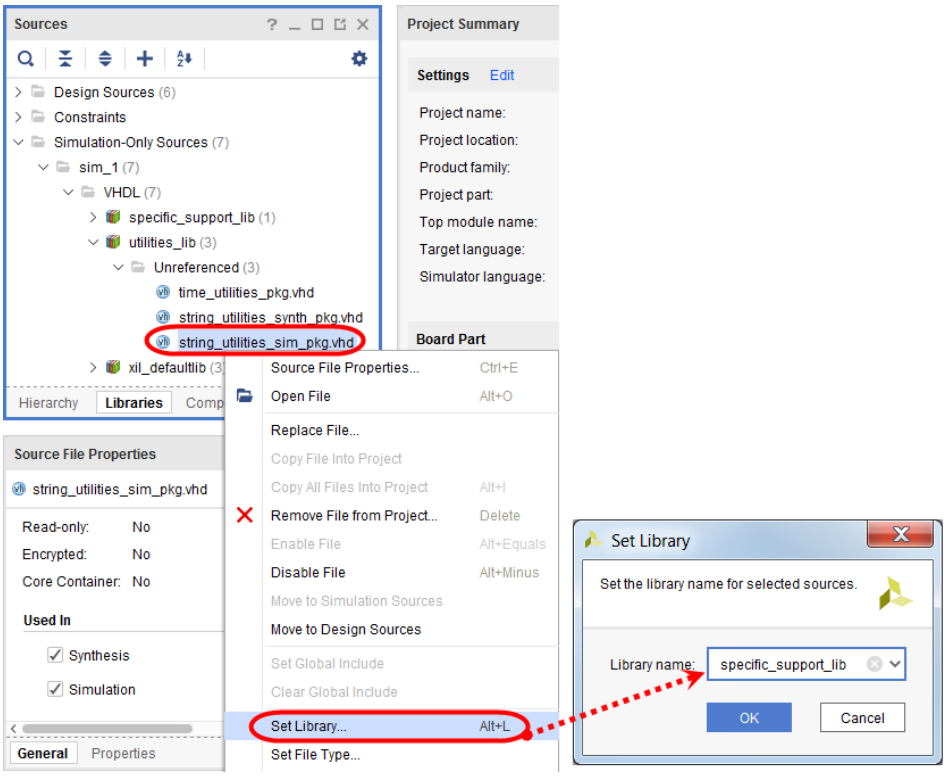
To create a new library, you can enter one or more library names in the library column in the Add or Create Design Sources dialog box. After the source is added, the libraries appear in the Libraries tab of the Sources view with the various packages within them shown hierarchically.

© Copyright 2019 Xilinx

Slide 20-7:

Developing Packages in Libraries

- Recall that a library is a collection of evaluated, or already compiled, packages
- If you are actively developing a package, you can elect to keep it as part of the working directory, and moved to the appropriate library when it is stable



To move a package to a selected Library, right-click the file and select **Set Library**. Choose a library to add the package file to.

www.xilinx.com
© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 20-8:

Accessing the Library and its Packages

- Declare the library in the module requiring access
  - Access is only possible in the module where the library is called out

- Specify which packages within the library are to be used
  - Elements within each package can be called out explicitly or
    - To overload specific functions or to avoid conflict with other functions from other packages
  - The entire contents can be accessed using the "all" designator

- Example

```
library string_utilities_synthesizable;
use string_utilities_synthesizable.conversionUtilities_pkg.all;
use string_utilities_synthesizable.alternateConversion_pkg.paddedHex;
```

Enumerating the library at the top module of the design does not make the library accessible to all levels of hierarchy below it. The library must be announced in each module that uses it.

library string_utilities_synthesizable;

- Makes the library visible to this module.

use string_utilities_synthesizable.conversionUtilities_pkg.all;

- Everything in the conversionUtilities package is now available to this module.

use string_utilities_synthesizable.alternateConversion_pkg.paddedHex;

- Only the paddedHex function in this package is available to this module.

© Copyright 2019 Xilinx

Slide 20-9:

Packages

- Packages are constructs that help collect and organize functions, procedures, types, subtypes, constants, and component declaration
- Like the entity/architecture primary/secondary pair, packages are comprised of the package and the package body
  - The package is a primary unit and can contain types, subtypes, constants, and function and procedure declarations
  - The package body is a secondary unit and contains the definition for the functions and procedures declared in the package
- The term "package" usually refers to both the package and package body
- Packages are available from IEEE, tool vendor, and users

Packages from Xilinx are contained in the Unisim and Simprim libraries.

Slide 20-10:

Syntax

- The syntax for the package is very similar to that of the entity and architecture

```
package <package name> is
   <package declarative items>
end [package] <package name>;
```

- The package body follows the same pattern and must share the same name as the package to be properly associated

```
package body <package name> is
   <package body declarative items>
end [package body] <package name>;
```


© Copyright 2019 Xilinx

Syntax:

- <package name> - any legal identifier. This name must not be the same as any other package name that is included in the same library that this package resides in. Identical package names may reside in other libraries.
- <package declarative items> - any combination of type, subtype, constant, function, or procedure declarations. Any function or procedure declarations must be defined in a package body. If no functions or procedures are present, then the package body is not required.

Slide 20-11:

When to Create a Package

- There is no upper limit on the number of packages a design can have
  - Requires intelligent organization of the contents of each package
- Typically, packages are created to support specific features such as
  - A simulation-only FIFO which contains all functions, procedures, and data structures
  - A collection of simulation-only text functions
    - Using the access type, more C-like functions and string behaviors can be created
  - A collection of synthesizable string manipulation functions
  - Other specialized functions such as custom physical types
- Many designers choose to keep all component declarations in one package
  - Reduces clutter in many files
  - Makes changing a component much easier

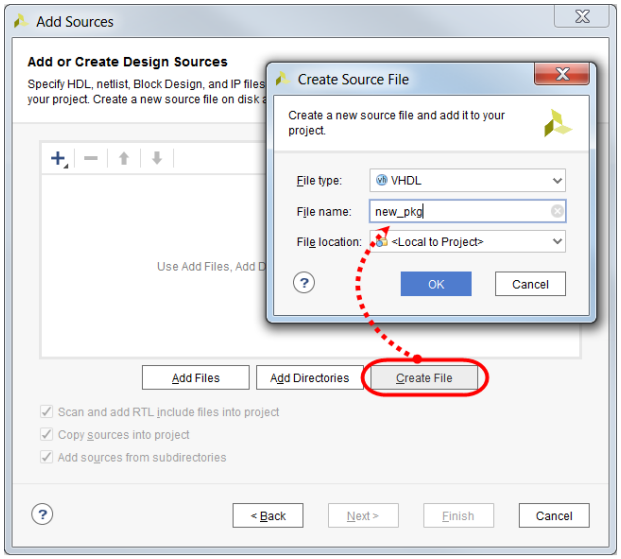
Recall that if any aspect of the component interface changes (anything within the entity statement), then all locations where that component is declared and instantiated must also be changed.

By keeping all component declarations in a single package, then the changed declaration only needs to be modified in one place, thus greatly reducing the amount of work in locating and modifying each declaration.

© Copyright 2019 Xilinx

Slide 20-12:

Creating a Package




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package <Package_Name> is
  type <new_type> is
    record
        <type_name> : std_logic_vector(7 downto 0);
        <type_name> : std_logic;
    end record;

-- Declare constants
  constant <constant_name> : time := <time_unit> ns;
  constant <constant_name> : integer := <value>;

-- Declare functions and procedure
  function <function_name> (signal <signal_name> : in
<type_declaration>) return <type_declaration>;

  procedure <procedure_name>    (<type_declaration>
<constant_name> : in <type_declaration>);

end <Package_Name>;
```

Select **Add Sources** under Project Manager. Select **Add or Create Design Sources**. Click **Create File** and select **VHDL** as the type.

Note: The example in this slide describes the steps for creating a package file in the Vivado IDE. You will need to manually code the package heading and the package body in an empty VHDL file.

User packages usually end with _pkg.

© Copyright 2019 Xilinx

Slide 20-13:

Declaring a Procedure or Function in a Package

- If a *procedure* or *function* is declared within the *package*, its "body" must be declared within the separate and dependent *package body*

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

package   MY_PACK   is
        procedure  SLV_REV (SLV_IN  : in  std_logic_vector ;
                            SLV_OUT : out std_logic_vector );
end package MY_PACK;

package body   MY_PACK  is

        procedure SLV_REV (SLV_IN  : in  std_logic_vector;
                           SLV_OUT : out std_logic_vector) is
          begin
             for i  in SLV_IN'length-1 downto 0  loop
               SLV_OUT(i) <=  SLV_IN(31- i);
             end loop;
          end  procedure SLV_REV;

end package body MY_PACK;
```

Procedure Declaration

Procedure Body

Note that the subprogram declaration is repeated in the package body. In general, this must match the declaration in the package.

Exceptions include the base specification for formal parameters—that is, octal versus hex or binary.

© Copyright 2019 Xilinx

Slide 20-14:

Calling the "Packaged" Procedure

In this call, the actual parameters in the call are positionally associated to the formal parameters in the procedure and must be of the same class*

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

entity MOD1 is
   port ( IN_BUS : in std_logic_vector (31 downto 0);
                . . .
          OUT_BUS : out std_logic_vector (31 downto 0)
        );
end  MOD1 ;

architecture RTL of MOD1 is
  begin
    process ( IN_BUS )
      variable TEMP_BUS : std_logic_vector (31 downto 0);
     begin
        SLV_REV ( IN_BUS, TEMP_BUS );
        OUT_BUS <= TEMP_BUS;
  end process;
end architecture RTL;
```

Sequential Call

* Note the use of the variable TEMP_BUS. The variable is declared within the process, passed as an argument into the procedure, and then copied to the output signal.

Although VHDL supports positional association when using functions, procedures, and instantiations, it is NOT considered good coding practice. Named association is much preferred as it is explicit and less prone to coding errors. The above “call” should read as: SLV_REV(SLV_IN=>IN_BUS, SLV_OUT=>TEMP_BUS);

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 20-15:

VHDL Predefined Packages

- The VHDL specification includes a built-in package called "standard"
  - You do not have to explicitly import the package
  - Includes definitions for a number of data types, including bit, character, string, real, integer, and time
  - A number of functions, including +, -, *, /, <, <=, =, =>, >
- Package "TEXTIO" is also included with VHDL
  - Includes declarations of types and subprograms that support I/O operations on text files
  - Must be explicitly imported into each module that uses any capability defined in TEXTIO

Slide 20-16:

IEEE Predefined Packages

- IEEE provides a number of standard packages
- Most common are
  - STD_LOGIC_1164 – Defines std_logic/std_logic_vector and supporting functions
  - NUMERIC_STD – Defines signed/unsigned and conversions with integers
  - STD_LOGIC_ARITH*
    - Not an official IEEE standard
    - Superseded by NUMERIC_STD
    - Originally written by Synopsys
  - STD_LOGIC_SIGNED/UNSIGNED*
    - Not an official IEEE standard
    - Superseded by NUMERIC_STD
    - Extends the capabilities of STD_LOGIC_ARITH

*Best design practice:

- Current version of Xilinx code templates offer a commented-out inclusion of the NUMERIC_STD package. If desired, the user need only to uncomment this line.

© Copyright 2019 Xilinx

- Do NOT use STD_LOGIC_ARITH/SIGNED/UNSIGNED for new designs!
- Older Xilinx code templates automatically inserted these obsolete packages for backward compatibility.

Slide 20-17:

Using Packages

- Use the *use* statement
- Syntax: *use* <library name>.<package name>.<item>
  - <item> may be the keyword "all" which will make the entire contents of the package available
  - <item> may be any single object within the package
- Examples
  - use std.TextIO.all;
  - use myLibrary.collectionOfMyFavoriteFunctions.some_specific_function;
- The <library name> must be declared prior to the *use* statement
  - Exceptions
    - Library "work" – Opened by default, this is the location of all of the source files
    - Library "STD" – Opened by default and contains packages "STANDARD" and "TEXTIO"

Syntax:

- <library name> - the name of the library where the package exists.
- <package name> - name of the package within the specified library.
- <item> - the name of the item within the package or the keyword “all” which will make the entire contents of the library available.

© Copyright 2019 Xilinx

Slide 20-18:

Using Components with the Same Name in Packages

- Consider a case where two different packages having two different functions or components, both with the same name

```
package first_pkg is
  component add_sub        -- 1st definition
        Port (a, b : in integer;
        c : out integer);
  end component;
end first_pkg;
```

```
package second_pkg is
  component add_sub        -- 2nd definition
   Port (a, b : in integer;
        c : out integer);
  end component;
end second_pkg;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.first_pkg.all;
use work.second_pkg.all;

entity operation is
end;

architecture Behavioral of operation is
signal x, y, z : integer;
signal p, q, r : integer;

begin
  pkg1 : entity work.first_pkg.add_sub (Behavioral) port map (a=>x, b=>y, c=>z);
  pkg2 : entity work.second_pkg.add_sub (Behavioral) port map (a=>p, b=>q, c=>r);

end Behavioral;
```


© Copyright 2019 Xilinx

Slide 20-19:

Summary

- Packages contain collections of constants, types, subtypes, functions, and procedure declarations and is a primary design unit
- Packages which contain functions and/or procedures require a package body (a secondary design unit) to describe the behavior of the functions and/or procedures
- Many packages already exist
  - VHDL intrinsically includes the STANDARD and TEXTIO packages
  - IEEE has standardized many packages
- You can create your own packages
  - Reduces code clutter
  - Supports code reuse
  - Organizes code for better ease of maintenance

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Interacting with the Simulation

Slide 21-1:

2019.1

This module explores the functionality of text I/O to extract meaningful textual information from the simulation tool and how to halt simulation by command. 15 minutes.

Slide 21-2:

Objectives

After completing this module, you will be able to:

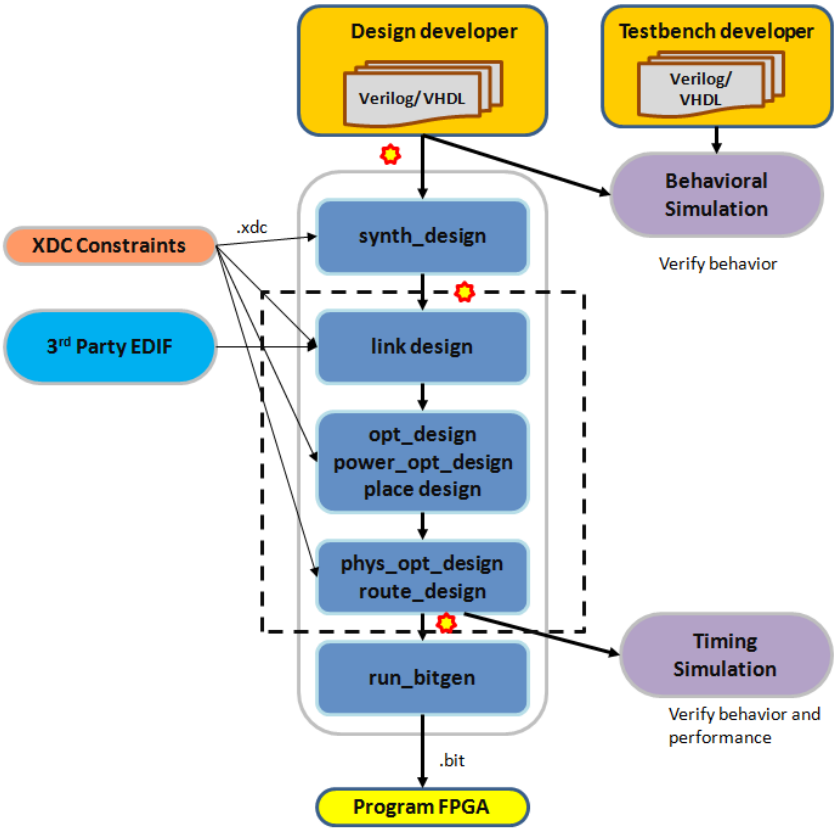
- Indicate the various points in the development flow appropriate for simulation
- Explain the contents of the TextIO library (output capabilities only)

© Copyright 2019 Xilinx

Slide 21-3:

Verification Plan

- Every project should have a verification plan
  - Defines points in the development process where testing must occur
  - Each test supports a specific specification or group of specifications
  - Defines what is considered a successful test



- Different types of verification
  - Simulation
  - Static timing analysis
  - Bench testing (Vivado® logic analyzer)
  - Field testing (Vivado logic analyzer)

In the figure, the stars represent common simulation points.

© Copyright 2019 Xilinx

Slide 21-4:

Types of Verification

- Simulation
  - RTL: Simulates the HDL code with no timing information
  - Functional: Simulates the synthesized netlist with no timing information
  - Timing: Simulates the implemented design, including timing
- Static timing analysis
  - Compares the implemented design to specified timing constraints
- Bench testing
  - Tests a physical system (prototype) in a controlled lab environment
- Integration
  - Get all parts of the system in one place talking to one another
- Field testing
  - Tests the final product in a real-world environment

Slide 21-5:

Interfacing with the Simulation Environment

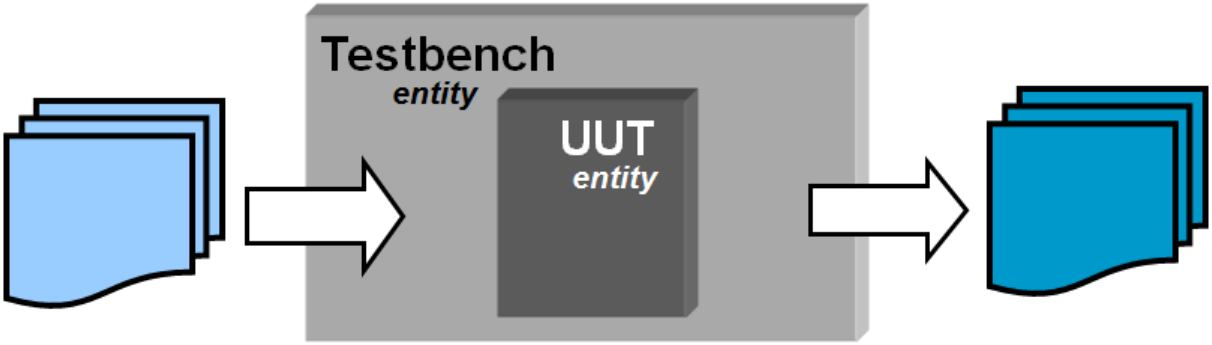
- Testbenches are quite useful for injecting and monitoring signals into and out of the UUT
- But this still leaves a gap
  - How can you monitor the progress of a simulation without having to trace out tedious waveforms?
  - How can you halt a simulation if something has gone wrong?
  - What about reporting the status of the simulation at specific (and absolute) times?

© Copyright 2019 Xilinx

Slide 21-6:

Simulation I/O Flexibility

- VHDL provides predefined overloaded functions and procedures for declaring and accessing files
  - These files can be used in the simulation environment for data input and output operations
- One type of data type is the "file" which supports binary file I/O
  - TextIO is a special case of this and offers ASCII text file operations



Slide 21-7:

Text I/O in VHDL

- A testbench can have a known test vector set with expected output vectors in a text file
- The text file can be read in, translated into input signals, and applied to the UUT
- The outputs from the UUT are compared with the translated expected signals from the file
- The number of mismatches, if any, can be counted and reported
- VHDL supports file input and output through the TextIO package
- The TextIO package provides a standard file type called TEXT and procedures for reading and writing

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 21-8:

Utility of Text I/O in the Real World

- Complex data sets can be generated using the MATLAB® software, C/C++, Java, or any other language capable of generating an ASCII text file
  - Data sets can be provided by third parties
  - Other languages are better at doing complex math than VHDL

- The Vivado logic analyzer has the ability to capture sampled data into a file
  - Used when an interface cannot be accurately modeled in simulation
  - Data is captured from the real world and played back in simulation

Saving Captured ILA Data to a File

Currently, the only way to upload captured data from an ILA core and save it to a file is to use the following Tcl command:

```
write_hw_ila_data my_hw_ila_data_file [upload_hw_ila_data
hw_ila_1]
```


© Copyright 2019 Xilinx

Slide 21-9:

TextIO and Its Basic Console Constructs

- Defined in the STD library
  - Library does not need to be explicitly listed, but TextIO must be accessed with a "use" statement
- Contains two important types
  - TEXT – References a file
  - LINE – Is a string corresponding to information being written to or read from the file
- Input and output to the console is managed through predefined "INPUT" and "OUTPUT" files
- Five primary functions
  - READLINE – Pulls a line of text from the specified file
  - READ – Pulls a specific piece of data from the line of text just read
  - WRITE – Puts a specific piece of data into the line
  - WRITELINE – Puts a line of text at the end of the specified file
  - ENDFILE – Indicates when the end of the file has been reached

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 21-10:

File Access

- You must define the file access
  - "INPUT" and "OUTPUT" are predefined in STD.TEXTIO for use with the console

- Syntax
  - file <file name> : TEXT open <mode> is <path designator>;

- After the file is open, it may be read from or written to, depending on how it was opened
  - Use the predefined procedures provided in STD.TEXTIO
    - Read, readline, endfile, write, writeline
    - Additional functions available for formatted I/O

Syntax:

- <file name> - any legal identifier which will be used to reference this file.
- <path designator> - any legal string specifying the relative or absolute path of a file in the designated operating system
- <mode> - either READ_MODE or WRITE_MODE or APPEND_MODE (defined in STD.STANDARD).

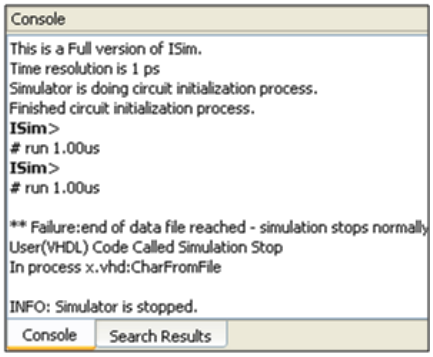
readline reads an entire line; writeline writes the presented string and appends a new-line character.

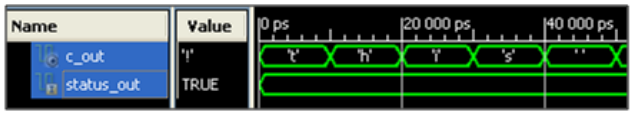
© Copyright 2019 Xilinx

Slide 21-11:

Reading Data from a File

```
charFromFile: process
    variable c : character := '?';
    file source_file : TEXT open READ_MODE is "source_on_disk.txt";
    variable lineOfTextFromFile : line;
    variable status : boolean := false;
  begin
    while (not ENDFILE(source_file)) loop
      readline(source_file,lineOfTextFromFile);
        for I in 1 to lineOfTextFromFile'length loop
          read(lineOfTextFromFile,c,status);
          status_out <= status;
          if (status) then
            c_out <= c;
          else
            c_out <= '?';
          end if;
          wait for 10 ns;
        end loop;
      end loop;

    assert false
    report "end of data file reached - simulation stops normally"
    severity failure;

  end process charFromFile;
```





Assume that c_out is a character signal and status_out is a boolean signal.

Arguments for the read function:

- lineOfTextFromFile is of type line that was filled by the readline call.
- c represents the object to read (in this case, a character because it was defined as such).
- status is a boolean that represents the status of the read command. If status is true, then the value placed in c can be trusted; otherwise, some type of error occurred in the reading process and c may not contain a valid value.

© Copyright 2019 Xilinx

Slide 21-12:

Writing Data to a File

```
stuffToFile: process
    variable message1 : string(1 to 36) := "***> Starting circuit initialization";
    variable charValue: character := '-';
    variable intValue : integer range 0 to 100 := 45;
    file source_file : TEXT open WRITE_MODE is "file_output_on_disk.txt";
    variable lineOut  : line;
  begin
    wait for 1 ps;

    write(lineOut, time'image(now) & ": " & message1);
    writeline(source_file,lineOut);
    wait for 50 ns;

    write(lineOut, time'image(now) & ": " & charValue & ' ' &
                   integer'image(intValue));
    writeline(source_file,lineOut);
    wait;

  end process stuffToFile;
```

file_output_on_disk.txt

```
1fs: ***> Starting circuit initialization
50001fs: - 45
```

Note that, like the time, the integer had to be converted to a string in order to be properly displayed.

So, why isn't the charValue treated in the same way?

Recall that a string is an array of characters, so by using the concatenation operators, the charValue merely becomes another member of the character array with no special conversion.

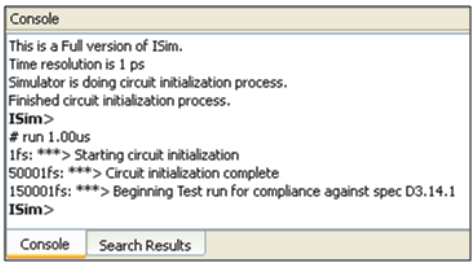
The other quantities (time and integer) are not comprised of characters or strings and, therefore, must be converted.

© Copyright 2019 Xilinx

Slide 21-13:

Writing Data to the Console

```
messagesToConsole: process
    variable message1 : string(1 to 36) := "***> Starting circuit initialization";
    variable message2 : string(1 to 36) := "***> Circuit initialization complete";
    variable message3 : string(1 to 59) := "***> Beginning Test run for compliance against
spec D3.14.1";
    variable lineOut  : line;
  begin
    wait for 1 ps;

    write(lineOut, time'image(now) & ": " & message1);
    writeline(OUTPUT,lineOut);
    wait for 50 ns;

    write(lineOut, time'image(now) & ": " & message2);
    writeline(OUTPUT,lineOut);
    wait for 100 ns;

    write(lineOut, time'image(now) & ": " & message3);
    writeline(OUTPUT, lineOut);
    wait;
  end process messagesToConsole;
```



Notice how the current time is displayed. Because the output requires a string, the current time provided by the function NOW must be converted. Using the predefined attribute “image” for type “time,” this value can now be displayed properly.

Note that some simulation tools will accept NOW without conversion. This is convenient, but not portable and may cause unforeseen problems.

This is a cleaner way of writing messages to the console instead of using the “assert false…” structure. To make things even cleaner, a designer could write a simple function that accepts a text string and prepends the current time and writes everything to the console. See the “writeNowToScreen()” function in the time utilities package which is part of the wave_gen lab design.

This code demonstrates an unformatted write where the width is given by the size of the string being written.

Some simulators support reading from the console which is similar to reading from a text file, but using the predefined file “INPUT”. ISim does not currently support this feature.

© Copyright 2019 Xilinx

Slide 21-14:

Formatting the Output

- Limited formatting is available with two additional arguments to the write function
  - Write (<line>, <value>, <justification>, <field_width> [, <digits> | , <units>]);

- Often std_logic and std_logic_vector data needs to be read or written
  - Support provided in IEEE.std_logic_textio
    - Use in addition to STD.textio
  - Provides additional functions
    - HREAD, HWRITE – Reading and writing hexadecimal values
    - OREAD, OWRITE – Reading and writing octal values
    - READ, WRITE – Reading and writing single digit values

- Third-party vendors may provide non-standard formatting packages
  - Limited formatting and string manipulation provided with the *wave_gen* design

Syntax:

- <line> - a string of type line.
- <value> - the value to write out.
- <justification> - either RIGHT or LEFT.
- <field_width> - an integer value representing the size of the field the string should be placed in.
- <digits> - only used in conjunction with a value of type REAL.
- <units> - only used in conjunction with time and may be of any type time (fs, ns, us, ms, s).

© Copyright 2019 Xilinx

Slide 21-15:

Apply Your Knowledge

1) A data file arrives from a vendor with five lines of comments before the actual data begins. How can VHDL skip to the useful data?

2) A simulation should terminate if the boolean signal "unrecoverable_error" is asserted. Write the assertion statement that will perform this.

3) How does a "wait;" differ from an assertion with a severity of failure?

A

1) A data file arrives from a vendor with five lines of comments before the actual data begins. How can VHDL skip to the useful data?

   A loop of five iterations can read a line of text using readline() and doing nothing with the data. The next readline from the file will contain useful data.

2) A simulation should terminate if the boolean signal “unrecoverable_error” is asserted. Write the assertion statement that will perform this.

   assert not unrecoverable_error

   report “Unrecoverable Error has been asserted – gotta quit now!”

   severity failure;

   Assertions trigger on false statements. Therefore, if unrecoverable_error asserts, then its complement is what should be checked for; otherwise the simulation will terminate when it should not terminate.

3) How does a “wait;” differ from an assertion with a severity of failure?

   Wait puts the process in which it is located into an infinite hold. Simulation continues, but the process does not. A severity of failure causes the simulation to stop, thus halting all processes.

© Copyright 2019 Xilinx

Slide 21-16:

Summary

- The TextIO package provides an easy mechanism for you to read and write to files and the user console
- The IEEE.std_logic_textio extends the capabilities of the STD.textio library to support std_logic and std_logic_vectors

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Finite State Machine Overview

Slide 22-1:

2019.1

This module provides an overview of finite state machines, one of the more commonly used circuits. 10 minutes.

Slide 22-2:

Objectives

After completing this module, you will be able to:

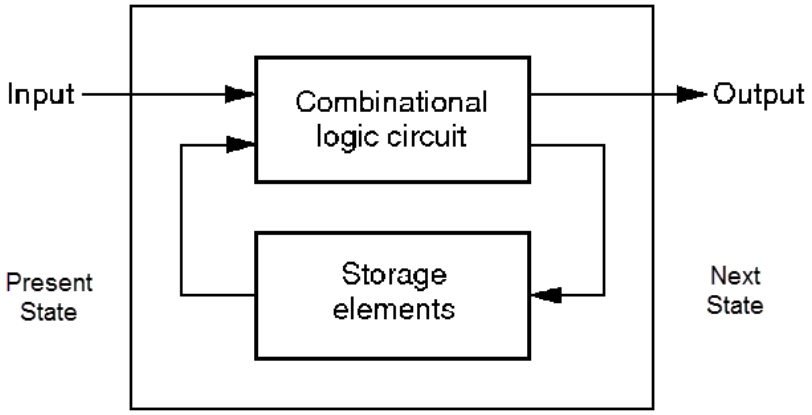
- Describe what a finite state machine is
- Describe the basic VHDL considerations for FSM

© Copyright 2019 Xilinx

Slide 22-3:

FSM Overview

- Sequential circuit having a combinational part along with a feedback
- Storage element is called a state
- Next states are outputs, while present states are past outputs



Slide 22-4:

FSM Types

- Finite State Machine can be of two types - Mealy or Moore machine
  - This has a direct bearing on how to approach coding
- In a Mealy machine, the output value is a function of the current state and the inputs
  - This means that output values can change while in a given state, which implies combinatorial logic on the outputs
- In a Moore machine, the output value is a function of the current state
  - In effect, there should be specific output values for each state, even if the same outputs occur in different states

Although a Mealy machine tends to produce somewhat fewer states than an equivalent Moore machine, it is generally recommended that Moore machines be used for FSMs in FPGAs.

© Copyright 2019 Xilinx

Having a single collection of outputs per state has several important advantages:

- It is much easier to test and validate a Moore machine. Consider if an output is predicated on a given state and five input conditions, then there are 32 possible combinations that must be evaluated for 100% coverage. Considering that a state machine is a causal device (that is, it is based on previous events), creating an exhaustive test suite would a daunting task.
- If the output is based on combinatorial logic, then it can cause extra layers of logic to be created, thereby degrading the performance of the FSM.
- Synthesis in the Vivado® Design Suite supports the implementation of properly coded state machines into block RAM. If the FPGA fabric is at a premium, then removing the FSM from fabric and placing it into block RAM will ease placement issues (providing, of course, that there are free block RAMs). An FSM that resides in a block RAM operates at the maximum capability of the FPGA, so if timing through the state machine is a critical issue, this is a worthwhile feature.

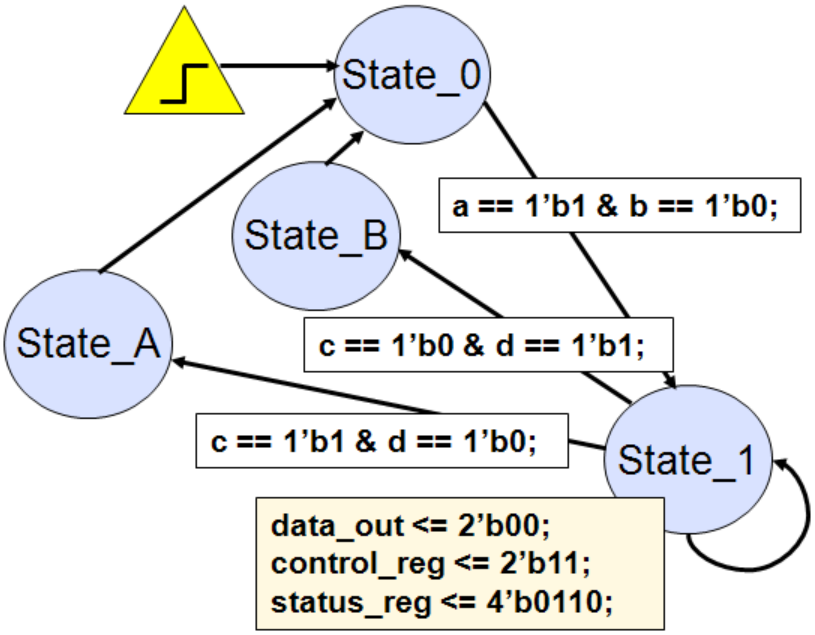
Best design practice:

- When coding an FSM, use Moore style machines.

Slide 22-5:

FSM State Diagram

- FSM is described using a state diagram
- Synchronous logic
  - Define states
  - Transition conditions
  - State events
  - Next-state determination
  - Initialization
  - Unreachable states



© Copyright 2019 Xilinx

Slide 22-6:

Encoding Methods (1)

- Auto-state encoding – the Vivado synthesis tool selects the encoding scheme
  - Recommended, but you can override with one of the following methodologies
- One-hot state encoding
  - Reduces decode logic to produce a fast FSM
  - Requires large number of flip-flops
- Gray state encoding
  - Guarantees only one-bit switches between two consecutive states
  - Good for FSMs with long state paths with little branching
  - Best suited for environments that may produce glitches in the FPGA

Note that whatever options are applied, using the synthesis options are applied globally.

Slide 22-7:

Encoding Methods (2)

- Johnson state encoding
  - Similar to gray state encoding
- Sequential state encoding – Default encoding scheme
  - Attempts to minimize next state equations
- User state encoding – Default encoding scheme
  - The Vivado synthesis tool uses the original encoding specified in the HDL file
  - See the *Vivado Design Suite User Guide: Synthesis* (UG901) for more details

© Copyright 2019 Xilinx

Slide 22-8:

Enumerated Data Types

- Enumerated data types enhance the readability of the code by allowing state names that reflect the functional description
  - The name represents a specific value or binary pattern
  - In some tools, you can directly control state encoding

```
architecture RTL of FSM is
                                  000,   001,    010,    011,   100
. . .
type MY_STATE is ( INIT, LOAD, FETCH, STOR_A, STOR_B) ;

signal CURRENT_STATE, NEXT_STATE : MY_STATE;
. . .
begin
```

Default Encoding*

*Default encoding refers to the VHDL Language Reference Manual (LRM) standard. A given synthesis compiler could choose to encode the signal according to its own optimization criteria. Encoding is controlled by the synthesis tool and can be selected by the designer or automatically selected by the tool.

For example, Synopsys uses a compiler directive called enum_encoding to allow you to specify what binary pattern each value of the designated enumerated type will carry.

Attribute enum_encoding of My_State is (“01010101”, “10001111”, …)

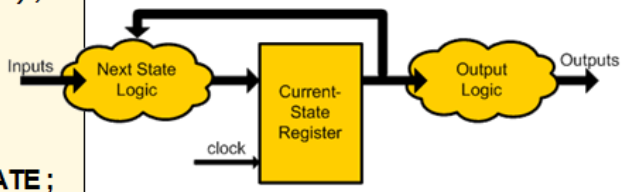
© Copyright 2019 Xilinx

Slide 22-9:

Using Constants

- Using VHDL constants allows you to directly control the encoding, which can be helpful when the current_state register output is used to directly drive other logic or device outputs
  - Note that during behavioral or RTL simulation, the user-specified binary value is shown, and not the literal—unlike enumerated data types

```
subtype MY_STATE is std_logic_vector ( 0 to 5 ) ;

constant INIT    : My_State := "111000" ;
constant LOAD    : My_State := "101010" ;
constant JUMP    : My_State := "000011" ;

signal CURR_STATE, NEXT_STATE : MY_STATE ;
. . .
begin --architecture
```



In this scenario, the FSM outputs are not decoded. They reflect the current-state value and can be used to directly drive the input to the next stage of logic.

Note that this type of user definition reduces the ability of the compiler to optimize the overall logic.

This can be a useful technique, but a difficult coding style to maintain. It is recommended that enumerated types should be used exclusively.

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 22-10:

Using Enumerated Data Types

- After signals of the defined enumerated data types are declared, they can be used throughout the code
  - They can only be assigned the specified literal values

```
architecture RTL of FSM is
...
case CURRENT_STATE is
      when LOAD =>
        if ( INP1 and INP2) then
           NEXT_STATE <= STOR_A ;
        else
           NEXT_STATE <= "010" ;
        ...
```

The primary reason for using enumerated data types is that they are visible during pre-synthesis simulation.

This is one of the powerful aspects of VHDL for source code debugging. It allows you to focus on functionality, rather than having to associate specific bit patterns.

Recall that the specific bit patterns are assigned after synthesis, and, therefore, are not known in the source code. More importantly, “010” is not of the same type as the state; therefore, the assignment is not legal.

© Copyright 2019 Xilinx

Slide 22-11:

Unreachable States

- Because most FPGA-orientated synthesis tools typically One-Hot Encode (OHE) an FSM, specifying a default condition at all times is good coding practice
  - OHE, or other encoding methods, creates a number of possible binary states not explicitly covered

```
type MY_STATE is
  ( INIT, LOAD, JUMP, STOR ) ;
signal CS, NS : MY_STATE ;
```

```
INIT   = "0001",
LOAD = "0010",
FETCH= "0100",
STOR  = "1000";
```

```
architecture RTL of FSM is. . .
 process  ( CS, INA, INB )
  begin
    case  CS  is
      when  INIT  => <sequential statements>;
      when  LOAD  => <sequential statements>;
      when  JUMP  => <sequential statements>;
      when  STOR  => <sequential statements>;
      when others => <default statements>;
    end case ;
 end process ;
```

Slide 22-12:

Summary

- A FSM is a way of modeling the behavior of systems, often described using a state diagram
- There are two types of FSM
  - Mealy FSM - Output is a function of the current states and the inputs
  - Moore FSM - Output is a function of the current states only
- Enumerated data types enhance the readability of the code by allowing state names that reflect the functional description

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Mealy Finite State Machine

Slide 23-1:

2019.1

This module describes the Mealy FSM and how to code for it. 10 minutes.

Slide 23-2:

Objectives

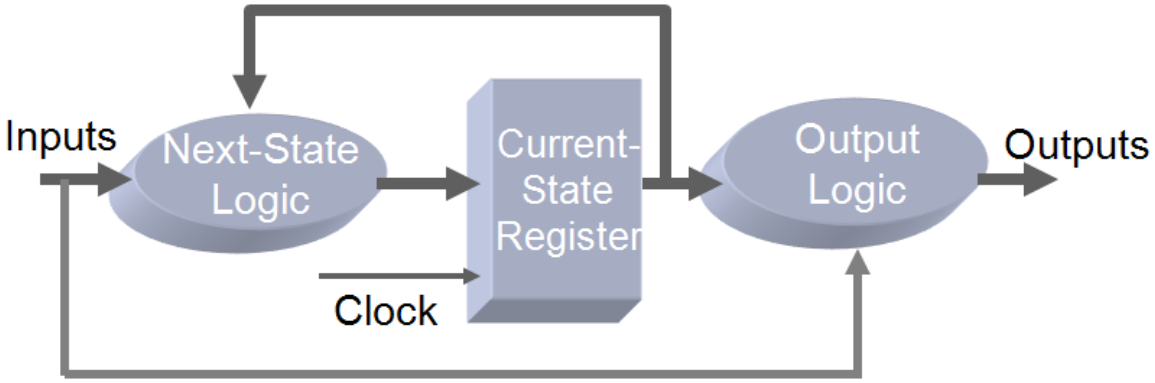
After completing this module, you will be able to:

- Describe what a Mealy finite state machine is

Slide 23-3:

Mealy Machine

- In a Mealy machine, the output value is a function of the current state and the inputs
  - Output values can change while in a given state, which implies combinatorial logic on the outputs



An advantage of a Mealy machine might be smaller overall size, which may be better suited to smaller FPGAs or FPGAs that use many state machines.

© Copyright 2019 Xilinx

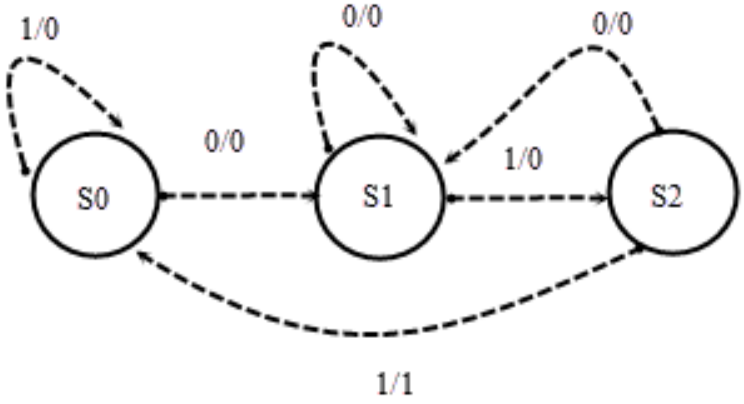
However, these are more difficult to verify as each input combination must be applied against every state. For example, if a Moore machine has five states, predicated on three inputs, then there are only five outputs to validate and the associated transitions from state to state.

A Mealy machine with five states, predicated on three inputs will require each input to be applied against each state output for a total of 15 verifications in addition to the transition validations.

Slide 23-4:

Mealy Machine State Diagram

- A Mealy FSM has fewer states compared to a Moore FSM
- Mealy outputs = function (state and input)
- In this example, a Mealy machine is used to model sequence detector logic
  - State diagram shows that the outputs are dependent on inputs
  - Detects 011 sequence (overlapping)



© Copyright 2019 Xilinx

Slide 23-5:

Assigning Mealy Outputs

- Mealy outputs reflect the current state and the inputs; thus, the inputs are considered in the process sensitivity

```
architecture RTL of FSM is
begin
. . .
process ( CURRENT_STATE, IN1, IN2, IN3 )
begin
        case ( CURRENT_STATE ) is
            when INIT =>
                if ( IN1 and IN2 ) = '1' then
                        OUT1_SIG     <=  '0'  ;
                        STATUS_REG <= "10" ;

                else  OUT1_SIG     <=  '1'  ;
                        STATUS_REG <= "00" ;

            when LOAD => . . .
```

In this example, the Mealy outputs are modeled as combinatorial.

© Copyright 2019 Xilinx

Slide 23-6:

Registered Mealy Outputs

- In a Mealy machine with registered outputs, make certain to keep the logic fully synchronous
- One approach is to declare signals within the architecture; use them to pass values to the outputs within the clocked process

```
architecture  RTL  of  FSM  is
 signal  OUT1_SIG_I : std_logic ;
 signal  STAT_REG_I : std_logic_vector ( 2 downto 0 ) ;
  begin . . .
    COMB: process ( CURRENT_STATE,  IN1,  IN2,  IN3 )
     begin
       case ( CURRENT_STATE )  is
         when  INIT =>
           if  ( IN1 and  IN2 ) = '1' then
              OUT1_SIG_I      <=  '0' ;
              STAT_REG_I      <= "10" ;
           else
             OUT1_SIG_I       <=  '1' ;
             STAT_REG_I       <= "00" ;
```

```
SYNC: process  ( CLK )
   begin
     if  rising_edge (CLK) then
       if ( RST = '1' ) then
         CURRENT_STATE <= INIT;
         OUT1_SIG      <= '0'  ;
         STAT_REG      <= "00" ;
       else
         CURRENT_STATE <= NEXT_STATE ;
         OUT1_SIG      <= OUT1_SIG_I ;
         STAT_REG      <= STAT_REG_I ;
       end if;
  end process SYNC ;
```

In this example, the output ports OUT1_SIG and STAT_REG are updated in the synchronous process, creating registered MEALY outputs.

Again, there is one additional clock-cycle latency on the output assignment. To compensate for this, you would need to code in a look-ahead fashion. This means assigning the output value within the previous state based on the state that you are branching to—rather than the state that you are currently in.

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Moore Finite State Machine

Slide 24-1:

2019.1

This module describes the Moore FSM and how to code for it. 10 minutes.

Slide 24-2:

Objectives

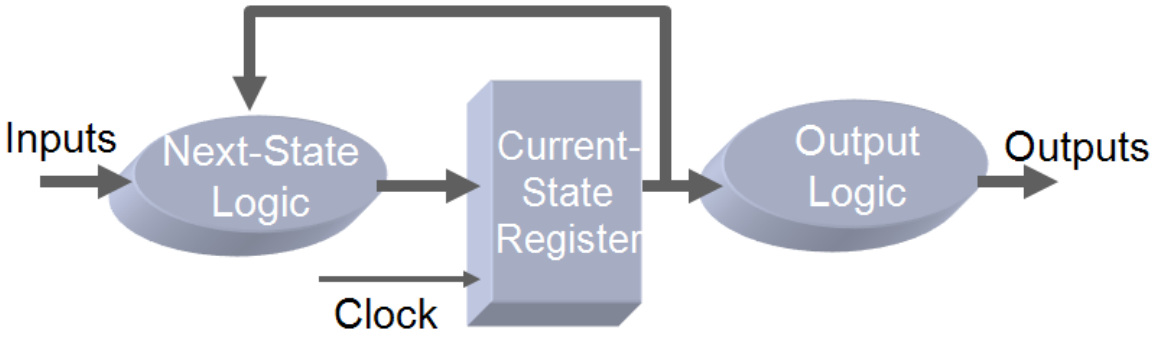
After completing this module, you will be able to:

- Describe what a Moore finite state machine is

Slide 24-3:

Moore Machine

- In a Moore machine, the output value is a function of the current state
  - In effect, there should be a specific output value for each state, even if the same outputs occur in different states



In general, Moore machines tend to have more states than a Mealy machine, but run faster and are much easier to validate and debug.

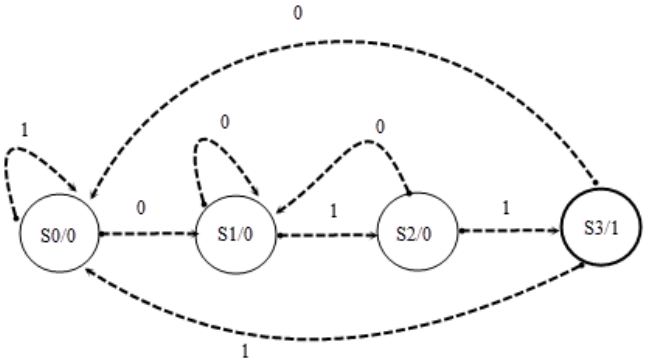
Synthesis in the Vivado® Design Suite also has a limited ability to move a properly coded state machine into block RAM. This can only be done for Moore machines.

© Copyright 2019 Xilinx

Slide 24-4:

Moore Machine State Diagram

- Moore outputs = function(state)
- Generally has more states than in a Mealy machine
- In this example, a Moore machine is used to model sequence detector logic
  - State diagram shows that the outputs are dependent on states only
  - Detects 011 sequence (non-overlapping)



Slide 24-5:

Assigning Moore Outputs

- Output values can be assigned along with next-state logic, as shown in the previous example or separately by using a process block

```
architecture RTL of FSM is
begin
. . .
process ( CURRENT_STATE )
begin
        case ( CURRENT_STATE) is
         when INIT =>
                OUT1_SIG      <=  '0' ;
                STATUS_REG <= "10" ;

         when LOAD =>
                OUT1_SIG      <=  '1' ;
                STATUS_REG <= "00" ;
                . . .
```

- This has little effect on synthesis results, but can greatly enhance readability

© Copyright 2019 Xilinx

In this example, the Moore outputs are modeled as combinatorial.

Also, coding in this fashion enables you to select a block RAM implementation. This frees up user logic, allows safe state implementation without performance impact, and enables very high speeds.

Slide 24-6:

Registered Moore Outputs

- One approach to making the Moore outputs registered is to place the assignments in a clocked process
- Note, however, that this approach creates one additional clock latency on the FSM output

```
architecture RTL of  FSM  is
begin
. . .
process ( CLK  )
begin
        if rising_edge ( CLK ) then

            case ( CURR_STATE)  is
             when  INIT =>
                        OUT1_SIG      <=   '0' ;
                        STATUS_REG <= "10" ;

             when LOAD =>
                       OUT1_SIG      <=   '1' ;
                       STATUS_REG <= "00" ;
                      . . .
```

- For Moore outputs, decoding NEXT_STATE rather than CURR_STATE in this process can compensate for the additional clock-cycle latency

In this example, the Moore outputs are modeled as sequential.

Also, coding in this fashion enables you to select a block RAM implementation. This frees up user logic, allows safe state implementation without performance impact, and enables very high speeds.

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

FSM Coding Guidelines

Slide 25-1:

2019.1

This module discusses finite state machines (FSM) as well as FSM implementation in an FPGA using VHDL. It also talks about the FSM guidelines that are used. 20 minutes.

Slide 25-2:

Objectives

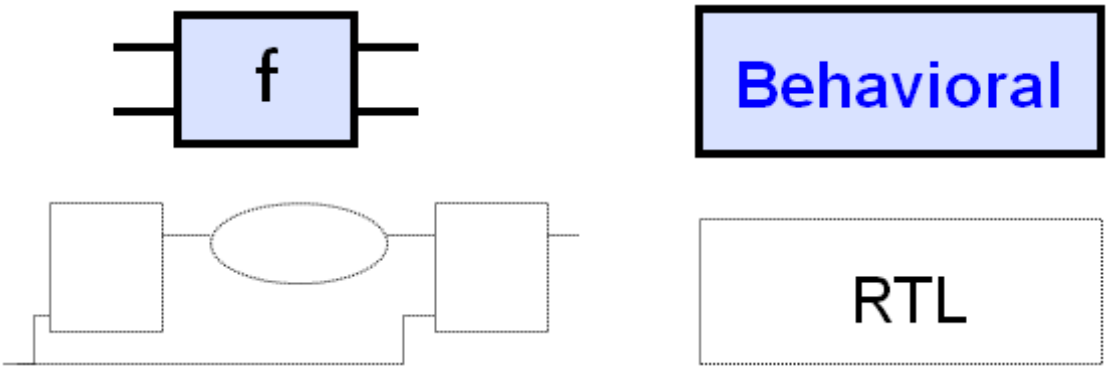
After completing this module, you will be able to:

- Enumerate principle good programming practices when using state machines
- List several mechanisms for implementing finite state machines

© Copyright 2019 Xilinx

Slide 25-3:

FSM Coding



- To model
  - Current-state logic
  - Next-state logic
  - Outputs
  - State names
  - Evaluate each state
  - Evaluate input conditions
- Use
  - Clocked process
  - Combinatorial process
  - Enumerated data types
  - Case statements
  - If/else statements

Notice that the language constructs you will use to model an FSM of any complexity are concepts that have already been covered in class.

© Copyright 2019 Xilinx

Slide 25-4:

Process Blocks

- Both the synchronous and combinatorial logic portions of the FSM are easily described by using VHDL process blocks
  - Use a synchronous process to describe current-state logic

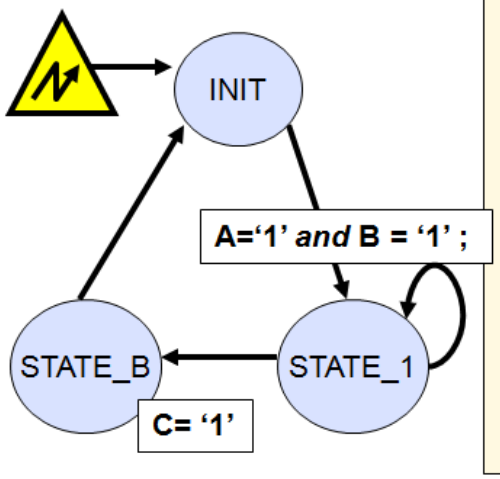
```
architecture RTL of  FSM  is
  begin
    . . .
      nxtSt: process  ( CLK )
         begin
           syncEvents: if rising_edge(CLK) then
              if (rst = '1') then
                CURRENT_STATE <= INITIAL_STATE;
              else
                CURRENT_STATE <= NEXT_STATE ;
              end if;
            end if syncEvents;
         end process nxtSt;
```

This example shows synchronous reset usage, which Xilinx generally recommends.

Slide 25-5:

Initialization

- Having an explicit form of initialization to force the FSM into a known state is absolutely essential
  - Otherwise, behavioral simulation and gate-level results may not match
  - Synthesis tool may not recognize code as a valid FSM and no optimization may occur




```
architecture RTL of  FSM  is
  begin
    . . .
    SYNC: process ( CLK )
      begin
        if  rising_edge ( CLK ) then
          if ( RST = '1' ) then
             CURRENT_STATE <= INIT ;
             sig_out       <= '0';
           else
             CURRENT_STATE <= NEXT_STATE ;
             . . .
          end if ;
        end if;
    end process  SYNC;
```

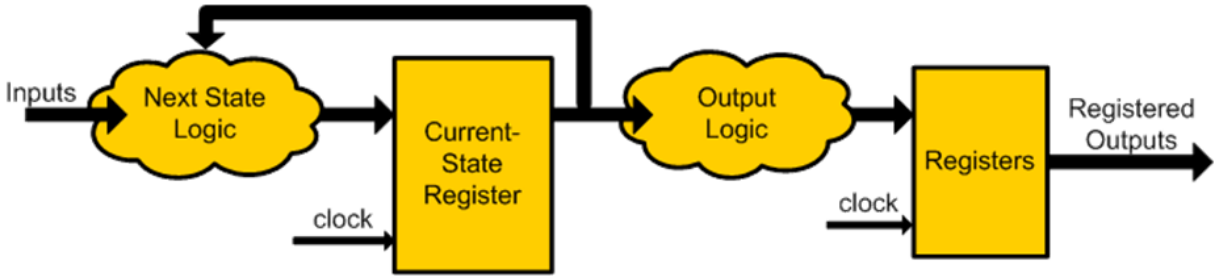

© Copyright 2019 Xilinx

Initialization can be synchronous or asynchronous, but synchronous resets are strongly preferred.

Slide 25-6:

Single-Process Block

- The additional registers on the output typically increase the amount of next-state combinatorial logic necessary to drive them
  - This also requires that state actions be coded in a look-ahead fashion



Again, within an FPGA, the use of more registers is the least of a designer's concerns and actually simplifies timing from the FSM output to the next stage of logic.

© Copyright 2019 Xilinx

Slide 25-7:

Single-Process FSM

- Although it is possible to code an FSM using a single-clocked process block, it is not generally recommended
- Advantages
  - Easier to code and maintain (everything to know about a given state is in one place)
- Disadvantages
  - The tools have a harder time optimizing code when used in this fashion
  - Cannot be "pushed" into block RAM

```
architecture RTL of  FSM  is
  begin
    . . .
    process ( CLK )
      begin
        if rising_edge ( CLK ) then
          if ( RST= '1' ) then
            STATE <= INIT;
            OUT1_SIG <= '0' ;
          else
           case (STATE) is
            when  INIT =>  . . .
            when  LOAD =>  . . .
            . .
```

Note that a single 'STATE' variable is used, not CURR & NEXT_STATE

Coding in this fashion guarantees that the outputs will be registered. If the outputs are exiting the module, this is congruent with best practices; if the signals are used internal to this module, it will add a clock cycle of latency.

Coding with multiple processes will remove the clock latency, but can introduce extra levels of logic which may negatively impact performance.

As noted earlier, FPGA architectures are register rich, so registering the outputs of the FSM is generally not of consequence, as some registers will otherwise go unused. Registering the FSM outputs usually helps to simplify the timing budget from this stage of logic to the next.

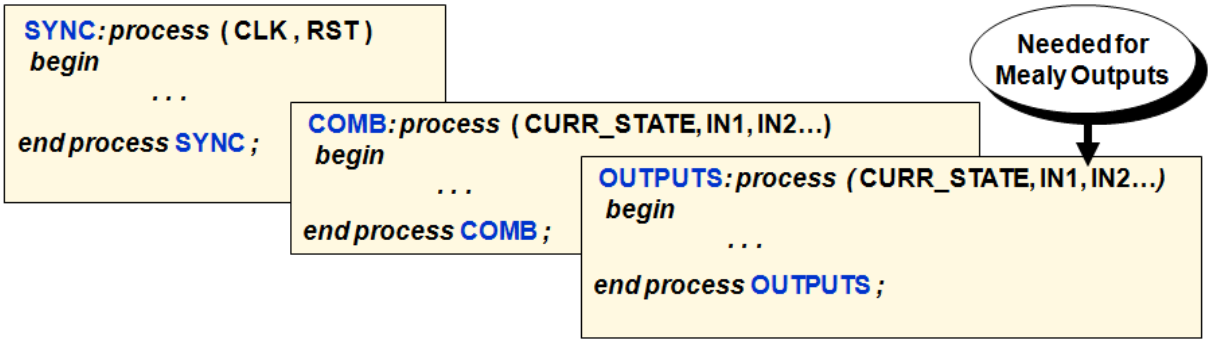
XST, which is the synthesis engine for the Project Navigator environment possesses the ability to "push" some state machines into a block RAM. This provides the highest level of performance offered by the FPGA. The FSM coding must follow certain guidelines for the tools to be able to map it to a block RAM.

© Copyright 2019 Xilinx

Slide 25-8:

Multi-Process FSM

- The standard recommendation is to use separate processes to model the sequential, combinatorial, and possibly the output logic
  - Usually requires less coding, can be more readable and intuitive, and can allow more compiler optimization by not forcing registered outputs
  - But information is spread out among multiple processes



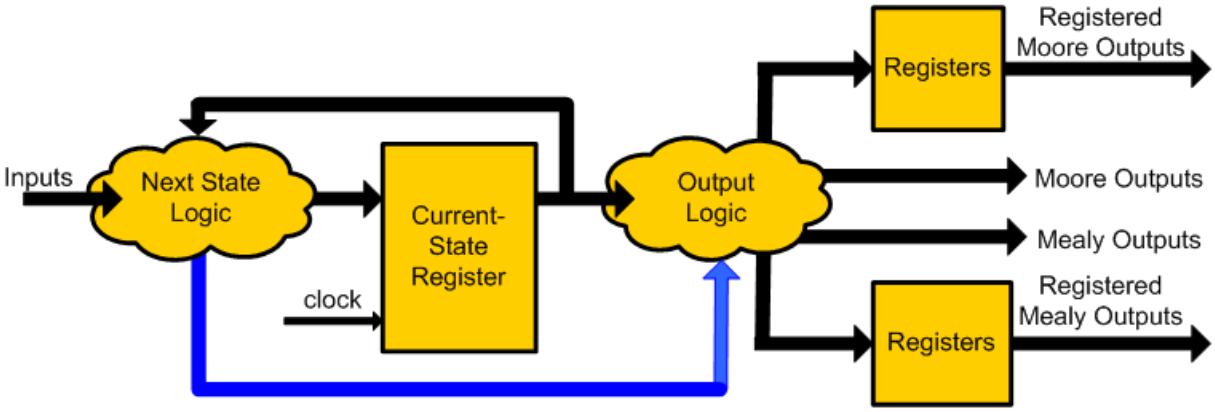
An FSM with registered outputs is generally a good idea for FPGAs, and the additional area and logic resource usage is negligible because the dedicated registers already exist in the datapath.

© Copyright 2019 Xilinx

Slide 25-9:

FSM Outputs

- In any given FSM, it is possible to have both Mealy and Moore outputs. In addition, the outputs can be registered or combinatorial
  - Conventional wisdom states that you avoid registered outputs because this increases area, making the FSM less optimal overall
  - However, within Xilinx FPGAs or similar FPGAs, registers are abundant and greatly simplify timing from one stage of logic to the next



The designer must be aware when registering the FSM outputs. This typically adds a one-clock cycle delay (latency) to the output assignment.

Best design practice:

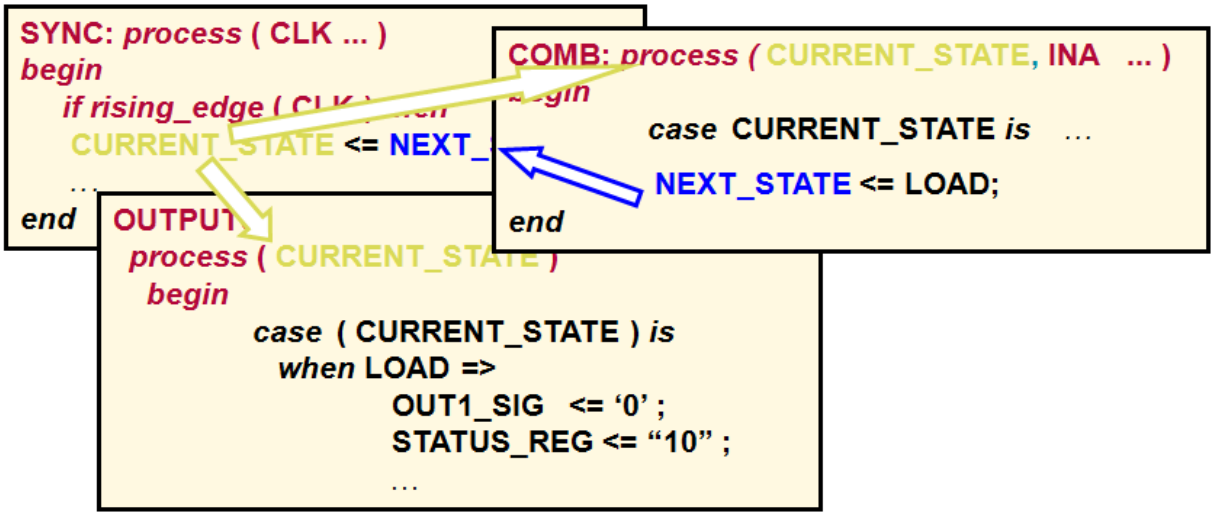
- Signals leaving every module should be registered. There are relatively few exceptions to this rule and they deal with very specific situations.

© Copyright 2019 Xilinx

Slide 25-10:

Concurrent Operations

- Because each process is concurrent, the synchronous update to CURRENT_STATE triggers execution of the asynchronous NEXT_STATE and output process blocks



When mixing synchronous and combinatorial processes it is important to realize that the combinatorial processes will be "running" while the synchronous process is waiting for the next clock edge.

Another way to look at this is that after the clock edge occurs and the synchronous process (usually the next state process) registers the new value, this new state value will propagate to the combinatorial processes and the outputs and next state signal will begin to change.

The point at which the outputs become stable, and thus usable, is dictated by the number of LUTs and routing resources that must be traversed.

When the state machine controls other devices in the design, it is best to register the outputs for a more deterministic (and usually higher performing) design.

© Copyright 2019 Xilinx

Slide 25-11:

Modelling Next-State Logic

- Use a separate combinatorial process to model NEXT_STATE and/or outputs
  - This separation can allow enhanced synthesis optimization

```
architecture RTL of FSM is
begin
. . .
COMB: process (CURRENT_STATE, INA, INB, INC ...)
begin
    case ( CURRENT_STATE ) is
        when INIT =>
                if ( INA = '1' ) then
                    NEXT_STATE <= LOAD ;
                else
                    NEXT_STATE <= FETCH ;
                end if ;
            OUT_SIG <= "10111010" ;
            . . .
```

Defining Next State

Assigning Outputs

The *when* statement constitutes another possible value for the selector expression and creates a separate branch within the case.

© Copyright 2019 Xilinx

Slide 25-12:

FSM Coding Example – UART Receiver Control

- Receives an 8-bit serial data stream from an RS-232 line
- Multiple-process state machine
  - genNextState computes the next state based on the current state, two counter states, and the incoming data
  - oversample_counter – the arriving serial signal is asynchronous, so this process generates a x16 sample clock
  - bitTracker keeps track of which one of the 8 data bits is currently being received
  - outCtrl captures the arriving data and generates the data_ready signal
  - ferrGen generates a framing error if the stop bit is not detected
- All processes are synchronous
- Part of the *wave_gen* design from the labs

This example differs from the previous types of state machines in that all the processes are synchronous. The positive effect of coding this way is that all output signals are registered, which is a good design practice.

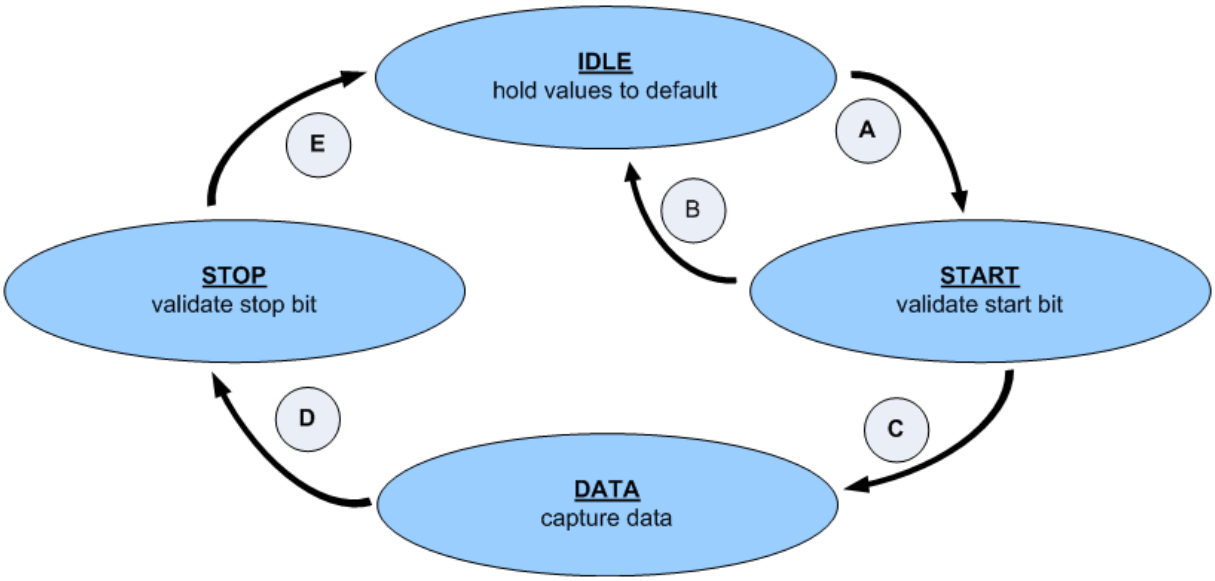
Module declarations:

```
type legal_rx_states is ( IDLE, START, DATA, STOP );
        signal state                    : legal_rx_states := IDLE;
        signal bit_cnt                  : integer range 0 to 9 := 0;    -- capture up
to 8 bits of data
        signal over_sample_cnt_done     : std_logic := 'U';             -- should
now be in the middle of the bit
        signal bit_cnt_done             : std_logic := 'U';             -- indicates
that all the data has been collected
```


© Copyright 2019 Xilinx

Slide 25-13:

Diagram



Transitions:

- A: When falling edge of serial data line is detected.
- B: When validation of start bit fails (serial data line high ½ a bit time later).
- C: When validation of start bit succeeds.
- D: After 8 data bits captured.
- E: Always

This state machine is completely synchronous and all transitions are not only timed to the clock edge, but can only occur every bit period (or half bit period in the case of START validation).

© Copyright 2019 Xilinx

Slide 25-14:

genNextState (1)

```
--
-- compute the next state
genNextState: process (clk_rx)
  begin
    if rising_edge(clk_rx) then                      -- when the CLOCK arrives
      if (rst_clk_rx = '1') then                     -- if the reset is asserted
         state <= IDLE;                              -- head back to the IDLE state
       else                                          --  do non-reset activities
         if (baud_x16_en = '1') then                 -- 16 times per bit...
           case state is                             -- do the tasks for this state
             when IDLE =>                            -- stay until line drops -START
               if (rxd_clk_rx = '0') then            -- is the receive line low?
                 state <= START;                     -- transition to the START state
               end if;                               -- end of test: receive line low

             when START =>                           -- verify this was not a glitch
               if (over_sample_cnt_done = '1') then  -- after 1/2 a bit time, then
                 if (rxd_clk_rx = '0') then          -- is the line still low?
                    state <= DATA;                   -- was the start bit get DATA
                  else                               -- line high, must've been glitch
                    state <= IDLE;                   -- return to the IDLE state
                  end if;                            -- end of checking line low
                end if;                              -- end of 1/2 bit time check
```

Slide 25-15:

genNextState (2)

```
           when DATA =>                          -- collect 8 bits of data
            if ((over_sample_cnt_done = '1') and -- if this bit is done AND
                (bit_cnt_done = '1')) then       -- all 8 bits are done
               state <= STOP;                    -- test for the STOP bit
             end if;                             -- end of check - all data collected

           when STOP =>                          -- test for the stop bit
             if (over_sample_cnt_done = '1') then-- have we captured another bit?
               state <= IDLE;                    -- look for the next START bit
             end if;                             -- end test of capturing STOP bit


         end case;                                -- end of case - determining state
       end if;                                    -- end of 16 times per bit
     end if;                                      -- end of non-reset activities
   end if;                                        -- end of synchronous events
 end process genNextState;
```


© Copyright 2019 Xilinx

Slide 25-16:

oversample_counter (1)

```
-- Oversample counter
-- Pre-load to 7 when a start condition is detected (rxd_sync is 0 while in
-- IDLE) - this will get us to the middle of the first bit.
-- Pre-load to 15 after the START is confirmed and between all data bits.
oversample_counter: process (clk_rx)
    variable over_sample_cnt : integer range 0 to 15 := 0; -- each bit sampled 16 times
  begin
    if rising_edge(clk_rx) then                  -- when the rising edge hits, then...
      if (rst_clk_rx = '1') then                 -- if the reset is asserted
        over_sample_cnt := 7;                    -- counter to .5 bits, since state IDLE
        over_sample_cnt_done <= '0';             -- don't tic during reset
      else                                       -- otherwise, do non-reset activities
        -- countdown if 16x baudrate is enabled
        if (baud_x16_en = '1') then              -- 16 times per bit...
          if (over_sample_cnt_done = '0') then   -- if the count is not yet complete,
            over_sample_cnt := over_sample_cnt - 1; -- then decrement the counter
            if (over_sample_cnt = 0) then         -- are we done yet?
              over_sample_cnt_done <= '1';        -- indicate that this count is complete
            end if;                               -- end of count check
```

Slide 25-17:

oversample_counter (2)

```
-- if the count has completed, should we preset the count?
        else                                     -- tasks between pulses,prep next count
          if ((state = IDLE) AND (rxd_clk_rx = '0')) then -- START bit condition
            over_sample_cnt := 7;                -- we need only wait 1/2 a bit width
            over_sample_cnt_done <= '0';         -- since count reset, clear done flag
          elsif (((state = START) AND (rxd_clk_rx = '0')) OR --  START & line is low, or
                  (state = DATA) OR              -- if we're collecting data, or
                  (state = STOP)) then           -- parsing the STOP bit, then
              over_sample_cnt := 15;             -- count the full 16 sample times
              over_sample_cnt_done <= '0';       -- count is reset, clear the done flag
            end if;                              -- end of check for times of 16 counts
          end if;                                -- end check of completion of count
        end if;                                  -- end of check for over sample enable
      end if;                                    -- end of non-reset activities
    end if;                                      -- end of synchronous events
  end process oversample_counter;
```


© Copyright 2019 Xilinx

Slide 25-18:

bitTracker

```
-- Track which bit we are about to receive
-- Set to 0 when we confirm the start condition
-- Increment in all DATA states
bitTracker: process (clk_rx)
  begin
    if rising_edge(clk_rx) then                -- when the rising edge hits, then...
      if (rst_clk_rx = '1') then               -- if the reset is asserted
        bit_cnt <= 0;                          -- reset the bit count
        bit_cnt_done <= '0';                   -- bit count done must be inactive
      else                                     -- otherwise, do non-reset activities
        if (baud_x16_en = '1') then            -- 16 times per bit...
          if (over_sample_cnt_done = '1') then -- has the count completed
            if (state = START) then            -- action depends on state we're in
              bit_cnt <= 0;                    -- clear the count as we're now starting
              bit_cnt_done <= '0';             -- starting new count
            elsif (state = DATA) then          -- collecting data?
              if (bit_cnt = 7) then            -- are we done?
                bit_cnt_done <= '1';           -- count done
              end if;                          -- end of check for completion of count
              bit_cnt <= bit_cnt + 1;          -- one more piece of data gathered
            end if;                            -- state checks
          end if;                              -- end of count not completed
        end if;                                -- end of oversample enabled tasks
      end if;                                  -- end of non-reset activities
    end if;                                    -- end of synchronous events
  end process bitTracker;
```

Slide 25-19:

outCtrl

```
-- Capture the data and generate the rdy signal
-- The rdy signal will be generated as soon as the last bit of data
-- is captured - even though the STOP bit hasn't been confirmed. It will
-- remain asserted for one BIT period (16 baud_x16_en periods)
outCtrl: process (clk_rx)
  begin
    if rising_edge(clk_rx) then                    -- when rising edge hits, then...
      if (rst_clk_rx = '1') then                   -- if the reset is asserted
        rx_data     <= X"00";                      -- drive the output to all zeros
        rx_data_rdy <= '0';                        -- ensure data ready inactive
      else                                         -- non-reset activities
        if ((baud_x16_en = '1') AND                -- 16 times per bit... AND
            (over_sample_cnt_done = '1')) then     -- time to keep this sample
          rx_data_rdy    <= '0';                   -- keep the data ready deasserted
          if (state = DATA) and (bit_cnt < 8) then -- capturing data, but not stop
             rx_data(bit_cnt) <= rxd_clk_rx;       -- capture into the array
           elsif (state = STOP) and                -- if we're in the stop state and
                 (over_sample_cnt_done = '1') then -- done counting means stop bit
             rx_data_rdy <= '1';                   -- assert the data ready
           end if;                                 -- end of check for getting data
         end if;                                   -- end of oversample check
       end if;                                     -- end of non-reset activities
     end if;                                       -- end of synchronous events
   end process;
```


© Copyright 2019 Xilinx

Slide 25-20:

ferrGen

```
-- Framing error generation
-- Generate for one baud_x16_en period as soon as the framing bit
-- is supposed to be sampled
ferrGen: process (clk_rx)
  begin
    if rising_edge(clk_rx) then                  -- when rising edge hits, then...
      if (rst_clk_rx = '1') then                 -- if the reset is asserted
        x <= '0';                          -- ensure framing error is disabled
      else                                       -- otherwise, do non-reset activities
        if (baud_x16_en = '1') then              -- 16 times per bit...
          if ((state = STOP) AND                 -- if we're in the STOP state and
              (over_sample_cnt_done = '1') AND   -- we've finished counting and
              (rxd_clk_rx = '0')) then           -- the line is low, then
            frm_err <= '1';                      -- error present
          else                                   -- otherwise
            frm_err <= '0';                      -- no error present
          end if;                                -- end of check for framing error
        end if;                                  -- end of oversample check
      end if;                                    -- end of non-reset activities
    end if;                                      -- end of synchronous events
  end process;
```

Slide 25-21:

Apply Your Knowledge

1) Why would separate process blocks allow greater logic optimization?

2) What is one possible advantage of a single-process FSM?

1) Why would separate process blocks allow greater logic optimization?

Separate blocks eliminate the additional registers on outputs and allow greater optimization of combinatorial next-state logic.

2) What is one possible advantage of a single-process FSM?

All aspects of the state machine are encapsulated into one block of code. Recall that this is not the preferred method for high-performance designs, but not every design must run at top speed.

© Copyright 2019 Xilinx

Slide 25-22:

Summary

- Enumerated data types enhance the readability of the code by allowing state names that reflect the functional description
- Both the synchronous and combinatorial logic portions of the FSM are easily described by using VHDL process blocks
- Use a separate combinatorial process to model next-state logic and/or outputs
- One-Hot Encoding (OHE) of an FSM generally provides improved performance within an FPGA device
  - Encoding options generally user-selectable

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Vivado Simulator and Race Conditions in VHDL

Slide 26-1:

2019.1

This module introduces the Vivado® simulator simulation environment, including how to set the default configuration for the tool, configuring the waveform, and setting breakpoints. It also talks about race conditions in VHDL. 15 minutes.

Slide 26-2:

Objectives

After completing this module, you will be able to:

- Describe the relationship between the Vivado simulator and tools in the Vivado Design Suite
- Using the Vivado simulator, demonstrate how to
  - Locate and select signals for simulation
  - Simulate for a fixed period of time
  - Set breakpoints
  - Single step
  - Locate points of interest in the waveform
- Define race conditions in VHDL

© Copyright 2019 Xilinx

Slide 26-3:

Vivado Simulator

- The Vivado IDE provides an integrated simulator environment using the Vivado simulator
- The Vivado simulator can be launched from the Vivado tool environment
- The Vivado simulator GUI window supports
  - Adding and removing signals
  - Navigating waveforms
  - Running and restarting simulation

The Vivado simulator supports functional and timing simulation.

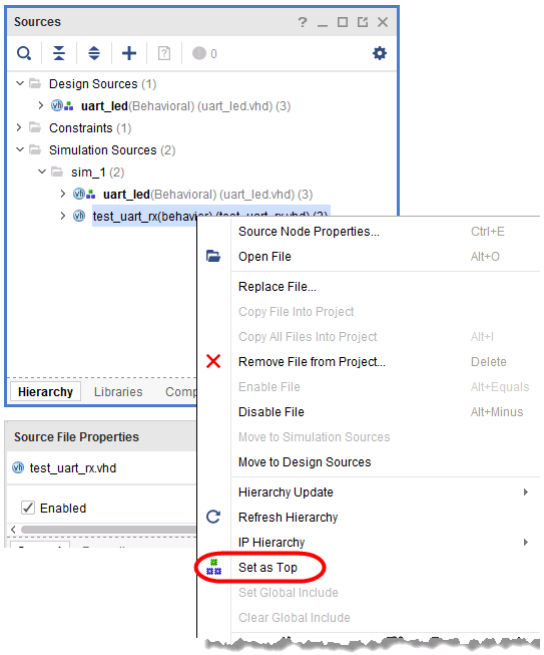
The Vivado simulator provides features including real-time waveform update, source code debugging (using breakpoint and step options), SDF annotation for timing simulation, and SAIF dumping for power analysis.

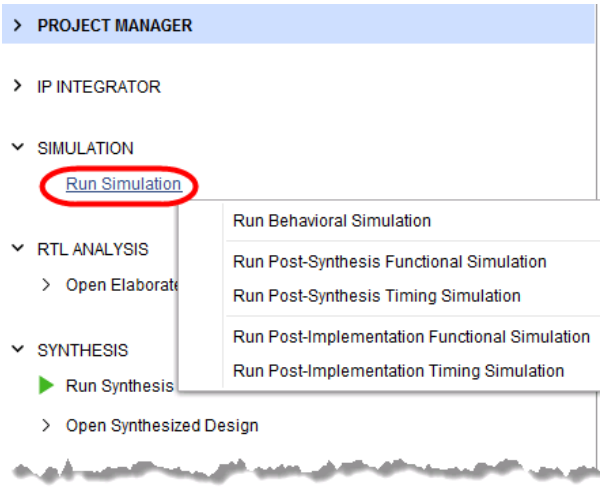
© Copyright 2019 Xilinx

Slide 26-4:

Accessing the Vivado Simulator in the Vivado IDE

- Behavioral simulation (direct simulation of the source code) is accessed from the Flow Navigator
- The Vivado tool requires the top of the hierarchy to be specified





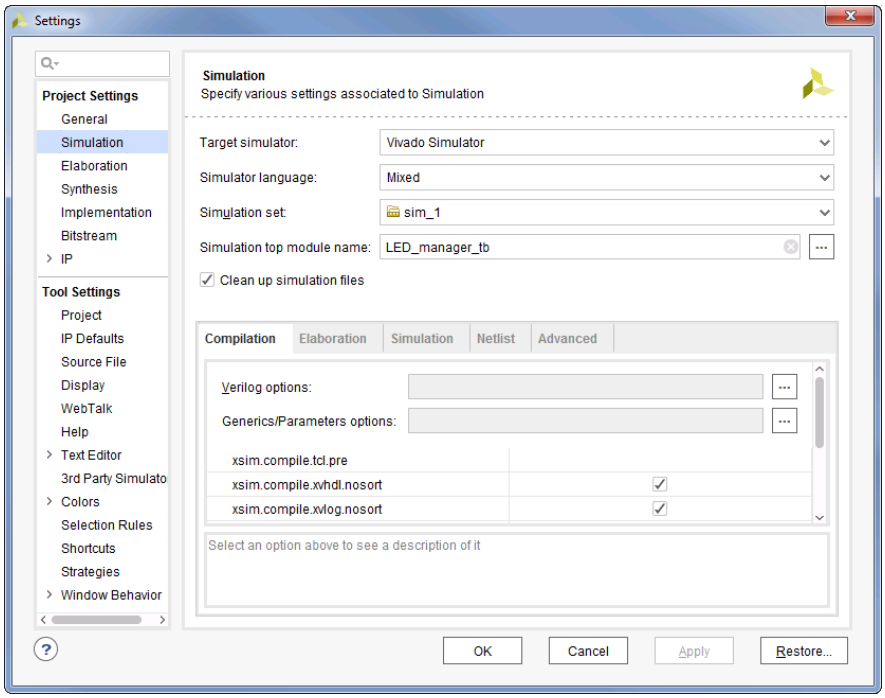
- When selected, all the top-level testbenches are displayed
  - Select the appropriate testbench to simulate
- *Warning:* The process will be applied to any item that is available in the top module list
  - Only simulate testbenches

© Copyright 2019 Xilinx

Slide 26-5:

Simulation Settings

- From the Flow Navigator, click Simulation Settings
- Allows selection of compilation and simulation properties
  - Additional options can be entered via more options



The Simulation settings dialog box allows to select between the third-party simulators or the Vivado simulator. You can provide the top module name of the design in Simulation top module name field.

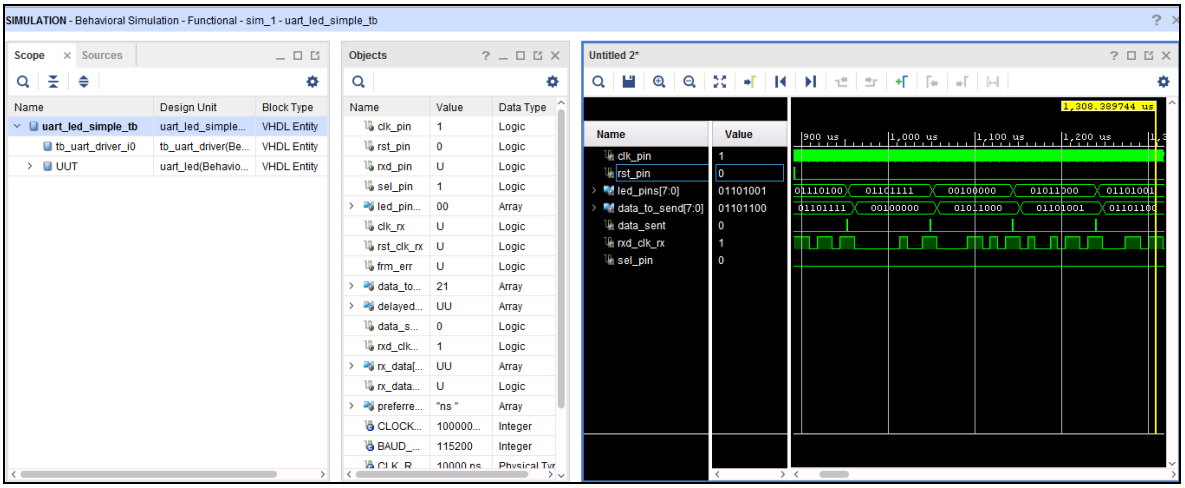
The tabs available in this dialog box are:

- Compilation tab – Manages compiler directives for the simulation fileset
- Elaboration tab – Manages elaboration directive.
- Simulation tab – Manages simulation directives.
- Netlist tab – Manages netlist configuration options.
- Advanced tab – Provides option that ensures all the files from design sources with the files from the current simulation set will be used for simulation.

© Copyright 2019 Xilinx

Slide 26-6:

Running the Vivado Simulator

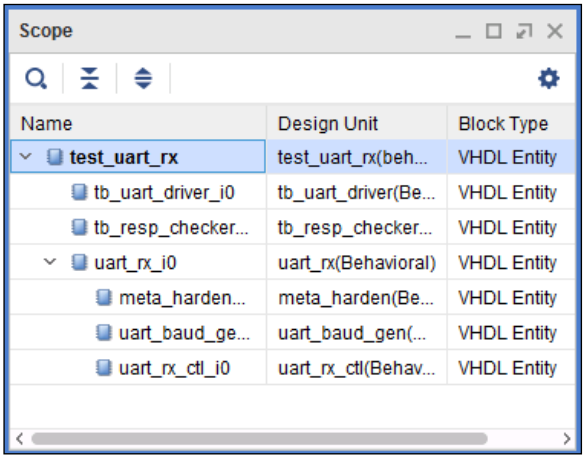


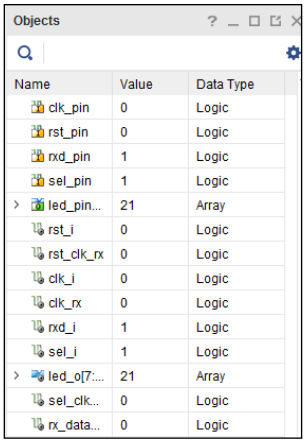
The Vivado software simulator supports behavioral (RTL), netlist, and timing simulation.

Slide 26-7:

Selecting Signals

- The Scopes view supports hierarchical signal browsing
  - Entire hierarchical signal collections can be imported into the viewer
- The Objects window enables individual signal selection





© Copyright 2019 Xilinx

Slide 26-8:

Simulator Simulation Controls

- Controlling the simulation
  - Restart - Restarts the simulation from time 0
  - Run all - Runs the open simulation for completion
  - Run for - Allows you to specify time for simulation
  - Step - Runs the simulation till next source line of code
  - Break - Pause the currently running simulation
  - Re-launch - Recompiles simulation files and reruns the simulation

Slide 26-9:

Simulator View Controls

- Viewing the results
  - Zoom in/out
  - Zoom full range
  - Zoom Fit
  - Go to Cursor
  - Go to time 0
  - Go to last time
  - Jump to previous transition on selected signal
  - Jump to next transition on selected signal
  - Insert marker
  - Jump to previous
  - Jump to next marker
  - Swap cursors

© Copyright 2019 Xilinx

Slide 26-10:

Debugging with Vivado Simulator

- Select any level of hierarchy or item from the object window, right-click, and select Go to Source Code
  - Immediately opens the containing file and positions the cursor at the object
- Step through the code line by line
- Breakpoints can be set or cleared by clicking the region between the line number and the source window
  - Only lines with executable code are selectable
  - Runs the simulation until breakpoint is reached

```
71      getHard: process (clk_dst)
72         begin
73            if rising_edge(clk_dst) then        -- detect synchronous events
74               if (rst_dst = '1') then          -- if reset is asserted
75                  signal_meta <= '0';           -- clear the output of the first flip-flop
76                  signal_dst  <= '0';           -- clear the output of the second and final flip-flop
77               else                             -- do non-reset activities
78                  signal_meta <= signal_src;    -- capture the arriving signal - higher probability of beir
79                  signal_dst  <= signal_meta;   -- resample the potentially meta-stable signal, lowering th
80               end if;                          -- end of reset/non-reset activities
81            end if;                             -- end of synchronous event check
82         end process getHard;
```


© Copyright 2019 Xilinx

Slide 26-11:

What is a Race Condition?

- Race condition can occur when there is a parallel execution of HDL code
  - Two expressions are scheduled to execute at the same time
  - The order of the execution is not deterministic
  - Expressions are interdependent
- Every change in the value of a signal or variable is consider an event
- All the processes are sensitive to events, and these events are evaluated in an arbitrary order
- Events can occur at different times
- Events are kept in the event queue to make sure they are processed in the correct order—known as the scheduling of an event

Slide 26-12:

Race Conditions in VHDL

- Language restriction results in the lower probability of a race condition in VHDL
  - Processes can communicate either through signals or shared variables
  - Signal update takes 1 delta time in the same event queue
  - Only a resolved signal can have multiple drivers, which avoids writing the same signal or variable in two different processes
  - Asynchronous designs have a higher probability of race conditions

© Copyright 2019 Xilinx

Slide 26-13:

Race Condition in VHDL Example

- Race condition example

```
process (clk)
        if clk'event and clk = '1' then
                var1 <= '1';
        endif;
end process;

process (clk)
        if clk'event and clk = '1' then
                var1 <= '0';
        endif;
end process;
```

RACE?

Slide 26-14:

Steps to Avoid Race Conditions

- Do not assign different values to the same signal in two different processes
- Do not read and write using the same signal
- Do not initialize at time zero
- Use concurrent assignments for combinational designs and process blocks for sequential designs
- If you want gated outputs from the flip-flops, put them in continuous assignments
- Put driving and sampling at different clock edges

© Copyright 2019 Xilinx

Slide 26-15:

Summary

- The Vivado simulator is integrated with the Vivado IDE
- Simulation executable is created within the Vivado IDE
- The Vivado simulator waveform viewer "runs" the simulation and supports
  - Customization of the waveform
  - Zoom/pan controls
  - Marker
  - Single stepping the source code
  - Breakpoints
- Race condition occurs when there is parallel execution of HDL code

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Writing a Good Testbench

Slide 27-1:

2019.1

Knowing all the right commands is not sufficient to write a good testbench. This module explores how time-agnostic, self-checking testbenches can be written and applied. 15 minutes.

Slide 27-2:

Objectives

After completing this module, you will be able to:

- Describe how a self-checking testbench can be constructed
- Illustrate proper annotation techniques

Slide 27-3:

Review of the Testbench

- Useful for injecting and monitoring signals into and out of the UUT
- Non-synthesizable modules can be added to the testbench
  - To model board or system-level devices
  - To perform data validation
- Per the previous module
  - How can you monitor the progress of a simulation without having to trace out tedious waveforms?
    - Assertions cover part of this, but is there another way?

© Copyright 2019 Xilinx

Slide 27-4:

Principle

- Based on multiple approaches
  - If the same problem is solved in multiple fashions and the same answer is produced by all approaches, then there is a high probability that the process is sound
- Technique often employed by chip builders
  - Simulations built in VHDL and in C/C++ and Java
  - Implemented by different groups to identify possible ambiguities in the specification
  - Simulations are compared to each other to validate results
- Multiple languages are not always necessary
  - VHDL can use pure behavioral (that is, simulation-only constructs) models and compare against a synthesizable design

It is best when the testbench is written by a different group than the one that coded the synthesizable design. Although specifications are supposed to be unambiguous, this may be a difficult requirement to meet and to test.

One method of determining the clarity of the specification is to have different groups read and implement it. Differences between the simulation model of the design and the synthesizable model of the design can be caught with this process. Once found, the specification must be corrected and the portions of the code—both in the behavioral model and the synthesizable model—must be made.

Once again, there is a distinction drawn between the behavioral model and the behavioral coding style. The behavioral model only exists within the simulation environment. It may be synthesizable, but this is irrelevant. The behavioral model may, and often will use, a mixture of all of the coding styles—RTL, structural, and behavioral.

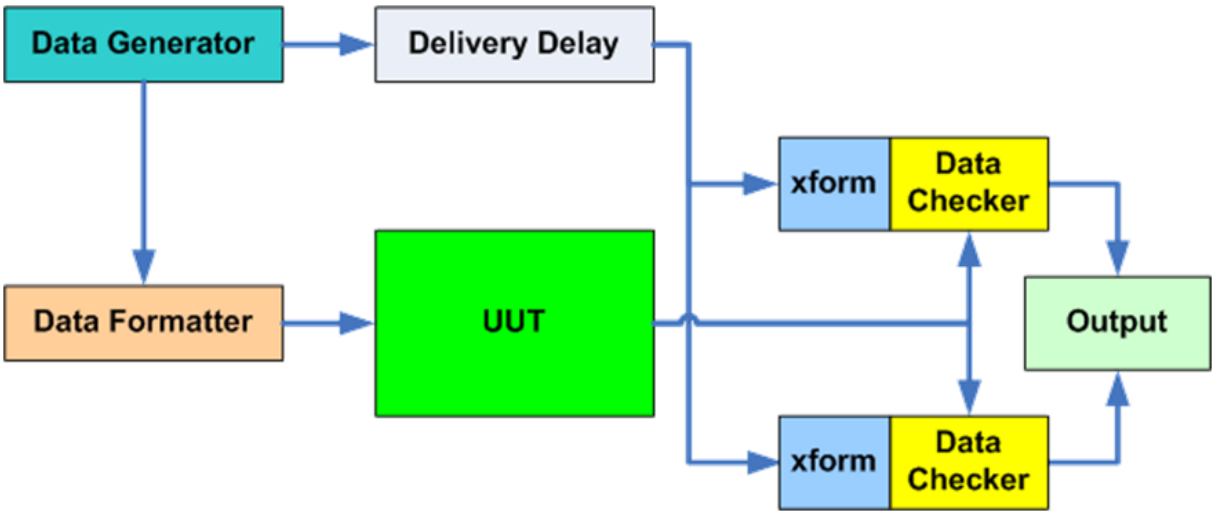
Behavioral coding style uses higher levels of abstraction, such as loops, case statements, and processes to describe the design, rather than specifying how signals and registers combine (RTL) or explicitly instantiating primitives or macros (structural).

This level of testbench design is recommended for the top-level of a design or major sub-portion of the design. Often, it is sufficient to validate a lower level design by staring at output waveforms.

© Copyright 2019 Xilinx

Slide 27-5:

Top Level

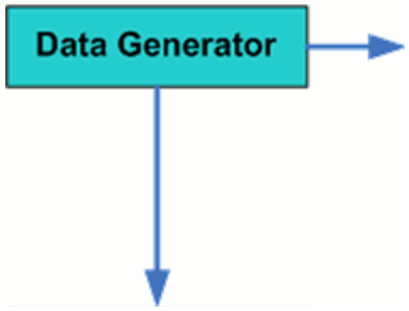


The next few slides provide an explanation of the modules in the diagram.

Slide 27-6:

Data Generator

- Produces appropriate data sets for testing
- Focus is on *what* data is input rather than *how* the data is provided to the UUT
  - Data can be statically stored within the VHDL code
  - Data can be generated using VHDL
    - Math libraries
    - Real and physical types to model the environment that the UUT will reside in
  - Use TEXTIO to read from data files that are produced
    - Manually
    - Automatic data generators, such as the MATLAB tool
    - Custom C/C++ or Java program



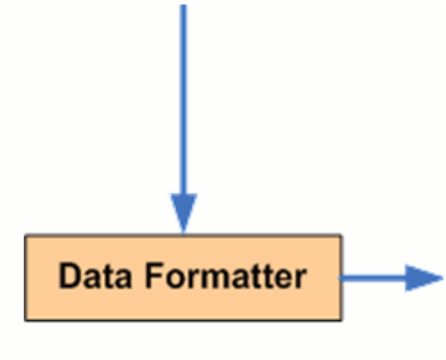
© Copyright 2019 Xilinx

Slide 27-7:

Data Formatter

- Converts the data produced by the Data Generator module into the proper sequence and type for use by the UUT

- Examples
  - Character to RS-232 converter
  - Block data to Transaction Layer Packet (TLP) for PCIe® core designs
  - Analog voltages converted to equivalent std_logic_vector for modeling ADCs



Slide 27-8:

Delivery Delay

- The UUT takes some amount of time to process the data

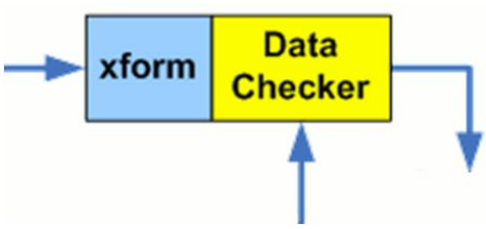
- Process time varies based on the type of simulation
  - Source simulation will have zero routing delays
    - Delays based on clocked processes (latency)
    - Post-place and route will have routing delays
  - Avoid re-writing the delivery delay mechanism for each type of simulation by using an asynchronous FIFO
    - Data is pulled from the FIFO when the UUT produces an output

© Copyright 2019 Xilinx

Slide 27-9:

Data Checker

- Compares the delayed original data with the result of the UUT
- May require a behavioral model of the portion of the UUT
  - Output of UUT may not be a straight transform of the input
- Outputs a success or failure message to the output
  - Console or a text file
  - Text file output has the additional benefit of documenting the test run

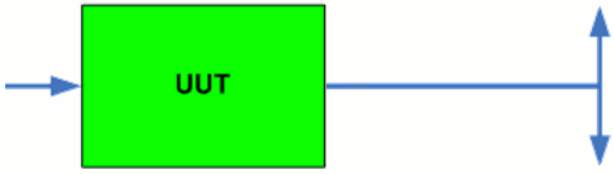


Xform – The transform model may be as simple as a straight-through connection, as it is in the uart_led lab example, or to a full and complete behavioral model of the UUT.

Slide 27-10:

Unit Under Test

- UUT – Unit under test
- Usually the synthesizable design that will ultimately reside in the FPGA
  - Occasionally a model is needed that is only intended to run in the simulation environment
    - Proof-of-concept type designs and academic studies
  - Does not need to be the complete design that resides in the FPGA
    - Remember that testing lower-level modules prior to integration with the next level of hierarchy is strongly recommended

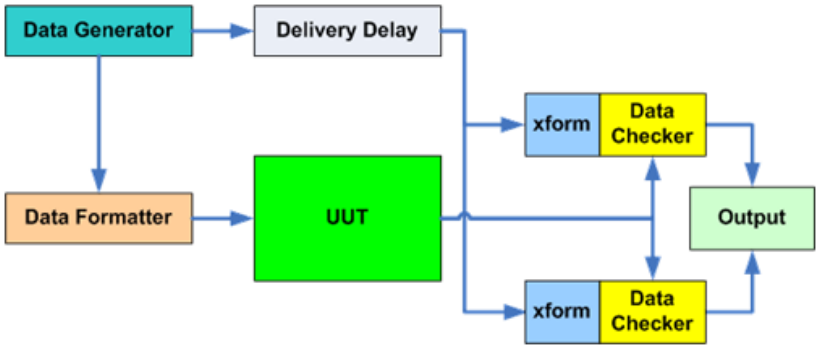


© Copyright 2019 Xilinx

Slide 27-11:

Overview of UART_led_tb.vhd

- Data Generator – Coded in module tb_datagen_i0
- Data Formatter – Included in module tb_datagen_i0
- Delivery Delay – Coded as asynchronous FIFO in tb_fifo_pkg in tb_datachk_i0
- UUT – Coded in module uart_led
- Xform/Data Checker – Coded in module tb_datachk_i0
- Additional processes
  - Welcome – Introduces the testbench and base parameters
  - gen_clk_rx – Produces the clock
  - reset_proc – Manages the reset



© Copyright 2019 Xilinx

Slide 27-12:

Understanding the Delivery Delay Example (1)

```
package tb_fifo_pkg is

    constant MAX_WIDTH : integer := 8;
    constant MAX_DEPTH : integer := 16;

    subtype word   is std_logic_vector(MAX_WIDTH-1 downto 0);
    type    memory is array(0 to MAX_DEPTH-1) of word;

    type fifo_type is record
        width    : integer range 1 to MAX_WIDTH;
        depth    : integer range 1 to MAX_DEPTH;
        read_ptr : integer range 0 to MAX_DEPTH-1
        write_ptr: integer range 0 to MAX_DEPTH-1;
        full     : boolean;
        empty    : boolean;
        n_writes : integer;
        n_reads  : integer;
        volume   : integer range 0 to MAX_DEPTH;
        contents : memory;
      end record;
```

Slide 27-13:

Understanding the Delivery Delay Example (2)

```
  -- Declare functions and procedure
  procedure reset(variable fifo_instance : inout fifo_type);
  procedure push(variable fifo_instance : inout fifo_type;
                 value : in std_logic_vector(MAX_WIDTH-1 downto 0));
  procedure pop (variable fifo_instance : inout fifo_type;
                 value : out std_logic_vector(MAX_WIDTH-1 downto 0));


end tb_fifo_pkg;
```


© Copyright 2019 Xilinx

Slide 27-14:

Understanding the Delivery Delay Example (3)

```
package body tb_fifo_pkg is

  procedure push(variable fifo_instance : inout fifo_type;
                 value : std_logic_vector(MAX_WIDTH-1 downto 0)) is
    begin
      if (fifo_instance.volume = MAX_DEPTH) then
         writeNowToScreen("!Warning - FIFO is full and cannot accept any more data!");
      else
        fifo_instance.contents(fifo_instance.write_ptr) := value;
        fifo_instance.write_ptr := (fifo_instance.write_ptr + 1) mod
                                   fifo_instance.depth;

        --manage the volume and flags
        fifo_instance.volume    := fifo_instance.volume + 1;
        fifo_instance.empty     := false;
        if (fifo_instance.volume = MAX_DEPTH) then
          fifo_instance.full := true;
         else
           fifo_instance.full := false;
         end if;
      end if;
    end push;
```


© Copyright 2019 Xilinx

Slide 27-15:

Understanding the Delivery Delay Example (4)

```
procedure pop(variable fifo_instance : inout fifo_type;
              value : out std_logic_vector(MAX_WIDTH-1 downto 0)) is
  begin
    if (fifo_instance.volume = 0) then
      writeNowToScreen("!Warning - FIFO now empty - can't read!");
      value := (others=>'U');
    else
      value := fifo_instance.contents(fifo_instance.read_ptr);
      fifo_instance.read_ptr := (fifo_instance.read_ptr + 1) mod
                                fifo_instance.depth;

      --manage the volume and flags
      fifo_instance.full := false;
      fifo_instance.volume   := fifo_instance.volume - 1;
      if (fifo_instance.volume = 0) then
        fifo_instance.empty := true;
      else
        fifo_instance.empty := false;
      end if;
    end if;
  end pop;

end tb_fifo_pkg;
```

Slide 27-16:

Summary

- Self-checking testbenches provide a text-based means of validating a design
  - Waveform analysis can be time consuming and error prone
  - File output provides post-simulation documentation
- An asynchronous delay mechanism removes many of the problems of aligning known good data with the output of the UUT
- Input data should take two paths to the output—one through the UUT, the other through an equivalent model—before being compared

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Targeting Xilinx FPGAs

Slide 28-1:

2019.1

This module focuses on Xilinx-specific implementation and chip-level optimization.
20 minutes.

Slide 28-2:

Objectives

After completing this module, you will be able to:

- Describe the challenges of using an HDL approach for FPGA designs
- Identify the factors that directly affect FPGA timing and performance
- Identify the trade-offs and guidelines for logic inference and instantiation
- Describe typical synthesis compiler options and their benefits
- List synthesis considerations unique to Xilinx FPGAs

© Copyright 2019 Xilinx

Slide 28-3:

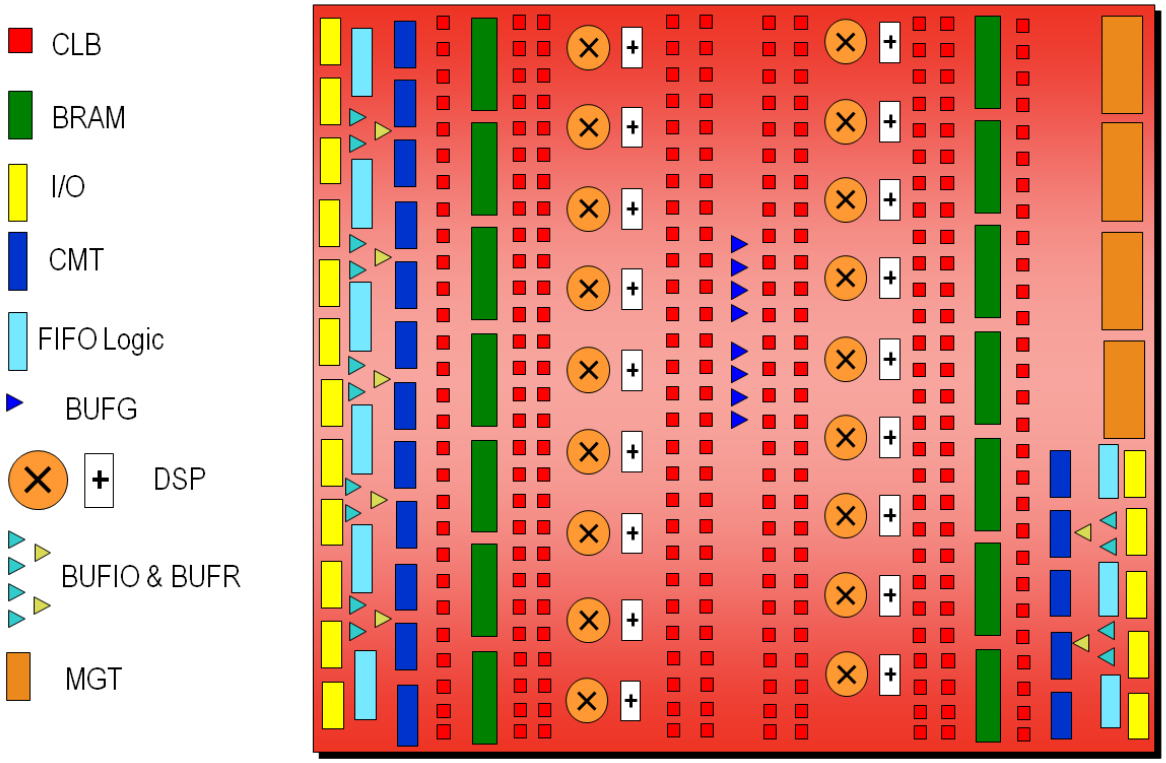
Device-Level Challenges

- Optimization within Xilinx devices via Verilog/VHDL synthesis requires that you address the following areas
  1. Effective use of I/O resources
  2. Using MMCM, BUFGCE, and BUFCTRL
  3. Using SRL, the SelectIO™ technology standard, and MGT
  4. Block RAM and FIFOs
  5. DSP48 and ISERDES/OSERDES
  6. DDR and IDELAY
  7. Accessing other dedicated resources

- Your success in addressing these issues depends largely on your coding style and your choice of synthesis tools

Slide 28-4:

7 Series FPGA Architecture



© Copyright 2019 Xilinx

All Xilinx FPGAs contain the same basic resources. Slices, which are grouped into Configurable Logic Blocks, or CLBs, contain combinatorial logic and register resources.

As you can see from the figure, there are a lot of CLBs on the die. While each device family is offered with a different number of CLBs, determining which density is right for your application often requires you to at least synthesize your design before choosing an appropriate density.

Only with some experience can you properly estimate how many slice resources will be necessary for your design, without at least synthesizing first. The Input/Output Blocks, or IOBs, interface between the FPGA and the outside world.

The IOBs contain interface logic that is designed to translate the internal voltage domain of the FPGA to whatever I/O standard that you are employing outside of the FPGA. This saves you from having to add interface logic outside of the FPGA. The IOBs also contain registers and some specialized resources. While the slice and IOB resources are the most significant and commonly used resources, each device family has a set of dedicated resources, which are specialized.

The 7 series FPGA families include serial Multi-Gigabit Transceivers (MGTs), DSP slices, block RAM, Mixed-Mode Clock Managers (MMCMs), and PLLs.

Finally, the programmable interconnect (which is not shown on this slide) is used to connect the signals between all of the architectural resources.

The implementation tools select an appropriate routing solution that meets your timing needs. Therefore, it is important that you employ timing constraints to communicate your timing objectives to the implementation tools.

FPGA resources can be inferred by all synthesis tools:

- Shift register LUT (SRLC32E)
- F7, F8, and F9 multiplexers
- Carry logic
- Multipliers and counters using the DSP block
- Global clock buffers (BUFG)
- SelectIO (single-ended) standard
- I/O registers (single data rate)
- Input DDR registers

© Copyright 2019 Xilinx

FPGA resources can be inferred by some synthesis tools:

- Memories
- Global clock buffers (BUFGCE, BUGFCTRL)
- Some complex DSP functions

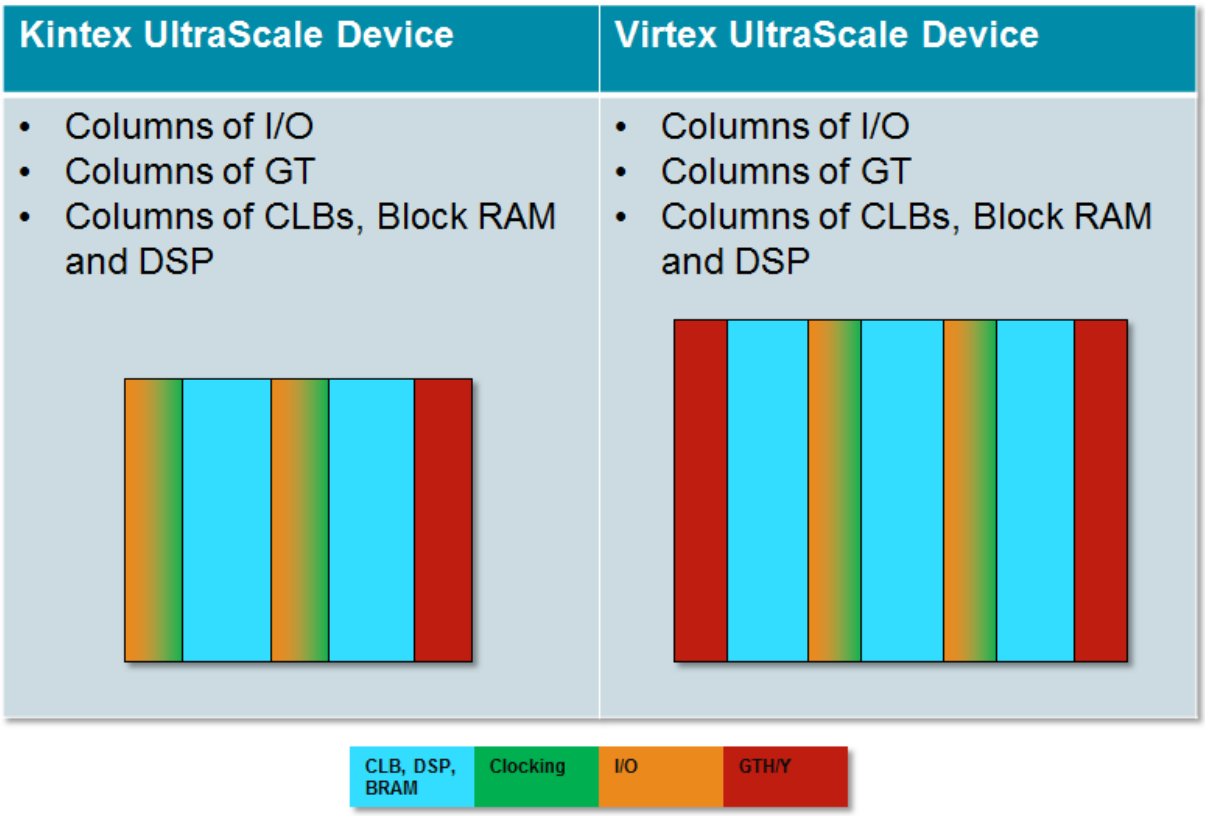
FPGA resources cannot be inferred by any synthesis tools:

- SelectIO (differential) standard
- Output DDR registers
- MMCM/PLL
- Local clock buffers (BUFIO, BUFR, BUFG_LEAF)

Slide 28-5:

UltraScale Architecture Layout

- Side-by-side layout comparison (typical layout)



The UltraScale™ architecture is arranged in columns of resources with the clock management columns tightly coupled with the I/O columns and second generation memory interface logic.

© Copyright 2019 Xilinx

The layouts depicted in this slide represent a typical layout per family. There are instances in which a Kintex® UltraScale FPGA has two columns of GTs and a Virtex® UltraScale FPGA has one column of GTs.

All UltraScale architecture-based FPGAs are divided into clock regions.

Slide 28-6:

Inference vs. Instantiation

- The key to Xilinx optimization is accessing and controlling device-level resources, as well as overall place & route results
  - From an HDL perspective, there are only two means to access any resource

- **Inference**
  - Generic HDL code
    - Synthesis tool decides which vendor library to use
  - Device optimization as per tool ability
  - Maximum portability

- **Instantiation**
  - Create "instance" of
  - Designer references specific vendor macro
  - Maximum device optimization may be achieved
  - Limits portability

© Copyright 2019 Xilinx

Slide 28-7:

Targeting Xilinx Resources

- Some resources must be instantiated as a library primitive or as an IP core from the IP Catalog
  - FIFO, ISERDES, OSERDES, and clock resources

- Certain resources require specific coding
  - DSP block registers have synchronous set/reset only
  - Distributed RAM/ROM and SRL do not have set/reset functionality after configuration

- Synchronous resets are preferred over asynchronous reset
  - Xilinx FPGAs have a configuration reset (GSR) that is used during the configuration stage to bring up the FPGA in a known state
    - GSR is also accessible to designers after configuration via the STARTUP block
    - No need for asynchronous initialization power-on reset

For more information, see the *Vivado Design Suite User Guide: Synthesis* (UG901).

© Copyright 2019 Xilinx

Slide 28-8:

Simple Coding Steps Yield 3x Performance

- Use pipeline stages—more bandwidth
- Use synchronous reset—better system control
- Use Finite State Machine (FSM) optimizations
- Use inferable resources
  - Multiplexer
  - Shift Register LUT (SRL)
  - Block RAM, LUT RAM
  - Cascade DSP
- Avoid high-level constructs (i.e., loops)—may yield slow implementations
- Avoid deeply nested *if-then-else* statements
  - Usually parallel; however, deep nesting can result in priority-encoded logic
- Use *case* statements for large decoding
  - Rather than if-then-else

These are just the most obvious suggestions. For every design, there may be more tricks or other clever things that can improve performance.

Pipelining is the one thing that helps the most, and for most systems today, pipelining is always an option because bandwidth is what defines the system, not the latency. Latency can be important, but if it is, it is usually the latency in a different order of magnitude than the one that is caused by pipelining.

FPGAs have lots of registers, so re-timing and clever use of arithmetic functions can yield tremendous performance. If designers need to balance the latency among different paths in the system, the SRLs can be used to compensate efficiently for delay differences.

© Copyright 2019 Xilinx

Slide 28-9:

When to Code Constraints

- The most intimate time a designer has with the code is at the time of writing
  - Subtleties are known which tend to be forgotten in very little time
- Either
  - Write the XDC while writing code
  - Take careful notes for later
- Note
  - Multi-cycle paths
  - Logic which requires especially tight timing/placement such as anti-meta-stability flip-flops
  - Timing paths which can be ignored

Slide 28-10:

Synchronous Design Techniques

- Minimize the number of clocks in the design
- Make sure that internally created resets are synchronous
- Use only one edge of the clock
- Use edge-triggered flip-flops (avoid latches)
- Cross-clock domains via synchronization circuits
- Register top-level inputs and outputs for fastest performance and increased pin-locking capability
- Register leaf-level outputs
- Use hierarchy to separate functionality and clock domains
- Employ pipelining for critical paths
- Comment your code to draw attention to multicycle paths and critical paths

Do not create clocks internally, except through the MMCM/PLL. If you have more than one clock registering inputs, synchronize all inputs to the fastest clock and use clock enables to slow down the propagation of data. This technique will make your design synchronous.

Internally created asynchronous resets can glitch.

© Copyright 2019 Xilinx

Clock edges drift—so when you use both edges, the reliability of the circuit is reduced. This situation occurs if the clock goes through the MMCM and uses duty cycle correction.

Non D-type flip-flops can confuse timing analysis.

Registering top-level inputs and outputs allows you to meet chip-to-chip requirements more easily. Consider double-registering all inputs and outputs to decouple the internal logic from the I/O layout—enabling early pin locking.

Registering leaf-level outputs allows you to safely preserve hierarchy during synthesis. It also makes it easier to associate critical paths from timing reports back to the code.

Proper use of hierarchy allows you to optimize each level individually, based on the functionality and performance requirements of each level.

Pipelining is an important concept that allows you to achieve higher performance. Pipelining entails more frequent registering of data to decrease the combinatorial delays in each path and, therefore, increase the performance.

Slide 28-11:

Metastability

- Flip-flop output enters a transitory state
  - Neither a valid 0 nor a valid 1
    - Can be interpreted as 0 by some loads and as 1 by others
  - Remains in this state for an unpredictable length of time before settling to a 0 or 1
- Using metastable signals can cause circuit failures
  - Occurrence of failures can only be reduced, not eliminated
- Mean Time Between Failure (MTBF) is exponentially related to the length of time that the flip-flop is given to recover
  - A few extra ns of recovery time can dramatically reduce the chances of a metastable event causing a circuit failure
- The circuits shown in this section allow maximum time for metastable recovery

© Copyright 2019 Xilinx

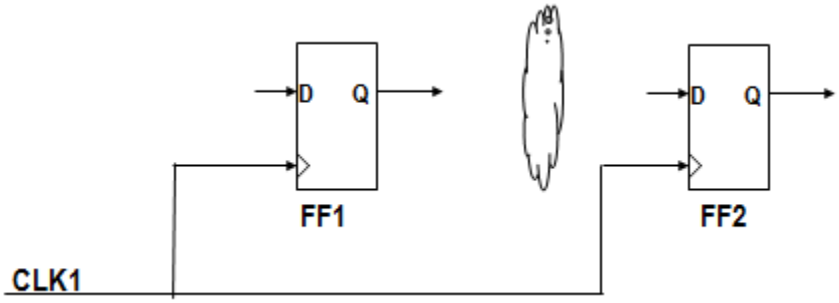
When a signal is at a metastable value, it can be interpreted as a logic 0 by some parts of the circuit and as a logic 1 by other parts. This inconsistency will never be shown during simulation and can be very hard to track down during board-level testing.

When the flip-flop leaves the metastable state, it can go to either a logic 0 or 1. There is no known "correct" state when the flip-flop input changes so close to the clock edge. Still, having an "incorrect" value propagating through your circuit is better than having a metastable value.

<recovery time> = <time before the data is used> - <datapath delay>

Note: The <datapath delay> should include the normal clock_to_q delays and setup times required.

Example: If CLK1 has a period of 10 ns, and the datapath delay is 4 ns, then the recovery time of FF1 is (10-4) = 6 ns.



Slide 28-12:

Timing Closure

- Timing closure techniques
  - Performance baseline procedure
  - Making the implementation tools work harder
  - Automatic and manual ways to improve design timing

- For the fabric to achieve maximum performance, note the following important considerations
  - Do not exceed more than one level of logic; use the existing registers
  - You may need placement constraints to keep functions together

For more information, refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 28-13:

Control Sets

- Controlling signal limitations can limit the grouping of slice resources into CLBs
- Avoid coding for active-low control signals
- No reset is best
- Use synchronous set/reset instead of asynchronous preset/clear
- Consider clock gating to minimize the timing impact of a global reset

In some instances, the increased combinatorial logic can be combined with existing logic or placed in an unused LUT connected to the flip-flop. The overall increase in LUT utilization may, therefore, be small.

When there are two sets and reset control signals are active-low, the software will automatically implement the equivalent logic by using LUT resources.

It is recommended to avoid using the synthesis option to convert asynchronous resets to synchronous. The synchronous SR port can be converted to a flip-flop without SR using a LUT.

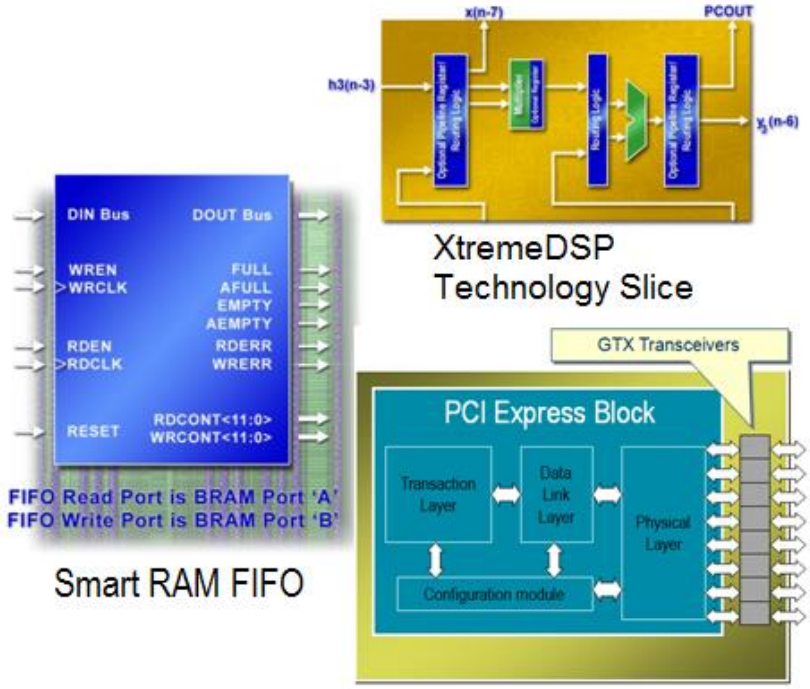
LUTs with common input signals can be packed into the same slice. While SRLs with common input signals can be packed into a single LUT, flip-flops with different control sets cannot be packed into the same half slice.

© Copyright 2019 Xilinx

Slide 28-14:

Use Embedded Blocks

- Embedded block timing is correct by construction
  - Not dependent on programmable routing
- Offers as much as 3x the performance of soft implementations
- Examples
  - FIFO at 600 MHz
  - DSP slices at 741 MHz
  - PCI Express (Gen1, Gen2, and Gen3)



Slide 28-15:

Pipelining for Performance and Power

- Maximum performance is seen when
  - There are six inputs to a logic function
    - This is different than previous architectures due to the 6-LUT
  - Extra caution is taken around multipliers and RAMs
    - Many critical timing paths originate or contain block RAM or DSP48s
      - Functions mapped into DSP48E are properly pipelined
      - Output registers on RAMs
- "Good" pipelining can help improve power
  - DSP48E gets best power when pipelined
    - Not using MREG can add more power
  - Good pipelining reduces “glitch energy”

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.1 Headingשגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 28-16:

State Machine Encoding

- Use enumerated types to define state vectors
  - Most synthesis tools have commands to extract and re-encode state machines described in this way
- Use one-hot encoding for high-performance state machines
  - Uses more registers, but simplifies next-state logic
  - Examine trade-offs: Gray and Johnson encoding styles can also improve performance
  - Refer to the documentation of your synthesis tool to determine how your synthesis tool chooses the default encoding scheme
- Register state machine outputs for higher performance

The advantage of using one-hot encoding in Xilinx FPGAs is that the next-state decoding logic can be simplified to logic equations with four inputs or fewer, which can fit into a single LUT. This maximizes the performance of the state machine.

Many synthesis tools automatically choose one-hot encoding for state machines when you target a Xilinx FPGA.

© Copyright 2019 Xilinx

Slide 28-17:

Why Synthesis Options?

- There are two aspects to any design
  - What does it do?
  - How fast does it do it?

- The code describes what gets done
  - Good coding techniques and synthesis options helps it do it fast enough

- Three aspects of meeting timing
  - Coding efficiency
    - How many LUTs exist between any two flops?
  - Synthesis options
    - Guides the synthesis tools in making choice for the specific design
  - Synthesis attributes
    - Informs the synthesis tools of timing targets to help synthesis tools make good inference decisions

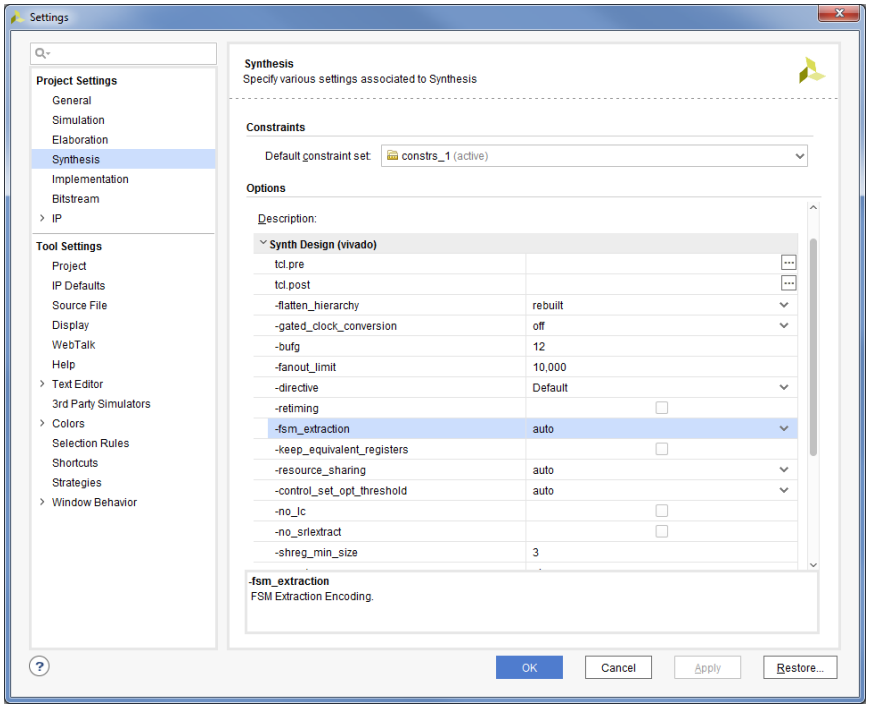
© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 28-18:

Synthesis Settings

- Flatten hierarchy (-flatten_hierarchy)
- Do not set I/O buffers (-no_iobuf)
- Maximum fanout limit (-fanout_limit)
- FSM extraction (-fsm_extraction)
- Synthesis effort level (-directive)



Not every coding option is explored in this section, only the most commonly used ones. These options, like all the others, are applied to all modules in the design. Specific items can be selected via VHDL attributes or synthesis constraints.

- Flatten hierarchy (-flatten_hierarchy)
  - Preserves modular naming and separation throughout the tool flow (rebuilt)
- Do not set I/O buffers (-no_iobuf)
  - Instructs the tool to not infer any input or output buffers
- Maximum fanout limit (-fanout_limit)
  - Fanout is the number of inputs an output must drive
  - Reducing fanout generally improves performance
- FSM extraction (-fsm_extraction)
  - Enable the user to override the automatic settings for FSM coding
- Synthesis effort level (-directive)
  - When specified, this option runs synthesis with different optimizations
  - Values are *Default* and *RunTimeOptimized*, which runs synthesis quickly and with less optimization

© Copyright 2019 Xilinx

Slide 28-19:

Apply Your Knowledge

1) Describe some simple coding techniques to improve performance. What is the benefit provided by each technique?

2) List a few of the options in the synthesis tools that help you increase performance.

3) Describe some unique requirements to obtain optimal results in a 7 series FPGA. Analyze these requirements. Which of these requirements do you believe are applicable to other Xilinx FPGAs?

A

1) Describe some simple coding techniques to improve performance. What is the benefit provided by each technique?
   - Use pipeline stages—more bandwidth
   - Use synchronous reset—better system control
   - Use Finite State Machine (FSM) optimizations
   - Use inferable resources
     - Multiplexer
     - Shift Register LUT (SRL)
     - Block RAM, LUT RAM
     - Cascade DSP
   - Avoid high-level constructs (i.e., loops)—may yield slow implementations
   - Avoid deeply nested *if-then-else* statements
     - Usually parallel; however, deep nesting can result in priority-encoded logic
   - Use *case* statements for large decoding
     - Rather than if-then-else

2) List a few of the options in the synthesis tools that help you increase performance.
   - Flatten hierarchy (-flatten_hierarchy)
   - Do not set I/O buffers (-no_iobuf)
   - Maximum fanout limit (-fanout_limit)

© Copyright 2019 Xilinx

- FSM extraction (-fsm_extraction)
- Synthesis effort level (-directive)

3) Describe some unique requirements to obtain optimal results in a 7 series FPGA. Analyze these requirements. Which of these requirements do you believe are applicable to other Xilinx FPGAs?
   - Use active-high control signals or invert the active-low signal at the top level prior to distributing throughout the design.
   - Pipelining:
     - Even in low-performance designs, this will reduce long, highly buffered, routing and "glitch" energy
     - Use DSP48E and block RAM registers
   - Use dedicated resources to greatly reduce power requirements and increase performance.
   - Only enable block RAMs as needed.
   - Use parameters and generics to pass primitive attributes.
   - Use KEEP_HIERARCHY to improve debug.

Slide 28-20:

Summary

- FPGA-centric synthesis tools use proprietary algorithms to optimize for unique device architecture and resources
- Not all device-level resources can be inferred from the HDL source code and may require instantiation
- Component instantiation is also a viable means to enhance designer productivity and device-level optimization
- For 7 series FPGAs, inverting flip-flop control signals at the global level is recommended

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Course Summary

Slide 29-1:

2019.1

This module provides a summary of the course. 10 minutes.

Slide 29-2:

Objectives Achieved?

- Do you understand
  - The differences between behavioral and structural coding styles
  - How to code for synthesis versus coding for simulation

- Can you now
  - Use scalar and composite data types to represent information
  - Use concurrent and sequential control structure to regulate information flow
  - Implement common VHDL constructs
    - Finite State Machines (FSMs)
    - RAM/ROM data structures
  - Simulate a basic VHDL design
    - Write a VHDL testbench and identify simulation-only constructs
  - Identify and implement coding best practices
    - Optimize VHDL code to target specific silicon resources within the Xilinx FPGA
  - Create and manage designs within the Vivado® Design Suite

© Copyright 2019 Xilinx

Slide 29-3:

Topics

- Synthesis versus simulation environments
- VHDL building blocks
  - Library
  - Entity – Ports, generics
  - Architecture – Naming, associating with an entity
  - Configuration
- Concurrent structures
  - Unconditional assignments
  - Conditional assignments
    - With/sel
    - When
  - Instantiation (instantiation versus inference)
  - Processes
  - Generates
  - Assertions

© Copyright 2019 Xilinx

על הטקסט שברצונך שיופיע כאן.Heading 1שגיאה! השתמש בברטיסיה בית כדי להחיל

Slide 29-4:

Topics

- Processes
  - Variables
  - Process sensitivity list and wait statements
  - Control structures
    - If/then/elsif/else/end if
    - Case
    - Looping
      - Infinite
      - For loop
      - While loop
      - Exit
      - Next
  - Delays

Slide 29-5:

Topics

- Introduction to testbenches
- Attributes
- Functions and procedures
- Package and package body
- Libraries
  - Importing existing – Numeric_Std, TEXTIO
  - Creating custom
- Techniques
  - Finite state machines
  - Creating memory
  - Targeting Xilinx FPGAs

© Copyright 2019 Xilinx

Slide 29-6:

Follow Xilinx

facebook.com/XilinxInc

twitter.com/XilinxInc

youtube.com/XilinxInc

linkedin.com/company/xilinx

plus.google.com/+Xilinx

© Copyright 2019 Xilinx

Capture Your Notes Here

© Copyright 2019 Xilinx

Appendix: Glossary

Appendix: Glossary

aggregate	Basic operation that combines one or more values into a composite value of a record or array type.
alias	Alternate name for an existing object.
architecture	Body associated with an entity declaration used to describe the internal organization or operation of a design entity. An architecture is used to describe the behavior, data-flow, or structure of a design entity.
array	Type, the value of which consists of elements that are all of the same subtype or type. Each element is uniquely distinguished by an index number or a sequence of indexes for a multidimensional array.
assertion statement	Statement that ensures that a specified condition is true, and reports an error if it is not.
attributes	Value, function, type, range, or constant that may be associated with one or more named objects in a description.
bit	Predefined in the standard package as an enumerated data type with two allowable values: "0" and "1."

© Copyright 2019 Xilinx

bit_vector	Predefined in the standard package as a one-dimensional array type with each element being of type bit.
boolean	The boolean type is predefined in the standard package as an enumerated data type with two possible values: false and true.
case	case is used to select one of several alternative sequences of statements for execution; the alternative is chosen based on the value of the selector expression.
component declaration	Component declaration declares a virtual design entity interface that may be used in component instantiation statements.
component instantiation	Component instantiation defines a sub-component of the design entity in which it appears and associates signals or values with the ports of that sub-component, as well as generics.
composite type	Object having multiple elements. This can be either an array or a record.
concatenation	Predefined adding operator for a one-dimensional array type.
configuration declaration	Construct that defines how component instances in a given block are bound to design entities—entities and their associated architectures—in order to describe how those design entities are put together to form a complete design.

© Copyright 2019 Xilinx

constant	Object whose value cannot be changed once defined in the design.
delay	Mechanism that allows introduction of timing parameters for specified systems.
entity	The description of a design in terms of its external environment.
enumeration type	Type whose values are defined by listing (enumerating) them explicitly.
event	Change in the current value of a signal that occurs when the signal is updated as the result of a transaction.
function	Sub-program that is called in the form of an expression and return a value.
generate statement	Mechanism for iterative or conditional elaboration of a portion of a description.
generic	Interface constant, declared in the block header of a block statement component or entity declaration. Generics provide a channel for static information to be passed to the block from its environment. Generics can be updated with each use in the design, unlike constants.
identifier	Names that identify various VHDL named objects.

© Copyright 2019 Xilinx

if statement	Statement that depends on the value of one or more corresponding conditions, selects for execution one or none of the enclosed sequences of statements.
integer	Scalar type whose set of values includes integer numbers within the specified range.
library clause	Defines logical names for design libraries in the host environment.
literal	Value that is directly specified in a description of a design.
loop statement	loop statement
null	Statement that does not perform any action (used primarily for simulation).
operator overloading	Function declaration whose designator is an operator symbol. The same operator may be used for various unique functions, according to the placement and type of formal parameters.
operators	Operators are the means for constructing expressions, along with operands.
package	Unit that groups various declarations, which can then be shared among several designs or design entities.

© Copyright 2019 Xilinx

package body	Contains the bodies of sub-programs and deferred constants declared in the package.
physical type	Numerical scalar representation of some type. It consists of the primary unit and secondary units, which must be multiples of the primary, used to quantify real-world values such as mass, length, or time, for example.
port	Channel for dynamic communication between a block and its environment.
procedure	Subprogram that defines an algorithm for computing values or exhibiting behavior. The procedure call is a statement.
process statement	Process statement defines an independent sequential process representing the behavior of some portion of the design.
range	Specified subset of values of a scalar type.
record	Composite type whose values consist of named elements. The elements may be of different types.
report statement	Statement that displays a message string.
resolution function	Function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal.

© Copyright 2019 Xilinx

resume	The action of a wait statement when the condition that statement is waiting for is satisfied.
return statement	Completes the execution of the function or procedure. Returns result to caller of subprogram.
scalar type	A singular, possibly one-bit wide, value having no elements.
sensitivity list	List of signals to which a process is sensitive. The process will resume execution on any event to a signal in the sensitivity list.
slice	One-dimensional array of a sequence of consecutive elements of another one-dimensional array.
standard package	Predefines a number of types, subtypes, and functions, which are visible to all design units.
std_logic	Nine-valued resolved type defined in the package IEEE Std 1164.
std_logic_1164 package	Defined in IEEE Std 1164, not part of the VHDL standard definition.
std_logic_vector	Predefined, unconstrained array of the type std_logic, as defined in the IEEE Std 164and its associated package.
string	Predefined data type in package standard, an unconstrained, one-dimensional array of type character.

© Copyright 2019 Xilinx

subtype	A limited use of an existing type. A more narrow range of the existing type.
suspend	The stage when the process completes execution and returns to its wait condition.
testbench	Design unit that instantiates another design unit to provide input stimulus and monitoring data outputs, used within a simulation environment for behavioral verification.
type	Set of values and a set of operations.
type conversion	Expression that converts the value of a sub-expression from one type to the designated type of the type conversion.
use clause	Achieves direct visibility with the design unit.
variables	Objects that are declared local to the process and may be updated immediately within the process. Unlike signals, variables cannot be used to carry values to other design units.
vector	Analogous to a bus, a one-dimension array of objects or signals.
wait	Statement used to suspend execution of a process. A sensitivity list is an implied wait condition.

© Copyright 2019 Xilinx

Appendix

Additional Xilinx Courses

For detailed information on any of the following courses, go to www.xilinx.com/training.

Embedded Design

Hardware:

- Zynq UltraScale+ MPSoC for the Hardware Designer
- Introduction to the Zynq SoC Architecture
- Embedded Systems Design
- Advanced Features and Techniques of Embedded Systems Design

Software:

- SDSoC Development Environment and Methodology
- Advanced SDSoC Development Environment and Methodology
- Zynq UltraScale+ MPSoC for the Software Developer
- Embedded Systems Software Design
- Advanced Features and Techniques of Embedded Systems Software Design
- Embedded Design with PetaLinux Tools
- Developing AWS F1 Applications Using the SDAccel Environment
- Accelerating C, C++, OpenCL, and RTL Applications with the SDAccel Environment

System Architect:

- Zynq SoC System Architecture
- Zynq UltraScale+ MPSoC for the System Architect

Machine Learning:

- Developing Xilinx AI Solutions for Edge-based Applications

© Copyright 2019 Xilinx

HLx and SDx Design

HLx:

- C-based Design: High-Level Synthesis with Vivado HLS
- C-based HLS Coding for Hardware Designers
- C-based HLS Coding for Software Designers

SDx:

- Embedded C/C++ SDSoC Development Environment and Methodology

Architecture

FPGAs:

- Designing with the UltraScale Architecture (two day)
- UltraScale Architecture Workshop (one day)
- Designing with the 7 Series Families
- Designing with the Spartan-6 and Virtex-6 Families
- Designing with the Virtex-5 Family

SoCs and MPSoCs:

- Zynq SoC System Architecture
- Zynq UltraScale+ MPSoC for the System Architect

FPGA Design

Vivado Design Suite:

- Designing FPGAs Using the Vivado Design Suite 1
- Designing FPGAs Using the Vivado Design Suite 2
- Designing FPGAs Using the Vivado Design Suite 3
- Designing FPGAs Using the Vivado Design Suite 4
- Xilinx Partial Reconfiguration Tools and Techniques

Vivado Design Suite Adopter:

- UltraFast Design Methodology
- Vivado Design Suite for ISE Software Project Navigator Users
- Vivado Design Suite Advanced XDC and Static Timing Analysis for ISE Software Users

© Copyright 2019 Xilinx

Machine Learning:

- Developing Xilinx AI Solutions for Cloud-based Applications

Connectivity Design

PCI Express Protocol:

- PCIe Protocol Overview
- Designing an Integrated PCI Express System

Transceivers:

- Designing with Xilinx Serial Transceivers

Memory Interface:

- How to Design a High-Speed Memory Interface

Ethernet MAC:

- Designing with Ethernet MAC Controllers

Signal Integrity and Board Design:

- Signal Integrity and Board Design for Xilinx FPGAs

RF:

- Designing with the Zynq UltraScale+ RFSoC
  - This course also includes content for hardware and software engineers

DSP Design

- Essential DSP Implementation Techniques for Xilinx FPGAs
- DSP Design Using System Generator

Languages

SystemVerilog:

- Designing with SystemVerilog
- Verification with SystemVerilog

Verilog:

- Designing with Verilog

VHDL:

- Designing with VHDL
- Advanced VHDL

© Copyright 2019 Xilinx