Designing with SystemVerilog

lang-svdes-2021.1-wkbp-rev1

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Designing with SystemVerilog 2021.1

© Copyright 2021 Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. 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.

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Table of Contents

Course Agenda **1-1**

Objectives (1) 1-1

Objectives (2) 1-2

Course Description 1-2

Day One Agenda 1-3

Day Two Agenda 1-4

Typographical Conventions 1-5

Introduction to SystemVerilog **2-1**

Objectives 2-1

What is SystemVerilog? 2-2

How Is SystemVerilog Different from Verilog? 2-3

Evolution of SystemVerilog 2-4

Why SystemVerilog? 2-5

FPGA Support for SystemVerilog 2-5

Xilinx Support for SystemVerilog 2-6

Apply Your Knowledge 2-6

Summary 2-7

Data Types **3-1**

Objectives 3-1

Data Objects and Data Types 3-2

4-State Data Types 3-3

logic and uwire Type Example 3-4

2-State Data Types 3-5

Bit Data Type 3-6

Integral and Non-Integral Data Types 3-7

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Integral Data Types 3-8
Net Data Types 3-9
Other Data Types 3-10
Apply Your Knowledge 3-11
Apply Your Knowledge 3-11
Summary 3-12

User-Defined and Enumerated Data Types 4-1

Objectives 4-1
User-Defined Data Types 4-2
Applications of User-Defined Data Types 4-3
Enumerated Data Types (1) 4-4
Enumerated Data Types (2) 4-5
Define New Enumerated Types 4-6
Range of Enumerated Values 4-6
One-Hot FSM Coding with Enumerated 4-7
Enumerated Example 4-8
Apply Your Knowledge 4-8
Summary 4-9

Type Casting 5-1

Objectives 5-1
Type Casting 5-2
Types of Type Casting 5-3
Enumerated Arithmetic and Casting 5-4
Apply Your Knowledge 5-4
Summary 5-5

Arrays and Strings 6-1

Objectives 6-1
Definition of Arrays 6-1

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

History of Arrays 6-2
SystemVerilog Enhancements to Verilog Arrays 6-2
Packed Arrays (1) 6-3
Packed Arrays (2) 6-3
Packed Arrays Example (1) 6-4
Packed Arrays Example (2) 6-5
Array Querying System Functions (1) 6-5
Array Querying System Functions (2) 6-6
Dynamic Arrays 6-7
Associative Arrays 6-8
Queues 6-9
Apply Your Knowledge 6-10
Summary 6-11

SystemVerilog Building Blocks 7-1

Objectives 7-1
Design and Verification Building Blocks 7-1
Modules 7-2
Programs 7-3
Interfaces 7-4
Checkers 7-4
Primitives 7-4
Packages 7-5
Config 7-6
Summary 7-6

Structures 8-1

Objectives 8-1
Definition of Structures (1) 8-1
Definition of Structures (2) 8-2

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Unpacked and Packed Structures 8-3
Packed Structures Example 8-4
Unsigned and Signed Packed Structures 8-5
Structure Assignment Patterns 8-6
Structure Assignment Patterns by Type and Default 8-7
Apply Your Knowledge 8-7
Summary 8-7

Unions 9-1

Objectives 9-1
Definition of Unions (1) 9-1
Definition of Unions (2) 9-2
Unpacked and Packed Unions 9-3
Tagged Unions 9-4
Apply Your Knowledge 9-5
Summary 9-5

Additional Operators in SystemVerilog 10-1

Objectives 10-1
Verilog Operators: A Revisit 10-2
Assignment Operators 10-3
Wildcard Equality Operators 10-4
Wildcard Equality Operator Example 10-5
Increment/Decrement Operators 10-5
Inside Operator 10-6
Streaming Operators (pack/unpack) (1) 10-7
Streaming Operators (pack/unpack) (2) 10-8
Packing Using Streaming Operators 10-9
Unpacking Using Streaming Operators 10-10
Operator Precedence in SystemVerilog 10-11

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Literals 10-12
Apply Your Knowledge 10-12
Apply Your Knowledge 10-12
Summary 10-13

Procedural Statements 11-1

Objectives 11-1
always: General-Purpose Procedural Block 11-1
Additional Procedural Blocks in SystemVerilog 11-2
always_comb 11-3
always_comb Example 11-4
Assignment 11-5
always_latch 11-5
always_ff 11-6
Final Procedure 11-7
Apply Your Knowledge 11-7
Summary 11-8

Control Flow Statements 12-1

Objectives 12-1
Looping Constructs in SystemVerilog 12-1
Enhanced for Loops 12-2
The do...while loop 12-3
The foreach Loop (1) 12-4
The foreach Loop (2) 12-5
New Jump Statements in SystemVerilog 12-5
Verilog case Statement 12-6
Priority Case Example 12-6
Priority, Unique0, and Unique Keywords 12-7
Logic for Priority, Unique0, and Unique Keywords 12-8

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Unique Case Example 12-9
Apply Your Knowledge 12-9
Apply Your Knowledge 12-10
Summary 12-11

Functions 13-1

Objectives 13-1
Functions 13-2
Function Example 13-3
SystemVerilog Enhancements to Functions (1) 13-4
SystemVerilog Enhancements to Functions (2) 13-5
Types of Functions 13-5
Static Functions 13-6
Automatic Functions 13-7
Constant Functions (1) 13-8
Constant Functions (2) 13-9
Passing Arguments by Reference 13-10
Passing Arguments by Reference Example 13-11
Apply Your Knowledge 13-11
Apply Your Knowledge 13-12
Summary 13-12

Tasks 14-1

Objectives 14-1
Tasks 14-2
Task Example 14-2
SystemVerilog Enhancements to Tasks 14-3
Types of Tasks 14-3
Static Tasks 14-4
Automatic Tasks 14-4

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Passing Arguments by Reference (1) 14-5
Passing Arguments by Reference (2) 14-5
Apply Your Knowledge 14-6
Summary 14-7

Packages 15-1

Objectives 15-1
Packages 15-2
Referencing Data in Packages 15-3
Package Example 15-4
Compilation Unit Package ($unit) 15-5
Apply Your Knowledge 15-6
Summary 15-6

Interfaces 16-1

Objectives 16-1
Interface 16-2
Verilog Module Ports 16-2
Interfaces Definition 16-3
Interface Declarations 16-4
Interface Declarations with Ports 16-5
Explicitly Named and Generic Interfaces 16-6
Explicitly Named and Generic Interfaces Example 16-6
Interface Modports: The Concept 16-7
Interface Modports 16-7
Using Modports 16-8
Interface Methods 16-9
Importing Interface Methods 16-10
Calling Imported Methods 16-11
Exporting Interface Methods 16-12

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Exported Interface Methods Example 16-13
Reconfigurable Interfaces 16-14
Apply Your Knowledge 16-14
Apply Your Knowledge 16-15
Apply Your Knowledge 16-16
Summary 16-16

Targeting Xilinx FPGAs 17-1

Objectives 17-1
Device-Level Challenges 17-2
7 Series FPGA Architecture 17-2
UltraScale Architecture Layout 17-4
Inference vs. Instantiation 17-5
Targeting Xilinx Resources 17-6
Simple Coding Steps Yield 3x Performance 17-7
When to Code Constraints 17-8
Synchronous Design Techniques 17-8
Metastability 17-9
Timing Closure 17-10
Control Sets 17-11
Use Embedded Blocks 17-12
Pipelining for Performance and Power 17-12
State Machine Encoding 17-13
Why Synthesis Options? 17-14
Synthesis Settings 17-15
Apply Your Knowledge 17-16
Summary 17-17

Appendix 18-1

Additional Xilinx Courses 18-1

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Course Agenda

Slide 1-1:

2021.1

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

Slide 1-2:

Objectives (1)

After completing this course, you will be able to:

- Describe the features and benefits of using SystemVerilog for RTL design
- Identify the new data types supported in SystemVerilog
- Use an enumerated data type for coding a finite state machine (FSM)
- Explain how to use structures, unions, and arrays
- Describe the new procedural blocks and analyze the affected synthesis results
- Define the enhancements and ability to reuse tasks, functions, and packages

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 1-3:

Objectives (2)

After completing this course, you will be able to:

- Identify how to simplify module definitions and instantiations using interfaces
- Examine how to efficiently code in SystemVerilog for FPGA design simulation and synthesis
- Target and optimize Xilinx FPGAs by using SystemVerilog
- Simulate SystemVerilog modules using the Vivado® simulator tool and/or Questa Sim
- Synthesize and analyze SystemVerilog designs using the Vivado® Design Suite
- Download a complete SystemVerilog design to an evaluation board

Slide 1-4:

Course Description

- Prerequisites
  - RTL coding using Verilog HDL
  - Basic digital design concepts
    - Flip-flops and logic gates
    - Basic understanding of synchronous design
- Course length
  - Two days

This course does not include a Verilog introduction. This course is intended for existing Verilog users.

The course briefly explains the improvements from Verilog and how to utilize these improvements. It does not provide detailed explanations of the issues Verilog users have faced and which SystemVerilog solves.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

For example, it describes new features such as user-defined data types, interfaces (which enable a group of signals to be represented as a single port), packages (a declaration space which can be shared by several building blocks, like in VHDL), etc. Each of these features is described in detail along with examples illustrating how to use them.

VHDL users may not understand the improvements in SystemVerilog compared to Verilog because they are not issues in VHDL.

It should be noted that this course covers only the hardware modeling and design aspects of SystemVerilog. It does not cover SystemVerilog verification constructs such as classes, assertions, etc.

Slide 1-5:

Day One Agenda

- Introduction to SystemVerilog {Lecture} - 15 min.
- Data Types {Lecture, Lab, Demo} - 100 min.
- User-Defined and Enumerated Data Types {Lecture} - 15 min.
- Type Casting {Lecture} - 10 min.
- Arrays and Strings {Lecture} - 20 min.
- SystemVerilog Building Blocks {Lecture} - 15 min.
- Structures {Lecture, Lab} - 45 min.
- Unions {Lecture, Lab} - 40 min.
- Additional Operators in SystemVerilog {Lecture} - 20 min.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 1-6:

Day Two Agenda

- Procedural Statements {Lecture, Lab} - 75 min.
- Control Flow Statements {Lecture} - 20 min.
- Functions {Lecture} - 20 min.
- Tasks {Lecture} - 15 min.
- Packages {Lecture, Lab} - 55 min.
- Interfaces {Lecture} - 25 min.
- Targeting Xilinx FPGAs {Lecture, Lab} - 80 min.

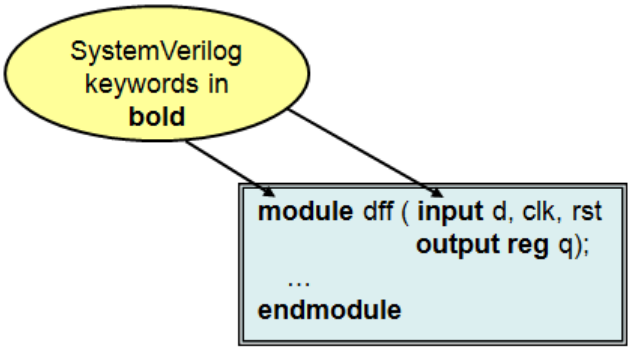
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 1-7:

Typographical Conventions

- SystemVerilog keywords in code examples will be marked as bold



- SystemVerilog keywords in the body of slides will be highlighted in italics
  - For example, SystemVerilog supports all Verilog variable types such as *reg* and *integer*

- SystemVerilog constructs are mentioned as synthesizable or not synthesizable only with respect to the Vivado® synthesis feature
  - For example, *always_ff* procedural blocks are synthesizable (i.e., the Vivado synthesis feature supports *always_ff* for synthesis)

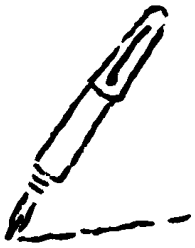
- The Vivado simulator feature supports the subset of SystemVerilog RTL that can be synthesized
  - For complete information refer to the *Vivado Design Suite Logic Simulation User Guide* (UG900)

All SystemVerilog constructs (covered in this course) supported by the Vivado synthesis tool are supported by the Vivado simulator except tagged unions (covered in the "Structures, Unions and Arrays" module) and $cast operator (covered in the "Date Types" module).

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Introduction to SystemVerilog

Slide 2-1:

2021.1

This module provides an introduction to the SystemVerilog language. 15 minutes.

Slide 2-2:

Objectives

After completing this module, you will be able to:

- Define SystemVerilog
- Explain the differences between SystemVerilog and Verilog
- Identify the motivation for using SystemVerilog
- Discuss the history and evolution of SystemVerilog
- Describe how SystemVerilog is supported by FPGAs

© Copyright 2021 Xilinx

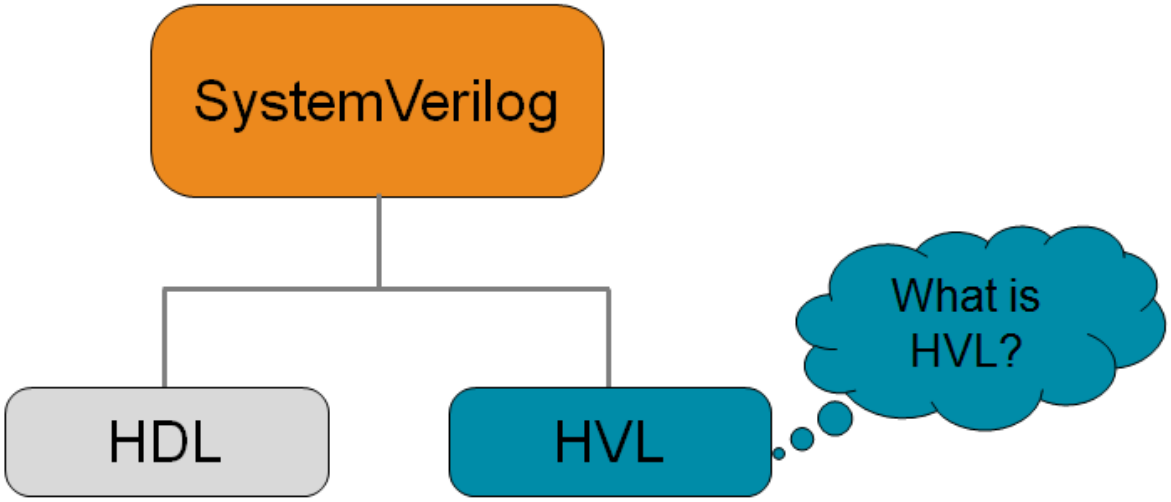
FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 2-3:

What is SystemVerilog?

- SystemVerilog is an enhancement of Verilog
  - It combines the features of
    - Verilog HDL (hardware descriptive languages)
    - Verification capabilities of HVLs (hardware verification languages) like Vera and Specman

- Provides a single platform for both design and verification
  - Industry's first hardware description and verification language (HDVL)



Hardware verification language (HVL) has constructs that allow the design to be verified extensively

- Generating constrained random stimulus
- Functional coverage provides the feedback
- Assertions to check for protocol violations
- Multithreading and interprocess communication
- Higher level abstraction (object-oriented programming)

The verification constructs of SystemVerilog are covered in detail in the Verification with SystemVerilog course.

© Copyright 2021 Xilinx

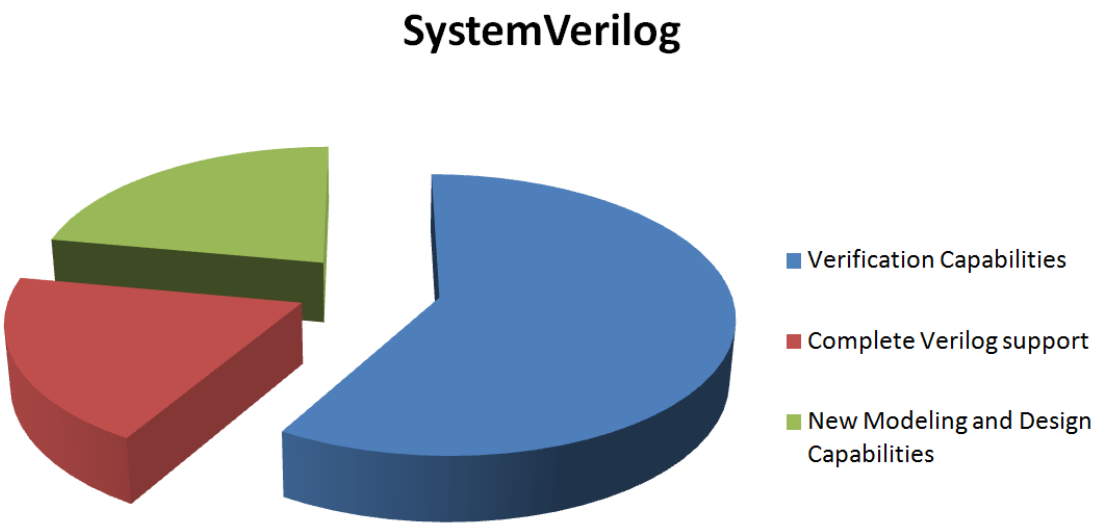
FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 2-4:

How Is SystemVerilog Different from Verilog?

- SystemVerilog extensions to Verilog can be generalized into two primary categories
  - Design enhancements addressing hardware modeling, in terms of abstraction levels and overall efficiency
  - Verification enhancements for writing efficient, race-free testbenches for large complex designs

- The biggest strength of SystemVerilog lies in supporting extensive object-oriented programming techniques (not supported by Verilog) for verification capabilities



The feature set of SystemVerilog can be divided into two distinct elements:

- SystemVerilog for RTL design is an extension of Verilog-2005; all features of Verilog plus additional modeling and design features are available in SystemVerilog
- SystemVerilog for verification uses extensive object-oriented programming techniques and is more closely related to C++ and Java than to Verilog

Object Oriented Programming (OOP) is based on the concept that a program is built using basic building blocks called objects. Objects are self-contained snippets of code and data which can perform a set of activities including sending and receiving messages from other objects. This modular nature of OOP provides a powerful model for the verification and debugging of software.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

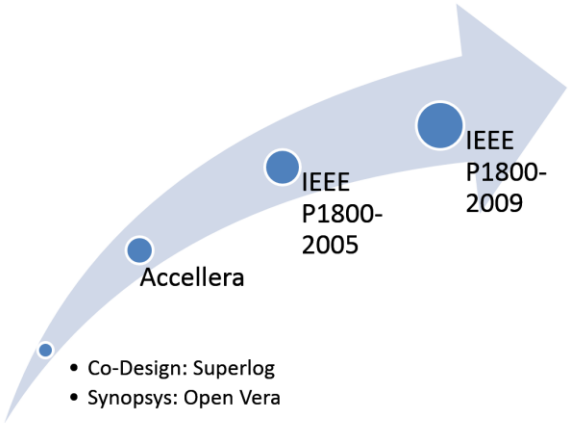
With regard to hardware verification, OOP enables:

- Partitioning complex system level verification environment into discrete objects
- Reusable solutions which can be configured and controlled across designs/projects
- Consistency and best practices for debugging

Slide 2-5:

Evolution of SystemVerilog

- Co-design automation developed Superlog
  - Donated to Accellera in 2002
- Synopsys donated Open Vera to Accellera
- In 2005, SystemVerilog was adopted as P1800 standard by IEEE
  - First released as P1800-2005 standard
  - Subsequent release as P1800-2009
- P1800-2012
  - Improvement of class (Java-style multiple inheritance for class)
  - Covergroup with multiple constraint
  - Net type with resolution function



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 2-6:

Why SystemVerilog?

- Primary motivation
  - System-level design and verification

- Features
  - System-level modeling
    - Higher level abstraction
    - Performance
    - Code reuse (packages)
  - Unification
    - Verilog, VHDL, SystemC
    - Vera
  - Verification
    - Powerful testbenches
    - Permits formal verification
    - Ease of writing complex verification checks
  - Removal of race and other annoyances in design and testbench
  - Synthesis friendly model of design
  - Backward compatibility with Verilog

Besides Verilog, SystemVerilog also supports VHDL features such as packages, configurations, etc. and SystemC and Vera features such as class, etc.

Slide 2-7:

FPGA Support for SystemVerilog

- Why is it important for FPGAs to support SystemVerilog?
  - With FPGAs handling increasingly complex designs like SoCs, SystemVerilog is an attractive consideration for design and verification

  - Given that ASIC prototyping is a popular application of FPGAs, the same SystemVerilog code can be used in both ASIC and FPGA designs

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 2-8:

Xilinx Support for SystemVerilog

- The Vivado® synthesis feature supports SystemVerilog for 7 series and later devices
- Vivado simulator supports SystemVerilog synthesis constructs (for design simulations) and SystemVerilog Testbench features (for verification simulations)
  - *UG900 - Vivado Design Suite, Logic Simulation User Guide* lists the SystemVerilog constructs supported by the Vivado simulator

Slide 2-9:

Apply Your Knowledge

1) How is SystemVerilog different from Verilog?
2) Why is SystemVerilog gaining more significance with FPGAs?

A

1) How is SystemVerilog different from Verilog?

   SystemVerilog is a superset of Verilog.

   The SystemVerilog extensions to Verilog can be generalized to two primary categories:
   - Enhancements in hardware modeling, both in terms of overall efficiency and abstraction levels
   - Introduction of object-oriented-programming-like verification constructs for writing efficient testbenches for large, complex designs

2) Why is SystemVerilog gaining more significance with FPGAs?

   Two important reasons for why SystemVerilog is gaining more significance in FPGAs:
   - With FPGAs handling increasingly complex designs like ASICs, SystemVerilog is an attractive consideration for design and verification
   - Given that ASIC prototyping is a popular application of FPGAs, the same SystemVerilog code can be used in both ASIC and FPGA designs

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 2-10:

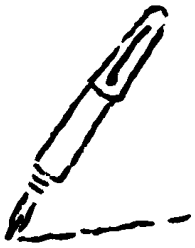
Summary

- SystemVerilog is a language created to provide a single platform for both design and verification
- It is a superset of Verilog HDL and provides several advantages with regard to hardware modeling and simulation which Verilog does not support
- SystemVerilog supports extensive object-oriented programming techniques for verification capabilities
- It was originally developed by Co-Design Automation in 2002 and was called Superlog
- The Vivado synthesis tool supports SystemVerilog
- The Vivado simulation tool supports SystemVerilog synthesis constructs and SystemVerilog Testbench features

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Data Types

Slide 3-1:

2021.1

This module describes the data types supported by SystemVerilog. 20 minutes.

Slide 3-2:

Objectives

After completing this module, you will be able to:

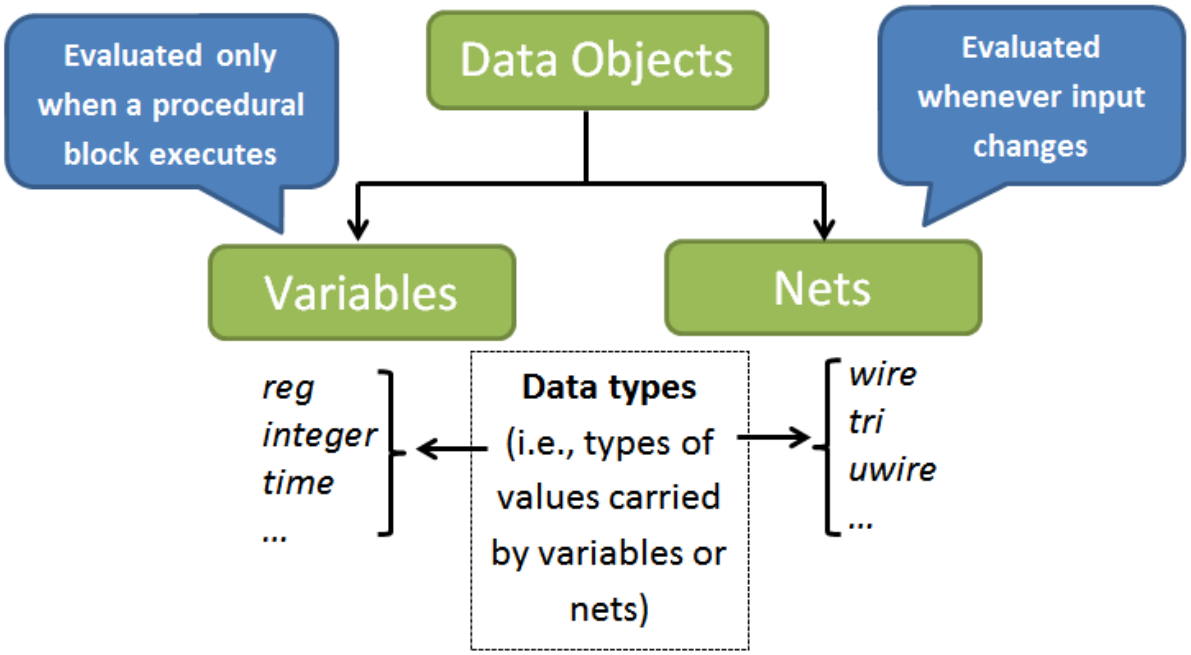
- Define data types
- Describe 4-state and 2-state data types
- Differentiate between integral and non-integral data types
- List the data types supported by SystemVerilog
- Identify the data types that are specific to SystemVerilog and not supported by Verilog

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-3:

Data Objects and Data Types



- SystemVerilog supports all variable data types supported by Verilog, such as *reg* and *integer*, plus new variable data types, such as *byte* and *int*

- Besides the net types supported by Verilog, SystemVerilog does not include any additional net type

There are two main data objects used in SystemVerilog for hardware modeling: *nets* and *variables*

The key difference is how they are updated in simulation:

- *net*: Continually updated, evaluated, and executed whenever inputs change
- *variable*: Evaluated only when a procedural block executes

The values carried by nets and variables are classified as data types.

VHDL users: A variable in Verilog/SystemVerilog is similar to a variable in VHDL except that in Verilog/SystemVerilog a variable can carry values out of a procedural block.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-4:

4-State Data Types

- A 4-state data type indicates that the variable or net can take one of four logic values: 0, 1, x and z
- In Verilog, all nets and variables use 4-state values
- SystemVerilog includes an additional 4-state data type called *logic*

```
logic [63:0] address; // a 64-bit wide variable
```

  - When the *logic* keyword is used by itself, a variable is implied

```
var logic [63:0] address; // keyword 'var' is optional
```

  - Declared as net type with explicit keyword 'wire'

```
wire logic [63:0] address; // a 64-bit wide net
```

  - *logic* cannot be used for multiple driven nets

VHDL users: Unlike the *std_logic* data type in VHDL, which uses the nine-value logic system, data types in Verilog/SystemVerilog can use only four logic values.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-5:

logic and uwire Type Example

- Logic data type combines properties of reg and uwire

```
logic [ 3:0] uwire_like_l4;
logic [ 3:0] reg_like_l4;

assign uwire_like_l4 = 4'bzx01;
// assign uwire_like_l4 = 4'b11zz; - illegal: multiple drivers

initial begin
   reg_like_l4 = 4'b10xz;
end

initial begin
   #1;
   $displayh("uwire_like_l4 = %b", uwire_like_l4);
   $displayh("reg_like_l4 = %b", reg_like_l4);
   $finish;
end
```

```
uwire_like_l4 = zx01
reg_like_l4 = 10xz
```

Note that uwire_like_l4 is a variable that can be assigned a value using an assign statement (similar to a net). However, such a variable can have only one driver.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-6:

2-State Data Types

- A 2-state data type indicates that the variable can be either 0 or 1
- 2-state data types: *int*, *shortint*, *longint*, *bit* and *byte*
- 2-state data types are used when X and Z values are not needed: for example, testbench variable or *for* loop variables
- Advantages of using 2-state data types
  - Faster simulation
  - Lesser memory utilization
  - Works better when interfacing with foreign languages (2-state variables can be exchanged over the Direct Programming Interface [DPI])
- When a 4-state data type is converted to 2-state, the X and Z are converted to 0

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-7:

Bit Data Type

- Bit data type adopts logic data type for two-state variables

```
bit [ 3:0] uwire_like_b4;
bit [ 3:0] reg_like_b4;

assign uwire_like_b4 = 4'b0001;
// assign uwire_like_l4 = 4'b11zz; - illegal: multiple drivers

initial begin
   reg_like_b4 = 4'b1000;
end

initial begin
   #1;
   $displayh("uwire_like_b4 = %b", uwire_like_b4);
   $displayh("reg_like_b4 = %b", reg_like_b4);
   $finish;
end
```

```
uwire_like_b4 = 0001
reg_like_b4 = 1000
```

Note that *uwire_like_b4* is a variable that can be assigned a value using an *assign* statement (similar to a net). However, such a variable can have only one driver.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-8:

Integral and Non-Integral Data Types

- Integral data types refer to the data types which hold a value that can be represented as a vector or a contiguous set of bits

- Value of integral data types are stored as contiguous bits in the memory while non-integral types are not stored as contiguous bits

Integral Data Types	Non-Integral Data Types
shortint	real
int	realtime
longint	shortreal
byte	string
bit	void
logic	user-defined
reg	class
integer	event
time	chandle
enumerated	

There is no defined way of how non-integral data types are stored in memory. Every simulator may store non-integral data types differently in the memory.

structs and *unions* do not have a contiguous structure by default. Hence, they are considered as non-integral data types. However, *structs* and *unions* can be defined as "packed", meaning all members of the *struct* or *union* can be stored as contiguous bits in which case they become integral.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-9:

Integral Data Types

- Integral data types refer to the data types which hold a value that can be represented as a vector or a contiguous set of bits

Type Name	2-State/4-State	Signed/Unsigned	Number of Bits	SystemVerilog/Verilog
shortint	2-state	Signed	16	SystemVerilog
int	2-state	Signed	32	SystemVerilog
longint	2-state	Signed	64	SystemVerilog
byte	2-state	Signed	8	SystemVerilog
bit	2-state	Unsigned	User-defined	SystemVerilog
logic	4-state	Unsigned	User-defined	SystemVerilog
reg	4-state	Unsigned	User-defined	Verilog
integer	4-state	Signed	32	Verilog
time	4-state	Unsigned	64	Verilog

The *byte* type can be used to hold an ASCII character.

The difference between *int* and *integer* is that *int* is a 2-state type and *integer* is a 4-state type.

All integral data types are synthesizable.

The default value of shortint, int, longint, byte, bit is zero and logic is 'x'.

The signed or unsigned nature of the operands influence the result of the operation performed. The general rule is as follows:

- If all the operands are signed, the operation is signed.
- If any of the operands are unsigned, the operation is unsigned (a common mistake that one needs to watch out for).

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Example:

```
integer    f = 10 ;          integer    g = -5 ;          reg [ 15:0 ] m = 10 ;
assign out1 =    f/g       // yields    -2        --signed integer math
assign out2 = m/g          // yields      0          --unsigned integer math*
```

*The last example is evaluated as 1010 / 1111_1111_1111_1111_1111_1111_1111_1011 = approximately 0

Slide 3-10:

Net Data Types

- Net is a structural entity that connects a driver (or multiple drivers) to a receiver
- Net is continually updated, evaluated, and executed whenever inputs change
- SystemVerilog supports all Verilog net data types and does not include any new net data type
- Net data types supported by Verilog and SystemVerilog

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	trireg	uwire

VHDL users: The *uwire* type is similar to an unresolved type in VHDL. It can only have a single driver.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 3-11:

Other Data Types

Type Name	Description	SystemVerilog/Verilog	Synthesizable*
real	Similar to *double* in C	Verilog	Yes**
realtime	Similar to *real,* used for representing simulation time	Verilog	No
shortreal	Similar to *float* in C	SystemVerilog	Yes
string	Holds a collection of characters	SystemVerilog	No
void	Represents nonexistent data, used to check if a function returns a value	SystemVerilog	Yes (for functions)
user-defined	New type created by user using *typedef* keyword	SystemVerilog	Yes
enumerated	For declaring variables that have a specific list of defined values	SystemVerilog	Yes
class	Provides handle to a class object	SystemVerilog	No
event	Provides handle to an event object	SystemVerilog	No
chandle	Stores pointers passed using DPI, for interfacing with other languages	SystemVerilog	No

* Synthesizable or not synthesizable is with respect to the Vivado® synthesis feature.

** Only the constant reals are synthesizable.

There is no defined way of how non-integral data types are stored in memory. Every simulator may store non-integral data types differently in memory.

Structures and unions do not have a contiguous structure by default; hence, they are considered as non-integral data types. However, structures and unions can be defined as "packed", meaning all members of the structure or union can be stored as contiguous bits, in which case they become integral.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Arrays of strings, real, and short real can also be defined.

String[5] names; // An array of 5 string elements

real[32] area; // An array named area with 32 real elements

Strings, user-defined and enumerated data types are covered in detail in other modules.

Slide 3-12:

Apply Your Knowledge

1) What are the advantages of using 2-state variables over 4-state variables?

2) What happens when a 4-state variable is assigned to a 2-state variable?

A

1) What are the advantages of using 2-state variables over 4-state variables?
   - Faster simulation
   - Lesser memory utilization
   - Works better when interfacing with foreign languages (2-state variables can be exchanged over the Direct Programming Interface [DPI])

2) What happens when a 4-state variable is assigned to a 2-state variable?

   The 4-state variable value gets converted to a 2-state value with X and Z converted to 0.

Slide 3-13:

Apply Your Knowledge

1) What are integral data types? Name some of the integral data types in SystemVerilog

2) What is the difference between the *reg* and *logic* data types?

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

1) What are integral data types? Name some of the integral data types in SystemVerilog.

Integral data types refer to the data types that hold a value that can be represented as a vector or a contiguous set of bits. All integer data types (*shortint*, *int*, *longint*, *byte*, *bit*, *logic*, *reg*, *integer*, *time*) plus enumerated (*enum*) data types are integral.

2) What is the difference between the *reg* and *logic* data types?

Both *reg* and *logic* are quite similar. The difference is that *reg* is a variable type and cannot be used for a net while *logic* can be used for both variables and nets.

Also *reg* is a misnomer because reg _in_ hardware means register and all *reg* variables do not infer a register. *logic* is perceived as a better name.

Slide 3-14:

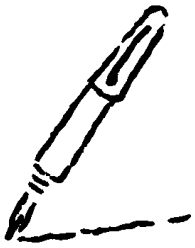
Summary

- Values carried by nets and variables are classified as data types
- SystemVerilog extends Verilog to support new data types, including a category called 2-state data type
- Data types that hold a value which is stored as contiguous bits in memory are called integral data types
- SystemVerilog supports all Verilog net types and does not include any new net data types
- Users can extend the language by creating their own user-defined and enumerated data types

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

User-Defined and Enumerated Data Types

Slide 4-1:

2021.1

This module describes user-defined and enumerated data types supported by SystemVerilog. 15 minutes.

Slide 4-2:

Objectives

After completing this module, you will be able to:

- Create user-defined variables in SystemVerilog
- Describe some of the popular uses of user-defined data types
- Define enumerated data types
- Use enumerated data types to code finite state machines

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-3:

User-Defined Data Types

- SystemVerilog allows users to create new variables or net types using existing types
- User-defined types
  - Allow modeling complex architectures at a more abstract level
  - Result in fewer lines of code and make the code easy to read
  - Are synthesizable
- Keyword *typedef* is used to create a user-defined type

```
`ifdef FOUR_VALUE
   typedef logic data_typ;
`else
   typedef  bit data_typ;
`endif

module fifo (
   input  clock,
   input  reset,
   input  wen,
   input data_typ[7:0] wdata,
   input  ren,
   output  data_typ[7:0] rdata
);

// ..................

endmodule
```

In the above example, if the text macro FOUR_VALUE is defined, then the variables *wdata* and *rdata* will be of type *logic*. Otherwise, they will be a bit type. Note that the example above is for illustration only.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-4:

Applications of User-Defined Data Types

- Some of the applications of user-defined variables include
  - Testbenches where variables of a specific type and length are widely used
  - Declaring the index in arrays which model large memories with multi-gigabyte address range
    - Such arrays use widely separated index values that can be 8-bit, 16-bit, 32-bit, or 64-bit values (or addresses)
  - Creating a 'package' of user-defined types and use them across projects and designs without defining them repeatedly

SystemVerilog also allows defining user-defined arrays, structures, and unions using the keyword *typedef*.

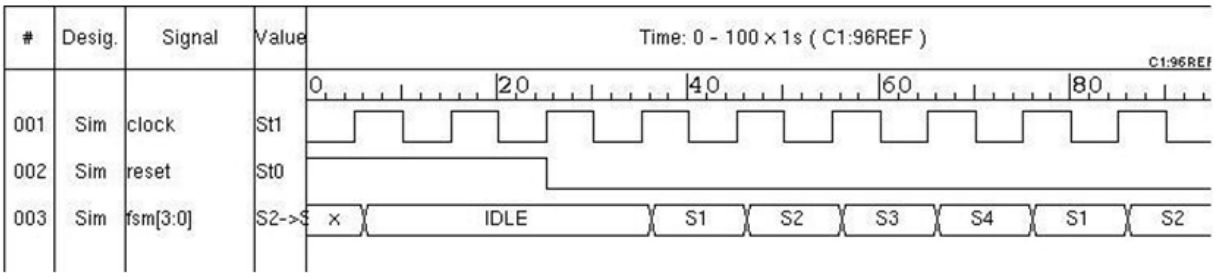
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-5:

Enumerated Data Types (1)

- Enumerated data types provide a means to declare an abstract variable that can have a specific list of valid values




```
enum logic[ 3:0] {IDLE=0, S1=1, S2=2, S3=4, S4=8} fsm;

always @(posedge clock)
   if (reset)
      fsm <= #1 IDLE;
   else
      case (fsm)
         IDLE: fsm <= #1 S1;
         S1: fsm <= #1 S2;
         S2: fsm <= #1 S3;
         S3: fsm <= #1 S4;
         S4: fsm <= #1 S1;
      endcase
```

VHDL users: The enumerated data type is borrowed from VHDL. Verilog does not support the enumerated type.

An enumerated object has a base type which is specified during declaration. The base type in the above example is *logic*. In the absence of a data base type declaration, the default data base type shall be *int*. The labels used in the enumerated type must be unique in their scope. Assignments to an enumerated net or variable are strongly typed.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-6:

Enumerated Data Types (2)

- If no explicit values are set to enumerated, default values are used

```
enum {RED=1, GREEN=2, BLUE=4} color;

initial begin
   #1 $display("color = %0d", color);
   color = RED;
   #1 $display("color = %0d", color);
   color = GREEN;
   #1 $display("color = %0d", color);
   color = BLUE;
   #1 $display("color = %0d", color);
end
```

```
color = 0
color = 1
color = 2
color = 4
```

```
enum {RED, GREEN, BLUE} color;

initial begin
   #1 $display("color = %0d", color);
   color = GREEN;
   #1 $display("color = %0d", color);
   color = BLUE;
   #1 $display("color = %0d", color);
end
```

```
color = 0
color = 1
color = 2
```

- Note that a compilation error will be issued if the enum element immediately after an element holding x value doesn't have a value assigned

In the above example, the default value of enum is changed to 1.

The enum *color* is not assigned any base type, so the base type *int* is used for enum *color*. The default value of an *int* variable is 0, so the default value of *color* is 0. In the next step, *color* is assigned RED. Hence, the value of *color* becomes 1. Then GREEN becomes 2 and again the BLUE value is set to 4.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-7:

Define New Enumerated Types

Define new enumerated type

```
typedef enum {Red, Green, Blue} Colors;
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;
Colors c;
Week day;

initial begin
   c = Blue;
   day = Th;
end
```

Declare variables of new enumerated type

Slide 4-8:

Range of Enumerated Values

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

Define new enumerated type

```
typedef enum bit[3:0]  {IDLE, GO, WORK[12], STOP, DONE} fsm_data_typ;

module enum_test;
   fsm_data_typ fsm;

   always #5 fsm <= fsm.next();

   initial $monitor("fsm now is to %02d {%s}", fsm, fsm.name);

endmodule
```

Enum range

Declare variables of new enumerated type

Specialized methods to access values of enum type variables

```
fsm now is to 00 {IDLE}
fsm now is to 01 {GO}
fsm now is to 02 {WORK0}
fsm now is to 03 {WORK1}
fsm now is to 04 {WORK2}
fsm now is to 05 {WORK3}
fsm now is to 06 {WORK4}
fsm now is to 07 {WORK5}
fsm now is to 08 {WORK6}
fsm now is to 09 {WORK7}
fsm now is to 10 {WORK8}
fsm now is to 11 {WORK9}
fsm now is to 12 {WORK10}
fsm now is to 13 {WORK11}
fsm now is to 14 {STOP}
fsm now is to 15 {DONE}
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

The *next()* method wraps to the start of the enumeration when the end of the enumeration is reached.

The specialized methods supported by the enumerated type are:

- *first()* – returns the value of the first member
- *last()* – returns the value of the last member
- *next()* – returns the next value after the current value of the given enum variable
- *prev()* – returns the previous value before the current value of the given enum variable
- *num()* – returns the number of members
- *name()* – returns the string representation of the given enumeration value

Slide 4-9:

One-Hot FSM Coding with Enumerated

```
typedef enum reg[8:0] {H1_1= 1, H1_2= 2, H1_3= 4, H1_4=  8, H1_5=16,
                       H1_6=32, H1_7=64, H1_8=128, H1_9=256}
                       OneHot_data_typ;

OneHot_data_typ fsm_fwd, fsm_back;

always @(posedge clock)
   if (reset)
      fsm_fwd <= #1 H1_1;
   else
      fsm_fwd <= #1 fsm_fwd.next;

always @(posedge clock)
   if (reset)
      fsm_back <= #1 H1_9;
   else
      fsm_back <= #1 fsm_back.prev;

assign leds = fsm_fwd | fsm_back;
```

LEDS output

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

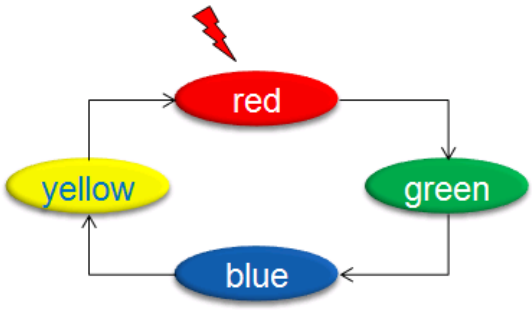
Slide 4-10:

Enumerated Example

```
enum { red, green, blue, yellow } c;
initial begin
   c = c.first;
   forever begin
      $display( "%s : %d", c.name, c );
      if( c == c.last ) break;
         c = c.next;
   end
end
```

c = red

```
red:  0
green:  1
blue:  2
yellow:  3
```



Slide 4-11:

Apply Your Knowledge

1) Why are enumerated data types suitable for modeling finite state machines (FSMs)?

2) What is the keyword used to declare a user-defined data type?

A

1) Why are enumerated data types suitable for modeling finite state machines (FSMs)?

Enumerated data types provide a means for declaring a variable that can have a specific list of valid values. Hence, the enumerated data type can be used to declare a variable that can hold any of the state names.

For example: *enum {IDLE, RED, AMBER, GREEN} current_state;*

This makes the FSM easier to code and understand.

2) What is the keyword used to declare a user-defined data type?

typedef

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 4-12:

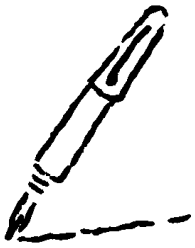
Summary

- SystemVerilog allows users to extend the language by creating their own user-defined data types
- Keyword *typedef* is used to create a user-defined type
- Enumerated data types provide a means to declare an abstract variable that can have a specific list of valid values
- Enumerated data types make hardware modeling more abstract and coding easier

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Type Casting

Slide 5-1:

2021.1

This module describes type casting in SystemVerilog. 10 minutes.

Slide 5-2:

Objectives

After completing this module, you will be able to:

- Describe type casting in SystemVerilog
- Explain the two types of type casting
- Convert the data type of a variable
- Assign values to variables that might not ordinarily be valid because of a differing data type

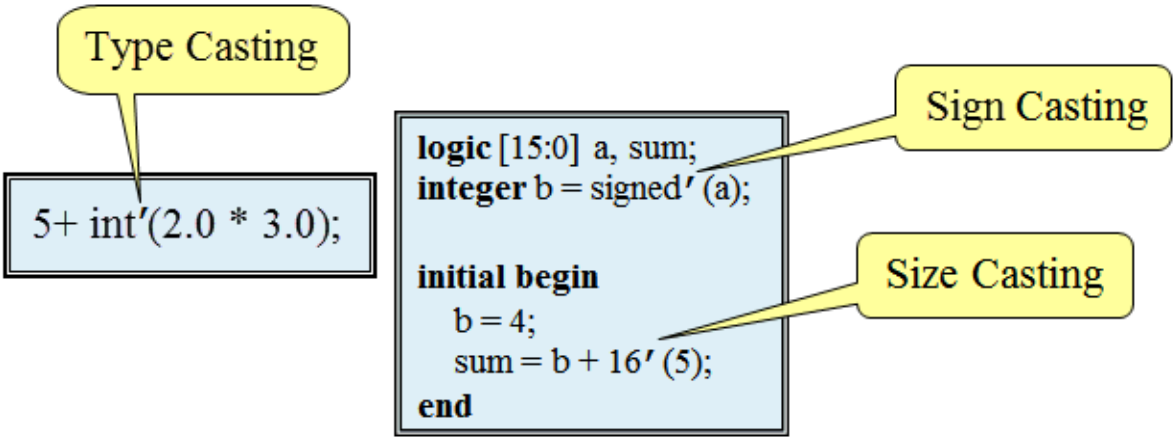
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 5-3:

Type Casting

- A variable of one type can be converted to another type using the cast (') operator
- Enables variables to take values that are invalid because of their data type
- Enables variables to be compatible for arithmetic operations
- SystemVerilog allows casting of type, size and sign



Type casting casts a value to any type, including user-defined types.

Size casting casts a value to any vector size.

Sign casting casts a value to signed or unsigned.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 5-4:

Types of Type Casting

Static Casting	Dynamic Casting
Uses' operator	Uses *$cast* operator
Casting is done at compile time	Casting is done at run time
Changes the data type of a given variable or expression	Enables a given variable to take a value that would have been incompatible
Requires one argument *some_data_type'(expr);*	Requires two arguments *$cast(dest_var, source_exp);*

Examples:
signed'(x); // Set the data type of x to signed
i=int'(9.7 – 3.2); // convert real to integer
	Example:
	logic [1:0] state;
	typedef enum {IDLE, RED, GREEN, AMBER} color;
	initial begin
	$cast(state, IDLE);
	end
	Dynamic Type Casting

Dynamic casting can be performed as a task or a function depending on how invalid assignments need to be handled. When called as a task, *$cast* will assign the expression to the variable. If the assignment is invalid, a run-time error will result. When called as a function, *$cast* will assign the expression to the variable and return 1. If the assignment is invalid, the casting operation will return 0 and no run-time error will result.

Syntax:

function int $cast (singular dest_var, singular source_exp);

task $cast (singular dest_var, singular source_exp);

Dynamic casting is popularly used for handling classes which form the basis of Object-Oriented data abstraction. It is covered in detail in the SystemVerilog for Verification course.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 5-5:

Enumerated Arithmetic and Casting

```
typedef enum logic[2:0] {NoNum = 3'hx, Zero=0, One, Two, Three,
                         Four, Five, Six} Nums_data_typ;

typedef enum logic[2:0] {NoDay = 3'hx, Mo=0,Tu,We,Th,Fr,Sa,Su} Days_data_typ;

Days_data_typ day;
Nums_data_typ num;

always @(posedge clock)
   if (reset)
      day <= Mo;
   else
      day <= Days_data_typ'((day + 1)%7);

assign num = Nums_data_typ'(day);

initial begin
   $monitor("day is set to ", day.name(), " and num is set to ", num.name());
   $finish;
end
```

Integer cannot be assigned to day

Casting

```
day is set to NoDay and num is set to NoNum
day is set to Mo and num is set to Zero
day is set to Tu and num is set to One
day is set to We and num is set to Two
day is set to Th and num is set to Three
day is set to Fr and num is set to Four
day is set to Sa and num is set to Five
day is set to Su and num is set to Six
```

Note that variables *day* and *num* cannot be assigned integer values because they are enum variables of type *Days_data_typ* and *Nums_data_typ*, respectively. Hence, the casting operation is used to assign appropriate values to them.

Slide 5-6:

Apply Your Knowledge

1) What are the two methods of casting?

2) What is the difference between type casting and type conversion?

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

1) What are the two methods of casting?

- Static casting using *`operator*
- Dynamic casting using *$cast* function

2) What is the difference between type casting and type conversion?

Type conversion involves multiple lines of RTL code to convert one variable type to another type.

With type casting, a variable or value can be converted to a new type within an expression without any assignment being made.

Slide 5-7:

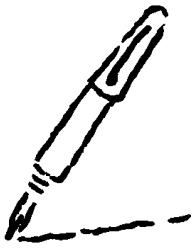
Summary

- Type casting can be used to simplify operations involving different data types
- Two types of type casting: static and dynamic casting
- Static casting is used to convert the data type of a variable or expression
- Dynamic casting is used to assign values to a variable that might not ordinarily be valid because of a differing data type
- Besides type casting, SystemVerilog also supports signed casting and size casting

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Arrays and Strings

Slide 6-1:

2021.1

This module explains the use of arrays in SystemVerilog. 20 minutes.

Slide 6-2:

Objectives

After completing this module, you will be able to:

- Identify the enhancements to Verilog arrays provided by SystemVerilog
- Describe the different types of arrays and their applications
- Define queues and illustrate their unique features

Slide 6-3:

Definition of Arrays

- An array is a data type that holds a fixed number of equally sized, same data type elements
- An array is packed if the vector range comes before the signal name
  - Example

    reg [7:0] a; // 8-bit packed array

    integer x [3:0]; // 4-bit unpacked array

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-4:

History of Arrays

- Verilog-95
  - Permitted up to two dimensional arrays where one is packed and the other is unpacked. Example: *reg [7:0] num [1023:0]*
  - Array elements could be of type reg, integer
- Verilog-2001
  - Permitted multiple dimensions of unpacked arrays. Example: *reg [7:0] num [1023:0] [3:0]*
  - All Verilog types except event could be used
- SystemVerilog
  - Permits multiple dimensions of packed arrays
  - Supports dynamic arrays, associative arrays, and queues
  - Permits arrays of structures and unions
  - Array literals '{ } can be used to assign constant member expressions
  - Includes several enhancements to Verilog arrays

Slide 6-5:

SystemVerilog Enhancements to Verilog Arrays

- Arrays can use event type and SystemVerilog types
- User-defined array types can be created
- SystemVerilog allows multi-dimensional packed arrays

Arrays can use Verilog event type and SystemVerilog types such as logic, bit, byte, int, longint, shortreal, and real

User-defined array types can be created including types such as struct and enum.

Verilog allows only one-dimensional packed arrays. SystemVerilog allows multi-dimensional packed arrays.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-6:

Packed Arrays (1)

- In both Verilog and SystemVerilog, an array is packed if the vector range comes before the signal name

- Example

```
reg [7:0] a;            // 8-bit packed array
integer x [3:0]; // 4-bit unpacked array
```

- Multi-dimensional packed arrays remove the restriction that only one array element, or bit-select or part-select of an element can be referenced

- Example

```
reg [7:0] [7:0] data;     //2-D packed array. Illegal in Verilog
reg [15:0] word;            // Legal in Verilog
word = data [1:0];       //2-slices assigned to word. Illegal in Verilog
```

Slide 6-7:

Packed Arrays (2)

- Packed arrays can only be created out of integral data types

- Packed arrays are suited for models that access sub-fields of a vector (e.g., data packets)

- Unpacked arrays are suited for models where one element is accessed at a given time (e.g., RAMs and ROMs)

- Packed and unpacked arrays are synthesizable

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Note that unlike packed structures and unions, packed arrays do not require the keyword *packed*.

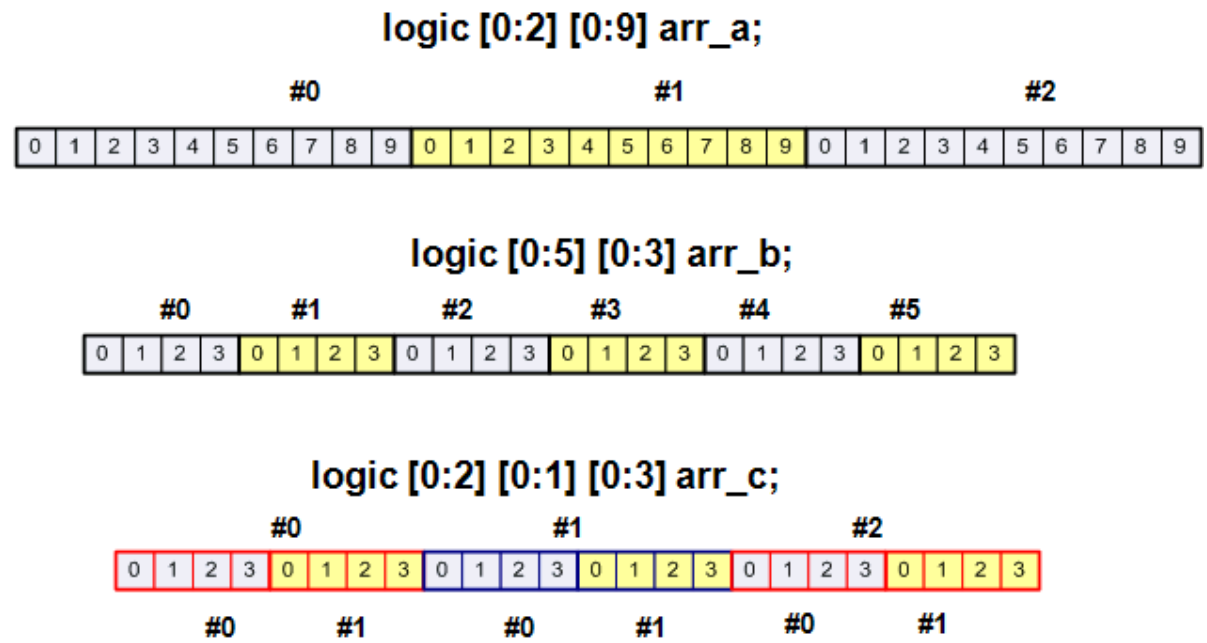
To illustrate the second point further, see the following Verilog example:

integer [3:0] [7:0] a;
integer b;
b = a[2][4]; //Legal in Verilog: references 1 element
b = a[2]; //Illegal in Verilog: references 8 elements

From a designer's perspective, it does not make much of a difference whether a construct is packed or unpacked. However, from a simulation perspective, packed arrays may result in better performance depending on the size of the array.

Slide 6-8:

Packed Arrays Example (1)



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-9:

Packed Arrays Example (2)

```
bit [0:3] [7:0] packed_array;

initial begin
   packed_array = 32'h01234567;
   #1 $displayh(packed_array);
   packed_array[2+:2] = packed_array[0+:2];
   #1 $displayh(packed_array);
   $finish;
end
```

Packed array of 4 bytes (packed into 32 bits)

Value assignment as 32-bit vector

Two bytes copy

```
01234567
01230123
```

Slide 6-10:

Array Querying System Functions (1)

Function Name	Description
$left()	Returns the left bound (MSB) of the dimension
$right()	Returns the right bound (LSB) of the dimension
$low()	Returns the minimum of $left and $right of the dimension
$high()	Returns the maximum of $left and $right of the dimension
$increment	Returns 1 if $left is greater than or equal to $right and -1 otherwise
$size()	Returns the number of elements in the dimension
$dimensions()	Returns the total number of dimensions in the array
$unpacked_dimensions()	Returns the total number of unpacked dimensions for an array

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

VHDL users: These are similar to array attributes in VHDL.

SystemVerilog also provides several built-in methods to facilitate array searching, ordering, and reduction. Covering each of these methods is beyond the scope of this module. The exhaustive list of methods which facilitate array manipulation can be found in the *SystemVerilog Language Reference Manual*.

Slide 6-11:

Array Querying System Functions (2)

```
reg [3:0][2:1] n [1:5][2:8];
int i;

initial begin
   $display("Dimensions  = %0d", $dimensions(n));
   for (i=1; i<= $dimensions(n); i++) begin
      $display("%s", {32{"="}});
      $display("The dimension #%0d statistics", i);
      $display("Left = %0d, Right = %0d", $left(n, i), $right(n, i));
      $display("Low  = %0d, High  = %0d", $low(n, i), $high(n, i));
      $display("Increment = %0X", $increment(n, i));
      $display("Size  = %0d", $size(n, i));
   end
end
```

```
Dimensions = 4
==============================
The dimension #1 statistics
Left = 1, Right = 5
Low  = 1, High  = 5
Increment = FFFFFFFF
Size = 5
==============================
The dimension #2 statistics
Left = 2, Right = 8
Low  = 2, High  = 8
Increment = FFFFFFFF
Size = 7
==============================
The dimension #3 statistics
Left = 3, Right = 0
Low  = 0, High  = 3
Increment = 1
Size = 4
==============================
The dimension #4 statistics
Left = 2, Right = 1
Low  = 1, High  = 2
Increment = 1
Size = 2
```

- Array querying functions are synthesizable

VHDL users: Unlike VHDL, SystemVerilog does not support unconstrained arrays (i.e., arrays of undefined length). However, SystemVerilog supports dynamic arrays, which are not synthesizable because the size can vary during run time. This disallows use of these functions the same way as attributes are used in VHDL for creating functions.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-12:

Dynamic Arrays

- A dynamic array is an unpacked array
- Array size can be set or changed at run time
- The *new[]* constructor is used to set or change the size of the array and initialize its elements
- Examples

```
integer dynamic_array1[]; //Default size of dynamic arrays is 0
integer dynamic_array2[];

dynamic_array1 = new[7]; // Allocates 8 elements; Each element has its default initial value

dynamic_array2 = new[5](`{1,2,3,4}); // dynamic_array2 has [1,2,3,4,0]

dynamic_array2 = new[8](dynamic_array2); // Preserves the previous 5 values

dynamic_array1[3:0] = dynamic_array2[7:4]; // Copying elements

dynamic_array2 = new[100]; //100 new elements allocated. Previous data lost
```

- Dynamic arrays are not synthesizable

The *size()* built-in method returns the current size of the array.

delete() clears all the elements yielding an empty array (zero size).

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-13:

Associative Arrays

- Associative arrays are used when the size of the collection is unknown
- The index expression can be of any type and is not restricted to integral expressions
- Can be used to implement a lookup table of the elements of its declared type
- Associative arrays are not synthesizable

Dynamic arrays, associative arrays, and queues are mostly used for verification routines. Therefore, these array types are not covered in detail in this course. More details on this object-oriented array type can be found in the *Designing with SystemVerilog for Verification* course.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-14:

Queues

- A queue is a variable-size, collection of elements similar to a one-dimensional unpacked array that grows and shrinks automatically
- A queue shall resize itself to accommodate any queue value that is written to it
- Queues are not synthesizable
- Examples

```
integer a[$];              // A queue of integers
int b[$] = {1, 2, 3};      // A queue of int types with 3 elements
logic c[$:15];             // A queue whose maximum size is 16 bits
int x = b[$];              // x is initialized with the last element of queue b

a[0] = 9;                  // write to the first element of queue a
b = {b,4};                 // insert 4 at the end of queue b
a = {};                    // clear queue a
b = {b[0:x],7,b[x+1,$]};   //insert 7 after position x in queue b
```

- The range of a queue slice can be arbitrary (such as Q[a:b]) and are not required to be constant expressions
- A queue supports constant-time access to its elements and constant-time insertion/removal at the beginning or end of the queue
- Queues are declared like unpacked arrays, but specifying *$* as the array size
- *$* also represents the position of the last element in the queue
- The maximum size of a queue can be limited by specifying its optional right bound (last index)
- Queues may be accessed by index, or by using methods such as push, pop, etc.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-15:

Apply Your Knowledge

1) List some of the SystemVerilog enhancements to Verilog arrays

2) Is the following array packed or unpacked?

reg [3:0] table [7:0]

1) List some of the SystemVerilog enhancements to Verilog arrays.
   - Arrays can use the event type and SystemVerilog types: *logic*, *bit*, *byte*, *int*, *longint*, *shortreal*, and *real*.
   - User-defined array types can be created, including types such as *struct* and *enum*.
   - Verilog allows only one-dimensional packed arrays. SystemVerilog allows multi-dimensional packed arrays.
   - SystemVerilog allows array declaration with dimension size.
     - For example: *logic [31:0] data [1024]; //same as logic [31:0 data [1023:0];*
   - Unlike Verilog, there is no restriction that only one array element can be referenced.
   - SystemVerilog provides several built-in functions to query and manipulate arrays.

2) Is the following array packed or unpacked?

reg [3:0] table [7:0]

Unpacked. The above declaration will store four array elements (each 8-bits wide) independently.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 6-16:

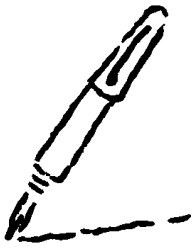
Summary

- SystemVerilog provides several enhancements to Verilog arrays
- Packed arrays are suited for models that access sub-fields of a vector (like data packets)
- Dynamic arrays and associative arrays are queues that are new types of arrays that are useful for verification

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

SystemVerilog Building Blocks

Slide 7-1:

2021.1

This module describes the design and verification building blocks in SystemVerilog.
15 minutes.

Slide 7-2:

Objectives

After completing this module, you will be able to:

- Explain the design and verification building blocks in SystemVerilog

Slide 7-3:

Design and Verification Building Blocks

- A design element in Systemverilog
  - *module*
  - *program*
  - *interface*
  - *checker*
  - *package*
  - *primitive*
  - *config*

These design elements (module, program, interface, checker, package, primitive, and config) are the primary building blocks to create a design and to verify the design (verification environment).

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

lide 7-4:

Modules

- The *module* is the basic block to represent the logic description
  - It contains all the information relevant to the hardware module
  - It can also be a container for verification code; interconnections between verification block and design block

```
`define < > //transcends module boundary

module <module name> #(parameter(s) )
(port declarations);

wire or reg declarations (internal to module)
…
Constant declarations
…
User-defined type definitions
…
Class definitions
…
Import of declarations from packages
…
behavioral, data flow, gate-level constructs
…
lower-level module instantiations(s)
…
Subroutine definitions
…
Continuous assignments

endmodule
```

The module is the basic entity in SystemVerilog and it encapsulates all the functionality. Defining a module determines the module inputs, outputs, and its functionality.

A SystemVerilog module can contain following constructs:

- Ports with port declarations
- Data declarations such as nets, variables, structures, unions
- Constant declarations
- Imports of declarations from packages
- Subroutine definition
- Instantiations of other modules, interfaces, programs, checkers, and primitives
- Instantiations of class objects
- Continuous assignments
- Procedural blocks

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 7-5:

Programs

- Program block
  - Modeling the testbench environment
  - Serves as a clear separator between design and testbench

```
interface my_interface(
  input clock, reset
);
  logic a,b,c;
endinterface
```

```
module my_design(
  input clock, reset,
  my_interface ifc
);
// …
endmodule
```

```
module testbench;
  logic clock = 1'b0;
  logic reset = 1'b1;

  always #5 clock = ~clock;
  initial #12 reset = 1'b0;

  my_interface ifc (
    .clock(clock),
    .reset(reset)
  );
  my_design dut (
    .clock(clock),
    .reset(reset),
    .ifc(ifc)
  );
  my_program test(
    .clock(clock),
    .reset(reset),
    .ifc(ifc)
  );

endmodule
```

```
program my_program(
  input clock, reset,
  my_interface ifc
);

  initial begin
    ifc.a = 0;
    ifc.b = 1;
    ifc.c = 0;
  end

// …
endprogram
```

In Verilog, testbench code and design code are in a separate *module*. To avoid race conditions between the design and testbench, SystemVerilog introduces the *program* block to separate the testbench.

The program block serves the following purposes:

- Provides an entry point to the execution of testbenches.
- Creates a scope that encapsulates program-wide data, tasks, and functions.
- Provides a syntactic context that specifies scheduling in the reactive region.

Together with clocking blocks, the program construct provides for race-free interaction between the design and the testbench and enables cycle and transaction-level abstractions.

The abstraction and modeling constructs of SystemVerilog simplify the creation and maintenance of testbenches.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 7-6:

Interfaces

- An interface is an abstract port type that allows many signals to be grouped together and represented as a single port

```
interface mem_bus(
  logic [31:0] mem0_wdata;
  logic [31:0] mem0_rdata;
  logic write;
  logic [15:0] addr;
  logic mem_enable;
);
  logic a,b,c;
endinterface
```

Signals for mem bus defined in just one place

```
module mController(
  mem_bus bus, // interface port
  input clock
);
  //...
endmodule
```

Each module definition has a single port for mem_bus

Each module instance has a single connection to mem_bus

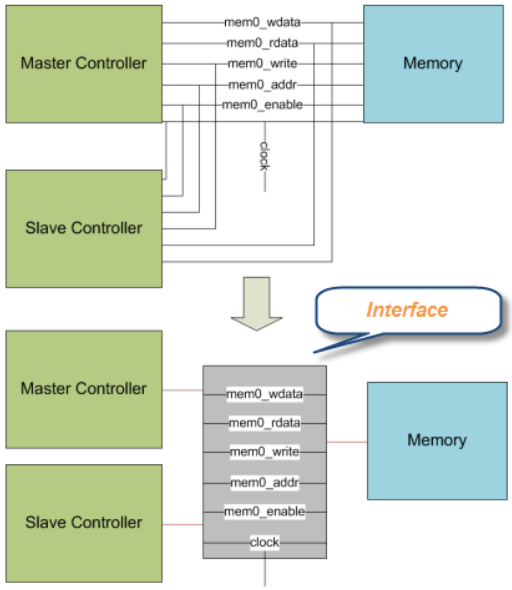
```
module mem(
  mem_bus bus, //interface port
  input clock
);
  //...
endmodule
```

```
module sController(
  mem_bus bus, // interface port
  input clock
);
  //…
endmodule
```

```
module uc_top(
  input logic clock
);
  mem_bus bus ();

  mController mC(
    .bus(bus)
    // other ports
    .clock(clock)
  );

  sController sC(
    .bus(bus)
    .clock(clock)
  );

  memory mem(
    .*
  )
endmodule
```

Instance of an interface and the instance name is *bus*

Interfaces support .*port connection style



Slide 7-7:

Checkers

- Checkers
  - Encapsulating assertions along with the modeling code
  - Use of checkers is to serve as verification library units

```
checker my_check(logic rx_data_rdy);
  property p(logic test_sig);
    …
  endproperty
  a1: assert property (p (rx_data_rdy) );
endchecker : my_check
```

Slide 7-8:

Primitives

- Primitives
  - Used to describe low-level gates and switches
  - SystemVerilog includes a number of built-in primitives
  - Designers can build their own primitives with user-defined primitives (UDPs)

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 7-9:

Packages

- Identifiers declared within a module, interface, program, or checker are local to that scope
  - Does not affect or conflict with declarations in other building blocks

- SystemVerilog packages provide an additional mechanism to share with other building blocks

```
package usbSetupPacket;
  typedef struct{
    byte bmRequestType,
    byte bRequest,
    byte wValue,
    byte wIndex,
    byte wLength;
  }usbSetup;

function usbSetup requestType(usbSetup setup);
  // Only bits (3-0) D3…0
  // Receipent – 0:Device, 1:Interface, 2:Endpoint, 3:Other
  receipent = setup.bmRequestType;
endfunction
…
endpackage : usbSetupPacket
```

SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence, property, and checker declarations among multiple SystemVerilog modules, interfaces, programs, and checkers.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 7-10:

Config

- A configuration specifies the exact version and source location of each SystemVerilog module
- Configuration is specified outside the module declaration, so the SystemVerilog module does not need to be modified to reconfigure a design
- Configuration name exists in the same namespace as module and primitive names
- Virtual libraries are used in the configuration
  - The virtual libraries are associated with physical locations with a separate library file

Slide 7-11:

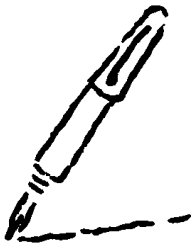
Summary

- Design elements are the basic building blocks to
  - Model and create a design
  - Create a verification environment
- The keywords for the design elements are
  - *module*, *program*, *interface*, *checker*, *package*, *primitive,* and *config*

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Structures

Slide 8-1:

2021.1

This module explains the use of structures in SystemVerilog. 15 minutes.

Slide 8-2:

Objectives

After completing this module, you will be able to:

- Describe structures and their usage in SystemVerilog

Slide 8-3:

Definition of Structures (1)

- A structure is a data type that represents a collection of data types to allow convenient grouping of signals

Definition of Structures

```
struct {
    int a;
    byte b;
    } my_struct;
```

Referencing Structures

Referencing the whole structure

```
my_struct = '{a:32'hFFFF_FFFF, b:8'FF};

my_struct.a = 32'hFFFF_FFFF;
```

Referencing an individual element of the structure

- Uses the keyword *struct*
- Structures are synthesizable

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Structures, unions, and arrays are classified as aggregate data types. They consist of one or more singular data types such as *logic*, *int*, *integer*, *bit*, etc.

Structures can be referenced as a whole or the individual variable that make up the structure.

Slide 8-4:

Definition of Structures (2)

- Advantages
  - Enables a higher level of abstraction
  - Easy to interface with other languages
  - Can improve simulation performance

Structures	Arrays
Can have members of different data types	Require all elements to be of the same type and size
Members are referenced by name	Elements of an array are referenced by index

- A structure can be a user-defined type with *typedef* keyword

- A structure declared without the *typedef* keyword is referred to as an anonymous structure

Advantages:
  - Enables a more abstract level of hardware modeling, making the code smaller, organized, and easier to read
  - Similar to *struct* in C. Easy to interface with and translate from other languages
  - Based on how often the structure is used, it can improve simulation performance

Structures are not supported by Verilog.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 8-5:

Unpacked and Packed Structures

Unpacked Structures	Packed Structures
By default, a structure is unpacked	The keyword *packed* is used to declare a packed structure
The members of the unpacked structure are treated as independent variables or constants grouped together under a common name	Packed structures store all members as contiguous bits in a specific order
The layout of the storage varies from one software to another	A packed structure is stored as a vector, with the first member occupying the left-most field
Members of an unpacked structure can be of integral or non-integral data types	All members of a packed structure must be integral data types

Integral data types hold values that can be represented as a vector or a contiguous set of bits. A structure cannot be packed if any of the members of the structure cannot be represented as a vector. This means that a packed structure cannot contain *real*, *shortreal*, or other non-integral variables.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 8-6:

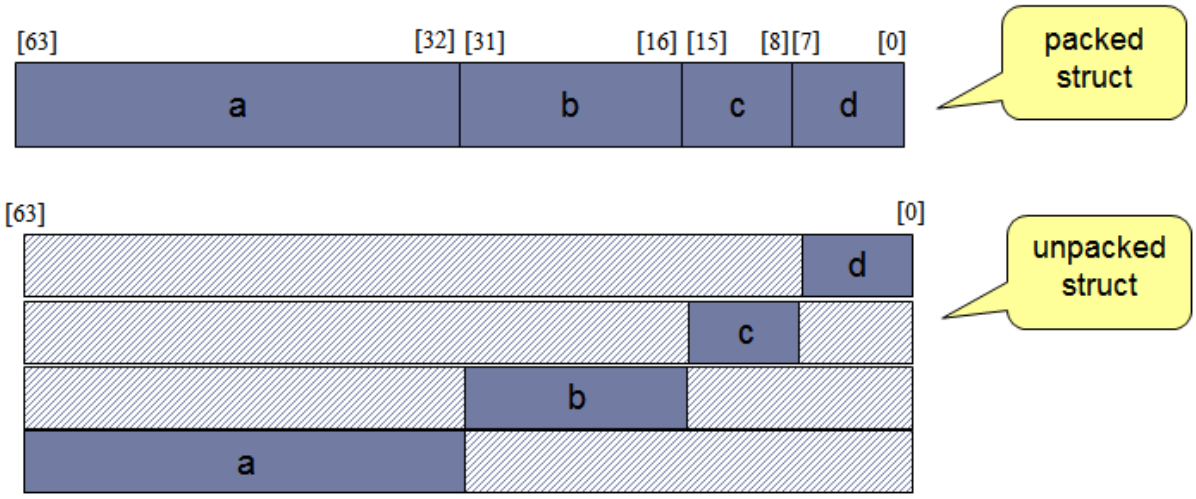
Packed Structures Example

```
struct packed {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1;
```

Packed constructs in SystemVerilog are organized as vectors

Members of a packed structure can be referenced by member name or part select

pack1.c = 8'h22 is same as pack1[15:8] = 8'h22;



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 8-7:

Unsigned and Signed Packed Structures

- Packed structures can be unsigned (default) or signed

```
struct packed unsigned {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1; // unsigned, 2-state

struct packed signed {
   integer a;
   logic [15:0] b;
   logic [ 7:0] c;
   logic [ 7:0] d;
} pack2; // signed, 4-state

longint pack1_vector;
logic [63:0] pack2_vector;

initial begin
   pack1.a = 32'hABCDDCBA;
   pack1.b = 16'hBABE;
   pack1.c = 8'h77;
   pack1.d = 8'h55;
   pack1_vector = pack1[63:0];
   pack2_vector = pack1_vector;
   pack2[63:0] = pack2_vector;
   // $display(...
   #1 $finish;
end
```

unsigned is a default and it is optional here

```
pack1 -> pack1.a = abcddcba
pack1 -> pack1.b = babe
pack1 -> pack1.c = 77
pack1 -> pack1.d = 55
pack1 = abcddcbababe7755
pack1_vector = abcddcbababe7755
pack2 -> pack2.a = abcddcba
pack2 -> pack2.b = babe
pack2 -> pack2.c = 77
pack2 -> pack2.d = 55
pack2 = abcddcbababe7755
pack2_vector = abcddcbababe7755
```

The signedness of the structures influences the mathematical or relational operations performed on these structures.

This does not affect how the members of the structure are perceived. Each member is considered signed or unsigned, based on the type declaration of that member.

A part-select of a packed structure will always be unsigned.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 8-8:

Structure Assignment Patterns

- Structures can be assigned using structure literals
- Structure literals are assignment patterns that use constant member expressions
- Assignments can be done by place, by name, by type, and by default

```
typedef struct {
   int x;
   int y;
   logic [7:0] z;
} my_struct;

my_struct s1;

int k = 1;

initial begin
 #1 s1 = '{123, 4, 8'h04};
  #1 $display(s1.x, s1.y, s1.z);
  #1 s1 = '{x:456, y:7, z:8'h05};
  #1 $display(s1.x, s1.y, s1.z);
  #1 s1 = '{int:789, logic:8'h06};
  #1 $display(s1.x, s1.y, s1.z);
  #1 s1 = '{x:100, default:0};
  #1 $display(s1.x, s1.y, s1.z);
  #1 $finish;
end
```

By place

By name

By type

By default

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 8-9:

Structure Assignment Patterns by Type and Default

```
typedef logic[3:0] nibble;

module test;

struct {
   logic [7:0] a;
   int b;
   bit signed [31:0] c;
   nibble d, e, f;
} s1;

initial begin
   s1 = '{int:1, a:2, e:5, nibble:3, default:0};
   #1
   $display("s1: .a = %0d, .b = %0d, .c = %0d", s1.a, s1.b, s1.c);
   $display("s1: .d = %0d, .e = %0d, .f = %0d", s1.d, s1.e, s1.f);
end

endmodule
```

- Set all int variables to 1
- Set a=2, e=5
- Set all other nibble variables to 3
- Set all others to 0

```
s1: .a = 2, .b = 1, .c = 0
s1: .d = 3, .e = 5, .f = 3
```

Slide 8-10:

Apply Your Knowledge

1) How are structures different from arrays?

2) Are unpacked structures synthesizable?

A

1) How are structures different from arrays?

Structures can have members of different data types while arrays require all elements to be of the same type and size. In addition, members of a structure are referenced by name while elements of an array are referenced by index.

2) Are unpacked structures synthesizable?

Yes, both packed and unpacked structures are synthesizable.

Slide 8-11:

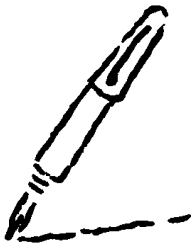
Summary

- Structures extend Verilog to represent a collection of elements

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Unions

Slide 9-1:

2021.1

This module explains the use of unions in SystemVerilog. 10 minutes.

Slide 9-2:

Objectives

After completing this module, you will be able to:

- Describe unions and their usage in SystemVerilog

Slide 9-3:

Definition of Unions (1)

- A union is an aggregate data type like structure but uses the same memory for all its members
- Only one member of the union can be used at a time

Defining a Union

```
union {
    int x;
    logic y;
    } my_union;
```

Referencing a Union

Only one member can be referenced at a time

```
my_union.x = -5;

my_union.y = 1;
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

If a union has two members of different data types, they are allocated the same memory. The memory allocated is equal to the maximum size of the members.

Members of a union can be of different types.

Uses the keyword *union*. Declaration is similar to structures.

Packed unions are well supported by simulators and synthesis tools compared to unpacked unions, which are not synthesizable.

Slide 9-4:

Definition of Unions (2)

- Have the same advantages as structures

Unions	Structures
Use the same memory for all members	Use different memory for each member
More efficient than structures when its members are not required to be accessed at the same time	Can access more than one member at the same time, unlike unions

- Not supported by Verilog
- A union can be user-defined type with the *typedef* keyword
- A union declared without the *typedef* keyword is referred as an anonymous union

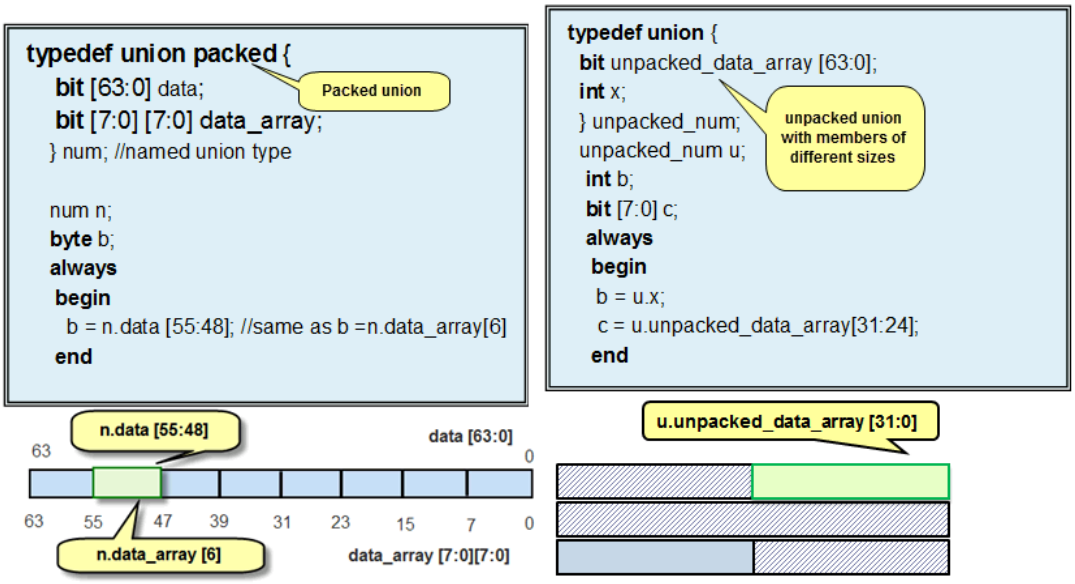
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 9-5:

Unpacked and Packed Unions

- Packed unions require that all members be the same size and be packed types
- By default, a union is unpacked (members can be different sizes)
- Packed unions are synthesizable while unpacked unions are not supported for synthesis
- Packed unions support writing to one member and reading from another



It is important that a value always be read back from the union in the same type with which it is written.

If a value is stored using one union member (which may be of different data type), and read back from a different union member, then the value that is read is not defined, and may yield different results in different software tools.

Both packed and unpacked unions are supported by the Vivado® Simulator tool but not by the synthesis tool.

Packed unions can be signed or unsigned.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

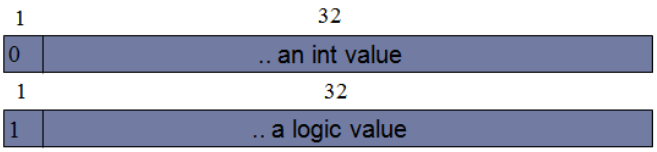
Slide 9-6:

Tagged Unions

- A tagged union stores both the member value and additional bits (tag) representing the current member name

```
// tagged union declaration
typedef union tagged {
   int a;
   logic [31:0] b;
} tagged_data;

tagged_data data;
int data_out;
initial begin
  // assigning to a tagged union member
  data = tagged a 7; //stores value 7 in data.a
  //reading from tagged union
   data_out = data.a;
  //data_out = data.b //ERROR: member doesn't match union's tag
  //correct read method
  data = tagged b 45;
  data_out = data.b;
end
```

If the member being read isn't the last assigned member, an error will occur



tag is 0 for a and 1 for b

- Tagged unions are not supported by either the synthesis tool or the simulation tool

Tagged unions eliminate the danger of storing a value of one type and interpreting those bits as another type.

The tag size is the minimum bits needed for all the member names (e.g., 5-8 members -> 3 tag bits).

The tag bits are always left-justified; for each member, the member bits are always right-justified.

Note that the synthesis tool will generate the logic for checking the tag, provided all members of the union are synthesizable. Members of type *void* are synthesizable.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 9-7:

Apply Your Knowledge

1) What are the differences between unions and structures?

1) What are the differences between unions and structures?

Unions use the same memory for all members while structures use different memory for each member. Also, a union is more efficient when its members are not required to be accessed at the same time.

Slide 9-8:

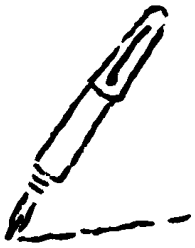
Summary

- Unions extend Verilog to represent a collection of elements
- Unlike structures, unions use only one memory location and only one of its member can be referenced

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Additional Operators in SystemVerilog

Slide 10-1:

2021.1

This module describes the operators supported by SystemVerilog beyond those found in Verilog. 20 minutes.

Slide 10-2:

Objectives

After completing this module, you will be able to:

- Recall the Verilog operators that are applicable in SystemVerilog
- Describe the additional operators provided by SystemVerilog
- Explain streaming operators
- Identify the precedence of operators in an expression

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-3:

Verilog Operators: A Revisit

All These Operators Are Synthesizable

Operator	Name	Operator	Name
{} {{}}	Concatenation, replication	||	Logical or
unary + unary -	Unary operators	==	Logical equality
+ - * / **	Arithmetic	!=	Logical inequality
%	Modulus	===	Case equality
> >= < <=	Relational	!==	Case inequality
!	Logical negation	~	Bitwise negation
&&	Logical and	&	Bitwise and
|	Bitwise inclusive or	^	Reduction xor
^	Bitwise exclusive or	^~ or ~^	Reduction xnor
^~ or ~^	Bitwise equivalence	<<	Logical left shift
&	Reduction and	>>	Logical right shift
~&	Reduction nand	<<<	Arithmetic left shift
|	Reduction or	>>>	Arithmetic right shift
~|	Reduction nor	?:	Conditional

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-4:

Assignment Operators

- SystemVerilog adds C-like assignment operators
- Assignment operators are blocking operators
- Synthesizable

Operator	Name	Operand Data Types	Example	Equivalent in Verilog
+= -= /= *=	Arithmetic assignment	integral, *real*, *shortreal*	A += B	A = A + B
%=	Arithmetic modulus assignment	integral	A %= B	A = A % B
&= |= ^=	Bit-wise assignment	integral	A &= B	A = A & B
>>= <<=	Logical shift assignment	integral	A >>= B	A = A >> B
>>>= <<<=	Arithmetic shift assignment	integral	A >>>= B	A = A >>> B

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-5:

Wildcard Equality Operators

- Wildcard equality operators (synthesizable)

Operator	Name	Operand Data Types	Example	Description
==?	Wildcard equality	integral	A ==? B	A equals B, X and Z values in B are treated as wildcards (i.e., X and Z values in B will match with any bit value - 0,1, X or Z)
!=?	Wildcard inequality	integral	A !=? B	A does not equal B, X and Z values in B are treated as wildcards

Difference between wildcard equality and other equality operators:

- The logical equality operators (== and !=) may result in X if any of their operands contains an X or Z.
- The case equality operators (=== and !==) explicitly check for 4-state values; therefore, X and Z values shall either match or mismatch, never resulting in X.
- The wildcard equality operators (==? and !=?) may result in X if the left operand contains an X or Z that is not being compared with a wildcard in the right operand.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-6:

Wildcard Equality Operator Example

```
module wildcard_equality ();
  logic [3:0] A = 4'b1111;
  logic [3:0] B = 4'bxxz1;
  logic [3:0] C = 4'bxxz0;

  initial
   begin
     if (A ==? B)
        $display ("A %b matches with B %b", A, B);
     if (A !=? C)
        $display ("A %b doesn't match with C %b", A, C);
   end
endmodule
```

```
A 1111 matches with B xxz1
A 1111 doesn't match with C xxz0
```

Slide 10-7:

Increment/Decrement Operators

- Increment/decrement operators (synthesizable)

Operator	Name	Operand Data Types	Example	Equivalent in Verilog
++	Unary increment	integral, *real*, *shortreal*	A = B++	A =B; B = B+1;
--	Unary decrement	integral, *real*, *shortreal*	A = --B	B = B-1; A = B;

Caution needs to be taken while using bump operators (a++, --x, ..) because different tools may handle such expressions differently.

For example: *x* = *a*++ * *a*--

The above expression can be interpreted as:

- Increment *a* by 1 and multiply it with the original value of *a*, or
- Multiply *a* with *a* and after executing the operation, increment and decrement *a*. (i.e, *a* remains unaltered)

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-8:

Inside Operator

- Also called a set membership operator
- Returns 1 if an expression is a member of a given list or range
- Example

```
int a;
initial begin
   a = 32;
   #1 a = 4;
   #1 a = 5;
   #1 a = 4;
end
```

```
a (=32) is inside
{2,4,8,16,32}
a (=4) is inside 1..8
a (=5) is inside 1..8
a (=4) is inside 1..8
```

```
always @(*)
   if (a inside{[1:8]})
      $display("a (=%0d) is inside 1..8", a);
   else if (a inside {2,4,8,16,32})
      $display("a (=%0d) is inside {2,4,8,16,32}", a);
```

- Inside operator is supported for synthesis

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-9:

Streaming Operators (pack/unpack) (1)

- The streaming operators are used to pack/unpack bitstreams
- Users can also specify the sequence of bit order
- There are two stream operators
  - >> causes blocks of data to be streamed in left-to-right order
  - << causes blocks of data to be streamed in right-to-left order
- Provides the option to specify the slice size or type before the expression to be streamed

x =	1	0	1	0
{<<{x}}	0	1	0	1
{>>{x}}	1	0	1	0
x =	1	0	1	0
{<<2{x}}	1	0	1	0

The streaming operation can also be described as a more concise way of slicing and concatenating to assign among different objects.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-10:

Streaming Operators (pack/unpack) (2)

- Examples

```
int X = { "A", "B", "C", "D" };

        { >> {X}} // generates stream "A" "B" "C" "D"

        { << byte {X}} // generates stream "D" "C" "B" "A" (little endian)

        { << 16 {X}} // generates stream "C" "D" "A" "B"

        { << { 8'b1000_1010 }} // generates stream 'b0101_0001 (bit reverse)

        { << 4 { 6'b10_0010 }} // generates stream 'b0010_10

        { >> 4 { 6'b10_0010 }} // generates stream 'b1000_10 (same)

        { << 2 { { << { 4'b1101 }} }} // generates stream 'b1110
```

- Streaming operators are synthesizable

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-11:

Packing Using Streaming Operators

- Streaming operators can be used to pack data when used in the right side of an expression

- The following example shows how stream operators can be used to pack a structure and an array

```
struct {
   int a;
   int b;
} myStruct;

bit [63:0] s;
bit [ 7:0] arr [0:7];

initial begin
   myStruct = {a :32'h01234567, b:32'h89ABCDEF};
   s = { >> {myStruct}};
   #1 $display("s = 0x%016h", s);

   arr = '{default:8'h55};
   s = { >> {arr}};
   #1 $display("s = 0x%016h", s);
end
```

```
s = 0x0123456789abcdef
s = 0x5555555555555555
```

The result of the pack operation can be assigned directly to any bit-stream type variable:

- If the left-hand side represents a fixed-size variable and the stream is larger than the variable, an error will be generated.
- If the variable is larger than the stream, the stream is left-justified and zero-filled on the right.
- If the left-hand side represents a dynamically sized variable, such as a queue or dynamic array, the variable is resized to accommodate the entire stream.
- If after resizing, the variable is larger than the stream, the stream is left-justified and zero-filled on the right.

The stream is not an integral value. To participate in an expression, type casting is required.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-12:

Unpacking Using Streaming Operators

- Streaming operator can be used to unpack data when used in the left side of an expression

- The following example shows how stream operators can be used to unpack a packed array

```
module test;

bit [7:0] arr[7:0]; // unpacked array
longint vec;

initial begin
   vec = {$urandom(), $urandom()};
   $display("vec = %p", vec);

   {<<8{arr}} = vec;
   $displayh("arr = ", arr);

   {>>{arr}} = vec;
   $displayh("arr = ", arr);
end

endmodule
```

arr is defined as an unpacked array because we will unpack in to it

Streaming operator can be used at left side of expression

```
vec = 05e2353657dea13c
arr = '{'h3c, 'ha1, 'hde, 'h57, 'h36, 'h35, 'he2, 'h5}
arr = '{'h5, 'he2, 'h35, 'h36, 'h57, 'hde, 'ha1, 'h3c}
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-13:

Operator Precedence in SystemVerilog

- SystemVerilog has the same operator precedence as Verilog
- The precedence of the additional operators are highlighted in the table below

Operator	Precedence
() [] :: .	highest
+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ -- (unary)	↓
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >= inside dist	
== != === !== ==? !=?	
& (binary)	
^ ~^ ^~ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>=	
{ } { { } }	lowest

The highlighted operators represent operators that are additionally supported by SystemVerilog.

The distribution operator (*dist)* is used for constraining vector generation and is covered in detail in the *Verification with SystemVerilog* course.

In a way, the = operator is also additionally supported by SystemVerilog because one can use = in an expression in SystemVerilog (unlike in Verilog).

Example: *a = (b=c);*

- if (a==b) //legal in both Verilog and SV
- if (a=b) //illegal in both Verilog and SV
- if ((a=b)) //illegal in Verilog and legal in SV, interpreted as *a=b; if (a) begin ... ;*

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 10-14:

Literals

- Unsized literal bit values can be specified with a preceding (')
- '0, '1, 'X, and 'Z sets all bits to this value
- Example
  - reg[31:0] a = '1;
  - Equivalent to reg[31:0] a = 32'hffff_ffff in Verilog

Slide 10-15:

Apply Your Knowledge

1) What is the difference between the wildcard equality (==?) and logical equality (==) operators?

2) What is the *inside* operator used for?

A

1) What is the difference between the wildcard equality (==?) and logical equality (==) operators?

   The difference is that the logical equality operator (==) can result in X if any of their operands contains an X or Z. The wildcard equality operator (==?) can result in X if the left operand contains an X or Z that is not being compared with a wildcard in the right operand.

2) What is the *inside* operator used for?

   The *inside* operator is used to check if an element is a member of a given list or range.

Slide 10-16:

Apply Your Knowledge

1) True or false: Streaming operators can be used to pack data when used in the left side of an expression

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

A

1) True or false: Streaming operators can be used to pack data when used in the left side of an expression.

False. Streaming operators can be used to pack data when used in the right side of an expression.

Slide 10-17:

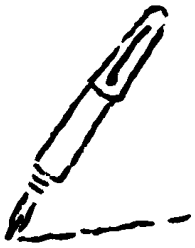
Summary

- SystemVerilog includes several new operators not supported by Verilog
- The streaming operators are used to pack/unpack bitstream types in a user-specified order
- SystemVerilog follows the same operator precedence as Verilog

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Procedural Statements

Slide 11-1:

2021.1

This module describes the different procedural blocks provided by SystemVerilog. 15 minutes.

Slide 11-2:

Objectives

After completing this module, you will be able to:

- Explain the three new types of *always* blocks that help with modeling synthesizable procedural blocks
- Describe the *final* procedural statement

Slide 11-3:

always: General-Purpose Procedural Block

- Verilog *always* procedural block is used to model a wide variety of hardware types and verification routines
  - At the RTL level: used to model combinational, latched, and sequential logic
  - At a more abstract level: used to model algorithmic logic, such as state machines
  - *always* block is also used in testbenches to model clocks and other verification tasks needed for verification process

Because the Verilog *always* block is general purpose, the simulation tools must allow *always* procedural blocks to be used in a variety of ways, and may not comply with the guidelines imposed by synthesis compilers. As a result, mismatches in simulation and synthesis results can occur if the designer does not follow strict, self-imposed modeling guidelines.

Owing to the general-purpose application of the always procedural block, the design intent is not clear

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-4:

Additional Procedural Blocks in SystemVerilog

- To indicate the designer's intent, SystemVerilog adds three more procedural blocks
  - *always_comb* - combinational logic procedural blocks
  - *always_latch* - latched logic procedural blocks
  - *always_ff* - sequential logic procedural blocks
- These are synthesizable and are best suited for design under test (DUT) rather than for testbenches
- All signals assigned within these always blocks cannot be assigned outside that block. The same is not true with the generic *always@* block

If the content of a specialized procedural block does not match the rules for that type of logic, software tools typically issue warning messages. The synthesis tool will issue warnings if, for example, always_latch is used and no latch is inferred.

It is easier for synthesis compilers and formal verification tools to deduce the type of hardware making them more efficient at modeling large and complex designs.

There is greater consistency and accuracy among simulators, synthesis tools, formal checkers, lint checkers, and other software tools.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-5:

always_comb

- *always_comb* procedural block is used to indicate the intent to model synthesizable combinational logic
- A sensitivity list is not required
  - The sensitivity list is inferred by signals read within the procedural block and which receive their value from outside the procedural block
- Triggered once at simulation time zero after all procedural blocks are started, to ensure that outputs start off consistent with inputs

Contents of the *always@** block don't have to adhere to synthesizable guidelines, whereas *always_comb* enforces certain synthesis guidelines such as variables being assigned that cannot be written to by other procedures.

The tool inferring the sensitivity list eliminates the possibility of gate-RTL mismatch which can occur if a parameter is not included in the sensitivity list.

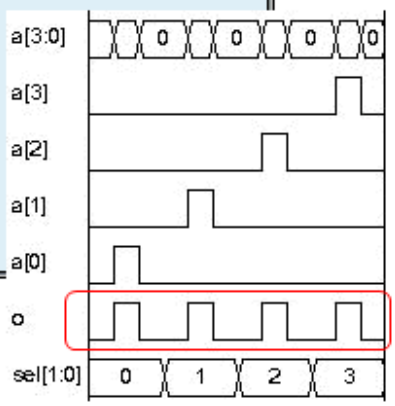
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-6:

always_comb Example

```
module func_mux (
   input [ 3:0] a,
   input [ 1:0] sel,
   output logic o
);

function f_mux_4_to_1 (input [1:0] sel);
   case (sel)
      0: f_mux_4_to_1 = a[0];
      1: f_mux_4_to_1 = a[1];
      2: f_mux_4_to_1 = a[2];
      3: f_mux_4_to_1 = a[3];
      default: f_mux_4_to_1 = 1'bx;
   endcase
endfunction

always_comb
   o = f_mux_4_to_1(sel);
endmodule
```



Notice how *always_comb* infers sensitivity to *sel*, which is a signal within the function *f_mux_4_to_1*.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-7:

Assignment

- Assignment to *wires* uses the *assign* primitive outside an *always* block
  - **assign c = a & b;**
  - c is continuously updated as a and b changes
- Continuous assignment can also be made inside an *always_comb* block
  - **always_comb c = a & b;**

Example:

```
module t1 (port declarations);
logic c, a, b;
assign c = a & b;
end module
```

Slide 11-8:

always_latch

- *always_latch* procedural block is used to indicate the intent to model synthesizable latched logic
- Triggered once at time 0 like *always_comb* and infers sensitivity list using the same rules as *always_comb*
- Similar to *always_comb* except that the software performs different checks on the code for latched logic (not combinational logic)
- Example

```
always_comb
    if (ck) q = d;
```

Software warning: Latch will be inferred (when the intent is combinational logic)

The correct way to model the combinational logic would be to include an *else* branch to update *q* for all conditions of *ck*.

```
always_latch
    if (ck) q <= d;
```

No software warnings will be issued

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-9:

always_ff

- *always*_ff procedural block is used to indicate the intent to model synthesizable sequential logic
- A sensitivity list must be specified and is not inferred
- Every signal in the sensitivity list must be qualified with either *posedge* or *negedge*
- Prohibits using event controls anywhere except at the beginning of the procedural block
- Variables being assigned in an *always_ff* procedure, including variables within called functions, cannot be written to by other procedures
- Example

```
always_ff @(posedge clk )
        if      (rst)  q <= 0;
        else           q <= d;
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 11-10:

Final Procedure

- A *final* procedure is a procedural block of statements like *initial* except that it occurs at the end of simulation time and executes without delays
- Allows only statements that execute in a single simulation cycle
- Not synthesizable
- Typically used to display statistical information about the simulation
- Example

```
final
  begin
      $display ("Number of cycles executed %d", $time/period);
      $display ("Final count = %d", cnt);
  end
```

Note that when *$finish* is called, the *final* procedure will be executed.

Slide 11-11:

Apply Your Knowledge

1) What are the advantages of replacing the generic *always@* procedural block with an *always_comb*, *always_latch* or *always_ff* block?

2) Can *assign* statements be replaced with one of the always blocks? If yes, can all *assign* statements be replaced?

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

A

1) What are the advantages of replacing the generic *always@* procedural block with an *always_comb*, *always_latch* or *always_ff* block?

- Easy to code. *always_comb* and *always_latch* do not require sensitivity list to be coded. They infer them automatically.
- Validates the intention of the designer. The synthesis tool will issue warnings if, for example, *always_latch* is used and no latch is inferred.
- The tool inferring the sensitivity list eliminates the possibility of gate-RTL mismatch, which can occur if a parameter is not included in the sensitivity list.
- Easier for synthesis compilers and formal verification tools to deduce the type of hardware making them more efficient in modeling large and complex designs.
- Greater consistency and accuracy among simulators, synthesis tools, formal checkers, lint checkers, and other software tools.

2) Can *assign* statements be replaced with one of the always blocks? If yes, can all *assign* statements be replaced?

Yes, in general, *assign* statements can be replaced with *always_comb* procedural statements. However, if the left side of the expression is a net type, then it will not be possible to replace the *assign* statement with *always_comb*. The net type will have to be changed to a variable type.

Slide 11-12:

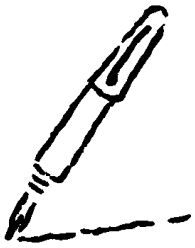
Summary

- *always_comb*, *always_latch*, and *always_ff* are used to create synthesizable procedures for combinational, latched, and sequential logic
- *final* procedure occurs at the end of simulation time and executes without delays

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Control Flow Statements

Slide 12-1:

2021.1

This module describes the different control statements provided by SystemVerilog.
20 minutes.

Slide 12-2:

Objectives

After completing this module, you will be able to:

- Identify the enhancements made to the *for* loop and the new *do...while* and *foreach* loops
- Utilize the new jump statements provided by SystemVerilog
- Illustrate the use of *priority*, *unique*, and *unique0* modifiers in *if* and *case* statements

Slide 12-3:

Looping Constructs in SystemVerilog

- Verilog provides *for*, *while*, *repeat*, and *forever* loops
- SystemVerilog enhances the Verilog *for* loop and adds a *do...while* loop and a *foreach* loop
- All SystemVerilog loops are synthesizable

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-4:

Enhanced for Loops

Verilog *for* Loop	SystemVerilog *for* Loop
Variable used to control a *for* loop must be declared prior to the loop. Example: *reg [15:0] i; for (i = 0; i <= 15; i = i + 1)*	Variable used to control a *for* loop can be declared locally within the *for* loop. Example: *for (reg [15:0] i = 0; i <= 15; i = i + 1)*
When multiple *for* loops are used, a control variable is required for each for loop	The same variable name can be defined locally in more than one *for* loop
The concept of automatic variables doesn't exist	Variables declared as part of *for* loop are automatic variables. They are created and initialized when the *for* loop is invoked and destroyed when the loop exits
Doesn't support multiple initial or step assignments	Supports multiple initial and step assignments. Example: *for (int cnt=1, j=0, done=0; cnt*j < 128; cnt++, j++)*
Synthesizable	Synthesizable

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-5:

The do...while loop

- The *do...while* loop differs from the *while* loop in that it tests the control expression at the end of the loop

- Used to execute a block of statements at least once to save duplication of the loop body

```
initial begin
   for(int i=0; i<4; i++)
      $display("print #%0d from for loop", i);

   ready = 0;
   do begin
      $display("print from do..while loop");
      ready = 1;
   end while (!ready);
end
```

```
print #0 from for loop
print #1 from for loop
print #2 from for loop
print #3 from for loop
print from do..while loop
```

- Synthesizable

Not supported by Verilog.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-6:

The foreach Loop (1)

- The *foreach* loop enables iteration over each element of an array
- The syntax for the argument includes the array name followed by one or more loop variables
- The iteration can be performed over a single or multi-dimensional array using a single or many loop variables
- Example

```
bit ready;
string words[2];

initial begin
   words[0] = "hello";
   words[1] = "foreach loop";
   foreach (words[ j ])
      $write("%s ", words[j]);
   $display("");
end
```

```
hello foreach loop
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-7:

The foreach Loop (2)

- To skip iterating over a dimension of the array, omit the corresponding loop variable
- Example

```
int A [2][3][4];
foreach (A  [i, ,k]);
```

Causes *i* to iterate from 0 to 1 and *k* from 0 to 3 (iteration over second index is skipped)

- Synthesizable

Not supported by Verilog.

Slide 12-8:

New Jump Statements in SystemVerilog

- The jump statements supported
  - *break* // break out of loop
  - *continue* // skip to end of loop and execute the loop control if present
  - *return* expression // exit from a function
  - *return* // exit from a task or void function
- The *continue* and *break* statements can only be used in a loop. They cannot be used inside a fork-join block to control a loop outside the fork-join block
- The *return* statement can only be used in a subroutine
- Synthesizable

© Copyright 2021 Xilinx

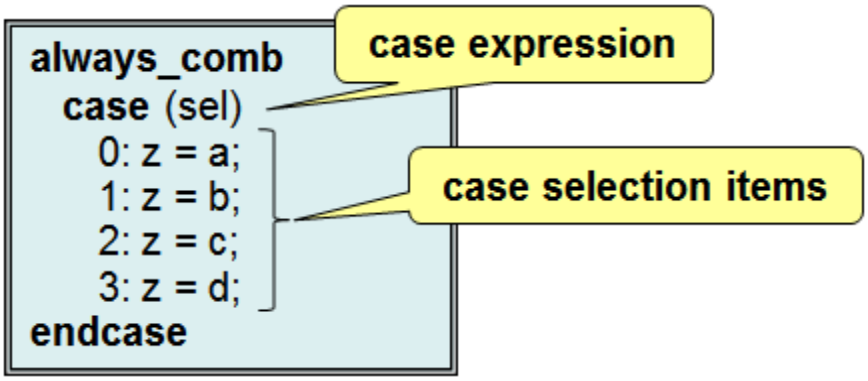
FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-9:

Verilog case Statement

- The Verilog *case* statement is a multi-branch statement that enables selection of one branch based on the evaluation of a given expression

- Example

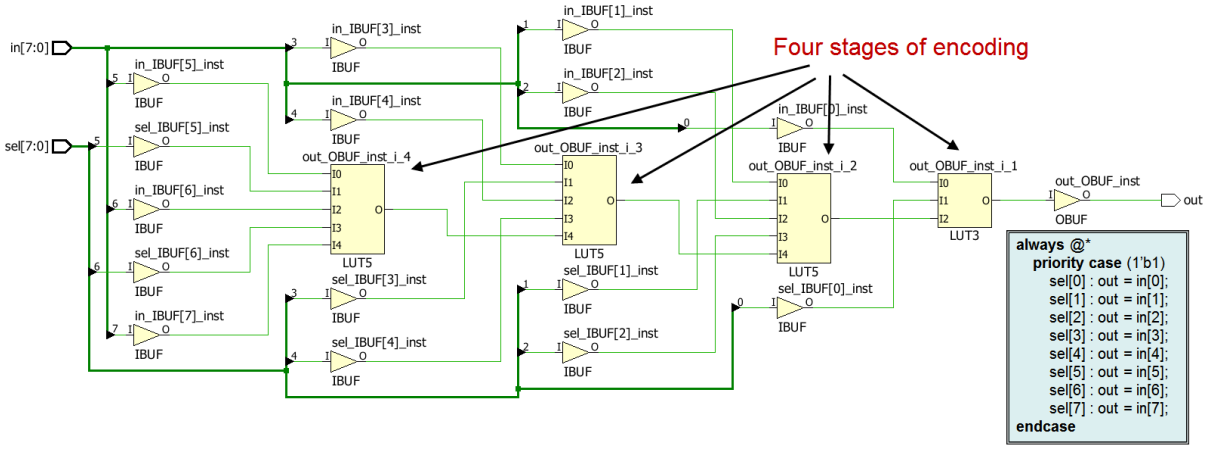


- Each *case* selection item is evaluated in the order in which it is listed; i.e., if there is more than one match, there is a priority

- Due to the priority, the simulator and synthesis tools can interpret the Verilog *case* statement differently
  - Simulator may interpret the *case* statement as a series of *if...else...if* statements
  - Synthesis tool may remove the logic for priority encoding if the selection items are mutually exclusive

Slide 12-10:

Priority Case Example



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-11:

Priority, Unique0, and Unique Keywords

- *priority*, *unique0*, and *unique* keywords are intended to notify simulation and synthesis tools how to interpret *case* statements

Keyword	Only One Match	More Than One Match	No Match
case	✓	✓	✓
priority	✓	✓	x
unique0	✓	x	✓
unique	✓	x	x

- These keywords can also be used with *if-else* statements

priority notifies that one or more match can be present; i.e., priority of *case* items matter (first matching item will be taken).

unique0 notifies that only one match or no match can be present; i.e., priority does not matter.

unique notifies that only one match can be present; i.e., priority does not matter.

When *priority* or *unique* keywords are used, the simulator would issue an error/warning when unlisted *case* conditions are observed.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-12:

Logic for Priority, Unique0, and Unique Keywords

Example: *always @** *case (1'b1)* *a:* ... *b:* ... *c:* ... *d:* ... *endcase*	**case**	**priority**	**unique0**	**unique**
	a: !a && b: !a && !b && c: !a && !b && !c && !d:	a: !a && b: !a && !b && c:	a: b: c: d:	a: b: c:
Synthesizable	Yes	Yes	No	Yes
Supported by Verilog	Yes	No	No	No

- *unique0* and *unique* cases remove any logic for implementing priority of *case* items
- *priority* and *unique* cases optimize out the last *case* item because at least one match is implied

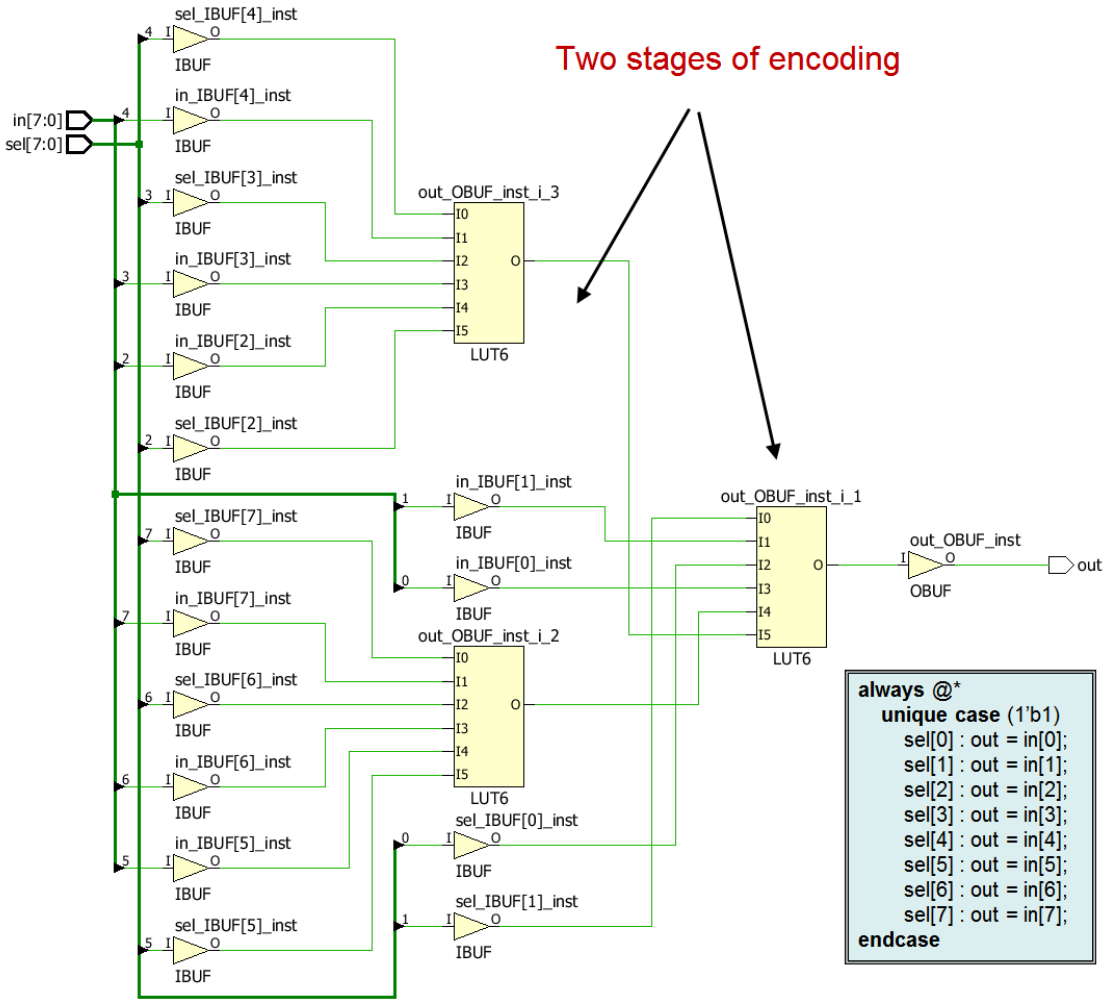
When the *priority* keyword is used, the synthesis tool adds logic for priority encoding.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-13:

Unique Case Example



Slide 12-14:

Apply Your Knowledge

1) List some of the SystemVerilog enhancements to Verilog *for* loops

2) Which are the additional looping constructs provided by SystemVerilog?

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

1) List some of the SystemVerilog enhancements to Verilog *for* loops.

- The variable used to control a *for* loop can be declared locally within the *for* loop.
  - Example: *for (reg [15:0] i = 0; i <= 15; i = i + 1)*
- The same variable name can be defined locally in more than one *for* loop
- Variables declared as part of a *for* loop are automatic variables. They are created and initialized when the *for* loop is invoked and destroyed when the loop exits.
- Supports multiple initial and step assignments.
  - Example: *for (int x=1, y=1; x*y < 9; x++, y++)*

2) Which are the additional looping constructs provided by SystemVerilog?

do..while and *foreach* loops.

Slide 12-15:

Apply Your Knowledge

1) What is the difference between a *priority* decision statement and a *unique* decision statement?

1) What is the difference between a *priority* decision statement and a *unique* decision statement?

The *priority* keyword implies that more than one match can be present and hence the priority of the case selection items is important.

The *unique* keyword implies that only one match can be present and hence the order of the case selection items is irrelevant.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 12-16:

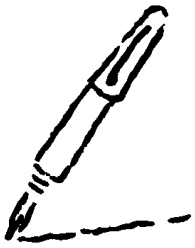
Summary

- SystemVerilog enhances the Verilog *for* loop and adds a *do...while* loop and a *foreach* loop
- C-like jump statements (*break*, *continue*, and *return*) are provided by SystemVerilog
- *priority*, *unique*, and *unique0* modifiers in *if* and *case* statements improve consistency between simulators and synthesis tools

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Functions

Slide 13-1:

2021.1

This module explains the SystemVerilog enhancements to functions. 20 minutes.

Slide 13-2:

Objectives

After completing this module, you will be able to:

- Explain what functions are
- Describe the enhancements added in SystemVerilog for functions
- Illustrate the concept of passing arguments by reference in functions

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-3:

Functions

- A function is a type of subprogram
- Avoids code repetition when the function needs to be called multiple times
- Statements in a function shall execute in one simulation time unit
- A function cannot enable a task
- Functions (except a void function) shall return a single value
- A non-void function can be used as an operand whose value is the value returned by the function

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-4:

Function Example

```
module  function_swap;

  function [7:0] reverse_nibble;
    input [7:0] inp_byte;
    logic [3:0] temp1;
    logic [3:0] temp2;

    for (int each_bit=0; each_bit<=3; each_bit++)
    begin
      temp1[each_bit]=inp_byte[(0+each_bit)];
      temp2[each_bit]=inp_byte[(4+each_bit)];
    end
  endfunction

  logic [7:0] given_byte;
  logic [7:0] new_byte;

  initial begin
    given_byte=8`b11001010;
    new_byte=reverse_nibble(given_byte);
    $display("%b has been reverse as %b", given_byte, new_byte);
  end

endmodule
```

```
11001010 has been reversed as 10101100
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-5:

SystemVerilog Enhancements to Functions (1)

Verilog	SystemVerilog
Multiple statements within a function or task must be grouped using begin...end or (for tasks only) using fork...join	Does not require begin...end grouping for multiple statements
A function exits only when the execution flow reaches end	Supports break, continue, return, and return value commands
Functions must have a return value	Functions can be declared as void functions. Example: *function void* my_func(*input int* a, …); … … *endfunction*
Values of arguments can be passed to a task or function only in the same order in which they were defined in the task or function	Includes the ability to pass values using the name of the arguments. Example: *function* math_func(*input int* num1, num2); … … *endfunction* answer=math_func(.num1(a),.num2(b));

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-6:

SystemVerilog Enhancements to Functions (2)

Verilog	SystemVerilog
Functions can only have inputs	Functions can have arguments declared as input, output, or inout (like tasks)
A function must have at least one input argument	A function can have zero or more arguments of any type (like tasks)
Each argument must have a direction (i.e., input, output, or inout)	Arguments are considered as input by default
Requires that functions contain at least one statement	Allows functions to be empty with no statements

The constraint to a function in SystemVerilog with output or inout arguments is that it cannot be called from:

- Event expression
- Expression within a procedural continuous assignment
- Expression that is not within a procedural statement

Slide 13-7:

Types of Functions

- Three types of functions: static, automatic, and constant functions
- Static is the default type

The static and automatic types apply to variables declared within functions, tasks, *begin...end*, and *fork...join* blocks. Such variables can be explicitly declared using a *static* or *automatic* keyword. Variables at the module level cannot be declared as *static* or *automatic*. All variables at module level are static.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

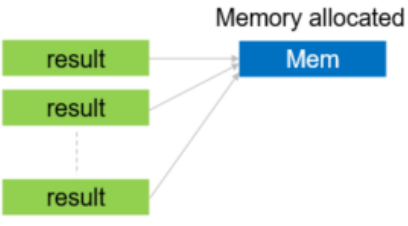
Slide 13-8:

Static Functions

- Static functions are functions defined within a module, interface, program, or package, with all declared items being statically allocated
- A static function has a single memory region, which is updated within each function call
- Functions can be defined to use static storage by explicitly defining the function as *static*
- Static functions are not synthesizable

```
module sum;
  function int cumulative_sum(input int n);
    int result=0;
    result+=n;
    return result;
  endfunction

  initial begin
    for(int i=0;i<10;i++)
      $display("i=%0d, cumulative_sum = %0d",i,cumulative_sum(i));
  end

endmodule
```

```
i=0, cumulative_sum=0
i=1, cumulative_sum=1
i=2, cumulative_sum=3
i=3, cumulative_sum=6
i=4, cumulative_sum=10
i=5, cumulative_sum=15
i=6, cumulative_sum=21
i=7, cumulative_sum=28
i=8, cumulative_sum=36
i=9, cumulative_sum=45
```



A function's internal signals cannot be accessed when the function is not running. A function's internal signals cannot be accessed from outside the function itself.

A static function is updated every time it is called.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

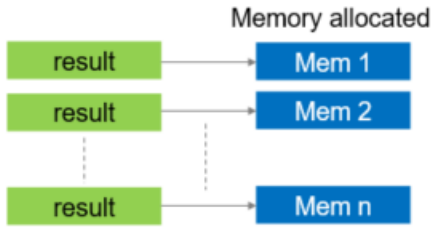
Slide 13-9:

Automatic Functions

- Automatic functions are functions that use items that are dynamically allocated by the software when required and deallocated when no longer needed
- Each time the function is called, new storage is created and when the call exists, the storage is destroyed
- The keyword *automatic* declares an automatic function

```
module sum;
  function automatic int cumulative_sum(input int n);
    int result=0;
    result+=n;
    return result;
  endfunction

  initial begin
    for(int i=0;i<10;i++)
      $display("i=%0d, cumulative_sum = %0d",i,cumulative_sum(i));
  end

endmodule
```

```
i=0, cumulative_sum=0
i=1, cumulative_sum=1
i=2, cumulative_sum=2
i=3, cumulative_sum=3
i=4, cumulative_sum=4
i=5, cumulative_sum=5
i=6, cumulative_sum=6
i=7, cumulative_sum=7
i=8, cumulative_sum=8
i=9, cumulative_sum=9
```



Unlike a static function, an automatic function creates new storage every time it can hold its previous value, which makes it useful for recursive functions.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-10:

Constant Functions (1)

- Subset of functions that are evaluated at elaboration time to support the building of a complex calculation
- Defined locally in the module calling it
- All arguments in a constant function call shall be constant expressions

Constant functions have the following restrictions. They cannot:

- Have hierarchical references
- Have parameters affected by *defaparam* statements
- Have *output*, *inout*, or *ref* arguments
- Invoke a function that is not a constant function and this is not local to the module

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-11:

Constant Functions (2)

```
module ram #(
   parameter WIDTH = 32,
   parameter DEPTH = 256
)(
   input clock,
   input [log2i(DEPTH)-1:0] addr,
   input [WIDTH-1:0] data_in,
   input write,
   input enable,
   output reg [WIDTH-1:0] data_out
);

function integer log2i( input [31:0] value);
begin
   value = value - 1;
   for (log2i = 0; value > 0; log2i = log2i + 1)
      value = value >> 1;

   $display("You will never see this print!!!");
end
endfunction

// ... memory implementation

endmodule
```

System tasks are ignored, when constant function is called during elaboration phase

In the example, the function log2i is a constant function that will be evaluated during elaboration phase based on the depth of the RAM (default depth being 256).

It is during elaboration that all parameters will propagate and the size of all signals will be fixed. Hence, the actual size of RAM will be fixed during elaboration.

RAM module parameters shall not be specified using defparam, but using 1364-2001 syntax.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-12:

Passing Arguments by Reference

- Arguments to a function can be passed by reference in SystemVerilog by declaring them with the keyword *ref*
- Arguments passed by reference are not copied into the function area. Instead, a reference to the argument is passed to the function
- The function can then access the argument value via the reference

Passing arguments by reference is similar to external referencing in Verilog. The difference being that in external referencing, the external signal name must be hard coded into the *function*, which limits the ability to code a general-purpose function.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 13-13:

Passing Arguments by Reference Example

```
int data[];

initial begin
        init_array(data);
        for (int i=0; i<data.size(); i++) begin
                $display("Data %d ", data[i]);
            end

        init_array(data, 2);
        for (int i=0; i<data.size(); i++) begin
                $display("Data %d ", data[i]);
            end
        end

function automatic void init_array(ref int arr[], input int fill_with=1);
        arr = new[5];
        for (int i=0; i<data.size(); i++) begin
                arr[i]=fill_with;
                $display("Arr %d ", arr[i]);
                end
endfunction
```

```
1 1 1 1 1
2 2 2 2 2
```

Output of first call and second call to init_array()

```
1 1 1 1 1
2 2 2 2 2
```

Pass by reference: reference to the original argument is passed to the task/function. Both the caller and the subroutine share the same variable of the argument. Any changes to the argument will be visible to each other.

Slide 13-14:

Apply Your Knowledge

1) True or false: Verilog allows values to be passed to a function in a different order compared to the order in which they were defined in the function

2) True or false: SystemVerilog supports void functions. Verilog does not

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

1) True or false: Verilog allows values to be passed to a function in a different order compared to the order in which they were defined in the function.

   False. Verilog does not allow this but SystemVerilog does allows this.

2) True or false: SystemVerilog supports void functions. Verilog does not.

   True.

Slide 13-15:

Apply Your Knowledge

1) By default, are functions static or automatic?

2) What are constant functions used for?

1) By default, are functions static or automatic?

   Static.

2) What are constant functions used for?

   Constant function calls are used to support the building of complex calculations of values at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

Slide 13-16:

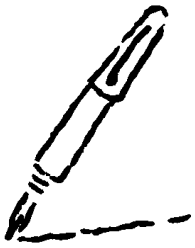
Summary

- Like subroutines, functions provide the ability to execute common procedures at several different places
- SystemVerilog has added several enhancements to functions
- The three types of functions are static, automatic, and constant
- Passing arguments by reference allows signals in functions to be read and modified without having to copy them in and out of the function

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Tasks

Slide 14-1:

2021.1

This module describes the task SystemVerilog construct. 15 minutes.

Slide 14-2:

Objectives

After completing this module, you will be able to:

- Define a task in SystemVerilog
- Describe the enhancements added in SystemVerilog for tasks
- Explain the different types of tasks
- Illustrate the concept of passing arguments by reference in tasks

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-3:

Tasks

- A task is a type of subprogram
- Avoids code repetition by calling the task again and again
- Tasks can be declared with or without arguments
  - Passing arguments allows greater utilization throughout the design
- Tasks cannot return any value
- Tasks may or may not include timing delays
- Tasks execute in non-zero simulation time

Slide 14-4:

Task Example

```
module task_counter;

  integer counter;

  task count;
    input [31:0] inc_by;
    output [31:0] next_val;
    integer cnt;
  begin
    cnt = cnt + inc_by;
    next_val = cnt;
  end
  endtask

  initial count.cnt = 0;
  always #5 count (1, counter);

  always @(counter)
    $display("%2t: Counter's value is now set to %0d", $time, counter);

  initial #30 $finish;

endmodule
```

```
5: Counter's value is now set to 1
10: Counter's value is now set to 2
15: Counter's value is now set to 3
20: Counter's value is now set to 4
25: Counter's value is now set to 5
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-5:

SystemVerilog Enhancements to Tasks

Verilog	SystemVerilog
Multiple statements within a task must be grouped using begin...end or (for tasks only) using fork...join	Does not require begin...end grouping for multiple statements
A task exits only when the execution flow reaches end	Supports break, continue, return, and return value commands
Values of arguments can be passed to a task only in the same order in which they were defined in the task	Includes the ability to pass values using the name of the arguments
Requires that tasks contain at least one statement	Allows tasks to be empty with no statements

Slide 14-6:

Types of Tasks

- Tasks are of two types: static and automatic
- Static is the default type for tasks

The static and automatic types apply to variables declared within tasks, *begin...end*, and *fork...join* blocks. Such variables can be explicitly declared using a *static* or *automatic* keyword. Variables at the module level cannot be declared as *static* or *automatic*. All variables at module level are static.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-7:

Static Tasks

- Static tasks are tasks defined within a module, interface, program, or package, with all declared items being statically allocated
- Tasks can be defined to use static storage by explicitly declaring the task as *static*
- Multiple invocation of same task will refer the same local variable
- Static tasks are not synthesizable

Changing a value in one invocation will make it visible to other invocations also.

Slide 14-8:

Automatic Tasks

- Automatic tasks allocate all the declared items dynamically by the software when required and deallocates them when no longer needed
- The keyword *automatic* declares an automatic tasks
- Multiple invocations of same task will have local variables unique to each invocation of the task
- Automatic tasks are re-entrant tasks

Changing a value in one invocation will not affect the other invocations.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-9:

Passing Arguments by Reference (1)

- Arguments to a task can be passed by reference in SystemVerilog by declaring them with the keyword *ref*
- Arguments passed by reference are not copied into the task area. Instead, a reference to the argument is passed to the task
- The task can then access the argument value via the reference

Passing arguments by reference is similar to external referencing in Verilog. The difference being that in external referencing, the external signal name must be hard coded into the *task*, which limits the ability to code a general-purpose task.

Slide 14-10:

Passing Arguments by Reference (2)

- Only automatic tasks can pass arguments by reference
- Advantages
  - Used to break a complex procedural block into smaller, structured coding blocks
  - Allows the task to sense when the external signal changes value
  - The changes made within the task can be immediately sensed outside of the task
- Synthesizable
- Not supported by Verilog

```
module passing_args;
int a;
   initial
     begin
      #10 a = 10;
      #10 a = 20;
      #10 a = 30;
      #10 $finish;
     end

   task automatic pass_by_val(int x);
    forever
        @x $display("pass_by_val: X is %0d",x);
   endtask

   task  automatic pass_by_ref(ref int x);
    forever
        @x $display("pass_by_ref: X is %0d",x);
   endtask

   initial
     pass_by_val(a);
   initial
     pass_by_ref(a);
endmodule
```

```
pass_by_ref: X is 10
pass_by_ref: X is 20
pass_by_ref: X is 30
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-11:

Apply Your Knowledge

1) True or false: Verilog allows values to be passed to a task in a different order compared to the order in which they were defined in the task

2) By default, tasks are static or automatic?

3) What are the advantages of passing arguments to a task by reference?

1) True or false: Verilog allows values to be passed to a task in a different order compared to the order in which they were defined in the task.
False. Verilog does not allow this but SystemVerilog does allows this.

2) By default, tasks are static or automatic?
Static.

3) What are the advantages of passing arguments to a task by reference?
   - Used to break a complex procedural block into smaller, structured coding blocks.
   - Allows the task to sense when the external signal changes value.
   - The changes made within the task can be immediately sensed outside of the task.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 14-12:

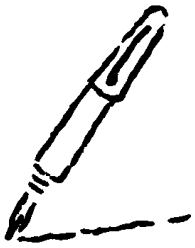
Summary

- Like subroutines, tasks provide the ability to execute common procedures at several different places
- SystemVerilog has added several enhancements to tasks
- The two types of tasks are static and automatic
- Passing arguments by reference allows signals in tasks to be read and modified without having to copy them in and out of the task

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Packages

Slide 15-1:

2021.1

This module describes the package SystemVerilog construct. 10 minutes.

Slide 15-2:

Objectives

After completing this module, you will be able to:

- Describe packages and their application in code reuse

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 15-3:

Packages

- A package is a collection of types (structs, enumerated), parameters, localparams, functions and tasks, which are commonly used in entire design
- A package is a separate declaration space. It is not embedded within a module
- An important use of packages is code reuse
- Packages are synthesizable

```
package ComplexPkg;
typedef struct {
   float i, r;
} Complex;

function Complex add(Complex a, b);
   add.r = a.r + b.r;
   add.i = a.i + b.i;
endfunction

function Complex mul(Complex a, b);
   mul.r = (a.r * b.r) - (a.i * b.i);
   mul.i = (a.r * b.i) + (a.i * b.r);
endfunction

endpackage : ComplexPkg
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 15-4:

Referencing Data in Packages

- One way to use declarations made in a package is to reference them using the class scope resolution operator

```
ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

- An alternate method for utilizing package declarations is via the import statement

```
import ComplexPkg::Complex;
import ComplexPkg::mul;
// …
Complex c = mul(1.0,3.4);
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 15-5:

Package Example

```
package my_package;

typedef enum {Red, Green, Blue} Colors;

typedef struct {
   int field_a;
   Colors c;
} MyStruct;

endpackage

module package_example;
import my_package::*;

MyStruct s;

initial begin
   s.field_a = 5;
   s.c = Red;

   #1 $display("s.field_a = %0d, s.c = %s", s.field_a, s.c.name());
end

endmodule
```

Package definition

Import all symbols

When package items are imported using a wildcard, only items actually used in the module or interface are actually imported. Package items that are not referenced are not imported.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 15-6:

Compilation Unit Package ($unit)

- A compilation unit refers to a set of files compiled together
- A compilation unit contains all declarations that lie outside of any scope
- $unit represents the top level of each compilation unit
- Any declaration outside the named declaration space is defined in $unit package

Example 1

```
typedef enum bit {FALSE, TRUE} decision;
const integer temp = 1;

module test;

decision d;
const integer temp = 2;

initial
$display( "temp=%0d    $unit::temp=%0d", i, $unit::i );
endmodule
```

In the above example, decision and temp are declared outside the module test and therefore are in the $unit declaration space.

The constant temp is declared outside the module test. Another constant named temp is again declared inside the module test. The declaration inside the module scope hides the declaration at the compilation unit scope.

However, the declaration at the compilation unit scope can still be accessed with the $unit:: prefix as shown in the code above. The example code, when simulated, would print temp=2 $unit::temp=1.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Example 2

Suppose there are two files named m1.sv and m2.sv:

- If these two files are compiled by passing both the file names together as parameter to the compilation command, then the compilation unit comprises both the files and the compilation unit scope ($unit::) is shared between the two files. In this case, there will be a single compilation unit.
- If these two files are compiled separately by passing each file name one at a time to the compilation command, then each file will have a separate compilation unit and scope. In this case, there will be two compilation units.

Slide 15-7:

Apply Your Knowledge

1) What is the primary purpose of using packages?

2) True or false: SystemVerilog packages are leveraged from the VHDL language

1) What is the main purpose of using packages?
Packages enable code reuse by enabling sharing of a user-defined type definition across multiple modules.

2) True or false: SystemVerilog packages are leveraged from the VHDL language
True.

Slide 15-8:

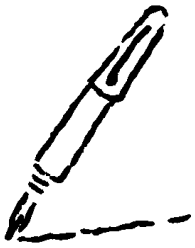
Summary

- Packages provide a mechanism for sharing types, parameters, data, tasks, and functions among multiple SystemVerilog modules and interfaces

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Interfaces

Slide 16-1:

2021.1

This module describes the concept of interfaces in SystemVerilog. 25 minutes.

Slide 16-2:

Objectives

After completing this module, you will be able to:

- Define interfaces and what they are used for
- Create SystemVerilog code that uses interfaces
- Describe interface modports
- Explain how functions and tasks can be used within interfaces to create a common protocol

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-3:

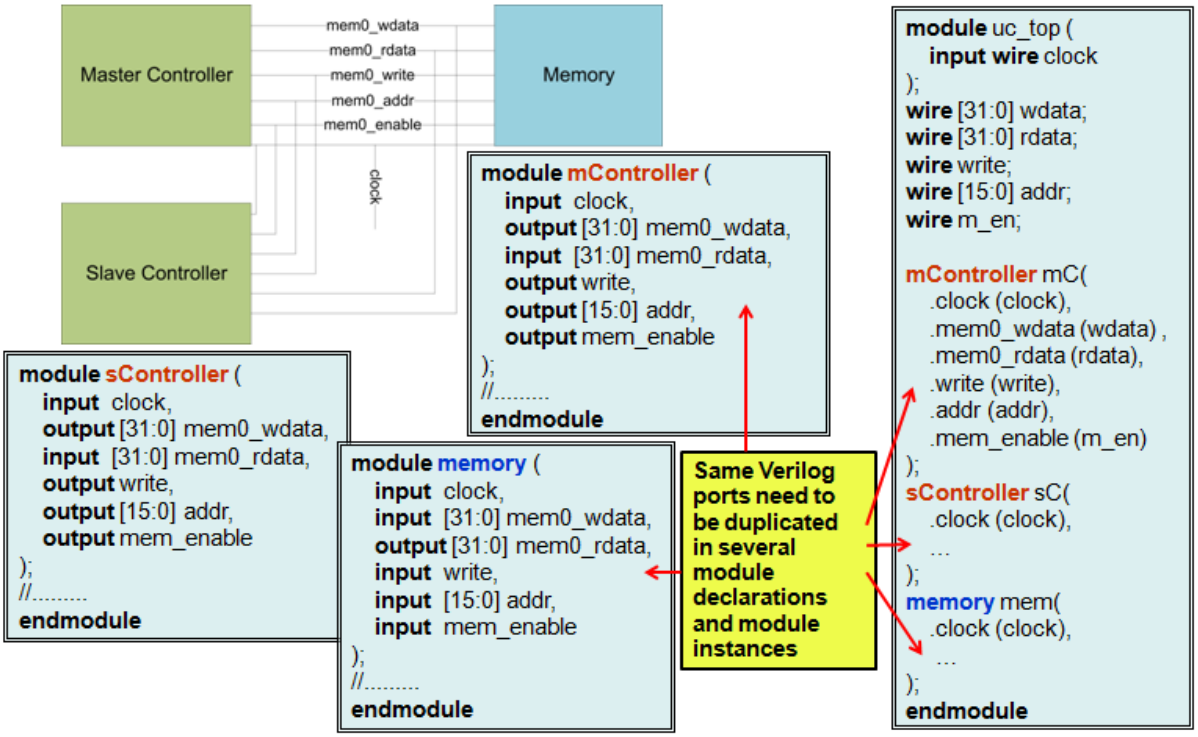
Interface

- Interface represents a bundle of nets or variables
- Interface encapsulates functionality and connectivity between modules
- Interface can be instantiated like a module, and can be accessed through ports like a signal

```
interface interface_name (arguments);
<interface_items>
endinterface
```

Slide 16-4:

Verilog Module Ports



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Verilog's module ports provide a simple and intuitive way of connecting blocks of a design.

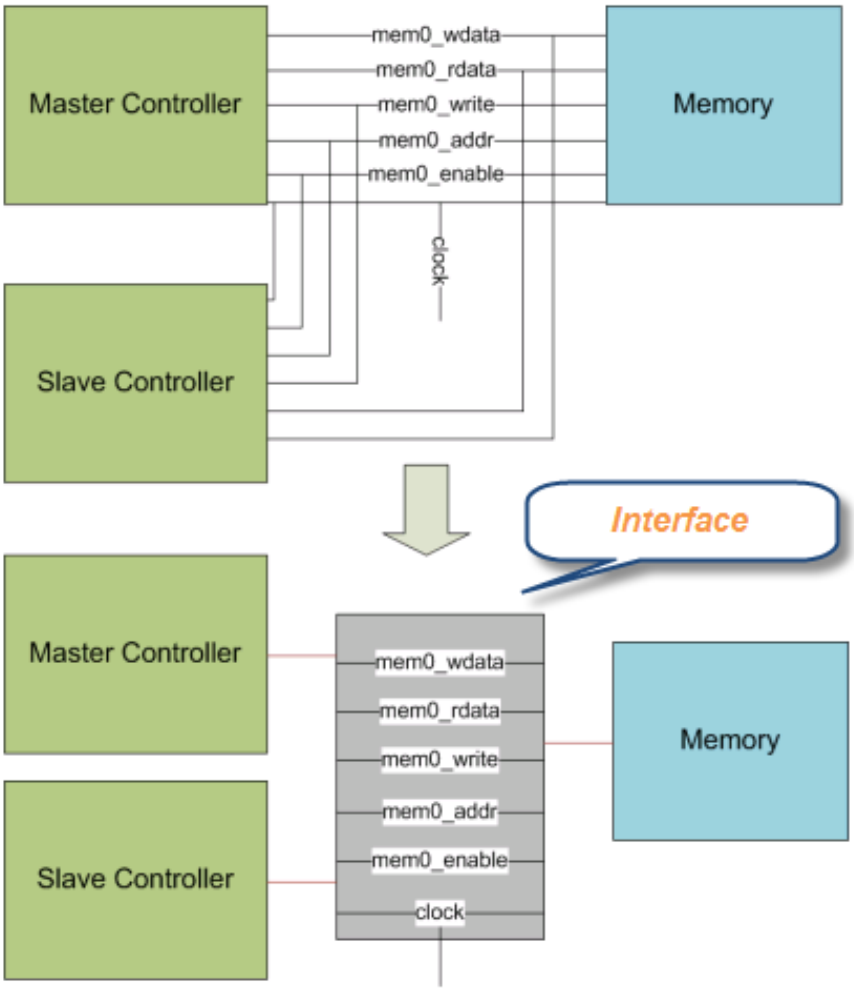
However, for large designs, using module ports has the following disadvantages:

- Declaring the same ports in several modules, resulting in verbose code and possibility of mismatched declarations
- Determining all aspects of the ports and interconnects even at an abstract level
- Duplication of code corresponding to the same protocol across different modules. Example: read and write control logic to a shared memory
- Updating several lines of code for a minor change in the design specification

Slide 16-5:

Interfaces Definition

- An interface is an abstract port type that allows many signals to be grouped together and represented as a single port
- Interfaces are synthesizable

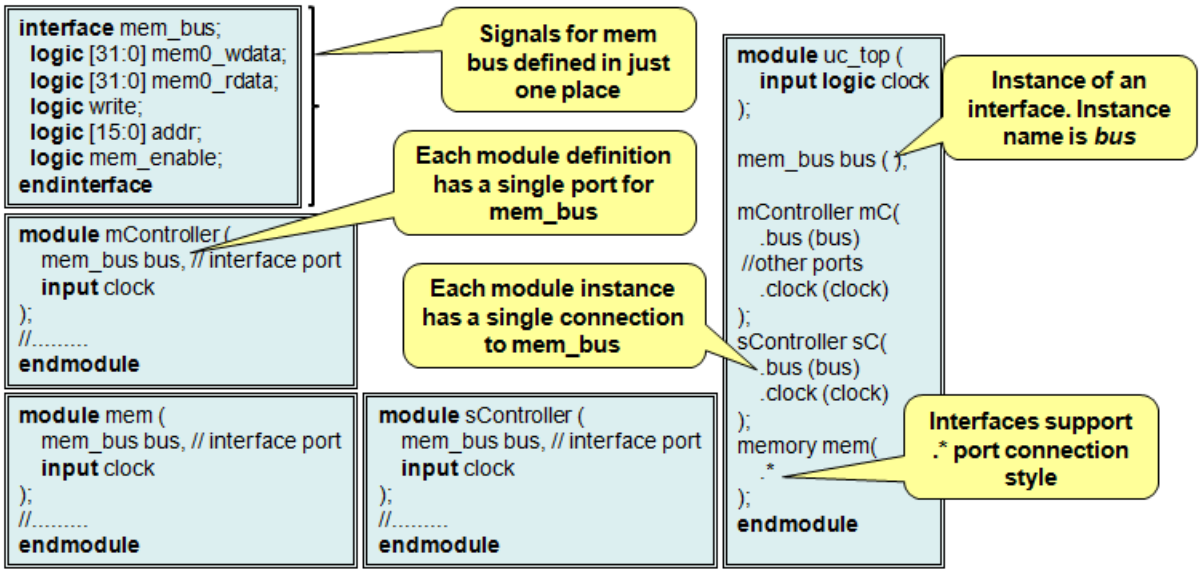


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-6:

Interface Declarations



The declarations of the signals that make up the interface are contained in a single location.

Each module that uses the signals declared in the interface has a single port of the interface type instead of many ports with the discrete signals.

Signals in an interface are referenced using the port name and the internal interface signal name. For example, the *mem* module can use *always@ (bus.clock) begin* ...

Interfaces support *.name* and .* port connection styles, which further simplify port connections.

Note: Interfaces having variable declarations are treated as *ref* ports when accessed by another module. This may not be supported by all synthesis tools. (In such cases, it is better to use the *net* type as it will result in inout.)

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-7:

Interface Declarations with Ports

- An interface can have ports (like modules) to make signals external to the interface become part of the interface

```
interface mem_bus (
   input logic clock
);
  logic [31:0] mem0_wdata;
  logic [31:0] mem0_rdata;
  logic write;
  logic [15:0] addr;
  logic mem_enable;
endinterface
```

Use interface ports to bring in discrete external signals in to the interface

Instance of an interface with discrete signal connected to the interface

```
module uc_top (
   input logic clock
);

mem_bus bus (
   .clock (clock)
);

mController mC(
   .bus (bus)
);
sController sC(
   .bus (bus)
);
memory mem(
   .*
);
endmodule
```

Discrete signals don't have to be connected to each module declaration or instance if they are part of the interface

```
module mController (
   mem_bus bus // interface port
);
//.........
endmodule
```

```
module sController (
   mem_bus bus // interface port
);
//.........
endmodule
```

```
module mem (
   mem_bus bus // interface port
);
//.........
endmodule
```

All ports of an interface must be connected unlike modules whose ports can be left unconnected.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-8:

Explicitly Named and Generic Interfaces

- There are two ways to declare an interface as a module port
  - Explicitly naming the interface port and
  - Using a generic interface port

Explicitly Named Interface Ports	Generic Interface Ports
Uses the name of an interface as the port type	Uses to the keyword interface in the place of an interface name
Can only be connected to an interface of the same name	Any interface can be connected to the generic interface port
Ensures that a wrong interface cannot be inadvertently connected to the port	Provides flexibility in connecting the module

Slide 16-9:

Explicitly Named and Generic Interfaces Example

```
interface mem_bus (
   input logic clock
);
  logic [31:0] mem0_wdata;
  logic [31:0] mem0_rdata;
  logic write;
  logic [15:0] addr;
  logic mem_enable;
endinterface
```

```
module mController (
   mem_bus bus, // interface port
   input reset_n
);
//.........
endmodule
```

Explicitly named interface port

```
module mController_gen (
   interface bus, // interface port
   input reset_n
);
//.........
endmodule
```

Generic interface port

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-10:

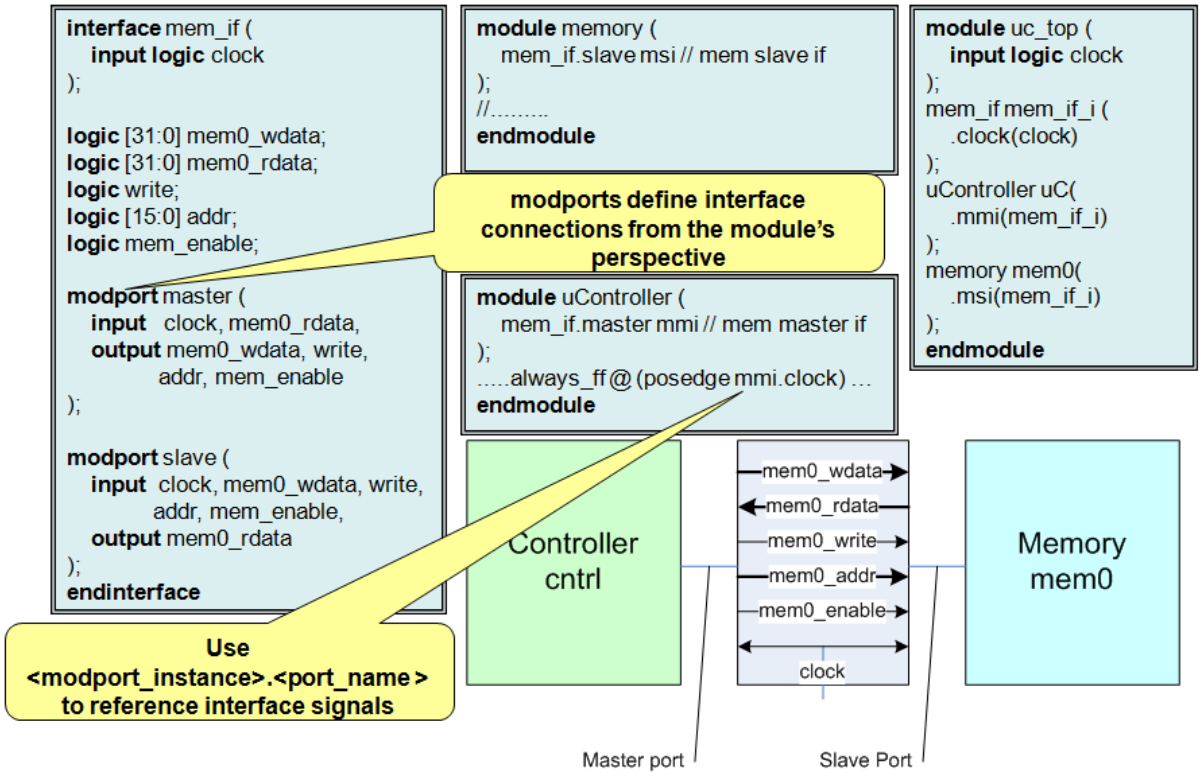
Interface Modports: The Concept

- Modports provide the ability to use the interface signals differently at each module interface port
- Without modports, all nets in the interface are considered bidirectional (*inout* type) and all variables are considered as references (*ref*)
- A modport defines the port direction that the module sees for the signals in the interface
- Modports are defined within the interface declaration

It is possible for modports to omit a few signals declared in the interface. This makes it possible to hide some signals within the interface from certain modules.

Slide 16-11:

Interface Modports



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-12:

Using Modports

- SystemVerilog provides two methods for specifying which modport a module interface port should use
  - In the module instance, as part of the interface connection. The syntax is *<interface_instance_name>.<modport_name>*
  - In the module definition, as part of the module port declaration. The syntax is *<interface_name>.<modport_name>*

```
interface mem_if (
   input logic clock
);

logic [31:0] mem0_wdata;
logic [31:0] mem0_rdata;
logic write;
logic [15:0] addr;
logic mem_enable;

modport master (
   input  clock, mem0_rdata,
   output mem0_wdata, write,
          addr, mem_enable
);

modport slave (
   input  clock, mem0_wdata, write,
          addr, mem_enable,
   output mem0_rdata
);
endinterface
```

```
module memory (
   mem_if.slave msi // mem slave if
);
//.........
endmodule
```

```
module uController (
   mem_if.master mmi // mem master if
);
//.........
endmodule
```

```
module uc_top (
   input logic clock
);
mem_if mem_if_i (
   .clock(clock)
);
uController uC(
   .mmi(mem_if_i)
);
memory mem0(
   .msi(mem_if_i)
);
endmodule
```

Selecting modport in the module declaration

Selecting modport in the module instance

```
module memory (
   mem_if msi // mem slave if
);
//.........
endmodule
```

```
module uController (
   mem_if mmi // mem master if
);
//.........
endmodule
```

```
module uc_top (
   input logic clock
);
mem_if mem_if_i (
   .clock(clock)
);
uController uC(
   .mmi(mem_if_i.master)
);
memory mem0(
   .msi(mem_if_i.slave)
);
endmodule
```

A modport can be selected in either the module instance or the module definition, but not both.

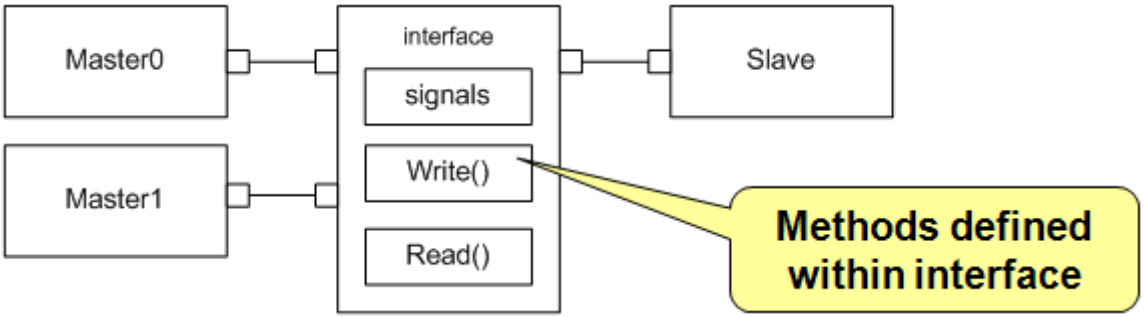
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-13:

Interface Methods

- In addition to encapsulating a collection of signals, an interface can also include communication protocols (collections of methods) to transfer data from one module to another
- These methods are defined using functions or tasks
- The methods can be placed in the interface itself so that other modules can import them from the interface



Besides functions and tasks, interfaces also contain procedural blocks (*always*, *always_comb*, *always_latch*, *always_ff*, *initial*, or *final* blocks) and continuous assignments (*assign* statements). This is very useful in verification where building protocol checkers in the interface becomes possible.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-14:

Importing Interface Methods

- In the example below, tasks that are defined in the same module as the interface are imported in to the modport

- When a task is imported, the port direction will be inferred from the task. Hence, there is no need to define the direction of the ports in the modport

```
interface if_method
( input  clock );

logic [31:0] wdata;
logic [15:0] addr;
logic enable;
logic write;
logic [31:0] rdata;
logic locked;
```

```
modport master (
   output wdata, addr,
          enable, write,
   input  clock, rdata,
   import write_task, read_task
);
modport slave (
   input  clock, wdata, addr,
          enable, write,
   output rdata
);
endinterface : if_method
```

```
task automatic write_task (
   input [15:0] a,
   input [31:0] d
);
   do @(posedge clock);
   while (locked);
   locked = 1;
   wdata <= d;
   addr <= a;
   enable <= 1;
   write <= 1;
   @(posedge clock);
   wdata <= 'bx;
   addr <= 'bx;
   enable <= 'b0;
   write <= 'bx;
   locked = 0;
endtask
```

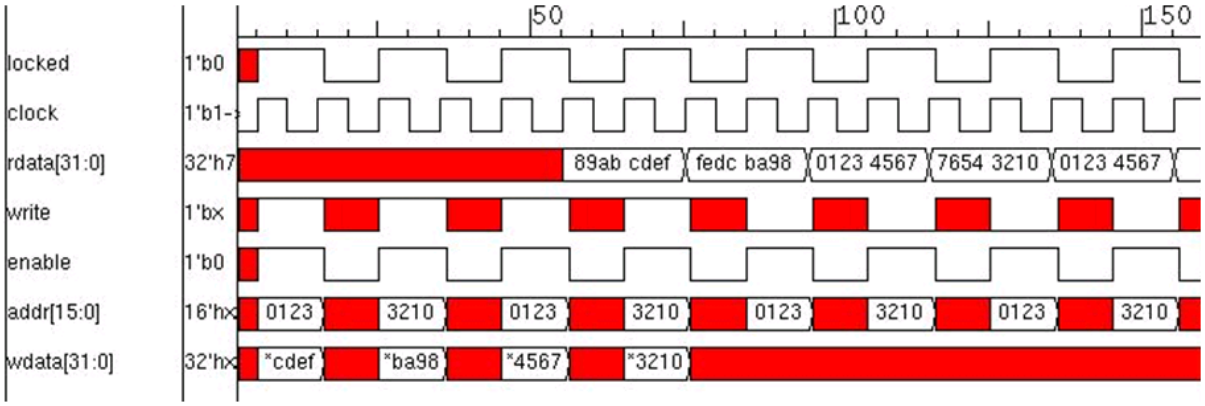
```
task automatic read_task (
   input  [15:0] a,
   output [31:0] d
);
   do @(posedge clock);
   while (locked);
   locked = 1;

   addr <= a;
   enable <= 1;
   write <= 0;
   @(posedge clock);
   d = rdata;
   addr <= 'bx;
   enable <= 'b0;
   write <= 'bx;
   locked = 0;
endtask
```


© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-15:

Calling Imported Methods




```
module master0 (if_method.master mi);
//......
initial begin
   mi.write_task(16'h0123, 32'h01234567);
   mi.write_task(16'h3210, 32'h76543210);
   mi.read_task(16'h0123, rdata);
   mi.read_task(16'h3210, rdata);
end
//......
```

```
module master1 (if_method.master mi);
//......
initial begin
   mi.write_task(16'h0123, 32'h89ABCDEF);
   mi.write_task(16'h3210, 32'hFEDCBA98);
   mi.read_task(16'h0123, rdata);
   mi.read_task(16'h3210, rdata);
end
//......
```

Modports enable use of different methods and protocols within the same interface. The interface can contain a variety of different methods, each using different protocols or types. For example, different read/write methods can be defined to access data in different formats.

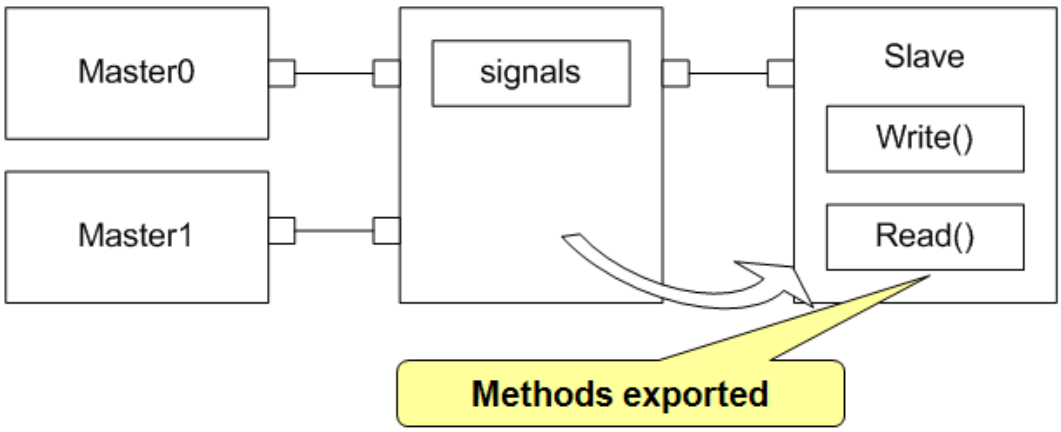
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-16:

Exporting Interface Methods

- A module can export functions and tasks through an interface to other modules
- Interface will serve as a “channel” that calls remote read and write methods
- Exporting tasks and functions is not synthesizable
- Used in abstract models



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-17:

Exported Interface Methods Example

- In the example below, the tasks *write_task* and *read_task* are imported in to the interface so that they can be exported from the module defining them to other modules

- Note that the syntax used to define which tasks to export only uses the task name; however, *import* requires the full task prototype

```
interface if_method (input  clock);

logic [31:0] wdata, rdata;
logic [15:0] addr;
logic enable, write;
modport slave (
   input  clock,
   inout  wdata, addr, enable, write,  rdata,
   export write_task,
          read_task
);

modport master (
   output wdata, addr, enable, write,
   input  clock, rdata,
   import task write_task(input [15:0] a, input [31:0] d),
          task read_task(input [15:0] a, output [31:0] d)
);
endinterface : if_method
```

```
module slave(if_method.slave si);
logic[31:0] mem[int];
logic locked;

task automatic si.write_task (
   input [15:0] a,
   input [31:0] d
);
//……
endtask

task automatic si.read_task (
   input  [15:0] a,
   output [31:0] d
);
//……
endtask
// ……
endmodule
```

A task or function can also be exported to an interface without using a *modport*. This is done by declaring an *extern* prototype of the task or function within the interface.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-18:

Reconfigurable Interfaces

- Interfaces can use parameter redefinition and generate statements the same as modules
- Interface models will be defined and reconfigured every time an interface is instantiated
- The Verilog '01 generate statement can be used to create reconfigurable interfaces
- Generate blocks indicates continuous statements or procedural blocks within an interface

Slide 16-19:

Apply Your Knowledge

1) How is an interface different from a module?
2) How can you access signals defined in an interface within the body of the module that uses the interface?
3) How can you access interface ports within the body of the module that uses the interface?

A

1) How is an interface different from a module?
   - An interface cannot contain the design hierarchy (primitives, instances of modules, etc.) like modules.
   - An interface can be used as a module port unlike modules.
   - Interfaces can have modports, which allow each module connected to the interface to see the interface differently.
   - Interface ports cannot be left unconnected. Module ports can.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

2) How can you access signals defined in an interface within the body of the module that uses the interface?

   Use the syntax *<interface_instance_name>.<signal_name>* to reference the interface signals. For example: *bus.mem0_wdata* where *bus* is the interface instance name used in the module (say *mController*) and *mem0_wdata* is a logic signal defined in the interface (say *mem_bus*).

3) How can you access interface ports within the body of the module that uses the interface?

   It is the same as accessing interface signals. Use the syntax *<interface_instance_name>.<port_name>*.

Slide 16-20:

Apply Your Knowledge

1) What is the purpose of including modports in an interface?

2) With regard to the syntax, is it necessary to specify the modport name at both the module where it is used and at the top-level module?

1) What is the purpose of including modports in an interface?

   Modports provide the ability to use the interface signals differently at each module interface port. Without modports, all nets in the interface are considered bidirectional (*inout* type) and all variables are considered as references (*ref*).

2) With regard to the syntax, is it necessary to specify the modport name at both the module where it is used and at the top-level module?

   No. The SystemVerilog Language Reference Manual recommends that the modport be specified either at the module level or at the top level. However, most software tools do not issue an error/warning when the modport is specified at both the module and top levels.

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 16-21:

Apply Your Knowledge

1) What are interface methods?

2) True or false: When modules export functions to tasks through the interface, the *export* keyword in the interface only requires the function or task name. However, the *import* keyword in an interface requires the full prototype of the function or task

A

1) What are interface methods?

Interface methods are essentially functions or tasks included in the interface. These functions or tasks typically define communication protocol. Other modules in the design may either import these functions or tasks from the interface or export to other modules through the interface.

2) True or false: When modules export functions to tasks through the interface, the *export* keyword in the interface only requires the function or task name. However, the *import* keyword in an interface requires the full prototype of the function or task.

True.

Slide 16-22:

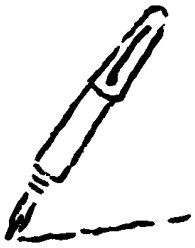
Summary

- Interface is an abstract port type that allows many signals to be grouped together and represented as a single port
- Once declared, an interface can be used as one of the ports of a module by either naming the interface port explicitly or by using a generic interface port
- Interface modports provide the ability to use the interface signals differently at each module interface port
- Functions and tasks can be used in interfaces to create a common code for a communication protocol that can be used by many modules

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Targeting Xilinx FPGAs

Slide 17-1:

2021.1

This module focuses on Xilinx-specific implementation and chip-level optimization.
20 minutes.

Slide 17-2:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-3:

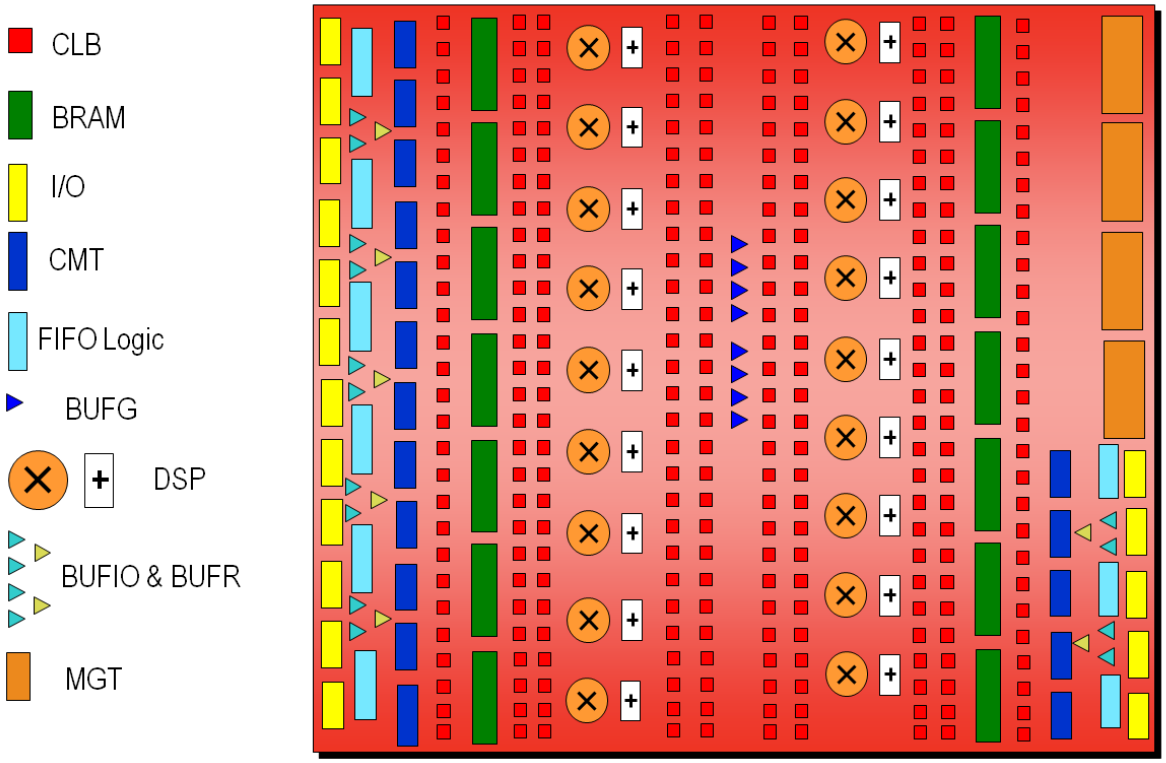
Device-Level Challenges

- 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 17-4:

7 Series FPGA Architecture



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

FPGA resources can be inferred by some synthesis tools:

- Memories
- Global clock buffers (BUFGCE, BUGFCTRL)
- Some complex DSP functions

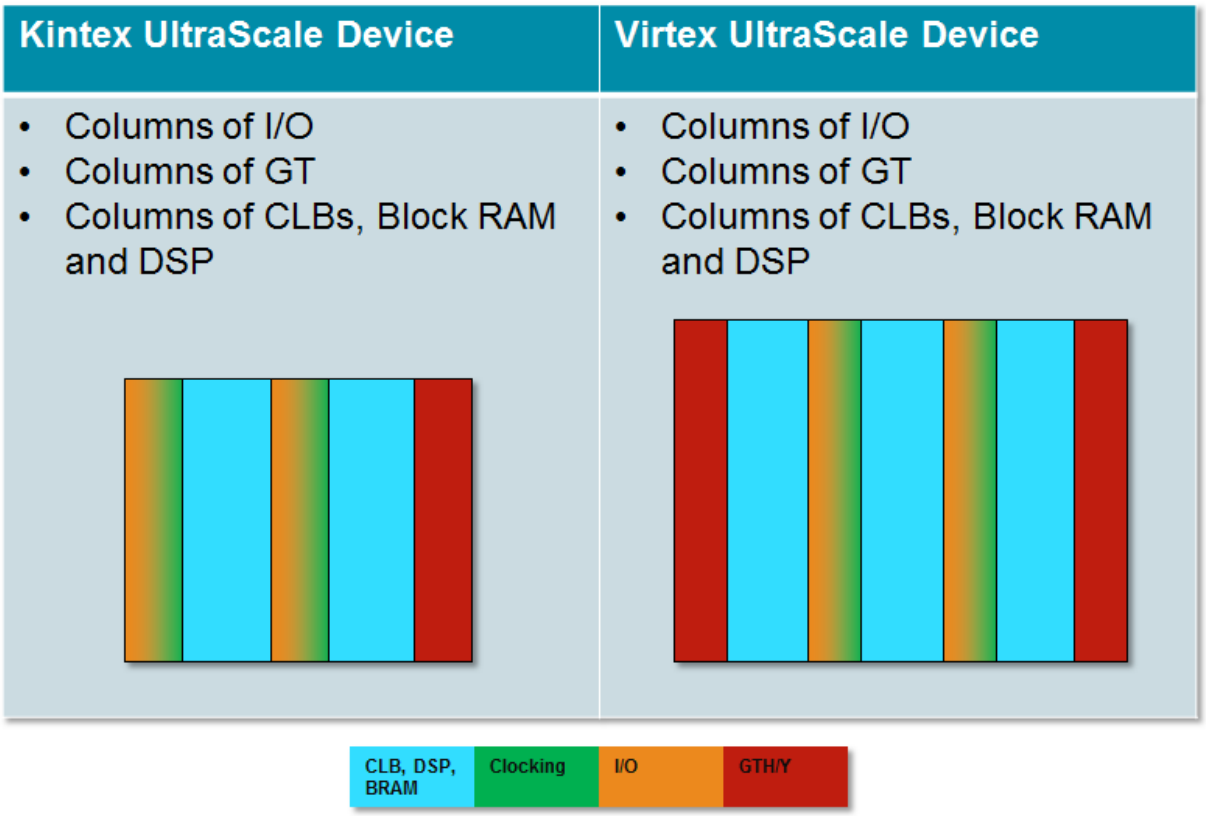
FPGA resources cannot be inferred by any synthesis tools:

- SelectIO (differential) standard
- Output DDR registers
- MMCM/PLL
- Local clock buffers (BUFIO, BUFR, BUFG_LEAF)

Slide 17-5:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

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 17-6:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-7:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-8:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-9:

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 17-10:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

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 17-11:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

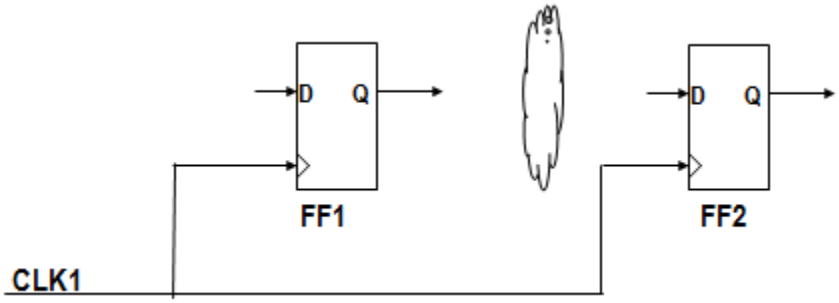
When a signal is at a metastable value, it can be interpreted as a logic 0 by some parts of the circuit and as a logic 1 by other parts. This inconsistency will never be shown during simulation and can be very hard to track down during board-level testing.

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 17-12:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-13:

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.

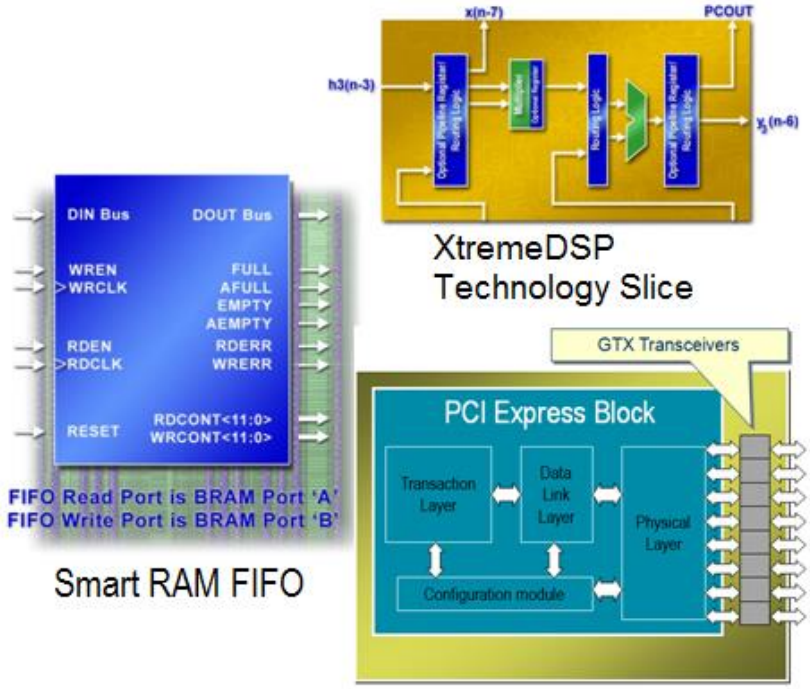
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-14:

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 17-15:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-16:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-17:

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

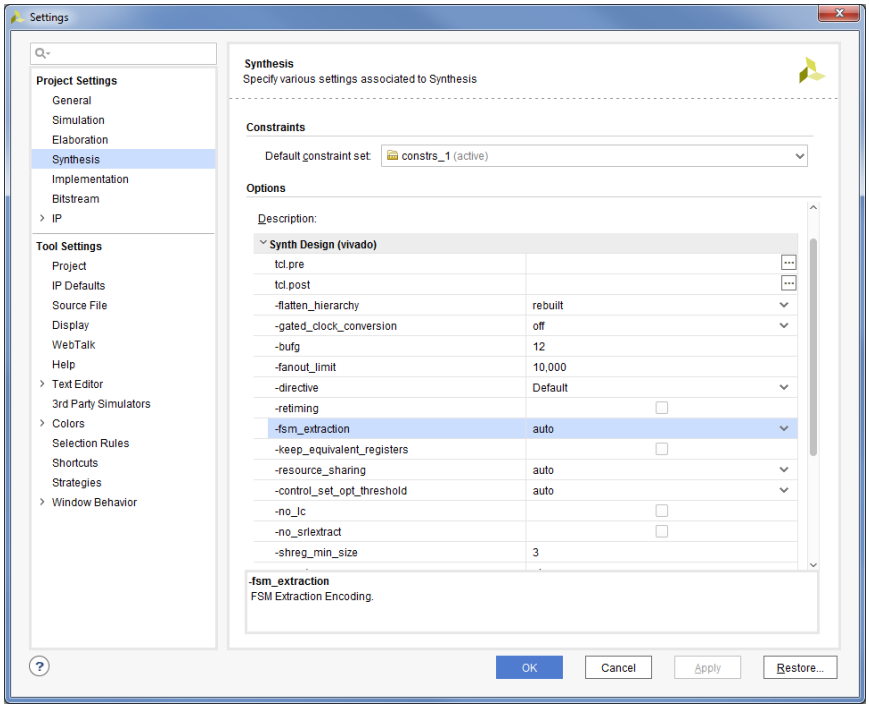
© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-18:

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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Slide 17-19:

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)
   - FSM extraction (-fsm_extraction)
   - Synthesis effort level (-directive)

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

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 17-20:

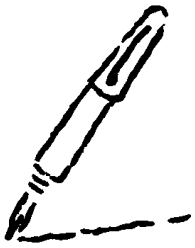
Summary

- 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 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Capture Your Notes Here



© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Appendix

Additional Xilinx Courses

Wondering which Xilinx course is right for you? Review the following persona-based curriculum paths to get started. You can start with any optional or prerequisite courses or go straight to a course that is required.

Level 1 indicates core knowledge, skills, and techniques required to become proficient with Xilinx products.

Level 2 indicates advanced knowledge, skills, and techniques required for better optimization of Xilinx products.

Level 3 indicates mastered knowledge, skills, and techniques required for best optimization and performance of Xilinx products.

Level 4 indicates expert knowledge, skills, and techniques required to innovate with Xilinx products.

For more information, go to www.xilinx.com/training.html.

Machine Learning and Data Science

Course Name	Required or Optional	Level
Developing AI Inference Solutions with the Vitis AI Platform	Required	3
Using Vision-based Applications with the Kria KV260 Vision AI Starter Kit & System-On-Module	Optional	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

System and Solution Planning

Course Name	Required or Optional	Level
Zynq SoC System Architecture	Required	3
Zynq UltraScale+ MPSoC for the System Architect	Required	3
Designing with the UltraScale and UltraScale+ Architecture	Required	3
Designing FPGAs Using the Vivado Design Suite 3	Required	3
Designing with Xilinx Serial Transceivers	Required	3
Designing with the Zynq UltraScale+ RFSoC	Optional	3
UltraFast Design Methodology	Optional	3
Designing with Dynamic Function eXchange (DFX) Using the Vivado Design Suite	Optional	4
Using Xilinx Alveo Cards to Accelerate Dynamic Workloads	Optional	3
DSP Design Using System Generator	Optional	3
Designing an Integrated PCI Express System	Optional	3
Accelerating Applications with the Vitis Unified Software Environment	Optional	3
Developing Multimedia Solutions with the Video Codec Unit Using the GStreamer Framework	Optional	3
Using Accelerated Applications with the Vision AI Starter Kit & System-on-Module (SOM)	Optional	3
Designing with the Versal ACAP: Architecture and Methodology	Required	3
Designing with the Versal ACAP: Network on Chip	Optional	3
Designing with the Versal ACAP: Power and Board Design	Optional	3
Designing with the Versal ACAP: PCI Express Systems	Optional	3
Designing with Versal AI Engine 1: Architecture and Design Flow	Required	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Course Name	Required or Optional	Level
Designing with Versal AI Engine 2: Graph Programming with AI Engine Kernels	Optional	3
Designing with Versal AI Engine 3: Kernel Programming and Optimization	Optional	4
Using Vision-based Applications with the Kria KV260 Vision AI Starter Kit & System-On-Module	Optional	3

Embedded Software Development

Course Name	Required or Optional	Level
Embedded Systems Software Design	Required	3
Zynq UltraScale+ MPSoC for the Software Developer	Required	3
Migrating to the Vitis Embedded Software Development IDE Workshop	Optional	3
Embedded Design with PetaLinux Tools	Optional	4
Designing with the Versal ACAP: Architecture and Methodology	Required	3
Designing with Versal AI Engine 1: Architecture and Design Flow	Required	3
Designing with Versal AI Engine 2: Graph Programming with AI Engine Kernels	Optional	3

Host Software Development

Course Name	Required or Optional	Level
Accelerating Applications with the Vitis Unified Software Environment	Optional	3
Developing AI Inference Solutions with the Vitis AI Platform	Required	3
Using Xilinx Alveo Cards to Accelerate Dynamic Workloads	Optional	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

AI Engine Development

Course Name	Required or Optional	Level
Designing with Versal AI Engine 1: Architecture and Design Flow	Required	3
Designing with Versal AI Engine 2: Graph Programming with AI Engine Kernels	Optional	3
Designing with Versal AI Engine 3: Kernel Programming and Optimization	Optional	4

Hardware, IP, and Development Platform

Course Name	Required or Optional	Level
Designing FPGAs Using the Vivado Design Suite 1	Required	1
Designing FPGAs Using the Vivado Design Suite 2	Required	2
Designing FPGAs Using the Vivado Design Suite 3	Required	3
Designing FPGAs Using the Vivado Design Suite 4	Optional	4
Designing with the UltraScale and UltraScale+ Architectures	Optional	3
Designing with Dynamic Function eXchange (DFX) Using the Vivado Design Suite	Optional	4
Designing with the IP Integrator Tool	Optional	2
UltraFast Design Methodology	Optional	3
Designing with Verilog	Required	1-2
Designing with SystemVerilog	Optional	1-2
Verification with SystemVerilog	Optional	1-2
Designing with VHDL	Required	1-2
Advanced VHDL	Optional	4
High-Level Synthesis with the Vitis HLS Tool	Optional	3
Designing with the Versal ACAP: Architecture and Methodology	Required	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Course Name	Required or Optional	Level
Designing with the Versal ACAP: Network on Chip	Optional	3
Designing with Versal AI Engine 1: Architecture and Design Flow	Required	3
Designing with Versal AI Engine 2: Graph Programming with AI Engine Kernels	Optional	3
Designing with Versal AI Engine 3: Kernel Programming and Optimization	Optional	4

System Integration and Validation

Course Name	Required or Optional	Level
Embedded Systems Design	Required	3
DSP Design Using System Generator	Required	3
High-Level Synthesis with the Vitis HLS Tool	Optional	3
Zynq UltraScale+ MPSoC for the Hardware Designer	Optional	3
Zynq UltraScale+ MPSoC for the System Architect	Optional	3
Zynq SoC System Architecture	Optional	3
Designing FPGAs Using the Vivado Design Suite 4	Optional	4
Designing with the IP Integrator Tool	Optional	2
Verification with SystemVerilog	Optional	1-2
Designing with the Versal ACAP: Network on Chip	Optional	3
Designing with Versal AI Engine 1: Architecture and Design Flow	Required	3
Designing with Versal AI Engine 2: Graph Programming with AI Engine Kernels	Optional	3
Using Vision-based Applications with the Kria KV260 Vision AI Starter Kit & System-On-Module	Optional	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE

Board System Design

Course Name	Required or Optional	Level
Designing with Xilinx Serial Transceivers	Required	3
Designing with the Zynq UltraScale+ RFSoC	Optional	3
Designing an Integrated PCI Express System	Optional	3
Designing with the Versal ACAP: Power and Board Design	Optional	3
Designing with the Versal ACAP: PCI Express Systems	Optional	3

© Copyright 2021 Xilinx

FOR PERSONAL USE ONLY – DO NOT DISTRIBUTE